Interview Cake

You left your computer unlocked and your friend decided to troll you by copying a lot of your files to random spots all over your file system.

Even worse, she saved the duplicate files with random, embarrassing names ("this_is_like_a_digital_wedgie.txt" was clever, I'll give her that).

Write a function that returns a list of all the duplicate files. We'll check them by hand before actually deleting them, since programmatically deleting files is really scary. To help us confirm that two files are actually duplicates, return a list of tuples.

A **tuple** is like a list:

```
(17, 3, "My name is Parker")
```
Python 2.7

(Tuples are written with parentheses to differentiate them from lists.)

Like lists, tuples are **ordered** and you can access elements by their index:

```
cake_tuple = ('angel', 'bundt')

cake_tuple[0]
# returns: 'angel'
```
Python 2.7

But tuples are **immutable!** They can't be edited after they're created.

```
cake_tuple = ('angel', 'bundt')
cake_tuple[1] = 'carrot'
# raises: TypeError: 'tuple' object does not support item assignment
```
Python 2.7

Tuples can have **any number of elements** (the 'tu' in tuple doesn't mean 'two', it's just a generic name taken from words like 'septuple' and 'octuple').

```
(90, 4, 54)
(True, False, True, True, False)
```

Python 2.7

where:

- the **first** item is the **duplicate** file
- the **second** item is the **original** file

For example:

```
[('/tmp/parker_is_dumb.mpg', '/home/parker/secret_puppy_dance.mpg'),
 ('/home/trololol.mov', '/etc/apache2/httpd.conf')]
```

You can assume each file was only duplicated once.

Gotchas

Are you correctly handling child folders as well as sibling folders? Be careful that you're traversing your file tree correctly...

When you find two files that are the same, don't just choose a random one to mark as the "duplicate." Try to figure out which one your friend made!

Does your solution work correctly if it's an empty file system (meaning the root directory is empty)?

Our solution takes $O(n)$ time and space, where n is the *number of files*. Is your solution order of the *total size on disc of all the files*? If so, you can do better!

To get our time and space costs down, we took a small hit on accuracy—we might get a small number of false positives. We're okay with that since we'll double-check before actually deleting files.

Breakdown

> **No idea where to start?** Try writing something that just walks through a file system and prints all the file names. If you're not sure how to do that, look it up! Or just *make it up*. Remember, even if you can't implement *working code*, your interviewer will still want to see you *think through* the problem.

One brute force solution is to loop over all files in the file system, and for each file look at every *other* file to see if it's a duplicate. This means n^2 file comparisons, where n is the number of files. That seems like a lot.

Let's try to save some time. Can we do this in *one* walk through our file system?

Instead of holding onto one file and looking for files that are the same, we can just keep track of *all* the files we've seen so far. What data structure could help us with that?

We'll use a dictionary.

A **hash table** organizes data so you can quickly look up values for a given key.

	Average	Worst Case
space	$O(n)$	$O(n)$
insert	$O(1)$	$O(n)$
lookup	$O(1)$	$O(n)$
delete	$O(1)$	$O(n)$

Strengths:

- **Fast lookups**. Lookups take $O(1)$ time *on average*.
- **Flexible keys**. Most data types can be used for keys, as long as they're hashable (/concept/hashing).

Weaknesses:

- **Slow worst-case lookups**. Lookups take $O(n)$ time *in the worst case*.
- **Unordered**. Keys aren't stored in a special order. If you're looking for the smallest key, the largest key, or all the keys in a range, you'll need to look through every key to find it.
- **Single-directional lookups**. While you can look up the *value* for a given key in $O(1)$ time, looking up the *keys* for a given *value* requires looping through the whole dataset—$O(n)$ time.
- **Not cache-friendly**. Many hash table implementations use linked lists (/concept/linked-list), which don't put data next to each other in memory.

In Python 3.6

In Python 3.6, hash tables are called dictionaries.

