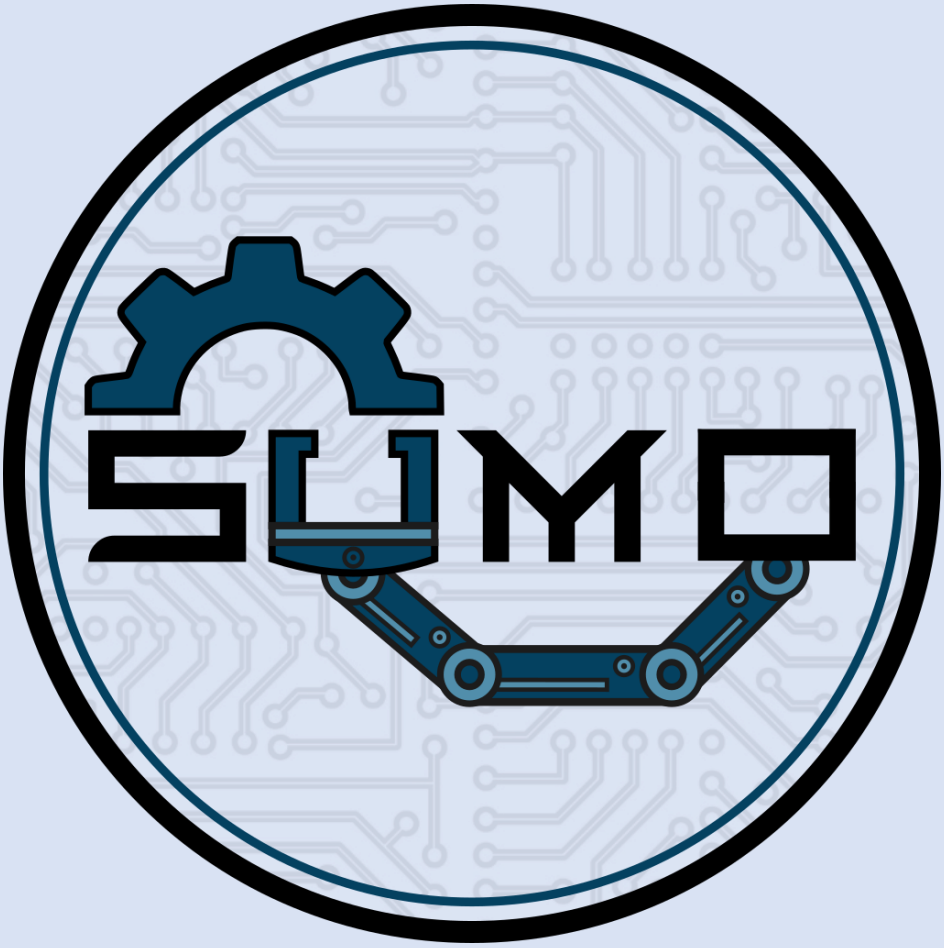
SUMO Hackathon 2023



What is a hackathon???????

A hackathon is basically an event in which the participants (in groups) are given a prompt/challenge, and a very short time span to solve that problem.

Like cramming before an exam, it forces you to think and learn in a quick amount of time, and hopefully develop some skills that can help you in your degree.

Schedule

SUMO is giving you the prompt in Week 11 Tuesday at 2pm, and we will receive submissions from you at the latest of Week 11 Thursday at 2pm. Unfortunately, no simple extensions :(

On Wednesday at 11am - 5pm SUMO will run a help desk for the hackathon in PNR, where any question related to the code will be answered (HINT: use this!!!).

GIVE PROMPT —---------> **HELP SESSION** —----------> ***SUBMIT***
(Tuesday 2pm) (Wednesday ANYTIME) (Thursday 2pm)

Reminder

The point of this hackathon is to have fun and *learn*. If you are stressed out and can't figure out a solution, ask one of the SUMO execs, or use the internet to learn it. We will check that the winners understand what code they wrote and why they wrote it, so cheating in any way is *pointless* for the competition and for yourself.

Basically don't spam Chat-GPT is what we are saying. If we find you cheating you will get 0 points and 0 PEP.

