What I know, or got to know through search and LLMs is listed in the preface. I did not know what *Lexicographical* meant at first, and LLMs helped me to identify common pitfalls to watch out for when working with files.

-Preface-

Lexicographical ordering
Like a dictionary but for all characters.

Numbers (0-9): 48-57
Example: '0' < '1' < '2' ... < '9'

Uppercase (A-Z): 65-90
Example: 'A' < 'B' < 'C' ... < 'Z'

Lowercase (a-z): 97-122
Example: 'a' < 'b' < 'c' ... < 'z'

'A' < 'B' (true)
'A' < 'a' (true, uppercase comes before lowercase)
'9' < 'A' (true, numbers come before letters)
'z' < '{' (true)

"cat" vs "dog"
Compare

- 'c' vs 'd'
- 'c' comes before 'd'

Therefore "cat" < "dog" (dog comes after cat)

"cat" vs "car"
- 'c' vs 'c' (equal)
- 'a' vs 'a' (equal)
- 't' vs 'r'
- 't' comes after 'r'

Therefore "cat" > "car" (cat comes after car)

For clarification the notation:
"apple" (comes first)
"car" (comes second)
"cat" (comes third)
"dog" (comes fourth)

“apple” < “car” (apple comes before car or car is after apple)

Writing and reading from files

- We need to read the files line by line.
- Need to properly open/close files to avoid resource leaks

Edge cases

- Empty files
- Files with identical lines
- Lines with special characters
- Trailing, leading whitespace
- Different line endings (Linux, Windows)
- Very large files that will not fit into RAM

Memory efficiency

- Reading entire files into memory isn't scalable
- Line-by-line processing is better for large files
- Memory usage needs to be minimized

Things that might appear to be common pitfalls

- Not handling the line endings properly
- Forgetting to close files
- Not accounting for empty lines
- Assuming all lines are unique
- Not considering character encoding issues

I will be taking an iterative approach, meaning I will start with the most simple and likely inefficient example possible, and will layer on top it. Iteratively.

Just like in kubernetes, you create a simple example for a service you need, then you use it (copy-paste it) as reference to iterate upon.

-Solution-

All the work was done in the following github repository: *https://github.com/ktalap/test-problem1*

The iterations I went through below, follow the history of the commits made by me. The pdf file is available inside of the git repository too.

I will proceed commit by commit.
You can go straight to the commit history here: https://github.com/ktalap/test-problem1/commits/master

Simple Implementation
Commit: 52740737385f851fe399236377f0f97be0e412dd

I have implemented a very simple implementation of the task. There is a scanner that I got from a go library that goes through files line by line. And it puts values into a string/number key map:

“value of the line” : true

It has a big O complexity of 1, which means it doesn't matter how large the map gets, to retrieve a single entry will always take the same time.

Then as we create these maps for both of the files, we iterate through them with loops searching for unique lines. If we find a line that's not contained in either file2map or file1map we add it to the corresponding output file.

I do not want to leave things here as they are, I would like to benchmark the code and jack it up slightly (just kidding, jack it up to olympic level).

First let's implement benchmarking to see how much time it takes for executions.

Time benchmarking of the code
Commit: 084d37015c25ab85a7592b4eb7b8cf5bbdd06cd1

Simple time tracking with start time now in the beginning of the execution and calculating the time elapsed at the end. That's microseconds given small sample files. Let's generate something humongous in the next commit.

The time is given for running two tiny test sets with couple words, test-set-1 and test-set-2.

```
❯ go run ./
Files processed successfully
Execution time: 468.398µs
```

Generating testing files
Commit: 8cd2923ca4858abfc2cc02c7ec1b70672b6ddb74

The files we worked with so far are not as large as we may want them to be, that is why I used dictionaries available in linux on the following path:

/usr/share/dict/words

I had to install it on my opensuse tumbleweed distro beforehand but it must be in there by default in Ubuntu.

sudo zypper install words

I didn't do the file generation in the main file itself because it started to get messy inside and I didn't want to have no more than a single main.go file for this task, that's why I did it with a bash script.

I kept the size of the files to 20,000 bytes due to how long it was taking to generate them. I didn't want to optimize

the script. Thus with smaller amounts of lines, it still generates a decent amount of lines in each file, which is 2000.

The generation of the second file happens through taking 70% of the first file and adding more unique lines.

Running with the two large sample files
Commit: cc2a73f020c66410be2ff0af8af574fdfc04d532

```
❯ go run ./
Files processed successfully
Execution time: 13.561715ms
```

Something to work with, 13 ms.

Let's start optimizing it, I will set a bar for myself not to spend too much time on this. And I want to conclude the making of this task. I will do a single optimization.

Optimization of findDifferences function
Commit: 46a6f865f5af8a10028d7336a33f49dc19ed7a93

Here is what I have done:

- Preallocated capacity for the key-value maps. That helps us avoid the memory reallocation overhead.

- Increased scanner buffer size, which reduces the number of the system calls. Like buying groceries, sometimes it is better to do one big trip. Doesn't help with small files though.
- Batch processing for writing, also reduces the number of system calls.
- Buffered writers, instead of writing to disk immediately, it stores data in the memory buffer first. When the buffer is full, write everything at once. Waiting for a bucket to fill before pouring it out, instead of emptying a bucket with a single drop of water at a time.

Which brought us to:

```
❯ go run ./
Total time: 1.691217ms
Files processed successfully
```

13.6/1.7=8
8x increase in the speed. Well done!

Simple testing
Commit: b6cdc01c83088c02215f26759bcbf67236731e77

I don't think any tests are needed for a simple program like this, but something tells me you want to see some testing of the code, even if it is simple. Here is the final commit with that.

It executes the function, compares the result and cleans up after itself afterwards.

-Conclusion-

And that is it. Thank you.