2

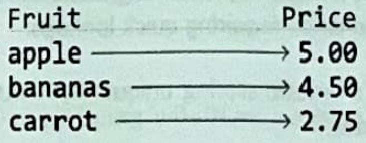
Hash Maps and Sets

Introduction to Hash Maps and Sets

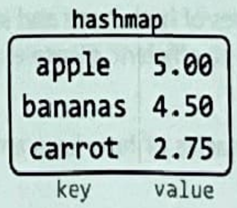
Picture working in a grocery store. A customer asks for the price of a fruit. If you just have a paper list of all the fruits, you would need to look through it to find the specific fruit, which can be time-consuming. However, by memorizing the list, you can know the prices of all fruits instantly, enabling you to promptly give the correct price to the customer. This is similar to how hash maps work, enabling quick access to information.

Hash maps

A hash map – also known as a hash table or dictionary, depending on the language – is a data structure that pairs keys with values. Let's illustrate this using the example of fruits and their prices:



Having a mental map of fruit prices is conceptually similar to being able to instantly access fruit prices using a hash map, where the fruit name is the key, and its price is the value. When we look up a price using the fruit's name as the key, the hash map will immediately return its price:



Hash maps are incredibly efficient for lookups, insertions, and deletions, as they typically perform these operations in constant time: $O(1)$. They're one of the most versatile, widely-used data structures in computer science, used for tasks such as counting the frequency of elements, caching data, and more.

Properties of hash maps

- Data is stored in the form of key-value pairs.
- Hash maps don't store duplicates. Every key in a hash map is unique, ensuring each value can

be distinctly identified and accessed.

- Hash maps are unordered data structures, meaning keys are not stored in any specific order.

Time complexity breakdown
Below, n denotes the number of entries in the hash map.

Operation	Average case	Worst case	Description
Insert	$O(1)$	$O(n)$	Add a key-value pair to the hash map.
Access	$O(1)$	$O(n)$	Find or retrieve an element.
Delete	$O(1)$	$O(n)$	Delete a key-value pair.

In coding interviews, we generally consider hash map operations to have a fast average time complexity of $O(1)$, as opposed to their worst-case complexities. This is based on the assumption that an efficient hash function minimizes collisions [1]. However, in the worst-case scenario where a poorly optimized hash function results in frequent collisions, the time complexity can deteriorate to $O(n)$, necessitating a linear search through all entries.

Hash sets
Hash sets are a simpler form of hash maps. Instead of storing key-value pairs, they store only the keys. Using the grocery store analogy, a hash set is like having a mental checklist of fruits without their prices. It's useful for quickly checking the presence or absence of an item, like checking whether a particular fruit is in stock.

When to use hash maps or sets
Common use cases of hash maps include implementing dictionaries, counting frequencies, storing key-value pairs, and handling scenarios requiring quick lookups.

Common use cases of hash sets include storing unique elements, marking elements as used or visited, and checking for duplicates.

In the description of a problem, pay attention to keywords like "frequency", "unique", "map", "dictionary", or "fast lookup", because these often indicate that hash maps or sets could be useful.

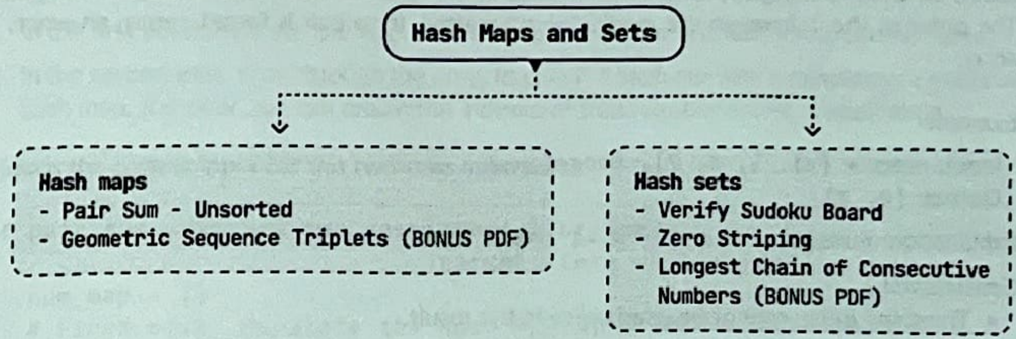
Essential concepts for mastering hash maps and sets
This chapter discusses the practical uses of hash maps and sets. For a more complete understanding of how they work and why they're so efficient, please explore topics that go beyond the scope of this chapter, such as:

- Hash functions: explore the intricacies of how keys are mapped to specific values in a hash table [2].
- Collision and collision-handling techniques: understand methods like chaining, or open addressing for resolving hash collisions [1].
- Load factors and rehashing: these make it easier to understand how hash tables grow and change size [3].

Real-world Example

Web browser cache: Hash maps and sets are used everywhere in real-world systems. A classic example of hash maps in action is in caching systems within web browsers. When you visit a website, your browser stores data such as images, HTML, and CSS files in a cache so it can load much faster on future visits.

Chapter Outline



Practice these problems on our online code editor ⟶ bit.ly/run-code

To receive the bonus PDF, sign up for our Coding Interview Patterns newsletter by using the link below:

| bit.ly/coding-patterns-pdf

References

[1] Hash collisions: https://en.wikipedia.org/wiki/Hash_collision

[2] Hash functions: https://en.wikipedia.org/wiki/Hash_function

[3] Load factor and rehashing: https://www.scaler.com/topics/data-structures/load-factor-and-rehashing/

Pair Sum - Unsorted

Given an array of integers, return the indexes of any two numbers that add up to a target. The order of the indexes in the result doesn't matter. If no pair is found, return an empty array.

Example:

Input: `nums = [-1, 3, 4, 2], target = 3`
Output: `[0, 2]`

Explanation: `nums[0] + nums[2] = -1 + 4 = 3`

Constraints:

- The same index cannot be used twice in the result.

Intuition

A brute force approach is to iterate through every possible pair in the array to see if their sum is equal to the target. This is the same as the brute force solution described in the *Pair Sum - Sorted* problem, which has a time complexity of $O(n^2)$, where n is the length of the array. We could also sort the array and then perform the two-pointer algorithm used in *Pair Sum - Sorted*, which would take $O(n \log(n))$ time due to sorting. Let's see if we can find an even faster solution.

