1428. Leftmost Column with at Least a One

(This problem is an interactive problem.)

A binary matrix means that all elements are `0` or `1`. For each **individual** row of the matrix, this row is sorted in non-decreasing order.

Given a row-sorted binary matrix binaryMatrix, return leftmost column index(0-indexed) with at least a `1` in it. If such index doesn't exist, return `-1`.

You can't access the Binary Matrix directly. You may only access the matrix using a `BinaryMatrix` interface:

- `BinaryMatrix.get(row, col)` returns the element of the matrix at index `(row, col)` (0-indexed).
- `BinaryMatrix.dimensions()` returns a list of 2 elements `[rows, cols]`, which means the matrix is `rows * cols`.

Submissions making more than `1000` calls to `BinaryMatrix.get` will be judged *Wrong Answer*. Also, any solutions that attempt to circumvent the judge will result in disqualification.

For custom testing purposes you're given the binary matrix `mat` as input in the following four examples. You will not have access the binary matrix directly.

Example 1:

```
Input: mat = [[0,0],[1,1]]
Output: 0
```

Example 2:

```
Input: mat = [[0,0],[0,1]]
Output: 1
```

Example 3:

```
Input: mat = [[0,0],[0,0]]
Output: -1
```

Example 4:

```
Input: mat = [[0,0,0,1],[0,0,1,1],[0,1,1,1]]
Output: 1
```

Constraints:

- `rows == mat.length`
- `cols == mat[i].length`
- `1 <= rows, cols <= 100`
- `mat[i][j]` is either `0` or `1`.
- `mat[i]` is sorted in a non-decreasing way.

Solution

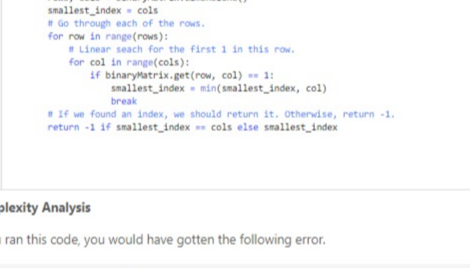
Approach 1: Linear Search Each Row

Intuition

This approach won't pass, but we'll use it as a starting point. Also, it might be helpful to you if you just needed an example of how to use the API, but don't want to see a complete solution yet!

The leftmost `1` is the `1` with the lowest *column* index.

The problem can be broken down into finding the index of the first `1` in each row and then taking the minimum of those indexes.



The simplest way of doing this would be a linear search on each row.

Algorithm

Java | Python3

```python
class Solution:
    def leftMostColumnWithOne(self, binaryMatrix: 'BinaryMatrix') -> int:
        rows, cols = binaryMatrix.dimensions()
        smallest_index = cols
        # Go through each of the rows.
        for row in range(rows):
            # Linear seach for the first 1 in this row.
            for col in range(cols):
                if binaryMatrix.get(row, col) == 1:
                    smallest_index = min(smallest_index, col)
                    break
        # If we found an index, we should return it. Otherwise, return -1.
        return -1 if smallest_index == cols else smallest_index
```

Complexity Analysis

If you ran this code, you would have gotten the following error.

```
You made too many calls to BinaryMatrix.get().
```

The maximum grid size is `100 by 100`, so it would contain `10000` cells. In the worst case, the linear search algorithm we implemented has to check every cell. With the problem description telling us that we can only make up to `1000` API calls, this clearly isn't going to work.

Let N be the number of rows, and M be the number of columns.

- Time complexity : $O(N \cdot M)$

  We don't know the time complexity of `binaryMatrix.get()` as its implementation isn't our concern. Therefore, we can assume it's $O(1)$.

  In the worst case, we are retrieving a value for each of the $N \cdot M$ cells. At $O(1)$ per operation, this gives a total of $O(N \cdot M)$.

- Space complexity : $O(1)$.

  We are only using constant extra space.

Approach 2: Binary Search Each Row

Intuition

This isn't the best approach, but it passes, and coding it up is a good opportunity to practice binary search.

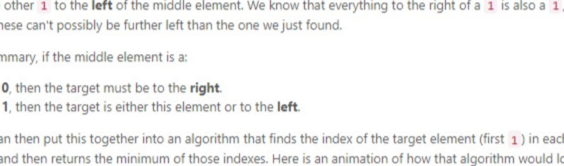
When linear search is too slow, we should try to find a way to use binary search. If you're not familiar with binary search, check out this Explore Card!. We recommend doing the first couple of binary search questions to get familiar with the algorithm before coming back to this problem.

Also, have a go at First Bad Version. The only difference between that problem and this one is that instead of `0` and `1`, it uses `false` and `true`.

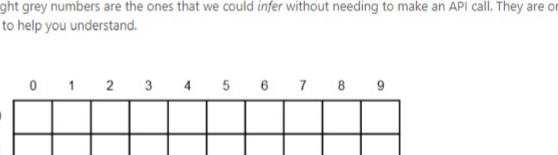
Like we did with the linear search, we're going to apply binary search independently on each row. The *target element* we're searching for is *the first 1 in the row.*

The core part of a binary search algorithm is how it decides whether the target element is to the left or the right of the middle element. Let's figure this out by thinking through a couple of examples.

