CMPUT 275 – Tangible Computing II **Winter 2020**
Weekly Exercise #2

Restaurants and Pointers: getRestaurantFast

In class, we implemented the function `getRestaurant` that would read the restaurant data for a given restaurant. This function is not as efficient as it could be. For example, if you get the information about restaurant 2 and then restaurant 3, `getRestaurant` reads the same block of the SD card twice. Given the slow speed of reading from the SD card, this is wasted time. We could get away with only reading a block once and then using it twice to return the information about both restaurants.

Your Task:

Your task is to implement a function called `getRestaurantFast` that helps reduce the overall number of block reads. You will also develop a touchscreen display and user interface to call both retrieval methods, `getRestaurant` and `getRestaurantFast`.

The function `getRestaurantFast` should behave exactly as the function `getRestaurant` developed in class (including having the same parameters), except if two consecutive calls request restaurants in the same block then the latter call should not have to re-read the block from the SD card. Use global variables to help you achieve this.

Using the TFT display, you will be able to select a retrieval method by pressing the corresponding button. You will also time how long it takes to read all restaurants from the SD card using the selected method and then report both that time (in milliseconds) and the average running time over all runs.

TFT Display:

When you first upload your program to the Arduino, you should see the following setup: On the righthand side of the display will be two buttons, one labelled `FAST` and the other labelled `SLOW`. The order of the buttons on the screen does not matter, but area used for the button column must be exactly **60 x 320 pixels on the right-hand-side** of the screen. This is to be consistent with the first major assignment.

In the main area of the display (the leftmost 420 x 320 pixels), you should display the following lines of text in order:

```
RECENT SLOW RUN:
Not yet run
SLOW RUN AVG:
Not yet run
RECENT FAST RUN:
Not yet run
FAST RUN AVG:
Not yet run
```

Note: The text `Not yet run` is printed initially to indicate that no data has been recorded for either `getRestaurant` or `getRestaurantFast` (there have been no previous runs for the screen to display). You will overwrite this text later.

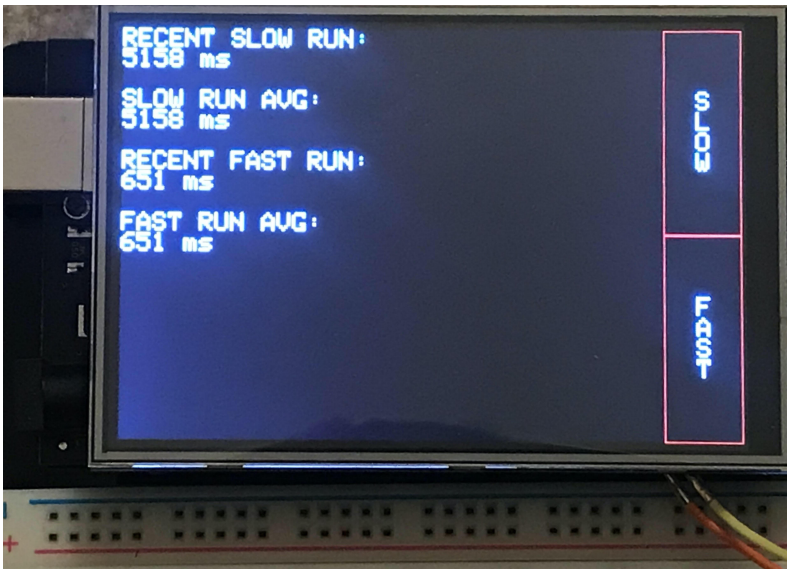

Figure 1: An example user interface.

When the `SLOW` button is pressed, your program should:

1. Read all restaurants in sequential order of index (from 0 to `NUM_RESTAURANTS`-1) from the SD card using `getRestaurant` (developed in class) and record the amount of time it takes to run.
2. Display this newest runtime on the TFT display under the text `RECENT SLOW RUN:` (replacing `Not yet run` if applicable). Do not print any restaurant information, only the total time for `getRestaurant`.
3. Compute and display the new average runtime on the TFT display under the text `SLOW RUN AVG:` (replacing `Not yet run` if applicable).

