An $O(n \log n)$ Algorithm for Single-Item Capacitated Lot Sizing with a One-Breakpoint All-Units Discount and Non-Increasing Prices

Kleitos Papadopoulos*

October 2025

Abstract

This paper addresses the single-item capacitated lot sizing problem with a 1-breakpoint all-units quantity discount in a monotonic setting where the purchase prices are non-increasing over the planning horizon. For this case, we establish several novel properties of the optimal solution and develop a hybrid dynamic programming approach that maintains a compact representation of the solution space by storing only essential information about the states and using linear equations for intermediate values. Our algorithm runs in $O(n \log n)$ time, where n denotes the number of periods. Our result is an improvement over the previous state-of-the-art algorithm, which has an $O(n^2)$ time complexity.

1 Introduction

In business, devising optimal production or ordering strategies is important for operational efficiency. Companies that aim to meet demand over a time frame while minimizing their expenses will encounter variants of the lot sizing problem. The problem requires optimizing orders, production costs and storage expenses, given the constraints of limited capacity and complex cost structures. There is extensive research in this field which signifies its importance in manufacturing efficiency. A closely related problem is the Gas Station Problem of [6], which generalizes a simple variant of the lot sizing inventory problem to graphs. Some of the relevant prior work is shown in the table below.

In this paper, we tackle the single-item capacitated lot sizing problem with a 1-breakpoint, all-units quantity discount. We focus on a specific monotonic variant of the problem where unit purchase prices follow a non-increasing trend over the planning horizon and there are no setup costs. We analyze key structural properties of the optimal solution under these conditions. These properties are then used to design a hybrid dynamic programming algorithm of $O(n \log n)$ time complexity, where n denotes the number of periods.

*Email: kleitospa@gmail.com

1.1 Problem Analogy

To provide intuition, we present our lot sizing problem using a gas station analogy and terminology where:

- Time periods correspond to refueling stations
- Inventory corresponds to fuel in the tank
- Demand corresponds to the distance between stations
- Item cost corresponds to the fuel price

Throughout this paper, we use both terminologies interchangeably as they are equivalent for our purposes.

1.2 A brief description of the result

A naive dynamic program (Appendix, Algorithm 9) can generate optimal costs for every feasible inventory level at each station. These solutions viewed as *solution sets* $S_b(i)$ (before consuming $d(i, i{+}1)$) and $S(i)$ (after consumption) reveal a certain structure. We formalize these objects and prove structural lemmas (boundedness of a short prefix, monotone legacy-price labels, contiguity of generated points, and single-checkpoint dominance) that let us *compress* whole runs of optimal states into the *segments* below and update them only at a few checkpoints (termini and equal-slope boundaries). We package these checks as *MV thresholds*, and use the lemmas to create our fast procedure.

Rather than computing individual states, our algorithm deals with groups of consecutive states. These groups of states are contained in structures that we call segments where each segment has one explicit state and an equation that can be used to generate its remaining implicit states. The coefficients of these equations as well as the endpoints of the segments can change from station to station. We compute the change of the endpoints only when necessary e.g, when an extension of a segment causes another to become suboptimal. The segments for any set of consecutive states at any station i have a specific structure which allows us to primarily use an augmented binary search tree(BST) and the standard BST operations to efficiently store and compute the optimal states for station $i + 1$.

2 Organization of the Paper

Section 3 formalizes the single-item capacitated lot-sizing model with a one-breakpoint all-units discount, non-increasing prices, and linear holding costs. Section 4 introduces the state/segment representation and other preliminaries used throughout. Section 5 develops the key structural observations and lemmas and introduces the MV-threshold machinery, including the holding-cost shift. Section 6 presents the main algorithm and its analysis: an overview (6.1), the auxiliary procedures (6.2), a correctness proof (6.3), an $O(n \log n)$ time

Reference(s)	**Production cost function**	**Holding and backlogging/subcontracting cost functions**	**Time complexity of the algorithm**
Chung and Lin (1988) [3]	Fixed plus linear, Non-increasing setup and unit production costs, Non-decreasing capacity	Linear holding cost	$O(n^2)$
Federgruen and Lee (1990) [5]	Fixed plus linear, Non-increasing setup and unit production costs, One-breakpoint all-units discount	Linear holding cost	$O(n^3)$
Atamtürk and Hochbaum (2001) [1]	Non-speculative fixed plus linear, Constant capacity	Linear holding cost, Non-speculative fixed plus linear subcontracting cost	$O(n^3)$
Atamtürk and Hochbaum (2001) [1]	Concave, Constant capacity	Linear holding cost, Concave subcontracting cost	$O(n^5)$
Mirmohammadi and Eshghi (2012) [10]	Fixed plus linear, One-breakpoint all-units discount and resale	Linear holding cost	$O(n^4)$
Li et al. (2012) [8]	Fixed plus linear, all-units discount (m stationary breakpoints) and resale	Linear holding cost	$O(n^{m+3})$
Koca et al. (2014) [7]	Piecewise concave with m stationary breakpoints	Concave holding cost, Concave backlogging cost	$O(n^{2m+3})$
Ou (2017) [11]	Piecewise linear with m stationary breakpoints	Concave holding cost	$O(n^{m+2} \log n)$
Malekian et al. (2021) [9]	Fixed plus linear, Non-increasing setup and unit production costs, One-breakpoint all-units discount, Constant capacity	Linear holding cost	$O(n^4)$
Down et al. (2021) [4]	Linear cost, Non-increasing unit production costs, One-breakpoint all-units discount	Linear holding cost	$O(n^2)$
This paper (2025)	Linear cost, Non-increasing unit production costs, One-breakpoint all-units discount, Variable inventory capacity	Linear holding cost	$O(n \log n)$

Table 1: Summary of relevant studies.

bound (6.4), and two practical extensions—handling queries when $B(i) > 2Q$ (6.5) and decision-tracking to recover optimal plans (6.6). Section 7 concludes. The Appendix details the augmented balanced-BST implementation (lazy tags, split/join, and related utilities) and includes the baseline dynamic programming solution for reference.

3 Problem Definition

The problem is about the planning of how to order inventory over a sequence of time periods(stations) $t = 1, 2, \ldots, n$. At each period:

- There is a demand d_t (we also use the notation $d(t, t+1)$ to describe the same thing) that must be fulfilled.
- You can order items (fuel) $x_t \geq 0 (x_t$ is constrained by the current inventory size and the leftover of the previous inventory) at a cost that depends on how many you order:
  - If $x_t < Q$, the unit price is $p_{1,t}$.
  - If $x_t \geq Q$, the discounted unit price $p_{2,t} \leq p_{1,t}$ applies.
- Leftover items can be stored in inventory I_t with a storage capacity $B(t)$, incurring a linear holding cost h_t per unit of inventory held at the end of period t.
- We start with no inventory: $I_0 = 0$.

