Algorithm Design and Analysis CW

Group 51A

March 27, 2025

Introduction

1.1 Problem Description

We have a warehouse with a storage capacity of S and a storage cost of C per item per day. For a number of days n we have a set of demands d representing the demands of each day in n. For each day, we have the quantity x_i representing the number of items to order at the beginning of day i based on the demand d_i and the inventory level from the day (i-1), 0 items stored at the warehouse at the end of day 0. Orders have a fixed cost K, regardless of the number of items ordered. Items ordered at the beginning of each day can either cover the demand of the day in full or in part, and if there is a surplus it is stored for the following days.

1.2 Objective

Our objective is to find the optimal daily order quantities, x_i to minimise the total storage and delivery costs for all days i=1 to n.

Part A: Counter Example

The **greedy algorithm** checks whether the demands of the current day have been met, if so, it places no new orders and moves to the next day. Otherwise, it sets the order quantity as the amount needed for the current day and calculates the maximum order size possible that can be stored in the warehouse while also being cheaper than making a separate order on another day. The counter example we devised uses the input instance:

$$\text{N} = 3,\ \text{S} = 5,\ \text{K} = 100,\ \text{C} = 20,$$
$$\text{demand} = [2,3,2]$$

Since we always start day 1 with no inventory, our cost for the day is K. It then checks if the following days demand is less than the storage capacity, $d_{i+1} < \text{S}$. In this instance $d_{i+1} = 3$, which is less than 5 so it proceeds.

Then it checks whether it is cheaper to store the items or make a separate order, $d_{i+1} \times C <$ K. Since $3 \times 20 < 100$, it decides to include it in its order but stops there since storing the items for the following day would make it 5C which is not less than K. On day three it makes another order for the demand of 2 items, making the total K + 3C + K = 260.

When calculated on paper, it is evident that it is cheaper to make an order of 2 items on day 1 and then an order of 5 items on day 2. The total cost of this would make it K + 2C + K = 240 rather than 260. This shows that the **greedy algorithm** doesn't return the sequence of orders with the minimum cost.

Part B: Algorithm Design

B.1 Constraints

For a given day i:

1. If the order quantity x_i is 0, we have no delivery cost for that day.
2. The order quantity x_i plus what is stored from day(i-1) is enough to meet the demand d_i.
3. The number of items stored in the warehouse, at the end of day i is less than or equal to the storage capacity S.

B.2 Observations

From the problem description, objective, and constraints, we observe:

- The optimal cost for day i depends on three things:
  1. The cost of day (i-1).
  2. And whether we place an order, which depends on the number of items stored, i.e., the ending inventory from the previous day (i-1).
  3. In addition to that, we need to consider the storage cost, for one day, for the items that we are storing at the end of day i.
- The number of items that we can store at the end of day i is limited by the storage capacity S, since we can only store between 0 and S items at the end of the day. Given that the number of possible ending inventories is finite, we can simply calculate the optimal cost for each ending inventory for a given day.
- For day n, it is assumed that we should end up with exactly 0 items stored, because after the last day we have no future demands, as we are dealing with a limited sequence of demands.

B.3 Solution Structure

To turn these observations into a formula, we first need to determine the solution structure. Let S(i, j) represent the optimal cost from day 1 to day i, ending with j items stored at the end of day i after meeting its demand, where $0 \leq \text{j} \leq S$.

- Since day 1 starts with 0 inventory, we know that for all j's in S(0, j), only S(0, 0) is valid with cost 0, since on day 0, we cannot have items stored and we cannot order.

B.4 Formulation

The recursion formula below states that the optimal cost starting from day 1 to day i is the optimal cost up until the previous day (i-1) in addition to any storage or order costs. In this formulation, for day i and each ending inventory $j \in [0, S]$ we consider all inventory levels $s \in [0, S]$ from the previous day and add-up the delivery costs, in case a delivery is required given that inventory from day (i-1) is not enough to satisfy the demand for the day and to end with j items stored. Additionally we calculate the cost of storing the desired ending inventory j. The final cost is the minimum cost from day one to the current day i with j items stored after meeting the day's demands. Finally, the base case for the recursion is that on day 0 we have no items stored and the cost is 0 i.e., $S(0, 0) = 0$.

$$S(i, j) = \min \left\{ \ (S(i-1, s) + K \cdot b + C \cdot j) \quad \text{for all } 0 \leq s \leq S \ \right\}$$

Where:

- K is the delivery cost per order.
- C is the storage cost per item per day.
- d_i is the demand on day i.
- j is the ending storage.
- s is the ending storage of the previous day (i-1).
- $b = 1$ if $d_i + j > s$, otherwise $b = 0$.

With:

$$S(0, 0) = 0$$

B.5 Bottom-up algorithm

All the ideas explained so far are put in algorithm **Algorithm 1** below.

As stated in the formula, we need to compute the optimal cost for each ending inventory for each day. This mean we need a $n+1 \times S$ table (n + 1 because we are accounting for day 0). Rows in the table represent days, and columns represent ending inventories. The cell [i][j] in the table stores the minimum cost of ending day i with j items stored after fulfilling its demand.

The algorithm initializes the table with infinity for all cells, except for the cell (0,0) which is initialized with 0 as it represent our base case. The table is then filled iteratively starting from day 1, in the following way: for each day and for each possible ending inventory (limited by the warehouse's capacity), check all previous inventories from the previous day and find the one that leads to the minimum cost today.

The algorithm also stores a table prev_storages, representing the traceback table that will be used to construct solutions. In this table the inventory of the previous day that leads to the optimal cost today is stored.