When the `FAST` button is pressed, your program should:

1. Read all restaurants in sequential order of index from the SD card using `getRestaurantFast` (which you will implement) and record the amount of time it takes to run.
2. Display this newest runtime on the TFT display under the text `RECENT FAST RUN:` (replacing `Not yet run` if applicable). Do not print any restaurant information, only the total time for `getRestaurantFast`.
3. Compute and display the new average runtime on the TFT display under the text `FAST RUN AVG:` (replacing `Not yet run` if applicable).

Note: Make sure you reset the pin modes after reading from the touch screen. Otherwise, you will not be able to write the running time to the TFT display.

Tracking the Average:

To compute the average running time, a naive approach might be to store all previous running times in an array. Then, to compute the average, you could simply sum all values in the array and divide the total by the length of the array. However, this is slow and requires recomputation (previous values must be summed multiple times!). One approach that avoids this redundant recomputation is to store and modify only the following two values:

- the sum S of all running times for this retrieval method
- the number of times n that this retrieval method has been called

In your program, initialize both variables to 0. Each time a retrieval method is called, add the new running time to S, increment n, and compute the new average A using:

$$A = S/n$$

You may assume that in your program, the sum will never become large enough to overflow the standard integer type on the Arduino.

Note: This is not the only method for quickly computing the average and you are welcome to use a different method as long as you avoid storing old running time values in an array. For example, it is possible to compute the new average using only the old average and the number of times this retrieval method has been called. This method does not require storing the sum S and the old running time values in an array.

Hints:

- You may want to consider using global variables to store values between calls to each function. Note, we are only concerned with improving the case where consecutive calls to `getRestaurant` are from the same block. If you call `getRestaurantFast` for restaurant 2, and then 10 and then 3, you will still need to read in block 0, then block 1, then block 0 again.
- The function millis may come handy.

Implementation Requirements:

- The unit of time display is in **milliseconds**. Do not report the runtime in any other unit. Indicating the unit using `ms` is perfectly acceptable, but not required.
- The text on your display must exactly match the specifications, but you may change display details such as colour, font, or casing (ie, `FAST` vs `Fast`). You are encouraged to play around and experiment, but be aware that **you may lose marks if your user interface is unreasonable** (ie, white text on a yellow background, or impossibly small buttons).
- Round the average running times should be rounded down to an integer number of ms.

- When redrawing the screen, it is acceptable to redraw inefficiently (ie, redraw more of the screen than is needed). However, at minimum, **you must not redraw the buttons at every update**. The buttons should never need to be updated, so you should limit your redraw to the rectangular area of the screen which contains the text display.
- We understand that touchscreens are finicky. Therefore, we will only test button presses in the center of your buttons.

Optional Challenges:

All of the following are acceptable modifications to the specifications, which you may find interesting or useful to implement in your solution. However, they **will not** affect your grade either way.

- Design your own custom `FAST` and `SLOW` buttons. You may change attributes such as letter case (ie, `Fast` or `fast`) as well as the shape, colour, or size of the buttons (be reasonable). However, **you may not** change the button column width or the button text itself.
- Note that it is also acceptable to change the font, text size, or colour of the text in the main part of the screen. However, the text must appear in the same order and you may not change the wording given in the specifications.
- When printing a new time or average to the screen, try to redraw only the minimum number of pixels. This requires more sophisticated bookkeeping.
- Add a third button to the righthand column labelled `BOTH` that will run both `getRestaurantFast` and `getRestaurant` and update all times accordingly.

Submission Guidelines:

Submit all of the following files as `restaurant.tar.gz` or `restaurant.zip`:

- `get_restaurant.cpp`, containing your solution to the weekly exercise
- the Makefile
- your `README`

Make sure your submission follows the Code Submission Guidelines!