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 vector 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 vector of `FilePaths` objects with variables for the original and duplicate paths :

C++

```
class FilePaths
{
public:
    string duplicatePath_;
    string originalPath_;

    FilePaths(const string& duplicatePath, const string& originalPath) :
        duplicatePath_(duplicatePath),
        originalPath_(originalPath)
    {
    }

    string toString() const
    {
        ostringstream str;
        str  << "(original: " << filePaths.originalPath_
            << ", duplicate: " << filePaths.duplicatePath_ << ")";
        return str.str();
    }
};
```

For example:

```
(original: /home/parker/secret_puppy_dance.mpg, duplicate: /tmp/parker_is_dumb.mpg)
(original: /etc/apache2/httpd.conf, duplicate: /home/trololol.mov)
```

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 unordered map. When we see a new file, we first check to see if it's in our unordered map. 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, obvi.) 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 an unordered map, 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. an **unordered map** to hold the files we've already seen
2. a **stack** to hold directories and files as we go through them
3. a **vector** to hold our output `FilePaths` object s

```C++
vector<FilePaths> findDuplicateFiles(const string& startingDirectory)
{
    unordered_map<string, FileInfo> filesSeenAlready;
    stack<string> pathsStack;
    pathsStack.push(startingDirectory);

    vector<FilePaths> duplicates;

    while (!stack.empty()) {

        string currentPath = pathsStack.top();
        pathsStack.pop();

    }
}
```

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

Here's one solution:

C++

```
#include <cstring>
#include <iostream>
#include <stack>
#include <string>
#include <unordered_map>
#include <vector>

#include <dirent.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

using namespace std;

class FilePaths
{
public:
    string duplicatePath_;
    string originalPath_;

    FilePaths(const string& duplicatePath = string(),
            const string& originalPath = string()) :
        duplicatePath_(duplicatePath),
        originalPath_(originalPath)
    {
    }

    string toString() const
    {
        ostringstream str;
        str  << "(original: " << filePaths.originalPath_
            << ", duplicate: " << filePaths.duplicatePath_ << ")";
        return str.str();
    }
};

class FileInfo
{
public:
```

```
    string path_;
    time_t lastModified_;

    FileInfo(const string& path, time_t lastModified) :
        path_(path),
        lastModified_(lastModified)
    {
    }
};

// We wrap our directory handle in a class to make it exception safe
// and ensure the directory always gets closed.
class DirectoryHandle
{
public:
    DIR* dir_;

    DirectoryHandle(const char* path) :
        dir_(opendir(path))
    {
    }

    ~DirectoryHandle()
    {
        if (_dir != nullptr) {
            closedir(dir_);
        }
    }
};

void scanDirectory(const string& directoryPath, stack<string>& pathsStack)
{
    DirectoryHandle dir(directoryPath.c_str());
    if (dir.dir_) {
        struct dirent* entry;
        while ((entry = readdir(dir.dir_)) != nullptr) {
            if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
                string path = directoryPath + '/' + entry->d_name;
                pathsStack.push(path);
            }
        }
```

```
    }
}

vector<unsigned char> readFile(const string& path, size_t sizeHint)
{
    // try to read file contents
    bool fileContentsRetrieved = false;
    vector<unsigned char> fileContents(sizeHint);
    int fd = open(path.c_str(), O_RDONLY);
    if (fd != -1) {
        ssize_t actuallyRead = read(fd, static_cast<void*>(&fileContents[0]), fileContents.
        close(fd);
        if (actuallyRead == fileContents.size()) {
            fileContentsRetrieved = true;
        }
        else {
            cerr << "Error: Couldn't read file '" << currentPath << "'." << endl;
        }
    }
    else {
        cerr << "Error: Couldn't open file '" << currentPath << "' for reading." << endl;
    }

    // we'll return an empty vector to indicate a problem reading the file
    if (!fileContentsRetrieved) {
        fileContents.clear();
    }

    return fileContents;
}

vector<FilePaths> findDuplicateFilesIterative(const string& startingDirectory)
{
    unordered_map<vector<unsigned char>, FileInfo> filesSeenAlready;
    stack<string> pathsStack;
    pathsStack.push(startingDirectory);

    vector<FilePaths> duplicates;

    while (!pathsStack.empty()) {
        string currentPath = pathsStack.top();
```

```
        pathsStack.pop();

        struct stat st;
        if (stat(currentPath.c_str(), &st) < 0) {
            cerr << "Error: Can't stat file '" << currentPath << "'." << endl;
            continue;
        }

        // if it's a directory,
        // put the contents in our stack
        if (S_ISDIR(st.st_mode)) {

            scanDirectory(currentPath, pathsStack);

        // if it's a file
        }
        else if (S_ISREG(st.st_mode)) {

            // get its contents
            vector<unsigned char> fileContents = readFile(currentPath, st.st_size);

            // if we've seen it before
            if (!fileContents.empty()) {
                auto it = filesSeenAlready.find(fileContents);
                if (it != filesSeenAlready.end()) {
                    // compare with its last edited time
                    if (st.st_mtime > it->second.lastModified_) {

                        // current file is the dupe!
                        duplicates.emplace(duplicates.end(), currentPath, it->second.path_)

                    }
                    else {

                        // old file is the dupe!
                        duplicates.emplace(duplicates.end(), it->second.path_, currentPath)

                        // but also update filesSeenAlready to have the new file's info
                        it->second.path_ = currentPath;
                        it->second.lastModified_ = st.st_mtime;
                    }
```

```
                }
                // if it's a new file, throw it in filesSeenAlready
                // and record its path and last edited time,
                // so we can tell later if it's a dupe
                else {
                    filesSeenAlready.insert(make_pair(fileContents,
                            FileInfo(currentPath, st.st_mtime)));
                }
            }
        }
    }

    return duplicates;
}
```