Python 3.6

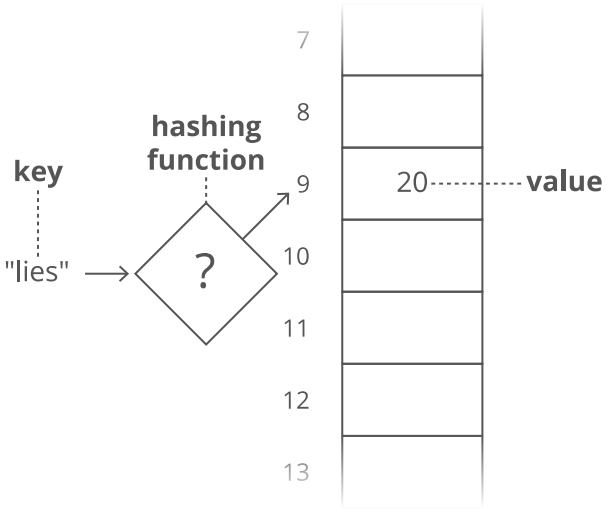
```
light_bulb_to_hours_of_light = {
    'incandescent': 1200,
    'compact fluorescent': 10000,
    'LED': 50000,
}
```

Hash maps are built on arrays

Arrays (/concept/array) are pretty similar to hash maps already. Arrays let you quickly look up the value for a given "key" . . . except the keys are called "indices," and we don't get to pick them—they're always sequential integers (0, 1, 2, 3, etc).

Think of a hash map as a "hack" on top of an array to let us use flexible keys instead of being stuck with sequential integer "indices."

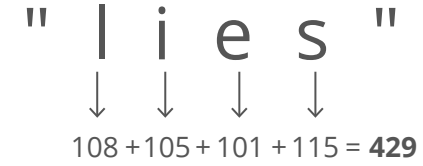
All we need is a function to convert a key into an array index (an integer). That function is called a **hashing function (/concept/hashing)**.



To look up the value for a given key, we just run the key through our hashing function to get the index to go to in our underlying array to grab the value.

How does that hashing function work? There are a few different approaches, and they can get pretty complicated. But here's a simple proof of concept:

Grab the number value for each character and add those up.



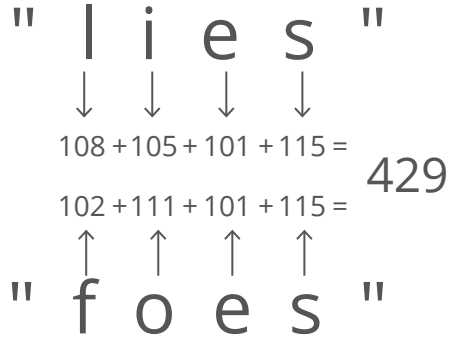
The result is 429. But what if we only have *30* slots in our array? We'll use a common trick for forcing a number into a specific range: the modulus operator (`%`). (/concept/modulus) Modding our sum by 30 ensures we get a whole number that's less than 30 (and at least 0):

$$429 \% 30 = 9$$

> The hashing functions used in modern systems get pretty complicated—the one we used here is a simplified example.

Hash collisions

What if two keys hash to the same index in our array? In our example above, look at "lies" and "foes":

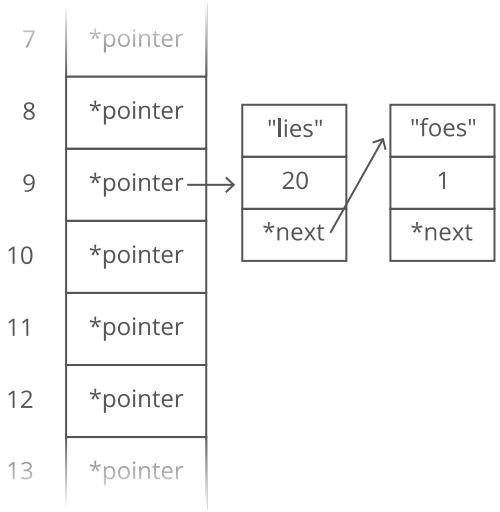


They both sum up to 429! So of course they'll have the same answer when we mod by 30:

$$429 \% 30 = 9$$

This is called a **hash collision**. There are a few different strategies for dealing with them.

