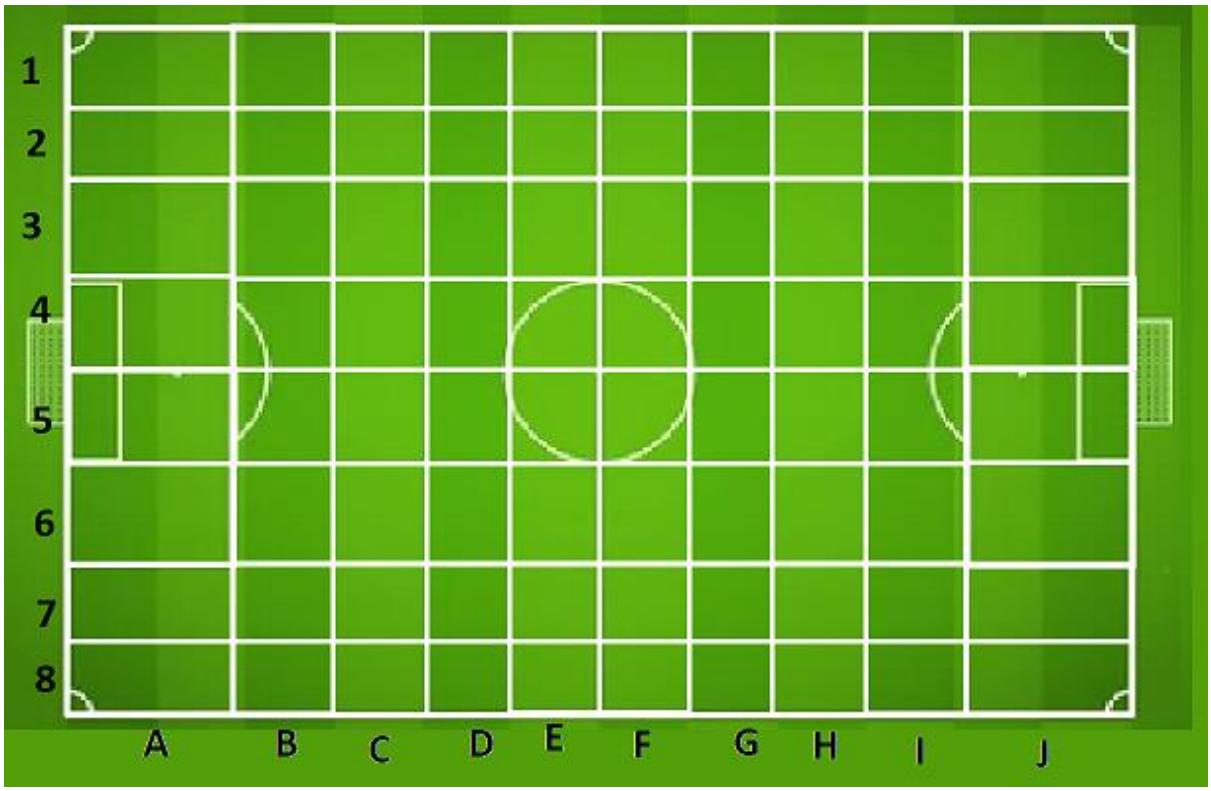
Assignment 2

Brian Moyles – 21333461

Part 1

I will be doing a hashmap approach to generate a heatmap for the football pitch. As shown below, I have divided the pitch into an 8x10 grid. The center of the pitch is divided more as that is where the majority of the play happens. The corners of the pitch are not divided as small as not as much play happens in that area. My grid uses 1-8 on the y-axis and a-j on the x-axis which guarantees unique cell coordinates.

My approach will take a hashmap that stores a key and a floats. As a heatmap is used to determine the amount of time spent in a cell, I do not feel the need to store in the hashmap the times at which the player was in the cell. Instead, I will use those in an algorithm to calculate how long the player was in each cell for. At the end of the game, my hashmap will be able to be used to determine the proportion of time spent in each cell.



I will store 2 values in the hashmap: a string to represent the key, and a float for the time

The key is a concatenation of the player id and the cell in which they are in on the pitch, in the format [playerID-xy]. Every player has their own unique id and every cell has its own unique x and y value. By concatenating these together, it gives a key that can be used to show the player and which cell they are in. It would make searching for a particular player in a cell easy as all you need is the player id and the cell x and y value

The float stores the amount of time the player spent in the cell in which the key is referring to. This is done by using an algorithm to calculate the time spent. My approach to this is to

get the time at which the player enters the cell and store it as a variable. Every second, the algorithm will check the players position to see if they move cell. If the player moves cell, the entry time to the other cell is stored as an exit time and the initial entry time is subtracted from the exit time. This gives the amount of time the player spends in the specific cell.

