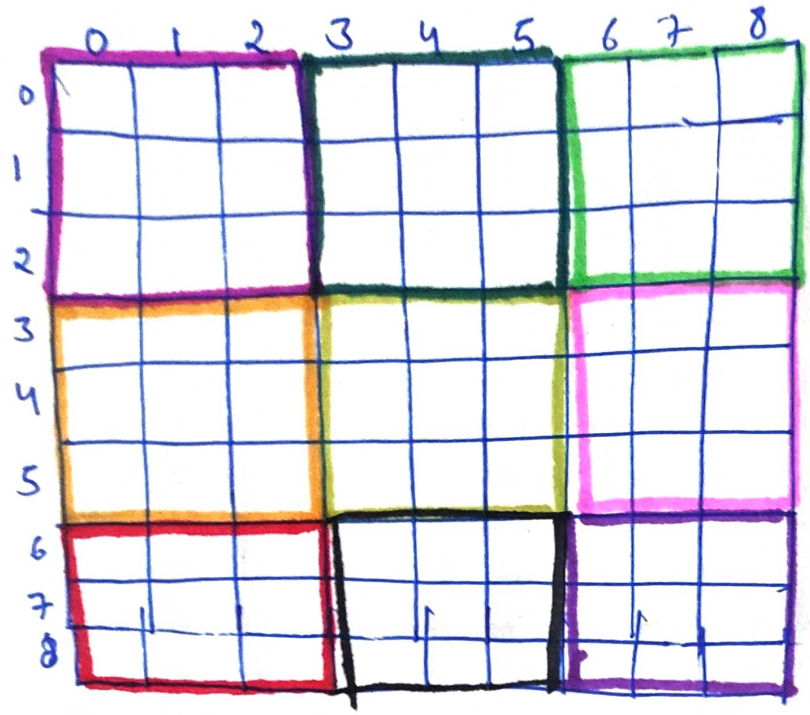
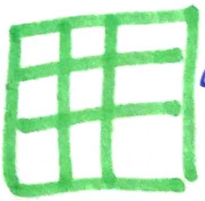
9 Valid Sudoku



Question Type : Matrix

Data structures : Hashmap , Hashset .

Conditions for validity of sudoku .

- Each row can contain 1-9 without repeatition
- Each col can contain 1-9 without repeatition
- Each grid box (3x3)  This guy also can contain 1-9 without repeatition.

- Partially filled is valid too. (even iff unsolvable).

- Only filled cells needs to be checked .

(Corner case)

if (current == '.') => skip.

Bruteforce.

→ For storing and calculating count, there are '2' approaches

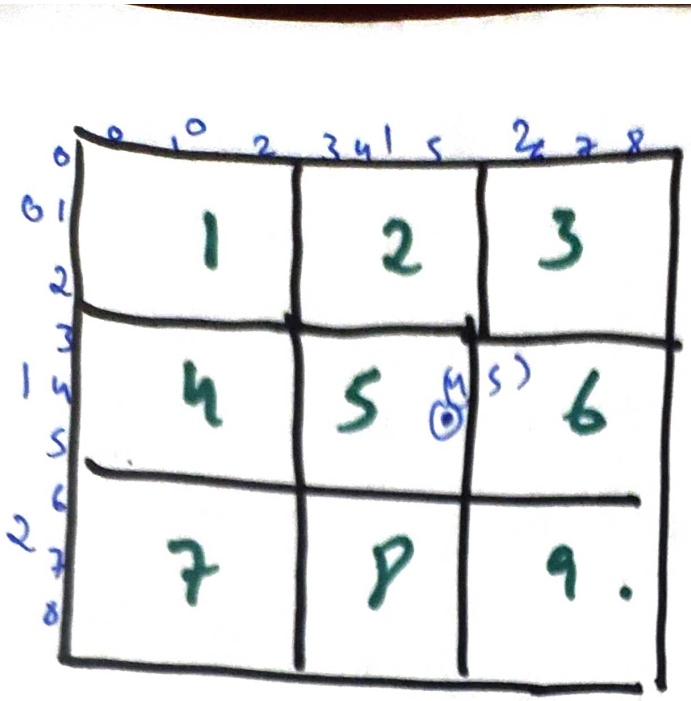
1.) Use loops and determine the counts of all elements ~~is duplicate~~ and then store it somewhere.

