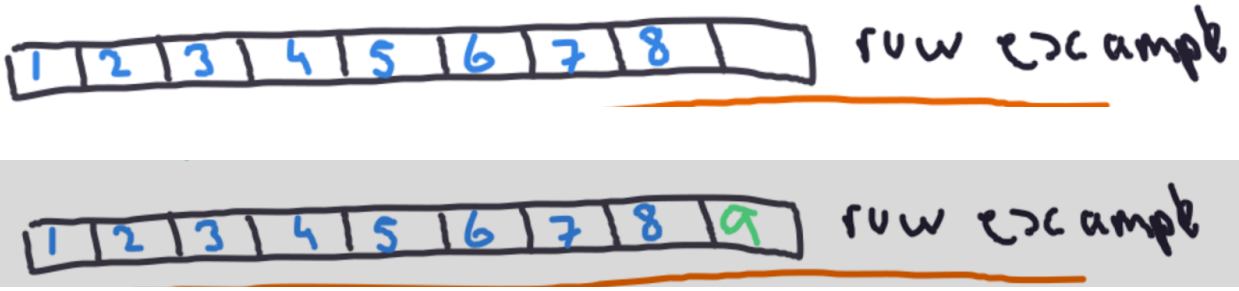
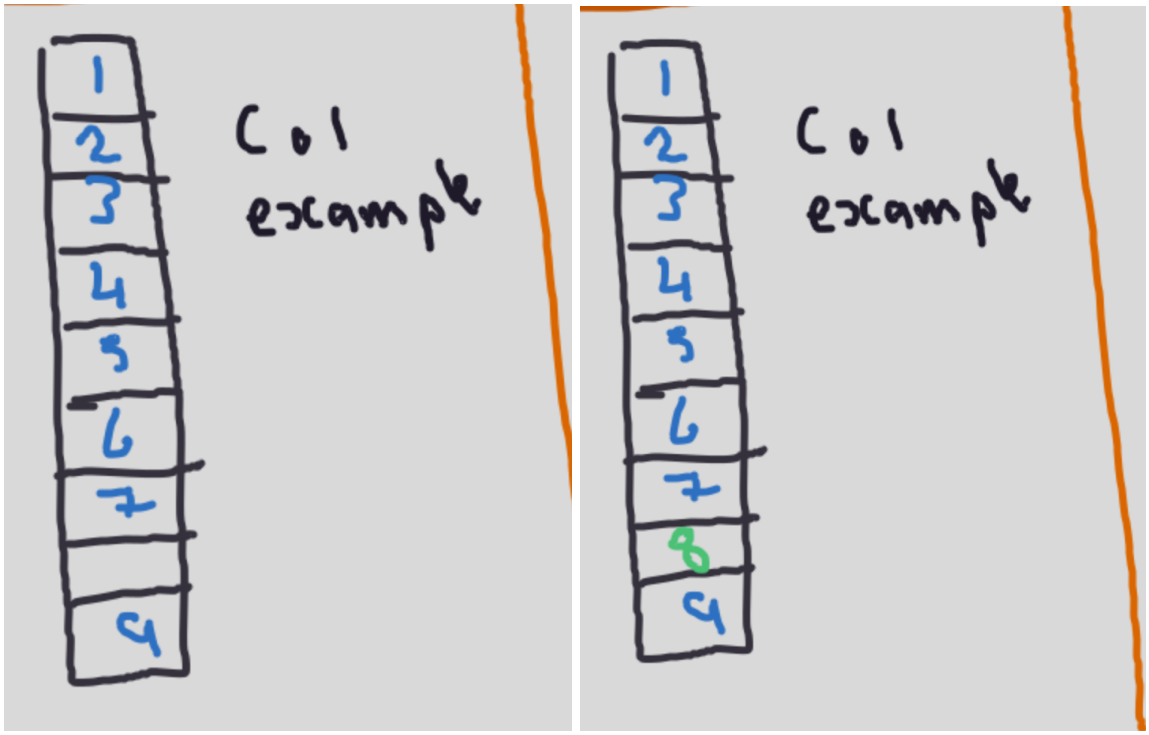
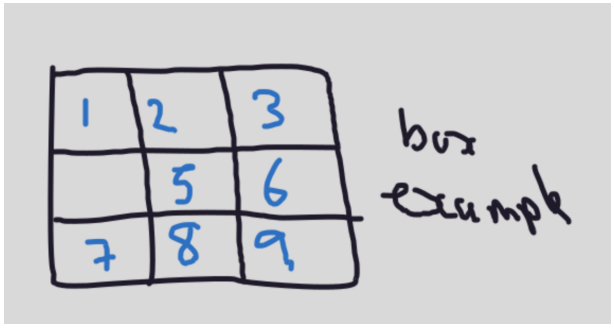
So this is the first write up I've ever done and well to say its been eye opening is an understatement, this will attempt to explain my sudoku solver

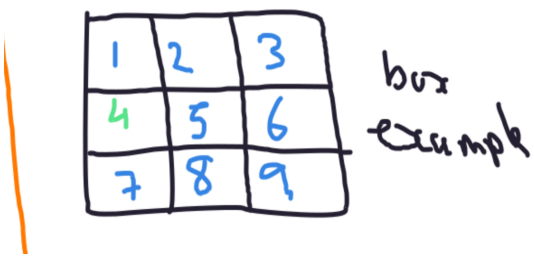
We are building on a lot of axioms, and hoping the logic is true
I interchange between row, col, box, and i, j , k a lot.

Axioms:
Every row must have 1 to 9
Every col must have 1 to 9
Every box must have 1 to 9
If any of the above 3 statements are incorrect the sudoku isn't valid
If you have every cell in a row, col or box filled except 1 it must be the remaining number



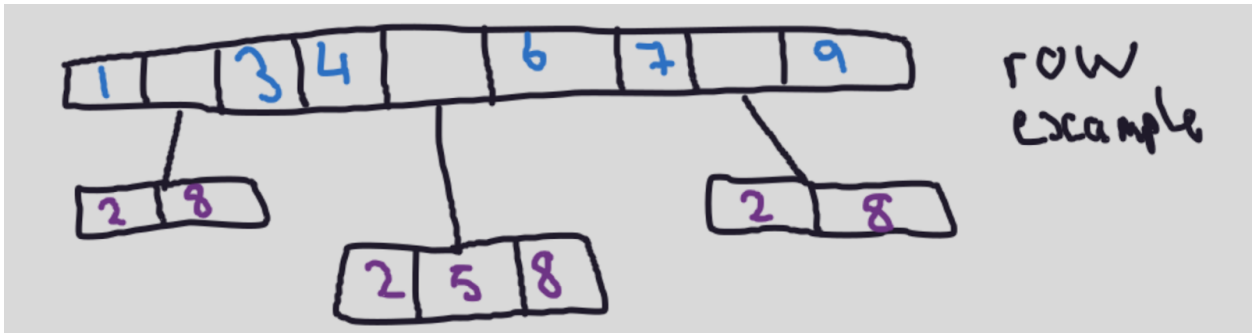


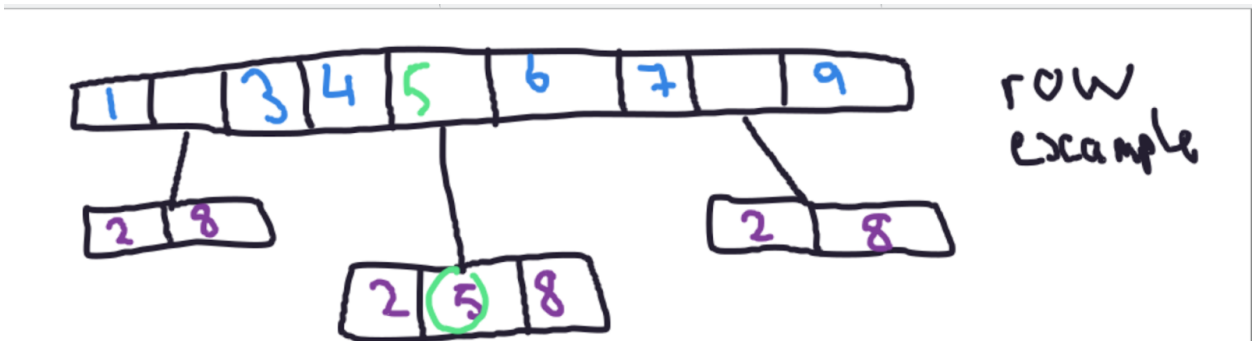




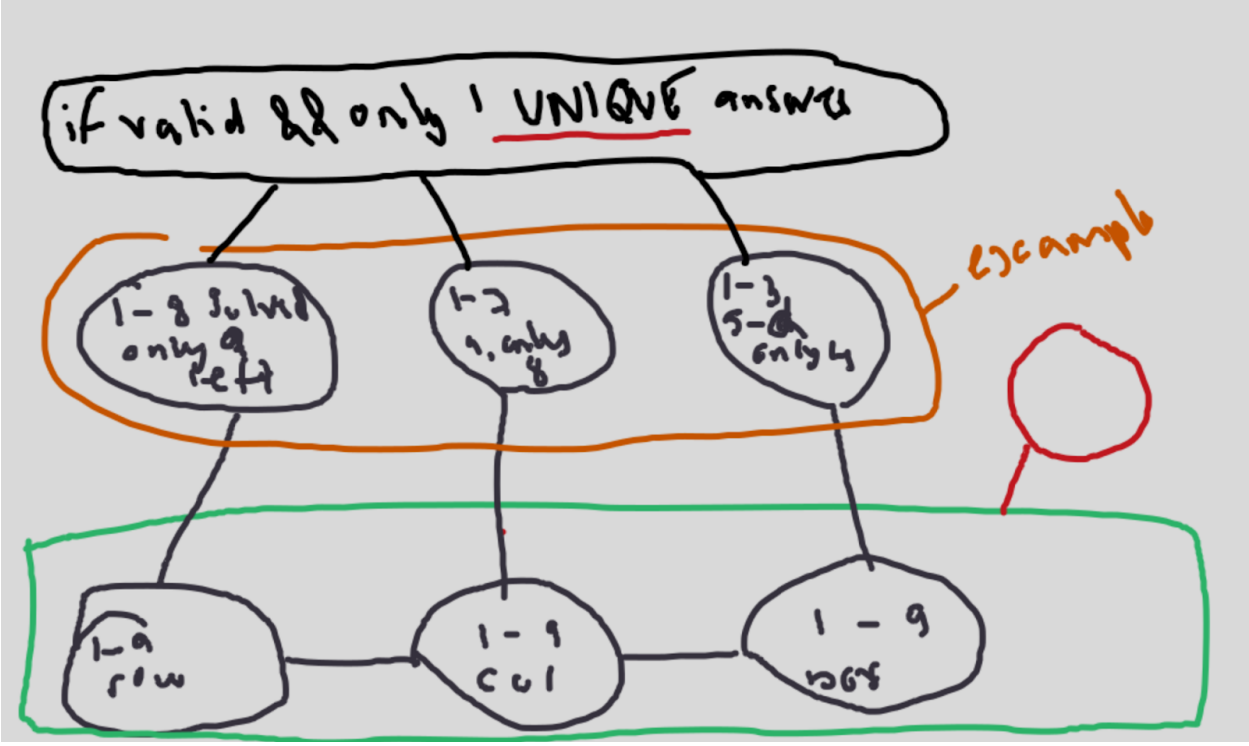
I think those examples are pretty self explanatory

The next example is a bit more complex but the same principal





If you have a set of cells in a row, col or box , and 1 cell has a unique solution not found in the other cells, then the cell containing the unique answer must be the unique answer, in the above example only 1 cell can have the answer 5 therefore it must be 5.



A lot of the functions build on these axioms.

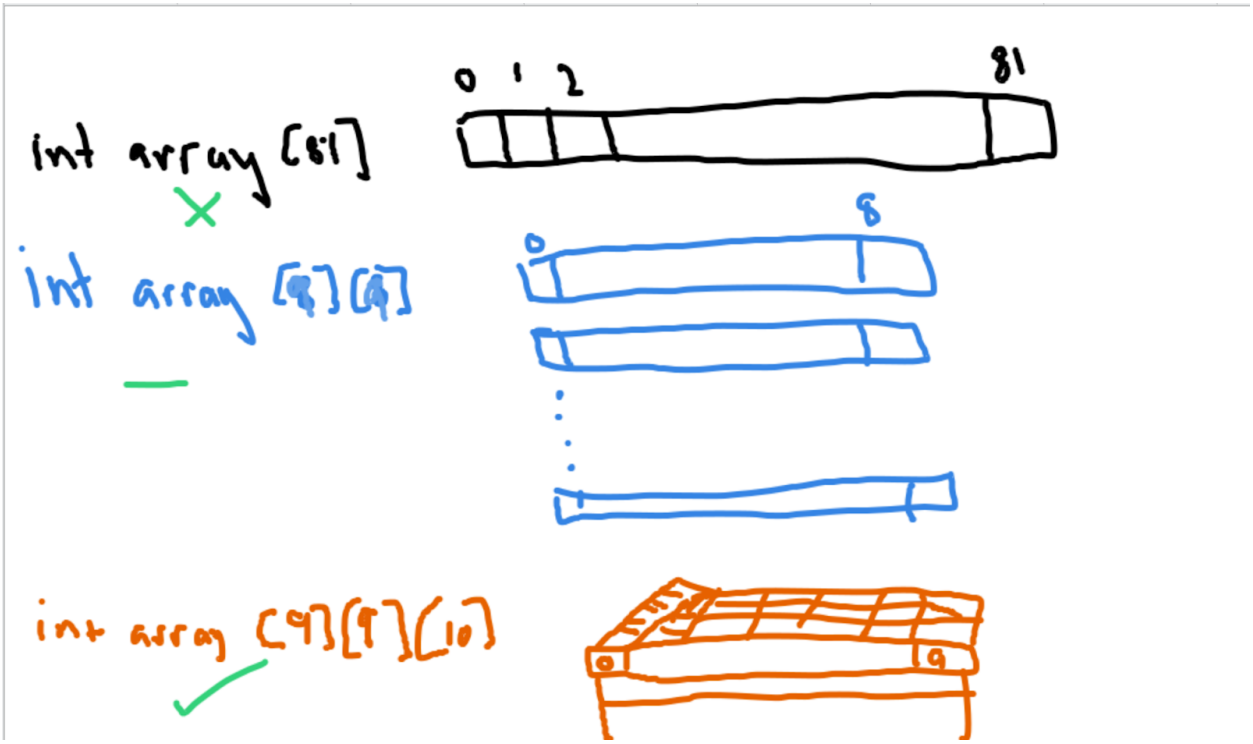
This is the main sudoku we are working with

```
0,9,0,    0,0,0,   0,0,0,
2,0,0,    0,6,0,   0,8,0,
0,0,8,    0,7,0,   0,0,9,

8,0,0,    0,0,7,   1,0,0,
0,0,0,    0,0,0,   0,0,0,
5,0,0,    0,4,0,   0,2,0,

0,0,0,    0,0,3,   0,9,0,
3,5,0,    0,0,1,   0,0,2,
7,0,0,    0,0,5,   0,4,0
```