Complement

We're asked to find a pair `(x, y)` such that `x + y == target`. In this equation, there are two unknowns: x and y. An important observation is that if we know one of these numbers, we can easily calculate what the other number should be.

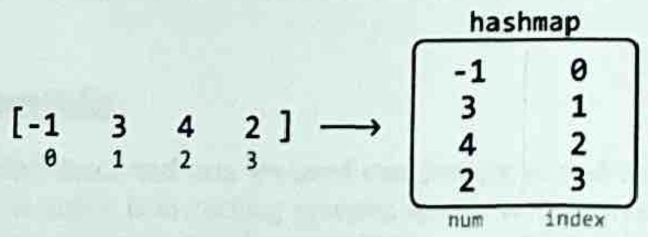
For each number x in `nums`, we need to find another number y such that `x + y = target`, or in other words, `y = target - x`. We can call this number the **complement** of x.

Keep in mind, we need to return the indexes of the pair of numbers, not the pair itself. So, we'll need a way to find a number's complement as well as its index.

One way we could do this is to loop through the array to find each number's complement and corresponding index. But this takes $O(n^2)$ time since we'd need to do a linear traversal to search for each number's complement. Instead, we'd like an efficient way to determine the index of any number in the array without needing to search the array. Is there a data structure that can help with this?

Hash map

A hash map works great because we can store and look up values in $O(1)$ time. Each number and its index can be stored in the hash map as key-value pairs:



This allows us to retrieve the index of any number's complement efficiently. Notice that duplicate numbers don't need to be considered here since only one valid pair needs to be found.

The most intuitive way to incorporate a hash map is to:

1. In the first pass, populate the hash map with each number and its corresponding index.
2. In the second pass, scan through the array to check if each number's complement exists in the hash map. If it does, we can return the indexes of that number and its complement.

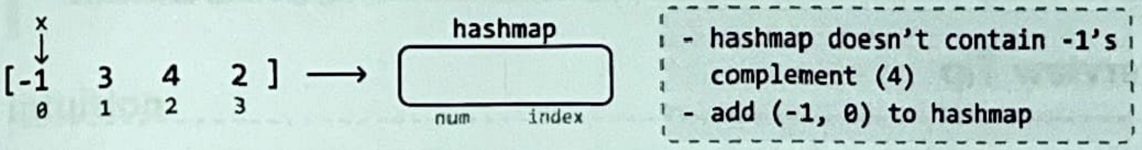
Below is the code snippet for this two-pass approach:

```python
def pair_sum_unsorted_two_pass(nums: List[int],
                               target: int) -> List[int]:
    num_map = {}
    # First pass: Populate the hash map with each number and its
    # index.
    for i, num in enumerate(nums):
        num_map[num] = i
    # Second pass: Check for each number's complement in the hash map.
    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_map and num_map[complement] != i:
            return [i, num_map[complement]]
    return []
```

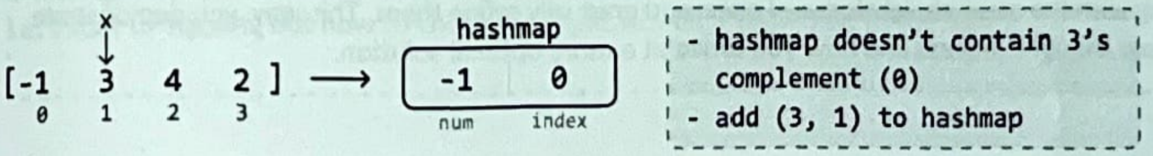
This algorithm requires two passes. Is it possible to do this in only one? A one-pass solution implies that we would need to populate the hash map while searching for complements. Is this possible? Consider the example below:

```
[-1  3  4  2 ], target = 3
  0  1  2  3
```

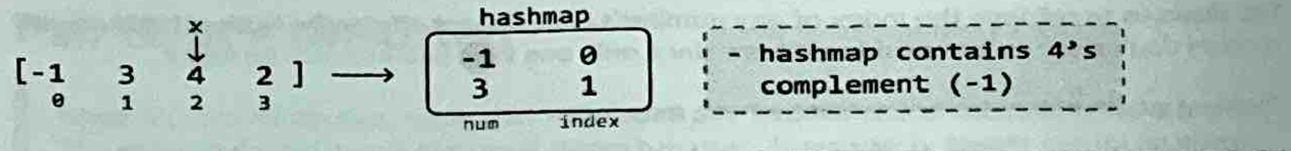
Start at index 0. Its complement would be 3 - (-1) = 4. Does our hash map have 4 in it? No, it's empty at the moment. So, let's add -1 and its index to the hash map:



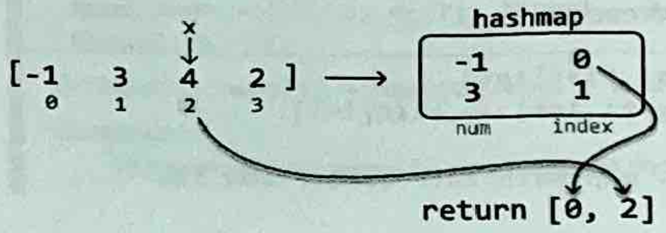
Next, let's look at index 1. Its complement (0) does not exist in the hash map. So, just add 3 and its index to the hash map:



At index 2, we notice 4's complement (-1) exists in the hash map. This means we found a pair that sums to the target:



Now, we can return the indexes of the two values. Fetch the index of 4 from the input array and the index of its complement from the hash map:



Implementation

```
def pair_sum_unsorted(nums: List[int], target: int) -> List[int]:
    hashmap = {}
    for i, x in enumerate(nums):
        if target - x in hashmap:
            return [hashmap[target - x], i]
        hashmap[x] = i
    return []
```

Complexity Analysis

Time complexity: The time complexity of `pair_sum_unsorted` is $O(n)$ because we iterate through each element in the nums array once and perform constant-time hash map operations during each iteration.

Space complexity: The space complexity is $O(n)$ since the hash map can grow up to n in size.

Interview Tip

Tip: Iterate through solutions.

