CSE 6010
Assignment 4
Parking Data

Initial Submission Due Date: 11:59pm on Thursday, October 24
Final Submission Due Date: 11:59pm on Thursday, October 31
Peer review assignment due Thursday, November 7 will be posted separately
Submit Makefile and code as described herein to Gradescope through Canvas
48-hour grace period applies to all deadlines

In this assignment, you will work with a data file containing information about cars being parked in several parking lots over a 24-hour period from 12:00am through 11:59pm. The car's parking permit information is recorded upon entry and exit along with the times of entry and exit and the parking lot used. You will sort the data using different key data to extract three types of information about the usage of the parking lots.

- Duplicate permits: Identify permits that were used more than once during the 24-hour period. Output will consist of all parking records where the parking permit was used more than once during the 24-hour period.
- Median durations: For each parking lot, the median duration cars were parked in that lot during the 24-hour period. Output will be M lines, where M is the number of garages, with each line giving the parking lot index followed by the median parking duration for that lot, in minutes.
- Maximum occupancy: The largest number of cars parked across the parking system (all lots) during the 24-hour period. Output will be a single number.

For this assignment, you will not be given any starting files beyond example Makefiles and are free to develop your code following the requirements and guidelines listed below.

To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure.

Specifications:

- Data will be input as a file of parking data including the following items.
  - The first line includes the number of records in the file.
  - Each following line includes one record that records the entry time, the exit time, the index of the parking lot, and the parking permit number.
    - Sample record line:
      `13:11 22:12 2 290`
    - Times are written as XX:YY, where XX represents the hours between 0 and 23 and YY represents minutes from 0 to 59. Thus, afternoon times will be recorded as hours from 12 to 23. There may be leading zeros or not, but that will not affect reading the input because you should read both the hours and minutes as integers. All cars enter no earlier than 0:00 and all cars exit no later than 23:59.

        - The parking lot index will be an integer from 0 to M-1. You will need to determine the number of different parking lots M directly from the parking records.
        - The parking permit number will be an integer from 0 to P-1, although not all integers between 0 and P-1 (inclusive) may be present in the record. You will not need to know what the maximum integer will be.
    - You will not be required to validate the input; you may assume it is in a proper format.
- Functionality will be specified through two command-line arguments.
    - First command-line argument: name of the input file.
    - Second command-line argument: one of the following three characters:
        - p
          (lower-case p without quotes): the program will print a list of records with the same permit number. Any time a permit number is used more than once, regardless of parking lot of times, output each record for that permit number. The information could be used to check how shared carpool permits were used or to analyze potential permit sharing.
        - d
          (lower-case d without quotes): the program will print the median duration **in minutes** each car was parked, according to the parking lot. For each parking lot, the median duration cars were parked in that lot will be calculated and output.
        - o
          (lower-case o without quotes): the program will print the maximum occupancy across all parking lots. This information indicates how many parking spaces across all of the lots were needed to accommodate the time of highest parking demand.
    - More information about what to do for these cases is given below.
    - We will not test cases where the command-line input is not valid (e.g., we will only test cases with names of files that exist and with one of the three arguments listed above). Your program may handle such cases in any way you see fit, including assuming such cases will not occur.
- Data structures: You should create a struct to represent a parking record. Dynamically allocate an array of such structs to hold the parking records, using the number of records contained in the input file and specified on the first line of that file.
- Each of the three types of functions the program should perform must be accomplished using sorting. Sorting requirements:
    - You must use merge sort to perform all sorting tasks, and you must use sorting to perform all tasks. Use of a different sorting algorithm will not receive credit. You may use either a recursive or non-recursive approach. Feel free to follow the approach outlined in class.
    - Although you will need to sort on different keys in the parking record, you can use the same merge sort functions by adding an additional argument that is used to indicate which key you should select. Then, in the Merge() function, you will specify what to compare depending on the key selected. In other

words, the statement in the notes `if L[i] <= R [j]` will need to be turned into a conditional statement where the comparison depends on the key specified (as you will need that information to know which struct member to use in the comparison).

- One of the reasons merge sort is required is to ensure that (1) the sorting will be efficient, in guaranteed O(n log n) time and (2) the algorithm is ***stable***, such that sorting a second time on a different key will essentially accomplish a sort with the first sorting key as a secondary key. For example, if we first sort on parking permit number and then sort on the parking lot index, the result will group all records with the same parking lot index together, but within those parking lot groupings, the records will be sorted by permit number. Such behavior will be needed to perform some of the tasks in this assignment.

- Detailed specifications for the three types of calculations/output:
  - List of records with the same permit number.
    - Identify records with the same permit number by sorting on permit number and outputting any cases with the same numbers. You can ignore all other fields when sorting.
    - You should output any record whose permit number occurs in another record only once, regardless of how many times that permit number occurs.
    - Print each such record to the screen in the same order it appeared in the input file (in other words, do not perform any sorting other than on the permit number). The record output should list the fields in the following order: permit number, lot number, entry time, exit time. Example output (note that the permit number may be used in the same parking lot or in different parking lots, depending on the records):

      ```
      290 2 13:11 22:12
      290 2 12:48 22:39
      ```
  - Median duration each car was parked in each lot.
    - Calculate and store the duration each car was parked (by properly subtracting the entry time from the exit time) <u>in minutes</u>.
    - Then sort on the duration, followed by a second sort on the parking lot index.
    - Finally, calculate the median duration for each parking lot. The median for a group of values is defined as the value separating the lower half of the values from the upper half of the values. If the number of values is odd, say 2j+1 values indexed from 0 to 2j, the median would be the value at index j in a sorted list. If the number of values is even, say 2j values indexed from 0 to 2j-1, the median would be the average of the values at index j-1 and index j.
    - Output a separate line for each parking lot with the median duration, keeping in mind that the duration may not be an integer value. E.g., the output may look something like this.