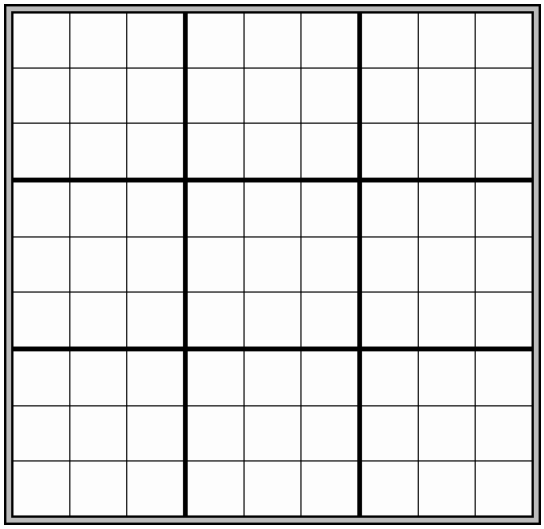
But let's start

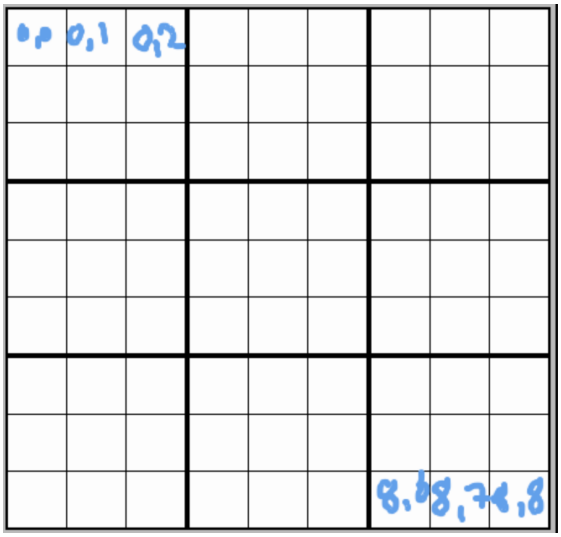
The first thing of significance I noticed is that honestly you can pretty much brute force this. It's not that many calculations in the grand scheme and even in a worst case scenario most computers could easily handle the backtest, especially when you prune paths early on.

The second thing I noticed early on is how you choose to represent the sudoku, will make life a lot easier or harder down the line at a later stage.

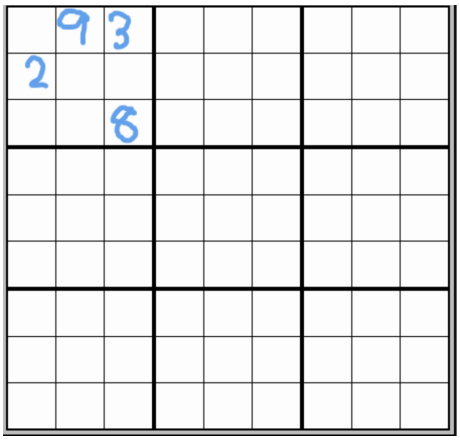


Initially i settle of a 2s array 9x9 , to represent the board

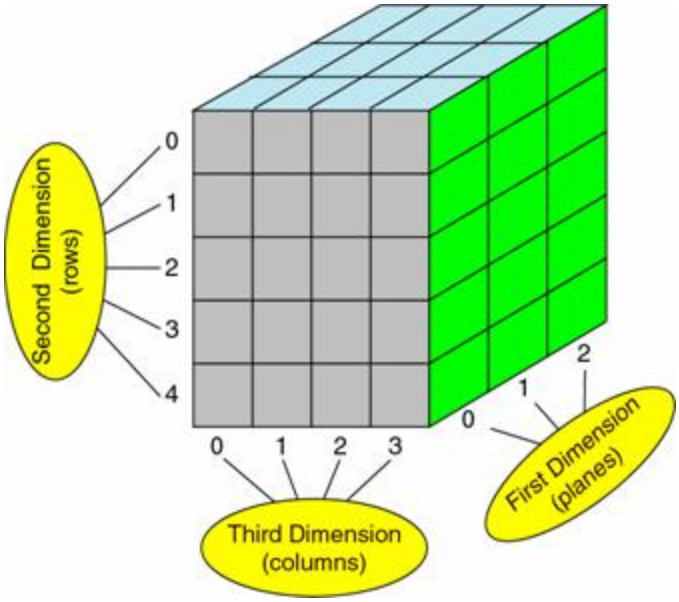





```
0,9,3,     0,0,0,     0,0,0,
2,0,0,     0,6,0,     0,8,0,
0,0,8,     0,7,0,     0,0,9,

8,0,0,     0,0,7,     1,0,0,
0,0,0,     0,0,0,     0,0,0,
5,0,0,     0,4,0,     0,2,0,

0,0,0,     0,0,3,     0,9,0,
3,5,0,     0,0,1,     0,0,2,
7,0,0,     0,0,5,     0,4,0
```



So this is the sudoku im using but in a 2 by 2 array the first array corresponds to the row and second column you see 0.0 in the top left cell, and 0,1 in the second cell would correspond to 9 but what good will that do me unless i could reference it. I have no means of parsing information, I could have used large number of functions to use solve this but I found a 3d multidimensional array to be the most effective solution(for my skill set)



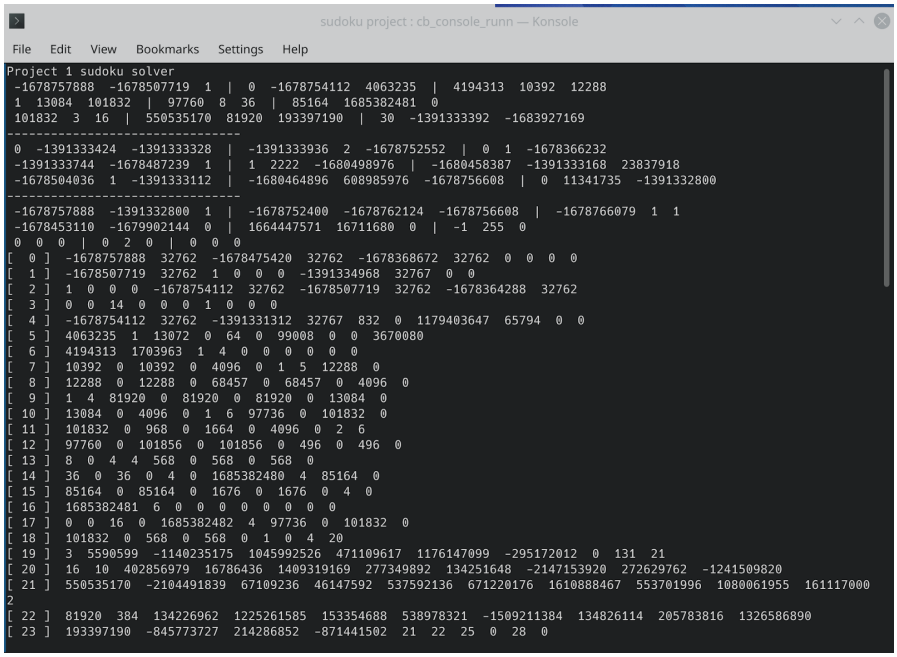
What I liked about this is the grey face can represent the sudoku, as we understand it , and the cells behind them can represent 1- 9, as this is one data structure, all the changes I can make will be permanent , I like that I can refer to every single element weight no issue.

```
std::cout << "Project 1 sudoku solver" << std::endl;

int sudoku[9][9][10];
```

Now we have a 9x9x10

```
void initialise3DArrayOverlay ( int puzzle[81], int sudoku[9][9][10]){ // could be better ways to write

      int i, j, k;
      i = j = k = 0;

          ///*
          for( i = 0; i < 9; i++)
              {
                  for(j = 0; j < 9 ; j++ )
                  {
                      for ( k = 0; k <= 9; k++)
                      {
                      sudoku[i][j][k] = k; // initialise every value of sudoku
                      // i = row, j = col, k = cells for row by col
                      }
                  }
              }
          //*/


          ///*
          for( i = 0; i < 9; i++)
          {
              for ( j = 0; j < 9; j++)
              {
                  sudoku[i][j][0] = puzzle[(i*9)+j]; // we need to convert the single array of 81 into [9][9][0] and store the number in location zero
                  // imagine the first few numbers
                  // i = 0, j = 0  : 0*9 = 0 + 0 so sudoku [i][j][0] = cell 0
                  // i = 0, j = 1  : 0*9 = 0 + 1 so sudoku [i][j][0] which is cell 1 and is a 9 gets stored in the sudoku
                  // i = 5, j = 5 :  5*9 = 45 + 5 = 50 so sudoku [i][j][0] is equal to the element in puzzle[50]
              }
          }
          //*/
```

So lets initialize I,j, k, I could have maybe used globals and called it rows and cols, but i didn't, I also set every cell to its respective number, because im not sure how an array is initialized by default it might be junk values of all zeroes, but this was as far as i am concerned we now have our data structure initialized the way i want. I ran this code without setting any values in the array and it gave me a set of junk values, the initialize will set all the elements of the array to the values they need to be.

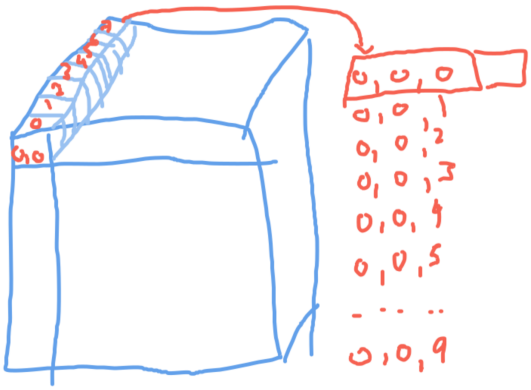


Here is what an example looked with junk values,

```
Project 1 sudoku solver
 0  9  3  |  0  0  0  |  0  0  0
 2  0  0  |  0  6  0  |  0  8  0
 0  0  8  |  0  7  0  |  0  0  9
-------------------------------
 8  0  0  |  0  0  7  |  1  0  0
 0  0  0  |  0  0  0  |  0  0  0
 5  0  0  |  0  4  0  |  0  2  0
-------------------------------
 0  0  0  |  0  0  3  |  0  9  0
 3  5  0  |  0  0  1  |  0  0  2
 7  0  0  |  0  0  5  |  0  4  0
[  0 ]  0  1  2  3  4  5  6  7  8  9
[  1 ]  9  1  2  3  4  5  6  7  8  9
[  2 ]  3  1  2  3  4  5  6  7  8  9
[  3 ]  0  1  2  3  4  5  6  7  8  9
[  4 ]  0  1  2  3  4  5  6  7  8  9
[  5 ]  0  1  2  3  4  5  6  7  8  9
[  6 ]  0  1  2  3  4  5  6  7  8  9
[  7 ]  0  1  2  3  4  5  6  7  8  9
[  8 ]  0  1  2  3  4  5  6  7  8  9
[  9 ]  2  1  2  3  4  5  6  7  8  9
[ 10 ]  0  1  2  3  4  5  6  7  8  9
[ 11 ]  0  1  2  3  4  5  6  7  8  9
```

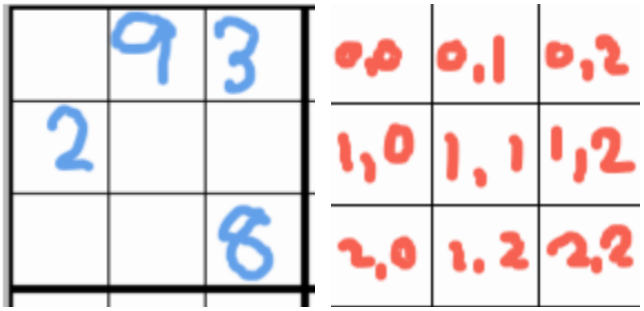
But this is what the data structure looks like now
Also the numbers in brackets are there to help me understand and reference where we stand

So the first cell 0(row),0(col) has 10 cells [row][col][0] -> [row][col[10]



So now i had 10 cells(is this correct term) attached to my each row&column
So the first cell of [row][col][0] is what I used to hold a value, the other 9 cells represented 1, 9 respectively.

Lets look at what the cell should look like
This is what a blank set would look like
[0][0][0] = 0 - 1,2,3,4,5,6,7,8,9
[0][0][1] = 0 - 1,2,3,4,5,6,7,8,9
[0][0][2] = 0 - 1,2,3,4,5,6,7,8,9
[1][0][0] = 0 - 1,2,3,4,5,6,7,8,9
[1][1][0] = 0 - 1,2,3,4,5,6,7,8,9
[1][2][0] = 0 - 1,2,3,4,5,6,7,8,9
[2][0][0] = 0 - 1,2,3,4,5,6,7,8,9
[2][1][0] = 0 - 1,2,3,4,5,6,7,8,9
[2][2][0] = 0 - 1,2,3,4,5,6,7,8,9



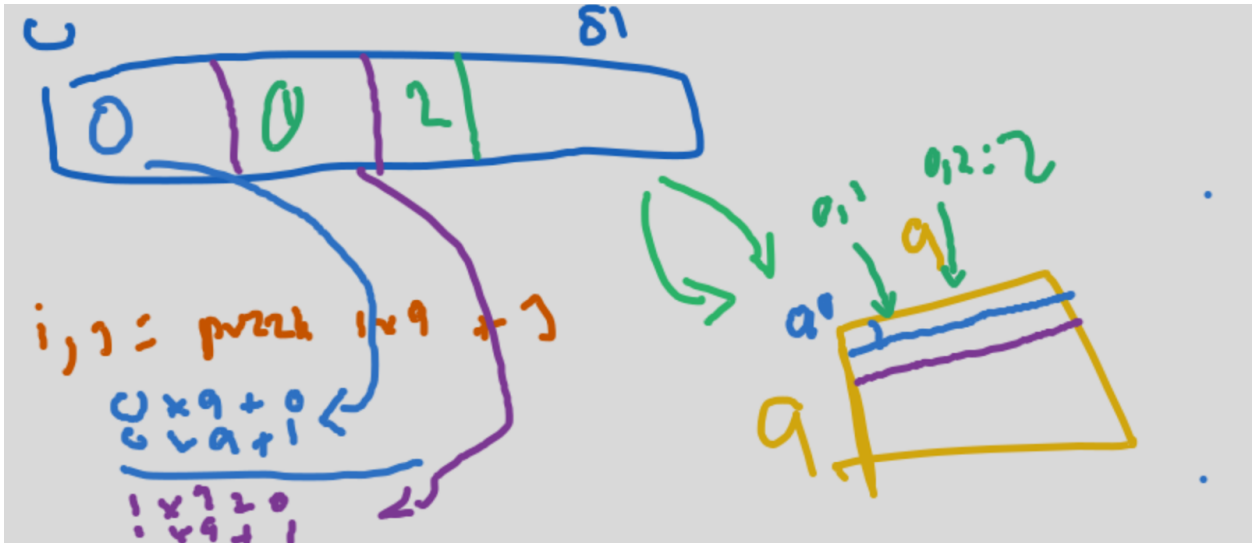
so this is cell 1 right now before
Now we overlay the array with the value we have been given

