Performance Engineering: C++ Extensions for Python

1 Introduction
- 1.1 Learning Goals
- 1.2 Strategy
- 1.3 Disclaimer

2 Tools
- 2.1 Watch
- 2.2 Timeit
- 2.3 Perf
- 2.4 Setuptools
- 2.5 Python/C API
- 2.6 Cython

3 Walk-through Exercise
- 3.1 Idiomatic Python
- 3.2 Early Exits
- 3.3 Switching to Sets
- 3.4 Caching Map Lookups
- 3.5 Translation to C++
- 3.6 Manual Parsing
- 3.7 Unordered Containers
- 3.8 C Array in Heap
- 3.9 C Array in Stack
- 3.10 Mutate Buffer in Place
- 3.11 No Dynamic Memory Allocation
- 3.12 Array Flattening + Binary Operations + Compiler Arguments + Others

Introduction

In February, I set out the goal to learn more about performance optimization in Python. I write Python code every day, yet rarely think about performance pitfalls—mostly because I didn't know much about them before. Sure, Python is almost like writing pseudo code, and its simplicity lets us code quickly and maintain it with ease. But it comes with its own challenges: it has significantly slower execution compared to compiled languages, which means longer computation times, higher costs, and even a bigger environmental footprint. This blog post walks you through my exploration of performance optimization. To share what I've learned, I've structured this page as a step-by-step exercise that demonstrates how to measure and progressively improve code performance.

Learning Goals

For this experiment, I have defined the following learning goals:

- Anticipate bottlenecks and performance traps in my code.
- Improve standard Python code by using hybrid C/C++ codebase.
- Become fluent with bench-marking and profiling tools to measure and compare performance.
- Experiment with new programming technologies like LLM-assisted development tools.

Strategy

To achieve these goals, my plan is to solve the first 10 days of Advent of Code 2024 using multiple approaches—idiomatic Python, optimized Python, Cython, C++ extensions for Python, and pure C++. I'll measure and compare the performance of each solution to see where the biggest improvements come from. Throughout this process, I'll use Cursor to boost efficiency and (hopefully) reduce bugs. I'll also keep a log on my learning progress, so I can refine my methods.

Disclaimer

This exercise is primarily about getting hands-on experience with performance optimization tools. I'm fully aware that it doesn't mirror real-world projects or everyday challenges we work on at Rewire. Think of it like a specific practice drill.

The big question now is how best to apply these skills to a real-world scenario. My next step is to find a larger codebase—ideally at PLG—where I can apply these new skills to reduce time and costs. If you have any ideas or know of relevant use cases, please reach out!

Tools

Watch

A command-line tool that automatically reruns commands at a set interval. I use it to keep a tight feedback loop on the changes I make to the code. It's nice to use in tandem with assert statements to immediately flag breaking changes.

Timeit

A Python module designed for benchmarking small snippets of code. The *number* parameter allows you to run a function many times, which helps to gauge performance improvements accurately by smoothing out noise from system variability.

`main.py` example:

```
import timeit

from aoc import solve
from aoc_cpp import solve_cpp

if __name__ == "__main__":
    with open("large_case.txt") as f:
        page = f.read()

    assert solve(page) == 4135
    assert solve_cpp(page) == 4135

    runs = 1000
    print(f"Python:\t\t{timeit.timeit(lambda: solve(page), number=runs):.4f}")
    print(f"C++:\t\t{timeit.timeit(lambda: solve_cpp(page), number=runs):.4f}")
```

Perf

A powerful Linux profiling tool that periodically samples the instruction pointer and unwinds the call stack to reveal which function call let to that instruction being executed on your CPU. It then presents the data as a hierarchical breakdown of all function calls with the percentage of samples that came from that call. Meaning you can basically see what percentage of the total time your CPU spend on all parts of your code.

Set the flag `-g` to use the counts of the task clock instead of raw CPU cycles for a more accurate measure of the time the CPU actually spends on the tasks (cycles are used when trying to find architectural bottlenecks).

Make sure when running perf to only run the function you want to measure (here that means: comment the rest of the code out in the main.py file). Use an otherwise idle system to decrease background noise. Carefully balance the number of samples. Too few samples can lead to imprecise data, while too many samples can introduce extra overhead (for me around a 1000 samples seemed to work nicely). If you're using a VM to run Linux (like I do), check if the necessary Performance Monitoring Counters (PMCs) are exposed by the hypervisor by running `perf list`.

- To start recording: `perf record -g -F 1000 -o /tmp/perf.data python3 main.py --stdio`
- To show report: `perf report --stdio -i /tmp/perf.data`
- For line profiling: `perf annotate -i /tmp/perf.data --stdio`

Setuptools

A packaging and distribution library for Python that streamlines the build process, especially for projects with C/C++ extensions, by handling many platform-specific details like linking and compiling, reducing manual configuration for different operating systems.

`setup.py` example:

```
from setuptools import setup, Extension
setup(ext_modules = [Extension("aoc_cpp", ["aoc.cpp"])])
```

Python/C API

Writing a C++ extension for Python means writing C++ code that is callable from Python as if it were a native Python module. This approach can be used to improve performance by replacing CPU-intensive parts of your Python code with optimized C++ code.

> A notable example of a company who uses this approach is **Instagram**. They started with a pure Python codebase and then incrementally migrated performance-critical sections to C++.

However, to make your C++ code work as a Python module, you need more than just the core function you want to expose. Python uses its own object model where every value is a Python object, so raw C/C++ types aren't directly compatible. This is where the Python C API comes into play.

The Python C API is a set of functions, macros, and structures that allow you to interact directly with the Python interpreter from C or C++ code. It provides the necessary tools to create, manipulate, and convert Python objects, as well as to define new modules and functions in a way that the Python runtime understands. In essence, it bridges the gap between your C++ code and the Python world, allowing you to write functions in C++ that Python can import and use just like native Python functions.

So, apart from writing the core function you want to expose, you must also write some additional code so that Python can recognize and interact with your C++ extension. This includes a method table, module definition and module initialization function (all neatly explained in the title link). To simplify this process, I've created a minimal template that handles all the boilerplate, allowing you to focus solely on your core function:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>

static PyObject* f(PyObject *self, PyObject *args) {
    // CORE FUNCTION
}

static PyMethodDef Methods[] = {
    {/*TODO name function*/, f, METH_VARARGS, "Function" /*description*/},
    {NULL, NULL, 0, NULL}
};

static struct PyModuleDef Module = {
    PyModuleDef_HEAD_INIT,
    /*TODO name extension*/,
    NULL, // documentation
    -1,
    Methods
};

