Middle East Technical University | Department of Computer Engineering

CENG 140

C Programming

Spring 2021
Take Home Exam 2

Ataberk Donmez
ataberk@ceng.metu.edu.tr
Due date: June 14, 2021, Monday, 23:55

Introduction

In this homework, you will be given two tables containing information about

1. Sector times of each lap for every driver in a single race, and
2. Finishing position of drivers for each race in a season

and you are expected to complete some tasks by implementing functions that make computations on the given data. More information about the tasks are provided in the next section. It is imperative that you understand the input data and format before starting with the tasks, so let's dive into the that!

Driver ID	**Lap 0**	**Lap 1**	**...**	**...**	Lap n
0	[22.17, 34.85, 27.46]	[20.19, 32.48, 27.51]	...	...	[23.17, 35.10, 29.02]
1	[25.27, 33.65, 29.50]	[24.07, 36.47, 25.16]	...	...	[19.48, 33.17, 29.56]
...	...	...	...	...	...
...	...	...	...	...	...
m	[20.51, 32.11, 25.44]	[21.99, 41.39, 23.00]	...	...	[21.45, 32.61, 23.48]

Table 1: Format of the first input containing sector times in a race.

First input is the sector times in a single race (Table 1). A sector is one part (or section) of a race track and race track is always divided into 3 sectors. So, when a driver completes all three sectors of the track s/he completes one full lap. Consequently, one lap time equals to sum of the sector times for that lap. For example `Driver0` completes `Lap 0` in $22.17 + 34.85 + 27.46 = 84.47$ seconds.

This input only contains the sector times as floating point numbers with two digits after the decimal point. It does not contain driver ID or lap number, those are inherent. You can think of it as a three dimensional array (and need to store it that way, more on this later!); where first dimension is the drivers, second dimension is laps and the third dimension is sectors. Thus, first element of the array holds sector times of `Driver0` grouped by laps. Similarly, second element of the first element of the array contains the sector times of `Driver0` for `Lap 1`.
Finally, each lap always consists of 3 sectors and all drivers completes the same number of laps. The number of drivers and number of laps in a race may vary and they will be given to you before the table itself. For your convenience, driver IDs and lap numbers start from 0, so that they match their corresponding array indices.

Driver ID	**Race 0**	**Race 1**	**...**	**...**	**Race q**
0	2	5	...	...	9
1	19	7	...	...	28
...	...	...	...	...	...
...	...	...	...	...	...
p	11	13	...	...	22

Table 2: Format of the second input containing finishing positions of drivers for all the races in a season.

The second input contains the finishing positions of drivers for each race in the season, as unsigned integers (Table 2). Both the number of drivers and the number of races in the season (which determines the number of finishing positions) are variables and will be given to you before the table itself. Each driver completes the same number of races in a season. This table can be viewed as a two dimensional array where the first element holds the positions of the driver with the ID `0`. Similar to the first table, driver and race IDs start from 0.

Tasks

1. `float*** create_sector_times(unsigned int* n_drivers, unsigned int* n_laps)`:
   This function will read sector times input from stdin. First two inputs will be the number of drivers and number of laps. These inputs should be stored in `n_drivers` and `n_laps` (which have been passed by reference) respectively. After allocating **exactly** the necessary memory for the 3-dimensional array, sector times should be read from stdin. Each input line after the first one consists of sector times of one lap. Once operations with reading inputs are complete, this function should return the 3-dimensional array containing the sector times. First 3 times are the sectors of the `Lap 0` of `Driver0`, second 3 times are the `Lap 1` of `Driver0` and so on. You should store these values in the correct position in the 3-dimensional array. **Example input:**
   ```
   2 3
   22.17 34.85 27.46
   20.19 32.48 27.51
   23.17 35.10 29.02
   25.27 33.65 29.50
   ```

```
24.07 36.47 25.16
19.48 33.17 29.56
```

2. `unsigned int** create_positions(unsigned int* p_drivers, unsigned int* n_races)`: Similar to `create_sector_times` this function will first read number of drivers and number of races and store them in their respective variables. Next, it will read finishing positions of drivers and return the 2-dimensional array containing these positions (with **exactly** the correct size). Each input line after the first one consists of finishing positions of one driver. Again, all inputs will be taken from stdin. First line holds the positions of `Driver0`, second line holds the positions of `Driver1` and so on. You should store these values in the correct indices in the 2-dimensional array. **Example input:**
```
2 5
2 5 3 11 9
7 24 13 17 17
```

3. `float** calculate_lap_times(float*** sector_times, unsigned int n_drivers, unsigned int n_laps)`:
Given sector times (`sector_times`), number of drivers (`n_drivers`) and number of laps (`n_laps`), this function will calculate lap times for all drivers. To calculate one lap time, you should sum the times of three sectors that makes that lap. Finally, the function will return 2-dimensional array containing the lap times. Ordering of the drivers and laps should be the same as in the `sector_times`.