Now that we overlaid the cells we change
[0][0][0] = 0 - 1,2,3,4,5,6,7,8,9
[0][0][1] = 9 - 1,2,3,4,5,6,7,8,9
[0][0][2] = 3 - 1,2,3,4,5,6,7,8,9
[1][0][0] = 2 - 1,2,3,4,5,6,7,8,9
[1][1][0] = 0 - 1,2,3,4,5,6,7,8,9
[1][2][0] = 0 - 1,2,3,4,5,6,7,8,9
[2][0][0] = 0 - 1,2,3,4,5,6,7,8,9
[2][1][0] = 0 - 1,2,3,4,5,6,7,8,9
[2][2][0] = 8 - 1,2,3,4,5,6,7,8,9

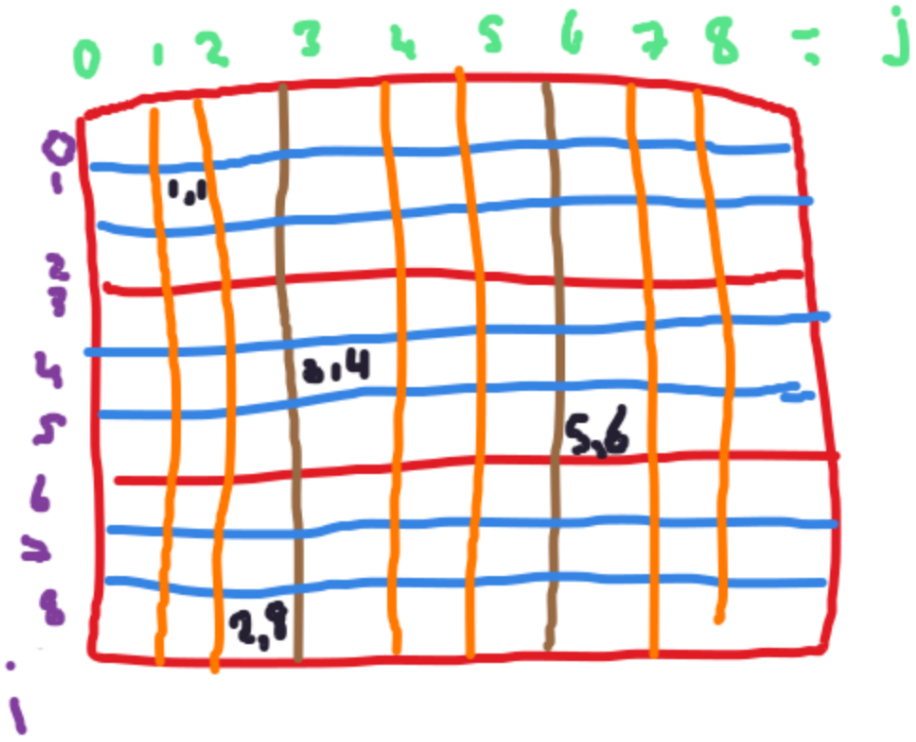
You can see the numbers have been overlaid

The code to overlay

```
for( i = 0; i < 9; i++)
{
    for ( j = 0; j < 9; j++)
    {
        sudoku[i][j][0] = testPuzzle[(i*9)+j];
    }
}
```



The blue array is from 0-81 in a contiguous block
We need to move that into a [9][9], so we loop from 0 to 9, we say [row][col][0] = testpuzzle[i*9+j]



Lets do some example
1,1 would be 9*1 = 9 + 9, that cell would be cell 10 in the 1d array
3,4 would be 9*3 = 27+ 4 = 31
5,6 would be 9*5 = 45+ 6 = 51
2,8 - i wrote wrong

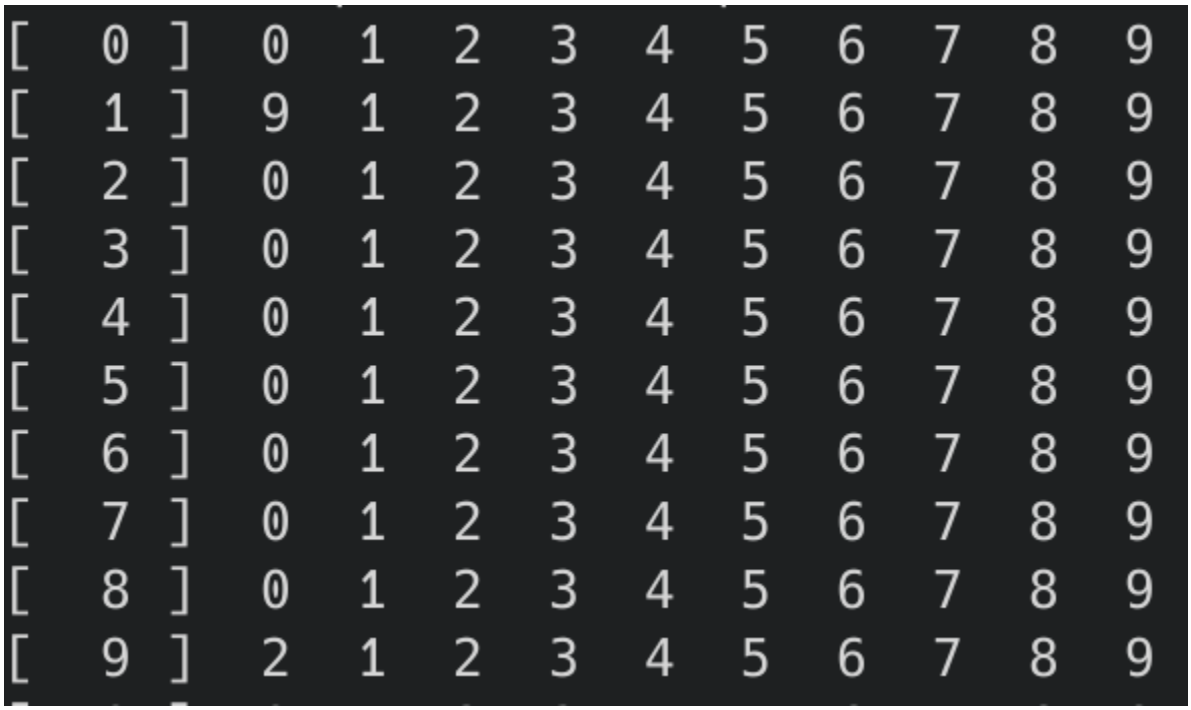
Ok I hope that is clear

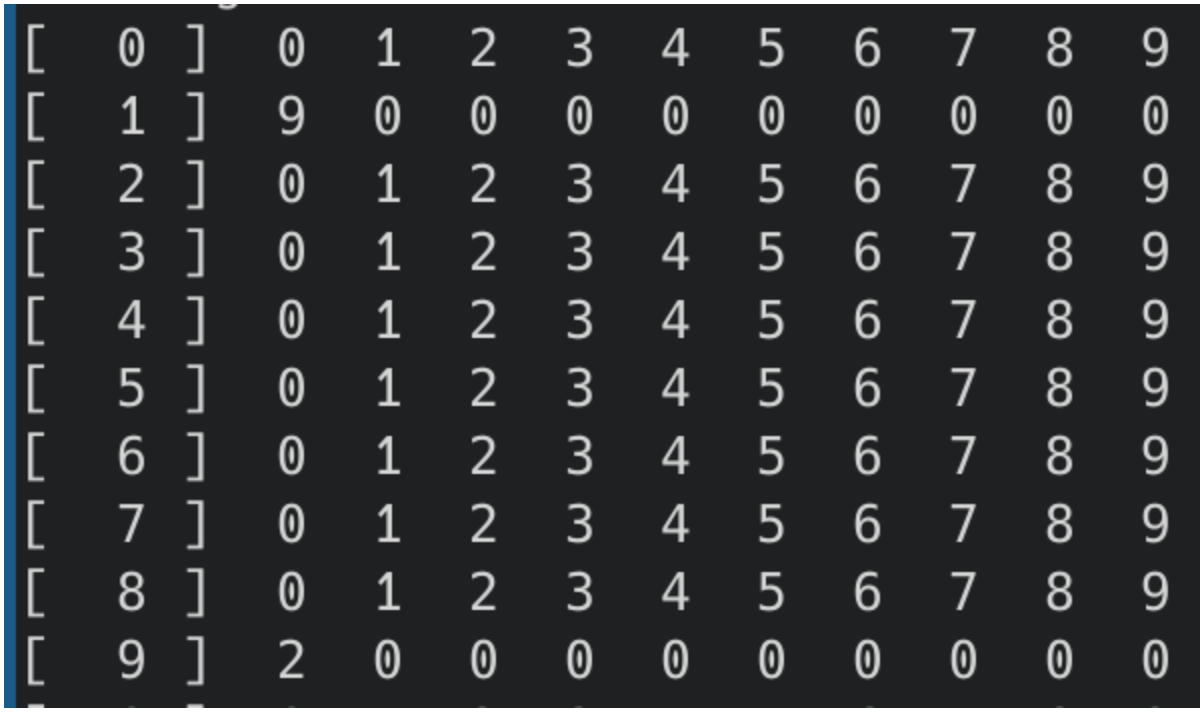
Now I want to do a bit of housekeeping

Clear cell if empty

```
void clearOptionIfSolvedCell( int sudoku[9][9][10]){

    int i, j , k;
    i = j = k = 0;
    int cellanswer = 0;

    for( i = 0; i < 9; i++)
        {
            for(j = 0; j < 9 ; j++ )
                {
                      {
                      if (sudoku[i][j][0] != 0)
                        {
                        //cellanswer = sudoku[i][j][0];

                        for( k = 1; k <= 9; k++)
                            {
                            sudoku[i][j][k] = 0;
                            }
                        }
                      }
                }
        }
        // if [row][col][0] is not zero then make every cell from [row][col][1]->[row][col][9] equal to zero
}
```

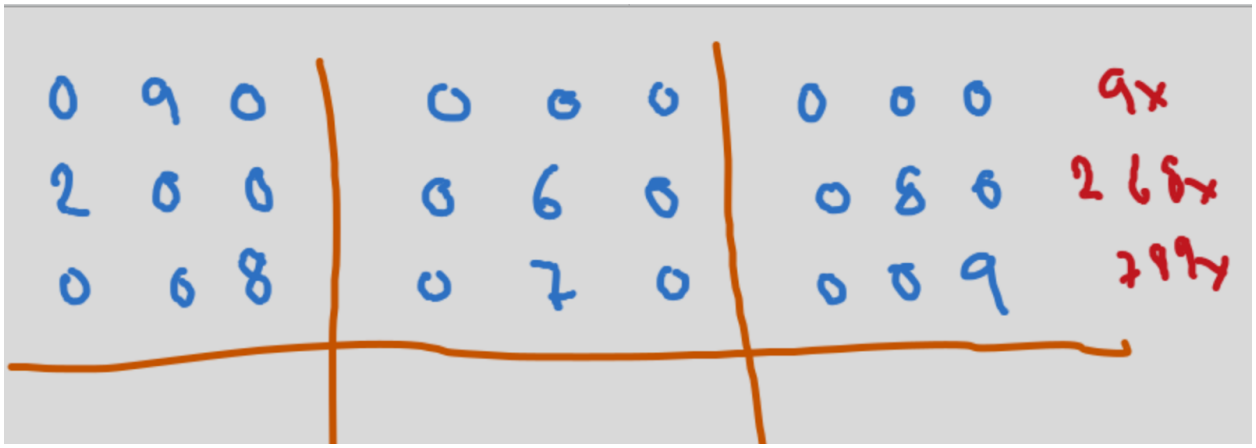
If the cell is solved if [row][col] [0] is an answer which is not zero please clean up every other answer



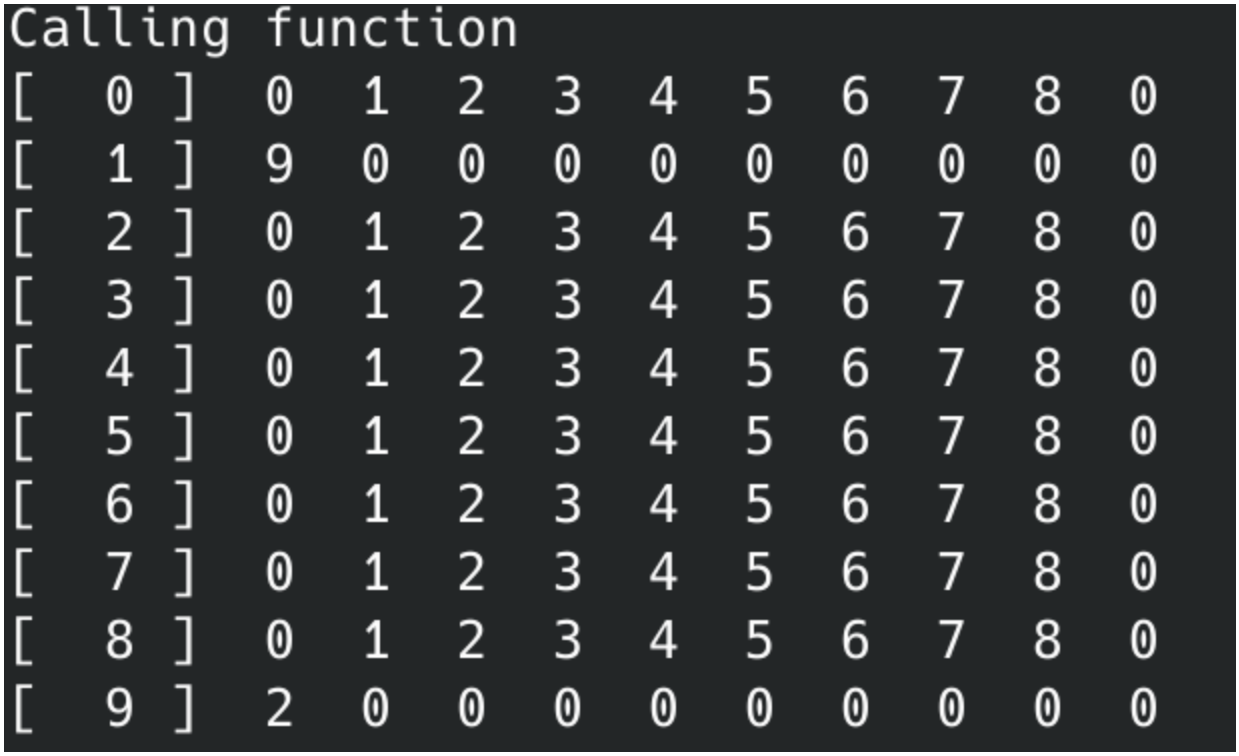


Ok lets eliminate horizontal possibilities

```
void horizontal( int sudoku[9][9][10]){

    int i, j , k;
    i = j = k = 0;
    int cellanswer = 0;
    //std::cout << " horizontal" << std::endl;
        for( i = 0; i < 9; i++)
        {
            for(j = 0; j < 9 ; j++ )
                {
                      {
                      if (sudoku[i][j][0] != 0)
                        {
                        cellanswer = sudoku[i][j][0];

                            for( k = 0; k < 9; k++)
                            {
                            sudoku[i][k][cellanswer] = 0;
                            }
                        }
                      }
                }
        }
/* what im asking is hey if the cell in location [i][j][0] is a number other than zero, make a note
make cell answer that note
and loop from the start of the row which we will call k, and for every [row][k][cellanswer] turn it into zero
so far the first row, our first number is 9, im saying 9 cannot be a soltion for anything in this row
*/
}
```




```
[  0 ]  0  1  2  3  4  5  6  7  8  9
[  1 ]  9  1  2  3  4  5  6  7  8  9
[  2 ]  0  1  2  3  4  5  6  7  8  9
[  3 ]  0  1  2  3  4  5  6  7  8  9
[  4 ]  0  1  2  3  4  5  6  7  8  9
[  5 ]  0  1  2  3  4  5  6  7  8  9
[  6 ]  0  1  2  3  4  5  6  7  8  9
[  7 ]  0  1  2  3  4  5  6  7  8  9
[  8 ]  0  1  2  3  4  5  6  7  8  9
[  9 ]  2  1  2  3  4  5  6  7  8  9
```