PyMODINIT_FUNC PyInit_/*TODO name extension*/(void) {
    return PyModule_Create(&Module);
}
```

Cython

Cython is a programming language that lets you write code in a Python-like syntax while automatically compiling it into C, boosting performance without the hassle of manually creating a C extension. It was designed to combine the speed of C with the ease and flexibility of Python.

During the Advent of Code exercises, I used Cython for the first couple of days as a way to gain performance improvements. However, I soon found that pure C++ extensions consistently outperformed Cython, and writing C++ code offered more flexibility for fine-tuning performance. More importantly, while Cython's main advantage is that you don't have to write C++ manually, Claude's "talent" for programming-language-to-programming-language translation (in my experience, far superior to translating english-pseudocode-to-programming language) makes it super straightforward to create an efficient C++ extension directly—essentially alleviating the need for a tool like Cython altogether. If you're interested in learning more about Cython anyway, check out the tutorial in the title link.

Walk-through Exercise

In order to demonstrate how to improve performance using an example exercise, you need to (unfortunately) understand the example exercise. I've chosen a very simple exercise from the Advent of Code, so that this won't take up too much of your time.

You are given a "page" in a text file that contains two parts: a list of number ordering "rules" and a list of "updates". Each update is simply a list of numbers of variable length. Each ordering rule is in the form of `X|Y`, indicating that if an update contains both numbers `X` and `Y`, then `X` must come before `Y` (not necessarily immediately before). The goal is to check for each update if they are in a valid order according to the rules. The function needs to return the sum of the middle numbers of all valid updates.

Here is an example input with a couple rules and one update:

```
47|53
97|13
97|61
97|47
75|29
61|13
75|53
29|13
97|29
53|29
61|53
97|53
61|29
47|13
75|47
97|75
47|61
75|61
47|29
75|13
53|13

75,47,61,53,29
```

Let's see if the update is valid:

- `75` is correctly first because there are rules that put each other number in the update after it: `75|47`, `75|61`, `75|53`, and `75|29`.
- `47` is correctly second because 75 must be before it (`75|47`) and every other page must be after it according to `47|61`, `47|53`, and `47|29`.
- `61` is correctly in the middle because 75 and 47 are before it (`75|61` and `47|61`) and 53 and 29 are after it (`61|53` and `61|29`).
- `53` is correctly fourth because it is before page number 29 (`53|29`).
- `29` is the only page left and so is correctly last.

This update is thus valid. Because the update does not include some numbers, the ordering rules involving those missing numbers are ignored.

If you want to try to solve the exercise for yourself, here are small and large text file inputs: small_case.txt large_case.txt.

Here is my psuedo-code for solving the exercise:

```
Initialize total sum to zero
Initialize empty dictionary (key: int, value: list[int]) to track
  for each number what numbers should come after it

Loop through each line in the input file:
  If the line is a rule (X|Y)
    Parse it
    Add Y to the dict value list of key X
  If the line is an update:
    Parse it
    Loop through each number in the update (A):
      Loop through each number that comes after A in the update (B):
        Check if B is in the value list of key A (meaning it should come after it):
          If B is not found, the update is invalid

    If it is a valid update, add the middle number to the total
  Return the total
```

Now let's solve the exercise in code step-by-step, where every time we measure the solution and try to improve its performance.

Idiomatic Python

On the left, you'll notice my initial "naive" Python solution—a direct translation of the pseudocode into Python. While it's tempting to immediately start optimizing by eyeballing obvious improvements, let's start by gathering some measurements. By profiling first, we can pinpoint exactly where the code spends most of its time. I found that rushing into optimizing without profiling often leads to suboptimal fixes, making it harder to achieve the best performance gains.

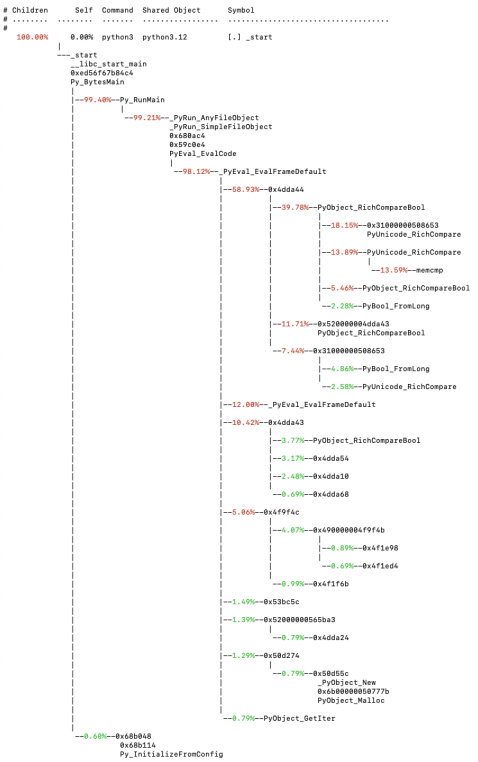
This first solution completed 1500 runs in 5.43 seconds.

On the right, the perf report output reveals that 39.78% of the samples are coming from PyObject_RichCompareBool calls. In other words, a significant portion of our execution time is consumed by Python object comparisons. This indicates that our method—specifically, comparing `update[j]` against the list of expected successors for `update[i]` in a nested loop—is inefficient.

The most straightforward idea to improve on this is to simply reduce the number of comparisons. Once we determine that an update is invalid, there's no reason to continue checking the remaining pairs. Breaking out of the nested loop as soon as a violation is found avoids unnecessary work.

```python
def solve(page):
    total = 0
    rules = {}
    parsing_rules = True # the start of the page contains the rules
    for line in page.splitlines():

        if line == "":
            parsing_rules = False
            continue # skip the whiteline

        if parsing_rules:
            x, y = line.split("|")
            rules[x] = rules.get(x, []) + [y]

        else:
            update = line.split(",")
            valid_update = True
            for i in range(len(update)):
                for j in range(i + 1, len(update)):
                    if update[j] not in rules.get(update[i]):
                        valid_update = False

            if valid_update:
                total += int(update[len(update) // 2])
    return total
```

1500- 5.43



Early Exits

On the left, you can see how I changed my code to use early exits.

This second solution completed 1500 runs in 2.54 seconds.

Even with early exits in place, a significant portion (34.34%) of samples still come from the comparisons. Each time we check if a number is “in” our list of expected successors, Python has to do a linear search through the entire list to verify membership. Additionally, our profiler points to frequent calls to the function memcmp (12.69%), highlighting that a large part of these comparisons were done at the string level, byte by byte.

While you might therefore consider converting these strings to integers to speed up the direct comparison, doing so would not eliminate the need to iterate through every element in the list—it would merely speed up each individual comparison. Instead, a more effective approach is to use sets. With sets, membership testing is handled through hash comparisons, allowing lookups to be performed in O(1) average-case time. This change not only avoids the costly linear scanning of the list but also eliminates the overhead of repeated string comparisons, leading to a much more efficient implementation.

