CS251 - Project 4: Searching and Sorting

Out: February 29, 2016 @ 9:00 am

Due: March 28, 2016 @ 9:00 am

Overview:

Usually when you are sorting a list of objects, those objects are held in memory and can be easily manipulated. However this is not always the case. It is possible that you may want to sort a list that is larger than the amount of memory you have available. These situations are fairly common in larger databases. To sort in in this situation, you have to sort the information in stages, making sure that you never exceed your available memory at any point in time.

Part 1: External Sorting

In part 1, you will be given a single large file to sort in **ascending order** (lexicographically)**.** For the purposes of this project the list will contain a list of strings to sort alphabetically, but the algorithm itself can be easily changed to work with any data type you wish. This file is a text file with each entry in one line.

Sorting in this project will be done in several stages.

1. Read in the large file in chunks according to a specified number of lines
2. Sort the chunk using an nlogn sorting algorithm (e.g., quicksort, mergesort, heapsort) and output it as a temporary file (note that quicksort is not an nlogn sorting algorithm but acts like one in practice).
3. Open all temporary files and perform a k-wise merge (defined later in this document) to an output file

Example:

For this example, let's imagine that our large file has 12 entries

```
abbc
aca
aab
aaa
aaac
acb
bac
acccccccc
aba
abb
bab
bbbaa
```

Assume we are told to sort in chunks of 3. We read in the file 3 lines at a time, sort the 3 items, then write the sorted list to a file. This is repeated until the file is empty.

aab abbc aca	aaa aaac acb	aba acccccccc bac	abb bbbaa bab

Finally, we perform a 4-way merge. The method is very similar to the merging technique you have seen in class. The only difference is that instead of writing the smallest item in two lists to the sorted larger list, you pick the smallest of k items to write to the new list. The pseudocode for this is below:

```
kMerge(int k, followed by k lists):
    Create k pointers pointing to the start of the k lists
    Open an output file
    While not done:
        Write the smallest of the items pointed to to the output file
        increment the pointer that was just used
```

Input/output format:

Your program will be called with the ***four additional*** command line arguments

- The number 1 indicating you should run the code for part 1.
- The name of the file to sort.
- The name of the file where the sorted list should be stored.
- The number of lines per chunk.
- The directory to save the chunks in

Note: You can use argv[2] argv[3] to access the second and third argument respectively.

The argv[4] is the number of lines per chunk.

Output of your program will be a single file which name is given as third argument.

For example,

```
./program 1 testdata/input-test1.txt  output-test1.txt 3 testdata/chunks/
```

Chunks should be named in the following format (exactly five digits):

chunks/chunk<chunk number>

READ NEXT PAGE for example of naming chunk

The four files created in this example would be

- chunks/chunk00001
- chunks/chunk00002
- chunks/chunk00003
- chunks/chunk00004

Note: You may assume the output directory for final sorted file and chunks will exist.

Let’s run through that with our given example. Pointers will be represented as underlined strings:

Stage 0:

Temporary Files

aab	aaa	aba	abb
abbc	aaac	acccccccc	bbbaa
aca	acb	bac	bab

Output

Stage 1: Select the first element and write to output. Increment that pointer

Temporary Files

aab	aaa	aba	abb
abbc	aaac	acccccccc	bbbaa
aca	acb	bac	bab

Output

aaa

Stage 2: Repeat a second time.

Temporary Files

aab	aaa	aba	abb
abbc	aaac	acccccccc	bbbaa
aca	acb	bac	bab

Output

```
aaa
aab
```

Continue n times, until all pointers are exhausted:

Stage n: The sort is complete

Temporary Files

aab abbc aca	aaa aaac acb	aba acccccccc bac	abb bbbaa bab

Final Output

```
aaa
aab
aaac
aba
abb
abbc
aca
acb
acccccccc
bbbaa
bab
bac
```

Part 2: Searching in a large file

In the next part, you will implement the ability to search for items in the large sorted list. Searching in a large file is a bit trickier than searching in memory. Use binary search, but instead of searching in memory, you will search in a file. To accomplish this, you will need to move the file pointer around until you find the desired item. The command to do this with a file pointer in C++ is fseek(). You can read its documentation at http://www.cplusplus.com/reference/cstdio/fseek/.

A second tricky part of this problem is that the file pointer does not move according to lines. It instead jumps to a character offset in the file. To get around this problem, when you jump to a position, you will need to look a few characters ahead or behind to find the next newline character. Take special care at the beginning of the file, the end of the file, and the endpoints of the search segment. You may assume the last line ends with a new line('\n').

Input/Output Format (Part 2):

Your program will be called with 2 as the first argument and the path to the file as the second argument and ***additional*** command line arguments for each search term.

Note: You can get the length of input argument with argc

For example to search in the file produced in part 1,

```
./program 2 output-test1.txt aaa aba car
```

Output of your program will be n lines containing either **found** or **missing**

Sample Output:

```
found
found
missing
```

Note: there should be '\n' after each search answer (even the last one; no special case for it). (in addition to the correctly stored out1.txt. Please see the provided testcases on Piazza for this file)

Things to note:

- There is no guarantee that the input file is an even multiple of the chunk size. Your last chunk may be a bit smaller.
- We will use test cases where you will be unable to load the entire list into memory at the same time. However you will have plenty of memory to handle one chunk at a time and deal with the k-wise merge.
- Your search algorithm should run in log(n) time.
- Your memory and time will be limited in the testcases. Ensure that your program meets efficiency requirements.
- Be sure to free or delete any memory you are finished using.

Programming Environment and Grading

Assignments will be tested in a Linux environment. You will be able to work on the assignments using the Linux workstations in HAAS and LAWSON (use your username and password). Compilation will be done using g++ and makefiles. You must submit all the source code as well as the Makefile that compiles your provided source code into an executable named program.

Your project must compile using the standard g++ compiler (v 4.9.2) on data.cs.purdue.edu. For convenience, you are provided with a template Makefile and C++ source file. You are allowed to modify such file at your convenience as long as it follows the I/O specification. Note some latest features from C++14 are not available in g++ 4.9.

The grading process consists of:

1. Compiling and building your program using your supplied makefile.
2. The name of produced executable program must be program (must be lowercase)
3. Running your program automatically with several test input files we have pre-made according to the strict input file format of the project.
4. Inspecting your output files
5. Inspecting your source code.

Important:

1. **If your program does not compile, your maximum grade will be 50% of the grade (e.g., 5 out of 10) -- no exceptions.**
2. **Plagiarism and any other form of academic dishonesty will be graded with 0 points as definitive score for the project and will be reported to the corresponding office.**

Submit Instructions

The project must be turned in by the due date and time using the turnin command. **Follow these directions closely, incorrect submission format can result in a lowered grade:**

1. Login to data.cs.purdue.edu (you can use the labs or a ssh remote connection).
2. Create a directory named with your username and copy your solution (makefile, all your source code files including headers and any other required additional file) there.
3. Go to the upper level directory and execute the following command:
   ```
   turnin -c cs251 -p project4 your_username
   ```
   (Important: previous submissions are overwritten with the new ones. Your last submission will be the official and therefore graded).
4. Verify what you have turned in by typing turnin -v -c cs251 -p project4
   (Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one). If you submit the wrong file you will not receive credit.
5. If you want to use a late day for this project, email your PSO TA telling them how many you want to use.