The objective is to minimize the total cost of fulfilling all demands, given the pricing and inventory constraints.

More formally we can define the problem as follows: Minimize the total procurement cost over n periods, subject to meeting demands d_t, respecting inventory capacities $B(t)$, and using a tiered pricing function $p_t(x_t)$.

$$
\begin{aligned}
\min_{x,I} \quad & \sum_{t=1}^{n} (p_t(x_t) + h_t I_t), \\
\text{s.t.} \quad & I_t = I_{t-1} + x_t - d_t, && t = 1, \ldots, n, \\
& 0 \leq I_t \leq B(t), && t = 1, \ldots, n, \\
& I_0 = 0, \quad x_t \leq B(t) - I_{t-1} + d_t, \; I_t \geq 0.
\end{aligned}
$$

where

$$
p_t(x) = \begin{cases} p_{1,t} x, & x < Q, \\ p_{2,t} x, & x \geq Q, \end{cases}
$$

for all $t = 1, 2.., n$ $p_{1,t} \geq p_{1,t+1} \quad \text{and} \quad p_{2,t} \geq p_{2,t+1}$

4 Preliminary Definitions

Central to our result is the following function:

Definition 1. *For each time period i and item (fuel) amount f, we define:*

$dp(i, f) =$ *the minimum cost for reaching time period i and having f items in the inventory when ready to leave the time period i.*

Definition 2. *A state tuple for a specific station i is a tuple containing two values, a fixed inventory level f and its corresponding minimum cost $dp(i, f)$.*

Definition 3. *A solution set $S(i)$ for a station i is a set of state tuples ordered in increasing order of their remaining fuel. For the sake of simplicity we will refer to these state tuples in a solution set as just states and represent them as $dp(i, x)$ (instead of $(x, dp(i, x))$) for a x in the subsequent sections. We say that a state $dp(i, x)$ for some x is in $S(i)$ when its state tuple is contained in $S(i)$.*

We distinguish between the solution sets before using fuel from i which we denote as $S_b(i)$ and $S(i)$ after possibly using fuel from i. After removing $d(i, i+1)$ from the remaining fuel of each state, the latter is equivalent to $S_b(i + 1)$.

A k-solution set for i is a solution set containing an optimal state tuple (e.g states) for each remaining fuel value between 0 and k. An approximate solution set is an optimal solution set where some states are implicit and are generated by other states as is the case below(see def 8). Instead of $dp(i, f)$ being stored for all f explicitly, only some entries are stored along with enough information to compute the rest. This is done in such a way that the entries that are not stored can be inferred from those that have been stored: If $dp(i, f)$ and $dp(i, g)$ are entries that are stored consecutively, then $dp(i, h)$ for $f < h < g$ can be calculated from $dp(i, f)$ using a linear equation.

A k-approximate solution set is an approximate solution set that contains the states from 0 to k some of which are implicit. Note that in this case the $S(i)$ associated with a k-approximate solution set contains states with up to $k + d(i, i + 1)$ remaining fuel.

Definition 4. *We say a state from $S_b(i)$ generates a state in $S(i)$ when the former is used to create the latter (e.g using one of the equations of definition 7). A state is carried over from $S_b(i)$ to $S(i)$ when it is the same within the two solution sets e.g doesn't change from $S_b(i)$ to $S(i)$. We also say a state $dp(i, f)$ covers a point p (or a set of points) when it is used to generate the optimal state that has remaining fuel p (e.g the state $dp(i, p)$).*

Definition 5. *A state-segment (also referred to simply as a segment) in an implicit solution set is a tuple of single state and the length of the continuous region covered by a specific state. When we say a segment covers a set of points or a region we mean its initial explicit state can be used to infer/generate the optimal states in that region.*

Definition 6. *A station i (located at $d(i - 1, i)$ from its previous station) is reachable by a segment (or a state) in $S_b(i - 1)$ or $S(i - 1)$ when that segment covers a region that includes a point y with $y \geq d(i - 1, i)$.*

Definition 7. *With each state-segment for an entry dp(i, f), we maintain the following associated state information:*

- $p(i, f)$*: the price* $p_{2,i'}$ *at which additional fuel could be bought at station* i' *(where* i' *is index of the station where the segment last received* p_2*). We also use this function for the price of the* p_2 *fuel that a state has received most recently.*
- $r(i, f)$*: the maximum amount of additional fuel that could be bought at* i' *(limited by station capacities)*
- *Besides the above fuel which is available for each segment individually there is also the fuel that is available at each station i for all segments. The* ***current station fuel*** *are the fuel types that can be used in combination with the state, namely* $p_{2,i}$ *and* $p_{1,i}$ *from the station* i*. The* ***optimal fuel*** *for a segment is the cheapest (per unit) of all the available fuels to it.*

Definition 8 (State Equations)**.** *The equations that generate a solution at point* $f + x$ *(e.g generate the* $dp(i, f + x)$*) in a segment where dp(i,f) is the initial state of the segment are:*

$$
\begin{aligned}
Eq(x) &= dp(i, f) + x \cdot p_{1,i} && \text{if } p_{1,i} \text{ is the optimal fuel} \\
Eq(x) &= dp(i, f) + x \cdot p(i, f) && \text{if } p_2 \text{ fuel is optimal} \\
Eq(x) &= dp(i, f) + r(i, f) \cdot p(i, f) + (x - r(i, f)) \cdot p_{1,i} && \text{if } x > r(i, f).
\end{aligned}
$$

The general equation for the optimal fuel for any state $dp(i, w)$ *is denoted as* $Eq_w(x)$*.*

We note that the price of the optimal fuel used in the equation that creates implicit states can change from station to station without the explicit state of the segment necessarily having been changed.

Definition 9. *We also say that:*

- *A segment* S_1 *is* ***better than*** S_2 *at a point p if the solution it gives has a lower value at that point.*
- *A segment* S_1 ***dominates*** *another segment* S_2 *when it is better for the totality of the region previously covered by* S_2*.*
- *When domination occurs, the removal of* S_2 *and the readjustment of* S_1 *is called* S_2*'s* ***replacement****.*

Definition 10. *The terminus of a segment is the right endpoint of the segment when the state is first created or when it is readjusted (e.g it replaces another state) , calculated by finding the last point where the state is optimal and can use* $p(i, f)$*. If a segment whose intial state is* $dp(i, f)$ *covers a region larger than* $r(i, f)$*, then its terminus is the point* $f + r(i, f)$*.*

Note that the right endpoint of a state can change from station to station due to the availability of cheaper p_1 fuel, while the terminus must remain invariant unless the state is removed or its adjacent state is removed.

