Best Time to Buy and Sell Stock with Cooldown

Say you have an array for which the i^{th} element is the price of a given stock on day i.

Design an algorithm to find the maximum profit. You may complete as many transactions as you like (ie, buy one and sell one share of the stock multiple times) with the following restrictions:

- You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).
- After you sell your stock, you cannot buy stock on next day. (ie, cooldown 1 day)

Example:

```
prices = [1, 2, 3, 0, 2]
maxProfit = 3
transactions = [buy, sell, cooldown, buy, sell]
```

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

The series of problems are typical dp. The key for dp is to find the variables to represent the states and deduce the transition function.

Of course one may come up with a O(1) space solution directly, but I think it is better to be generous when you think and be greedy when you implement.

The natural states for this problem is the 3 possible transactions : `buy`, `sell`, `rest`. Here `rest` means no transaction on that day (aka cooldown).

Then the transaction sequences can end with any of these three states.

For each of them we make an array, `buy[n]`, `sell[n]` and `rest[n]`.

`buy[i]` means before day `i` what is the maxProfit for any sequence end with `buy`.

`sell[i]` means before day `i` what is the maxProfit for any sequence end with `sell`.

`rest[i]` means before day `i` what is the maxProfit for any sequence end with `rest`.

Then we want to deduce the transition functions for `buy` `sell` and `rest`. By definition we have:

```
buy[i]  = max(rest[i-1]-price, buy[i-1])
sell[i] = max(buy[i-1]+price, sell[i-1])
rest[i] = max(sell[i-1], buy[i-1], rest[i-1])
```

Where `price` is the price of day `i`. All of these are very straightforward. They simply represents :

```
(1) We have to `rest` before we `buy` and
(2) we have to `buy` before we `sell`
```

One tricky point is how do you make sure you `sell` before you `buy`, since from the equations it seems that `[buy, rest, buy]` is entirely possible.

Well, the answer lies within the fact that `buy[i] <= rest[i]` which means `rest[i] = max(sell[i-1], rest[i-1])`. That made sure `[buy, rest, buy]` is never occurred.

A further observation is that and `rest[i] <= sell[i]` is also true therefore

```
rest[i] = sell[i-1]
```

Substitute this in to `buy[i]` we now have 2 functions instead of 3:

```
buy[i] = max(sell[i-2]-price, buy[i-1])
sell[i] = max(buy[i-1]+price, sell[i-1])
```

This is better than 3, but

we can do even better

Since states of day `i` relies only on `i-1` and `i-2` we can reduce the O(n) space to O(1). And here we are at our final solution:

Java

```
public int maxProfit(int[] prices) {
    int sell = 0, prev_sell = 0, buy = Integer.MIN_VALUE, prev_buy;
    for (int price : prices) {
        prev_buy = buy;
        buy = Math.max(prev_sell - price, prev_buy);
        prev_sell = sell;
        sell = Math.max(prev_buy + price, prev_sell);
    }
    return sell;
}
```

C++

```
int maxProfit(vector<int> &prices) {
    int buy(INT_MIN), sell(0), prev_sell(0), prev_buy;
    for (int price : prices) {
        prev_buy = buy;
        buy = max(prev_sell - price, buy);
        prev_sell = sell;
        sell = max(prev_buy + price, sell);
    }
    return sell;
}
```

For this problem it is ok to use `INT_MIN` as initial value, but in general we would like to avoid this. We can do the same as the following python:

Python

```
def maxProfit(self, prices):
    if len(prices) < 2:
        return 0
    sell, buy, prev_sell, prev_buy = 0, -prices[0], 0, 0
    for price in prices:
        prev_buy = buy
        buy = max(prev_sell - price, prev_buy)
        prev_sell = sell
        sell = max(prev_buy + price, prev_sell)
    return sell
```

written by dietpepsi original link here

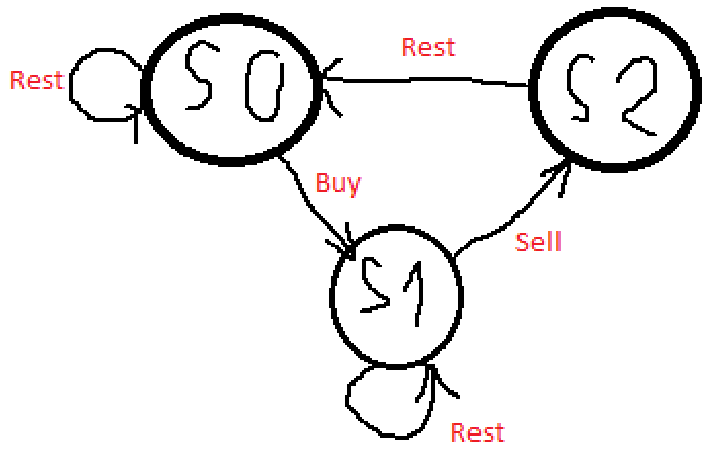
Solution 2

Hi,

I just come across this problem, and it's very frustating since I'm bad at DP.