Here's a common one: instead of storing the actual values in our array, let's have each array slot hold a *pointer* to a *linked list (/concept/linked-list)* holding the values for all the keys that hash to that index:



Notice that we included the *keys* as well as the values in each linked list node. Otherwise we wouldn't know which key was for which value!

> There are other ways to deal with hash collisions. This is just one of them.

When hash table operations cost $O(n)$ time

Hash collisions

If *all* our keys caused hash collisions, we'd be at risk of having to walk through all of our values for a single lookup (in the example above, we'd have one big linked list). This is unlikely, but it *could* happen. That's the worst case.

Dynamic array resizing

Suppose we keep adding more items to our hash map. As the number of keys and values in our hash map exceeds the number of indices in the underlying array, hash collisions become inevitable.

To mitigate this, we could expand our underlying array whenever things start to get crowded. That requires allocating a larger array and rehashing all of our existing keys to figure out their new position—$O(n)$ time.

Sets

A **set** is like a hash map except it only stores keys, without values.

Sets often come up when we're tracking groups of items—nodes we've visited in a graph, characters we've seen in a string, or colors used by neighboring nodes. Usually, we're interested in whether something is in a set or not.

Sets are usually implemented very similarly to hash maps—using hashing to index into an array—but they don't have to worry about storing values alongside keys. In Python, the set implementation is largely copied from the dictionary implementation (https://markmail.org/message/ktzomp4uwrmnzao6).

Python 3.6

```
light_bulbs = set()

light_bulbs.add('incandescent')
light_bulbs.add('compact fluorescent')
light_bulbs.add('LED')

'LED' in light_bulbs  # True
'halogen' in light_bulbs  # False
```

When we see a new file, we first check to see if it's in our dictionary. If it's not, we add it. If it is, we have a duplicate!

Once we have two duplicate files, how do we know which one is the original? It's hard to be sure, but try to come up with a reasonable heuristic that will probably work most of the time.

Most file systems store the time a file was last edited as metadata on each file. The more recently edited file will *probably* be the duplicate!

One exception here: lots of processes like to regularly save their state to a file on disc, so that if your computer suddenly crashes the processes can pick up more or less where they left off (this is how Word is able to say "looks like you had unsaved changes last time, want to restore them?"). If your friend duplicated some of *those* files, the most-recently-edited one may *not* be the duplicate. But at the risk of breaking our system (we'll make a backup first, obviously.) we'll run with this "most-recently-edited copy of a file is probably the copy our friend made" heuristic.

So our function will walk through the file system, store files in a dictionary, and identify the more recently edited file as the copied one when it finds a duplicate. Can you implement this in code?

Here's a start. We'll initialize:

1. a **dictionary** to hold the files we've already seen
2. a **stack**

A **stack** stores items in a last-in, first-out (LIFO) order.

Picture a pile of dirty plates in your sink. As you add more plates, you bury the old ones further down. When you take a plate off the top to wash it, you're taking the last plate you put in. "Last in, first out."

	Worst Case
space	$O(n)$
push	$O(1)$
pop	$O(1)$
peek	$O(1)$

Strengths:

- **Fast operations**. All stack operations take $O(1)$ time.

Uses:

- **The call stack** is a stack that tracks function calls in a program. When a function returns, which function do we "pop" back to? The last one that "pushed" a function call.
- **Depth-first search** (/concept/dfs) uses a stack (sometimes the call stack) to keep track of which nodes to visit next.
- **String parsing**—stacks turn out to be useful for several types of string parsing (/question/bracket-validator).