Definition 11. *We say that a set of segments or states in $S_b(i)$ is bulk updated at a station i when their explicit states are given Q units of $p_{2,i}$ and their implicit states are given more than Q units of $p_{2,i}$ implicitly via the segment equation.*

Any implicit state if made explicit before the next bulk-update will receive its $p_{2,i}$ in the sense that a separate tuple will be created that will store information about the value and the remaining fuel of that specific state . This is done only in special circumstances, such as when a segment is split and one of the new segments requires an initial state.

When we say that a state is individually updated we mean that it is given enough $p_{1,i}$ or $p_{2,i}$ to extend to its adjacent to the right segment because it dominates it.

When we say that a set of states will be individually updated we mean that all the dominated states have been removed by individual updating.

5 Key Properties and Lemmas

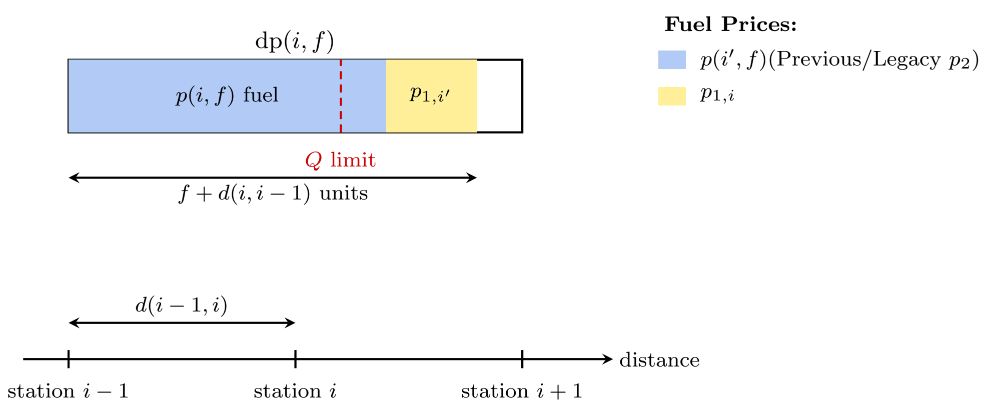

Figure 1: Pictorial representation of a state dp(i, f) in $S_b(i)$. We note that the minimum amount of p_2 fuel that must be used is denoted by the Q-limit. Any "excess" additional amount may be added voluntarily but can lead to a suboptimal solution.

The construction of an optimal solution set $S(i)$ can be done through the use of various algorithms. For the sake of proving the properties and lemmas, we will assume that the construction of $S_b(i)$ for any i itself can be made given the $S_b(i-1)$ using the naive dynamic programming method presented in the appendix as Algorithm 9. In the following proofs the capacity $B(i)$ will be absent due to the sake of simplicity.

Observation 1. *It is never necessary for any state $dp(i, x)$ in $S_b(i)$ that reaches station $i+1$ (e.g. $x \geq d(i, i+1)$) to generate a state in $S(i)$ (and by extension to $S_b(i+1)$) by receiving fuel from station i. The states created in this way are suboptimal(or could have been generated at a subsequent station) in $S(i+1)$.*

Proof. This can be proven by the following simple replacement argument. For any new state created for $S(i+1)$ from $\mathrm{dp}(i, x)$ with $p_{2,i}$ or $p_{1,i}$ there is a state in $S(i+1)$ with the same remaining fuel that can be generated from $\mathrm{dp}(i+1, x-d(i, i+1))$ in $S_b(i+1)$ with cheaper(or equal price) $p_{2,i+1} \leq p_{2,i}$ or $p_{1,i+1} leq p_{1,i}$. □

Note that this holds a fortiori with linear holding costs Observation 1 is crucial to the analysis of cases in the next lemma since it implies that states with more remaining fuel than $d(i, i+1)$ are excluded from the creation of new optimal states with fuel from i.

Lemma 1. *At any given station i, the optimal solution set $S_b(i)$ with the first $2Q$ states contains all the states necessary to construct the optimal solution set $S_b(i+1)$.*

Proof. We prove that any state not included in the 2Q-solution set $S_b(i)$ is never necessary to generate an optimal state in set $S(i)$ and thus in $S_b(i+1)$.

First, recall the structure of a solution set $S_b(i)$: It contains all the optimal states with $f \leq 2Q$.

Consider any optimal state $\mathrm{dp}(i, w)$ with $w > 2Q$ that could potentially exist but is excluded from $S_b(i)$. We show such states are never necessary to generate optimal states in $S(i)$ and consequently in $S_b(i+1)$:

Let's assume that we have a state $\mathrm{dp}(i, f)$ that has remaining fuel $2Q+x$ for some positive x units of fuel in $S(i)$ and can only be created by a state $\mathrm{dp}(i, w)$ that has $w > 2Q$ of remaining units of fuel. Observe that the remaining fuel used in $\mathrm{dp}(i, w)$ must be either a previous (its more recently received) $p_{2,i'}$ from a station $i' < i$ alone or in combination with $p_{1,i-1}$ which are the cheapest p_2 and p_1 fuels available to it up to station i' due to the problem definition (any other combination of fuel can be ruled out by a simple exchange argument). Consider two cases for the above purchase decision of $p_{2,i'}$ at station i':

- **More than Q units are unavoidable ($d(i', i) \geq Q + c$) where c is the remaining fuel of the state that is used to generate the state $\mathrm{dp}(i, w)$:** The fuel w' needed to simply reach station i already exceeds the discount threshold Q. The "excess" fuel, h, is any amount purchased beyond this required minimum threshold. This is the whole inventory w that remains upon arrival at i. By purchasing h fewer units at i', the total units bought are above the Q threshold. If the state created from that with $h = w$ less units of $p_{2,i'}$ is optimal then it is already included in the solution set as $\mathrm{dp}(i, 0)$ and can be used to generate an even cheaper state with f units of remaining fuel (in combination with the current fuel at i), thus contradicting our assumption that $\mathrm{dp}(i, w)$ is necessary to generate $\mathrm{dp}(i, f)$.

