Final Project COP4533

Anthony Mezzatesta
Developer

Luis Gonzalez
Developer

Jonathan Palma
Developer

Mert Dayi
Project Manager / Developer

Task 1

Brute Force Problem 1)

```
function brute_force_prob1(A):
    m = rows in A
    n = columns in A
    best_profit = 0
    best_stock = 0
    best_buy = 0
    best_sell = 0

    # Iterate over each stock
    for i from 0 to m-1:
        # Buy day index
        for j from 0 to n-2:
            # Sell day index (must be after buy)
            for k from j+1 to n-1:
                profit = A[i][k] - A[i][j]
                if profit > best_profit:
                    best_profit = profit
                    best_stock = i
                    best_buy = j
                    best_sell = k
    # No profitable transaction
    if best_profit <= 0:
        return (0, 0, 0, 0)
    else:
        # Convert to 1-based indexing
        return (best_stock+1, best_buy+1, best_sell+1, best_profit)
```

Task 1 | Task 2 | Task 3 | Task 4 | Task 6

Task 2

```
Greedy Problem 1)
function greedy_prob1(A):
   m = rows in A
   n = columns in A
   best_profit = 0
   best_stock = -1
   best_buy = -1
   best_sell = -1

         # Iterate over each stock
         for i from 0 to m-1:
                  min_price = A[i][0]
                  min_index = 0
                  current_profit = 0
                  current_buy = 0
                  current_sell = 0

                  # Start from day 1
                  for j from 1 to n-1:
                           # Found new minimum price
                           if A[i][j] < min_price:
                                    min_price = A[i][j]
                                    min_index = j
                           else:
                                    profit = A[i][j] – min_price
                                    # Update if better profit
                                    if profit > current_profit:
                                             current_profit = profit
                                             current_buy = min_index
                                             current_sell = j

                  # Update global best
                  if current_profit > best_profit:
                           best_profit = current_profit
                           best_stock = i
                           best_buy = current_buy
                           best_sell = current_sell

         if best_profit <= 0:
                  return (0, 0, 0, 0)
         else:
                  return (best_stock+1, best_buy+1, best_sell+1, best_profit)
```

Task 3

DP Problem 1)

```
function dp_prob1(A):
    m = rows in A
    n = columns in A
    best_profit = 0
    best_stock = -1
    best_buy = -1
    best_sell = -1

    # Iterate over each stock
    for i from 0 to m-1:
        min_price = A[i][0]
        min_index = 0
        max_profit_i = 0
        buy_i = 0
        sell_i = 0

         # Start from day 1
        for j from 1 to n-1:
            # Update minimum price if lower found
            if A[i][j] < min_price:
                min_price = A[i][j]
                min_index = j
            # Calculate profit if selling today
            profit = A[i][j] - min_price
            if profit > max_profit_i:      # Update if better profit
                max_profit_i = profit
                buy_i = min_index
                sell_i = j

        # Update global best
        if max_profit_i > best_profit:
            best_profit = max_profit_i
            best_stock = i
            best_buy = buy_i
            best_sell = sell_i

    if best_profit <= 0:
        return (0, 0, 0, 0)
    else:
        return (best_stock+1, best_buy+1, best_sell+1, best_profit)
```

Key Algorithmic Concepts

- Exhaustive search (Brute force)
- Single pass efficiency (Greedy)
- Optimal substructure (DP)

Data Structures Used

- 2D array, price matrix. Primitive variables, price trackers (Brute force)
- Tuple for final result (Greedy)
- State variables min_price, max_profit (DP)

Critical Decisions During Implementation

- 0-based indexing converted to 1-based indexing for output (Brute force)
- Immediate profit comparison avoids additional storage (Greedy)
- Avoided DP table in order to save space (DP)

Task 4

Dynamic Programming Algorithm for Problem 2 ($O(m \cdot n^2k)$ Time)

```
function dp_prob2_n2k(A):
    m = number of rows in A
    n = number of columns in A
    k = [number of  operations]

    DP = 3D array of size [m+1][n+1][k+1], initialized to -infinity
    DP[0][0][0] = 0  # or base_case_value

    for i from 1 to m:
        for j from 1 to n:
            for x from 0 to k:
                for p from 1 to n:
                    for q from 0 to k:
                        if valid_transition(p, q, j, x):
                            DP[i][j][x] = max(
                                DP[i][j][x],
                                DP[i-1][p][q] + cost(B, i, j, x)
                            )

    answer = maximum DP[m][j][x] over all valid j, x
    return answer
```

Key Algorithmic Concepts

- Breaks the problem into overlapping subproblems, storing their solutions in a 3D table.
- Considers valid previous states and updates the DP state using the `max()` function.
- Tracks row index `i`, column index `j`, and number of operations `x`.

Data Structures Used

- Stores the optimal result for the first `i` rows, ending at column `j` using `x` operations.
- Encapsulates the cost of transitioning into the current state.
- Makes sure transitions between states are legal based on constraints.

Critical Decisions During Implementation

- Carefully chosen to make sure every valid transition is considered while avoiding out-of-bounds errors.
- The initialization (`DP[0][0][0] = 0`) is crucial for building up the DP values correctly.
- Makes sure the DP always keeps the optimal (maximum) value at each state.

Task 6

```
function dp_prob3_slow(A, c):
    m = rows in A
    n = columns in A
    dp = array of size n initialized to 0
    choice = array of size n initialized to null

    for i from 1 to n-1:
        dp[i] = dp[i-1]
        choice[i] = null

        for j from 0 to i-1:
            prev_profit = (j-c-1 >= 0) ? dp[j-c-1] : 0

            for stock from 0 to m-1:
                profit = A[stock][i] - A[stock][j]
                total = prev_profit + profit

                if total > dp[i]:
                    dp[i] = total
                    choice[i] = (stock, j, prev_profit)

    if dp[n-1] <= 0:
        return []

    result = []
    i = n-1
    while i > 0 and choice[i] != null:
        stock, buy_day, prev_profit = choice[i]
        result.add((stock+1, buy_day+1, i+1))

        j = buy_day - c - 1
        while j >= 0 and dp[j] != prev_profit:
            j = j - 1
        i = j

    reverse(result)
    return result
```

Key Algorithmic Concepts

- Dynamic programming used to store the optimal profit up to each day.
- The backtracking reconstructs the sequqnece of transaction that lead to the optimal profit.
- State Optimization evaluates profit by considering the previous problemas with constraints

Data Structures Used

- 1D array stores max profit up to each day.
- 1D array stores decisions for reconstruction the solution.
- Tuples are used to store stock transaction decisions in the choice array

Critical Decisions During Implementation

- The cooldown constraint (c) affects how far back we look valid previous profits.
- With the choice array for reconstruction we use the optimal path
- The edge case handling returns an empty list when no profitable transactions exist.

Thank you!