Implementation

You can implement a stack with either a linked list (/concept/linked-list) or a dynamic array (/concept/dynamic-array)—they both work pretty well:

	Stack Push	Stack Pop
Linked Lists	insert at head	remove at head
Dynamic Arrays	append	remove last element

(we'll implement ours with a list) to hold directories and files as we go through them
3. a **list** to hold our output tuples

Python 3.6

```
def find_duplicate_files(starting_directory):
    files_seen_already = {}
    stack = [starting_directory]

    # We'll track tuples of (duplicate_file, original_file)
    duplicates = []

    while len(stack):
        current_path = stack.pop()

    return duplicates
```

(We're going to make our function iterative instead of recursive to avoid stack overflow.)

Overview

The **call stack** is what a program uses to keep track of function calls. The call stack is made up of **stack frames**—one for each function call.

For instance, say we called a function that rolled two dice and printed the sum.

Python 3.6

```
def roll_die():
    return random.randint(1, 6)

def roll_two_and_sum():
    total = 0
    total += roll_die()
    total += roll_die()
    print(total)

roll_two_and_sum()
```

First, our program calls `roll_two_and_sum()`. It goes on the call stack:

```
roll_two_and_sum()
```

That function calls `roll_die()`, which gets pushed on to the top of the call stack:

```
roll_die()
```

```
roll_two_and_sum()
```

Inside of `roll_die()`, we call `random.randint()`. Here's what our call stack looks like then:

```
random.randint()
```

```
roll_die()
```

```
roll_two_and_sum()
```

When `random.randint()` finishes, we return back to `roll_die()` by removing ("popping") `random.randint()`'s stack frame.

```
roll_die()
```

```
roll_two_and_sum()
```

Same thing when `roll_die()` returns:

```
roll_two_and_sum()
```

We're not done yet! `roll_two_and_sum()` calls `roll_die()` *again*:

```
roll_die()
```

```
roll_two_and_sum()
```

Which calls `random.randint()` again:

```
random.randint()
```

```
roll_die()
```

```
roll_two_and_sum()
```

`random.randint()` returns, then `roll_die()` returns, putting us back in `roll_two_and_sum()`:

```
roll_two_and_sum()
```

Which calls `print()()`:

```
print()()
```

```
roll_two_and_sum()
```

What's stored in a stack frame?

What *actually* goes in a function's stack frame?

A stack frame usually stores:

- Local variables
- Arguments passed into the function
- Information about the caller's stack frame
- The *return address*—what the program should do after the function returns (i.e.: where it should "return to"). This is usually somewhere in the middle of the caller's code.

> Some of the specifics vary between processor architectures. For instance, AMD64 (64-bit x86) processors pass some arguments in registers and some on the call stack. And, ARM processors (common in phones) store the return address in a special register instead of putting it on the call stack.

The Space Cost of Stack Frames

Each function call creates its own stack frame, taking up space on the call stack. That's important because it can impact the *space complexity* of an algorithm. *Especially* when we use **recursion**.

For example, if we wanted to multiply all the numbers between 1 and n, we could use this recursive approach:

Python 3.6

```
def product_1_to_n(n):
    return 1 if n <= 1 else n * product_1_to_n(n - 1)
```

What would the call stack look like when $n = 10$?

First, `product_1_to_n()` gets called with $n = 10$:

```
    product_1_to_n()     n = 10
```

This calls `product_1_to_n()` with $n = 9$.

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

Which calls `product_1_to_n()` with $n = 8$.

```
product_1_to_n()    n = 8
```

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

And so on until we get to $n = 1$.

```
product_1_to_n()    n = 1
```

```
product_1_to_n()    n = 2
```

```
product_1_to_n()    n = 3
```

```
product_1_to_n()    n = 4
```

```
product_1_to_n()    n = 5
```

```
product_1_to_n()    n = 6
```

```
product_1_to_n()    n = 7
```

```
product_1_to_n()    n = 8
```

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

Look at the size of all those stack frames! The entire call stack takes up $O(n)$ space. That's right—we have an $O(n)$ space cost even though our function itself doesn't create any data structures!

What if we'd used an iterative approach instead of a recursive one?