- **More than Q units are unnecessary** ($Q + c > d(i', i)$ where (due to observation 1) $c < d(i', i)$ is the remaining fuel of the state that generates $\text{dp}(i, w)$): To receive the discount, we must have chosen to buy more than $2Q$ units even though we could have reached station i with less than $2Q$ and still had the discounted price. The "excess" fuel, h, is the amount purchased on top of the Q-unit threshold. There is an optimal state with $w - h \leq Q$ (because $c < d(i', i)$) remaining fuel (without the extra h fuel from the purchase at i') that is included in $S_b(i)$ since it has less than $2Q$ remaining fuel. This state can be used to construct $\text{dp}(i, f)$ even more cheaply (since $f \geq 2Q$, $w - h < Q$ and $p_{2,i} \leq p_{2,i'}$) thus also contradicting our initial assumption. If $p_{1,i-1}$ is also used by the $\text{dp}(i, w)$ state then the same argument as above can be used to show it's suboptimal.

So the state $\text{dp}(i, w) = \text{dp}(i, w - h) + hp_{2,i}$ with the $p(i', w)$ exchanged by $p_{2,i}$ is cheaper. Furthermore by replacing any quantity h' of $p_{1,i-1}$ with $p_{1,i}$ we also get a cheaper state.

So we conclude that there is a state $\text{dp}(i, w - h' - h)$ in $S_b(i)$ which when used with $p_{2,i}$ and/or $p_{1,i}$ generates a cheaper $\text{dp}(i, f)$ thus contradicting our initial assumption that no state in $S_b(i)$ can be used to create $\text{dp}(i, f)$.

Furthermore since $S_b(i+1)$ is just $S(i)$ with $d(i, i+1)$ being subtracted from all its elements we have proved that no state in $S_b(i)$ with more remaining fuel than $2Q$ is necessary for the construction of $S_b(i + 1)$. □

Lemma 2. *For any period i and any two states $dp(i, f)$ and $dp(i, w)$ in $S_b(i)$ with $f < w$, we have $p(i, f) \geq p(i, w)$ (if the solution sets are explicit then $p(i, \cdot)$ represents the price of the p_2 fuel the corresponding states last received). In other words: states with less fuel have more expensive (or equal) previous p_2 prices.*

Proof. We adopt the tie-breaking convention that, when multiple optimal states exist for the same fuel, we keep the one whose last p_2 purchase is as late as possible, i.e., has the smallest unit price. This preserves optimality and only affects labels $p(i, \cdot)$.

Base case ($i = 1$): All states in $S_b(1)$ have $p(1, f) = +\infty$ (no previous p_2), so the property holds trivially.

Inductive hypothesis: Assume the property holds for $S_b(i)$: for any $f < w$, we have $p(i, f) \geq p(i, w)$.

Inductive step: We show the property holds for $S_b(i+1)$ by analyzing how states in $S(i)$ are created from $S_b(i)$.

By Lemma 1 we may restrict attention to the first $2Q$ states of $S_b(i)$.

1) States carried over (no ordering at period i). Their labels $p(\cdot)$ are unchanged, so the ordering inherited from $S_b(i)$ is preserved.

2) States created with $p_{2,i}$ (ordering $\geq Q$ units at i). Every such state has last discounted price exactly $p_{2,i}$. Consider, for each base state $\text{dp}(i, u) \in S_b(i)$, the linear function

$$\phi_u(x) \;=\; \text{dp}(i, u) + (x - u)\, p_{2,i}, \qquad x \geq u + Q,$$

which is the cost of reaching inventory x by buying at least Q at period i at unit price $p_{2,i}$. All ϕ_u have the same slope $p_{2,i}$, so their lower envelope for large x is attained by the minimum intercept $\min_u\{\mathrm{dp}(i,u) - u\,p_{2,i}\}$, and thus, beyond some inventory threshold, the optimal states in $S(i)$ are exactly those created with $p_{2,i}$ (with label $p_{2,i}$). Hence the set of $p_{2,i}$-created states forms a *suffix* in the inventory order of $S(i)$. Since prices are non-increasing over time, $p_{2,i} \le p(i,\cdot)$ for any carried state, so placing a constant (weakly smaller) label on a suffix preserves the nonincreasing order of labels.

3) States created with $p_{1,i}$ (ordering $< Q$ units at i). Fix a base state $\mathrm{dp}(i,g) \in S_b(i)$ and consider the affine function

$$\psi_g(x) \;=\; \mathrm{dp}(i,g) + (x-g)\,p_{1,i}, \qquad g \le x < g+Q,$$

which gives the cost of reaching x using only $p_{1,i}$. All ψ_g have the same slope $p_{1,i}$. Therefore, if ψ_g is optimal at some e with $g \le e < g+Q$, then for any $e < e' < g+Q$ and any $x < g$,

$$\mathrm{dp}(i,x)+(e'-x)p_{1,i}-\big(\mathrm{dp}(i,g)+(e'-g)p_{1,i}\big) = \Big(\mathrm{dp}(i,x)+(e-x)p_{1,i}\Big)-\Big(\mathrm{dp}(i,g)+(e-g)p_{1,i}\Big),$$

so the sign of the difference is preserved as we move to the right within $[g, g+Q)$. Hence the points generated from $\mathrm{dp}(i,g)$ using $p_{1,i}$ form a *contiguous interval* to the right of g (possibly empty) and they inherit the same label $p(i,g)$ as their base state. Because the base states' labels are nonincreasing by the inductive hypothesis, concatenating these constant-label intervals preserves the nonincreasing order up to the beginning of the $p_{2,i}$ suffix.

Combining 1)–3), the labels in $S(i)$ are nonincreasing in inventory. Passing from $S(i)$ to $S_b(i+1)$ subtracts the constant demand $d(i,i+1)$ from inventories but does not change any labels, so the property holds for $S_b(i+1)$ as well. □

Observation 2. *First, observe that to generate an optimal state with fuel of station i for $S(i)$ we must also use an optimal state in $S_b(i)$. The latter state if its remaining fuel is equal or higher than $d(i,i+1)$ also generates an optimal state in $S(i)$ with no additional fuel (e.g it exists in both $S_b(i)$ and $S(i)$). This means that no state before it in $S_b(i)$ can be used to generate a state after it in $S(i)$(also due to the fuel of the earlier state being more or equally expensive,for this we may need to define the solution sets as always containing the leftmost possible state), this can be proven by an exchange argument.*

Lemma 3. *If a state $dp(i,f) \in S_b(i)$ is used to generate optimal states in $S(i)$ using fuel of a specific price from period i, then these generated states form a continuous interval $[f+a, f+b]$ for some $b \ge a$ and a specific type of used fuel ($p_{1,i}$ or $p_{2,i}$). Furthermore the same applies for any state $dp(i,f) \in S_b(i)$ where the solution set is approximate and the fuel used is a previous p_2 or a combination of that with some p_1.*

Proof. We prove this by contradiction. Suppose $\mathrm{dp}(i,f)$ generates optimal states at two separate continuous sets of points, with a gap in between where the rightmost point x_0 in the gap is generated by another state $\mathrm{dp}(i, w)$. From our previous assumption we know that $\mathrm{dp}(i, f)$ generates the optimal state at $x_0 + 1$. We can also assume that $f < w$ without loss of generality.

