Daily Coding Problem #130

Problem

This problem was asked by Facebook.

Given an array of numbers representing the stock prices of a company in chronological order and an integer k, return the maximum profit you can make from k buys and sells. You must buy the stock before you can sell it, and you must sell the stock before you can buy it again.

For example, given k = 2 and the array `[5, 2, 4, 0, 1]`, you should return 3.

Solution

This problem is quite a bit harder than Daily Coding Problem #47, where we only needed to buy and sell a stock once to get the max profit.

Let's consider the last price in the array. Either we used that price in a transaction or we didn't. Thus, the max profit for *our* input should be the max of either:

- The max profit of prices[:-1] using k transactions (we didn't use the last price)
- The best you can get by selling the stock on the last day (we use the last price)

We can use dynamic programming to represent the maximum profit we can get. Let `dp[i][j]` represent the maximum profit using at most `i` transactions up to day `j` (inclusive).

So the recurrence is:

```
dp[0][j] = 0 # If k = 0 then no way to profit
dp[i][0] = 0 # If no prices then no way to profit
```

```
dp[i][j] = max(
    dp[i][j - 1], # We don't use the last price
    max(price[j] - price[m] + dp[i - 1][m] for m in range(j)) # Best we can do by
completing last transaction on jth day
)
```

So let's build our matrix bottom-up and fill it according to the recurrence:

```
from math import inf

def max_profit(k, prices):
    n = len(prices)
    dp = [[0 for _ in range(n)] for _ in range(k + 1)]

    for i in range(k + 1):
        dp[i][0] = 0

    for j in range(n):
        dp[0][j] = 0

    for i in range(1, k + 1):
        for j in range(1, n):
            best_so_far = -inf
            for m in range(j):
                best_so_far = max(best_so_far, prices[j] - prices[m] + dp[i - 1][m])
            dp[i][j] = max(best_so_far, dp[i][j - 1])

    return dp[k][n - 1]
```

This will run in $O(kn^2)$ time and take $O(kn)$ space.

We can improve the time complexity by noticing that we're redoing some of the same calculations in the innermost for loop. In the term `prices[j] - prices[m] + dp[i - 1][m]`, `prices[j]` stays constant and the rest of the term can be factored into the outer loop.

```
from math import inf

def max_profit(k, prices):
    n = len(prices)
    dp = [[0 for _ in range(n)] for _ in range(k + 1)]

    for i in range(k + 1):
```

```
        dp[i][0] = 0

    for j in range(n):
        dp[0][j] = 0

    for i in range(1, k + 1):
        best_so_far = -inf
        for j in range(1, n):
            best_so_far = max(best_so_far, dp[i - 1][j - 1] - prices[j - 1])
            dp[i][j] = max(dp[i][j - 1], prices[j] + best_so_far)

    return dp[k][n - 1]
```

Now that we got rid of the inner for loop, this runs in O(kn) time and space.

© Daily Coding Problem 2018

Privacy Policy

Terms of Service

Press