Python 3.6

```
def product_1_to_n(n):
    # We assume n >= 1
    result = 1
    for num in range(1, n + 1):
        result *= num

    return result
```

This version takes a constant amount of space. At the beginning of the loop, the call stack looks like this:

```
product_1_to_n()    n = 10, result = 1, num = 1
```

As we iterate through the loop, the local variables change, but we stay in the same stack frame because we don't call any other functions.

```
product_1_to_n()    n = 10, result = 2, num = 2
```

```
product_1_to_n()    n = 10, result = 6, num = 3
```

```
product_1_to_n()    n = 10, result = 24, num = 4
```

In general, even though the compiler or interpreter will take care of managing the call stack for you, it's important to consider the depth of the call stack when analyzing the space complexity of an algorithm.

Be especially careful with recursive functions! They can end up building huge call stacks.

> What happens if we run out of space? It's a **stack overflow**! In Python 3.6, you'll get a `RecursionError`.

> If the *very last* thing a function does is call another function, then its stack frame might not be needed any more. The function *could* free up its stack frame before doing its final call, saving space.

> This is called **tail call optimization** (TCO). If a recursive function is optimized with TCO, then it may not end up with a big call stack.
>
> In general, most languages *don't* provide TCO. Scheme is one of the few languages that guarantee tail call optimization. Some Ruby, C, and Javascript implementations *may* do it. Python and Java decidedly don't.

)

Here's one solution:

Python 3.6

```
import os

def find_duplicate_files(starting_directory):
    files_seen_already = {}
    stack = [starting_directory]

    # We'll track tuples of (duplicate_file, original_file)
    duplicates = []

    while len(stack) > 0:
        current_path = stack.pop()

        # If it's a directory,
        # put the contents in our stack
        if os.path.isdir(current_path):
            for path in os.listdir(current_path):
                full_path = os.path.join(current_path, path)
                stack.append(full_path)

        # If it's a file
        else:
            # Get its contents
            with open(current_path) as file:
                file_contents = file.read()

            # Get its last edited time
            current_last_edited_time = os.path.getmtime(current_path)

            # If we've seen it before
            if file_contents in files_seen_already:
                existing_last_edited_time, existing_path = files_seen_already[file_contents]
                if current_last_edited_time > existing_last_edited_time:
                    # Current file is the dupe!
                    duplicates.append((current_path, existing_path))
                else:
                    # Old file is the dupe!
                    # So delete it
                    duplicates.append((existing_path, current_path))
                    # But also update files_seen_already to have
                    # the new file's info
```

```
                files_seen_already[file_contents] = (current_last_edited_time, current_

            # If it's a new file, throw it in files_seen_already
            # and record the path and the last edited time,
            # so we can delete it later if it's a dupe
            else:
                files_seen_already[file_contents] = (current_last_edited_time, current_path)

    return duplicates
```

Okay, this'll work! What are our time and space costs?

We're putting the full contents of every file in our dictionary! This costs $O(b)$ time and space, where b is the *total amount of space taken up by all the files on the file system*.

That space cost is pretty unwieldy—we need to store a duplicate copy of our entire filesystem (like, several gigabytes of cat videos alone) in working memory!

Can we trim that space cost down? What if we're okay with losing a bit of accuracy (as in, we do a more "fuzzy" match to see if two files are the same)?

What if instead of making our dictionary keys *the entire file contents*, we hashed↴

A **hash function** takes data (like a string, or a file's contents) and outputs a *hash*, a fixed-size string or number.

For example, here's the MD5 hash (MD5 is a common hash function) for a file simply containing "cake":

```
DF7CE038E2FA96EDF39206F898DF134D
```

And here's the hash for the same file after it was edited to be "cakes":

```
0E9091167610558FDAE6F69BD6716771
```

Notice the hash is *completely* different, even though the files were similar. Here's the hash for a long film I have on my hard drive:

```
664f67364296d08f31aec6fea4e9b83f
```

The hash is the same length as my other hashes, but this time it represents a much bigger file—461Mb.