Don't always jump straight to the most optimal or clever solution, as this won't give the interviewer much insight into your problem-solving process. Consider multiple approaches, starting with the more straightforward ones, and gradually refine them. This way, you demonstrate your thought process and how you arrive at a more optimal solution.

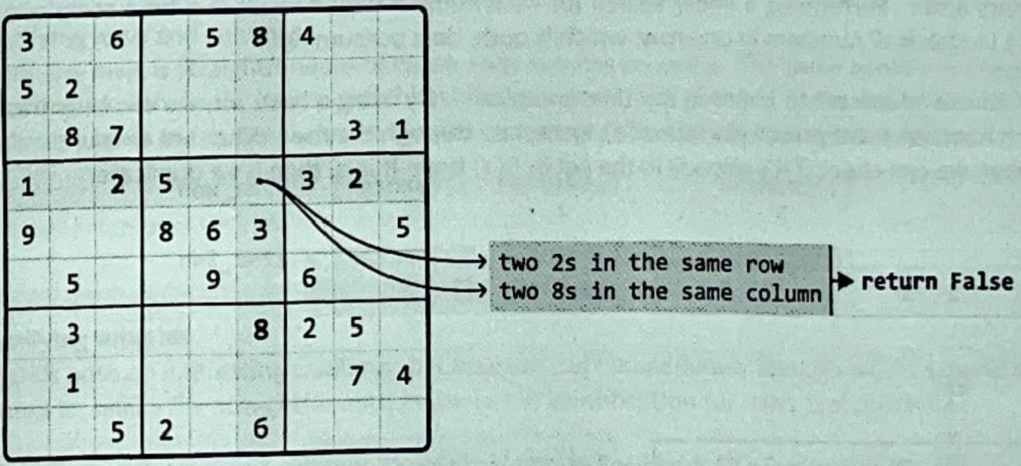
Verify Sudoku Board

Given a partially completed 9×9 Sudoku board, determine if the current state of the board adheres to the rules of the game:

- Each row and column must contain unique numbers between 1 and 9, or be empty (represented as 0).
- Each of the nine 3×3 subgrids that compose the grid must contain unique numbers between 1 and 9, or be empty.

Note: You are asked to determine whether the **current state of the board** is valid given these rules, **not** whether the board is solvable.

Example:



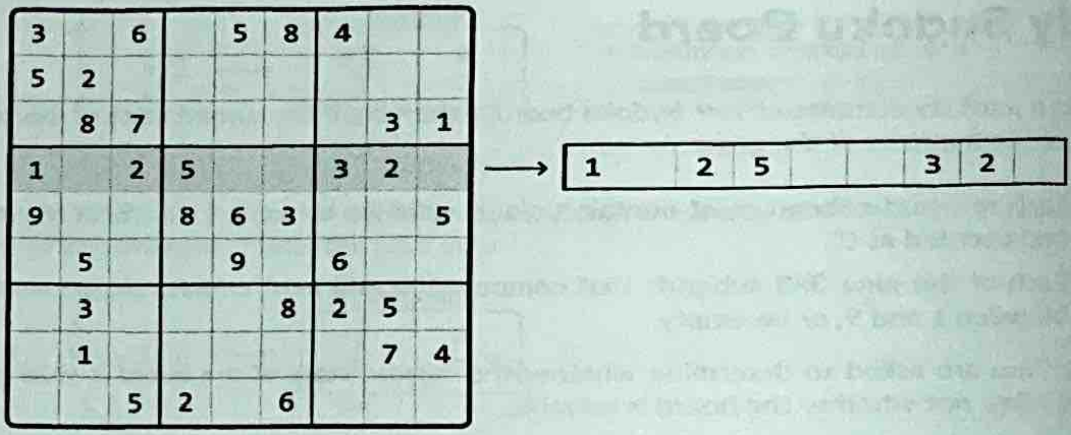
Output: `False`

Constraints:
- Assume each integer on the board falls in the range of [0, 9].

Intuition

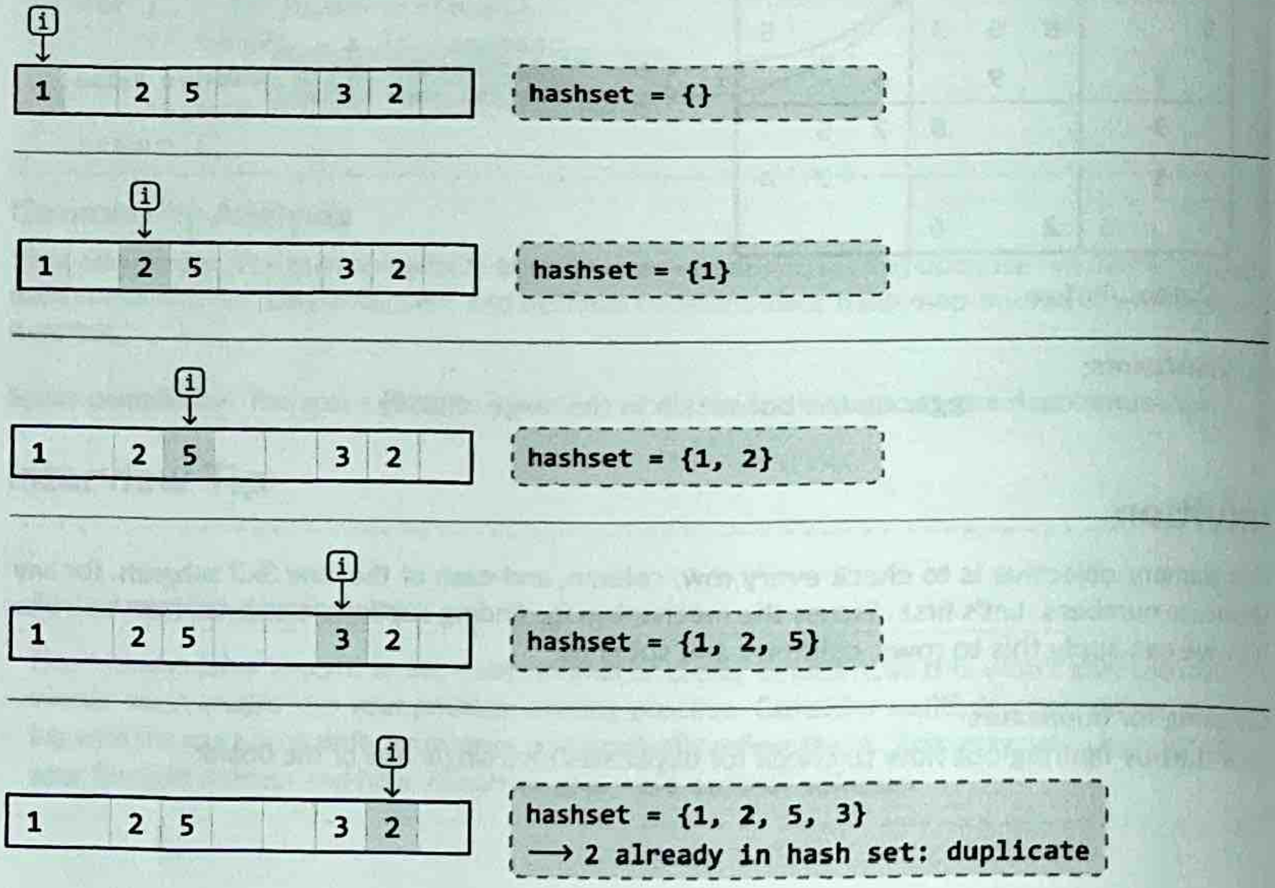
Our primary objective is to check every row, column, and each of the nine 3×3 subgrids, for any duplicate numbers. Let's first discuss the mechanism for finding duplicate elements, then look into how we can apply this to rows, columns, and subgrids.

