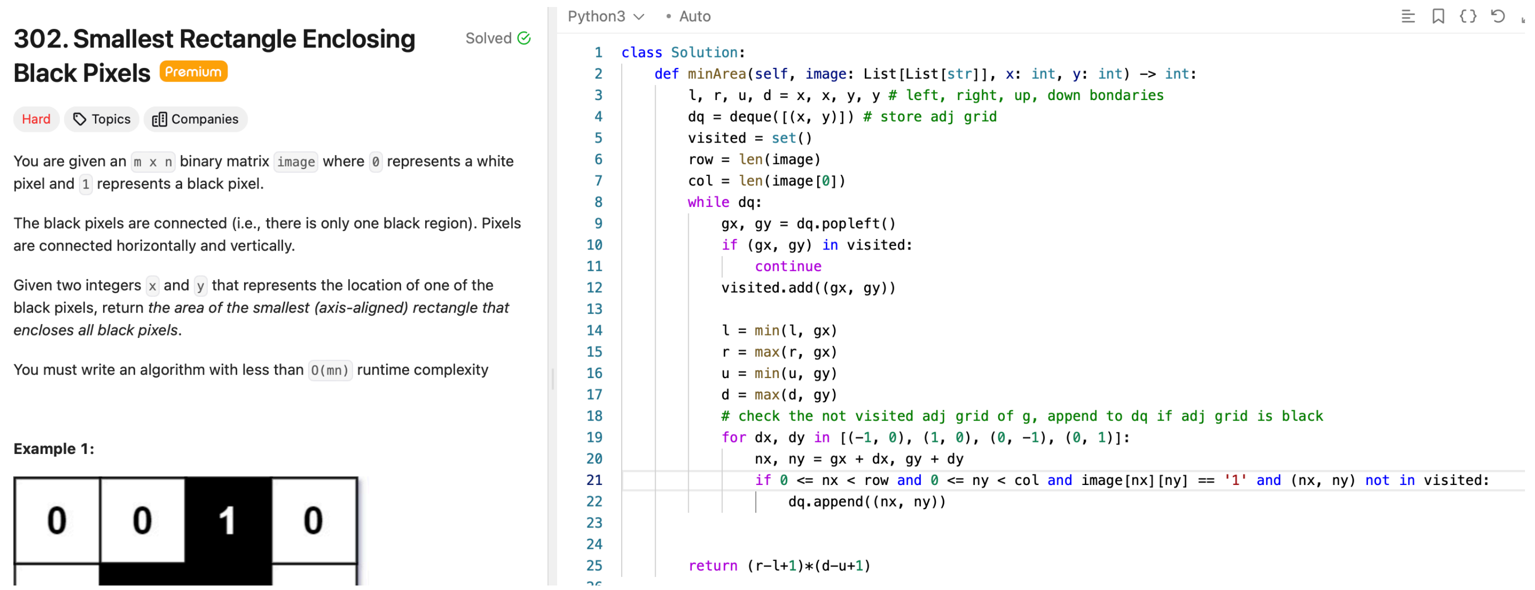


302. Smallest Rectangle Enclosing Black Pixels

You are given an `m x n` binary matrix `image` where `0` represents a white pixel and `1` represents a black pixel.

The black pixels are connected (i.e., there is only one black region). Pixels are connected horizontally and vertically.

Given two integers `x` and `y` that represents the location of one of the black pixels, return *the area of the smallest (axis-aligned) rectangle that encloses all black pixels*.

You must write an algorithm with less than `O(mn)` runtime complexity

Example 1:

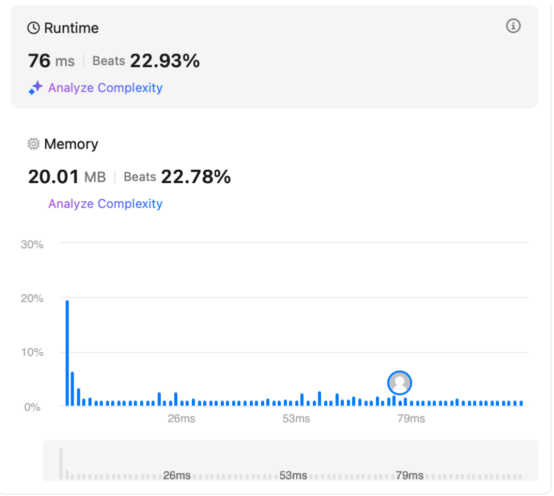
0	0	1	0

```python
class Solution:
    def minArea(self, image: List[List[str]], x: int, y: int) -> int:
        l, r, u, d = x, x, y, y # left, right, up, down bondaries
        dq = deque([(x, y)]) # store adj grid
        visited = set()
        row = len(image)
        col = len(image[0])
        while dq:
            gx, gy = dq.popleft()
            if (gx, gy) in visited:
                continue
            visited.add((gx, gy))

            l = min(l, gx)
            r = max(r, gx)
            u = min(u, gy)
            d = max(d, gy)
            # check the not visited adj grid of g, append to dq if adj grid is black
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = gx + dx, gy + dy
                if 0 <= nx < row and 0 <= ny < col and image[nx][ny] == '1' and (nx, ny) not in visited:
                    dq.append((nx, ny))


        return (r-l+1)*(d-u+1)
```

correct but not efficient



Runtime

76 ms | Beats 22.93%

Analyze Complexity

Memory

20.01 MB | Beats 22.78%

Analyze Complexity

binary search is more efficient

```python
class Solution:
    def minArea(self, image: List[List[str]], x: int, y: int) -> int:
        m, n = len(image), len(image[0])

        # Helper to check if there is any black pixel in a row
        def hasBlackInRow(row):
            return '1' in image[row]

        # Helper to check if there is any black pixel in a column
        def hasBlackInCol(col):
            return any(image[i][col] == '1' for i in range(m))

        # Binary search for top
        top = self.binarySearch(0, x, hasBlackInRow, True)
        # Binary search for bottom
        bottom = self.binarySearch(x + 1, m, hasBlackInRow, False)
        # Binary search for left
        left = self.binarySearch(0, y, hasBlackInCol, True)
        # Binary search for right
        right = self.binarySearch(y + 1, n, hasBlackInCol, False)

        return (bottom - top) * (right - left)

    def binarySearch(self, low, high, hasBlack, goLower):
        while low < high:
            mid = (low + high) // 2
            if hasBlack(mid):
                if goLower:
                    high = mid
                else:
                    low = mid + 1
            else:
                if goLower:
                    low = mid + 1
                else:
                    high = mid
        return low
```

📌 How it Works:

- `hasBlackInRow(row)`: Checks if any black pixel is in that row.
- `hasBlackInCol(col)`: Checks if any black pixel is in that column.
- We binary search rows and columns to find the min/max bounds efficiently.

🧠 Time Complexity:

- **O(m log n + n log m)**:
    - We do binary search over rows and columns.
    - Each check takes O(n) or O(m), but only for log steps.