PEP HOURS

Obviously PEP is very important. And you are probably wondering how to get all 40 hours. We will base whether you are valid for PEP based on the quality of your final submission and the effort you put in. If you don’t have much to show, but at least tried to learn (eg. Showed up at the help session, asked questions on the discord, etc.), you will get PEP. Don’t worry about making it perfect just try to put an effort in.

DISCORD -> ***https://discord.gg/CScBEB3u***

SUMO SIGNUP ->

!!! Prompt !!!

Your job over the next two days is to complete as many objectives as you can from each task below. There are 3 tasks, and each task has around 5 objectives associated with each one worth a certain amount of points. The team with the most points by the end of the time wins - so be smart about which tasks you attempt. You will get 10 bonus points for completing a task completely. The maximum amount of points achievable is 100.

TASK 1: BOX, BOX, BOX

30 points

TASK 2: I SWEAR I HAVE MATES

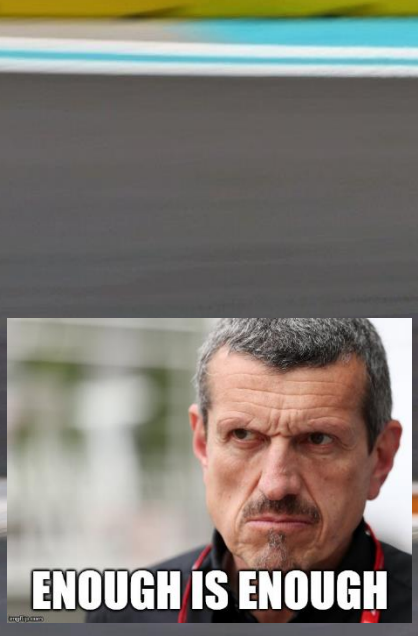
20 points

TASK 3: MAZE RUNNER

20 points

+ 10 points for fully completing each challenge.

TASK 1: BOX, BOX, BOX



After another bad race last weekend at the F1 Miami Grand Prix, Oscar Piastri is looking for a new team of race engineers ahead of Italy and your group has been chosen.
Lucky for you, they use Matplotlib to visualize lots of the race data, and you are *specialists*.
You and the team break down the challenge into smaller parts to be completed below:

Objective 1: Read the input file (4 points)

The first task your group must complete is reading a file. The other engineers have given you a file called *splits.txt*, which is a data sheet from a bunch of sensors on the track. These sensors are positioned at each turn, and whenever the car drives past them, the time is recorded.

An example data sheet from a practice with 3 laps and 5 sensors is shown below:

```
3, 5
665, 149, 8.014424865262042, 43.758597678190995, 75.05042556710528,
953, 879, 15.714064682426706, 51.50811582436499, 80.7792168145455,
690, 15, 24.27099454533803, 58.965273425867, 88.6859920135904,
545, 215, 31.883035273651032, 63.46726684901262, 93.42445885574435,
33, 206, 36.03317740285296, 69.47456450689647, 101.44091987941457,
```

The first line tells you the number of laps and sensors, and then each line after represents each sensor. These sensors have a x coordinate and y coordinate as the first two values of the line, and then the times which the car passes it.

To get the marks for this objective, receive the text file in, and store each of these lines within a list called **input_lines**.

Objective 2: Store the Variables (4 points)

In **input_lines** we currently have each line as a string, and we want to convert them into readable integers and floats.

Create a new list called **sensor_data,** and store in it tuples of (x_position, y_position, time_1, time_2, … time_n) for each sensor. Both x_position and y_position should be converted to integers and all the times into floats.

Objective 3: Error Check (6 points)

An integral process within your team is making sure that there are no errors within the input file and how you received it.

To complete this objective, your code must be able to check and handle:

- Does the text file exist or not?
- Are any of the times invalid (negative or in the wrong order)
- Are there too few or too many lap times or sensors?
- Are the inputs all the right type? (integers and floats)

To handle these errors, return an informative message to the user of your own choice.

Objective 4: Visualise (8 points)

Next on the list is visualizing the path that the sensors map out. Create a matplotlib plot of the path, assuming that the car travels in a straight line and at the same speed between each of the sensors (obviously not very realistic but just go with it). Make the path a green line, and the sensors red dots. Create a legend for each of the sensors.

