Vector Thinking For Max Profit

Solving four variations of Best Time to Buy and Sell Stock leetcode questions:

1. Best Time to Buy and Sell 1 transaction (121)
2. Best Time to Buy and Sell unlimited transactions (122)
3. Best Time to Buy and Sell 2 transactions (123)
4. Best Time to Buy and Sell 2 transactions (188)

Setup:

- Given a list of prices of a security, price[i] is price on day i
- Find the max profit you could have made
- Constraints:
    - you must buy before you sell
    - you can never hold more than 1 unit

Q for Mortals solution to version 1

Jeff Borror gives an excellent version in an offhand explanation of qSQL

```
q)select max px-mins px from trades where sym=`aapl
```

Two hints if Zen enlightenment is slow to dawn:

- Take the perspective of looking back from a potential optimum sell
- The optimum buy must happen at a cumulative local minimum; otherwise, you could back up to an earlier, lower price and make a larger profit

These two hints describe the essential qualities of a dynamic programming problem

- Optimal substructure
- Overlapping subproblems

Classically these properties map to a solution involving:

- Recursion
- Memoization

We can do better using `scan` and `over` iterators via a bottom up approach:

- find the shortest path through the computation**^1**.

Let's warm up by looking a bit more closely at the 1 transaction version

```
/p prices
{[p]max p - mins p}
```

In [2]:
```
#example prices
       kprices=kx.q('3 2 5 1 3 2 4 9 5 12 3 5')
prices=kprices.py()
prices
```

Out[2]:

```
[3, 2, 5, 1, 3, 2, 4, 9, 5, 12, 3, 5]
```

- 3 -> 0
    - Given just one price we can only buy and sell on the same day, so effectively max profit is 0
- 3 2 -> 0
    - Given these two points we can still only earn 0, since we must sell after we buy.
- 3 2 5 -> 3
    - Now that we have a higher number after a lower number we can plainly see that of the two options buy at 3 or buy at 2, buy at 2 and sell at 5 is better.

- 3 2 5 1 -> 3
    - The answer is still 3, since we can't sell higher than 5, and even if we bought at 1 at the end there are no days left to sell
- 3 2 5 1 3 -> 3
    - We still do best by buying at 2 and selling at 5, buying at 1 and selling at 3 only earns 2.
- 3 2 5 1 3 2 -> 3
    - Plainly the 2 at the end doesn't improve

- 3 2 5 1 3 2 4 -> 3
    - We can now either buy at 2 and sell at 5 or buy at 1 and sell at 4, but our max profit is still 3.
- 3 2 5 1 3 2 4 9 -> 8
    - Finally, the max profit changes! We can now buy at 1 and sell at 9. As new elements are added the new reward will be a function of the lowest element seen thus far.

In [6]:
```
%%q
      p:3 2 5 1 3 2 4 9 5 12 3 5 /prices
{[p] max p - mins p} p
```

```
::
11
```

In [7]:
```
%%q
      d:`px`mins_px`p_sub_mins_px`maxs_p_sub_minsp!(
  p; /prices
  {[p]mins p} p; /mins prices
  {[p] p - mins p} p; /p - mins prices
  {[p] maxs p - mins p} p) /maxs p - mins prices
d
```

```
::
px              | 3 2 5 1 3 2 4 9 5 12 3  5
mins_px         | 3 2 2 1 1 1 1 1 1 1  1  1
p_sub_mins_px   | 0 0 3 0 2 1 3 8 4 11 2  4
maxs_p_sub_minsp| 0 0 3 3 3 3 3 8 8 11 11 11
```

In [8]:
```
#or in python we can write something like:
      import itertools
from operator import sub
def max_profit(p):
    return max(map(sub,p,itertools.accumulate(p,min)))
```

Unlimited Transactions case II

We can benefit from any positive movement since we can only sell after we buy

- Keeping the same price series as before:
    - 3 2 5 1 3 2 4 9 5 12 3 5
- Let’s look at a smaller example to get some intuition.
- 3 2 5 -> 3
    - we buy at 2 and sell at 5, purchasing at 3 doesn’t improve our profit since selling at 2 would incur a loss.
- 3 2 5 1 3 -> 5
    - buy at 2 sell at 5, buy at 1 sell at 3

- 3 2 5 1 3 2 4 -> 7
    - just add one more purchase at 2 and sell at 4,
- 3 2 5 1 3 2 4 9 -> 12
    - Here it becomes interesting, we can look at two interpretations
        - buy at 2 sell at 5 (3),buy at 1 sell at 3 (2), buy 2 sell at 9 (7) for a total of 12
        - buy at 2 sell at 5 (3),buy at 1 sell at 3 (2), buy at 2 sell at 4 (2),buy at 4 sell at 9 (5) for a total of 12.

