Problem 1: Reading/Writing Files in Binary and Text Format (30 points)

Implement **TWO** C programs.
The first program receives integers from user inputs, and saves a copy of the integers into a text file and the other copy into a binary file.
The second program receives indexes from user input, finds the corresponding integers from the text file and binary file, and prints out the integers.

For example, when you run the first program and provide the following integers, the program will generate two files. (If the files already exist, the program needs to truncate them before writing new data into the files. If the files did not exist, create them.) One file is a text file (e.g., you can name it *numbers.txt*). You may choose to save each number on each line or to use a special character to separate the integers. You can verify that it is really a text file by printing out the first part (e.g., *head -c 100 ./numbers.txt*). The other file is a binary file (e.g., you can name it *numbers.bin*), with each integer taking 4 bytes.

You can hard-code the file names in your program.

```
Input data 1 : 5
Input data 2 : 126
Input data 3 : 37
…
Input data 100 : 3794
Input data 101 : 794
…
Input data 200 : 74
Input data 201 : 94
```

User can press ctrl-d to stop the input (scanf() returns EOF).
Then, if you run the second program and provide the following indexes, the program will print out the corresponding integers saved in the two files.

```
Input index: 100
Integer saved in text file: 3794
Integer saved in binary file: 3794

Input index: 200
Integer saved in text file: 74
Integer saved in binary file: 74

Input index: 2
Integer saved in text file: 126
Integer saved in binary file: 126
```

I know the integers that the second program reads from the two files are the same. The program is for practice. For each index, your program MUST read a separate integer from each file, and CANNOT reuse/copy the integer read from one file as if it was from the other file. But, you are encouraged to this trait to verify that your program is correct --- if the two numbers are different, there must be some bugs.
The second program needs to assume that the files may be too large to be read into memory completely. (This often happens! Consider bigdata processing and scientific computing, which often generate terabytes

of data). Thus, it's not realistic to send up an array and read all the integers from a file into the array. Instead, it needs to use `fseek` and/or `rewind` to change file offsets based on index and then read into the integers.

Submission Instructions

Submit individual files following the instructions below. DO NOT SUBMIT A ZIP FILE.

- **Your C programs**: Name your programs *problem1_write.c* and *problem1_read.c*.
- **1~2 screenshots in jpg format for each program showing that your program really works**: Name your screenshots using the pattern problem1_index.jpg. Index is 1, 2, or, 3…
- **1 screenshot in jpg format to show** that one of the files is in text format (`head -c 100 ./numbers.txt`).

Testing

Test your program manually first. The numbers printed out by the second program should match those provided to the first program. Also, check the sizes of the files (numbers.txt and numbers.bin). Which uses more space?
To fully test your program with more integers, modify and use the following scripts. $1 of the script is the number of the integer values. $2 is the number of indexes. You need to inspect the output of the second program.
Take screenshots when you test your program manually, such that we can see the numbers provided to the program and the output of the program.

```
#!/bin/bash
count=$1
echo "================== Writing files ======================"
for (( i=0; i<${count}; i++ ));
do
  echo $RANDOM
done | PATH_NAME_OF_YOUR_1st_PROGRAM
ls -l ./numbers.txt ./numbers.bin
echo "================== Reading files ======================"
for (( i=0; i<$2; i++ ));
do
  echo $(($RANDOM % ${count}))
done | PATH_NAME_OF_YOUR_2st_PROGRAM
```

Problem 2: Traverse a Directory (70 points)

Write **TWO** C programs. The programs print out the pathnames of all the files in a directory, including all the files under all the sub-directories, sub-sub-directories, etc. (i.e., sub-directories at all the levels). The pathnames should be printed out in order of file sizes with the pathname of the smallest file on top (i.e., ascending). One use recursion and one does not use recursion.

Submission Instructions

Submit individual files following the instructions below. DO NOT SUBMIT A ZIP FILE.

- **Your C programs**: Name your programs *problem2_resursive.c* and *problem2_nonresursive.c*.

- **1~2 screenshots in jpg format for each program showing that your program really works**: Name your screenshots using the pattern problem2_recursive_index.jpg or problem2_nonrecursive_index.jpg . Index is 1, 2, 3…
- **1 screenshot in jpg format to show the difference between the two programs**. On the screenshot, **highlight or circle the code that does the recursion**. You may use `diff` to get the differences and then take a screenshot. Name your screenshot problem2_diff.jpg

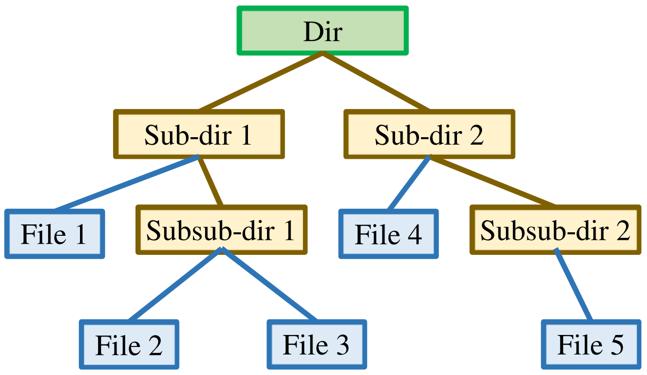
```
diff   problem2_resursive.c   problem2_nonresursive.c
```

Objectives

- To learn how to traverse a directory
- To learn how to check file status
- To gain more experience on using pointers and linked lists
- To gain more experience on sorting a linked list.
- To learn how to write recursive functions
- To learn how to test a program.