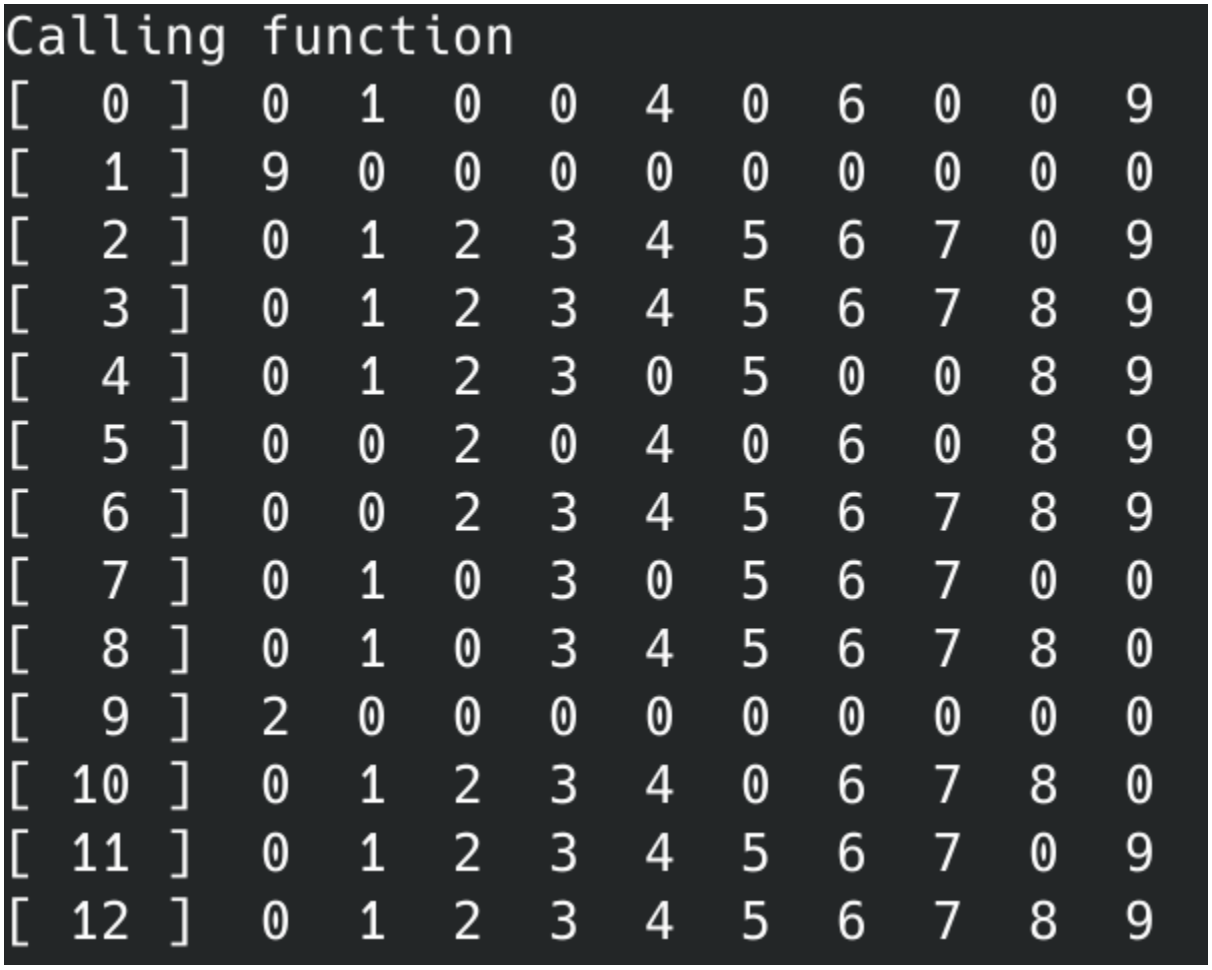


You can see the option 9 has been eliminated from row 1

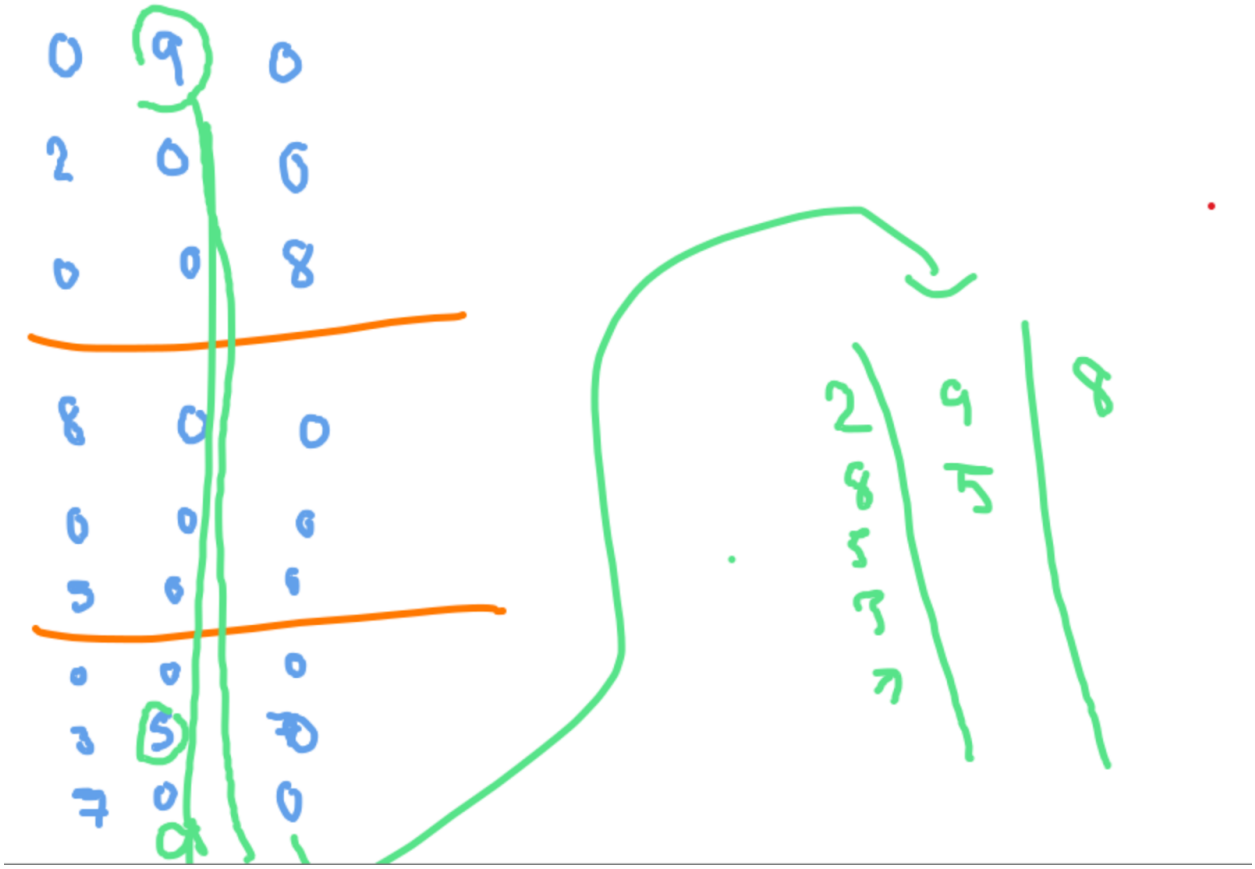
Vertical

```
void vertical( int sudoku[9][9][10]){
    int i, j , k;
    i = j = k = 0;
    int cellanswer = 0;
    //std::cout << " vertical" << std::endl;

        for( i = 0; i < 9; i++)
        {
            for(j = 0; j < 9 ; j++ )
                {
                      {
                      if (sudoku[i][j][0] != 0)
                        {
                        cellanswer = sudoku[i][j][0];

                            for( k = 0; k < 9; k++)
                            {
                            sudoku[k][j][cellanswer] = 0;

                            }
                        }
                      }
                }
        }
/* what im asking is hey if the cell in location [i][j][0] is a number other than zero, make a note
make cell answer that note
and loop from the start of the row which we will call k, and for every [k][col][cellanswer] turn it into zero
so far the first col, our first number is 9, im saying 9 cannot be a soltion for anything in this col
*/

}
```

```
[   0 ]   0   1   2   3   4   5   6   7   8   9
[   1 ]   9   1   2   3   4   5   6   7   8   9
[   2 ]   0   1   2   3   4   5   6   7   8   9
[   3 ]   0   1   2   3   4   5   6   7   8   9
[   4 ]   0   1   2   3   4   5   6   7   8   9
[   5 ]   0   1   2   3   4   5   6   7   8   9
[   6 ]   0   1   2   3   4   5   6   7   8   9
[   7 ]   0   1   2   3   4   5   6   7   8   9
[   8 ]   0   1   2   3   4   5   6   7   8   9
[   9 ]   2   1   2   3   4   5   6   7   8   9
[  10 ]   0   1   2   3   4   5   6   7   8   9
[  11 ]   0   1   2   3   4   5   6   7   8   9
[  12 ]   0   1   2   3   4   5   6   7   8   9
```



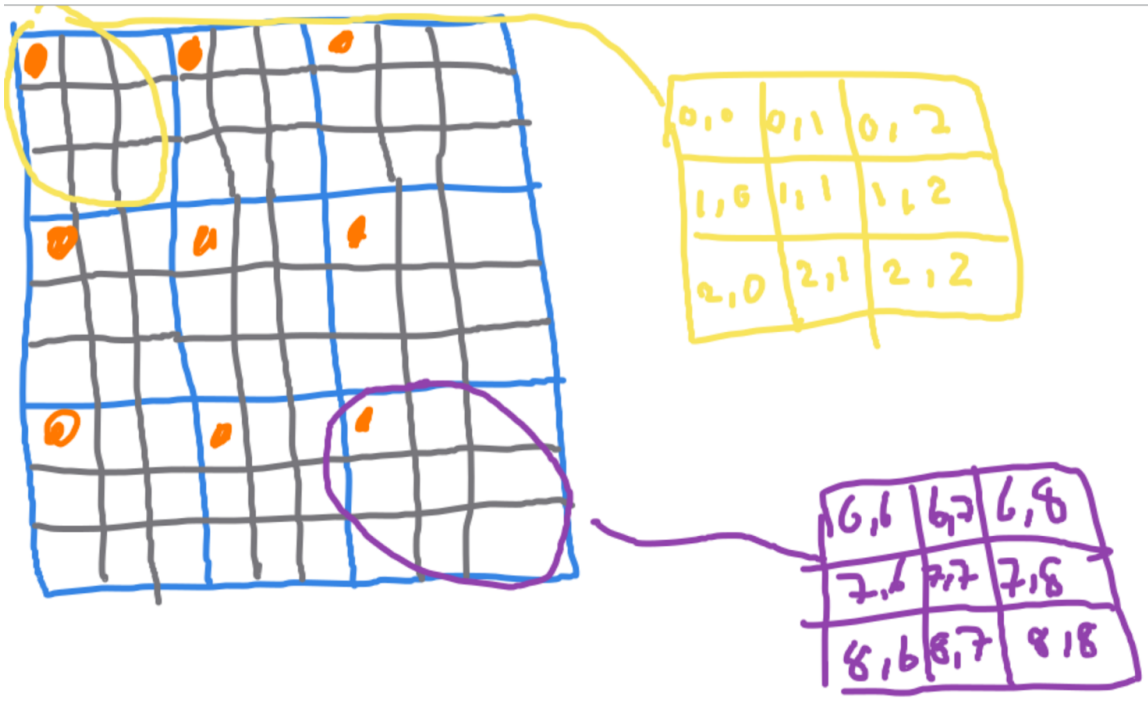
You can cell option 2 has been eliminated from the first cell



