Clearly DP is one of solutions to this kind of problems. I will like to summarize how to start and analyze for beginners:

Goal: find the max profit at day n-1 with at most k transactions.  
Here we know we have two keys: day and transaction, of course it depends on past history (previous day with certain transactions), so if you try to solve this in a pure math. calculation and struggle with so many possibilities, that means we had better switch to consider how to build the math. model.

Notations:  
f[k][i]: max profit up to day i (included) with at most k transactions (global optimal objective)  
g[k][i]: max profit up to day i (included) with at most k transactions AND we sell at day i (local optimal objective, why local? think about it!)

DP recursive formula:  
(1). f[k][i] = max ( f[k][i-1], g[k][i] )  
(2). g[k][i] = max\_{j=0,...,i-1} (p[i] - p[j] + f[k-1][j-1])  
// (1): this means if we don't sell at day i, then f[k][i] is just f[k][i-1]; otherwise f[k][i] will be the max profit that we can achieve if we sell at day i  
// (2): since we will sell at day i anyway, that means we need to buy at a certain previous day, for a particular j, the best we can have is p[i] - p[j] + f[k-1][j-1].

With these formulas, we can start to code by computing and storing all the information of f[k][i] and g[k][i]. One trick is that for (2), the max is easy to handle because we compute g[k][i] for i=0,1,2,... (with fixed k), so use one temp variable to keep updating the value will be enough.

If we think carefully, (1) and (2) only use the information on k-1, that means we can reduce the space complexity O(k\*n) to O(n) pretty easily.

Can we reduce the space to be O(k)? The answer is yes. Of course, the formula (2) looks a bit ugly. Let's rewrite it a bit:  
(2)'. g[k][i] = max ( g[k][i-1], f[k-1][i-1]) ) + p[i] - p[i-1].

Now let me put (1) and (2)' together:  
(1). f[k][i] = max ( f[k][i-1], g[k][i] )  
(2)'. g[k][i] = g[k][i-1], f[k-1][i-1]) + p[i] - p[i-1]  
It's a lot simpler, right? The key is to get the current state (k,i), we only need the k-1 or i-1 information. So in the code, we have two loops but with outer: i-loop and inter: k-loop. This will makes space to be O(k). If k is way more small than n, then you can consider it's a good improvement.

You might come up (1) and (2)' directly, but here I just want to explain what happened in peterleetcode's post. So I start with his formulation. Finally, attach my code (may not be optimized) Whoops, this code is for IV, but should work for III case also.

**class** **Solution** {

**public**:

**int** **maxProfit**(**int** k, vector<**int**>& prices) {

**int** n=prices.size();

**if** (n<2) **return** 0;

**if** (k>=n/2) { // buy-sell-II case, unlimited

**int** maxProfit=0;

**for** (**int** i=1; i<n; ++i) {

**if** (prices[i]>prices[i-1]) maxProfit += prices[i] - prices[i-1];

}

**return** maxProfit;

}

// if k<n/2, use DP approach but only O(k) space

vector<**int**> f(k+1,0), g(k+1,0);

**for** (**int** i=1; i<n; ++i) {

**int** diff = prices[i] - prices[i-1], temp = f[0];

**for** (**int** kk=1; kk<=k; ++kk) {

g[kk] = max(g[kk], temp) + diff;

temp=f[kk];

f[kk] = max(f[kk], g[kk]);

}

}

**return** f[k];

}

};

PS: be careful in (2)', we have f[k-1][i-1], so before updating f[kk], we should save it for later use. Or I believe you can use different order of k to remove this "ugly" part.