Checking for duplicates
Let's start by figuring out how to check for duplicates on a single row of the board:



A naive way to do this is to take each number in the row and search the row to see if that number appears again. Performing a linear search for each number would result in a time complexity of $O(n^2)$ to check all numbers in one row, which is quite time consuming.

We can use a **hash set** to improve the time complexity. By using a hash set, we can keep track of which numbers were previously visited as we iterate through the row. When we encounter a new number, we can check if it's already in the set in $O(1)$ time. If it is, then it's a duplicate:



```
hashset = {}
```

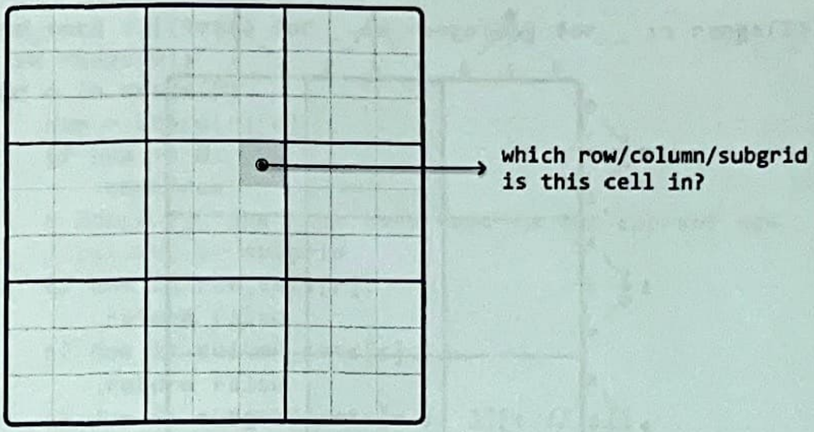
```
hashset = {1}
```

```
hashset = {1, 2}
```

```
hashset = {1, 2, 5}
```

```
hashset = {1, 2, 5, 3}
⟶ 2 already in hash set: duplicate
```

If we had a hash set for each of the 9 rows, we could keep track of duplicates in each row separately. We can do this for columns and subgrids as well, with one hash set for each column, and one hash set for each subgrid. The challenge here is determining which hash sets correspond to each cell's row, column, and subgrid, so we know which hash sets to reference:



So, let's discuss how we can identify a cell's row, column, or subgrid.

Identifying rows and columns

Identifying rows is straightforward because each row has an index. The same applies to columns. Therefore, we can create an array of 9 hash sets for each row, allowing us to access a row's hash set directly by its index. Similarly, we can set up an array of hash sets for each column.

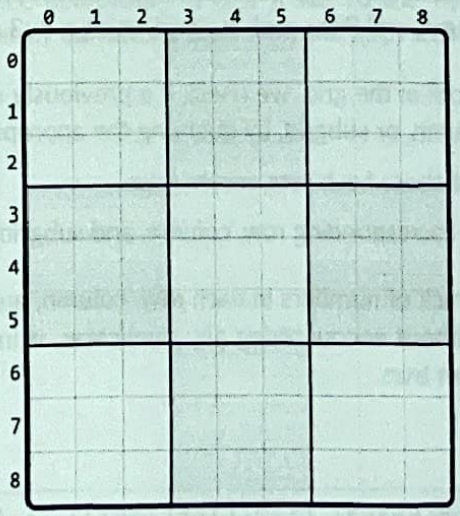
```
row_sets = [hashset(), hashset(), ..., hashset()]
                0           1              8

col_sets = [hashset(), hashset(), ..., hashset()]
                0           1              8
```

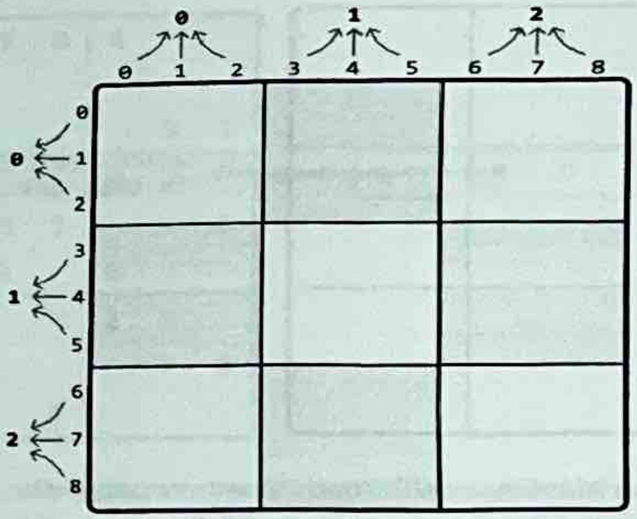
Identifying subgrids

Subgrids pose an interesting challenge because we can't immediately identify which subgrid a cell belongs to, unlike the straightforward index-based identification for rows and columns.