4. `unsigned int find_fastest_lap(float** lap_times, unsigned int n_drivers, unsigned int n_laps)`:
Given lap times (`lap_times`), number of drivers (`n_drivers`) and number of laps (`n_laps`) this function will find the driver who completed the fastest lap in the race. In order to find the fastest lap you need to check every lap of every driver. This function will return the ID of the driver who has the fastest lap time. ID of a driver corresponds to the driver's index in the first dimension of `lap_times`. If times are the same for two drivers, assume that the driver with the smaller ID finishes before the other.

5. `unsigned int find_driver_fastest_lap(float** sector_times_of_driver, unsigned int n_laps)`:
Given sector times of a **single** driver (`sector_times_of_driver`) and number of laps (`n_laps`), this function will return the lap number of this driver's fastest lap. If the laps have the same time, assume that the lap with the smaller lap number is completed faster.

6. `float* selection_sort(float* arr, unsigned int len, char order)`: Given an array `(arr)`, length of that array `(len)` and an ordering (`order`), this function will return a sorted copy of `arr` (without modifying the original `arr`). `order 'A'` (ascending) or `'D'` (descending) denotes the ordering of the sort operation. Selection sort (ascending ordering) works the following way:

   (a) Assume that the list consists of two parts "sorted" and "unsorted". Initially the sorted part is empty and the unsorted part is the entire list.

   (b) Find the minimum in the unsorted part of the list.

(c) Swap the minimum with the first element of the unsorted part.

(d) Once you complete the swap, one more item is put in the correct place in the sorted part. Starting index of the sorted part is shifted by 1. Unsorted part now starts after this element.

(e) Repeat until the whole list is sorted.

Consider the following example with the array [3 2 7 9 4]:

(a)
- Initially sorted part is empty and the entire list is unsorted.
- We find the minimum in the unsorted part (in the entire array) which is 2.
- We swap 2 with the first element in the unsorted part. Since unsorted part is the entire array, first element in the unsorted part is the first element of the array which is 3.
- After the swap, the array looks like this: [2 3 7 9 4]. Sorted and unsorted parts are as follows (with parenthesis indicating sorted and unsorted parts): [(2) (3 7 9 4)].
- First iteration is complete.

(b)
- Now onto second iteration. We find the minimum in the unsorted part (3 7 9 4) which is 3.
- Swap it with the first element in the unsorted part. Since the first element is itself, swap has no effect. Sorted and unsorted parts are as follows: [(2 3) (7 9 4)]
- Second iteration is complete.

(c)
- We find the minimum in the unsorted part (7 9 4) which is 4.
- Swapping it with the first element in the unsorted part (7) we get: [(2 3 4) (9 7)]
- Third iteration is complete.

(d)
- Again, we find the minimum in the unsorted part (9 7) which is 7.
- We swap it with first element in the unsorted part (9). Sorted and unsorted parts: [(2 3 4 7) (9)]
- Fourth iteration is complete.

(e)
- Starting final iteration.
- Unsorted part contains only one element so it is the minimum. We swap it with itself. The array is: [(2 3 4 7 9) ()].
- Now that the whole array is sorted, the operation is completed.

If you want a short, visual example you can check this video. Just like the rest of this homework, you are not allowed to use any code from the internet or other sources. You must implement this function on your own!

7. `unsigned int* find_finishing_positions(float** lap_times, unsigned int n_drivers, unsigned int n_laps):` Given lap times (`lap_times`), number of drivers (`n_drivers`) and number of laps (`n_laps`) this function will find the race result, that is, the finishing position of each driver. Finishing positions are based on the sum of lap times for each driver (shorter time means higher ranking). The array this function will return, holds IDs of the drivers sorted on their race completion time. For example if there are three drivers with finishing positions as follows:

`Driver 0:` 2^{nd},
`Driver 1:` 1^{st},
`Driver 2:` 3^{rd},
the returned array should be {`1, 0, 2`}. If times are the same for two drivers, assume that the driver with the smaller ID finishes before the other. Also notice that rankings start from 1, not 0
Hint: For this task, it may be beneficial to write a helper function where you use an auxiliary array with driver IDs (i.e. {0, 1, 2, 3, ...}).