So I just draw all the actions that can be done.

Here is the drawing (Feel like an elementary ...)



There are three states, according to the action that you can take.

Hence, from there, you can now the profit at a state at time i as:

```
s0[i] = max(s0[i - 1], s2[i - 1]); // Stay at s0, or rest from s2
s1[i] = max(s1[i - 1], s0[i - 1] - prices[i]); // Stay at s1, or buy from s0
s2[i] = s1[i - 1] + prices[i]; // Only one way from s1
```

Then, you just find the maximum of s0[n] and s2[n], since they will be the maximum profit we need (No one can buy stock and left with more profit that sell right :))

Define base case:

```
s0[0] = 0; // At the start, you don't have any stock if you just rest
s1[0] = -prices[0]; // After buy, you should have -prices[0] profit. Be positive!
s2[0] = INT_MIN; // Lower base case
```

Here is the code :D

```
class Solution {
public:
    int maxProfit(vector<int>& prices){
        if (prices.size() <= 1) return 0;
        vector<int> s0(prices.size(), 0);
        vector<int> s1(prices.size(), 0);
        vector<int> s2(prices.size(), 0);
        s1[0] = -prices[0];
        s0[0] = 0;
        s2[0] = INT_MIN;
        for (int i = 1; i < prices.size(); i++) {
            s0[i] = max(s0[i - 1], s2[i - 1]);
            s1[i] = max(s1[i - 1], s0[i - 1] - prices[i]);
            s2[i] = s1[i - 1] + prices[i];
        }
        return max(s0[prices.size() - 1], s2[prices.size() - 1]);
    }
};
```

written by npvinhphat original link here

Solution 3

Here I share my no brainer weapon when it comes to this kind of problems.

---

1. Define States

To represent the decision at index i:

- `buy[i]`: Max profit till index i. The series of transaction is ending with a **buy**.
- `sell[i]`: Max profit till index i. The series of transaction is ending with a **sell**.

To clarify:

- Till index `i`, the **buy** / **sell** action must happen and must be the **last action**. It may not happen at index `i`. It may happen at `i - 1, i - 2, ... 0`.
- In the end `n - 1`, return `sell[n - 1]`. Apparently we cannot finally end up with a buy. In that case, we would rather take a rest at `n - 1`.
- For special case no transaction at all, classify it as `sell[i]`, so that in the end, we can still return `sell[n - 1]`. Thanks @alex153 @kennethliaoke @anshu2.

---

2. Define Recursion

- `buy[i]`: To make a decision whether to buy at `i`, we either take a rest, by just using the old decision at `i - 1`, or sell at/before `i - 2`, then buy at `i`, We cannot sell at `i - 1`, then buy at `i`, because of **cooldown**.
- `sell[i]`: To make a decision whether to sell at `i`, we either take a rest, by just using the old decision at `i - 1`, or buy at/before `i - 1`, then sell at `i`.

So we get the following formula:

```
buy[i] = Math.max(buy[i - 1], sell[i - 2] - prices[i]);
sell[i] = Math.max(sell[i - 1], buy[i - 1] + prices[i]);
```

---

3. Optimize to O(1) Space

DP solution only depending on `i - 1` and `i - 2` can be optimized using O(1) space.

- Let `b2, b1, b0` represent `buy[i - 2], buy[i - 1], buy[i]`
- Let `s2, s1, s0` represent `sell[i - 2], sell[i - 1], sell[i]`

Then arrays turn into Fibonacci like recursion:

```
b0 = Math.max(b1, s2 - prices[i]);
s0 = Math.max(s1, b1 + prices[i]);
```

---

4. Write Code in 5 Minutes

First we define the initial states at `i = 0`:

- We can buy. The max profit at `i = 0` ending with a **buy** is `-prices[0]`.
- We cannot sell. The max profit at `i = 0` ending with a **sell** is `0`.

Here is my solution. Hope it helps!

```
public int maxProfit(int[] prices) {
    if(prices == null || prices.length <= 1) return 0;

    int b0 = -prices[0], b1 = b0;
    int s0 = 0, s1 = 0, s2 = 0;

    for(int i = 1; i < prices.length; i++) {
        b0 = Math.max(b1, s2 - prices[i]);
        s0 = Math.max(s1, b1 + prices[i]);
        b1 = b0; s2 = s1; s1 = s0;
    }
    return s0;
}
```

written by yavinci original link here