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

We're putting the full contents of every file in our unordered map! 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 filsystem (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 unordered map keys *the entire file contents*, we hashed those contents first? So we'd store a constant-size "fingerprint" of the file in our unordered map, 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 an *unordered map* as we go.

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

```
#include <cstring>
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <stack>
#include <string>
#include <unordered_map>
#include <vector>

#include <dirent.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#include <openssl/sha.h>

using namespace std;

class FilePaths
{
public:
    string duplicatePath_;
    string originalPath_;

    FilePaths(const string& duplicatePath = string(),
            const string& originalPath = string()) :
        duplicatePath_(duplicatePath),
        originalPath_(originalPath)
    {
    }

    string toString() const
    {
        ostringstream str;
        str  << "(original: " << filePaths.originalPath_
            << ", duplicate: " << filePaths.duplicatePath_ << ")";
        return str.str();
    }
```

```
};

class FileInfo
{
public:
    string path_;
    time_t lastModified_;

    FileInfo(const string& path, time_t lastModified) :
        path_(path),
        lastModified_(lastModified)
    {
    }
};



// We wrap our directory handle in a class to make it exception safe
// and ensure the directory always gets closed.
class DirectoryHandle
{
public:
    DIR* dir_;

    DirectoryHandle(const char* path) :
        dir_(opendir(path))
    {
    }

    ~DirectoryHandle()
    {
        if (_dir != nullptr) {
            closedir(dir_);
        }
    }
};

void scanDirectory(const string& directoryPath, stack<string>& pathsStack)
{
    DirectoryHandle dir(directoryPath.c_str());
    if (dir.dir_) {
        struct dirent* entry;
```

```
        while ((entry = readdir(dir.dir_)) != nullptr) {
            if (strcmp(entry->d_name, ".") != 0 && strcmp(entry->d_name, "..") != 0) {
                string path = directoryPath + '/' + entry->d_name;
                pathsStack.push(path);
            }
        }
    }
}

vector<unsigned char> sampleFileData(const string& path)
{
    // read file attributes
    struct stat st;
    if (stat(path.c_str(), &st) < 0) {
        ostringstream errorMessage;
        errorMessage << "Can't stat file '" << path << "'." << endl;
        throw runtime_error(errorMessage.str());
    }
    if (!S_ISREG(st.st_mode)) {
        ostringstream errorMessage;
        errorMessage << "File '" << path << "' is not a regular file.";
        throw runtime_error(errorMessage.str());;
    }

    // determine how much to read from file
    const off_t BLOCK_SIZE = 4000;
    const off_t MAX_DATA_SIZE = BLOCK_SIZE * 3;
    const off_t dataSize = std::min(st.st_size, MAX_DATA_SIZE);

    // reserve storage of the appropriate size for file data
    vector<unsigned char> fileData(dataSize);

    // Read the file using low-level system call based file I/O.
    // (C++ STL may not properly handle files larger than 2 GB.)

    // Open file
    int fd = open(path.c_str(), O_RDONLY);
    if (fd != -1) {
        // file opened, read it
        ssize_t readLength  = 0;
```

```
        // If size is too small to take 3 samples, read the entire file
        // into the provided buffer, updating readLength.
        if (dataSize <= MAX_DATA_SIZE) {
            readLength = read(fd, static_cast<void*>(&fileData[0]), dataSize);
        }
        // Otherwise, read the samples into the buffer at the right offsets,
        // and update the readLength.
        else {
            readLength += read(fd,
                static_cast<void*>(&fileData[0]), BLOCK_SIZE);
            if (lseek(fd, (st.st_size / 2) - (BLOCK_SIZE / 2), SEEK_SET) == (off_t) -1) {
                throw runtime_error("Can't lseek file.");
