**Problem Solution - ALG3**

**Problem Definition:**

Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer k (positive), find a sequence of at most k transactions that gives maximum profit. [Hint :- Try to solve for k = 2 first and then expand that solution.]

**Algorithm 1:** Design a Θ(m ∗ n ∗ k) time dynamic programming algorithm for solving Problem

**Task:**

**Task 6a** Give a recursive implementation of Alg6 using Memoization.

**Task 6b** Give an iterative BottomUp implementation of Alg6.

**Solution:**

For dynamic programming, we intend to find out maximum profit using k or less than k transactions and using all the stocks.

Let K be the maximum transactions possible and N be total days available.

Def: OPT(k,n) - Maximum profit earned by doing at most k transactions until n days using any stocks.

Goal: Max{ OPTi (K,N) } where i = 1 to m

Case 1: OPT(k,n) sells the stock on day n

- Current profit will be maximum of two choices

1. OPT( k , n-1)
2. max( OPT( k-1, n-1 ) - A[ m ][ n-1 ] + A[ m ][ n ] ) )

profit on day n-1 and transaction k-1 added with transaction profit

- If current profit is greater than the max profit so far, set max profit to current profit

Case 2: OPT(k,n) does not sell the stock on day n

- Current profit must be equal max profit made from so far i.e. OPT(k,n-1)

**Bellman Equation:**

**OPT[k][n] = {**

0, k = 0 or n = 0

Max {

OPT[k][n-1],

Max { Max\* { OPT[k-1][n-1] - A[M][n-1] } + A[M][n] } for M = 1 to m

}

**}**

**\* -** We store the difference value i.e max yet, which results in reduction of time complexity by removing a `for` loop.

**Proof of correctness:**

Invariant: OPT[k][n] gives maximum profit for at most i transactions and j th day.

Proof: Proof by induction.

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

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

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

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

If **case 1**: If profit by selling is less, the OPT will return the OPT( k, n-1) value which is maximum profit earned without inclusion of day n.

If **case 2**: if profit by selling on day n is more than case 1, that means one of stocks sold on day n will generate more profit that previous day.

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

So, by induction, we can deduce that for any value of k 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: First iterating over possible values of number of transactions i.e. k. For each day, it calculates maximum profit that is possible to earn while iterating over all stocks i.e. m.

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

**Space Complexity analysis:**

The algorithm stores the stocks and their prices in a 3d array of size m\*k\*3. For each transaction and day it stores 3 integers namely maximum profit earned at that point, stock sold to make maximum profit, Day at which Stock was bought.

For **memoization**:

So the space complexity for the algorithm will be:

O (m \* k \* 3 + c) = **O (m \* k \* 3)** [Here c is a constant]

For **top down approach**:

So the space complexity for the algorithm will be:

O (m \* k \* 3 + c) = **O (m \* k \* 3)** [Here c is a constant]

**Pseudo Code:**

**Memoization:**

**Task6A(A, m, n, k)**

**Step1:** OPT → Ø, DiffSoFar → Ø

// Both OPT and DiffSoFar are 2D arrays where each element is a set of three and two objects respectively

// OPT[i][j] = {profit, Stock, buy}, we wil access them as OPT[i][j].profit etc.

// DiffSoFar[i][j] = { Difference, day }, we will access them same as OPT

**Step 2: DPMemoization(A, OPT, DiffSoFar, k, n) // Populate OPT**

**Step 3:** return **BackTrack(OPT)**

**DPMemoization(A, OPT, DiffSoFar, current transaction index T, current day D)**

**Step1:** if OPT[i][j] is not Ø then return OPT[i][j] // As this is recursive we can have multiple calls to same sub proble,

**Step 2:** if T is 1 or D is 1 then

OPT[T][D] → {0, 0, 0} // initialize base case

Return OPT[T][D]

**Step 3:** If T > D then

OPT[T][D] = DPMemoization(A, OPT, DiffSoFar, T-1, D)

// If T > D i.e. Transactions greater than days we need to return previous transaction value which is obvious

else

NoTransactionTodayProfit → DPMemoization(A, OPT, DiffSoFar, T, D-1)

maxProfit → NoTransactionTodayProfit.profit

Stock → NoTransactionTodayProfit.stock

Buy → NoTransactionTodayProfit.buy

IfTransactionTodayProfit → DPMemoization(A,OPT, DiffSoFar, T-1, D-1).profit

For i = 1 to m:

Diff → IfTransactionTodayProfit - A[i][D-1]

If Diff > DiffSoFar[T][i].difference then

DiffSoFar[T][i].difference = Diff

DiffSoFar[T][i].day = D-1

Current profit → A[i][D] + DiffSoFar[T][i].profit

If maxProfit < Current Profit then

maxProfit = Current profit

Stock = i

Buy = DiffSoFar[T][i].day

OPT[T][D] ← { maxProfit, Stock, Buy }

**Step 4:** return OPT[T][D]

**Bottom-up Approach:**

**Task6B(A, m, n, k)**

**Step 1:** OPT → Ø

**Step 2:** For i = 1 to k:

DiffSoFar = Φ // Here DiffSoFar is same as above just as a Single Dimensional array with same properties as in 6A

For j = 1 to n:

If i > j

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

Continue

maxProfit → OPT[i][j-1].profit

Stock → OPT[i][j-1].stock

Buy → OPT[i][j-1].buy

For p = 1 to m:

Diff → OPT[i-1][j-1].profit- A[p][j-1]

If Diff > DiffSoFar[p].difference then

DiffSoFar[p].difference = Diff

DiffSoFar[p].day = j-1

Current profit → A[p][j] + DiffSoFar[p].profit

If maxProfit < Current Profit then

maxProfit = Current profit

Stock = p

Buy = DiffSoFar[p].day

OPT[T][D] ← { maxProfit, Stock, Buy }

**Step 3:** return **BACKTRACKERSULT(OPT)**

**BACKTRACKERSULT(OPT)**

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

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

Current = OPT[i][j

If i is zero // base case

j -= 1

else If j is zero //base case

i -= 1

else if OPT[i][j].profit == OPT[i][j-1].profit // previous day

j -= 1

else

// Transaction occurred

result = result U { { current.stock, current.profit, current.buy, j } }

// Go to profit where we buy from

j = buy

i -= 1

**Step 3:** return result

**MAIN()**

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

**Step 2 :**

result = Task6A(A[i], m, n, k) // *BOTTOM UP*

result = Task6B(A[i], m, n, k) // *MEMOIZATION*

**Step 3:** Return result