Day 9

Part 1

Another push of the button leaves you in the familiar hallways of some friendly amphipods! Good thing you each somehow got your own personal mini submarine. The Historians jet away in search of the Chief, mostly by driving directly into walls.

While The Historians quickly figure out how to pilot these things, you notice an amphipod in the corner struggling with his computer. He's trying to make more contiguous free space by compacting all of the files, but his program isn't working; you offer to help.

He shows you the disk map (your puzzle input) he's already generated. For example:

```
2333133121414131402
```

The disk map uses a dense format to represent the layout of files and free space on the disk. The digits alternate between indicating the length of a file and the length of free space.

So, a disk map like `12345` would represent a one-block file, two blocks of free space, a three-block file, four blocks of free space, and then a five-block file. A disk map like 90909 would represent three nine-block files in a row (with no free space between them).

Each file on disk also has an ID number based on the order of the files as they appear before they are rearranged, starting with ID 0. So, the disk map `12345` has three files: a one-block file with ID 0, a three-block file with ID 1, and a five-block file with ID 2. Using one character for each block where digits are the file ID and . is free space, the disk map `12345` represents these individual blocks:

```
0..111....22222
```

The first example above, `2333133121414131402`, represents these individual blocks:

```
00...111...2...333.44.5555.6666.777.888899
```

The amphipod would like to move file blocks one at a time from the end of the disk to the leftmost free space block (until there are no gaps remaining between file blocks). For the disk map `12345`, the process looks like this:

```
0..111....22222
02.111....2222.
022111....222..
0221112...22...
02211122..2....
022111222......
```

The first example requires a few more steps:

```
00...111...2...333.44.5555.6666.777.888899
009..111...2...333.44.5555.6666.777.88889.
0099.111...2...333.44.5555.6666.777.8888..
00998111...2...333.44.5555.6666.777.888...
```

```
009981118..2...333.44.5555.6666.777.88....
0099811188.2...333.44.5555.6666.777.8.....
009981118882...333.44.5555.6666.777.......
0099811188827..333.44.5555.6666.77........
00998111888277.333.44.5555.6666.7.........
009981118882777333.44.5555.6666...........
0099811188827773336446555566.............
```

The final step of this file-compacting process is to update the filesystem checksum. To calculate the checksum, add up the result of multiplying each of these blocks' position with the file ID number it contains. The leftmost block is in position 0. If a block contains free space, skip it instead.

Continuing the first example, the first few blocks' position multiplied by its file ID number are `0 * 0 = 0`, `1 * 0 = 0`, `2 * 9 = 18`, `3 * 9 = 27`, `4 * 8 = 32`, and so on. In this example, the checksum is the sum of these, 1928.

Compact the amphipod's hard drive using the process he requested. What is the resulting filesystem checksum? (Be careful copy/pasting the input for this puzzle; it is a single, very long line.)

In [1]:
```
from pathlib import Path

import numpy as np
```

In [2]:
```
test_path = Path("data/day09_test.txt")
data_path = Path("data/day09_data.txt")

def load_data(fpath: Path) -> list[int|None]:
    """Return integer representation of the filesystem."""
    data = fpath.open().read().strip()

    fsm = []  # holds returned filesystem
    idx = 0   # file ID
    block = True  # are we in a file block or empty space
    for val in list(data):
        if block:  # in file block
            fsm += [idx] * int(val)
            idx += 1
        else:      # in empty space
            fsm += [None] * int(val)
        block = not block

    return fsm


def sort_filesystem(fsm: list[int|None]) -> list[int|None]:
    """Returns the sorted, fragmented filesystem"""
    sorted_fs = []  # Hold sorted, fragmented filesystem

    # Iterate over the filesystem LTR
    while len(fsm):
        val = fsm.pop(0)
        if val is not None:  # If popped value not a gap...
            sorted_fs.append(val)  # ...add to sorted FS
        elif len(fsm):  # If popped value is a gap...
            move_val = fsm.pop()  # ...add rightmost non-gap to sorted FS
            while move_val is None and len(fsm):
                move_val = fsm.pop()
            if move_val is not None:
                sorted_fs.append(move_val)
```

```
    return sorted_fs


def calculate_checksum(filesystem: list[int|None]) -> int:
    """Return filesystem checksum."""
    return sum([idx * int(val) for idx, val in enumerate(filesystem)])


filesystem = load_data(test_path)
print(filesystem)
sorted_fs = sort_filesystem(filesystem)
calculate_checksum(sorted_fs)
```

```
[0, 0, None, None, None, 1, 1, 1, None, None, None, 2, None, None, None, 3, 3, 3, Non
e, 4, 4, None, 5, 5, 5, 5, None, 6, 6, 6, 6, None, 7, 7, 7, None, 8, 8, 8, 8, 9, 9]
```

Out[2]: 1928

In [3]:
```
filesystem = load_data(data_path)
sorted_fs = sort_filesystem(filesystem)
calculate_checksum(sorted_fs)
```

Out[3]: 6519155389266

Part 2

Upon completion, two things immediately become clear. First, the disk definitely has a lot more contiguous free space, just like the amphipod hoped. Second, the computer is running much more slowly! Maybe introducing all of that file system fragmentation was a bad idea?