Objective 5: Animate (8 points)

Finally, you need to animate each lap that the car completes. Create a method that animates between two points (x0, y0) -> (x1, y1) at a certain speed, and then call that multiple times to go through each lap (the prev lap should dissapear each time). If the split is the fastest split that has ever happened, animate the line as purple, and if not keep it as green. In the legend, hold the fastest time for each split. Once the animation is complete, create a log of a new file called *log.txt.* This should look like this:

```
FastestLap:
FastestSpeed:
FastestSplit1:
…
FastestSplitN:
```


TASK 2:
I SWEAR I
HAVE
MATES

Unfortunately, you got sacked from the F1 team because you spent too much time on Chess.com. You now want to dedicate your life to making chess in python and test your *logic* and *problem solving* skills in an enjoyable way.

Objective 1: Read the input file (2 points)

The first task is to receive the input from the terminal and convert it into a 2D array representing the chess board. The board that will be inputted will be a 5 X 5 board with a completely random position.

The input file will look something like this:

000K0
00QB0
00000
0pbn0
00k00

Where 0 represents an empty tile, and:

K/k = King, Q/q = Queen, B/b = Bishop, N/n = Knight, R/r = Rook, P/p = Pawn.

Capital letters represent the black pieces and lowercase the white. We can assume that we will always be playing as the white pieces, it is always our move, we are always playing such that pawns move up the page, and that it is not already in check.

Store the input in a 2D list.

Objective 2: Find the possible moves (4 points)

In order to figure out this problem, we must start by creating a list of possible moves that we could play from our position.

We want this list, called **possible_moves** to hold a tuple of four items:

- **Starting_row** - the piece's start row
- **Starting_col** - the piece's start col
- **End_row** - the piece's end row
- **End_Col** - the piece's end col

Before we add it to the list however, we must check that the position that we are moving the piece to is not the same colour.

To get all the marks for this section, **possible_moves** must hold every possible move that any white piece can make in the current position, ignoring checks on its own king. Make a function that takes an input of the 2D array, and a colour, and return the possible moves that the colour can make (as this will help you later).

Objective 3: Check position? (4 points)

The next part of the puzzle is determining whether a certain version of the board is in check. To do this, we can run our function to get the possible moves and then see whether any of them are attacking an opposing king. Once again have this as its own function.

Objective 4: Simulate possible moves (6 points)

Before we can run the possible moves, we are required to clean up the possible moves that we have. This is because we might be moving pieces that put our own king into check, which is illegal.

Make a function that simulates the possible moves, and checks whether the new position is one that in a "Check" position.

To get the points you must have a list that holds the actual moves that each piece can make after being cleaned.

Objective 5: Solution (4 points)

Return to the user whether a check is possible from the original position or not as a boolean. (Print to terminal) Also print the piece that is making the check.

NOTE:
Obviously there are lots of rules in chess. A lot. Some things that we are assuming for the inputs:

- We are always playing as White, and have the pieces moving up the board
- Pawns can only always move one square and are not allowed to do En Passant or promote.
- There is always a black king on the board.
- No castling will be available.

- Unlimited timing

TASK 3: MAZE RUNNER

After playing to much Chess, you lie down to take a break. When you wake up…

You are Thomas from The Maze Runner. The conditions in the Glade are worsening and you need to escape otherwise you will DIE. But because you're doing engineering, you're going to try find the way out with code…

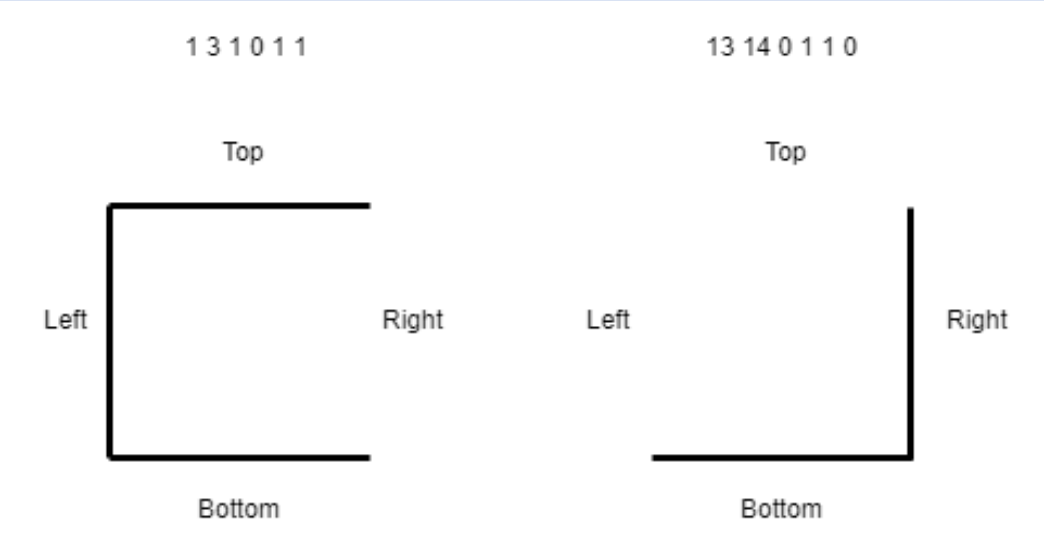
Objective 1: Display the maze (3 points)

