To watch the presentation video click here

Stock Trading Algorithms From Brute Force to Dynamic Programming

By : Loubna Benchakouk

INTRODUCTION

Analysis of 5 different algorithms for stock trading optimization

Tasks covered:

Problem-1

- Brute Force (Task 1) → $O(m \times n^2)$
- Greedy (Task 2) → $O(m \times n)$
- Single-Transaction DP (Task 3) → $O(m \times n)$

Problem-2

- Multi-Transaction DP (Task 5) → $O(m \times n \times k)$

Problem-3

- Multi-Transaction DP With Cooldown Period (Task 6) → $O(m \times n^2)$

Focus: Algorithm design, performance analysis, and practical trade-offs

Learning objective: Understanding how algorithm choice impacts real-world performance

THE STOCK TRADING PROBLEM

Input: Matrix A[m × n] where m = stocks, n = days

Goal: Maximize profit from buy/sell transactions

Example:

```
A = [
    [7, 1, 5, 3, 6],     # stock 1
    [2, 4, 3, 7, 9],     # stock 2
    [5, 8, 9, 1, 2],     # stock 3
    [9, 3, 14, 8, 7]     # stock 4
]
```

- Stock 4, buy day 2 ($3) → sell day 3 ($14) = $11 profit

Constraints: Must buy before selling, temporal ordering matters

Variations: Single transaction vs. multiple transactions (up to k)

Task 1 - Brute Force Approach: Check Everything

Algorithm

Three nested loops (stocks → buy days → sell days)

Time Complexity

$O(m \times n^2)$

Example

1000 stocks × 1000 days = ~1 billion operations

Advantages

Simple to understand, guaranteed correct answer

Disadvantages

Extremely slow for large datasets

Best Use

Learning, small problems, algorithm verification

```python
def MaxProfitBruteForce(A, m, n):
    """
    Brute force algorithm to find the maximum profit from a single buy/sell trans.

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks (rows)
        n (int): Number of days (columns)

    Returns:
        Tuple[int, int, int, int]: (bestStock, bestBuyDay, bestSellDay, maxProfit)
        All values are 1-based indices.
        Returns (0, 0, 0, 0) if no profit is possible.
    """

    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A or n < 2:
        return (0, 0, 0, 0)

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return (0, 0, 0, 0)

    # --- initialize variables to store the best result found ---
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # --- try every possible stock ---
    for i in range(m):  # stock index (0-based)
        # --- try every possible buy day ---
        for j1 in range(n - 1):  # buy day
            # --- try every possible sell day after the buy day ---
            for j2 in range(j1 + 1, n):  # sell day
                # --- calculate profit for the current transaction ---
                profit = A[i][j2] - A[i][j1]


# --- if this transaction gives higher profit, update the result ---
                if profit > maxProfit:
                    maxProfit = profit
                    # 1-based index
                    bestStock = i + 1
                    bestBuyDay = j1 + 1
                    bestSellDay = j2 + 1

    # --- return result depending on whether any profit was made ---
    if maxProfit == 0:
        return (0, 0, 0, 0)  # if no profitable transaction found
    else:
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

Task 2 - Greedy Optimization

For any sell day, optimal buy day = lowest previous price

Algorithm

Track minimum price while processing days sequentially

Time Complexity

O(m × n) - 1000× improvement!

Example

1000×1000 dataset = ~1 million operations

Advantages

Fast, elegant, mathematically optimal

Best Use

Production systems, single-transaction scenarios

```python
def MaxProfitGreedySolution(A, m, n):
    """
    Greedy algorithm to find the maximum profit from a single buy/sell transaction.

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks
        n (int): Number of days

    Returns:
        Tuple[int, int, int, int]: (bestStock, bestBuyDay, bestSellDay, maxProfit)
        All values use 1-based indexing.
        Returns (0, 0, 0, 0) if no profit is possible.
    """


    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A or n < 2:
        return (0, 0, 0, 0)

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return (0, 0, 0, 0)

    # --- initialize variables to store the best result ---
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # --- iterate over each stock ---
    for i in range(m):
        # --- assume the first day's price is the lowest seen so far ---
        minPrice = A[i][0]      # track minimum price for current stock
        minDay = 0              # track the day of the minimum price

        # --- iterate through the rest of the days ---
        for j in range(1, n):   # start from day 1 (second day)
            profit = A[i][j] - minPrice  # potential profit if sold today

            # --- if this transaction gives better profit, update result ---
            if profit > maxProfit:
                maxProfit = profit
                # 1-based index for stock, buy day, and sell day
                bestStock = i + 1
                bestBuyDay = minDay + 1
                bestSellDay = j + 1