Pseudocode

```
Main Class:
  playerMap: HashMap to store player ID and time spent in each cell

  Constructor:
    Initialize playerMap

  Method movePlayerToCell(x, y, playerId):
    key = String.format("%d-%c%d", playerId, x, y)
    if (player moves to a different cell):
      entryTime = Current local time
      if (playerMap does not contain key):
        create entry for playerMap with key and initialize the time to 0
        calculate time spent in the cell using getTimeSpent(entryTime) algorithm
      else:
        calculate time spent in the cell using getTimeSpent(entryTime) algorithm

  Method getTimeSpent(entryTime):
    Calculate timeSpent by subtracting entryTime from the current local time
    Return timeSpent as a float representing the time in seconds
```

This algorithm will allow for the updating of the float value whenever the player spends time in the cell and then leaves. I believe this is an efficient way to track how much time spent in each cell as in order to generate a heatmap, only the amount of time spent is needed and therefore, storing the time entered and exited in a hashmap is not as efficient.

Part 2

I could parallelize the approach in part 1 by using inter-query parallelization. I would treat each player as their own entity and parallelize their movements.

As the players move independently on the pitch, they could move cells at different times. Hence, their movements will be processed concurrently.

By treating each player separately and running their queries on different processors, it would allow for efficiency and the simultaneous updating of their heatmaps.

Each player would be assigned to a separate processor so that their movements are independently processed.

Part 3

In the context of a 3x3 heatmap tracking player positions on the pitch, given the unpredictable data size due to being unable to know how many times players move cell, and

due to the players moving independently on the pitch, I believe dynamic hashing is a suitable indexing approach for the task. Because of the dynamic nature of movement by players on the pitch, a static hash map would struggle to store all the data efficiently.

Dynamic hashing addresses the issue of dynamic growth of the data size. It allows the hash table to allocate/deallocate space as needed in order to adapt its size based on the number of elements present. This approach seems very viable and efficient when it comes to multiple players constantly change cells in the heatmap

Dynamic hashing is suitable due to Data Size and Resizing of the table

Due to player movements and frequency of movements being unpredictable, dynamic indexing is the most suitable approach as it will automatically allocate and deallocate space and adjust the hash table size when needed, accommodating the unpredictable nature of the data and ensuring no issues in relation to it.

Attributes of this approach include key generation, hash table insertion and updates, checking time spent in the cell.

The use of hash table operations allows for resizing of the dynamic hash table. When a player moves cell, the load factor of the hash table is compared to the threshold and the table is resized If needed in order to maintain efficiency.

Pseudocode

```
Method movePlayerToCell():

Check load factor

If (load factor > threshold)

        Resize hash table

Insert data in correct place
```

This is a snippet of pseudocode that can be used to demonstrate how dynamic hash table is a more suitable approach. As the player movements is unpredictable, this function is designed to handle effectively the insertion and updating of a dynamic hash table. It shows that whenever a new cell is entered by someone, the load factor is always checked to see if there needs top be a resizing of the hash table. Key generation and checking time spent in the cell (As shown in previous parts) is also used in this indexing approach.

Part 4

My approach involves using the dynamic hash table previously used and by checking if players are in the same cell when a player enters a new cell. When the player enters a new cell, a method will run checking every other active player in a cell (by using their timestamp) and comparing if they are in the same cell as anyone else at the same time.

Because my keys are set up as [id-xy], I would parse the key at '-' and compare xy values. Any xy values that match indicate players are in the same cell. The times could also be checked

as each player has an entry and exit time stored until they leave the cell. Therefore, we could compare if the time spent in the cells overlap and therefore confirming players being in the same cell at the same time.

Key attributes of this approach include key generation, hash table operations (insertions and updates), calculating time spent, check if players are in the same cell. Most of these attributes are inherited from part 3 except confirming players are in the same cell. The below pseudocode outlines an approach to carry this out.

Pseudocode

```
Method movePlayerToCell(x, y, playerId):
   key = String.format("%d-%c%d", playerId, x, y)
   check Load Factor
   if (load factor > threshold):
      resize hash table;
   if (player moves to a different cell):
      entryTime = Current local time
      if (playerMap does not contain key):
         create entry for playerMap with key and initialize the time to 0
         check if other players are in the cell
         calculate time spent in the cell using getTimeSpent(entryTime) algorithm
      else:
         calculate time spent in the cell using getTimeSpent(entryTime) algorithm

Method checkForPlayersInSameCell():
   for every active player // Player with no exit time
      parse key at '-' and store xy value;
      if xy values match:
         players share the same cell at the same time

Method getTimeSpent(entryTime):
   Calculate timeSpent by subtracting entryTime from the current local time
   Return timeSpent as a float representing the time in seconds
```

When the player moves to a new cell, multiple things must be checked: if the hash table needs to be resized, the time entered and spent in the cell, etc. we must now also compare if players share the same cell. The above pseudocode carries this out by running a check whenever a player moves cell. It iterates through every player in a cell and obtains the cell value that they are in, one by one comparing these to the cell value that the player just entered. By comparing these, you can accurately track when players share cells. You could even return in the code which players share the cells together at which time.