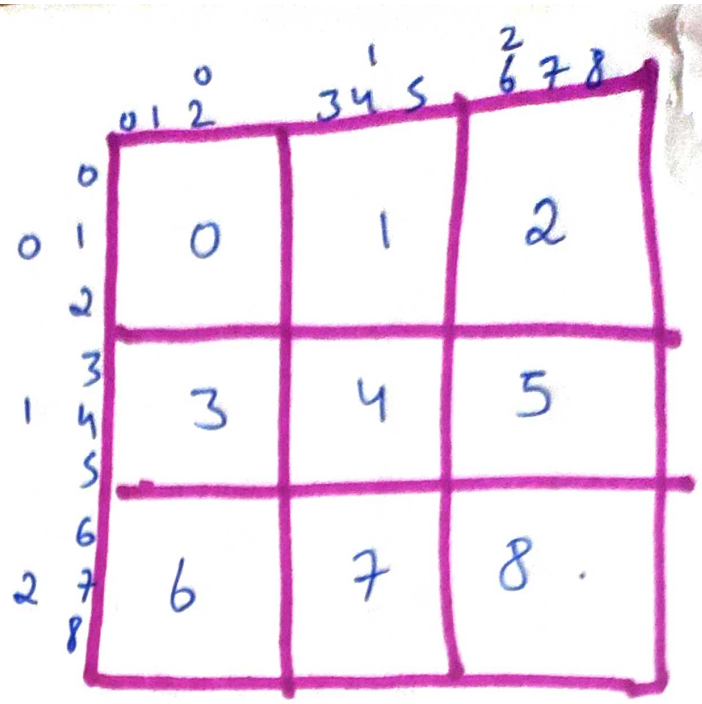
2.) Use hashmap or hashset for efficient find
Hashset would be better there but we would explore the hashmap route for just checking of how that works.

3.) For calculating count we have 9 rows, 9 cols, 9 gridboxes so for each of them we need to mark counts and check for validity at end.

- For rows and cols, its pretty straightforward
- For grid boxes its a bit hard.

Each grid has to be identified in some way so that elements are identified in particular of which grid they are.





If we remove the smaller boxes and consider each grid box as its own; we can assign indexes to the grids in this way.

For rows : 0, 1, 2

For cols : 0, 1, 2.

So if any element has index (4, 5)
what would be its index in terms to grid box.

As we have identified that each gridbox is made of 3×3 matrix; there are '3' rows and '3' cols. in each box

∴ If we divide the indexes by 3 we would get position in terms of whole box.

But why we are doing all these things
because if we know elements index in terms of gridbox it will be easier to store count of each box.

The issue is the column number uniquely identifies the grid but the rows number still won't give you the grid. **what do you mean!?**

You will say Each index is already getting identified according to gridbox but the original indexs of each grid box should be (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

~~for~~ ~~root~~ ~~(0, 1, 2)~~ ~~and~~ not (0, 1, 2)

~~for cols~~ for rows.

~~Why~~ we need to scale that number to form the grid index accurately for row.

i.e new gridRowindex = $\left(\frac{i}{3}\right)$ * scalingfactor

(i') = $\left(\frac{i}{3}\right)$ * 3

because each grid is 3×3 and if its resetting to (0,1,2) we need to use 3 to upscale it.

∴ Whole box index will be calculated in similar way as we calculate index of element in normal matrix = $(i+j) = (i'+j')_2$

$= (i/3) * 3 + (j/3)$

$\therefore$ grid $X = \frac{i}{3}$, grid $Y = \frac{j}{3}$.

$(4,5) \equiv \left(\frac{5}{3}, \frac{4}{3}\right) \equiv (1,1)$.

But this has an issue this will correctly locate the element in the grid if we expand the element calculated grid indexs of whole 9x9 grid.