            # --- update minPrice if a new lower price is found ---
            if A[i][j] < minPrice:
                minPrice = A[i][j]
                minDay = j

    # --- return result ---
    if maxProfit == 0:
        return (0, 0, 0, 0)     # if no profitable transaction found
    else:
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

```python
def MaxProfitDynamicProgramming(A, m, n):
    """
    DP approach to find the maximum profit from a single buy/sell transaction.

    Parameters:
        A (List[List[int]]): Matrix representing stock prices (m stocks * n days)
        m (int): Number of stocks
        n (int): Number of days

    Returns:
        Tuple[int, int, int, int]: (bestStock, bestBuyDay, bestSellDay, maxProfit)
        All values use 1-based indexing.
        Returns (0, 0, 0, 0) if no profit is possible.
    """

    # --- edge case: check if the input is empty or missing ---
    if m <= 0 or n <= 1 or not A:
        return (0, 0, 0, 0)

    # --- edge case: check if matrix has proper dimensions
    if len(A) != m or any(len(row) != n for row in A):
        return (0, 0, 0, 0)

    # --- initialize variables to store the best result ---
    maxProfit = 0
    bestStock = 0
    bestBuyDay = 0
    bestSellDay = 0

    # --- loop through each stock ---
    for i in range(m):
        minPrice = A[i][0]  # lowest price seen so far for this stock
        minDay = 0          # day when the lowest price occurred

        # --- check each day starting from day 1 ---
        for j in range(1, n):
            currentProfit = A[i][j] - minPrice  # profit if we sell today

            # --- update result if we found a better profit ---
            if currentProfit > maxProfit:
                maxProfit = currentProfit
                bestStock = i + 1           # 1-based index
                bestBuyDay = minDay + 1
                bestSellDay = j + 1

            # --- update minPrice if today's price is lower ---
            if A[i][j] < minPrice:
                minPrice = A[i][j]
                minDay = j

    # --- return result  ---
    if maxProfit == 0:
        return (0, 0, 0, 0) # if no profitable transaction found
    else:
        return (bestStock, bestBuyDay, bestSellDay, maxProfit)
```

Task 3 - Single Transaction DP

Dynamic Programming Foundation

Approach:

DP principles applied to single-transaction problem

Performance:

$O(m \times n)$ - identical to greedy algorithm

Purpose:

Framework for extensibility, not immediate speed gains

Value:

Foundation for complex multi-transaction scenarios

Comparison:

Same performance as greedy, better structure for extensions

Best Use:

When planning to extend to Tasks 5-6 later

Task 5 - Multiple Transactions DP

Section 1: Algorithm Overview & Input Validation

```
# --- edge case: check if the input is empty or missing ---
if m <= 0 or n <= 1 or not A or k <= 0:
    return []

# --- edge case: check if matrix has proper dimensions
if len(A) != m or any(len(row) != n for row in A):
    return []
```

Section 2: DP Table Initialization

```
    # --- initialize DP tables ---
    # DP table to store the best profit at each day for up to t transactions
    max_profit = []
    for transaction_num in range(k + 1):
        day_profits = []
        for day in range(n):
            day_profits.append(0)  # start with 0 profit for each day
        max_profit.append(day_profits)

    # DP table to remember which (buy, sell) days gave us that profit
    best_days = []
    for transaction_num in range(k + 1):
        day_pairs = []
        for day in range(n):
            day_pairs.append(None)  # no transaction yet
        best_days.append(day_pairs)
```

Section 3: Core DP Algorithm

```
    for t in range(1, k + 1):
        max_diff = -prices[0]        # best value of (max_profit[t-1][d] - prices[d])
        best_buy_day = 1             # start with day 1 (1-based)

        for day in range(1, n):
            current_profit = prices[day] + max_diff

            # ---  option 1: no new transaction today ---
            if max_profit[t][day - 1] >= current_profit:
                max_profit[t][day] = max_profit[t][day - 1]
                best_days[t][day] = best_days[t][day - 1]
            else:
                # ---  option 2: sell today using the best past buy day ---
                max_profit[t][day] = current_profit
                best_days[t][day] = (best_buy_day, day + 1)  # store 1-based

            # --- update max_diff and corresponding buy day ---
            prev = max_profit[t - 1][day] - prices[day]
            if prev > max_diff:
                max_diff = prev
                best_buy_day = day + 1
```