We can think of a hash as a "fingerprint." We can trust that a given file will always have the same hash, but we can't go from the hash back to the original file. Sometimes we have to worry about multiple files having the same hash value, which is called a **hash collision**.

Some uses for hashing:

1. **Dictionaries**. Suppose we want a list-like data structure with constant-time lookups, but we want to look up values based on arbitrary "keys," not just sequential "indices." We could allocate a list, and use a hash function to translate keys into list indices. That's the basic idea behind a dictionary!
2. **Preventing man-in-the-middle attacks**. Ever notice those things that say "hash" or "md5" or "sha1" on download sites? The site is telling you, "We hashed this file on our end and got this result. When you finish the download, try hashing the file and confirming you get the same result. If not, your internet service provider or someone else might have injected malware or tracking software into your download!"

those contents first? So we'd store a constant-size "fingerprint" of the file in our dictionary, instead of the whole file itself. This would give us $O(1)$ space per file ($O(n)$ space overall, where n is the number of files)!

That's a huge improvement. But we can take this a step further! While we're making the file matching "fuzzy," can we use a similar idea to save some *time*? Notice that our time cost is still order of the total *size* of our files on disc, while our space cost is order of the *number* of files.

For each file, we have to look at every bit that the file occupies in order to hash it and take a "fingerprint." That's why our time cost is high. Can we fingerprint a file in *constant* time instead?

What if instead of hashing the *whole* contents of each file, we hashed three fixed-size "samples" from each file made of the first x bytes, the middle x bytes, and the last x bytes? This would let us fingerprint a file in constant time!

How big should we make our samples?

When your disc does a read, it grabs contents in constant-size chunks, called "blocks."

How big are the blocks? It depends on the file system. My super-hip Macintosh uses a file system called HFS+, which has a default block size of 4Kb (4,000 bytes) per block.

So we could use just 100 bytes each from the beginning middle and end of our files, but each time we grabbed those bytes, our disc would actually be grabbing 4000 bytes, not just 100 bytes. We'd just be throwing the rest away. We might as well use all of them, since having a bigger picture of the file helps us ensure that the fingerprints are unique. So our samples should be the the size of our file system's block size.

Solution

We walk through our whole file system iteratively. As we go, we take a "fingerprint" of each file in constant time by hashing the first few, middle few, and last few bytes. We store each file's fingerprint in a *dictionary* as we go.

If a given file's fingerprint is already in our dictionary, we assume we have a duplicate. In that case, we assume the file edited most recently is the one created by our friend.

Python 3.6

```
import os
import hashlib

def find_duplicate_files(starting_directory):
    files_seen_already = {}
    stack = [starting_directory]

    # We'll track tuples of (duplicate_file, original_file)
    duplicates = []

    while len(stack):
        current_path = stack.pop()

        # If it's a directory,
        # put the contents in our stack
        if os.path.isdir(current_path):
            for path in os.listdir(current_path):
                full_path = os.path.join(current_path, path)
                stack.append(full_path)

        # If it's a file
        else:
            # Get its hash
            file_hash = sample_hash_file(current_path)

            # Get its last edited time
            current_last_edited_time = os.path.getmtime(current_path)

            # If we've seen it before
            if file_hash in files_seen_already:
                existing_last_edited_time, existing_path = files_seen_already[file_hash]
                if current_last_edited_time > existing_last_edited_time:
                    # Current file is the dupe!
                    duplicates.append((current_path, existing_path))
                else:
                    # Old file is the dupe!
                    duplicates.append((existing_path, current_path))
                    # But also update files_seen_already to have
                    # the new file's info
                    files_seen_already[file_hash] = (current_last_edited_time, current_path
```

```
            # If it's a new file, throw it in files_seen_already
            # and record its path and last edited time,
            # so we can tell later if it's a dupe
            else:
                files_seen_already[file_hash] = (current_last_edited_time, current_path)

    return duplicates


def sample_hash_file(path):
    num_bytes_to_read_per_sample = 4000
    total_bytes = os.path.getsize(path)
    hasher = hashlib.sha512()

    with open(path, 'rb') as file:
        # If the file is too short to take 3 samples, hash the entire file
        if total_bytes < num_bytes_to_read_per_sample * 3:
            hasher.update(file.read())
        else:
            num_bytes_between_samples = (
                (total_bytes - num_bytes_to_read_per_sample * 3) / 2
            )

            # Read first, middle, and last bytes
            for offset_multiplier in range(3):
                start_of_sample = (
                    offset_multiplier
                    * (num_bytes_to_read_per_sample + num_bytes_between_samples)
                )
                file.seek(start_of_sample)
                sample = file.read(num_bytes_to_read_per_sample)
                hasher.update(sample)

    return hasher.hexdigest()
```