```python
def solve(page):
    total = 0
    rules = {}
    parsing_rules = True # the start of the page contains the rules
    for line in page.splitlines():

        if line == "":
            parsing_rules = False
            continue # skip the whiteline

        if parsing_rules:
            x, y = line.split("|")
            rules[x] = rules.get(x, []) + [y]

        else:
            update = line.split(",")
            valid_update = True
            for i in range(len(update)):
                for j in range(i + 1, len(update)):
                    if update[j] not in rules.get(update[i]):
                        valid_update = False

            if valid_update:
                total += int(update[len(update) // 2])
                    break  # break inner loop when invalid pair found
                else:
                    continue  # if inner loop completed without break, continue outer loop
                break  # if inner loop broke, break outer loop too
            else:
                # if outer loop completed without break, update is valid
                total += int(update[len(update) // 2])
    return total
```

1500 - 2.54

Switching to Sets

With the switch to sets, the code ran 1500 times in 1.06 and we no longer see the explicit comparison calls that were previously draining performance. Instead, the profiler now displays larger sample percentages as raw memory addresses. These addresses are actually instruction pointers where the CPU is executing low-level machine code—often due to unresolved symbols. This indicates that the remaining overhead is primarily from inherent, compiled operations rather than inefficiencies in our Python logic. By the way, the appearance of `_pyeval_eval_frame_default` reflects the interpreter's core evaluation loop, which represents a baseline overhead that is intrinsic to Python.

The most notable function call we now see is `PyDict_Contains`, accounting for 3.56% of the samples. This suggests that the repeated dictionary key lookups—specifically when retrieving the set of values for `update[i]` in our nested loop—are consuming a significant portion of execution time. By caching these lookup results, we can avoid redundant dictionary accesses.

We are also eliminating unnecessary list concatenations—a welcome improvement. Remember that common Python operations such as list comprehensions, concatenation, and slicing always generate new lists. This behavior not only increases processing time due to the extra allocations but also raises memory usage, so it's important to use these constructs judiciously.

```python
def solve(page):
    total = 0
    rules = {}
    parsing_rules = True # the start of the page contains the rules
    for line in page.splitlines():

        if line == "":
            parsing_rules = False
            continue # skip the whiteline

        if parsing_rules:
            x, y = line.split("|")
            rules[x] = rules.get(x, []) + [y]

            if x not in rules:
                rules[x] = set()
            rules[x].add(y)
        else:
            update = line.split(",")
            for i in range(len(update)):
                for j in range(i + 1, len(update)):
                    if update[j] not in rules.get(update[i]):
                        break  # break inner loop when invalid pair found
                else:
                    continue  # if inner loop completed without break, continue outer loop
                break  # if inner loop broke, break outer loop too
            else:
                # if outer loop completed without break, update is valid
                total += int(update[len(update) // 2])
    return total
```

1500 - 1.06

Caching Map Lookups

After caching the map lookups, the code ran 1500 times in 0.7 seconds. I also experimented with some other common Python-level optimizations. While these tweaks didn't yield further improvements in this exercise, they remain useful reminders:

- Converting recursive functions to iterative ones can lower call overhead and help avoid hitting the maximum stack depth.
- Using `"".join()` for string concatenation prevents the creation of unnecessary intermediate strings.
- Caching frequently accessed global variables in local variables speeds up retrieval.
- Replacing manual index management with `enumerate` can offer modest performance gains.
- Using generator expressions—when you only need to process items one at a time—reduces memory overhead compared to building full lists.
- Leaning on built-in functions taps into their efficient C implementations.

However, Python can only get you so far. To reach maximum efficiency, we now need to move to a hybrid codebase that leverages C/C++ extensions alongside Python.

Let's get to the juicy part!

```python
def solve(page):
    total = 0
    rules = {}
    parsing_rules = True # the start of the page contains the rules
    for line in page.splitlines():

        if line == "":
            parsing_rules = False
            continue # skip the whiteline

        if parsing_rules:
            x, y = line.split("|")
            if x not in rules:
                rules[x] = set()
            rules[x].add(y)
        else:
            update = line.split(",")
            for i in range(len(update)):
                rule_i = rules.get(update[i])
                for j in range(i + 1, len(update)):
                    if update[j] not in rules.get(update[i]):
                    if update[j] not in rule_i:
                        break  # break inner loop when invalid pair found
                else:
                    continue  # if inner loop completed without break, continue outer loop
                break  # if inner loop broke, break outer loop too
            else:
                # if outer loop completed without break, update is valid
                total += int(update[len(update) // 2])
    return total
```

1500 - 0.70

Translation to C++

In this step, I've manually translated our optimized Python code to C++. This approach makes it easier to track and understand the upcoming C++ optimizations. Interestingly, this direct translation actually runs slower than our Python solution, with the code executing 1500 times in 1.11 seconds.

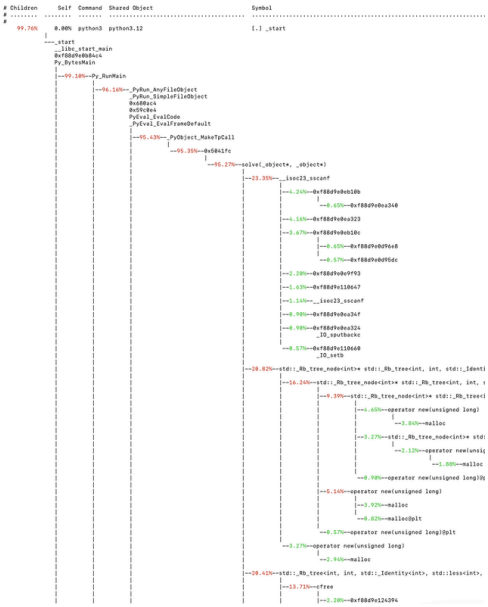
It's worth noting that in other exercises, I would use Cursor to automatically translate the code to C++ and achieve significant performance gains immediately. So please don't mistakenly conclude that C++ is inherently slower than Python!

Throughout the remainder of this exercise, I'll demonstrate how we can dramatically improve performance over the Python solution by enhancing this initial C++ implementation. This particular exercise does not even showcase the most dramatic optimization gains I got in all exercises.

Analyzing the performance profile with perf reveals that most samples are coming from the sscanf function. For our next optimization step, let's replace this with custom line parsing logic.

```cpp
static PyObject* solve(PyObject *self, PyObject *args) {
    PyObject* page;
    PyArg_ParseTuple(args, "O", &page);

    PyObject* lines = PyUnicode_Splitlines(page, 0);
    Py_ssize_t n_lines = PyList_Size(lines);

    std::map<int, std::set<int> > rules;

    int total = 0;
    bool parsing_rules = true;
    for (Py_ssize_t n = 0; n < n_lines; n++) {
        PyObject *line_obj = PyList_GetItem(lines, n);
        char *line = (char *) PyUnicode_AsUTF8(line_obj);

        if (strcmp(line, "") == 0) {
            parsing_rules = false;
            continue;
        }

        if (parsing_rules) {
            int x = 0, y = 0;
            sscanf(line, "%d|%d", &x, &y);
            rules[x].insert(y);
        } else {
            std::vector<int> update;
            int pos = 0; // number of chars processed by sscanf
            int num = 0; // next number extracted by sscanf
            while(sscanf(line, "%d%n", &num, &pos) == 1) {
                line += pos; // move char ptr past extracted number
                update.push_back(num);
                if (*line == ',') line++;
            }