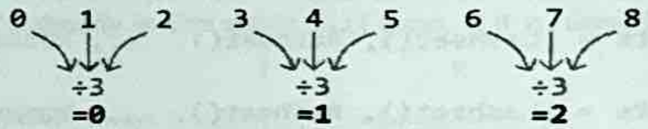
That said, as with rows and columns, there are still only 9 subgrids. If we visualize the subgrids, we can see them displayed in a 3×3 grid:



What we'd like is a way to index each of these subgrids as if indexing a 3×3 matrix. To do this, we require a method to convert the indexes ranging from 0 to 8 to the corresponding adjusted indexes from 0 to 2, as illustrated below:



Since we got these **adjusted indexes** from shrinking a 9×9 grid to a 3×3 grid – which is effectively dividing the number of rows and columns by 3 – we can get the new subgrid row and column indexes by dividing by 3 (using integer division), as well:



With these modified indexes, we can organize nine hash sets within a 3x3 table, one for each subgrid. Each cell in this table represents the corresponding subgrid in the above 3×3 representation. So, we can access the hash set of a subgrid at any cell by using our adjusted indexes (i.e., `subgrid_sets[r // 3][c // 3]`) (the `//` operator performs integer division).

One-pass Sudoku verification

We now have everything needed for a one-pass solution. We can start by initializing hash sets, 9 for each row, 9 for each column, and 9 for each subgrid, using a 3×3 array.

As we iterate through each cell in the grid, we check if a previously encountered number already exists in the current row, column, or subgrid, by querying the appropriate hash sets:

- If the number is in any of these hash sets, return false.
- Otherwise, add it to the corresponding row, column, and subgrid hash sets.

This process helps us keep track of numbers in each row, column, and subgrid. If we successfully iterate through the board without encountering any duplicates, it indicates the Sudoku board is valid. Therefore, we can return true.

Implementation

```
def verify_sudoku_board(board: List[List[int]]) -> bool:
    # Create hash sets for each row, column, and subgrid to keep
    # track of numbers previously seen on any given row, column, or
    # subgrid.
    row_sets = [set() for _ in range(9)]
    column_sets = [set() for _ in range(9)]
    subgrid_sets = [[set() for _ in range(3)] for _ in range(3)]
    for r in range(9):
        for c in range(9):
            num = board[r][c]
            if num == 0:
                continue
            # Check if 'num' has been seen in the current row,
            # column, or subgrid.
            if num in row_sets[r]:
                return False
            if num in column_sets[c]:
                return False
            if num in subgrid_sets[r // 3][c // 3]:
                return False
            # If we passed the above checks, mark this value as seen
            # by adding it to its corresponding hash sets.
            row_sets[r].add(num)
            column_sets[c].add(num)
            subgrid_sets[r // 3][c // 3].add(num)
    return True
```

Complexity Analysis

In this problem, the length of the board is fixed at 9, effectively reducing all approaches to a time and space complexity of $O(1)$. However, to better understand the efficiency of our algorithm in a broader context, let's use n to denote the board's length, allowing us to evaluate the algorithm's performance against arbitrary board sizes.

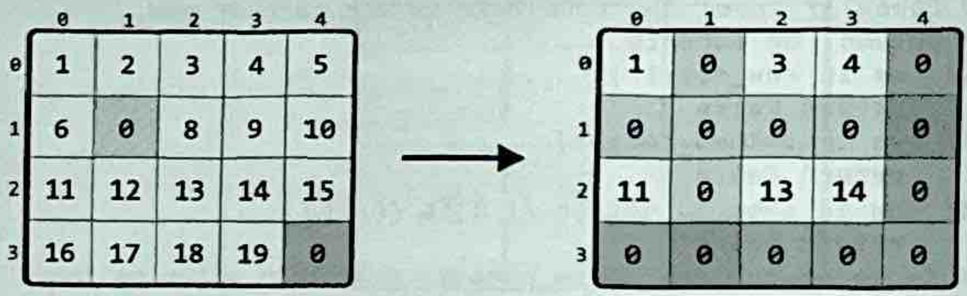
Time complexity: The time complexity of `verify_sudoku_board` is $O(n^2)$ because we iterate through each cell in the board once, and perform constant-time hash set operations.

Space complexity: The space complexity is $O(n^2)$ due to the `row_sets`, `column_sets`, and `subgrid_sets` arrays. Each array contains n hash sets, and each hash set is capable of growing to a size of n.

Zero Striping

For each zero in an $m \times n$ matrix, set its entire row and column to zero in place.

Example:



Intuition – Hash Sets

A brute-force solution involves recording the positions of all 0s initially in the matrix and, for each of these 0s, iterating over their row and column to set them to zero. However, imagine an input array that's filled with many zeros. In the worst case, iterating over every row and column for each zero will take $O(m \cdot n \cdot (m + n))$, where $m \cdot n$ denotes the number of 0s, and $(m + n)$ represents the total number of cells in a row and column combined. This approach is quite inefficient, so let's look for a better solution.

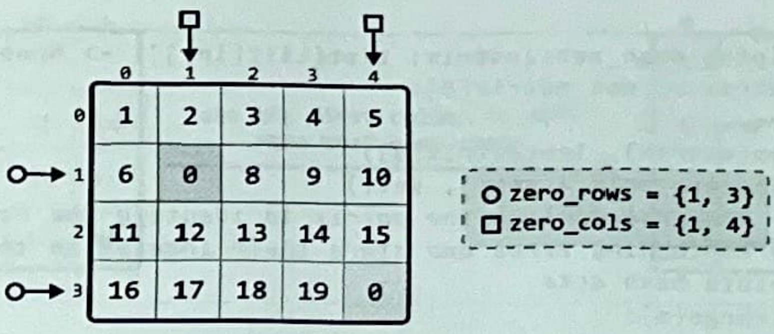
Imagine any cell in the matrix. After the matrix is transformed, this cell will either retain its original value, or become zero. Is there a way to tell if a cell is going to become zero?

