**Problem 3:**

Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer c (positive), find the maximum profit with no restriction on the number of transactions. However, you cannot buy any stock for c days after selling any stock. If you sell a stock at day i, you are not allowed to buy any stock until day i+c+1.

**Task:**

Algorithm 7: Design a Θ(m ∗ 2n) time brute force algorithm for solving Problem3

Algorithm 8: Design a Θ(m ∗ n2) time dynamic programming algorithm for solving Problem3

Algorithm 9: Design a Θ(m ∗ n) time dynamic programming algorithm for solving Problem3

**Algorithm 7:**

**Design and Analysis of Algorithms:**

For the brute force algorithm, the approach is the to create a recursive function which tries to compute maximum profit obtained by finding all the possible transactions.

**Proof of correctness:**

This algorithm checks all the possible transactions possible for each day and finds out maximum profit possible from all possibilities. Hence, this algorithm will always find the correct desired output. Brute force algorithms always find the correct solution. All the possible transaction on current day can be found as follow:

* If a stock has already been bought than it can either sold or retained on that day.
* Otherwise, any of the stock can be bought on a particular day considering no transaction have taken place since last c days or we can skip buying any stock on that day.

**Time complexity analysis:**

Time complexity for the given task can be calculated as follows:

First iterating over all days. Then finding maximum profit from all possible states at that day. For each day there are at 2 states possible. So, time complexity is

N = Total number days.

M = Total number of Stocks

T(n, m) = {

| If Stock has been already bought, | 2\*T(N-1) + θ(M) |
| --- | --- |
| If no stock has been bought, | 2\*T(N-1) |
| } |  |

So the time complexity for this algorithm will be: O (M \* 2**N**)

**Space Complexity analysis:**

The algorithm stores the stocks and their prices in a 2d array of size m\*n. It also stores the all possible transactions. So maximum possible transactions to store is N/(C+2). Each transaction has 4 integers stored in it. So overall space size comes up to N/(C+2)\*4.

So the space complexity for this algorithm will be: O (m \* n + N/(C+2)\*4) = **O (m \* n)** [Here C is a cooldown period]

**Pseudo Code:**

**Step 1 :** Input A[m][n], C

**Step 2 :** profit = 0, stock = 0, buy = -1, sell = -1

**Step 3 :** Function BruteForce2n(current\_day, bought\_stock , bought\_day, Buy)

Base case: current\_day >= N

Return 0

If Buy == True

Transactions\_didnt\_buy = BruteForce2n(A, current+1, -1, -1, buy, c)

Transactions\_did\_buy = for i = 1 to m (MAX(BruteForce2n(A, current+1, i, current, false, c) ))

Return MAX( Transactions\_didnt\_buy, Transactions\_did\_buy)

else

Transactions\_didnt\_sell= BruteForce2n(A, current+1, S, D, false, c)

Transactions\_did\_sell = BruteForce2n(A, current+c+1, -1, -1, buy, c)

Transactions\_did\_sell.add(new transaction {S, A[S][current] - A[S][D], D, current});

return MAX(Transactions\_didnt\_sell, Transactions\_did\_sell);

**Algorithm 8:**

**Design and Analysis of Algorithms:**

**Solution:**

For dynamic programming, we intend to find out maximum profit using compatible transactions using all the stocks.

Def: OPT(M,N) - Maximum profit earned by using first m stocks until n days.

Goal: Max profit earning at OPT (m,n)

Where m and n represent total stocks and total days.

**Bellman Equation:**

OPT( M , N ) = {

0 , n = 0;

Max ( OPT( M , N-1 ) , OPT( M - 1 , N ) , Max\* ( A[ M ][ N ] - A[ M ][ k ] + OPT[ m -1 ][ Max(0, k - c - 1) ) k = 0 to N-1 ), Otherwise;

}

Here, m is total number of stocks.

**Proof of correctness:**

Invariant: OPT( m , n ) gives maximum profit for first m stock and n th day.

Proof: Proof by induction.

Base case: n = 1 is then maximum profit is 0.

Next case: k = 2, n = 2 maximum profit is greatest of following values:-

* OPT( m , n-1 )
* OPT( m-1 , n )
* Max( A[ M ][ N ] - A[ M ][ k ] + OPT[ m -1 ][ Max(0, k - c - 1) ) ) where M = 0 to m.

Inductive Hypothesis: Assuming OPT[ m ][ n ] gives maximum profit after i transactions.

If **case 1**: profit earned at day n-1 is highest. In this case stock m is not sold on day n.

If **case 2**: profit earned using first m-1 stocks is highest. In this case stock m is not sold on day n.

If **case 3**: profit earned by selling stock m is highest. In this case, stock m is sold on day n. Also, maximum profit is earned by selling on day k. So, along with profit from this transaction we can also add profit earned by using all stocks and day k - c.