            bool valid_update = true;
            for (int i = 0; i < update.size() && valid_update; i++) {
                const std::set<int> rule_i = rules[update[i]];
                for (int j = i + 1; j < update.size(); j++) {
                    if (rule_i.find(update[j]) == rule_i.end()) {
                        valid_update = false;
                        break;
                    }
                }
                if (!valid_update) break;
            }
            if (valid_update) total += update[((int) update.size()) / 2];
        }
    }
    return PyLong_FromLong(total);
}
```

1500 - 1.11



Manual Parsing

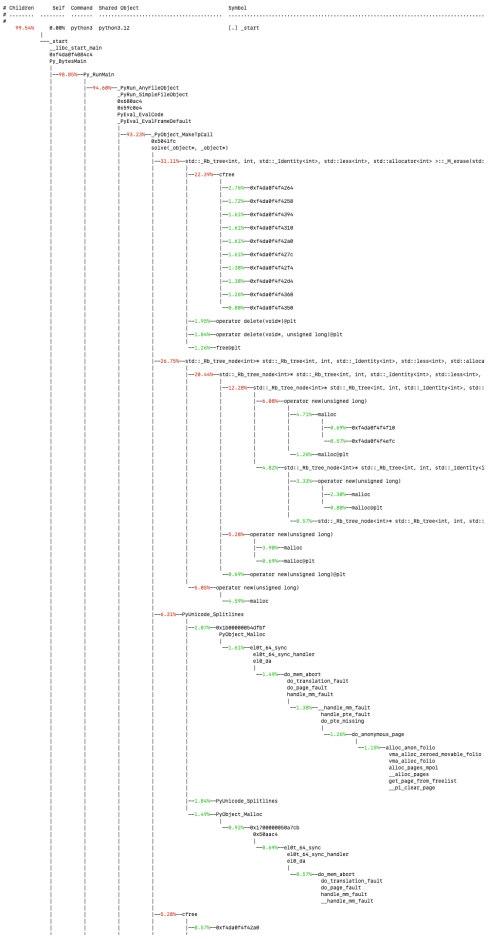
By replacing this standard library parsing function with custom manual parsing logic, we achieved a notable improvement in execution time - down to 0.8 seconds for 1500 runs.

However, our perf output now reveals a new dominant source of overhead: numerous samples related to "Rb_tree" operations. These are coming from the C++ standard library's implementation of ordered containers (like std::map and std::set). Under the hood, these containers maintain their elements in a sorted order using a balanced binary tree data structure known as a red-black tree. While this provides useful ordering guarantees for range queries, nearest-neighbor searches, and applications requiring sorted traversal, the tree-balancing operations introduce significant overhead for simple lookup operations. Lookup takes O(log n), since you need to traverse the tree from root to leaf to find each element.

Since our algorithm only performs direct lookups to check if elements exist or retrieve specific values, this presents a clear optimization opportunity. In the next step, I'll replace these ordered containers with their unordered counterparts (std::unordered_map and std::unordered_set), which use hash tables internally. These hash-based containers typically offer O(1) average-case lookups by computing the element's position directly from its key, which should significantly reduce our computational overhead.

```cpp
static PyObject* solve(PyObject *self, PyObject *args) {
    PyObject* page;
    PyArg_ParseTuple(args, "O", &page);

    PyObject* lines = PyUnicode_Splitlines(page, 0);
    Py_ssize_t n_lines = PyList_Size(lines);

    std::map<int, std::set<int> > rules;

    int total = 0;
    bool parsing_rules = true;
    for (Py_ssize_t n = 0; n < n_lines; n++) {
        PyObject *line_obj = PyList_GetItem(lines, n);
        char *line = (char *) PyUnicode_AsUTF8(line_obj);

        if (strcmp(line, "") == 0) {
            parsing_rules = false;
            continue;
        }

        if (parsing_rules) {
            int x = 0, y = 0;
            sscanf(line, "%d|%d", &x, &y);

            char* ptr = line;
            while (*ptr >= '0' && *ptr <= '9') {
                x = x * 10 + (*ptr - '0');
                ptr++;
            }
            ptr++;

            while (*ptr >= '0' && *ptr <= '9') {
                y = y * 10 + (*ptr - '0');
                ptr++;
            }

            rules[x].insert(y);
        } else {
            std::vector<int> update;

            int pos = 0; // number of chars processed by sscanf
            int num = 0; // next number extracted by sscanf
            while(sscanf(line, "%d%n", &num, &pos) == 1) {
                line += pos; // move char ptr past extracted number
            char* ptr = line;
            while (*ptr) {
                int num = 0;
                while (*ptr >= '0' && *ptr <= '9') {
                    num = num * 10 + (*ptr - '0');
                    ptr++;
                }
                update.push_back(num);
                if (*line == ',') line++;
                if (*ptr == ',') ptr++;
            }

            bool valid_update = true;
            for (int i = 0; i < update.size() && valid_update; i++) {
                const std::set<int> rule_i = rules[update[i]];
                for (int j = i + 1; j < update.size(); j++) {
                    if (rule_i.find(update[j]) == rule_i.end()) {
                        valid_update = false;
                        break;
                    }
                }
                if (!valid_update) break;
            }
            if (valid_update) total += update[((int) update.size()) / 2];
        }
    }
    return PyLong_FromLong(total);
}
```

1500 - 0.8



Unordered Containers

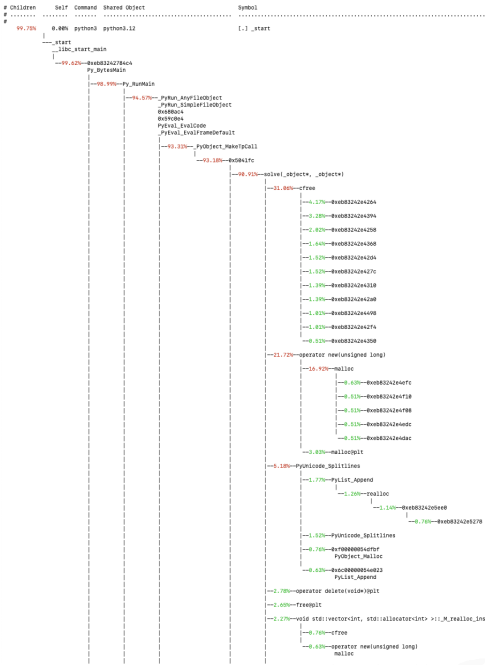
After replacing the ordered containers with their unordered counterparts, we achieved another performance boost, with execution time improving to 0.73 seconds for 1500 runs.

However, when examining the new perf profile, I noticed we're still spending significant time in memory management functions - specifically cfree, operator new, and malloc. This indicates that despite using more efficient hash-based containers, we're still paying a substantial overhead for dynamic memory allocation and deallocation. Every time we insert elements into our unordered containers, the standard library potentially needs to allocate memory for new elements, occasionally resize and rehash the entire container and later free this memory when containers are destroyed.

Now we of course do not want to write our own implementation of a hashmap. But there is an elegant optimization opportunity here. Since all the keys in our rules maps are integers, we can use a nested array instead of a map from integers to sets. Let's see how we do this in the next section.