> The key observation is that if **a cell is in a row or column containing a zero, that cell will become zero.**

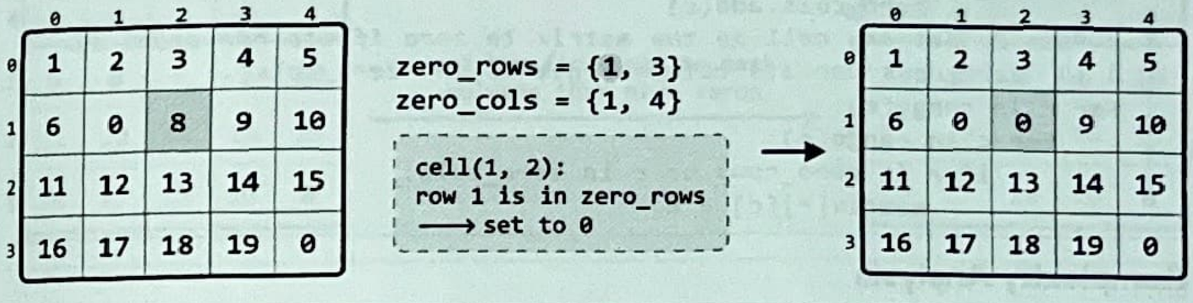
We could search a cell's row and column to check if they contain a zero, meaning each search would take $O(m + n)$ time. But it would be more efficient if we had a way to check this in constant time, and this is where **hash sets** would be useful. If we create two hash sets – one to track all the rows containing a zero and another to track all the columns containing a zero – we can determine if a specific cell's row or column contains a zero in $O(1)$ time.

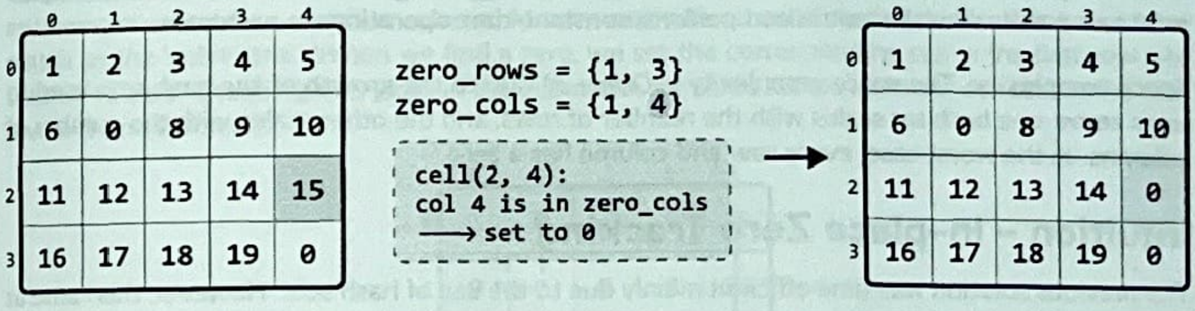
With these hash sets created, the next step is to populate them. As we iterate through the matrix, when encountering a cell containing zero, we:

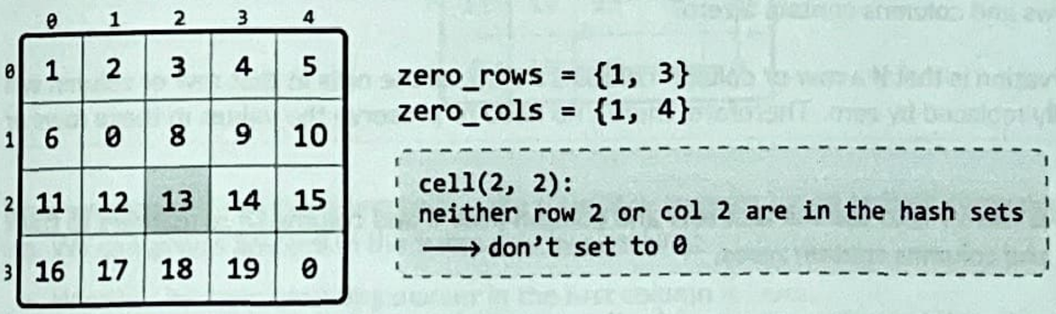
- Add its row index to the row hash set (`zero_rows`).
- Add its column index to the column hash set (`zero_cols`).



Next, we identify the cells whose row or column indexes are present in the respective hash sets, and change their values to zero. Let's look at how this works with a few examples:







This provides a general strategy:

1. In one pass of the matrix, identify each cell containing a zero and add its row and column indexes to the `zero_rows` and `zero_cols` hash sets, respectively.
2. In a second pass, set any cell to zero if its row index is in `zero_rows` or its column index is in `zero_cols`.

Implementation – Hash Sets

```python
def zero_striping_hash_sets(matrix: List[List[int]]) -> None:
    if not matrix or not matrix[0]:
        return
    m, n = len(matrix), len(matrix[0])
    zero_rows, zero_cols = set(), set()
    # Pass 1: Traverse through the matrix to identify the rows and
    # columns containing zeros and store their indexes in the
    # appropriate hash sets.
    for r in range(m):
        for c in range(n):
            if matrix[r][c] == 0:
                zero_rows.add(r)
                zero_cols.add(c)
    # Pass 2: Set any cell in the matrix to zero if its row index is
    # in 'zero_rows' or its column index is in 'zero_cols'.
    for r in range(m):
        for c in range(n):
            if r in zero_rows or c in zero_cols:
                matrix[r][c] = 0
```

Complexity Analysis

Time complexity: The time complexity of `zero_striping_hash_sets` is $O(m \cdot n)$ because we perform two passes over the matrix and perform constant-time operations in each pass.

Space complexity: The space complexity is $O(m + n)$ due to the growth of the hash sets used to track zeros: one hash set scales with the number of rows, and the other scales with the number of columns. In the worst case, every row and column has a zero.