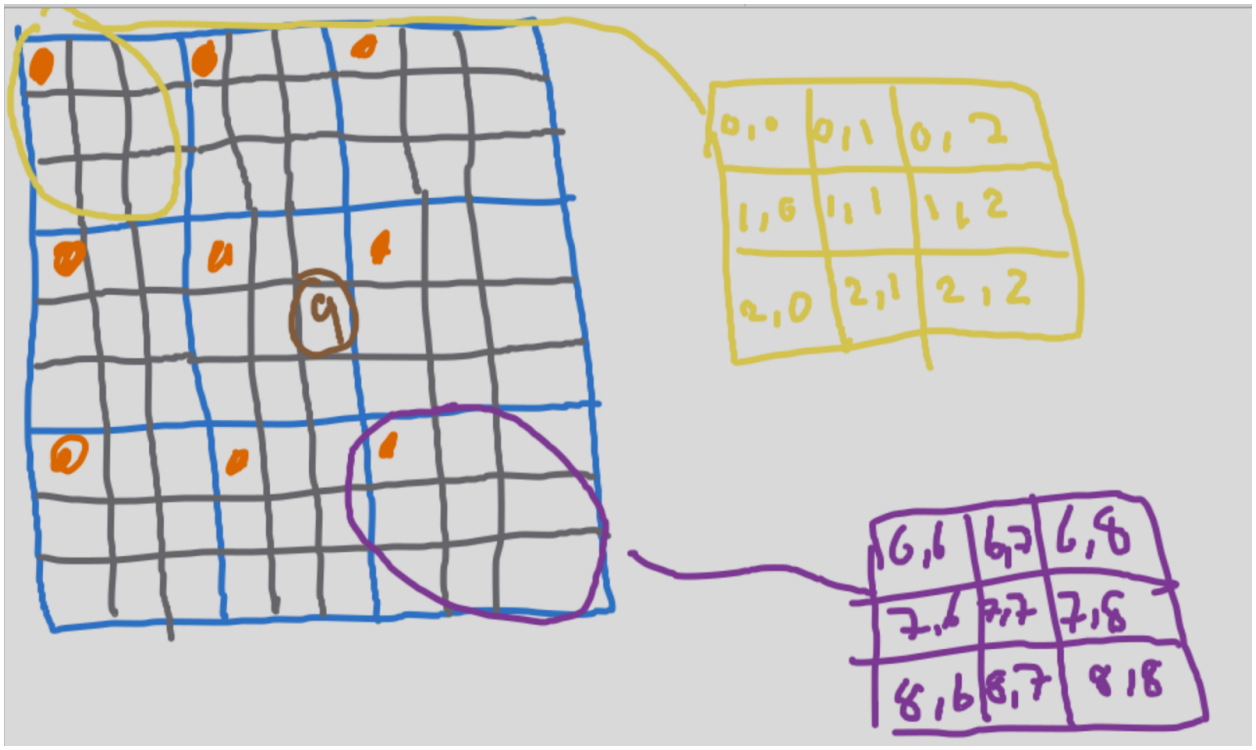
Boxoption

```
void boxoptions(int sudoku[9][9][10]){

    int i, j , k;
    i = j = k = 0;
    //std::cout << " ---box---" << std::endl;


        for( i = 0; i < 9; i++)
        {
            for( j = 0; j < 9; j++ )
            {
                int startpointrow = 0;
                int startpointcol = 0;
                // lets say i is 7 and j is 5
                startpointrow = i/3; // divide yourself by 3 and truncate so 7/3 becomes 2
                startpointcol = j/3; // divide yourself by 3 and trucate so  5/3 becomes 1

                startpointrow *= 3; // times the 2 by 3 becomes 6
                startpointcol *= 3; // times the 1 by 3 becomes 3
                // so the top left most cell in the square beloning to 7,5, is 6,3


                int boxRow, boxCol;
                boxRow = startpointrow;
                boxCol = startpointcol;
                int boxcellanswer = sudoku[i][j][0]; // loop throuigh every cell and store a non zero value
                int cellanswer = boxcellanswer;
                int boxRowlimit = boxRow+2;
                int boxCollimit = boxCol+2;


                for(boxRow; boxRow <= boxRowlimit; boxRow++)
                {
                    for(boxCol; boxCol<= boxCollimit; boxCol++)
                    {
                        if(boxcellanswer != 0) // is the value we stored earlier is non zero
                        {
                        sudoku[boxRow][boxCol][boxcellanswer] = 0;
                        }
                    }
                   boxCol-=3;
                }
            }
        }
// this code basically find if a cell has a value that is no zero,
}
```

Box options should keep a note of every cell in the top left corner,

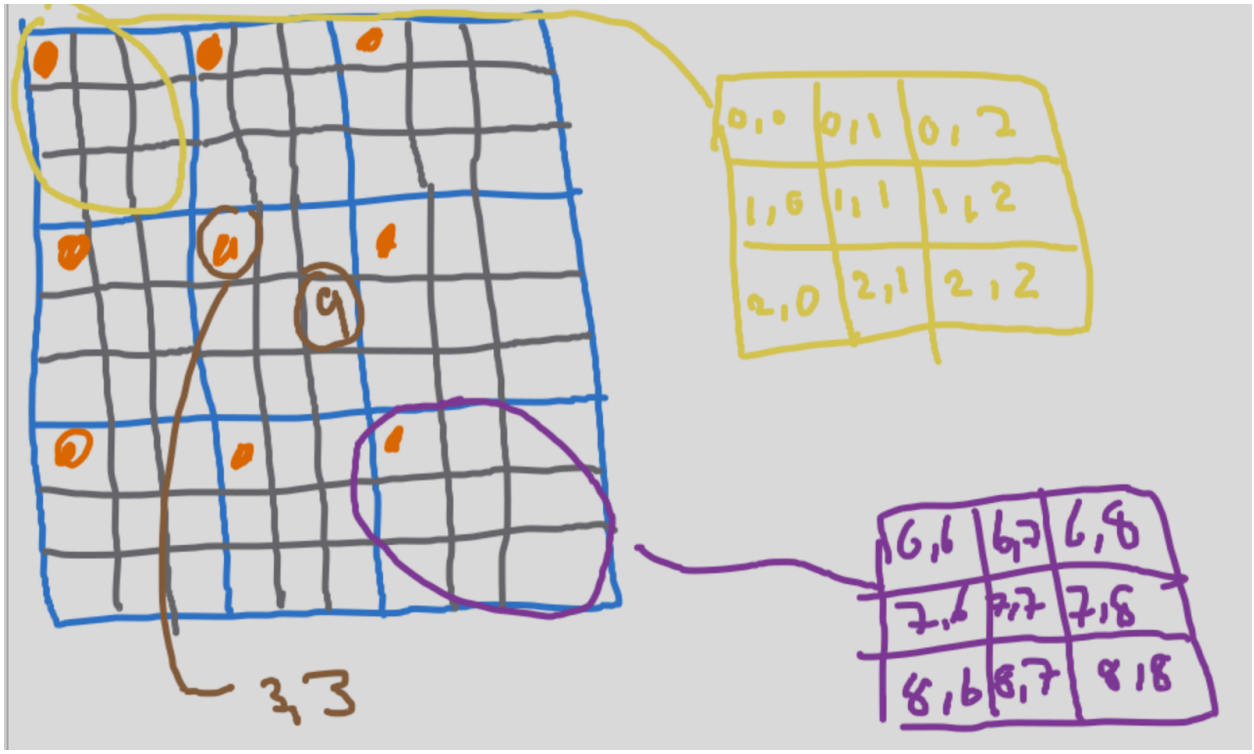


What we need to do is make note of the top left cell of every box, make a note of the cell to check and then eliminate options



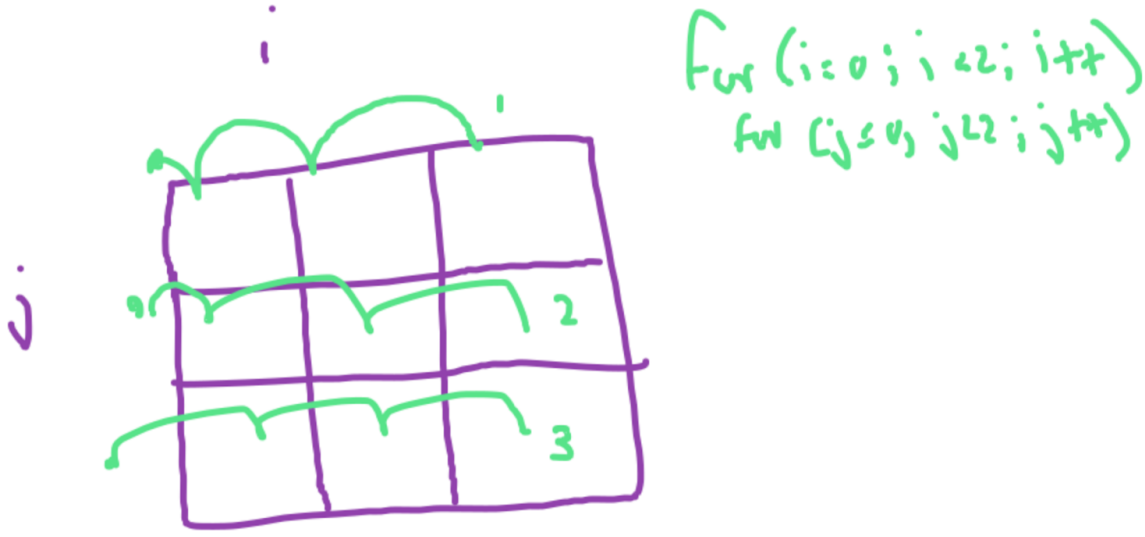
Lets look at 9 for our example, 9 is is cell 4,5

We take 9 as the cell option to eliminate
We divide 4 by 3 , we get 1.3 then truncate to get 1
We divide 5 by 3 we get 1.6 then truncate to get 1
We then times 1* 3 and 1 * 3 to get 3 and 3



Now we set i and j to 3 and 3

And we loop through the cells



We do this for every cell,

In essence what we do is search for every cell, once we have a number we go the left up most cell of that box and loop through the box

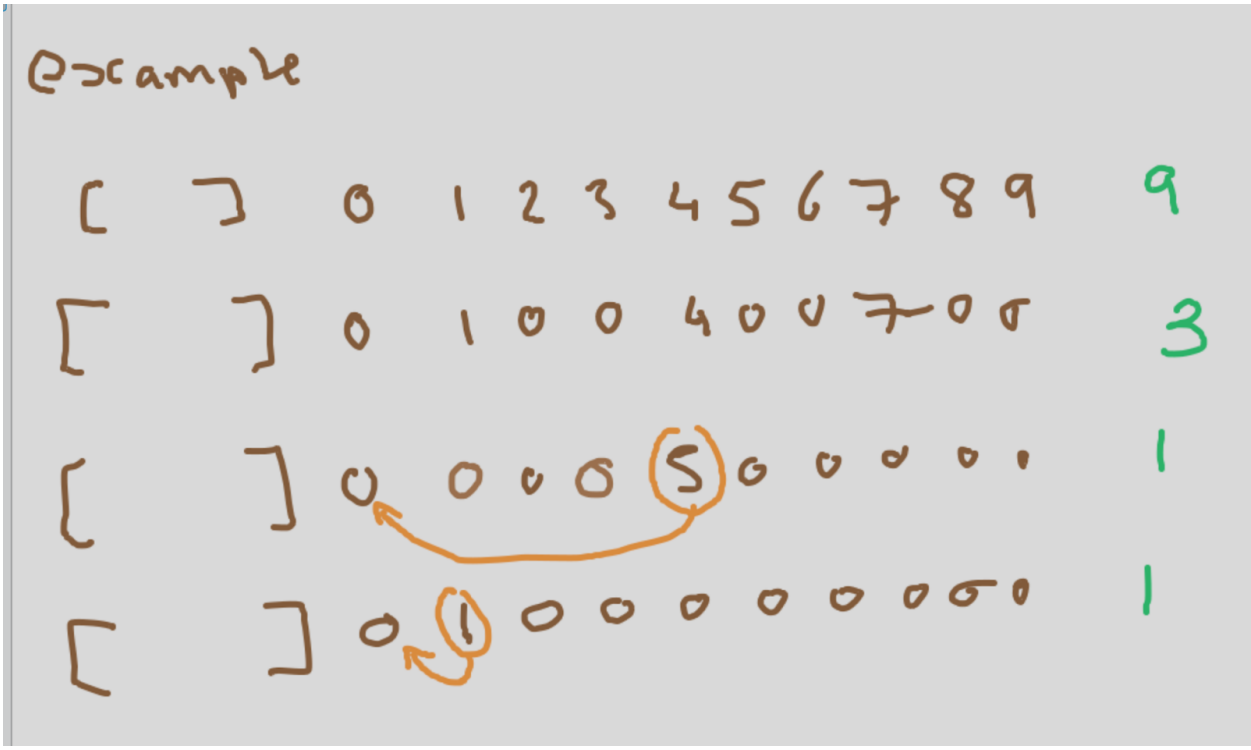
Horizontal

Tell me how many options

```
int tellmehowmanyoptions ( int sudoku [9][9][10]){
        int i, j ,k;
        i = j = k = 0;
        int totaloptions = 0;
        int celloptions = 0;
        for( i = 0; i < 9; i++)
        {
            for( j = 0; j < 9; j++)
            {

                    k = 0; // might be redundant
                    if (sudoku[i][j][k] == 0)
                    {
                        for( k = 1; k < 10; k++)
                        {
                            if(sudoku[i][j][k] != 0)
                            {
                                totaloptions++;
                            }
                        }
                    }
            }
        }
        //std::cout << std::endl;
        //std::cout << "Total options: " << totaloptions << std::endl;

        // loop through the 3D array if the option is not 0 then add it to total
         return totaloptions;
}
```

Check if single in array

```
void checkForSingleInArray(int sudoku[9][9][10]){
        int i, j, k;
        i = j = k = 0;
        int numbercount = 0;
        int value = 0;

        for( i = 0; i < 9; i ++)
        {
            for (j = 0; j < 9; j++)
            {

               if(sudoku[i][j][0] == 0)
               {

                    for(k = 1; k < 10; k++)
                    {
                        if (sudoku[i][j][k] != 0)
                        {
                        value = sudoku[i][j][k];
                        }
                        else
                        {
                            numbercount++;
                        }

                        if (numbercount == 8)
                        {

                        sudoku[i][j][0] = value;
                        sudoku[i][j][value] = 0;
                        }
                    }
                    numbercount = 0;
                    value = 0;
                }
            }
        }
}
```

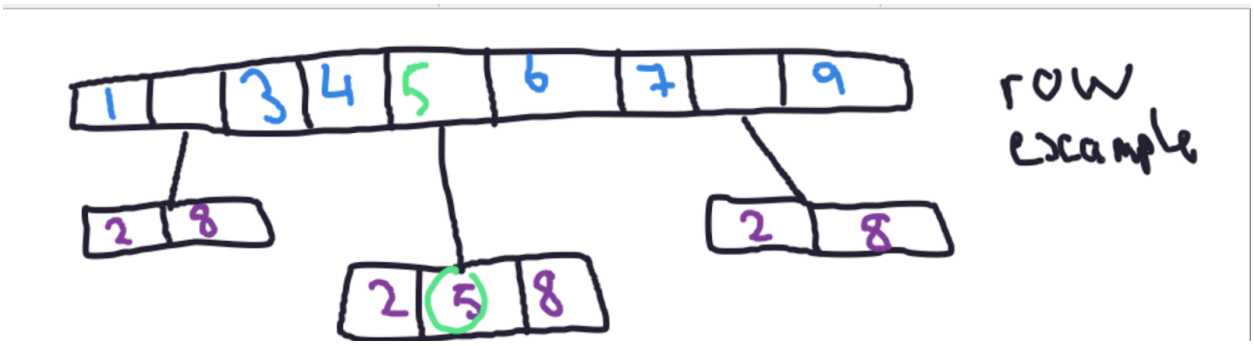


Here are some example cells
We loop through the sudoku and we see how many options each cell has, if a cell only has 1 possible solution then by default it becomes the answer for that cell

