CS205 - LAB3

This lab is a group project. You should work by groups of 2 or 3 people (not more than 3). There will be one submission and one grade for all members in the group. The submitted file(s) must have in their names all the student ids from group members, separated by underscores. Repeat student id and your names in comments in the code for safety.

Presentation of the Program

The goal of this lab is to write a tool that may or may not exist already but should be quite useful. On a computer, after a while you have a lot of files that are duplicated (in fact you may have more than two identical copies of a file). Sometimes they may be copied with the same name to different directories (for instance you may at one point copy all the files of a project to a `BACKUP` directory - not all of them will be changed later), files may be copied under different names in the same directory (`my_prog.c` and `my_prog_old.c` or `my_prog.sav` - and the new version was never modified), or you can have another copy under a different name in another director (`IMG1234567890.JPG` copied to `me_and_my_family.jpg` elsewhere). After a while, you end up with a lot of files that you might want to destroy, archive to a USB device or perhaps simply move to a single backup directory. The purpose of this program is to take a directory name as argument, to search this directory and its subdirectories, and when two identical files are found to display the two names, including directories, of identical files. The tool will only display the pairs of file names, the program user will then use the information to decide what to do. That means that as we discover files in directories we must store some information about them in a data structure (you'll decide on what you need...) and compare every file to all the files discovered before the current one.

It's of course the content that says whether two files are identical. However, comparing every file byte-by-byte with all other ones could take hours. We must find ways of going faster, and one way might be to do it in several steps.

When you read directories (reminder: functions `opendir()`, `readdir()` and `closedir()`) one of the pieces of information available is the file size. Two files with different file sizes cannot be equal, so you should only consider the other files you already know that have exactly the same size. But the size isn't enough. You can choose of comparing either the full files byte by byte, or try to find clever ways to go faster. You can for instance compare byte-by-byte small files (up to you to decide what "small" is), or a fixed-size block taken

somewhere, say in the middle for bigger files (reminder: function `fseek()` allows to move to any position in a file, and you can read a sample fixed number of bytes from there), or even decide to check a number of blocks of bytes at random positions (reminder: function `random()`) in the two files, and if everything matches use this statistical evidence to decide that both files must be identical. The choice of how many blocks to test and of the size of blocks is entirely up to you. You can experiment with different combinations. You can have other ideas and other strategies to obtain a satisfying result in a reasonable amount of time.

Out of 100 points, 10 will be assigned for speed (the fastest program will get 10 points, the slowest one zero). You can measure the speed of a command by typing `time` followed by the name of the command (it displays three lines when the command has finished, the first one is the important one). However, speed is great but accuracy is more important; if your program says that two files are identical when they aren't, you'll lose more points than if it's very slow. We won't test with "trick" files that just differ by one byte somewhere, we'll use regular files in regular directories on our computers. If it works well on your computer, it should be OK on our computers.

If you discover that a file is identical to another one you already know, there is of course no need to "remember" it - once you have displayed its name on the screen with the name of the identical file, you can forget about it and move on.

If there are 3 files (same size and same content) that are discovered in this order:

```
some_dir/alog
some_dir/local/b.log
some_otherdir/local/lab3/c.log
```

The tool will say when it discovers b.log that

`some_dir/alog` is identical to `some_dir/local/b.log`

then, when it discovers c.log, that

`some_dir/alog` is identical to `some_otherdir/local/lab3/c.log`

There is no need to repeat that `/user/local/lab3/c.log` and `/user/local/b.log` are identical, it can be inferred.

Code Organization

As this is a group project, we expect several .c files (at least two, but you can have more) that must be combined into a single program. There should be at least one header file containing prototypes for your functions, typedefs and constants (`#define`) if you need any. You should also provide a makefile.

Interface

We shall apply industrial methods to evaluate your lab project and run your program through a series of tests to see how it behaves and compare its output to a known correct result. You must therefore exactly follow the specifications provided.
Your program must take as command-line argument the name of a directory to search for duplicates; if no arguments are provided, it's up to you to decide whether to abort the program or run it with a "reasonable" default value. It should write the names of pairs of identical files to stdout; they must be separated by a tab (character '\t').

The names to write are path names relative to the start of the search.

Example:

If the program is called find_dup, to find duplicate files under directory `/home/users/dev` you should type

`./find_dup  /home/users/dev`

Now, suppose that in this directory the two following files are identical:

`/home/users/dev/proj1/util/strutil.c`

and

`/home/users/dev/proj2/src/strfunc.c`

Then the command above should display

`proj1/util/strutil.c  proj2/src/strfunc.c`

(**`/home/users/dev/`** is omitted as we know it's the starting point). What you have between the filenames is a tab (or tabulation), represented by \t in a C `printf` format. The first file that you have met when searching the directory should come first. No text other than filenames should go to stdout. If you want to display other messages, you should write them to stderr so that they don't interfere if one redirects the output to a file.

You may or may not want to use `getopt()` as in lab2 to implement optional flags; using options is not required. There is something though I personally find useful, which is an undocumented `-d` flag for a debug mode that displays some information about what the program is doing - very helpful in a development phase. Many commands also have a `-v` flag that stands for *verbose* (which is to writing what *talkative* is to speaking - a text is verbose when you use far more words than required to convey the meaning). With this flag, a program displays more information to the user about what is going on; you are free to implement anything you find useful, or to have no flags (`-v` and `-d` are suggestions, not requirements).

As this program will probably often run for several minutes, it would be a good idea to display from time to time some information (on stderr, so that it doesn't interfere with the regular output if you decide to redirect stdout to a file), such as for instance the number of files and/or the total size of what has been compared so far, or perhaps names of directories searched - once again, up to you. What is important is that users see some progress and don't believe that the program is stuck in an infinite loop.

Try to see yourselves are users of the program, and to make it a nice and useful utility.