Let $C_f(x)$ and $C_w(x)$ denote the costs of generating a state with fuel level x from states $\mathrm{dp}(i, f)$ and $\mathrm{dp}(i, w)$ respectively.

Since $\mathrm{dp}(i, f)$ generates the optimal solution at $x_0 + 1$ but $\mathrm{dp}(i, w)$ at x_0:

$$C_f(x_0 + 1) < C_w(x_0 + 1) \tag{1}$$
$$C_f(x_0) > C_w(x_0) \tag{2}$$

We analyze the following cases based on the price of the fuel used:

Case 1: Both states use $p_{1,i}$ fuel

The cost functions are as follows:

$$C_f(x) = \mathrm{dp}(i, f) + (x - f) \cdot p_{1,i}$$
$$C_w(x) = \mathrm{dp}(i, w) + (x - w) \cdot p_{1,i}$$

From equation (1):

$$\mathrm{dp}(i, f) + (x_0 + 1 - f) \cdot p_{1,i} < \mathrm{dp}(i, w) + (x_0 + 1 - w) \cdot p_{1,i}$$

This implies:

$$\mathrm{dp}(i, f) + (x_0 - f) \cdot p_{1,i} < \mathrm{dp}(i, w) + (x_0 - w) \cdot p_{1,i}$$

Therefore $C_f(x_0) < C_w(x_0)$, contradicting equation (2).

Case 2: Both states use $p_{2,i}$ fuel

This holds if $x_0 \geq \max\{f + Q, w + Q\}$. For $\mathrm{dp}(i, f)$ to generate a state with fuel $x_0 + 1$, it needs $(x_0 + 1 - f)$ units of fuel. Since it must order at least Q units to use p_2 pricing.

The cost functions are:

$$C_f(x) = \mathrm{dp}(i, f) + (x - f) \cdot p_{2,i}$$
$$C_w(x) = \mathrm{dp}(i, w) + (x - w) \cdot p_{2,i}$$

From $C_f(x_0 + 1) < C_w(x_0 + 1)$, we can show (by a similar argument) that $C_f(x_0) < C_w(x_0)$, again contradicting equation. (2).

Case 3 (both states may use previous p_2 then $p_{1,i}$). Both constructions first use their legacy discounted fuel $p(i, \cdot)$ up to what is available, then switch to $p_{1,i}$. Define the switch points

$$x_f^\star := f + r(i, f), \qquad x_w^\star := w + r(i, w),$$

and write the piecewise costs in a compact form using constants

$$A_f := \mathrm{dp}(i, f) + r(i, f)\, p(i, f) - (f + r(i, f))\, p_{1,i}, \quad A_w := \mathrm{dp}(i, w) + r(i, w)\, p(i, w) - (w + r(i, w))\, p_{1,i}.$$

Then, for all feasible x,

$$C_f(x) = \begin{cases} \mathrm{dp}(i,f) + (x-f)\,p(i,f), & x \le x_f^\star, \\ A_f + x\,p_{1,i}, & x \ge x_f^\star, \end{cases} \qquad C_w(x) = \begin{cases} \mathrm{dp}(i,w) + (x-w)\,p(i,w), & x \le x_w^\star, \\ A_w + x\,p_{1,i}, & x \ge x_w^\star. \end{cases}$$

Let $D(x) := C_f(x) - C_w(x)$. By Lemma 2, when $f < w$ we have $p(i,f) \ge p(i,w)$.

Assume, for contradiction, that there is a gap whose rightmost point is x_0, with $\mathrm{dp}(i,w)$ optimal at x_0 and $\mathrm{dp}(i,f)$ optimal at x_0+1; i.e.

$$D(x_0) > 0 \quad \text{and} \quad D(x_0+1) < 0. \tag{3}$$

Since $\mathrm{dp}(i,f)$ is optimal on an interval strictly to the left of x_0, there exists $x^- < x_0$ with $D(x^-) \le 0$. Without loss of generality, assume $x_f^\star \le x_w^\star$. Consider the position of x_0, x_0+1 relative to $x_f^\star, x_w^\star$:

(i) $x_0, x_0+1 \le x_f^\star$. Both can be fulfilled by their $p(i,\cdot)$, so $D'(x) = p(i,f) - p(i,w) \ge 0$ on $(-\infty, x_f^\star]$; hence D is nondecreasing there and $D(x_0+1) \ge D(x_0) > 0$, contradicting (3).

(ii) $x_0 < x_f^\star \le x_0+1 \le x_w^\star$. On left both are using $p(i,\cdot) \Rightarrow D$ nondecreasing; thus $D(x_f^\star) \ge D(x_0) > 0$. On $[x_f^\star, \cdot)$ the slopes are $p_{1,i}$ (left) and $p(i,w)$ (right). If $p_{1,i} \ge p(i,w)$, then D is nondecreasing and $D(x_0+1) \ge D(x_f^\star) > 0$; if $p_{1,i} < p(i,w)$, then D is strictly decreasing for $x \ge x_f^\star$, so from $D(x^-) \le 0$ and monotonicity to the left we get $D(x_f^\star) \le 0$, contradicting $D(x_f^\star) > 0$.

(iii) $x_f^\star \le x_0 < x_0+1 \le x_w^\star$. Here C_f uses $p_{1,i}$ and C_w uses legacy, so D is affine with slope $p_{1,i} - p(i,w)$. If $p_{1,i} \ge p(i,w)$, then $D(x_0+1) \ge D(x_0) > 0$; if $p_{1,i} < p(i,w)$, then D is strictly decreasing for $x \ge x_f^\star$, and as in (ii) the existence of x^- with $D(x^-) \le 0$ forces $D(x_0) \le 0$, contradicting $D(x_0) > 0$.