Requirements and Instructions

- Your program should take one argument, which is the pathname of the directory. (Important! If you don't follow strictly, we may not be able to run your program correctly, and you may get lower grades.) For example, the following command sorts the files under directory `/usr/share/man`:
  `./your_program  /usr/share/man`
- All files directly and indirectly under the directory should be considered. This means that your program needs to traverse the sub-directories, sub-sub-directories, sub-sub-sub-directories, etc, and check the files there. For example, when your program is run against directory *`Dir`* on the right, all 5 files (*`File 1`* ~ *`File 5`*) should be included.



- For an implementation without using recursion, use a linked list to organize the sub-directories that have not been scanned. Refer to the slides on directory traversal in the bash part. The slides on state space search can also give you some ideas.
- Assume the longest pathname does not exceed 1024 single-byte characters.
- For the files with the same size, there is no requirement on how their pathnames should be sorted.
- Print out the results on the screen in text format, one line for each file with file size followed by a tabular symbol ('\t') and the file pathname. Programs using different formats in outputs will fail the tests and lead to lower grades.

The sample output when the program is run with the command below against directory /usr/share/man/ is shown as follows:

```
$./your_program  /usr/share/man
30      /usr/share/man/man1/cpp-4.8.1.gz
30      /usr/share/man/man1/cpp-5.1.gz
30      /usr/share/man/man1/gcc-4.8.1.gz
30      /usr/share/man/man1/gcc-5.1.gz
35      /usr/share/man/man3/XauDisposeAuth.3.gz
35      /usr/share/man/man3/XauFileName.3.gz
35      /usr/share/man/man3/XauGetAuthByAddr.3.gz
…
86618   /usr/share/man/man1/bash.1.gz
104755  /usr/share/man/man5/smb.conf.5.gz
287622  /usr/share/man/man1/x86_64-linux-gnu-gcc-6.1.gz
303306  /usr/share/man/man1/x86_64-linux-gnu-g++-7.1.gz
303306  /usr/share/man/man1/x86_64-linux-gnu-gcc-7.1.gz
```

Testing

- Check whether the program can print out correct result when it is run against a directory containing a few files with different sizes

  Step 1. Create a directory and a few files in the directory
  ```
  mkdir ./test1
  dd if=/dev/zero of=./test1/file1 bs=10 count=1
  dd if=/dev/zero of=./test1/file2 bs=100 count=1
  dd if=/dev/zero of=./test1/file3 bs=1000 count=1
  ```
  Step 2. Execute the program against `./test1`:
  ```
  ./your_program   ./test1
  ```
  Step 3. Check the output of the program, which should look like
  ```
  10      ./test1/file1
  100     ./test1/file2
  1000    ./test1/file3
  ```

- Check whether the program can print out correct result when it is run against a directory containing sub-directories and regular files in sub-directories

  Step 1. Create a directory and a few files in the directory
  ```
  mkdir ./test2
  dd if=/dev/zero of=./test2/file1 bs=10 count=1
  dd if=/dev/zero of=./test2/file2 bs=100 count=1
  mkdir ./test2/dir1
  dd if=/dev/zero of=./test2/dir1/file3 bs=20 count=1
  dd if=/dev/zero of=./test2/dir1/file4 bs=200 count=1
  mkdir ./test2/dir2
  dd if=/dev/zero of=./test2/dir2/file5 bs=30 count=1
  ```

```
dd if=/dev/zero of=./test2/dir2/file6 bs=300 count=1
mkdir ./test2/dir3
dd if=/dev/zero of=./test2/dir3/file7 bs=40 count=1
dd if=/dev/zero of=./test2/dir3/file8 bs=400 count=1
```

Step 2. Execute the program against `./test2`:

```
./your_program   ./test2
```

Step 3. Check the output of the program, which should look like

```
10      ./test1/file1
20      ./test1/dir1/file3
30      ./test1/dir2/file5
40      ./test1/dir3/file7
100     ./test1/file2
200     ./test1/dir1/file4
300     ./test1/dir2/file6
400     ./test1/dir3/file8
```

- Check whether the program can print out correct result when it is run against a large directory, such as /usr/share/man.

Step 1. Run the program and redirect the result into a file *`output_test.txt`*

```
./your_program /usr/share/man > output_test.txt
```

Step 2. Check whether the sizes are correctly sorted in the output. If they are correctly sorted, you will not see *`diff`* print out any text.

```
cut -f 1 output_test.txt > sizes_test.txt
sort -s -n sizes_test.txt > sizes_test_sorted.txt
diff sizes_test.txt sizes_test_sorted.txt
```

Step 3. Check whether the output includes the pathnames of all the files --- the output should have 9000+ lines.