DOCUMENTATION

- **Design**
    - We took our original ***sort*** and threw it in a function, with parameters of:
        - File to sort, column to sort on, file name, and output directory
        - The contents of this function are similar to the previous project
            - Takes a file, parses through the first line using fgets.
            - It takes this line and separates it by commas (if any) and stores each token into a linked list, while counting the number of headers
            - Once headers are parsed, then the function moves on to parse the csv line by line. It will now count the number of tokens in each row to make sure its = to the number of headers. If not, the function returns with error, because it is an invalid csv
        - The addition to this function is the writing of a new file to output the sorted csv file.
            - First, the name is created by stripping the original file name of the ".csv" and then appends "-sorted-" to it, we used snprintf to format stuff to a variable
            - Then it appends the column it sorted by to the name, and then finally ".csv"
            - The file is written to using fprintf.
    - The ***dirwalk*** function is a new function that recursively traverses a directory
        - It utilizes DIR and dirent to walk through a directory. It also used S_ISDIR and S_ISREG macros to determine if its a directory or not. If its a directory, the program forks, and then recurses to handle that directory. The parent waits for the child to return before continuing
        - If its a file, the program forks and calls the sort function on the file. However, it first tests that the file exists.
        - It also checks if a file is already sorted, by looking for the substring "-sorted-" in the file name
        - After testing if its a child or parent, the parent documents the pid and then waits for the child to return.
    - We also have a ***numproc*** function that counts the number of expected processes. At first thought to be inefficient, it is only scanning the directory one more time, so essentially it doesn't do much to the runtime. The structure of ***numProc*** is similar to that of ***dirwalk*** except it only counts the number of processes.
    - We made this sorter **generic**. It is generic in the sense that it doesn't expect any specific number of columns - it dynamically allocates space for the columns as they are read in. It also doesn't assume the type of the tokens either. In our mergesort, we check the type of the entry and sort accordingly, whether it be a string, double, or integer. We worked on adding functionality to sorting things other that just string, double or integer.

- **Changes to original sorter**
    - As described above, nothing from the original sorter was taken out. It was basically thrown into a function with some extra parameters, and wrote to a file instead of stdout in the end
    - Also fixed an error where there was not enough space being allocated to certain strings - reason for low score on last assignment
    - We also had a bug where it wasn't correctly stripping the new line character from the end of each line. There was some invisible character that was messing with the last token of each line
- **Assumptions**
    - Literally no assumptions - everything is checked and exited upon failure
    - Flags
        - If less than 3 arguments, exit with error
        - If -c flag not specified, exit with error
        - If 5 arguments (only 2 flags)
            - Checks for either -d or -o flags, if they are incorrect then exit with error
            - Sets internal flags for hasDir and hasOut, to be used later
        - If 7 arguments (all flags)
            - Checks that 2nd flag is -d and 3rd is -o, otherwise exit with error
            - Sets hasDir and hasOut only if both directories specified
        - If incorrect flag specified, exit with error
        - Checks that the directories exist
            - If either -o or -d do not exist, then program ends, exits with error
- **Difficulties you had and testing procedure**
    - The main difficulty we experienced in this project is keeping track of the PID's. We concluded at the only workaround we could think of, and that's writing all the PID's to the file, and then reading from the file. We could not come up with any other way to keep track of all the child PID's and passing all that back to the parent. Though there is probably a simpler way to do this, we came up with this method and it works with our test cases, assuming that we are allowed to create a file buffer for this.
    - At first we also had difficulty in figuring out how to recurse properly through the directory, and figured out what our while loop wasn't structured correctly. After some changes we got our recursive directory traversal to work. This was the basis of our project and everything stemmed from there
    - We also had some fork bombs go off because we weren't waiting in the right place. Some debugging also resolved the issue
- **Include any test CSV files you used in your documentation**
    - Test directory structure: (**folder**, *file*)

      asst1
        - *Small.csv*
        - *Somefile.csv*

        - *sc.csv*
        - *t3.csv*
        - **outdir**
        - **Test1**
            - *Stuff1.csv*
            - *Stuff2.csv*
        - **testDir**
            - *Smallerdata.csv*
            - *Smallerdata24.csv*
            - *Subtest.csv*
            - *Test.csv*
    - This sample structure should yield 13 child processes + 1 parent process
- **Do not neglect your header file. Describe the contents of it and why you needed them.**
    - In our header file, we have defined two structs, CSVrecord, and hNode.
    - CSVrecord is a node that keeps one record of a table
        - It stores the sortValue, an array of all the tokens in the record, the number of columns, as well as a reference to the next node
    - hNode is a node that stores header data, so you can dynamically allocate an internal representation of the CSV.
        - It contains the data (column) as well as a reference to the next node
    - The header includes all the prototypes of the functions used in mergesort as well as in the main sorter file.
- **How to use code**
    - Format for parameters: -c <colName> -d <directoryName> -o <outputDirectoryName>
        - In that order
    - Note: -d <directoryName> and -o <outputDirectoryName> are optional\n");
    - Input must require at least the column and other flags are optional.
    - Code calls mergesort from another file, mergesort.c