CSE 3100 Systems Programming	Fall 2021
Homework #4	Due:10/11/2021

Do not forget to submit your work before the deadline. A Makefile is provided. Run make sub to create a zip file and submit it. Only Makefile and *.c are needed.

Exercise 1. (50 points) mygrep. grep is a very useful tool that searches for a string pattern in files. In this exercise, we implement mygrep, a simplified version of grep.

mygrep takes two arguments, argv[1] and argv[2], from the command line. argv[1] is a string pattern and argv[2] is a filename. The program prints all the lines in the file specified by argv[2] that have argv[1] as a substring. mygrep does not support regular expressions.

The main task of this exercise is to complete the print_matched_lines() function in the starter code. It has the following prototype.

```
int print_matched_lines(const char *s, const char *filename);
```

The function opens the file specified by filename, reads lines from the file (using fgets()), and for each line, prints the line to the standard output if (and only if) they contain the string s. The function calls strstr() to determine if a string appears in another string. We can assume the lines have at most 1022 ASCII characters before the new line.

mygrep should behave the same as grep if the pattern does not contain special characters used in regular expressions. If there are punctuation marks, the first argument (the pattern) needs to be enclosed by single quotation marks, as in the second command in the sample sessions listed below.

```
$ ./mygrep mygrep Makefile
TARGETS=mygrep data-extract data-gen

$ ./mygrep 'main(' mygrep.c
int main(int argc, char **argv)
```

Exercise 2. (50 points) data-extract. Many files on our computers, such as executables and many music and video files, are binary files (in contrast to text files). The bytes in these files must be interpreted in ways that depend on the file format. In this exercise, we write a program data-extract to extract integers from a file and save them to an output file.

The format of the binary files in this exercise is very simple. The file stores n integers (of type int). Each integer consists of 4 bytes (in the native format on the Mimir platform). We number the integers in a file from 0 to n-1. So the first 4 bytes (bytes 0 to 3) are integer 0, the next 4 bytes (bytes 4 to 7) are integer 1, and so on. The size of the file is always a multiple of 4.

A program data-gen is provided to generate binary files we can work on. It takes three arguments from the command line:seed, n, and filename. The program writes n pseudo-random numbers determined by seed to file filename. For example, the following command writes 100 integers to file a.dat. The 100 integers are random numbers determined by seed 3100. Please study the source code data-gen.c to learn how integers are written to the output file.

```
$ ./data-gen 3100 100 a.dat
```

The task of this exercise is to complete the code in data-extract.c. The program data-extract takes at least three command line arguments (as shown below), and extract integers from file input-file and saves the extracted integers to file output-file.

```
$./data-extract input-file output-file range [range ..]
```

The integers to be extracted from input-file are specified by ranges. A range can be two numbers separated by '-', like start-end, or a single number. start-end specifies integers starting from integer start to integer end. If a range is a single number pos, it specifies integer pos, which is the same as pos-pos.

For example, the following command extract 10 integers from a.dat to t1.dat. The integers written to t1.dat are integers at indices 0, 1, 15, 16, 17, 18, 19, 1, 2, and 3 from a.dat. Note that an integer in a.dat can be extracted multiple times.

```
$ ./data-extract a.dat t1.dat 0 1 15-19 1 2 3
```

If there are 100 integers in a.dat, the following command moves the integer at the beginning to the end.

```
$ ./data-extract a.dat t2.dat 1-99 0
```

In the starter code, main() parses the command line arguments. The remaining tasks are as follows.

1. Add necessary statements in main() to open and close files properly (before and after the for loop). If any file operation fails, call my_error() to report error and exit. See comments in the starter code.

2. Complete the copy_integers() function, which has the following prototype.

   ```
   int copy_integers(FILE *outfp, FILE *infp, int start, int end);
   ```

   outfp is the outpot stream and infp is the input stream. The function copies integers start to end from infp to outfp.

   The function returns -1 on errors and 0 on success. Do not call my_error() in this function.

Read the comments in the starter code for more implementation details.
Some output files (*.dat) are provided. They are generated with the following commands.

```
$ ./data-gen 3100 100 a.dat
$ ./data-extract a.dat t1.dat 0 1 15-19 1 2 3
$ ./data-extract a.dat t2.dat 1-99 0
$ ./data-extract a.dat t3.dat 10 20 30 40 50 5 15 25 35 45
$ ./data-extract a.dat t4.dat 30-49 50-59 80-99 10-40
```

Some tools can help us to compare and examine binary files. Below are some examples. Lines that starts with '#' are comments explaining the following commands. See the manual pages for more options of cmp, xxd, and diff commands.

```
# compare a.dat and b.dat
$ cmp -b a.dat b.dat

# compare a.dat and b.dat, and show detailed information
$ cmp -bl a.dat b.dat

# list groups of 4 bytes file a.dat in hexadecimal
$ xxd -g4 a.dat

# compare the hex dump of a.dat and b.dat
$ diff <(xxd a.dat) <(xxd b.dat)
```