(iv) $x_0 < x_w^\star \le x_0+1$. On $[x_0, x_w^\star]$ the slopes are $p_{1,i}$ (left) vs. $p(i,w)$ (right), so D is affine with slope $p_{1,i} - p(i,w)$; on $[x_w^\star, \cdot)$ both use $p_{1,i}$ and D is constant. If $p_{1,i} \ge p(i,w)$, then D is nondecreasing on $[x_0, \cdot)$ and $D(x_0+1) \ge D(x_0) > 0$; if $p_{1,i} < p(i,w)$, then as in (iii) we get $D(x_0) \le 0$, contradicting $D(x_0) > 0$.

(v) $x_0, x_0+1 \ge x_w^\star$. Both use $p_{1,i}$, so D is constant and $D(x_0+1) = D(x_0)$, contradicting (3).

In case where $p_{1,i} < p(i,\cdot)$ both states can use $p_{1,i}$ thus the case is reduced to the first.

All possibilities yield a contradiction; therefore no gap can occur when both states use their previous p_2 and then $p_{1,i}$. The points generated by $\mathrm{dp}(i,f)$ in this mixed setting thus form a single continuous interval. □

Also note that there may be discontinuity (at most one gap) between states generated by the same state $\mathrm{dp}(i,f)$ but different fuel type $p_{1,i}$ or $p_{2,i}$.

Implications of Lemma 3: This lemma implies that if a state $dp(i,g)$ is optimal at points f and $p > f$, its segment S_1 must be optimal to all other points between f and p. So, by this locality property, we only need to examine if the adjacent segment to S_1 is dominated by it before checking subsequent states in order to determine the optimal states for any $S(i)$ from $S_b(i)$. If there isn't any state that dominates its adjacent state, that means that no state can dominate another so these states from $S_b(i)$ can be transferred to $S(i)$ unchanged.

Observation 3. *If a state in $S(i)$ doesn't generate any state that covers any points of the next station after $p_{1,i}$ (or its own $p(i,\cdot)$ if $S(i)$ is implicit) is used to update the state, that means its only chance to lead to an optimal solution in $S(i+1)$ is to be used with $p_{2,i}$.*

Lemma 4. *Let S_1 and S_2 be adjacent state-segments in $S(i)$ with explicit states $\mathrm{dp}(i,f)$ and $\mathrm{dp}(i,g)$, where $g > f$. Let t be the terminus of S_2. Assume that at t the state generated from $\mathrm{dp}(i,f)$ using fuel from period i (or using its own legacy p_2 fuel if that is the optimal fuel) is strictly cheaper than the state from S_2:*

$$C_1(t) < C_2(t).$$

If, on the interval of S_2 to the left *of t, the* unit price of the fuel used *by S_1 is everywhere at least the one used by S_2 (i.e., $c_1^{\leftarrow}(x) \geq c_2^{\leftarrow}(x)$ for all such x), then S_1 dominates S_2.*

Conversely, if no other segment is optimal at the terminus t of S_2, then S_2 covers at least the single point t.

Proof. Let $C_1(\cdot)$ and $C_2(\cdot)$ be the costs induced at period i by S_1 and S_2. Each is piecewise-affine; on any subinterval where both use fixed fuel types the slopes are constant and equal to the *unit prices of the fuel being used locally.*

Left of t. Consider the maximal subinterval $I \subset [0,t]$ within S_2's left coverage on which both fuel choices remain fixed; denote their unit prices by $c_1^{\leftarrow}$ and $c_2^{\leftarrow}$. For any $x \in I$,

$$C_1(x) - C_2(x) = \big(C_1(t) - C_2(t)\big) + (c_2^{\leftarrow} - c_1^{\leftarrow})(t-x) \leq C_1(t) - C_2(t) < 0,$$

because $c_1^{\leftarrow} \geq c_2^{\leftarrow}$ by hypothesis. Hence $C_1(x) < C_2(x)$ on I. If there is another kink further left, repeat the same argument on the next subinterval. Thus S_1 is strictly better than S_2 everywhere to the left of t within S_2's coverage.

Right of t. By definition of terminus, either (i) S_2 ends at t, or (ii) to the right it can only use regular fuel $p_{1,i}$ (slope $p_{1,i}$). In case (i) there is nothing to prove. In case (ii), S_1 can also use $p_{1,i}$ (or a cheaper legacy discounted fuel p_2 if available), so its right-hand unit price $c_1^{\rightarrow} \leq p_{1,i}$. For any $x \geq t$ in S_2's right extension,

$$C_1(x) - C_2(x) = \big(C_1(t) - C_2(t)\big) + (c_1^{\rightarrow} - p_{1,i})(x-t) \leq C_1(t) - C_2(t) < 0,$$

so $C_1(x) < C_2(x)$ there as well.

Combining both sides, S_1 is strictly cheaper than S_2 on all points covered by S_2, so S_1 dominates S_2.

Conversely. If no other segment is optimal at t, then S_2 itself attains the minimum at t, hence it covers at least that point; by the segment's piecewise-affine construction it covers a nonempty interval containing t. ☐

Observation:

We note that, a segment S_1 can cover a region that includes points from its next segment without dominating it. In that case it can't cover the second segment's terminus point. Our algorithm doesn't always update the states in partial coverings since it is computationally costly. Since the region in question is always bounded by the two segments we can calculate any point in that region by generating the optimal state by picking the minimum from the generated states at that point using both segments. **Implications of this lemma:** With this lemma, one only needs to check if a state is better at the terminus of another state to know if the second state is dominated.

Lemma 5. *If a state $dp(i, f)$ from segment S_1 with added $w - f$ units of fuel from i generates a cheaper state than $dp(i, w)$ of S_2 at point $w \geq f$ and the optimal fuel of both segments is of the same price per unit throughout the entire interval of S_2 then the first state dominates the second.*

Proof. This is because at any subsequent point $p + w \geq w$ the equations of the two states are

$$Eq_f(w) + C \cdot p \quad \text{and} \quad Eq_w(w) + C \cdot p$$

Given that $Eq_f(w) \leq Eq_w(w)$, then

$$Eq_f(w) + C \cdot p \leq Eq_w(w) + C \cdot p.$$

where C is the fuel's price. □

Implications of Lemma 5 This can be applied when two adjacent state segments have fuel of the same price available to them and then a new cheaper fuel becomes available to both of them so that the first dominates the second. The other case that this can be applied is when the first segment has more expensive fuel than the second and then a new cheaper fuel becomes available to both of them so that the first dominates the second. This isn't the case when the fuel of the second state, is less expensive than the optimal fuel of the first state. In such a case, which happens when one of the two states has received cheaper p_2 fuel in a bulk-update, the terminus must be used.

