For the Rush hour lab, I tried to use bitmasking to represent the boards, and partially implemented it, but I think my initial plan was to ambitious and is unable to be completed in 80 minutes. I got inspiration from [this](https://www.michaelfogleman.com/rush/#Enumerating_States) article, and decided to try something similar using Python, but utilizing Python’s unique capabilities. For one, Python doesn’t have a set size for integers. Integers are not size limited and are automatically allocated the required amount of memory to store them.

Rush hour boards are defined by a 6 x 6 board, and cars are of length 2 x3. The key to realize here is that the cars do not need to be unique. The goal is to get the car to the end. The way to check if the goal state has been achieved is through bitmasking. The bitwise operation in Python are:

| Operations | Example |
| --- | --- |
| a >>b | Shifting the bits in a b places left  110 >> 1 = 011 |
| a << b | Shifting bits left |
| a ^ b | a XOR b - returns 1(True) if a and b are either 1 and 0 or 0 and 1. |
| a | b | a OR b -- true if a or b is true |
| a & b | a AND B -- True only if both are true |

When applying bit shifts, Python requires applying a bit mask to prevent strange things happening with negative values. Python uses the twos complement system -- negative integers and positive integers are represented by binary, and the first place is used to represent the negative sign.

For my boards, I decided to represent them using 36 bits (using Python’s infinite number of bits for storage). The numbering system is the same as the article: where the 0th place is 2^0, 1 is 2^st, etc, so the board would be represented as 2 ^ 36-1 to get 35 digits .

| 0 | 1 | 2 | 3 | 4 | 5 |
| --- | --- | --- | --- | --- | --- |
| 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 |
| 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 |
| 30 | 31 | 32 | 33 | 34 | 35 |

The board can be represented like above.

The positions of cars are uniquely defined by three board states : all, horizontal, and vertical. All is every occupied space on the board, horizontal is the spaces horizontally occupied by cars, and vertical is all the vertical spaces occupied by cars. To move each car, this can be accomplished using bit shifts(taken from article) -- moving a car horizontally is equivalent to doing board >> 1 or board << 1, moving a car vertically is equivalent to board >> 6 or board << 6. The vertical requires a shift of 6 because it must “wrap around”, the horizontal representation must go up by 6 places to get to the next place. Of course, one issue of using bits shifts to do moves is the bit shift changes the entire board. I tried to use a combination of bit masking on the initial state and the change in the desired state to do so. The bit masking combination I used was this for horizontal shifts:

def move\_left(board,row\_mask,row):

all,horz,vert = board

new\_horz = horz >> 1

printboard(board)

child = new\_horz & size\_mask

only\_row = child & row\_mask[row]

no\_row = all & ~row\_mask[row]

new\_b = only\_row|no\_row

new\_vert = new\_b^child

return new\_b,child,new\_vert

board is the board. Row mask would be the sequence of bits that could “zero out” everything in a row or only preserve the row. For example, if I wanted row 0 to be “zeroed out”, I would do board & ~row\_mask[0]. Row mask 0 would 111111 padded by additional zeroes. Using ~ gets the complement of that and allows me to zero out the row. I would use the original version to preserve only the row I wanted.

1 1 1 1 1 1 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1

0 0 0 0 0 0 → 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0

To get the shifted version, of a particular row, I would:

1. shift the horizontal board and save changed board
2. Keep the size of the board by doing a bit mask of ( 2\*\*36-1) to avoid negative signs
3. “zero out” everything in the changed board except for the row I wanted to shift
4. “zero out” the original horizontal board’s row n
5. take the OR operation of 3 and 4 to get the horizontal board

The vertical board for a horizontal shift would be the same. To get the ALL board, I would want to take the AND of the new horizontal board and the original vertical board.

To check if a particular move is valid, I would need to take the AND of the horizontal and the vertical board. If the value is 0, that means there are no collisions between any of the cars.

I wrote two versions of code for rush hour, linked here:

* [rushhour2.py](https://drive.google.com/file/d/1ZHOwIA4d3ALWMSFxE0qZInE9CWLrl9j3/view?usp=sharing)
* [rushhour.py](https://drive.google.com/file/d/1ZCx6Q3oKt7dcz0ZBl_2vHNWSlbRz61YY/view?usp=sharing)

rushhour.py contains my initial attempt with string manipulation. rushhour2.py contains my current attempt.

rushhour2.py contains code to get children of horizontal shifts to the right and check for collisions. Additionally, it contains the code to generate the bit masks for rows and columns. For the rows, I took 0b111111 and multiplied it accordingly by powers of two to move the sequence of ones where they are supposed to be. To do vertical shifts, I created a list of “0” strings and used a for loop and modulo to place the ones where they were supposed to be. I then joined the strings together and converted it into a binary number.

The horizontal shift and isValid() seems to be successful, but I haven’t come up with a method to successfully detect cars going of the board. One of the outputs when I call get\_children with the board 11( car of length 2 on the first row) gets 1, ( car of half a length)