We've made a few assumptions here:

Two *different* files won't have the same fingerprints. It's not impossible that two files with different contents will have the same beginning, middle, and end bytes so they'll have the same fingerprints. Or they may even have different sample bytes but still hash to the same value (this is called a "hash collision"). To mitigate this, we could do a last-minute check whenever we find two "matching" files where we actually scan the full file contents to see if they're the same.

The *most recently edited* file is the duplicate. This seems reasonable, but it *might* be wrong—for example, there might be files which have been edited by daemons (programs that run in the background) *after* our friend finished duplicating them.

Two files with the same contents are the same file. This seems trivially true, but it could cause some problems. For example, we might have empty files in multiple places in our file system that aren't duplicates of each-other.

Given these potential issues, we definitely want a human to confirm before we delete any files. Still, it's much better than combing through our whole file system by hand!

Some ideas for further improvements:

1. If a file wasn't last edited around the time your friend got a hold of your computer, you know it probably wasn't created by your friend. Similarly, if a file wasn't *accessed* (sometimes your filesystem stores the last accessed time for a file as well) around that time, you know it wasn't copied by your friend. You can use these facts to skip some files.
2. Make the file *size* the fingerprint—it should be available cheaply as metadata on the file (so you don't need to walk through the whole file to see how long it is). You'll get lots of false positives, but that's fine if you treat this as a "preprocessing" step. Maybe you *then* take hash-based fingerprints only on the files which which have matching sizes. *Then* you fully compare file contents if they have the same hash.
3. Some file systems also keep track of when a file was *created*. If your filesystem supports this, you could use this as a potentially-stronger heuristic for telling which of two copies of a file is the dupe.
4. When you *do* compare full file contents to ensure two files are the same, no need to read the entire files into memory. Open both files and read them one block at a time. You can short-circuit as soon as you find two blocks that don't match, and you only ever need to store a couple blocks in memory.

Complexity

Each "fingerprint" takes $O(1)$ time and space, so our total time and space costs are $O(n)$ where n is the *number of files* on the file system.

If we add the last-minute check to see if two files with the same fingerprints are *actually* the same files (which we probably should), then in the worst case *all the files are the same* and we have to read their full contents to confirm this, giving us a runtime that's order of the total size of our files on disc.

Bonus

If we wanted to get this code ready for a production system, we might want to make it a bit more modular. Try separating the file traversal code from the duplicate detection code. Try implementing the file traversal with a generator!

What about concurrency? Can we go faster by splitting this procedure into multiple threads? Also, what if a background process edits a file *while our script is running*? Will this cause problems?

What about link files (files that point to other files or folders)? One gotcha here is that a link file can point *back up the file tree*. How do we keep our file traversal from going in circles?

What We Learned

The main insight was to save time and space by "fingerprinting" each file.

This question is a good example of a "messy" interview problem. Instead of one optimal solution, there's a big knot of optimizations and trade-offs. For example, our hashing-based approach wins us a faster runtime, but it can give us false positives.

For messy problems like this, focus on clearly explaining to your interviewer what the trade-offs are for each decision you make. The actual choices you make probably don't matter that much, as long as you show a strong ability to understand and compare your options.

Ready for more?

Check out our full course ➡

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.