THE TWO APPROACHES ARE IDENTICAL SINCE ADDITION COMMUTES, IT DOESN'T MATTER HOW YOU GET FROM 2 – 9 YOU WILL ALWAYS EARN 7.

Which means that we can simply add up the positive steps in the price series. That will be the maximum profit for an unlimited number of transactions:

In [9]:
```
%%q
      /sum all the positive adjacent differences
{[p] (p>0) wsum p:1 _ deltas p} 3 2 5 1 3 2 4 9 5 12 3 5
```

```
::
21
```

Let's look at version III 2 transactions

Recognize the symmetry: Divide and Conquer

The transactions can't overlap

In [10]:
```
#assume max_profit function as before:
        def max_profit(p):
    return max(map(sub,p,itertools.accumulate(p,min)))
r=range(len(prices))
max([max_profit(prices[0:i+1])+max_profit(prices[i:]) for i in r])
```

Out[10]:

```
15
```

In [11]:
```
%%q
        maxprofit:{[p] max p - mins p}
max (maxprofit each _[1] scan p)+maxprofit each reverse _[-1] scan p
```

```
::
15
```

We are calculating max profit over prefixes and suffixes

We already spent sometime understanding how max profit prefixes behave

Let's look at how suffixes behave

- To get a sense of this, look at the solutions to the suffixes
- 3 5 -> 2 (buy at 3 sell at 5)
- 12 3 5 -> 2 (buy at 3 sell at 5)
- 5 12 3 5 -> 7 (buy at 5 sell at 12)
- 9 5 12 3 5 -> 7 (buy at 5 sell at 12)
- 4 9 5 12 3 5 -> 8 (buy at 4 sell at 12)

We can see that the suffixes are governed by the largest element we can sell into,
i.e. the rolling max and the current element at the left
Which leads us to this solution in q:

```
In [23]: %%q
         {max reverse[maxs maxs[x]-x:reverse x]+maxs x-mins x} p
```

```
15
```

Finally Let's Tackle k transactions IV

THIS TIME WE CAN START WITH THE RECURSIVE SOLUTION

To quote MIT's Erik Demaine CS6006 dynamic programing is recursion plus memoization.

- Let’s setup the recursive solution:
- and assume we are at a particular (i)ndex in our price series, with k transactions left.
- Let’s start with the base cases:
    - If k equals 0
        - return 0
    - If i is greater than the last index, i.e. there are no elements left in the list:
        - return 0

- Otherwise the solution to the problem is simply the maximum of 2 options:
    - do nothing at this step:
        - 0+the function increment i
    - do something:
        - If we are (h)olding a share we can sell
            - which adds the current price + the result of:
                - function with one less k and i incremented
            - we are no longer holding a share
        - Otherwise we buy at this step,
            - subtract current price (we spend money to buy) + the result of
                - this function with i incremented
            - we are holding a share

In [13]:
```
%%q
        cache:(0b,'0,'til[count p])!count[p]#0;
f:{[h;k;i]$[i=count p;0;(h;k;i) in key cache;cache[(h;k;i)];
 :cache[(h;k;i)]:.z.s[h;k;i+1]|$[h;p[i]+.z.s[0b;k-1;i+1];.z.s[1b;k;i+1]-p i]]}
```

::
::

We can test this and see that it results in the right answer for k=0,1,2

In [14]:
```
%%q
        f[0b;0;0] /0 no surprise with 0 transaction no profit is possible
f[0b;1;0] /11 the original problem
f[0b;2;0] /15 buy at 1, sell at 9, buy at 5 sell at 12
f[0b;3;0] /??
f[0b;4;0] /??
f[0b;5;0] /??
```

```
0
11
15
18
20
21
```

Can we do better?

However, we know that a vector solution is possible for k=1,2,infinity,

- We might hope there is a vector solution for when k is 3 and above.
- We can analyze the intermediate results of the cache to get some sense of this.