```cpp
static PyObject* solve(PyObject *self, PyObject *args) {
    PyObject* page;
    PyArg_ParseTuple(args, "O", &page);

    PyObject* lines = PyUnicode_Splitlines(page, 0);
    Py_ssize_t n_lines = PyList_Size(lines);

    std::map<int, std::set<int> > rules;
    std::unordered_map<int, std::unordered_set<int> > rules;

    int total = 0;
    bool parsing_rules = true;
    for (Py_ssize_t n = 0; n < n_lines; n++) {
        PyObject *line_obj = PyList_GetItem(lines, n);
        char *line = (char *) PyUnicode_AsUTF8(line_obj);

        if (strcmp(line, "") == 0) {
            parsing_rules = false;
            continue;
        }

        if (parsing_rules) {
            int x = 0, y = 0;

            char* ptr = line;
            while (*ptr >= '0' && *ptr <= '9') {
                x = x * 10 + (*ptr - '0');
                ptr++;
            }
            ptr++;

            while (*ptr >= '0' && *ptr <= '9') {
                y = y * 10 + (*ptr - '0');
                ptr++;
            }

            rules[x].insert(y);
        } else {
            std::vector<int> update;

            char* ptr = line;
            while (*ptr) {
                int num = 0;
                while (*ptr >= '0' && *ptr <= '9') {
                    num = num * 10 + (*ptr - '0');
                    ptr++;
                }
                update.push_back(num);
                if (*ptr == ',') ptr++;
            }

            bool valid_update = true;
            for (int i = 0; i < update.size() && valid_update; i++) {
                const std::set<int> rule_i = rules[update[i]];
                const std::unordered_set<int> rule_i = rules[update[i]];
                for (int j = i + 1; j < update.size(); j++) {
                    if (rule_i.find(update[j]) == rule_i.end()) {
                        valid_update = false;
                        break;
                    }
                }
                if (!valid_update) break;
            }
            if (valid_update) total += update[((int) update.size()) / 2];
        }
    }
    return PyLong_FromLong(total);
}
```

1500 - 0.73



C Array in Heap

So how can we use an array of arrays instead of a map here? We know from the problem statement that:

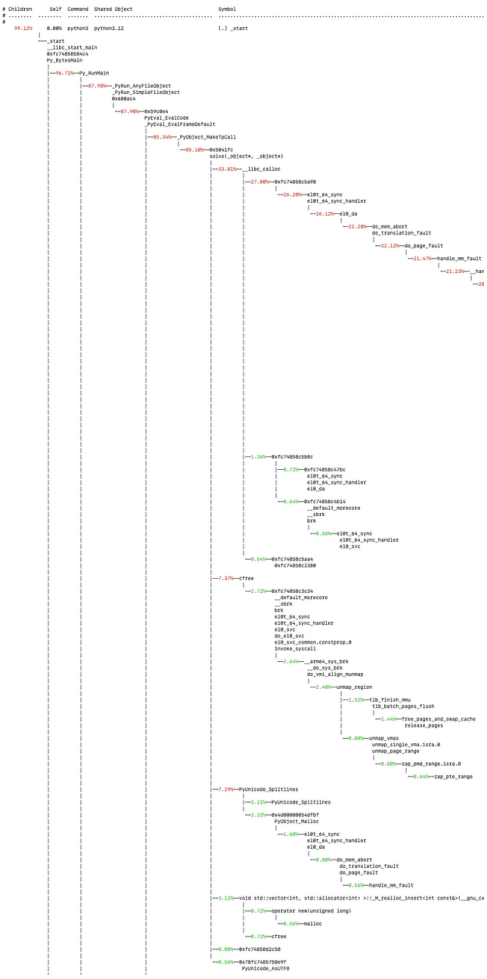
1. The total number of rules cannot exceed the number of lines in the file (`n_lines`). Each rule key is an integer, so that's our maximum number of keys.
2. Each page number is two digits (ranging from `00` to `99`), so any "rule list" (i.e., all successors to a particular page number) can have at most 100 entries.

Leveraging these constraints, we allocate a single, continuous block of memory on the heap—an outer array of size `n_lines` so there's at least one slot for each possible rule key, and each element is an inner array of 100 integers so there is space for all the possible successors. We then initialize everything to zero and fill in the rules as we parse them. Checking if "X must come before Y" now boils down to scanning the `rules[X]` array until we find `Y` (or a zero entry if it must not). This eliminates the overhead of iterators, hash computations, or tree balancing and gives us effectively O(1) lookups through simple pointer arithmetic in a contiguous block of memory. This change slashed execution time from ~0.73 seconds to 0.23 seconds for 1500 runs—direct proof that dropping container overhead in favor of a direct indexing approach can pay off.

We do, however, still see a final call to free memory at the end, showing there's more overhead to trim. Normally, this is about as far as you can push memory usage optimizations without restructuring your program entirely. But in our case, we know our rules block isn't that large. That invites the question: can we put this entire rules memory block on the stack to avoid the operating system calls for allocation and deallocation entirely?

Of course, if the array were much bigger, we'd risk hitting a segmentation fault by blowing out the stack size. You'd definitely need to test carefully in a real-world scenario. However, given our limit of `n_lines` ×100 (and knowing `n_line ~= 100`), putting the entire data structure on the stack is completely feasible. Let's see just how much overhead that removes next.

```c
static PyObject* solve(PyObject *self, PyObject *args) {
    PyObject* page;
    PyArg_ParseTuple(args, "O", &page);

    PyObject* lines = PyUnicode_Splitlines(page, 0);
    Py_ssize_t n_lines = PyList_Size(lines);

    std::unordered_map<int, std::unordered_set<int> > rules;
    const int MAX_DEPS = 100;
    int** rules = (int**) malloc(n_lines * sizeof(int*));

    // Initialize each rule's array with zeros
    for (int i = 0; i < n_lines; i++) {
        rules[i] = (int*) calloc(MAX_DEPS, sizeof(int));  // calloc initializes to 0
    }

    int total = 0;
    bool parsing_rules = true;
    for (Py_ssize_t n = 0; n < n_lines; n++) {
        PyObject *line_obj = PyList_GetItem(lines, n);
        char *line = (char *) PyUnicode_AsUTF8(line_obj);

        if (strcmp(line, "") == 0) {
            parsing_rules = false;
            continue;
        }

        if (parsing_rules) {
            int x = 0, y = 0;

            char* ptr = line;
            while (*ptr >= '0' && *ptr <= '9') {
                x = x * 10 + (*ptr - '0');
                ptr++;
            }
            ptr++;

            while (*ptr >= '0' && *ptr <= '9') {
                y = y * 10 + (*ptr - '0');
                ptr++;
            }

            rules[x].insert(y);
            // Find first empty slot (0) and insert y
            int* rule = rules[x];
            int pos = 0;
            while (rule[pos] != 0) pos++;
            rule[pos] = y;
        } else {
            std::vector<int> update;

            char* ptr = line;
            while (*ptr) {
                int num = 0;
                while (*ptr >= '0' && *ptr <= '9') {
                    num = num * 10 + (*ptr - '0');
                    ptr++;
                }
                update.push_back(num);
                if (*ptr == ',') ptr++;
            }

            bool valid_update = true;
            for (int i = 0; i < update.size() && valid_update; i++) {
                const std::unordered_set<int> rule_i = rules[update[i]];
                int* rule_i = rules[update[i]];

                for (int j = i + 1; j < update.size(); j++) {
                    if (rule_i.find(update[j]) == rule_i.end()) {
                        bool found = false;
                        // Search until we hit a 0 or find the number
                        for (int k = 0; rule_i[k] != 0; k++) {
                            if (rule_i[k] == update[j]) {
                                found = true;
                                break;
                            }
                        }
                        if (!found) {
                            valid_update = false;
                            break;
                        }
                    }
                    if (!valid_update) break;
                }
                if (valid_update) total += update[((int) update.size()) / 2];
            }
        }

    // Cleanup
    for (int i = 0; i < n_lines; i++) {
        free(rules[i]);
    }
    free(rules);

    return PyLong_FromLong(total);
}
```



1500 - 0.28

C Array in Stack

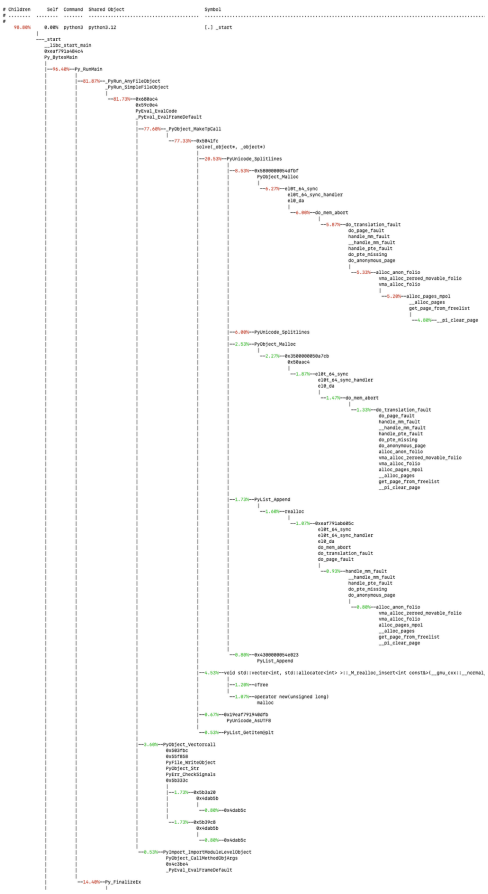
One way to remove even the small overhead of calling `malloc` and `free` is to place our entire rules data structure on the stack. The compiler knows our fixed `MAX_DEPS` size at compile time, but the actual memory is claimed on the stack at run time, each time the function is called. Crucially, no explicit `free` is required—once the function returns, the stack pointer is reset automatically, freeing that space. Because stack allocation is extremely fast—typically just incrementing or decrementing the stack pointer—it avoids many system calls inherent in heap usage.

In our case, the `n_lines × 100` block is only a few tens of kilobytes, so it fits comfortably in typical stack space. In a real-world scenario with a far bigger structure, you'd risk a segmentation fault by exceeding stack limits. Still, for our data size, moving from heap to stack further reduced runtime from ~0.23 seconds down to ~0.17 seconds for 1500 runs. A quick look at the profiler confirms fewer calls to OS-level memory-management routines, reflecting how stack-based allocation sidesteps the overhead of heap usage.

We do still see some overhead from Python calls like `splitlines` and other Python C API interactions. In general, the less we bounce between Python and C, the faster we can go. That naturally leads us to the next optimization step: parsing the input more directly in C, so we minimize Python calls (and conversions) even further.

```c
static PyObject* solve(PyObject *self, PyObject *args) {
    PyObject* page;
    PyArg_ParseTuple(args, "O", &page);

    PyObject* lines = PyUnicode_Splitlines(page, 0);
    Py_ssize_t n_lines = PyList_Size(lines);

    const int MAX_DEPS = 100;
    int** rules = (int**) malloc(n_lines * sizeof(int*));

    // Initialize each rule's array with zeros
    for (int i = 0; i < n_lines; i++) {
        rules[i] = (int*) calloc(MAX_DEPS, sizeof(int));  // calloc initializes to 0
    }
    const int MAX_DEPS = 100; // For larger depth you get a segmentation fault
    int rules[MAX_DEPS][MAX_DEPS] = {0};

    int total = 0;
    bool parsing_rules = true;
    for (Py_ssize_t n = 0; n < n_lines; n++) {
        PyObject *line_obj = PyList_GetItem(lines, n);
        char *line = (char *) PyUnicode_AsUTF8(line_obj);

        if (strcmp(line, "") == 0) {
            parsing_rules = false;
            continue;
        }

        if (parsing_rules) {
            int x = 0, y = 0;

            char* ptr = line;
            while (*ptr >= '0' && *ptr <= '9') {
                x = x * 10 + (*ptr - '0');
                ptr++;
            }
            ptr++;

            while (*ptr >= '0' && *ptr <= '9') {
                y = y * 10 + (*ptr - '0');
                ptr++;
            }

            // Find first empty slot (0) and insert y
            int* rule = rules[x];
            int pos = 0;
            while (rule[pos] != 0) pos++;
            rule[pos] = y;
        } else {
            std::vector<int> update;

            char* ptr = line;
            while (*ptr) {
                int num = 0;
                while (*ptr >= '0' && *ptr <= '9') {
                    num = num * 10 + (*ptr - '0');
                    ptr++;
                }
                update.push_back(num);
                if (*ptr == ',') ptr++;
            }

            bool valid_update = true;
            for (int i = 0; i < update.size() && valid_update; i++) {
                int* rule_i = rules[update[i]];

                for (int j = i + 1; j < update.size(); j++) {
                    bool found = false;
                    // Search until we hit a 0 or find the number
                    for (int k = 0; rule_i[k] != 0; k++) {
                        if (rule_i[k] == update[j]) {
                            found = true;
                            break;
                        }
                    }
                    if (!found) {
                        valid_update = false;
                        break;
                    }
                }
                if (!valid_update) break;
            }
            if (valid_update) total += update[((int) update.size()) / 2];
        }
    }

    // Cleanup
    for (int i = 0; i < n_lines; i++) {
        free(rules[i]);
    }
    free(rules);

    return PyLong_FromLong(total);
}
```

1500 - 0.17



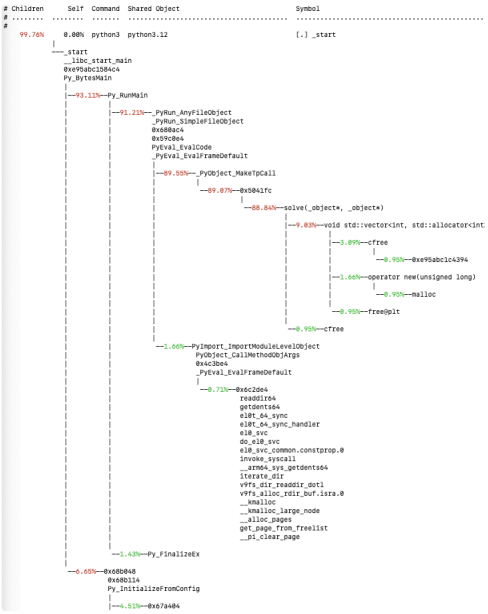
Mutate Buffer in Place

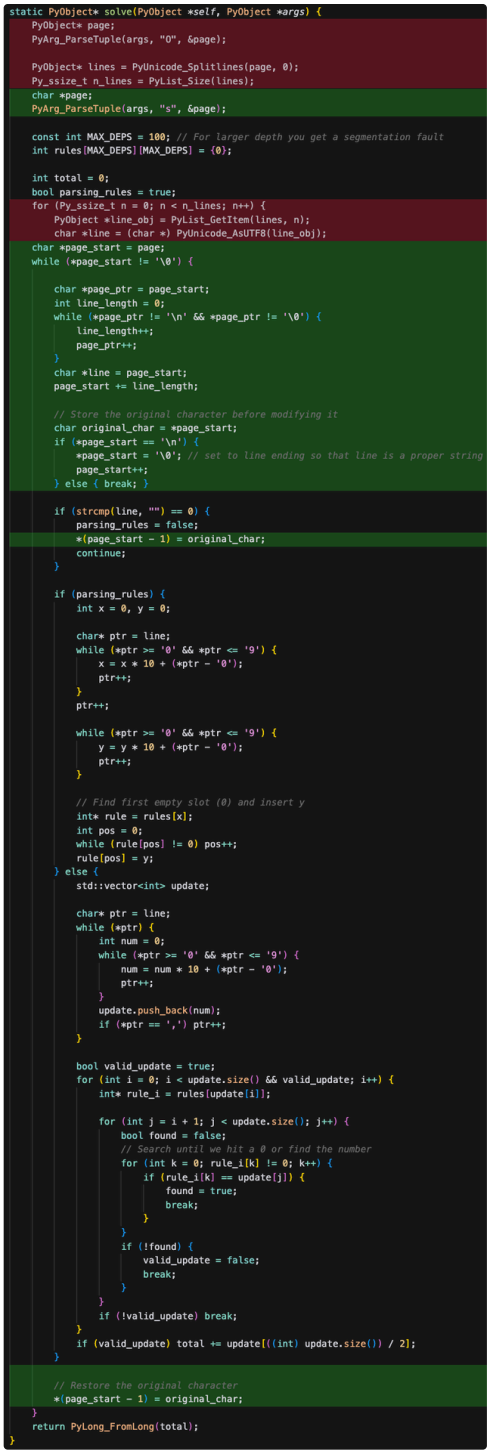
Until now, we'd been calling Python's `splitlines` to separate our input into lines, internally creating Python list objects. This section removes that overhead by manually processing the raw text buffer in C. Rather than building an extra data structure, we keep a pointer

that scans through each character of the file. When we encounter a newline (`\n`), we replace it with a null terminator (`\0`), turning that segment into a standalone C string representing the current line.

By handling the file in place, we avoid copying or allocating memory for each line—unlike `splitlines`, which creates a brand-new Python list (and corresponding Python strings). This slashes the number of Python C API calls, cutting down on overhead from both reference counting and object creation. It's also worth stressing that mutating a buffer in place must be done with caution; if you're not careful, you could corrupt or lose data. In our code, we restore the original characters at the end of the script, ensuring the program state remains consistent. It's a delicate dance, but when done correctly, this is a powerful way to squeeze out extra performance.

In practice, switching to a fully manual parse yielded another clear speedup. The runtime improved from ~0.17 seconds down to ~0.13 seconds for 1500 runs. Reviewing the `perf` output now shows that Python's string and list operations are largely out of the picture—our new bottleneck is the standard library's `std::vector`, which we're using to construct the updates before checking their validity. That will be our next focus as we refine the code further.





```
static PyObject* solve(PyObject *self, PyObject *args) {
    PyObject* page;
    PyArg_ParseTuple(args, "O", &page);

    PyObject* lines = PyUnicode_Splitlines(page, 0);
    Py_ssize_t n_lines = PyList_Size(lines);
    char *page;
    PyArg_ParseTuple(args, "s", &page);

    const int MAX_DEPS = 100; // For larger depth you get a segmentation fault
    int rules[MAX_DEPS][MAX_DEPS] = {0};

    int total = 0;
    bool parsing_rules = true;
    for (Py_ssize_t n = 0; n < n_lines; n++) {
        PyObject *line_obj = PyList_GetItem(lines, n);
        char *line = (char *) PyUnicode_AsUTF8(line_obj);
    char *page_start = page;
    while (*page_start != '\0') {

        char *page_ptr = page_start;
        int line_length = 0;
        while (*page_ptr != '\n' && *page_ptr != '\0') {
            line_length++;
            page_ptr++;
        }
        char *line = page_start;
        page_start += line_length;

        // Store the original character before modifying it
        char original_char = *page_start;
        if (*page_start == '\n') {
            *page_start = '\0'; // set to line ending so that line is a proper string
            page_start++;
        } else { break; }

        if (strcmp(line, "") == 0) {
            parsing_rules = false;
            *(page_start - 1) = original_char;
            continue;
        }

        if (parsing_rules) {
            int x = 0, y = 0;

            char* ptr = line;
            while (*ptr >= '0' && *ptr <= '9') {
                x = x * 10 + (*ptr - '0');
                ptr++;
            }
            ptr++;

            while (*ptr >= '0' && *ptr <= '9') {
                y = y * 10 + (*ptr - '0');
                ptr++;
            }

            // Find first empty slot (0) and insert y
            int* rule = rules[x];
            int pos = 0;
            while (rule[pos] != 0) pos++;
            rule[pos] = y;
        } else {
            std::vector<int> update;

            char* ptr = line;
            while (*ptr) {
                int num = 0;
                while (*ptr >= '0' && *ptr <= '9') {
                    num = num * 10 + (*ptr - '0');
                    ptr++;
                }
                update.push_back(num);
                if (*ptr == ',') ptr++;
            }

            bool valid_update = true;
            for (int i = 0; i < update.size() && valid_update; i++) {
                int* rule_i = rules[update[i]];

                for (int j = i + 1; j < update.size(); j++) {
                    bool found = false;
                    // Search until we hit a 0 or find the number
                    for (int k = 0; rule_i[k] != 0; k++) {
                        if (rule_i[k] == update[j]) {
                            found = true;
                            break;
                        }
                    }
                    if (!found) {
                        valid_update = false;
                        break;
                    }
                }
                if (!valid_update) break;
            }
            if (valid_update) total += update[((int) update.size()) / 2];
        }

        // Restore the original character
        *(page_start - 1) = original_char;
    }
    return PyLong_FromLong(total);
}
```

1500 - 0.13

No Dynamic Memory Allocation

We now replace our final remaining dynamic allocation—specifically, the `std::vector<int>` for updates—with a fixed-size stack array (of length 50, since no update list exceeds about 40 items). Although this didn't measurably speed things up further, the `perf` output now shows no calls left that we can easily optimize. In other words, we've reached the end of the line for low-hanging performance fruit: our solution is already running close to peak efficiency for this problem!

We had about a 50x speedup from our original solution!

```c
static PyObject* solve(PyObject *self, PyObject *args) {
    char *page;
    PyArg_ParseTuple(args, "s", &page);

    const int MAX_DEPS = 100; // For larger depth you get a segmentation fault
    int rules[MAX_DEPS][MAX_DEPS] = {0};

    int total = 0;
    bool parsing_rules = true;
    char *page_start = page;
    while (*page_start != '\0') {

        char *page_ptr = page_start;
        int line_length = 0;
        while (*page_ptr != '\n' && *page_ptr != '\0') {
            line_length++;
            page_ptr++;
        }
        char *line = page_start;
        page_start += line_length;

        // Store the original character before modifying it
        char original_char = *page_start;
        if (*page_start == '\n') {
            *page_start = '\0'; // set to line ending so that line is a proper string
            page_start++;
        } else { break; }

        if (strcmp(line, "") == 0) {
            parsing_rules = false;
            *(page_start - 1) = original_char;
            continue;
        }

        if (parsing_rules) {
            int x = 0, y = 0;

            char* ptr = line;
            while (*ptr >= '0' && *ptr <= '9') {
                x = x * 10 + (*ptr - '0');
                ptr++;
            }
            ptr++;

            while (*ptr >= '0' && *ptr <= '9') {
                y = y * 10 + (*ptr - '0');
                ptr++;
            }

            // Find first empty slot (0) and insert y
            int* rule = rules[x];
            int pos = 0;
            while (rule[pos] != 0) pos++;
            rule[pos] = y;
        } else {
            std::vector<int> update;

            int update[50] = {0};  // Fixed-size array initialized to zero
            int update_size = 0;   // Track number of elements
            char* ptr = line;
            while (*ptr) {
                int num = 0;
                while (*ptr >= '0' && *ptr <= '9') {
                    num = num * 10 + (*ptr - '0');
                    ptr++;
                }
                update.push_back(num);
                update[update_size++] = num;  // Add number and increment size
                if (*ptr == ',') ptr++;
            }

            bool valid_update = true;
            for (int i = 0; i < update.size() && valid_update; i++) {
            for (int i = 0; i < update_size && valid_update; i++) {
                int* rule_i = rules[update[i]];

                for (int j = i + 1; j < update.size(); j++) {
                for (int j = i + 1; j < update_size; j++) {
                    bool found = false;
                    // Search until we hit a 0 or find the number
                    for (int k = 0; rule_i[k] != 0; k++) {
                        if (rule_i[k] == update[j]) {
                            found = true;
                            break;
                        }
                    }
                    if (!found) {
                        valid_update = false;
                        break;
                    }
                }
                if (!valid_update) break;
            }
            if (valid_update) total += update[((int) update.size()) / 2];
            if (valid_update) total += update[update_size / 2];
        }

        // Restore the original character
        *(page_start - 1) = original_char;
    }
    return PyLong_FromLong(total);
}
```

1500 - 0.11

```
# Children      Self  Command  Shared Object                          Symbol
# ........  ........  .......  .....................................  ......................
#
    99.42%     0.00%  python3  python3.12                             [.] _start
           |
           ---_start
              __libc_start_main
              0xedec6f9f84c4
              Py_BytesMain
              |
              |--95.05%--Py_RunMain
              |          |
              |          |--93.16%--_PyRun_AnyFileObject
              |          |          _PyRun_SimpleFileObject
              |          |          0x680ac4
              |          |          |
              |          |           --93.01%--0x59c0e4
              |          |                     PyEval_EvalCode
              |          |                     _PyEval_EvalFrameDefault
              |          |                     |
              |          |                     |--92.14%--_PyObject_MakeTpCall
              |          |                     |          |
              |          |                     |          |--91.56%--0x5041fc
              |          |                     |          |          solve(_object*, _object*)
              |          |                     |          |
              |          |                     |           --0.58%--0x51e804
              |          |                     |                     0x5238f0
              |          |                     |                     _PyObject_Call_Prepend
              |          |                     |                     _PyEval_EvalFrameDefault
              |          |                     |
              |          |                      --0.73%--PyImport_ImportModuleLevelObject
              |          |                               PyObject_CallMethodObjArgs
              |          |                               0x4c3be4
              |          |                               _PyEval_EvalFrameDefault
              |          |
              |           --1.60%--Py_FinalizeEx
              |                     |
              |                      --0.58%--PyGC_Collect
              |                               0x5b364c
```

Array Flattening + Binary Operations + Compiler Arguments + Others

At this stage, we're already close to peak efficiency for our particular exercise. However, there are still a few additional tweaks worth mentioning—sometimes they help in other scenarios, even if they didn't show measurable gains here:

1. **Flattening Nested Arrays.** Instead of a 2D structure like `arr[row][col]`, you can collapse everything into a single dimension, for instance: `int arr[MAX_ROWS * MAX_COLS]`. Then you manually index with `arr[i + j * MAX_COLS]`. This can sometimes help the compiler optimize memory access and reduce pointer arithmetic, but whether you see a real benefit depends on how your code traverses the data.
2. **Binary Operations Instead of Arithmetic.** For operations like multiplying or dividing by 2, bitwise shifts can be slightly faster. For example, `x << 1` (left shift by 1) instead of `x * 2`. Modern compilers already optimize many simple cases, so it's not always guaranteed to speed things up, but it's worth being aware of. For more details, see this Stack Overflow thread.

3. **Compiler Flags (e.g., -O3).** Turning on `-O3` or similar optimization flags in `setup.py` gives the compiler freedom to apply aggressive optimizations, such as loop unrolling and vectorization. It can occasionally yield further speedups, though the effects vary by platform, compiler, and code structure.

Beyond these, I've also tried other optimizations in different exercises that had real impact:

- **Using Smaller C Types**
  If your values always fit in a `short` or `char`, switching from `int` can pack more data into caches and registers.
- **Bitarrays**
  Representing booleans as single bits can drastically reduce memory footprints, which is often beneficial for large datasets or tight loops.
- **Branch Prediction**
  Simple reordering of logic can help the CPU predict branches more accurately. Here's a classic discussion on data ordering and its effect on performance.
- **Reducing Type Conversions**
  Bouncing between floats, doubles, and ints can add unnecessary overhead and sometimes degrade performance if done frequently.
- **Cache Misses & Loop Unrolling**
  Cache misses are costly, so ensuring your data layout is friendly to sequential access can speed things up. Loop unrolling manually or letting the compiler do it can reduce jump instructions and streamline execution flow.

While none of these tweaks moved the needle in this specific exercise, they remain valuable techniques to keep in your performance toolbox. If your codebase or datasets grow more complex, these so-called "final mile" optimizations might make all the difference.