On Bulk-updates As it will be shown in the next sections all the segments in $S_b(i)$ whose current right endpoint lies below $d(i, i+1)$ are bulk-updated. These segments receive initially Q units of $p_{2,i}$. Some of these segments may overlap with other segments in the range $[Q, 2Q]$. In order to determine which segments are optimal and remove the rest efficiently, we will use a procedure that uses the following lemma.

Lemma 6. *When creating $S(i)$ from $S_b(i)$, let L_1 be the set of state segments whose coverage intersects the band $[Q, 2Q]$ and that were created at earlier stations (so within $[Q, 2Q]$ their unit price is either legacy $p(i, \cdot)$ or $p_{1,i}$), and let L_2 be the set of newly bulk-updated segments at station i (so within $[Q, 2Q]$ they use $p_{2,i}$). Then exactly one of the following holds:*

1. ***Complete takeover.*** *If the lower envelope of L_2 is no higher than the lower envelope of L_1 at some point x in the region covered by $L_1 \cap [Q, 2Q]$, then for all $x' \geq x$ in that region the lower envelope of L_2 is no higher than that of L_1. Consequently, all segments of L_1 to the right of x are dominated by L_2.*

2. ***No takeover inside (tail extension).*** *If the lower envelope of L_2 is strictly above that of L_1 at every point of $L_1 \cap [Q, 2Q]$, then all segments of L_2 are suboptimal throughout L_1's coverage in $[Q, 2Q]$. However, the rightmost L_2 segment may become optimal immediately to the* right *of L_1's band—e.g., on $(2Q,\ 2Q + d(i, i{+}1)]$.*

Proof. By the non-increasing price assumption, within $[Q, 2Q]$ every segment in L_2 uses unit price $p_{2,i}$, while every segment in L_1 uses a unit price $\geq p_{2,i}$ (either a legacy $p(i, \cdot)$ or $p_{1,i}$). Thus, on $[Q, 2Q]$, *all* lines from L_2 have slope $p_{2,i}$, and *every* line contributing to the lower envelope of L_1 has slope $\geq p_{2,i}$.

Case 1 (complete takeover). Suppose there exists x in $L_1 \cap [Q, 2Q]$ where an L_2 segment attains a cost no higher than the L_1 envelope. Since the L_2 envelope has slope $p_{2,i}$ and the L_1 envelope has slope $\geq p_{2,i}$ (piecewise-constant), the difference "L_2 minus L_1" is nonincreasing in x. Hence for all $x' \geq x$ the L_2 envelope remains no higher than the L_1 envelope. This implies all L_1 segments to the right (and the right suffix of the segment containing x) are dominated.

Case 2 (no takeover inside). If for all x in $L_1 \cap [Q, 2Q]$ the L_2 envelope is strictly higher than the L_1 envelope, then L_2 is suboptimal there. Outside that band, L_1 may no longer be defined (e.g., beyond $2Q$); the L_2 envelope can then become optimal as the only available candidate, in particular just to the right of $2Q$, up to $2Q + d(i, i{+}1)$ depending on feasibility. This yields the tail-extension behavior. □

High-level sketch for constructing the Q-approximate $S(i)$ from $S_b(i)$. First, determine the state at zero remaining fuel for $S_b(i{+}1)$ by comparing: (i) the best carry-over state that already reaches $d(i, i{+}1)$ without purchasing at i, and (ii) the cheapest "top-up" among the segments that cannot reach $d(i, i{+}1)$ (adding just enough fuel, possibly $\geq Q$, to reach exactly $d(i, i{+}1)$). Keep the cheaper of the two as $\mathrm{dp}(i{+}1, 0)$.

Next, split the current structure at $d(i, i{+}1)$ into: X = segments with right endpoint $< d(i, i{+}1)$ (cannot reach $i{+}1$) and R = the remaining segments (can reach $i{+}1$). If a segment straddles $d(i, i{+}1)$, cut it at that point.

Bulk-update X by (lazily) adding Q units at price $p_{2,i}$ to each segment, then run individual updates with $p_{2,i}$ inside X to delete dominated segments. Line-merge the resulting bulk-updated segments with the portion of R that covers the $[Q, 2Q]$ band, and then merge X back into the main structure R (applying the relevant lazy updates). Finally, run individual updates with $p_{1,i}$ to remove any remaining dominated segments.

Lemma 7. *The number of segments in an optimal Q-approximate solution set $S_b(i)$ (after processing station $i - 1$) is at most $2i$.*

Proof. We prove by induction on i.

Base ($i = 1$). At the first station there are at most two segments (one using discounted fuel p_2 and one using regular fuel p_1), so $|S_b(1)| \leq 2$.

Inductive step. Assume $|S_b(i-1)| \leq 2(i-1)$. We show $|S_b(i)| \leq 2(i-1)+2$.

Consider the transformation from $S_b(i-1)$ to $S_b(i)$ as in our algorithm (individual updates, split at $d(i-1,i)$, bulk update of the "cannot reach" side, line-merge in $[Q, 2Q]$, merge back, and final pruning). The only events that can *increase* the number of segments are:

(A) *A split at the next-station distance.* At most one segment can straddle $d(i-1,i)$, so cutting there creates *at most one* extra segment.

(B) *Insertion of the zero-fuel state for the next station.* Forming $\mathrm{dp}(i, d(i-1,i))$ (which becomes $\mathrm{dp}(i,0)$ in $S_b(i)$) may introduce *at most one* new segment (and it may be absorbed if it is dominated).

All other operations cannot increase the count:

- *Individual updates (with $p_{1,i}$ or $p_{2,i}$)* and the *line-merge in* $[Q, 2Q]$ only remove dominated segments or extend/replace them. By the "dominance from a terminus test," a strict win at a boundary with a slope (unit-price) no smaller to the left (and $\leq p_{1,i}$ to the right) implies full dominance of the adjacent segment; this eliminates segments and never creates new ones.
- *Bulk update* applies lazily to the "cannot reach" subset; subsequent pruning within that subset again only deletes segments. No new internal boundaries are introduced besides the possible single cut at $d(i-1,i)$ accounted for in (A).