[c is cooldown period]

Thus, this shows that all possibility of generating a maximum profit at day n has been covered in by case 1, 2 and 3. So, for any value of m and n, maximum profit can be achieved by using previously calculated optimal values from case 1, 2 and 3.

So, by induction, we can deduce that for any value of m and n can obtain its optimal value through this efficient algorithm.

**Time complexity analysis:**

Time complexity for the given task can be calculated as follows we iterate n to find buy and sell day which has complexity of (n)\*(n-1)/2 and all these operations are done m times because it is done for each stock.

So the time complexity for this algorithm will be: O (M \* N \* (N - 1) / 2 ) = **O (M \* N2)**

**Space Complexity analysis:**

The algorithm stores the stocks and their prices in a 2d array of size m\*n.

So space complexity of this algorithm is **O (m \* n )**

**Pseudo Code**:

**Task8(A, m, n, c)**

**Step 1:** DP, DiffSoFar, Buy → Φ

**Step 2:**

For j = 1 to n:

For i = 1 to m:

For k = 1 to j:

Current = DP[i][j]

PreviousDayProfit = DP[i][j-1]

previousStockProfit = i == 0 ? 0 : DP[i-1][j]

profitIfWeBuy = A[i][j] + DP[m-1][max(0,k-c-1]

DP[i][j] = max(PreviousDayProfit, previousStockProfit,profitIfWeBuy )

If current != DP[i][j] then

Buy[i][j] = k

**Step 3: return BACKTRACKERSULT(OPT, BUY, A)**

**BACKTRACKERSULT(OPT, BUY, A)**

**Step 1:** i = m-1, j = n-1, result = Φ

**Step 2:** While(j is not 0)

Current = OPT[i][j

If i is zero and OPT[i][j] == OPT[i][j-1]

j -= 1

else If i > 0 and OPT[i][j] == OPT[i-1][j]

i -= 1

else if OPT[i][j] == OPT[i][j-1]

j -= 1

else

// Transaction occurred

result = result U { { i, A[i][j] - A[i][Buy[i][j]], Buy[i][j], j} }

// Go to profit where we buy from

j = buy[i][j] - c - 1

i = m - 1

**Step 3:** return result

**MAIN()**

**Step 1 :** Input c, m, n, A[m][n]

**Step 2 :**

result = Task8(A, m, n, c)

**Step 3:** Return result

**Algorithm 9A:**

**Design and Analysis of Algorithms:**

**Proof of correctness:**

This algorithm checks all the possible transactions possible for each day and finds out maximum profit possible from all possibilities. Hence, this algorithm will always find the correct desired output. Brute force algorithms always find the correct solution. All the possible transaction on current day can be found as follow:

* If a stock has already been bought than it can either sold or retained on that day.
* Otherwise, any of the stock can be bought on a particular day considering no transaction have taken place since last c days or we can skip buying any stock on that day.

**Time complexity analysis:**

Time complexity for the given task can be calculated as follows:

N = Total number days.

M = Total number of Stocks

T(n, m) = {

T(N-1, M) + T(N,M-1) + T(N-C, m-1) + θ(1)

}

So, the time complexity for this algorithm will be: O (m \* n)

**Pseudo Code:**

**OPT(M,N)= {**

**N=0, 0**

**M=0, max( opt(M,N-1), opt(m, max(1,k-c)-A[i][k]+A[i][j] ) )**

**Otherwise, max(opt(M-1,N), opt(M,N-1), maxProfit[M] +A[M][N] )**

}

**Algorithm 9:**

**Design and Analysis of Algorithms:**

**Solution:**

For dynamic programming, we intend to find out maximum profit using compatible transactions using all the stocks.

Def: OPT(M,N) - Maximum profit earned by using first m stocks until n days.

Goal: Max profit earning at OPT (m,n)

Where m and n represent total stocks and total days.

**Bellman Equation:**

OPT( M , N ) = {

0 , n = 0;

Max ( OPT( M , N-1 ) , OPT( M - 1 , N ) , A[ M ][ N ] - A[ M ][ k ] + OPT[ m -1 ][ Max(0, k - c - 1) where k is stored and updated whenever we encounter max profit while buying at day k for m stock), Otherwise;

}

Here, m is the total number of stocks.

**Proof of correctness:**

Invariant: OPT( m , n ) gives maximum profit for first m stock and n th day.

Proof: Proof by induction.

Base case: n = 1 is then maximum profit is 0.

Next case: k = 2, n = 2 maximum profit is greatest of following values:-

* OPT( m , n-1 )
* OPT( m-1 , n )
* Max( A[ M ][ N ] - A[ M ][ k ] + OPT[ m -1 ][ Max(0, k - c - 1) ) ) where M = 0 to m.

