Daily Coding Problem #136

Problem

This question was asked by Google.

Given an N by M matrix consisting only of 1's and 0's, find the largest rectangle containing only 1's and return its area.

For example, given the following matrix:

```
[[1, 0, 0, 0],
 [1, 0, 1, 1],
 [1, 0, 1, 1],
 [0, 1, 0, 0]]
```

Return 4.

Solution

The brute force method for solving this problem would involve enumerating every possible subrectangle in the matrix, checking if they are all 1s and summing them up if they are, keeping track of the largest subrectangle of 1s so far. This would take constant space but $O(M^3N^3)$ time, since we have to enumerate every subrectangle, which is $O(N^2M^2)$, and for each subrectangle we may have to sum up all the area inside it, which is another $O(MN)$.

The code for the brute force method would look like this:

```
def is_valid(matrix, top_left_row, top_left_col, bottom_right_row, bottom_right_col):
    for i in range(top_left_row, bottom_right_row):
        for j in range(top_left_col, bottom_right_col):
            if matrix[i][j] == 0:
                return False
    return True

def area(top_left_row, top_left_col, bottom_right_row, bottom_right_col):
```

```
    return (bottom_right_row - top_left_row) * (bottom_right_col - top_left_col)

def largest_rectangle(matrix):
    n, m = len(matrix), len(matrix[0])
    max_so_far = 0
    for top_left_row in range(n):
        for top_left_col in range(m):
            for bottom_right_row in range(n, top_left_row, -1):
                for bottom_right_col in range(m, top_left_col, -1):
                    if is_valid(
                            matrix,
                            top_left_row,
                            top_left_col,
                            bottom_right_row,
                            bottom_right_col):
                        max_so_far = max(
                                max_so_far,
                                area(
                                    top_left_row,
                                    top_left_col,
                                    bottom_right_row,
                                    bottom_right_col))
    return max_so_far
```

To improve the runtime, we can actually only look at the whole matrix once. If we keep a row in our cache, then we can keep incrementing the elements in our row for each column if we encounter a 1, or reset it to 0 if we encounter a 0. Then we can infer the largest subrectangle we've seen so far, and keep track of the largest.

Let's go through an example:

```
101
011
111
```

For this example, our cache starts off as `[0, 0, 0]`. After we read in the first row, it becomes `[1, 0, 1]`. The largest subrectangle here is either at column `0` or 2, with an area of 1. When we read in the second row, we'll reset to zero each column we see a 0 in and increment each column we see a 1 in, so it'll be `[0, 1, 2]`.

Now we know that we've seen one 1 in the middle column and two 1s in a row in the last column, so we can infer that the largest subrectangle of 1s so far has area 2.

After reading the third row, we get `[1, 2, 3]`, and again infer that the largest area is 2 * 2 = 4.

How can we infer the area? We're given a row of heights, similar to a histogram, and we want the largest rectangle we can make given these values. We can do this using brute force: calculating the

area of each rectangle starting and ending at each index by multiplying the min height by the width.

The code for this would look like this:

```
def infer_area(cache):
    max_area = 0
    for i in range(len(cache)):
        for j in range(i + 1, len(cache) + 1):
            current_rectangle = min(cache[i:j]) * (j - i)
            max_area = max(max_area, current_rectangle)
    return max_area


def largest_rectangle(matrix):
    n, m = len(matrix), len(matrix[0])
    max_so_far = 0

    cache = [0 for _ in range(m)]
    for row in matrix:
        for i, val in enumerate(row):
            if val == 0:
                cache[i] = 0
            elif val == 1:
                cache[i] += 1
        max_so_far = max(max_so_far, infer_area(cache))

    return max_so_far
```

This would take $O(MN^3)$ time and O(N) space, since we keep one row in memory (as the cache), and we run a double for loop over each row to calculate the area.