8. `float* find_time_diff(float** lap_times, unsigned int n_drivers, unsigned int n_laps, unsigned int driver1, unsigned int driver2):` Given lap times (`lap_times`), number of drivers (`n_drivers`), number of laps (`n_laps`) and two driver IDs (`driver1` and `driver2`) this function will find the time difference between two drivers for each lap. This will be done by subtracting `driver2`'s lap time from `driver1`'s lap time. The function will return the array containing these differences. Time differences for laps are accumulated (i.e. carries onto the next lap). For example, in a 3 lap race with the following lap times:
`Driver 1:` {`18, 21, 20`}
`Driver 2:` {`20, 20, 22`}
Time differences for these two drivers would be: {`-2, -1, -3`}.

9. `unsigned int* calculate_total_points(unsigned int** positions, unsigned int p_drivers, unsigned int n_races):` Given positions (`positions`), number of drivers (`p_drivers`) and number of races (`n_races`), This function will read `position-point` mapping from stdin and return the total point each driver received. For example if `p_drivers` is 10 and finishing positions for two drivers are:
`Driver0:` {`1, 2, 2, 4`}
`Driver1:` {`3, 1, 5, 6`},
and the `position-point` mapping read from stdin is:
`25 18 15 12 10 8 6 4 2 1`
Total points for `Driver0` is (25+18+18+12 = 73), and for `Driver1` it is (15+25+10+8 = 58). The `position-point` mapping you are going to read from stdin will contain `p_drivers` of unsigned integer values where the first integer corresponds to points received for finishing first, second integer for finishing second and so on. This function should return the array containing total points for each driver in the index determined by driver's ID (e.g. following the above example, first element of the array should be 73 and the second element at index 1 should be 58).

10. `unsigned int find_season_ranking(unsigned int* total_points, unsigned int p_drivers, unsigned int id):` Given total points (`total_points`), number of drivers (`p_drivers`) and a driver ID (`id`) this function should return the season ranking of driver with ID `id`. Higher total points means the driver finished at a higher ranking and the driver with the highest points is ranked 1^{st}. If points are the same for two drivers, assume that the driver with the smaller ID is ranked higher.
Hint: You do **not** need to sort drivers based on their times like you do in task 7.

Clarifications

- Although this shouldn't change your implementation, two input tables are independent of each other.
- Higher ranking means the rank is smaller, e.g. rank 2 is higher than rank 5.
- Each task is worth 10 points.
- While testing your codes, output of one function may be input to another. This may cause an error in a task to propagate to the others. Be extremely careful about the first two tasks, as errors in those tasks affect all others.
- Do not modify any function arguments passed by reference, except for tasks 1 and 2. For example if you modify `lap_times` in `find_fastest_lap`, any consecutive calls to `find_finishing_positions` may return an incorrect value.
- You need to complete the function definitions provided in `functions.c`
- You should make extensive use of dynamic memory allocation.
- Download `the2_files.zip` from CengClass and take a look at the provided `Makefile`, `main.c`, `functions.h` and `functions.c` files.

Regulations

- **Programming Language:** C
- You are only allowed to use topics from the first 10 weeks of the syllabus. Please do not use `structs`.
- **Submission:** Submissions will be made via CengClass. Upload a single file named `functions.c` which contains your function implementations and nothing else. Since you will not be uploading any other file, anything you do in `main.c` and `functions.h` will be lost.
- Using the provided `Makefile`, your codes will be compiled as follows :
  `gcc functions.c main.c -ansi -Wall -pedantic errors -o main`
  `main.c` is the file that calls the functions you defined in `functions.c`.
- Before submission, please test your codes (compile and run) on **inek** machines by connecting via SSH.
- You may define helper functions.
- **Evaluation:** Your codes will be graded using black-box technique. So, make sure that your functions return exactly the expected output and arrays have the expected length.
- **Newsgroup:** You must follow COW **daily** for discussions, possible updates, and corrections.

- **Cheating:** We have zero tolerance policy for cheating. Your solution must be your own. People involved in cheating will be punished according to the university regulations and will get 0 from the homework. Sharing code between students or using third party code is strictly forbidden. Even if you take only a "part" of the code from somewhere or somebody else, this is also cheating. Please be aware that there are "very advanced tools" that detect if two codes are similar.