The algorithm returns the minimum cost and the traceback table.

Algorithm 1 Minimum delivery and order cost over n days

$cost[i][j] \leftarrow \infty$ for all $0 \leq i \leq n, 0 \leq j \leq S$ ▷ //Initialize Cost table
$cost[0][0] \leftarrow 0$ ▷ //No cost on day 0 and we can only have 0 items stored
$prev\_storages[i][j] \leftarrow 0$ for all i, j ▷ //Initialize traceback table
for $i \leftarrow 1$ to n **do**
    for $j \leftarrow 0$ to S **do**
        for $s \leftarrow 0$ to S **do**
            $order\_size \leftarrow \max(0, d_i + j - s)$
            $order\_cost \leftarrow K$ if $order\_size > 0$ else 0
            if $cost[i-1][s] != \infty$ **then** ▷ // to avoid checking the invalid costs in day 0
                $total\_cost \leftarrow cost[i-1][s] + order\_cost + j \cdot C$
                if $total\_cost < cost[i][j]$ **then**
                    $cost[i][j] \leftarrow total\_cost$
                    $prev\_storages[i][j] \leftarrow s$
                end if
            end if
        end for
    end for
end for
return $cost[n][0]$, $prev\_storages$

B.6 Constructing Solutions (Traceback)

Algorithm 2 is used to construct the final solution, i.e, the list of orders. To do so, we trace the table starting from the first cell in the last row in the table. The reason behind this is that, as observed earlier, the number of items stored in the warehouse on the last day must be equal to 0, and as such the storage of the previous day that leads to the optimal cost today must be stored at that cell.

Starting from that position we can determine the order size for day n by simply checking if that optimal inventory from the previous day was enough to satisfy the demand for the current day plus the desired ending inventory, if it was enough then our order size is 0, otherwise it is the the amount we need to satisfy the demand of the day and the desired ending inventory after using all stored items. To get the order size for the day n - 1, we move up the table to the row n - 1 and the column with the number corresponding to the previous inventory we used for day n as it will have the optimal inventory level from day n - 2. We repeat this process until day 1.

Algorithm 2 Traceback algorithm

1: $orders[i] \leftarrow 0$ for all $0 \leq i < n$ ▷ Initialize orders array
2: $j \leftarrow 0$ ▷ Start traceback from last day's optimal inventory (0)
3: **for** $i \leftarrow n$ down to 1 **do**
4: $prev\_inventory \leftarrow prev\_storages[i][j]$ ▷ Retrieve previous inventory level
5: $orders[i-1] \leftarrow \max(0, \text{demands}[i-1] + j - prev\_inventory)$
6: $j \leftarrow prev\_inventory$ ▷ Move to the column representing that storage next
7: **end for**
8: **return** *orders*

Part C: Space and Time Complexity

C.1 Time Complexity

In **Algorithm 1**:

- The first loop runs for n times.
- The second loop runs for S times for each i
- The third loop runs for S times for each pairs of (i, j).
- The content of the third loop takes constant time to complete.

Thus the time complexity for **Algorithm 1** is:

$$O(n \times S \times S) = O(n \times S^2)$$

In **Algorithm 2**:

- The for loop runs for n times.
- The content of the for loop takes constant time to complete.

Thus the time complexity for **Algorithm 2** is:

$$O(n)$$

C.2 Space Complexity

In **Algorithm 1**:

- We are storing two $n \times S$ arrays: costs and prev_storages.
- The content of the third loop is using constant space.

Thus the space complexity for **Algorithm 1** is:

$$O(n \times S) = O(n \times S)$$

In **Algorithm 2**:

- We are storing one array of length n
- The content of the for loop is using constant space.

Thus the space complexity for **Algorithm 2** is:

$$O(n)$$

From the analysis of time and space complexities, **Algorithm 1** uses space and runs in time polynomial in n and S, which is efficient and what was specified the assignment.

Part D: Input Instance

The instance given to our group is:

$$n = 5,\ S = 5,\ K = 4,\ C = 1,$$
$$\text{demand} = [6,2,1,1,6]$$

We implemented Algorithm 1 and Algorithm 2 in Python. We received the following results from our algorithms and entered them into tables.

Table 1: Final Cost table

Day	**S=0**	**S=1**	**S=2**	**S=3**	**S=4**	**S=5**
0	0	inf	inf	inf	inf	inf
1	4	5	6	7	8	9
2	6	8	10	11	12	13
3	8	11	12	13	14	15
4	11	13	14	15	16	17
5	15	16	17	18	19	20

Based on **Table 1** we can find the minimum cost which is located at the nth day where S=0. In this case its **15**.

Table 2: Traceback Table

Day	**S=0**	**S=1**	**S=2**	**S=3**	**S=4**	**S=5**
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	2	3	0	0	0	0
3	1	0	0	0	0	0
4	1	0	0	0	0	0
5	0	0	0	0	0	0

Table 2 allows us to calculate the orders that must be made on each day. As was stated in **Section B.6**, we start the traceback from day 5, S=0. The value of the cell is 0, indicating that we started the day with 0 items stored, and thus we need to order the total demand for the day which is 6 items. Then we move to day 4, S=0 (inventory value used in day 5). The value in this cell indicates we started the day with one item stored which is enough to cover the days demand and since we are in the column S=0 meaning we want to end up with 0 items stored, therefore we don't need to make an order. We repeat this process for days 3,2 and 1 which results in the final order list becoming [**8,0,2,0,6**].