In the row below, we've determined that the middle element is a `0`. On what side must the target element (first `1`) be? The left, or the right? Don't forget, *all the 0's are before all the 1's.*



In this next row, the middle element is a `1`? What side must the target element be on? Could it also possibly be the `1` we just found?



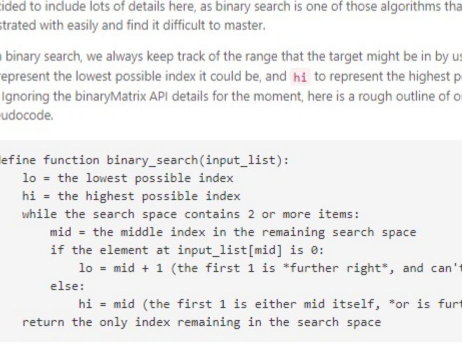
For the first example, we can conclude that the target element (*if* it exists) must be to the **right** of the middle element. This is because we know that everything to the left of a `0` must also be a `0`.

For the second example, we can conclude that the target element is either the middle element itself or it is some other `1` to the **left** of the middle element. We know that everything to the right of a `1` is also a `1`, but these can't possibly be further left than the one we just found.

In summary, if the middle element is a:

- **0**, then the target must be to the **right**.
- **1**, then the target is either this element or to the **left**.

We can then put this together into an algorithm that finds the index of the target element (first `1`) in each row, and then returns the minimum of those indexes. Here is an animation of how that algorithm would look. The light grey numbers are the ones that we could *infer* without needing to make an API call. They are only there to help you understand.



Algorithm

If you're already quite familiar with binary search, feel free to skip down to the implementation below. I've decided to include lots of details here, as binary search is one of those algorithms that a lot of people get frustrated with easily and find it difficult to master.

In a binary search, we always keep track of the range that the target might be in by using two variables: `lo` to represent the lowest possible index it could be, and `hi` to represent the highest possible index it could be. Ignoring the binaryMatrix API details for the moment, here is a rough outline of our binary search in pseudocode.

```
define function binary_search(input_list):
    lo = the lowest possible index
    hi = the highest possible index
    while the search space contains 2 or more items:
        mid = the middle index in the remaining search space
        if the element at input_list[mid] is 0:
            lo = mid + 1 (the first 1 is *further right*, and can't be mid itself)
        else:
            hi = mid (the first 1 is either mid itself, *or is further left*)
    return the only index remaining in the search space
```

As always in binary search, there are a couple more key implementation details we still need to deal with:

1. An even-length search space has two middles. Which do we choose?
2. The row might be all 0's.

Let's tackle these issues one at a time.

The first issue, the choice of middle, is important, as otherwise, the search space might stop shrinking when it gets down to two elements. When the search space doesn't shrink, the algorithm does the exact same thing the next loop cycle, resulting in an infinite loop. Remember that when the search space only contains two elements, they are the ones pointed to by `lo` and `hi`. This means that the lower middle equals `lo`, and the upper-middle equals `hi`. We, therefore, need to think about which cases would shrink the search space, and which would not.

If we use the *lower-middle*

- If it is a `0`, then we set `lo = mid + 1`. Because `hi == mid + 1`, this means that `lo == hi` (search space is of length-one).
- If it is a `1`, then we set `hi = mid`. Because `mid == lo`, this means that `hi == lo` (search space is of length-one).

If we use the *upper-middle*

- If it is a `0`, then we set `lo = mid + 1`. Because `hi = mid`, we now have `hi > lo` (search space is of length-zero).
- If it is a `1`, then we set `hi = mid`. Because `hi == mid` was already true, the search space stays as is (of length-two).

If we use the *lower-middle*, we know the search space will always shrink. If we use the *upper-middle*, it might not. Therefore, we should go with the *lower-middle*. The formula for this is `mid = (low + high) / 2`.

The second issue, a row of all zeroes, is solved by recognizing that the algorithm always shrinks down the search space to a single element. This is supposed to be the first `1`, but if that doesn't exist, then it *has* to be a `0`. Therefore, we can detect this case by checking whether or not the last element in the search space is a `1`.

It is good practice to think these details through carefully so that you can write your binary search algorithm decisively and confidently. Resist the urge to Program by Permutation!

Anyway, putting this all together, we get the following code.

Java | Python3

```python
class Solution:
    def leftMostColumnWithOne(self, binaryMatrix: 'BinaryMatrix') -> int:
        rows, cols = binaryMatrix.dimensions()
        smallest_index = cols
        for row in range(rows):
            # Binary Search for the first 1 in the row.
            lo = 0
            hi = cols - 1
            while lo < hi:
                mid = (lo + hi) // 2
                if binaryMatrix.get(row, mid) == 0:
                    lo = mid + 1
                else:
                    hi = mid
            # If the last element in the search space is a 1, then this row
            # contained a 1.
            if binaryMatrix.get(row, lo) == 1:
                smallest_index = min(smallest_index, lo)
        # If smallest_index is still set to cols, then there were no 1's in
        # the grid.
        return -1 if smallest_index == cols else smallest_index
```

Complexity Analysis

Let N be the number of rows, and M be the number of columns.