Inductive Hypothesis: Assuming OPT[ m ][ n ] gives maximum profit after i transactions.

If **case 1**: profit earned at day n-1 is highest. In this case stock m is not sold on day n.

If **case 2**: profit earned using first m-1 stocks is highest. In this case stock m is not sold on day n.

If **case 3**: profit earned by selling stock m is highest. In this case, stock m is sold on day n. Also, maximum profit is earned by selling on day k. So, along with profit from this transaction we can also add profit earned by using all stocks and day k - c.

[c is cooldown period]

Thus, this shows that all possibility of generating a maximum profit at day n has been covered in by case 1, 2 and 3. So, for any value of m and n, maximum profit can be achieved by using previously calculated optimal values from case 1, 2 and 3.

So, by induction, we can deduce that for any value of m and n can obtain its optimal value through this efficient algorithm.

**Time complexity analysis:**

Time complexity for the given task can be calculated as follows. We iterate each day while also iterating on each stock.

So the time complexity for task 9a and 9b will be: O (m \* n ) = **O (m \* n)**

**Space Complexity analysis:**

The algorithm stores the stocks and their prices in a 2d array of size m\*n. So the space complexity for task 9a and 9b will be: O (m \* n ) = **O (m \* n)**

**Pseudo Code:**

**Task9A(A, m, n, c)**

**Step 1:** DP, DiffSoFar, Buy → Φ

**Step 2: DPMemoizationMN(A, DiffSoFar, DP, Buy, m, n, c)**

**Step 3:** return **BACKTRACKERSULT(OPT, BUY, A)**

**DPMemoizationMN(A, DiffSoFar, DP, Buy, M, N, c)**

**Step 1:** if N <= 0

DP[M][N] = 0

Return DP[M][N]

**Step 2:** Current = DP[M][N]

**Step 3:** PreviousDayProfit = DP[M][N-1] == 0 ? **DPMemoizationMN(A, DiffSoFar, DP, Buy, M, N-1, c) :** DP[M][N-1]

**Step 4:** previousStockProfit = M <= 0 ? 0 : DP[M-1][N] == 0 ? DPMemoizationMN(A, DiffSoFar, DP, Buy, M-1, N, c) : DP[M-1][N]

**Step 5:** profitIfWeBuy = DP[m-1][max(0, N-c-2)] == 0 ? DPMemoizationMN(A, DiffSoFar, DP, Buy, m-1, max(0, N-c-2), c) : DP[m-1][max(0, N-c-2)]

**Step 6:**

currentDiff = profitIfWeBuy - A[M][N-1]

If currentDiff > DiffSoFar[M][0]

DiffSoFar[M][0] = currentDiff

DiffSoFar[M][1] = N-1

**Step 7:**

DP[M][N] = max(PreviousDayProfit, previousStockProfit, A[M][N] + DiffSoFar[M][0])

**Step 8:**

If current != DP[M][N]

buy[M][N] = DiffSoFar[M][1]

**Step 9:**

Return DP[M][N]

**Task9B(A, m, n, c)**

**Step 1:** DP, DiffSoFar, Buy → Φ

**Step 2:**

For j = 1 to n:

For i = 1 to m:

currentDiff = DP[m-1][max(0, j-c-2)] - A[i]j-1]

If currentDiff > DiffSoFar[i][0]

DiffSoFar[i][0] = currentDiff

DiffSoFar[i][1] = j-1

Current = DP[i][j]

PreviousDayProfit = DP[i][j-1]

previousStockProfit = i == 0 ? 0 : DP[i-1][j]

profitIfWeBuy = A[i][j] + DiffSoFar[i][0]

DP[i][j] = max(PreviousDayProfit, previousStockProfit,profitIfWeBuy )

If current != DP[i][j] then

Buy[i][j] = DIffSoFar[i][1]

**Step 3: return BACKTRACKERSULT(OPT, BUY, A)**

**BACKTRACKERSULT(OPT, BUY, A)**

**Step 1:** i = m-1, j = n-1, result = Φ

**Step 2:** While(j is not 0)

Current = OPT[i][j

If i is zero and OPT[i][j] == OPT[i][j-1]

j -= 1

else If i > 0 and OPT[i][j] == OPT[i-1][j]

i -= 1

else if OPT[i][j] == OPT[i][j-1]

j -= 1

else

// Transaction occurred

result = result U { { i, A[i][j] - A[i][Buy[i][j]], Buy[i][j], j} }

// Go to profit where we buy from

j = buy[i][j] - c - 1

i = m - 1

**Step 3:** return result

**MAIN()**

**Step 1 :** Input c, m, n, A[m][n]

**Step 2 :**

result = Task9A(A[i], m, n, c) // *BOTTOM UP*

result = Task9B(A[i], m, n, c) // *MEMOIZATION*

**Step 3:** Return result