Task 5 - Multiple Transactions DP

Section 4: Transaction Reconstruction

```
# --- reconstruct transactions for this stock ---
t = k
day = n - 1
while t > 0 and day > 0:
    if best_days[t][day]:
        buy_day, sell_day = best_days[t][day]
        profit = prices[sell_day - 1] - prices[buy_day - 1]
        all_transactions.append((stock_index + 1, buy_day, sell_day, profit))
        day = buy_day - 2  # go to the day before the buy_day
        t -= 1
    else:
        day -= 1
```

Test Result

```
Selected Non-Overlapping Transactions:
ID    (Stock, Buy, Sell)    Profit
------------------------------------
T1        (2, 1, 2)          7
T2        (2, 3, 5)          6
T3        (1, 2, 3)          4
------------------------------------
[(2, 1, 2), (2, 3, 5), (1, 2, 3)] with total profit = 17
```

Section 5: Global Selection & Return

```
# --- sort all transactions by profit in descending order  ---
all_transactions.sort(key=lambda x: x[3], reverse=True )


# --- pick top-k non-overlapping transactions ---
selected = []
for stock, buy1, sell1, profit in all_transactions:
    overlaps = False
    for _, buy2, sell2, _ in selected:
        if not (sell1 <= buy2 or buy1 >= sell2):  # overlapping interval
            overlaps = True
            break
    if not overlaps:
        selected.append((stock, buy1, sell1, profit))
        if len(selected) == k:
            break

return selected
```

Task 6 - Multiple Transactions With Cooldown period DP

Problem Overview

Challenge: Cannot buy any stock for c days after selling any stock
Real-World Application: Models trading restrictions, settlement periods, or regulatory constraints
Example: If you sell on day 3 with cooldown=2, you cannot buy until day 6
Algorithm: Dynamic Programming with constraint handling

Section 1: Input Validation

Similar to Task-5

Section 2: Find all profitable transactions

```python
# --- 1: find all profitable transactions ---
# transactions = [(stock_index, buy_day, sell_day, profit, next_day)]
# where next_day = sell_day + cooldown + 1 (1-based index)
transactions = []
for stock_id in range(m):
    prices = A[stock_id]
    for buy_day in range(n - 1):
        for sell_day in range(buy_day + 1, n):
            profit = prices[sell_day] - prices[buy_day]
            if profit > 0:
                buy = buy_day + 1
                sell = sell_day + 1
                next_day = sell + cooldown + 1
                transactions.append((stock_id + 1, buy, sell, profit, next_day))
```

Section 3:Dynamic Programming Optimization

```python
# --- 3: DP table where dp[day] = (max_profit, best_sequence) ---
dp = [(0, []) for _ in range(n + 2)]  # allows access to sell + cooldown + 1
prefix_max = [(0, []) for _ in range(n + 2)]

for t in transactions:
    stock, buy, sell, profit, next_day = t
    next_day = min(next_day, n)  # cap to make sute next_day does not exceed n

    # --- compute the best result before or on the buy day  ---
    best_profit_before = prefix_max[buy]
    total_profit = best_profit_before[0] + profit
    sequence = best_profit_before[1] + [(stock, buy, sell)]

    if total_profit > dp[next_day][0]:
        dp[next_day] = (total_profit, sequence)

    # --- update prefix max at next_day ---
    prefix_max[next_day] = max(prefix_max[next_day], dp[next_day], key=lambda x: x[0])
```

Performance Comparison

Complexity Overview

Task	Problem	Algorithm Type	Time Complexity	Problem Description
Task 1	Problem 1	Brute Force	$O(m \times n^2)$	Single transaction
Task 2	Problem 1	Greedy	$O(m \times n)$	Single transaction
Task 3	Problem 1	Dynamic Programming	$O(m \times n)$	Single transaction
Task 5	Problem 2	Dynamic Programming	$O(m \times n \times k)$	Multiple transactions
Task 6	Problem 3	Dynamic Programming	$O(m \times n^2)$	Cooldown constraints

Performance Analysis (1000×1000 Dataset)

Task	Operations Required	Relative Speed	Best Use Case
Task 2/3	~1 million	1000× faster	Production single-transaction
Task 1	~1 billion	Baseline (slowest)	Learning/verification
Task 6	~1 billion	Similar to Task 1	Cooldown scenarios
Task 5 (k=10)	~10 million	100× faster than brute force	Multiple transactions

Algorithm Selection Guide

Brute Force: Learning, datasets <100×100, algorithm verification

Greedy: Production systems, single transactions, speed priority

Single DP: Planning future extensions to multi-transaction problems

Multi DP: Realistic trading, portfolio optimization, k-transaction scenarios, cooldown period

Decision Factors:

Dataset size, performance requirements, future extensibility needs

Understanding level, maintenance complexity, correctness guarantees

THANK YOU