- Time complexity : $O(N \log M)$.

  There are M items in each row. Therefore, each binary search will have a cost of $O(\log M)$. We are performing N of these binary searches, giving a time complexity of $N \cdot O(\log M) = O(N \log M)$.

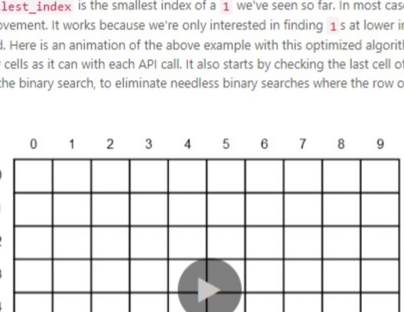
- Space complexity : $O(1)$.

  We are using constant extra space.

Approach 3: Start at Top Right, Move Only Left and Down

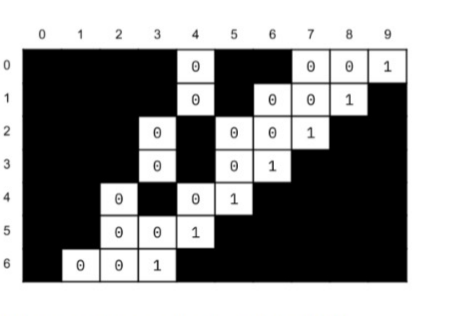
Intuition

Did you notice in Approach 2 that we didn't need to finish searching all the rows? One example of this was row 3 on the example in the animation. At the point shown in the image below, it was clear that row 3 *could not possibly be better than the minimum we'd found so far.*

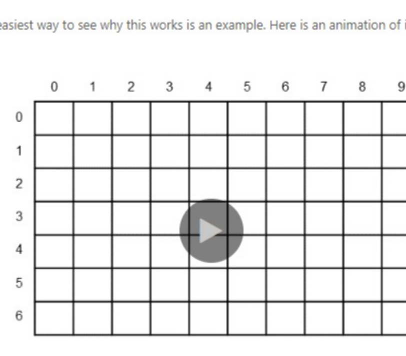


Therefore, an optimization we could have made was to keep track of the minimum index so far, and then abort the search on any rows where we have discovered a `0` at, or to the right of, that minimum index.

We can do even better than that; on each search, we can set `hi = smallest_index - 1`, where `smallest_index` is the smallest index of a `1` we've seen so far. In most cases, this is a substantial improvement. It works because we're only interested in finding `1`s at lower indexes than we previously found. Here is an animation of the above example with this optimized algorithm. The algorithm eliminates as many cells as it can with each API call. It also starts by checking the last cell of the row before proceeding with the binary search, to eliminate needless binary searches where the row only had `0`s left in it.



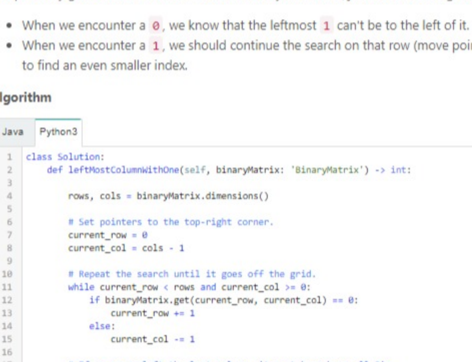
Here is what the worst-case looks like. Like before, its time complexity is still $O(M \log N)$.



While this is no worse than Approach 2, there is a better algorithm.

> Start in the top right corner, and if the current value is a `0`, move down. If it is a `1`, then move left.

The easiest way to see why this works is an example. Here is an animation of it.



You probably gained some intuitive sense as to why this works, just from watching the animation.

- When we encounter a `0`, we know that the leftmost `1` can't be to the left of it.
- When we encounter a `1`, we should continue the search on that row (move pointer to the left), in order to find an even smaller index.

Algorithm

Java | Python3

```python
class Solution:
    def leftMostColumnWithOne(self, binaryMatrix: 'BinaryMatrix') -> int:

        rows, cols = binaryMatrix.dimensions()

        # Set pointers to the top-right corner.
        current_row = 0
        current_col = cols - 1

        # Repeat the search until it goes off the grid.
        while current_row < rows and current_col >= 0:
            if binaryMatrix.get(current_row, current_col) == 0:
                current_row += 1
            else:
                current_col -= 1

        # If we never left the last column, it must have been all 0's.
        return current_col + 1 if current_col != cols - 1 else -1
```

Complexity Analysis

Let N be the number of rows, and M be the number of columns.

- Time complexity : $O(N + M)$.

  At each step, we're moving 1 step left or 1 step down. Therefore, we'll always finish looking at either one of the M rows or N columns. Therefore, we'll stay in the grid for at most $N + M$ steps, and therefore get a time complexity of $O(N + M)$.

- Space complexity : $O(1)$.

  We are using constant extra space.

Analysis written by @hai_dee