Oliver Legg – 201244658 O.Legg@student.liverpool.ac.uk
sgolegg

COMP281 – Assignment 2

1. Game of Life (Problem 1081)

For this problem, I stored the input of the user's desired rows, columns and steps as integers in the main function. The second part of the user input was the game of life board itself. It was fitting to call this 2d char array 'board'. I defined its size using the previous input.

```
char *board = (char *) malloc(rows * columns * sizeof(char));
```

The board could be storing thousands of values and would be too large to store on the stack. Therefore, I used malloc to store the board on the heap. I was unable to reference the array as such: board[0][0], therefore, I had to reference the array as if it were a one dimensional array - Board[i]. To get specific indexes in my 2d array I had to use a formula to get the correct index:

$$row * columns + column$$

Instead of typing this function out many times, I made a function which did this for me. The function took the row and column I wanted, along with the number of columns. This was defined here:

```
int get_index(int row, int column, int columns)
```

I made a function which processed the game itself and how it interacted with each cell in each step. This function was fittingly called step:

```
void step(char *board, int rows, int columns, int steps);
```

An important feature of this game was how cells interact with other cells. Therefore, I made function called neighbour_count which took the x and y coordinate of a cell and counted how many alive cells surrounded the cell.

```
int neighbour_count(char *board, int rows, int columns, int x, int y);
```

I got the neighbour count by using a nested for loop which checked the top left of a cell above the given coordinate (x - 1, y – 1). This would continue from -1 to 1 for both x and y, checking all permutations. The function would return how many alive cells it found. Once I counted how many cells a neighbour has, I acted up on that and went to the game rules:

- Any live cell with fewer than two live neighbours dies, as if caused by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

If one of these conditions were made, I changed a dead cell from '.' to 'X' or a live cell from 'X' to '.'. Afterwards, I had a problem which was changing the board from one state to another. If I was to check board[0][0] for neighbours and change the cell state given it's current state and neighbour count, it would then affect board[0][1]'s neighbours. Each step had to be isolated from other steps, so I had to create another array on the heap which was:

```
char *board_next = (char *) malloc(rows * columns * sizeof(char));
```

Each step, I put every cell into the board next. I didn't check board_next for neighbours because this would conflict. At the end of each step, I set the board to board_next.

Every step iterated for the number of times it was entered by the user and by the end. I printed the board with a function called:

```
void print_board(char *board, int rows, int columns);
```

This was easily made by iterating through every element of the list and printing the character out with new lines at the end of the rows. This would then produce the answer.

2. Highway Lite (Problem 1084)

I started the highway lite problem by taking the number of rows, columns for the highway and the number of time steps for vehicles travelling down the highway. Afterwards, I created another input for the cars on the highway which required the arrival time and row index. You can keep inserting these indefinitely but can be interrupted with an EOF. Because the inputs are indefinite, I created a linked list which allowed me to indefinitely add more vehicles. To create the nodes, I had to use '`malloc()`' which allocated memory to the size of a given node. To use this, I required the following library:

```
#include <stdlib.h>
```

I learnt how to do this in the previous assignment and used the following article to help: https://www.learn-c.org/en/Linked_lists. Each node of the linked list had the arrival time, the row index and the address to next node inside. This allowed me to traverse the linked lists very quickly. I was able to allocate the space in the memory with the malloc passing the structure of node in. To put the values into the linked list, I created the head of the list, set the current to the head and put the values into the head of the list. To access the same linked lists, I used pointers throughout the program so that functions can access the right variables.

```
//Construct of a linked list

typedef struct node {
    int arrival_time;
    int row_index;
    struct node * next;
} node_t;
```

Just like the game of life, I had to create a 2d char array to hold the vehicles and empty spaces. To start, I filled the array with dots on all the cells. Just like the game of life, you can expect to have hundreds of rows and hundreds of columns. This would require a lot of memory to save all this data – which is why I stored it on the heap – not the stack.

```
board = (char *) malloc(rows * columns * sizeof(char));
```

Again, I was unable to reference the array as such: board[0][0], therefore, I had to reference the same as before and used the same formula to get the correct index:

$$row * columns + column$$

I also reused the same function as before so I didn't have to type out the forumla

```
int get_index(int row, int column, int columns)
```

Afterwards, I called a function which works like so:

```
void step(char *board, int t, node_t ** head, int rows, int columns) // Defined like this
step(board, t, &head, rows, columns); //Called like this
```

This takes the boards pointer; the time step it is currently on, the head of the linked list and the number of rows and columns. The step function algorithm works by making the last column of the board empty (turned into dots) as this would simulate the cars moving to the right off the board. Afterwards, I used a nested for loop to move every column element further down by 1 for each row. The final part was adding new cars to the board if it was scheduled to do so. This works by traversing through the whole linked list, to see if each node's arrival_time matches up with the passed 't' variable in the step function. After the node was found, the vehicle is added to the board given the row_index. The node is then deleted from the list and the links are reattached. This step function will be called as dictated by the number of timesteps given from at the start of the program by the user.

The board is then printed out to the user using a function which takes the board (char array pointer) as a variable along with the number of rows and columns.

```
void print_board(char *board, int rows, int columns)
```

This was easily made by iterating through every element of the list and printing the character out with new lines at the end of the rows. This would then produce the answer.