Check for single in rows

```
void checkSingleNumberRows(int sudoku[9][9][10]){
   //std::cout << " rows" << std::endl;
        int answerCounter = 0;
        int i, j , k;
        i = j = k = 0;
         int iVal, jVal, kVal;
        iVal = jVal = kVal = 0;
        // i = row, j = col , k is number value

        for(i = 0; i < 9; i++)
        {
            k = 0;
                for( k = 1; k < 10; k++) // loop through values
                {
                     for( j = 0; j < 9; j++) // loop through rows
                        {
                            if(sudoku[i][j][k] != 0) // if an answer is blank as in cant exist
                            {
                            answerCounter++; // add to answer counter
                            }
                            if(sudoku[i][j][k] != 0)
                            {
                                iVal = i;
                                jVal = j;
                                kVal = k;
                                // if  value is non zero store where it occurs
                                // could maybe add a second counter
                            }
                        }
                         if(answerCounter == 1)
                        {
                           //std::cout << "--------we have a single---- for :" << k <<std::endl;
                           //std::cout << " occurs at " << iVal << " " << jVal << " " << kVal << std::endl;
                           //std::cout << "--------changing i & j vals:" <<std::endl;
                           sudoku[iVal][jVal][0] = kVal;
                           jVal = 0;
                           iVal = 0;
                           kVal = 0;
                        }
                        else
                        {
                            jVal = 0;
                            iVal = 0;
                        }
                        //std::cout << " k  = " << k << " occurs : " <<
                        //answerCounter << " times" << "in row : [" << i << "]" << std::endl;
                        answerCounter = 0;
                }
            }

}
```

We check for the single in a row as the example earlier highlighted, same principal



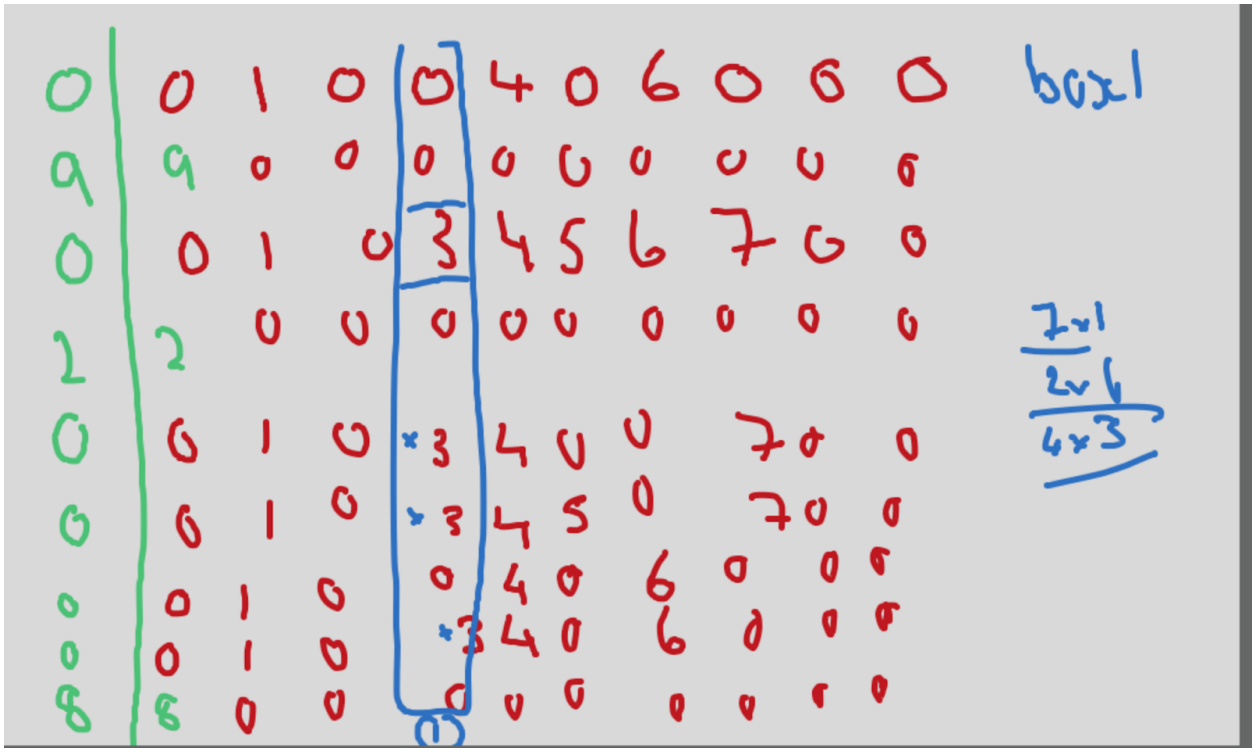
Check single col

```
void checkSingleNumberCols(int sudoku[9][9][10]){
    //std::cout << " col" << std::endl;
    int answerCounter = 0;
        int i, j , k;
        int iVal, jVal, kVal;
        i = j = k = 0;
        iVal = jVal = kVal = 0;

        for(i = 0; i < 9; i++)
        {
            k = 0;
            //std::cout << std::endl;
                for( k = 1; k < 10; k++)
                {
                     for( j = 0; j < 9; j++)
                        {
                            if(sudoku[j][i][k] != 0)
                            {
                            answerCounter++;
                            }
                            if(sudoku[j][i][k] != 0)
                            {
                                iVal = i;
                                jVal = j;
                            }
                        }
                        if(answerCounter == 1)
                        {
                           //std::cout << "--------we have a single---- for :" << k <<std::endl;
                           //std::cout << "--------changing i & j vals:" <<std::endl;
                           //std::cout << " occurs at " << jVal << " " << iVal << " " << kVal << std::endl;
                           sudoku[jVal][iVal][0] = k;
                           jVal = 0;
                           iVal = 0;
                           kVal = 0;
                        }
                        else
                        {
                            jVal = 0;
                            iVal = 0;

                        }
                        //std::cout << " k  = " << k << " occurs : " <<
                        //answerCounter << " times" << "in col : [" << i << "]" << std::endl;
                        answerCounter = 0;
                }
            }
}
```

In this example we have run a few options one time of the sudoku, and eliminated a lot of possibilities now we are going to check the column, this column has only 1 occurrence of 9, the cell which has the 9 must be 9.

Check for single in box

```
void checkSingleNumberBox(int sudoku[9][9][10]){

            int answerCounter = 0;
            int i, j , k;
            i = j = k = 0;

            //std::cout << "box 0 " << std::endl;
           for( i = 1; i < 10; i++)
           {
                if( sudoku[0][0][i] != 0 )
                    answerCounter++;
                if( sudoku[0][1][i] != 0 )
                    answerCounter++;
                if( sudoku[0][2][i] != 0 )
                    answerCounter++;
                if( sudoku[1][0][i] != 0 )
                    answerCounter++;
                if( sudoku[1][1][i] != 0 )
                    answerCounter++;
                if( sudoku[1][2][i] != 0 )
                    answerCounter++;
                if( sudoku[2][0][i] != 0 )
                    answerCounter++;
                if( sudoku[2][1][i] != 0 )
                    answerCounter++;
                if( sudoku[2][2][i] != 0 )
                    answerCounter++;
                //std::cout << i << " occurs " << answerCounter << " times (box 0)" << std::endl;
                if(answerCounter == 1)
                {
                    //std::cout << i << "(box 0) occurs only once" << std::endl;

                    if(sudoku[0][0][i] == i)
                        sudoku[0][0][0] = i;

                    else if (sudoku[0][1][i] == i)
                        sudoku[0][1][0] = i;

                    else if (sudoku[0][2][i] == i)
                        sudoku[0][2][0] = i;

                    else if (sudoku[1][0][i] == i)
                        sudoku[1][0][0] = i;

                    else if (sudoku[1][1][i] == i)
                        sudoku[1][1][0] = i;

                    else if (sudoku[1][2][i] == i)
                        sudoku[1][2][0] = i;

                    else if (sudoku[2][0][i] == i)
                        sudoku[2][0][0] = i;

                    else if (sudoku[2][1][i] == i)
                        sudoku[2][1][0] = i;

                    else if (sudoku[2][2][i] == i)
                        sudoku[2][2][0] = i;


                }
```




```
int testPuzzleA[81] =
{
0,9,0,    0,0,0,   0,0,0,
2,0,0,    0,6,0,   0,8,0,
0,0,8,    0,7,0,   0,0,9,

8,0,0,    0,0,7,   1,0,0,
0,0,0,    0,0,0,   0,0,0,
5,0,0,    0,4,0,   0,2,0,

0,0,0,    0,0,3,   0,9,0,
3,5,0,    0,0,1,   0,0,2,
7,0,0,    0,0,5,   0,4,0
};
```

In box 0 , we have a 9,2,8, and we are checking if there is cell with a unique answer

Try strategies

```
void tryStrategies( int sudoku[9][9][10]){

        horizontal(sudoku);
        vertical(sudoku);
        boxoptions(sudoku);
        checkSingleNumberRows(sudoku);
        checkSingleNumberCols(sudoku);
        checkSingleNumberBox(sudoku);
        checkForSingleInArray(sudoku);
        clearOptionIfSolvedCell(sudoku);
        //tellmehowmanyoptions(sudoku);
        tellMeIfACellIsInvalid(sudoku);
        //printSudoku(sudoku);
}
```

Try strategies takes everything we have made and run them , in the sudoku we have used it can solve one a few cells and eliminate possibilities

```
0 9 0 | 0 0 0 | 0 0 0
2 0 0 | 0 6 0 | 0 8 0
0 0 8 | 0 7 0 | 0 0 9
---------------------
8 0 0 | 0 0 7 | 1 0 0
0 0 0 | 0 0 0 | 0 0 0
5 0 0 | 0 4 0 | 0 2 0
---------------------
0 0 0 | 0 0 3 | 0 9 0
3 5 0 | 0 0 1 | 0 0 2
7 0 0 | 0 0 5 | 0 4 0
```

```
0 9 0 | 0 0 0 | 0 0 0
2 0 0 | 0 6 9 | 0 8 0
0 0 8 | 0 7 0 | 0 0 9
---------------------
8 0 0 | 0 0 7 | 1 0 0
9 0 0 | 0 0 0 | 0 0 0
5 0 0 | 0 4 0 | 9 2 0
---------------------
0 0 0 | 0 0 3 | 0 9 0
3 5 0 | 0 0 1 | 0 0 2
7 0 0 | 0 0 5 | 0 4 0
```

We have managed to solve 3 cells with 9s and eliminate a lot of possibilities

Here is an easy puzzle, which was solved using the basic functions that we have made

```
int testPuzzleB[81] =
{
0,0,0,    0,2,3,   0,7,9,
0,0,4,    5,0,9,   0,0,0,
0,0,0,    7,0,0,   0,0,8,

5,6,0,    0,7,2,   0,0,0,
0,0,2,    0,5,0,   0,8,0,
1,0,7,    6,0,0,   0,0,4,

9,2,0,    0,1,7,   0,0,0,
0,0,0,    3,9,0,   0,0,6,
0,7,0,    0,0,5,   0,9,0
};
```

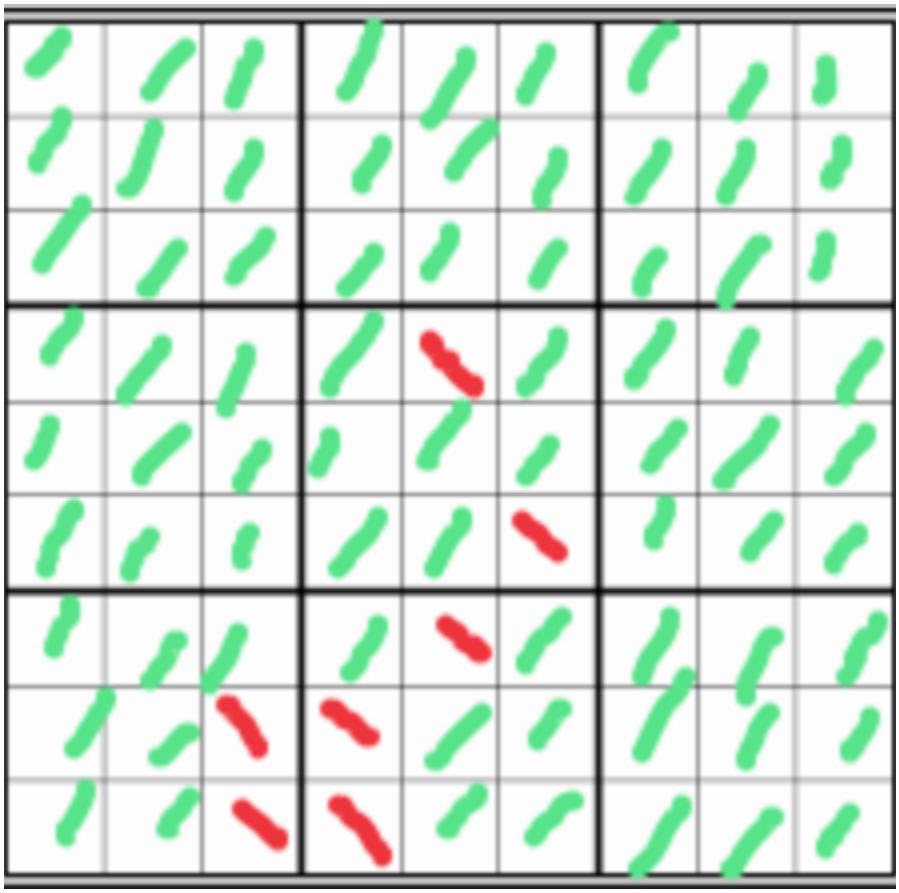
```
 0   0   0   |   0   2   3   |   0   7   9
 0   0   4   |   5   0   9   |   0   0   0
 0   0   0   |   7   0   0   |   0   0   8
-------------------------------------------
 5   6   0   |   0   7   2   |   0   0   0
 0   0   2   |   0   5   0   |   0   8   0
 1   0   7   |   6   0   0   |   0   0   4
-------------------------------------------
 9   2   0   |   0   1   7   |   0   0   0
 0   0   0   |   3   9   0   |   0   0   6
 0   7   0   |   0   0   5   |   0   9   0
```

```
 6   8   5   |   1   2   3   |   4   7   9
 7   3   4   |   5   8   9   |   1   6   2
 2   1   9   |   7   4   6   |   5   3   8
-------------------------------------------
 5   6   8   |   4   7   2   |   9   1   3
 3   4   2   |   9   5   1   |   6   8   7
 1   9   7   |   6   3   8   |   2   5   4
-------------------------------------------
 9   2   6   |   8   1   7   |   3   4   5
 8   5   1   |   3   9   4   |   7   2   6
 4   7   3   |   2   6   5   |   8   9   1
```