In [15]:
```
%%q
        t:(flip `h`k`j!flip key cache)!([]v:value cache);
exec (`$string[asc j])!v by k from t where not h
```

```
::
 | 0 1 2 3 4 5  6  7  8  9  10 11
-| ------------------------------
0| 0 0 0 0 0 0  0  0  0  0  0  0
1| 0 2 2 7 7 8  10 10 11 11 11 11
2| 0 2 2 9 9 12 14 14 15 15 15 15
3| 0 2 2 9 9 14 16 16 17 17 18 18
4| 0 2 2 9 9 14 16 16 18 18 20 20
5| 0 2 2 9 9 14 16 16 18 18 21 21
```

```
In [16]: %%q
        t:(flip `h`k`j!flip key cache)!([]v:value cache);
exec (`$string[asc j])!v by k from t where not h
```

```
::
 | 0 1 2 3 4 5  6  7  8  9  10 11
-| ------------------------------
0| 0 0 0 0 0 0  0  0  0  0  0  0
1| 0 2 2 7 7 8  10 10 11 11 11 11
2| 0 2 2 9 9 12 14 14 15 15 15 15
3| 0 2 2 9 9 14 16 16 17 17 18 18
4| 0 2 2 9 9 14 16 16 18 18 20 20
5| 0 2 2 9 9 14 16 16 18 18 21 21
```

We notice that each row is the previous row + 1 additional transaction

Putting this into action we get the following:

In [17]:
```
%%q
        maxprofitk:{[k;p]last {maxs x+maxs y-x}[p]/[k;p*0]}
maxprofitk[0;p]
maxprofitk[1;p]
maxprofitk[2;p]
maxprofitk[3;p]
maxprofitk[4;p]
maxprofitk[5;p]
```

```
::
0
11
15
18
20
21
```

Finally if we take a closer look at the inner function

```
{[p;c] maxs p +maxs c - p }[p]
```

we might notice something funny: `maxs c - p` is the same as `mins p -`
we can omit the current state variable

Which leads us to this super elegant solution

```
In [24]: %%q
         k:3
p:3 2 5 1 3 2 4 9 5 12 3 5
P:maxs p- mins p- /General Max Profit expresion
/solutions
P 0 /1 transaction
P P 0 /2 transactions
P/[k;0] /k transactions
P/[0] /unlimited transactions
```

```
::
::
::
::
0 0 3 3 3 3 3 8 8 11 11 11
0 0 3 3 5 5 6 11 11 15 15 15
0 0 3 3 5 5 7 12 12 18 18 18
0 0 3 3 5 5 7 12 12 19 19 21
```

Questions?

Appendix

In [3]:
```
#https://leetcode.com/problems/best-time-to-buy-and-sell-stock/solutions/1956412/python3-tc
    def max_profit(prices: list[int], k=1) -> int:
    @lru_cache(None)
    def rec(i, k, f):
        if k == 0 or i == len(prices):
            return 0
        c1 = rec(i+1, k, f) # we dont buy or sell
        c2 = 0
        if f:
            c2 = rec(i+1, k-1, False) + prices[i]  # we sell
        else:
            c2 = rec(i+1, k, True) - prices[i]  # we buy
        return max(c1, c2)

    return rec(0, k, False)  # k == 1, means, can do only 1 transaction
max_profit(prices)
```

Out[3]:

```
11
```

In [4]:
```
%%q
       /classical version using memoization
.memo.M:{[f].memo.f[f]:()!();{[f;x]$[any x~/:key .memo.f[f];.memo.f[f;x];:first .memo.f[f;enlist x
rec:.memo.M {[x] p:x`p;i:x`i;k:x`k;f:x`f;
     if[(i=count p)|k=0;:0];
    c1:rec[`p`i`k`f!(p;i+1;k;f)];
    c2:$[f;
         rec[`p`i`k`f!(p;i+1;k-1;0b)] + p i;
         rec[`p`i`k`f!(p;i+1;k;1b)] - p i];
     :c1|c2};
maxProfit:{[prices] rec[`p`i`k`f!(prices;0;1;0b)]};
maxProfit 3 2 5 1 3 2 4 9 5 12 3 5
```

```
::
::
::
::
11
```

In [5]:
```
#or from python calling q
       qmaxProfit=kx.q('maxProfit')
qmaxProfit(kprices).py()
```

Out[5]:

11