Daily Coding Problem #19

Problem

This problem was asked by Facebook.

A builder is looking to build a row of N houses that can be of K different colors. He has a goal of minimizing cost while ensuring that no two neighboring houses are of the same color.

Given an N by K matrix where the nth row and kth column represents the cost to build the nth house with kth color, return the minimum cost which achieves this goal.

Solution

The brute force solution here would be to generate all possible combinations of houses and colors, filter out invalid combinations, and keep track of the lowest cost seen. This would take O(N^K) time.

We can solve this problem faster using dynamic programming. We can maintain a matrix cache where every entry [i][j] represents the minimum cost of painting house i the color j, as well as painting every house < i. We can calculate this by looking at the minimum cost of painting each house < i - 1, and painting house i - 1 any color except j, since that would break our constraint. We'll initialize the first row with zeroes to start. Then, we just have to look at the smallest value in the last row of our cache, since that represents the minimum cost of painting every house.

```python
def build_houses(matrix):
    n = len(matrix)
    k = len(matrix[0])
    solution_matrix = [[0] * k]

    # Solution matrix: matrix[i][j] represents the minimum cost to build house i with color j.
    for r, row in enumerate(matrix):
        row_cost = []
        for c, val in enumerate(row):
            row_cost.append(min(solution_matrix[r][i] for i in range(k) if i != c) + val)
        solution_matrix.append(row_cost)
    return min(solution_matrix[-1])
```

This runs in O(N *K^2) time and O(N* K) space. Can we do even better than this?

First off, notice that we're only ever looking at the last row when computing the next row's cost. That suggests that we only need to keep track of one array of size K instead of a whole matrix of size N * K:

```python
def build_houses(matrix):
    k = len(matrix[0])
    soln_row = [0] * k

    for r, row in enumerate(matrix):

        new_row = []
        for c, val in enumerate(row):
            new_row.append(min(soln_row[i] for i in range(k) if i != c) + val)
        soln_row = new_row
    return min(soln_row)
```

Now we're only using O(K) space! Can we improve this any more?

Hold on a second. When we're looking at the previous row's total cost, it looks like we're almost computing the same thing each time: the minimum of the previous row that isn't the current index.

For every element that **isn't** that index, it will be the same value. When it **is** that index, it will be the second-smallest value.

Now, armed with this insight, we only need to keep track of three variables:

- The lowest cost of the current row
- The index of the lowest cost
- The second lowest cost

Then, when looking at the value at each row, we only need to do the following:

- Check if the index is the index of the lowest cost of the previous row. If it is, then we can't use this color -- we'll use the second lowest cost instead. Otherwise, use the lowest cost of the previous row
- Calculate the minimum cost if we painted this house this particular color

- Update our new lowest cost/index or second lowest cost if appropriate

Now we'll always have our lowest cost in a variable, and once we've gone through the matrix we can just return that.

```python
from math import inf

def build_houses(matrix):
    lowest_cost, lowest_cost_index = 0, -1
    second_lowest_cost = 0

    for r, row in enumerate(matrix):
        new_lowest_cost, new_lowest_cost_index = inf, -1
        new_second_lowest_cost = inf
        for c, val in enumerate(row):
            prev_lowest_cost = second_lowest_cost if c == lowest_cost_index else lowest_cost

            cost = prev_lowest_cost + val
            if cost < new_lowest_cost:
                new_second_lowest_cost = new_lowest_cost
                new_lowest_cost, new_lowest_cost_index = cost, c
            elif cost < new_second_lowest_cost:
                new_second_lowest_cost = cost
        lowest_cost = new_lowest_cost
        lowest_cost_index = new_lowest_cost_index
        second_lowest_cost = new_second_lowest_cost

    return lowest_cost
```

Now the runtime is only O(N * K) and the space complexity is O(1) - constant, since we keep track of only three variables!

Thanks to Alexander Shirkov for pointing out these optimizations!