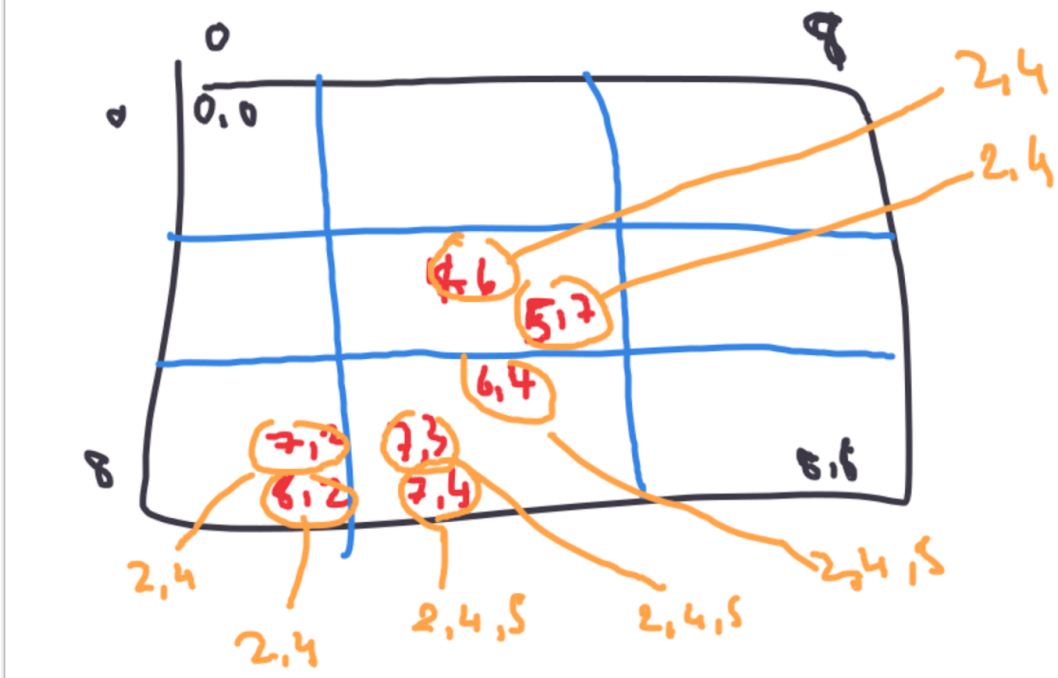
Now that our basic strategies may not work for every sudoku, lets look at the basic principal

```
bool backtracker ( int sudoku[9][9][10]){

    struct cellNumberstruct lowest = checkhowmanyoptionsincell(sudoku);
    std::vector<int> options;
    options = optionsVector(sudoku, lowest);
    int copySudoku[9][9][10];
    int i = 0;
    bool answerforComplete = 0;
    bool answerForInavlid = 1;

    makeCopy(sudoku, copySudoku);


    answerforComplete = isSolved(sudoku);

    if ( answerforComplete == 1)
    {
        // maybe make it global
        printSudoku(copySudoku);
        std::cout << "Finished" << std::endl;
        return 1;
    }

    answerForInavlid = isValid(copySudoku);
    if ( answerForInavlid == 0)
    {
        return 0;
    }

    for( i = 0; i < options.size();i++)
    {

    makeCopy(sudoku, copySudoku);
    copySudoku[lowest.row][lowest.col][0] = options[i];
    tryStrategies(copySudoku);
        if(backtracker(copySudoku) == 1)
        {
        //printSudoku(copySudoku);
        return 1;
        }
    }
    return 0;
}
```

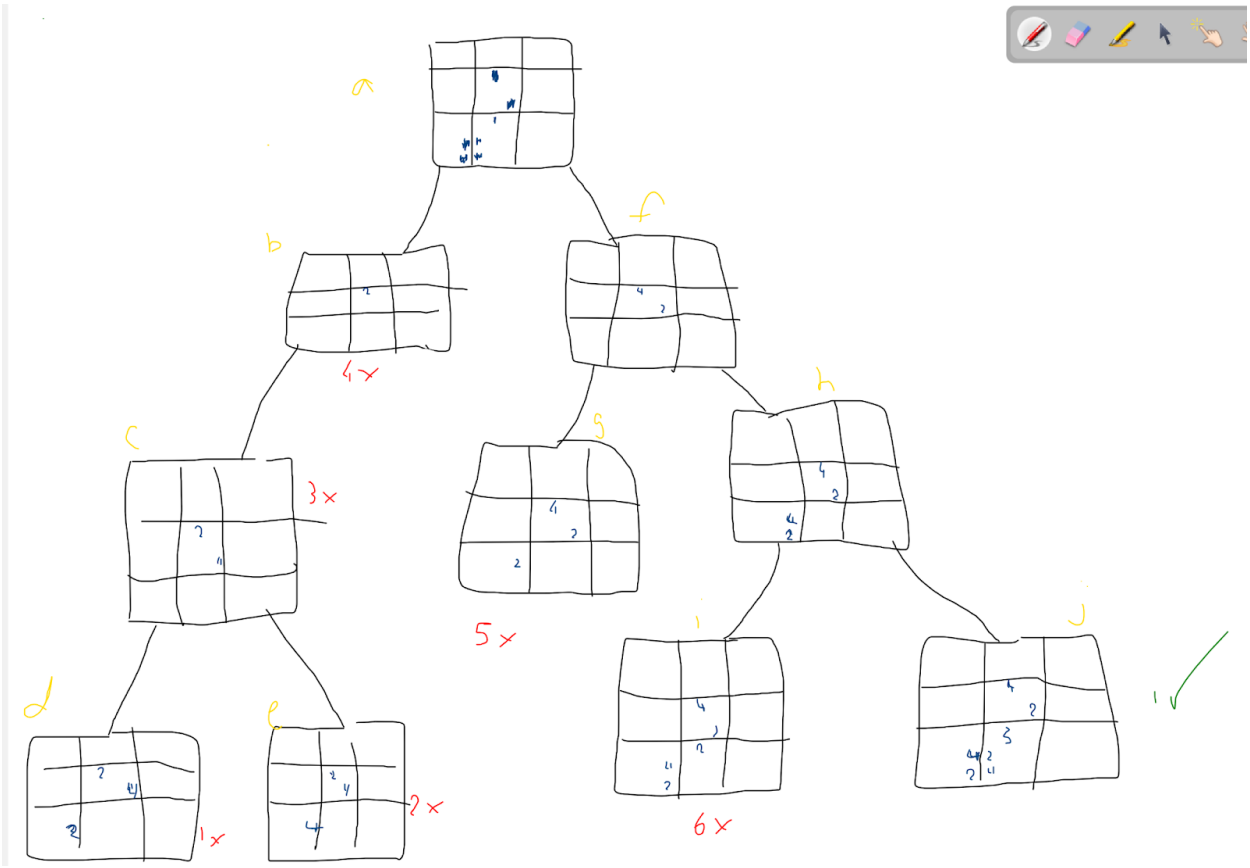
This is just an example



Lets assume every red cell is not solves
Lets say the the boxes with 2 options are 2,4
And the boxes with the option is 2,4,5



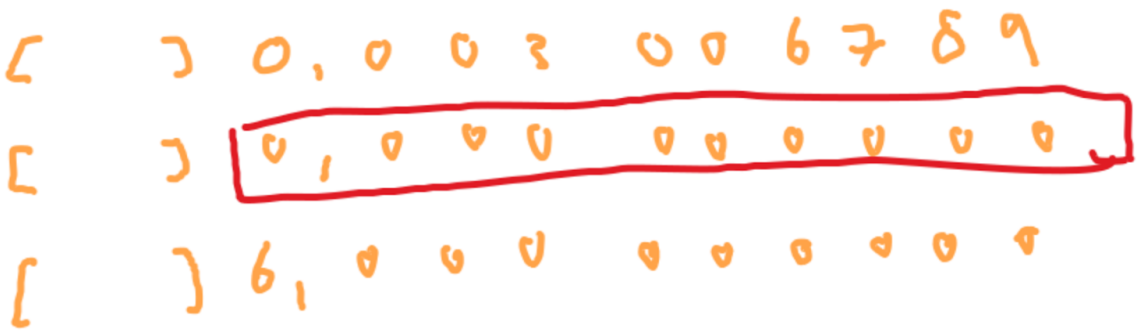
4, 6 is the lowest cell in the middle, now we try the lowest cell option which is 2, now we check is 2 valid if it is lets, try the next cell which is 5,7 , and if that is still valid we move on to the next cell with the lowest number of options and try the lowest option , once the solution is invalid we stop and go up one level



Lets take an almost sudoku solved sudoku
Lets imagine we start from the lowest cell which isnt solved from

```
bool isValid(int sudoku[9][9][10]){
    bool startState = true; // 1 is true

    int cellOption = 0;
    int i, j , k;
    i = j = k = 0;

    for(i = 0; i < 9 ; i++)
    {
        for( j = 0; j < 9; j++)
        {
            if(sudoku[i][j][0] == 0)
            {
                for(k = 1; k < 10; k++)
                {
                    cellOption+= sudoku[i][j][k];
                }
                if (cellOption == 0)
                {
                //std::cout << "Cell " << i << j << " is not valid" << std::endl;
                return false;
                }
                else
                {
                cellOption = 0;
                }

            }
        }
    }
        return startState;
}
```

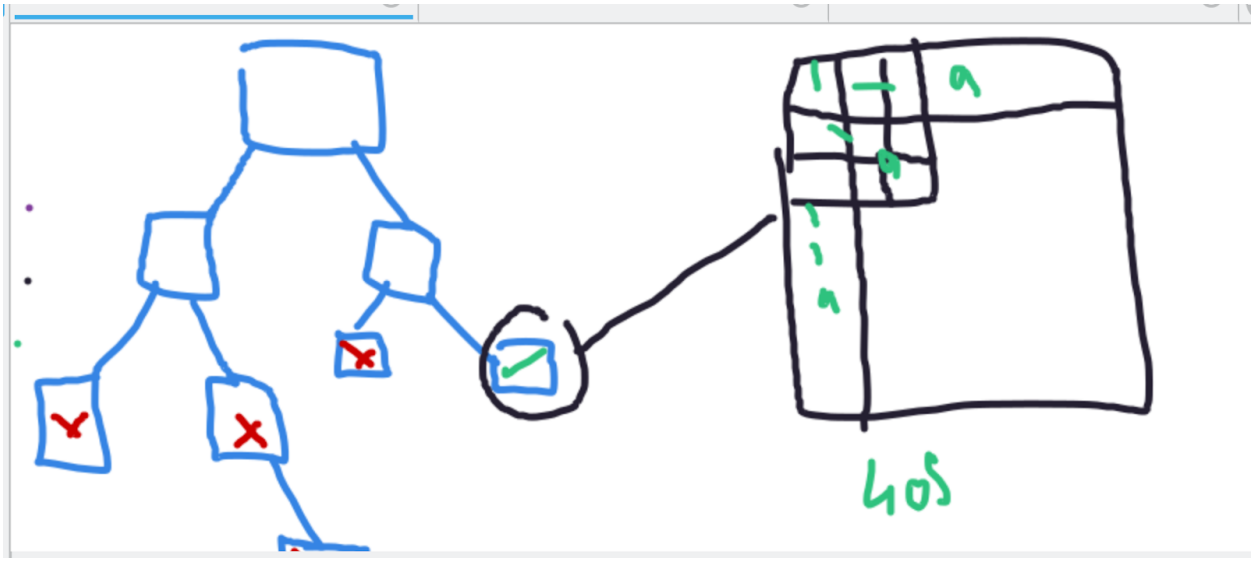
How we check if the sudoku is valid is, by looking at every option, if there was a cell that was empty and had no solutions



In this example we have 1 solved cell which is a 6 , one empty cell with 5 options and one empty cell with no options at this point we know the sudoku is invalid

```
bool isSolved(int sudoku[9][9][10]) { // very lazy poor method
    bool startState = 0; // 1 is true

    //int cellOption = 0;
    int i, j , k;
    i = j = k = 0;
    int sum  = 0;

    int oneCount = 0;
    int twoCount = 0;
    int threeCount = 0;
    int fourCount = 0;
    int fiveCount = 0;
    int sixCount = 0;
    int sevenCount = 0;
    int eightCount = 0;
    int nineCount = 0;

     for(i = 0; i < 9 ; i++)
    {
        for( j = 0; j < 9; j++)
        {
            if(sudoku[i][j][0] == 0)
            {
               return false;
            }
        }
    }

    for ( i = 0; i < 9; i++)
    {
        for(j = 0; j < 9; j++ )
        {
        sum+=sudoku[i][j][0];
        }
```

To check if the sudoku is solved we see if there is a 1- 9 in every row, col , box, and the sum is 405.