$(i/3, j/3)$ of table

	0	1	2	3	4	5	6	7	8
0	(0,0)	(0,0)	(0,0)	(0,1)	(0,1)	(0,1)	(0,2)	(0,2)	(0,2)
1	(0,0)	(0,0)	(0,0)	(0,1)	(0,1)	(0,1)	(0,2)	(0,2)	(0,2)
2	(0,0)	(0,0)	(0,0)	(0,1)	(0,1)	(0,1)	(0,2)	(0,2)	(0,2)
3	(1,0)	(1,0)	(1,0)	(1,1)	(1,1)	(1,1)	(1,2)	(1,2)	(1,2)
4	(1,0)	(1,0)	(1,0)	(1,1)	(1,1)	(1,1)	(1,2)	(1,2)	(1,2)
5	(1,0)	(1,0)	(1,0)	(1,1)	(1,1)	(1,1)	(1,2)	(1,2)	(1,2)
6	(2,0)	(2,0)	(2,0)	(2,1)	(2,1)	(2,1)	(2,2)	(2,2)	(2,2)
7	(2,0)	(2,0)	(2,0)	(2,1)	(2,1)	(2,1)	(2,2)	(2,2)	(2,2)
8	(2,0)	(2,0)	(2,0)	(2,1)	(2,1)	(2,1)	(2,2)	(2,2)	(2,2)

Note:
We didn't multiplied cols width scaling factor because cols were correct.

For example if we added (1 , 2) it would not point to right grid box index
↓ row ↓ col

= 1 + 2 = 3 ≠ 5

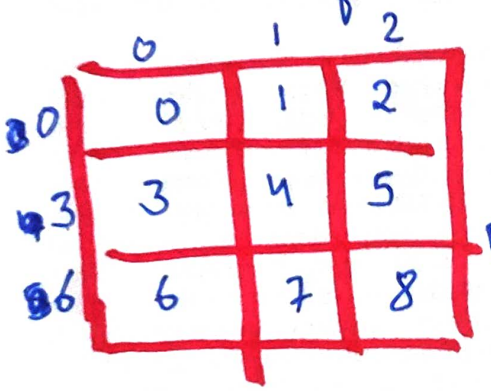
But if we add (1 * 3 + 2).
↓row ↓scaling factor ↑col

= 5 (✓)

Also if we did the reverse

(1 * + 2 * 3)

= 1 + 6 = 7 (✗)



← imagine as it as this for ease.

Code

```
vector < unordered_map <int, int >> rows (9);
vector <unordered_map <int, int>> cols (9);
vector <unordered_map< int, int>>> boxes(9);
board [ ][ ] // matrix.

// For rows validation

for i to 9
    for j to 9
        curr = board [i][j]
        if curr == '.'
            continue;
        rows [i] [curr] ++;

// For cols validation.

for i to 9
    for j to 9
        curr = board [j] [i]
        if if curr == '.'
            continue;
        cols [i] [curr] ++;
```

// For checking grid bones.

```
boxIndex = (i/3) * 3 + j/3
curr = board[i][j];
if curr == '.'
    continue;
bones[boxIndex][curr]++;
```

$T(n): O(n^2)$

$S(n) = O(18n)$

// End check for the count.

```
for i to 9
    for it : rows[i]
        if it.second > 1
            return false.
    for it : cols[i]
        if it.second > 1
            return false
    for it : bones[i]
        if it.second > 1
            return false.

return true;
```

Optimised.

1:) In Brute force we saw there were many repeatitions done.

Taking '3' vectors of a hashmaps each which is a lot of memory

Now we would try to reduce that by using temp unordered sets for ~~each~~ ~~row~~ rows and cols and vector of unordered sets for boxes.
(because all boxes need to be checked ~~for~~ and there is no indexing for which is the current box).

// For rows

```
for i to 9
    unordered-set <char> rows
    for j to 9
        char curr = board[i][j];
        if curr == '.'
            continue;
        if rows.count(curr)
            return false;
        rows.insert(curr)
```

← return true;

```
// For cols.
    for i to 9
        unordered_set<char> col.
        for j to 9.
            curr = board[j][i]
            if curr == '.'
                continue;
            if col.count(curr)
                return false;
            col.insert(curr;

// For boxes.
    vector<unordered_set<char>> boxes(9);
    for i to 9
        for j to 9.
            curr = board[i][j];
            boxIndex = (i/3) * 3 + j/3
            if curr == '.'
                continue;
            if boxes[boxIndex].count(curr)
                return false.
            boxes[boxIndex].insert(curr);
    return true;
```