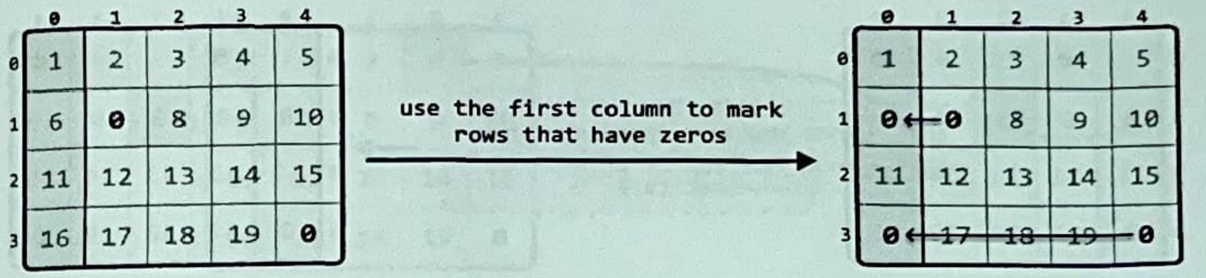
Intuition – In-place Zero Tracking

The previous solution was time efficient mainly due to the use of hash sets. However, this came at the cost of extra space used to store the hash set values. Is there an alternate way to keep track of which rows and columns contain a zero?

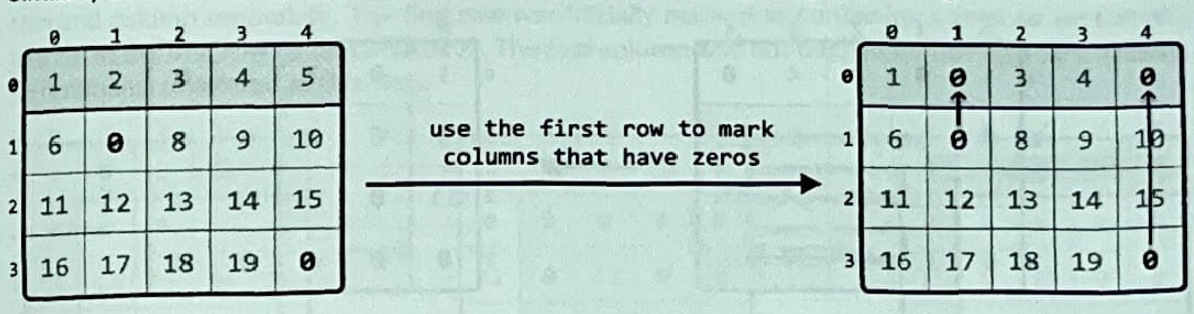
A key observation is that if a row or column contains a zero, all the cells in that row or column will be eventually replaced by zero. Therefore, there's no need to preserve the values in these rows or columns.

A strategy we can try is to **use the first row and column (row 0 and column 0) as markers to track which rows and columns contain zeros.**

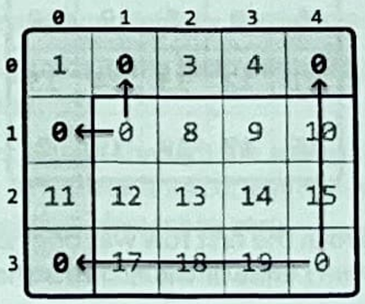
To understand how this would work, consider the example below. For rows, we can use the first column to mark the rows that contain a zero. Specifically, this means if any cell in a row is zero, we set the corresponding cell in the first column to zero. This zero in the first column serves as a marker to indicate the entire row should eventually be set to zeros.



Similarly to how we marked rows, we can mark columns containing zeros using the first row:



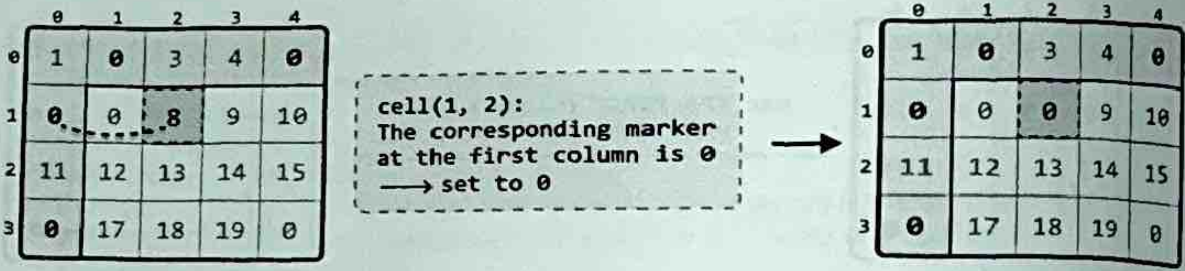
To set markers for the first row and first column, we can begin searching the rest of the matrix, excluding the first row and first column, for any zero-valued cells. Let's refer to this part of the matrix as the 'submatrix.' When we find a zero, we set the corresponding cell in the first row and column to zero. Scanning every cell in the submatrix for zeros allows us to set markers in the first row and first column:



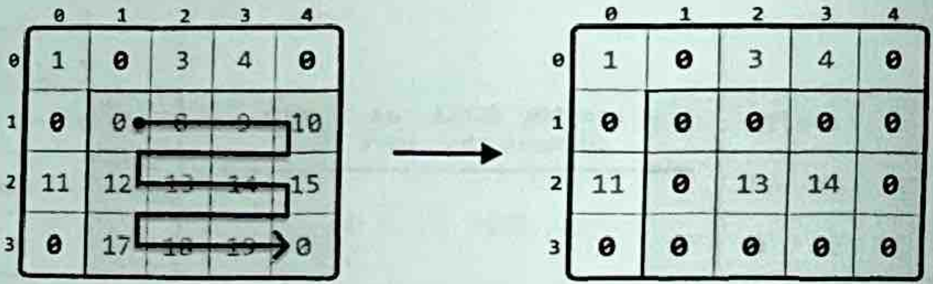
Now, we should start converting cells in the submatrix to zeros based on their corresponding markers. We can assess any cell in the submatrix by checking:

- Whether its corresponding marker in the first column is zero.
- Whether its corresponding marker in the first row is zero.