            }
            readLength += read(fd,
                static_cast<void*>(&fileData[BLOCK_SIZE]), BLOCK_SIZE);
            if (lseek(fd, -BLOCK_SIZE, SEEK_END) == (off_t) -1) {
                throw runtime_error("Can't lseek file.");
            }
            readLength += read(fd,
                static_cast<void*>(&fileData[BLOCK_SIZE * 2]), BLOCK_SIZE);
        }

        close(fd);

        // Check whether we have read proper number of bytes
        if (readLength != dataSize) {
            ostringstream errorMessage;
            errorMessage << "Couldn't read file '" << path << "'.";
            throw runtime_error(errorMessage.str());
        }
    }
    // file wasn't opened
    else {
        ostringstream errorMessage;
        errorMessage << "Couldn't open file '" << path << "' for reading.";
        throw runtime_error(errorMessage.str());
    }

    return fileData;
}
```

```
string sampleFileHash(const string& path)
{
    try {
        // get data from file
        vector<unisgned char> fileData = sampleFileData(path);

        // compute SHA-512 hash
        ostringstream result;
        unsigned char hashValue[SHA512_DIGEST_LENGTH];
        SHA512(&fileData[0], fileData.size(), hashValue);
        result << hex;
        for (size_t i = 0; i < SHA512_DIGEST_LENGTH; ++i) {
            if (hashValue[i] < 16) {
                result << '0';
            }
            result << static_cast<unsigned int>(hashValue[i]);
        }

        return result.str();
    }
    catch (exception& ex) {
        cerr << "Error: " << ex.what() << endl;
        return string();
    }
}

vector<FilePaths> findDuplicateFiles(const string& startingDirectory)
{
    unordered_map<string, FileInfo> filesSeenAlready;
    stack<string> pathsStack;
    pathsStack.push(startingDirectory);

    vector<FilePaths> duplicates;

    while (!pathsStack.empty()) {

        string currentPath = pathsStack.top();
        pathsStack.pop();

        struct stat st;
        if (stat(currentPath.c_str(), &st) < 0) {
```

```
            cerr << "Error: Can't stat file '" << currentPath << "'." << endl;
            continue;
        }

        // if it's a directory,
        // put the contents in our stack
        if (S_ISDIR(st.st_mode)) {
            scanDirectory(currentPath, pathsStack);
        }
        // if it's a file
        else if (S_ISREG(st.st_mode)) {
            string hash = sampleFileHash(currentPath);

            // if we've seen it before
            if (!hash.empty()) {
                auto it = filesSeenAlready.find(hash);
                if (it != filesSeenAlready.end()) {
                    // compare with its last edited time
                    if (st.st_mtime > it->second.lastModified_) {
                        // current file is the dupe!
                        duplicates.emplace(duplicates.end(), currentPath, it->second.path_);
                    }
                    else {
                        // old file is the dupe!
                        duplicates.emplace(duplicates.end(), it->second.path_, currentPath);

                        // but also update filesSeenAlready to have the new file's info
                        it->second.path_ = currentPath;
                        it->second.lastModified_ = st.st_mtime;
                    }
                }
                // if it's a new file, throw it in filesSeenAlready
                // and record its path and last edited time,
                // so we can tell later if it's a dupe
                else {
                    filesSeenAlready.insert(make_pair(hash, FileInfo(currentPath, st.st_mtime)));
                }
            }
        }
    }
```

```
    return duplicates;
}
```

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 "pre-processing" 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.

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 method 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.

Want more coding interview help?

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