For this challenge you are to be given a large text file called *maze.txt* with data for a 15 x 15 maze.

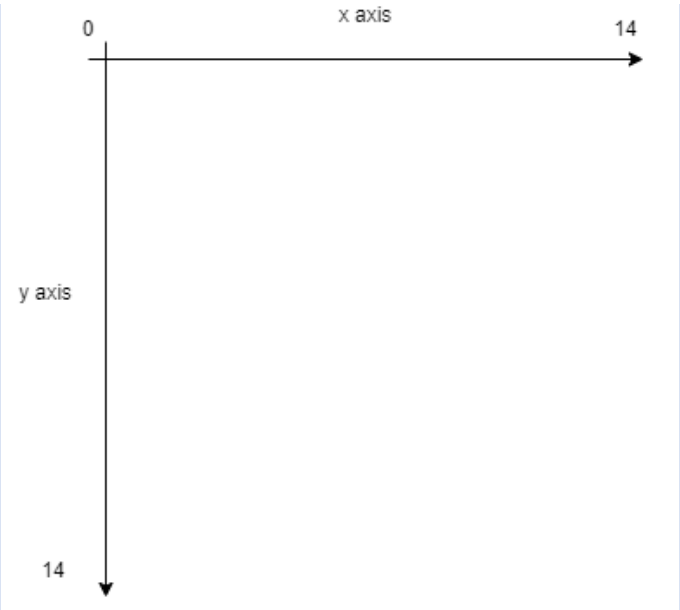
Each line of the text file will correspond to a grid cell on the maze. The first value on each line will be the x coordinate of the cell, the second value will be the y coordinate of the cell. The next four

values represent the top, right, bottom, and left walls. If the value is 1, it means that there is a wall present on that side of the cell and if the value is 0 there is no wall on the side.
Here are a few examples:

1 3 1 0 1 1 : This grid cell is at position x = 1 and y = 3
13 14 0 1 1 0 : This grid cell is at position x = 13 and y = 14



The



The *starting position* will always be at coordinate 7,7 (centre of the maze) and the end of the maze will always be in the bottom right corner of the maze at coordinate 14,14. Add some sort of identifier so that it is clear where the start and the end of the maze is. It is also given that the cells on the edges already have the boundaries of the maze.

Visualise the maze using some sort of display.

Objective 2: Is it possible? (4 points)

You will be given several mazes in the form of a text file. You need to determine if you can actually escape the maze or will get eaten by Grievers. You must create an algorithm that determines if the exit of the maze is reachable from the start of the maze. Return a Boolean of the answer in the terminal.

Objective 3: Traversing the maze (6 points)

For this challenge you must create a visualisation of your algorithm working to find an exit using a visual display. If a dead end is reached, the visualisation must show the path backtracking.

If Thomas cannot find the way out of the maze, then show him returning to the Glade.

Objective 4: Shortest Escape (7 points)

You and your team are very short of time. You need to escape before night fall, otherwise they will get violently eaten and ripped apart by the Grievers which would be sad. You need to find the shortest path to the end of the maze, otherwise they won't be able to reach the end by nightfall.

The algorithm must return the directions that the escapees must take to reach the end in the shortest time (so if there are multiple ways to escape it must take the fastest way)

Animate this fastest path on the map, and create a file *directions.txt* which holds on each line either *LEFT, RIGHT* or *STRAIGHT* to show which way to go at each intersection.