The eager amphipod already has a new plan: rather than move individual blocks, he'd like to try compacting the files on his disk by moving whole files instead.

This time, attempt to move whole files to the leftmost span of free space blocks that could fit the file. Attempt to move each file exactly once in order of decreasing file ID number starting with the file with the highest file ID number. If there is no span of free space to the left of a file that is large enough to fit the file, the file does not move.

The first example from above now proceeds differently:

```
00...111...2...333.44.5555.6666.777.888899
0099.111...2...333.44.5555.6666.777.8888..
0099.1117772...333.44.5555.6666.....8888..
0099.111777244.333....5555.6666.....8888..
00992111777.44.333....5555.6666.....8888..
```

The process of updating the filesystem checksum is the same; now, this example's checksum would be 2858.

Start over, now compacting the amphipod's hard drive using this new method instead. What is the resulting filesystem checksum?

In [4]:
```
def load_data(fpath: Path) -> list[tuple[int|None, int]]:
    """Returns filesystem representation."""
    data = fpath.open().read().strip()

    fsm = []  # holds returned filesystem
    idx = 0   # file ID
    block = True  # are we in a file block or empty space
    for val in list(data):  # Iterate over input data
```

```
        if block:  # in file block
            fsm.append((idx, int(val)))  # Add file representation
            idx += 1
        else:      # in empty space
            fsm.append((None, int(val)))  # Add space representation
        block = not block

    return fsm


def sort_filesystem(fsm: list[tuple[int|None, int]]) -> list[tuple[int|None, int]]:
    # Mark each file/gap as not yet having had a move attempt
    fsm = [[fileid, val, False] for (fileid, val) in fsm]

    # Sort the filesystem
    while len([_ for _ in fsm if _[0] and _[2] is False]):

        # Find rightmost file to work with (can't yet have tried a move)
        ridx = len(fsm) - 1
        while fsm[ridx][0] is None or fsm[ridx][2] is True:  # Iterate RTL
            ridx -= 1
        # Working with rightmost file
        fsm[ridx][2] = True
        move_file = fsm[ridx]  # Work with this file

        # Find leftmost gap to fit the file (if there is one)
        test_gap = None
        for lidx in range(ridx):  # Iterate LTR
            # Keep the index/space if it's longer than the file being considered
            if fsm[lidx][0] is None and fsm[lidx][1] >= move_file[1]:
                test_gap = fsm[lidx]
                break

        # If there's a suitable gap, move the file
        if test_gap is not None:
            if test_gap[1] == move_file[1]:  # file fits exactly
                # Swap gap and file
                fsm = fsm[:lidx] + [move_file] + fsm[lidx+1:ridx] + [test_gap] + fsm[
            elif test_gap[1] > move_file[1]:  # file fits within gap
                fsm = fsm[:lidx] + [move_file, [None, test_gap[1] - move_file[1], Fal
                      fsm[lidx+1:ridx] + [[None, move_file[1], False]] + fsm[ridx+1:]
            # Merge adjacent gaps after move, proceeding LTR
            newfsm = [fsm.pop(0)]
            for pos in fsm:
                if newfsm[-1][0] is None and pos[0] is None:
                    newfsm[-1][1] += pos[1]
                else:
                    newfsm.append(pos)
            fsm = newfsm

    # Expand filesystem and return
    sorted_fs = []
    for fileid, val, seen in fsm:
        sorted_fs += [fileid] * val
    return sorted_fs


def calculate_checksum(filesystem) -> int:
    """Returns filesystem checksum"""
    return sum([idx * int(val) for idx, val in enumerate(filesystem) if val is not No


filesystem = load_data(test_path)
sorted_fs = sort_filesystem(filesystem)
calculate_checksum(sorted_fs)
```

Out[4]: 2858

In [5]:
```
filesystem = load_data(data_path)
sorted_fs = sort_filesystem(filesystem)
calculate_checksum(sorted_fs)
```

Out[5]: 6547228115826

In [ ]: