PYTHON:

`[*...]`: The asterisk (`*`) is the **unpacking operator**. It unpacks the elements from the map object into the surrounding list literal (`[]`), creating the final list.

Ruff setup:

- Install ruff: `pip install ruff`
- Open VSCode settings.
- Search for "python.analysis.autoImportCompletions".

data[0::2] is a list slice. It starts at the index of the first element, and steps by the second element, i.e. 2.

sorted(...) sorts each resulting list numerically.

A **lambda function** in Python is a small, anonymous function defined with the `lambda` keyword instead of `def`. It's typically used for short, throwaway functions that are written inline, such as in sorting or filtering operations.

Activate python venv: source venv/bin/activate

GOLANG:

:= is the short variable declaration operator for declaration + initialization purposes.

strings.TrimSpace(inputStr) removes leading/trailing whitespace.

fmt.Sscanf: Scans formatted data **from a string** (unlike `fmt.Scan` which reads from standard input).

`%d` is a **format verb** that matches and parses a base-10 integer (optionally signed). This is the common choice for parsing decimal integers. Other formatting verbs incllude %b and %x

In `fmt` scanning, **any whitespace in the format string (one or more spaces, tabs, newlines)** will match **any amount of whitespace in the input, including zero**.

&n1, &n2: These are pointers. You must pass pointers so `Sscanf` can store the parsed values into those variables. The variable types should be integer types (e.g. `int`, `int64`) compatible with what you expect to parse.

append(list, element) dynamically adds the element to the slice.

slices.Sort performs an in-place sort on the elements of the slices.

Go's math.Abs only works on float64, so explicit casting is required. Explicit casting in Go is when you manually convert a value from one type to another using the syntax `T(value)`, such as `float64(i)` or `int(f)`. Go requires explicit casting because it does not perform automatic (implicit) type conversions between different numeric types.

You open terminal in neovim using :term

Golang's "console.log()" is **fmt.Println().**

C++

To configure task runner in VS Code:

- Open C++ file:
- select **Terminal > Configure Default Build Task...**.
- Select a compiler, typically **`C/C++: g++.exe build active file`**
- This action creates a file named **`tasks.json`** inside a `.vscode` folder in your project directory. This file defines how VSCode compiles your code.

std::cout is the standard output stream (console).

The **<<** operator sends the string literal to the console.

std::endl inserts a newline character and flushes the buffer.

When including a **Standard Library header file** (like `iostream`), you must enclose the filename in **angle brackets (< >)**. This tells the compiler to look in the standard compiler include paths.

In C++, standard library components like `cout` and `endl` are inside the **`std` namespace** (short for "standard"). To access a member of a namespace, you must use the **scope resolution operator**, which is two colons (`::`).

The most popular and effective tool for C++ formatting is **Clang-Format**. It integrates perfectly with the VSCode C/C++ extension and handles **prettification** (code style enforcement) on save.

C++ code can be structured as member functions belonging to a class, meaning it will not build and run as a standalone executable file without the corresponding class definition (a .h file) and a main function to create an object of that class and call the methods. Conversely, it can be structured into a single, runnable .cpp file using a standard main function.

std::from_chars is a fast C++17 function for string-to-integer conversion. It reads from the string specified by [start, end).

std::sort(list1.begin(), list1.end()); is the sorting function used in C++.

std::abs from <cstdlib> is used for integer absolute value.

To fix the warning: 'auto' type specifier is a C++11 extension..., you must **explicitly pass the C++ standard flag** (`-std=c++20`) to the compiler within your `tasks.json`. This is done by adding **"-std=c++20",** to the args property to specify the C++ standard.

Some code can compile and run correctly without explicitly including `<string>`, `<vector>`, and `<algorithm>` is due to a concept called **transitive inclusion**. In C++, when you include one header file, that header file often includes other headers that it relies on to define its components. If header A includes header B, and header B includes header C, then a file including A effectively gets C as well.

- **`<iostream>`:** The `<iostream>` header (which provides `std::cout`, `std::endl`, etc.) is often implemented to include `<string>` internally, especially to define how standard output handles `std::string`.

- **`<fstream>`:** The `<fstream>` header (which provides `std::ifstream`, `is_open`, etc.) is almost guaranteed to include `<string>` internally because file paths and names are usually handled as strings. It also commonly includes `<iostream>`.

- **`<charconv>`:** The `<charconv>` header (used for `std::from_chars`) and other complex C++ headers (like `<map>`) often pull in utility headers that contain the definitions for things like **`std::vector`** and **`std::algorithm`** helper functions.

Relying on transitive inclusion is bad practice in professional C++ dev because:

- **Compiler/Library Dependent:** The specific headers included by another header are **not mandated** by the C++ standard. If you switch to a different compiler (e.g., from GCC to Clang), or if your current compiler library updates, the internal dependencies might change. Your code would then suddenly **fail to compile**.

- **Poor Readability:** Explicit includes tell other programmers (and yourself) exactly what dependencies your file has. Without them, it's impossible to know where `std::vector` or `std::sort` is being defined, making the code harder to maintain.

- **Lack of Self-Containment:** A header file or source file should be **self-contained**; that is, it should explicitly include everything it needs to compile.

`std::array` vs. `std::vector`

- **`std::vector` (Current Choice):** This is the **correct** choice for your problem. It is a **dynamic array** whose size is determined at runtime (when reading the file). Since you don't know exactly how many lines are in `input.txt` beforehand, you must use a dynamic container.
- **`std::array`:** This is a **fixed-size array** whose size is known at **compile time**. If you used `std::array`, you would have to define a fixed size (e.g., `std::array<int, 100>`) and your code would fail if the input file had more than 100 lines.

2. `.reverse()` / `std::reverse`

- **Usage:** The function `std::reverse()` (or the member function `.reverse()` on some containers, though not `std::vector`) is used to reverse the order of elements in a sequence.
- **Relevance to Current Logic:** Your current logic relies on **sorting** (`std::sort`) to find the minimum sum of absolute differences. It matches the smallest value in `list1` with the smallest value in `list2`, the second smallest with the second smallest, and so on. Reversing the lists after sorting would only change the calculation if your problem required pairing the largest with the smallest, but for the minimum sum of differences, simple sorting is correct. You do **not** need to use `.reverse()` here.

JAVASCRIPT

The use of functional decomposition:

the overall **asymptotic complexity** is dominated by the sorting step, making both $O(N \log N)$ where N is the number of data points.

1. Data reading: formatLists - Reads and splits the file once, iterates through the lines once, and cleanly separates the data. This avoids redundant ffile processing and string splitting.
2. Parsing / Type Coercion: +left and +right performs explicit, single type coercion during the initial read.
3. Calculation: The use of a standard forEach loop makes the code slightly faster due to simpler loop mechanics versus chained HOFs (High Order Functions) within a custom prototype method.