So we understand how the backtracker basically works and we understand how to check if the sudoku is solved or invalid

```
struct cellNumberstruct {
    int row;
    int col;
};
```

```
struct cellNumberstruct checkhowmanyoptionsincell(int sudoku[9][9][10]){
        // this assumes sudoku hasnt been filled
        int i,j,k;
        i = j = k = 0;
        int options = 0; // could be a better way than brute force
        int answerToOptions = 9; // assumes there must be solutions left
        cellNumberstruct mystructure;
        mystructure.row = 9;
        mystructure.col = 9;

        for(i = 0; i < 9; i++)
        {
            for( j = 0; j < 9; j++)
            {
                for ( k = 1; k < 10; k++)
                {
                    if(sudoku[i][j][0] == 0)
                    {
                        if(sudoku[i][j][k] != 0)
                        {
                            options++;
                        }
                    }
                }
                    if(sudoku[i][j][0] == 0)
                    {
                    //std::cout << " cell i , j " << i << " " << j << " has " << options << std::endl;
                        if(options < answerToOptions )
                        {
                            answerToOptions = options;
                            mystructure.row = i;
                            mystructure.col = j;
                        }
                    }
                options = 0;
            }
        }
        //std::cout << " cell row = " << mystructure.row << std::endl;
        //std::cout << " cell col = " << mystructure.col << std::endl;
        return mystructure;
}
```

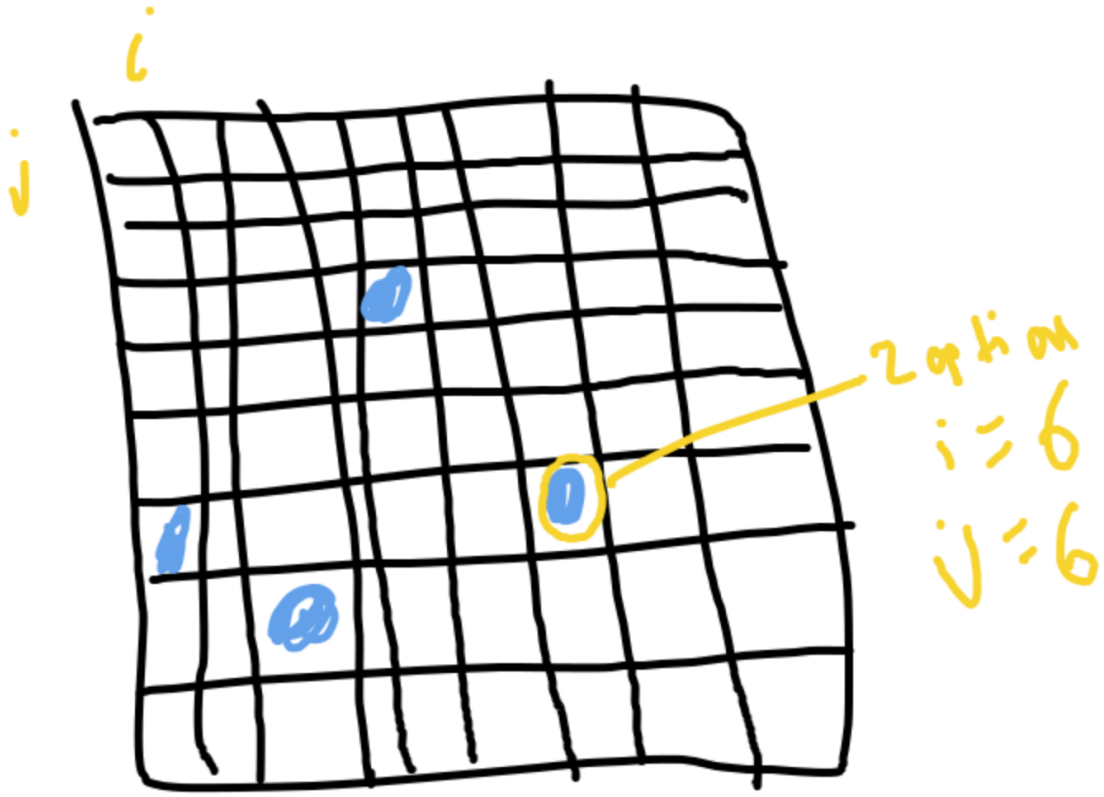
So I could have just started trying the lowest set of options from 1-9 from the leftmost upper cell to the bottom

```
int testPuzzleA[81] =
{
0,9,0,    0,0,0,   0,0,0,
2,0,0,    0,6,0,   0,8,0,
0,0,8,    0,7,0,   0,0,9,

8,0,0,    0,0,7,   1,0,0,
0,0,0,    0,0,0,   0,0,0,
5,0,0,    0,4,0,   0,2,0,

0,0,0,    0,0,3,   0,9,0,
3,5,0,    0,0,1,   0,0,2,
7,0,0,    0,0,5,   0,4,0
};
```

Look at the top row

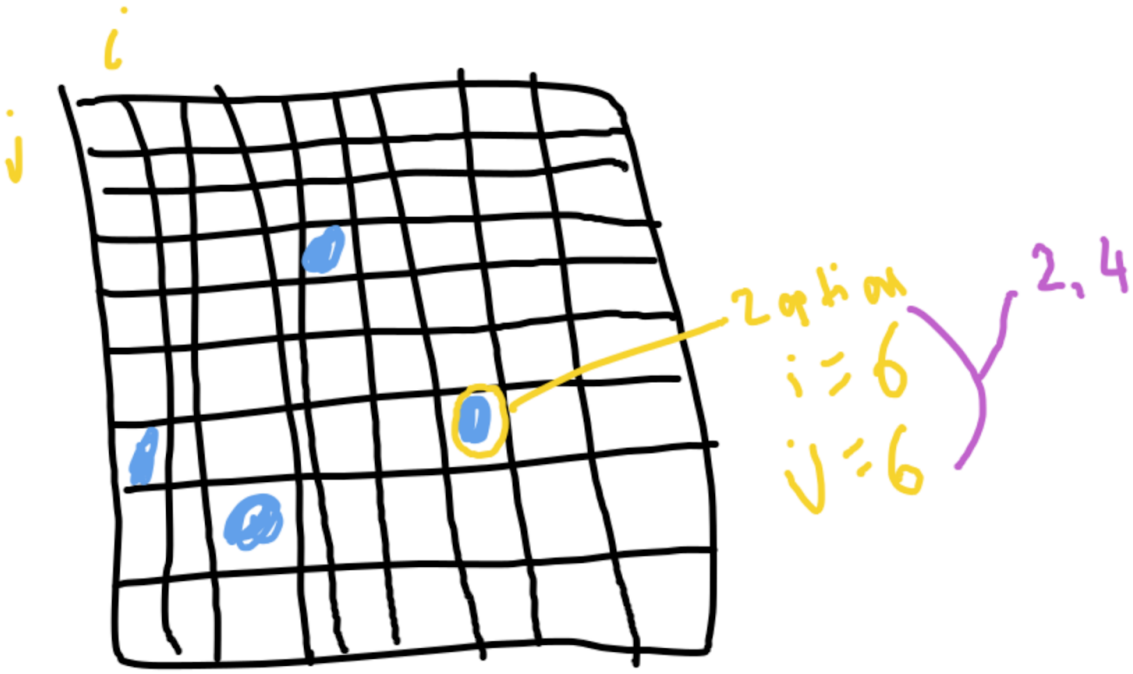
0,9,0, 0,0,0, 0,0,0
We could try making the left most cell the lowest number
1,9,0, 0,0,0, 0,0,0
We could try making the left most cell the lowest number
1,9,2, 0,0,0, 0,0,0
This would work but be very inefficient

What i want to do is scan the sudoku and see how many options each cell has, I want to take a note of which cell have the lowest number of option



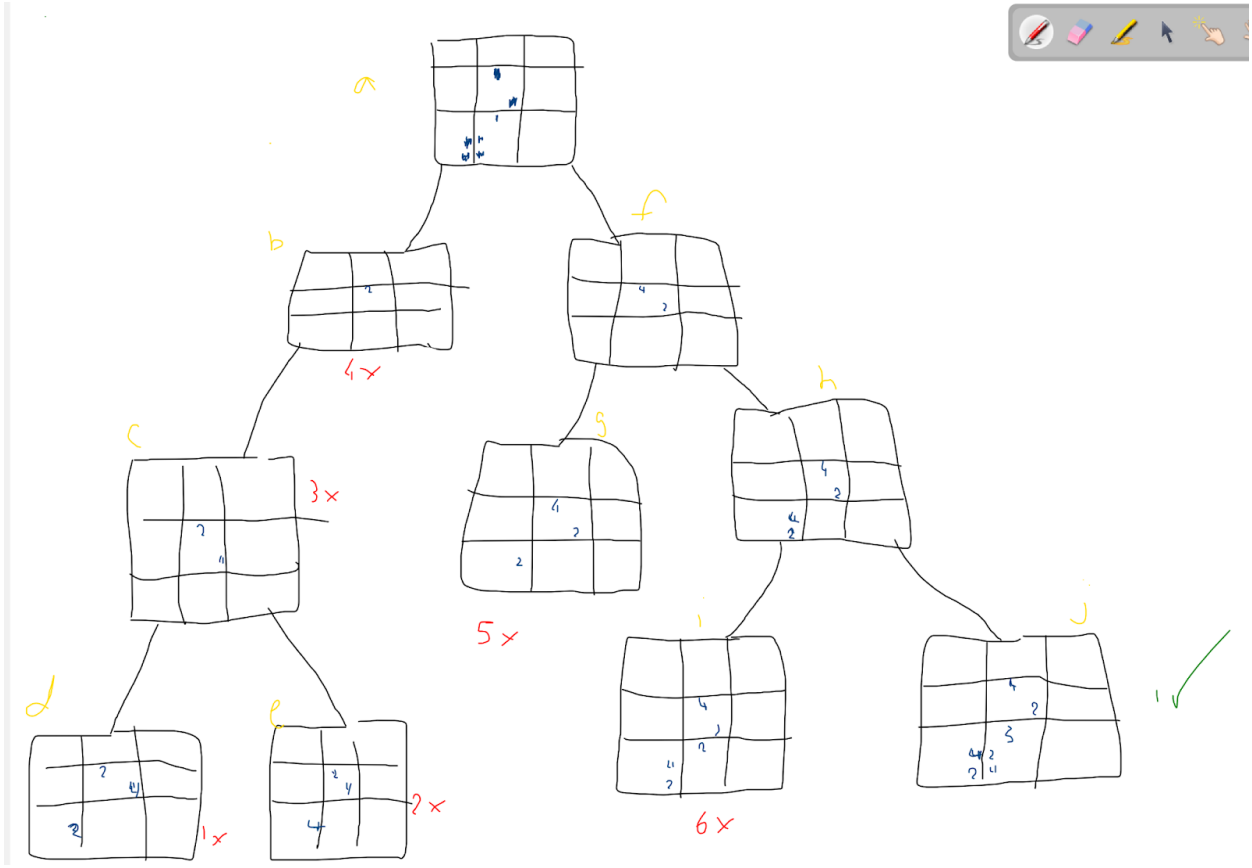
So we see cell 6,6, has the lowest number of options and we send back the number 6,6,

```
std::vector<int> optionsVector(int sudoku[9][9][10], struct cellNumberstruct lowestCell){
    int i, j, k = 0;
    std::vector<int>options;
     // int options = {0,0,0,0,0,0,0,0,0};

    i = lowestCell.row;
    j = lowestCell.col;
    ////std::cout << " cell row = " << lowestCell.row << std::endl;
    //std::cout << " cell col = " << lowestCell.col << std::endl;
    // vector is the wrong way round and this function doesnt accoutn for finshed sudoku
            for( k = 1; k < 10; k++)
            {
                    if(sudoku[i][j][k] != 0)
                    {
                    //std::cout << "pushing back " << sudoku[i][j][k] << std::endl;
                    options.push_back(sudoku[i][j][k]);
                    }
            }
         //std::cout << " Cell " << lowestCell.row << " " << lowestCell.col << " has " << options.size() << " options" << std::endl;
        //int counter = 0;
        //for( counter; counter <= options.size()-1; counter++)
        //{
        //std::cout << options[counter] << std::endl;

        //}

         return options;
}
```



So now we have the lowest cell, we go to it, and we then work out options are available and save them as a vector, and pass that to our backtracker

```
void makeCopy(int sudoku[9][9][10], int copySudoku[9][9][10]){

    int i, j, k;
    i = j = k = 0;

    for( i = 0; i <= 8 ; i++)
    {
        for( j = 0; j <= 8; j++)
        {
            for( k = 0; k <= 10; k++ )
            {
                copySudoku[i][j][k] = sudoku[i][j][k]; // turn this is into a func
            }
        }
    }
}
```



So the make copy takes the original, makes a copy then tries the lowest option for the cell with the lowest number and runs until it solves

```
void solveSudoku ( int sudoku[9][9][10]){
    cellNumberstruct solutionStructure;
    solutionStructure.row = 0;
    solutionStructure.col = 0;
    bool answer = 0;
    int possibleOptionsStart = 0;
    possibleOptionsStart = tellmehowmanyoptions(sudoku);
    int possibleLoop = 0;
    possibleLoop =  tellmehowmanyoptions(sudoku);
    possibleLoop++;
    int loopcount = 0;

    while ( possibleOptionsStart < possibleLoop && possibleOptionsStart != 0)
    {
        possibleLoop = possibleOptionsStart;
        tryStrategies(sudoku);
        possibleOptionsStart = tellmehowmanyoptions(sudoku);
        loopcount++;
    }


    //printSudoku(sudoku);
    bool isTheSudokuSolved = 0;
    isTheSudokuSolved = isSolved(sudoku);
    //std::cout << "The answer is " << isTheSudokuSolved << std::endl;
    //bool solved



        solutionStructure = checkhowmanyoptionsincell(sudoku);

        //std::cout << "-----------" << std::endl;
       // std::cout << "-----------" << std::endl;
        //std::cout << "we need to back test " << std::endl;
        optionsVector(sudoku, solutionStructure);
        int finalAnswer = backtracker(sudoku);


        //std::cout << isValid(sudoku) << std::endl;




}
```

Here is the solve sudoku, what we do is try , the try strategies functions until we can no longer eliminate possibilities, once we can no longer eliminate options we need to backtest, we go into the backtest then try to solve the sudoku, and it should solve based on the logical rules of the sudoku, as long as you are valid you are solvable.

Check if empty

```
void checkIfEmpty(int sudoku[9][9][10]){
// sum up every cell and if the answer is zero then the cell is empty
// if the sudoku is empty set the first cell to 1, (not sure why My program did not like an empty sudoku)
// could make any or cell random if you wanted by the first cell to 1 is easiest
    int i, j;
    i = j = 0;
    int total = 0;

    for(i = 0; i < 9; i++)
    {
        for(j = 0; j < 9; j++)
        {
            total = sudoku[i][j][0];
        }
    }

    if ( total == 0)
    {
        //std::cout << "Sudoku is empty " << std::endl;
        sudoku[1][1][0] = 1;
    }
    else
    {
        //std::cout << "Sudoku is not empty" << std::endl;
    }


}
```

For some reason my program didnt like an empty sudoku , I couldnt be bothered to work out why so if the sudoku is empty I would just set the first cell to 1

Check if invalid

```
bool initialSudokuCheck ( int sudoku[9][9][10]){

    bool startState = 1;
    int i, j , k;
    i = j = k = 0;

    int oneCount = 0;
    int twoCount = 0;
    int threeCount = 0;
    int fourCount = 0;
    int fiveCount = 0;
    int sixCount = 0;
    int sevenCount = 0;
    int eightCount = 0;
    int nineCount = 0;

   // std::cout << "Check for rows " << std::endl;
    for(i = 0; i < 9 ; i++)
    {
        for( j = 0; j < 9; j++)
        {
            if(sudoku[i][j][0] == 1)
               oneCount++;
            else if (sudoku[i][j][0] == 2)
                twoCount++;
            else if (sudoku[i][j][0] == 3)
                threeCount++;
            else if (sudoku[i][j][0] == 4)
                fourCount++;
            else if (sudoku[i][j][0] == 5)
                fiveCount++;
            else if (sudoku[i][j][0] == 6)
                sixCount++;
            else if (sudoku[i][j][0] == 7)
                sevenCount++;
            else if (sudoku[i][j][0] == 8)
                eightCount++;
            else if (sudoku[i][j][0] == 9)
```

If the sudoku is invalid then it wouldnt be worth solving, in this function I just hard coded some checks to make sure there isnt any of the same number in a row, col, or box.

Could generate random,
Tell me if a sudoku has a unique solution

Could try to scan a sudoku in
Could create a visual solver and log output

Some names were not as explicit as they could have been
Some extra variables may not have been used