      ```
      0 280.500000
      1 139.000000
      2 342.500000
      3 270.000000
      ```

- Maximum occupancy across all parking lots: Calculate the maximum number of cars parked in the system of lots at any time during the day. There are various ways to accomplish this task; one method is described below.
  - Create a duplicate set of records.
  - Sort one set of records based on the entry times and sort the other on exit times.
  - Loop over the two sets of records in a modified "Merge" procedure to process each record in order. Calculate a running sum where entry times count as +1 and exit times as -1. Track the maximum value, which will correspond to the maximum occupancy.
  - If one car arrives at the same time another exits, treat the entering car as arriving first (that is, count so that occupancy will be higher).
  - Output the maximum occupancy as a single integer value. E.g., your output may look like this.
    ```
    21
    ```

Initial submission (worth 1 point): Submit your code to the Initial Submission on Gradescope. Your initial submission will not be graded for correctness or even tested but rather will be graded based on the appearance of a good-faith effort to complete the majority of the assignment. Any time you wish to test your code for correctness, please submit it to the Final Submission on Gradescope.

Final submission (worth 11 points): Submit your code to the Final Submission on Gradescope. You may submit any number of files named in any way you like, but you will be required to submit a Makefile that will be used to compile your program, and the name of your executable must be **parking** (no extension). Do not zip the files or upload a directory; instead, submit your files directly.

Sample output: For the provided sample file parkingdata40.txt, your program should generate the following output.

```
./parking parkingdata40.txt p
290 2 13:11 22:12
290 2 12:48 22:39

./parking parkingdata40.txt d
0 280.500000
1 139.000000
2 342.500000
3 270.000000

./parking parkingdata40.txt o
21
```

Some hints:

- Start early!
- Identify a logical sequence for implementing the desired functionality. Then implement one piece at a time and verify each piece works properly before

proceeding to implement the next. Working through the tests outlined below in sequence may be helpful.

- Because we will not cover MergeSort until the Tuesday before the initial submission is due, you may use a different sorting algorithm (like insertion sort) for your initial submission and replace it for the final submission. If you keep the interface the same, you should be able to swap out one sorting algorithm for another without altering any of the other functionality.
- None of the test cases should take long to run if implemented correctly. If running your code takes more than a few seconds, you can force it to stop with the keyboard Ctrl+C or command C-c (hold control and press the "c" key).

Grading

Program correctness will be assessed using the autograder on Gradescope; valgrind will be used to detect memory errors, including failure to free all allocated memory. Detailed grading specifications are described below. Numbers in parentheses indicate the total points available for a specific component.

1. **Initial submission (1)**: Graded manually. The point is earned for making meaningful progress toward completing the assignment. The program does not need to compile or run correctly at this stage.
2. **Autograder (8)**
    a. **Test 0:**
        i. **Compile the program (0)**: This test is not worth any points, but it will show the output if errors are encountered compiling your program on the autograder. This output should be helpful if your code compiles on your machine but not on the autograder. Fix these errors before moving on to anything else!
        ii. **Test output using provided file (0).** This test is not worth any points, but it will allow you to verify that the output for the provided parkingdata40.txt file is the same on the autograder as it is on your system.
    b. **Test 1: Command-line argument is p (1.5)**: Two tests will be performed with different input files.
    c. **Test 2: Command-line argument is d (2.5)**: Two tests will be performed with different input files.
    d. **Test 3: Command-line argument is o (3)**: Two tests will be performed with different input files.
    e. **Test 4: Testing for memory leaks (1)**: This test comprises three separate tests.
        i. Test 4a (0.25) tests for memory leaks when the second command-line argument is p.
        ii. Test 4b (0.25) tests for memory leaks when the second command-line argument is d.
        iii. Test 4c (0.5) tests for memory leaks when the second command-line argument is o.

3. **Program structure and implementation (1.5)**: Graded manually. Points are earned for implementing the algorithms and data structures correctly as described in the assignment and making reasonable decisions when designing your program.
4. **Program style and documentation (1.5)**: Graded manually. Points are earned for using abstractions to avoid redundant code and making your program readable by using appropriate comments, variable names, whitespace, indentation, etc. Write your code so someone else can easily follow what is going on.

The autograder will run immediately when the program is submitted and should present the results quickly. The manually graded components of the assignment will not be available until after the grades are published. Autograder feedback will only be available with the final submission for the assignment, as the initial submission will not be graded for correctness.

Note that autograder points may be deducted if the instructions of the assignment are not followed, for example if a sorting algorithm other than merge sort is used. Additionally, autograder points will be manually deducted if the program output is manually set to pass the autograder without correctly implementing the solution.

For this assignment, each autograder test case will have a corresponding additional case (worth 0 points) that tests against the provided files and presents the output. These cases are included to help debug any issues in your program.