If either of these conditions are met, we should set that cell's value to zero, as shown below:

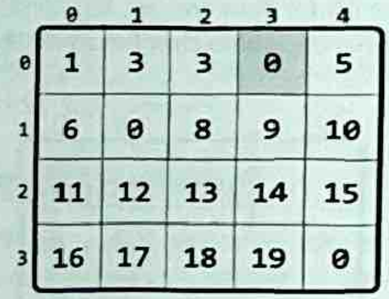


To update this submatrix, we can iterate from the second row and column and update cell values based on the logic we just mentioned:

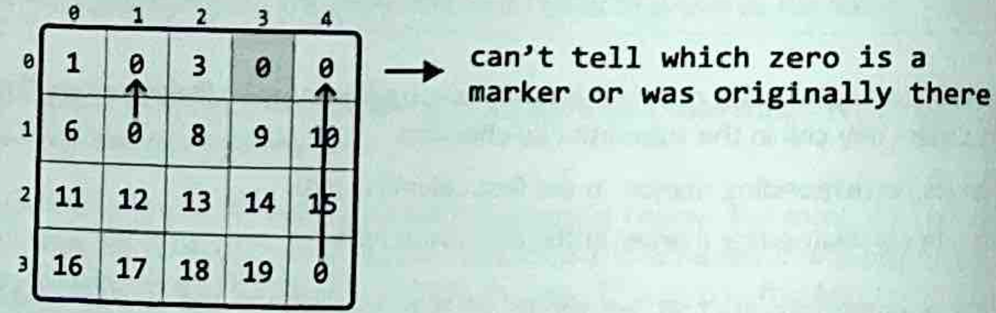


Handling zeros in the first row and column

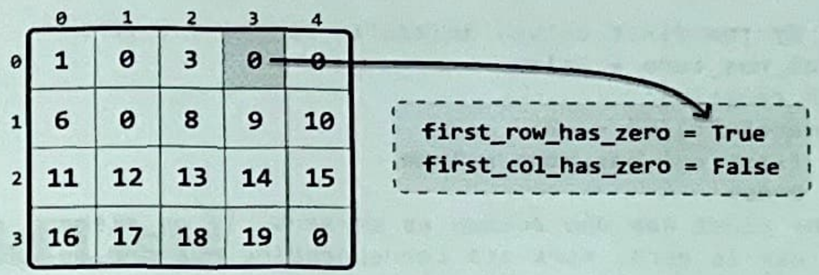
After completing the previous step, there's just one issue to address. What if the first row or column originally had a zero, like in the example below?



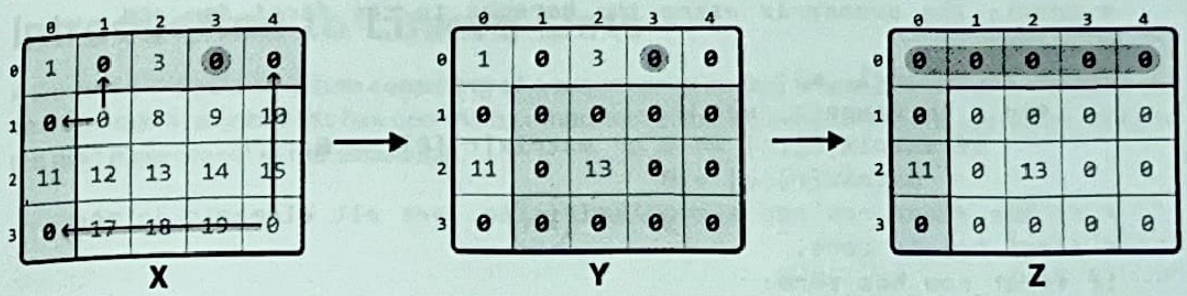
Here, we can't distinguish which zero in the first row was originally present, or resulted from being used as a marker. This means we won't know if the first row should be zeroed:



The remedy for this is to flag whether a zero exists in the first row or first column before using them as markers.



Once we've filled the first row and column with markers, as shown in matrix X below, and set the appropriate cell values in the submatrix to zero, as shown in matrix Y, we then evaluate the first row and column separately. The first row was initially marked as containing a zero, so we convert all cells in the first row to zero (matrix Z). The first column was not flagged for having a zero initially, so it remains unaltered at this step:



In-place zero-marking strategy

Let's summarize the above approach into the following steps:

1. Use a flag to indicate if the first row initially contains any zero.
2. Use a flag to indicate if the first column initially contains any zero.
3. Traverse the submatrix, setting zeros in the first row and column to serve as markers for rows and columns that contain zeros.
4. Apply zeros based on markers: iterate through the submatrix that starts from the second row and second column. For each cell, check if its corresponding marker in the first row or column is marked with a zero. If so, set that element to zero.
5. If the first row was initially marked as containing a zero, set all elements in the first row to zero.
6. If the first column was initially marked as having a zero, set all elements in the first column to zero.

Implementation – In-place Zero Tracking

```
def zero_striping(matrix: List[List[int]]) -> None:
    if not matrix or not matrix[0]:
        return
    m, n = len(matrix), len(matrix[0])
    # Check if the first row initially contains a zero.
    first_row_has_zero = False
    for c in range(n):
        if matrix[0][c] == 0:
            first_row_has_zero = True
            break
```

```
# Check if the first column initially contains a zero.
first_col_has_zero = False
for r in range(m):
    if matrix[r][0] == 0:
        first_col_has_zero = True
        break
# Use the first row and column as markers. If an element in the
# submatrix is zero, mark its corresponding row and column in the
# first row and column as 0.
for r in range(1, m):
    for c in range(1, n):
        if matrix[r][c] == 0:
            matrix[0][c] = 0
            matrix[r][0] = 0
# Update the submatrix using the markers in the first row and
# column.
for r in range(1, m):
    for c in range(1, n):
        if matrix[0][c] == 0 or matrix[r][0] == 0:
            matrix[r][c] = 0
# If the first row had a zero initially, set all elements in the
# first row to zero.
if first_row_has_zero:
    for c in range(n):
        matrix[0][c] = 0
# If the first column had a zero initially, set all elements in
# the first column to zero.
if first_col_has_zero:
    for r in range(m):
        matrix[r][0] = 0
```

Complexity Analysis

Time complexity: The time complexity of `zero_striping` is $O(m \cdot n)$. Here's why:

- Checking the first row for zeros takes $O(m)$ time, and checking the first column takes $O(n)$ time.
- Then, we perform two passes of the entire matrix, one to mark 0s and another to update the matrix based on those markers. Each pass takes $O(m \cdot n)$ time.
- Finally, we iterate through the first row and first column up to once each, which takes $O(m)$ and $O(n)$ time, respectively.

Therefore, the overall time complexity is $O(m) + O(n) + O(m \cdot n) = O(m \cdot n)$.

Space complexity: The space complexity is $O(1)$ because we use the first row and column as markers to track which rows and columns contain zeros, instead of using auxiliary data structures.