Therefore, from $S_b(i-1)$ to $S_b(i)$ the segment count increases by at most 2 (events (A) and (B)), and may decrease due to pruning. Hence

$$|S_b(i)| \leq |S_b(i-1)| + 2 \leq 2(i-1) + 2 = 2i.$$

This completes the induction. □

5.1 MV thresholds and their use

Before describing the full procedure, we introduce a threshold quantity that lets us apply Lemmas 2 and 3 together with the dominance tests efficiently.

Setup. Let S_1, S_2 be two *adjacent* segments in $S(i)$ with explicit states $\mathrm{dp}(i,f)$ (left) and $\mathrm{dp}(i,g)$ (right), where $g > f$. Let c denote a unit fuel price available at station i (in our algorithm $c \in \{p_{1,i}, p_{2,i}\}$). For a chosen checkpoint, define $MV(S_1)$ as the *largest* value of c for which the state generated from $\mathrm{dp}(i,f)$ is no more expensive than the competing state from S_2. Whenever the actual price at i satisfies $c \leq MV(S_1)$, S_1 wins at that checkpoint; combined with the dominance lemmas, this lets us remove S_2.

Regular MV (boundary checkpoint). This applies when, at the boundary $x = g$, *both* segments would buy at station i with the *same* unit price c (e.g., both use $p_{1,i}$, or both use $p_{2,i}$ after a bulk update). Equating costs at $x = g$ gives

$$\mathrm{dp}(i, f) + (g - f)c \;\le\; \mathrm{dp}(i, g) \quad \Longleftrightarrow \quad c \;\le\; \frac{\mathrm{dp}(i, g) - \mathrm{dp}(i, f)}{g - f}.$$

We set

$$MV(S_1) \;:=\; \frac{\mathrm{dp}(i, g) - \mathrm{dp}(i, f)}{g - f}. \tag{4}$$

Terminus MV (used when S_2 uses $p_{2,i}$ on its right). Let $t = \mathrm{terminus}(S_2) = g + T$ be the last point where S_2 uses its "p_2" fuel on the right of g. In the bulk-updated case of interest here, S_2 uses the *current* discounted price $p_{2,i}$ on $[g, t]$, so

$$C_2(t) = \mathrm{dp}(i, g) + T\, p_{2,i}.$$

From $\mathrm{dp}(i, f)$, the additional units to reach t are $L = (g - f) + T$. Let $a := \min\{L,\, r(i, f)\}$ be the number of those units that S_1 can cover using its *own legacy* discounted price $p(i, f)$; the remaining $(L - a)$ units must be bought at i at unit price $\min\{c,\, p_{1,i}\}$. Two regimes are relevant:

- **If** $c \ge p(i, f)$, S_1 uses its legacy $p(i, f)$ for a units and price c for the remaining $(L - a)$ units:

  $$C_1(t) = \mathrm{dp}(i, f) + a\, p(i, f) + (L - a)\, c,$$

  so

  $$MV_{\mathrm{term}}(S_1) = \frac{\mathrm{dp}(i, g) + T\, p_{2,i} - \mathrm{dp}(i, f) - a\, p(i, f)}{L - a}, \qquad (\text{apply with } c \ge p(i, f)). \tag{5}$$

  When checking against $c = p_{2,i}$, the purchase at i in this regime is $(L - a)$ units, so the discount $p_{2,i}$ is actually *available* only if $(L - a) \ge Q$; otherwise compare using $c = p_{1,i}$.

- **If** $c \le p(i, f)$, S_1 uses c for *all* L units (no legacy is used):

  $$C_1(t) = \mathrm{dp}(i, f) + L\, c, \quad \Longrightarrow \quad MV_{\mathrm{term}}(S_1) = \frac{\mathrm{dp}(i, g) + T\, p_{2,i} - \mathrm{dp}(i, f)}{L},$$

  and, when checking against $c = p_{2,i}$, the discount requires $L \ge Q$ (here all L units are bought at i); if $L < Q$, compare with $c = p_{1,i}$ instead.

If S_2 uses its *own legacy* p_2 on the right. In this case S_2's unit price on $[g, t]$ is $p(i, g)$, and we only need to check the *last* point where that legacy fuel is used, i.e., the same terminus t (not S_2's rightmost coverage). The formulas above apply verbatim with $p_{2,i}$ replaced by $p(i, g)$ in $C_2(t)$.

Which MV to use. Use the *Regular MV* (4) at the boundary $x = g$ whenever both segments would buy at i with the same unit price (equal slopes). Use the *Terminus MV* when the right segment S_2 is using $p_{2,i}$ on its right (bulk-updated case); if S_2 is using its own legacy p_2, evaluate the terminus comparison at the last legacy point t as noted above.

How we store and use MV. For each adjacent pair (S_1, S_2) we store either $MV_{\text{reg}}(S_1)$ or the appropriate $MV_{\text{term}}(S_1)$ and maintain these keys in decreasing order. At station i, comparing the actual price $c \in \{p_{1,i}, p_{2,i}\}$ against the stored thresholds (and the quantity conditions $L \geq Q$ or $L - a \geq Q$ when $c = p_{2,i}$) allows us to perform individual updates: whenever $c \leq MV(S_1)$, S_1 wins at the checkpoint and S_2 is removed with neighbors re-adjusted.

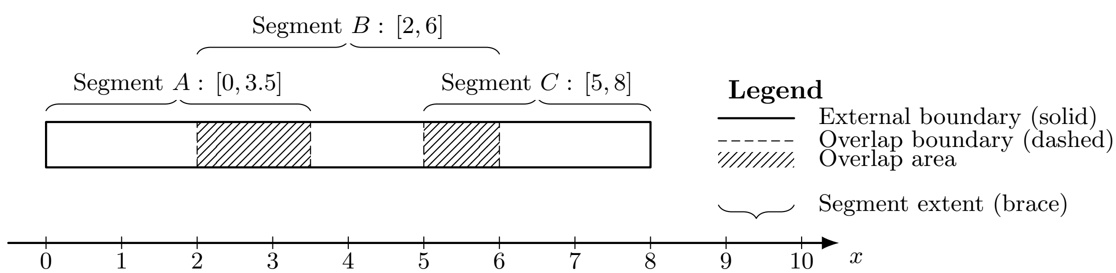

Figure 2: The figure represents a number of segments on an X-axis showing their overlapping regions. Notice that the terminus point of a segment is always in the non-overlapping part of a segment while the current endpoints of the segment in the overlap area may be unknown

5.2 Extending the Result to Accommodate Linear Holding Costs

When holding costs are incorporated into the model, they affect the solution in three ways. First, the value of each state at station i increases by $h(t)$ times its remaining fuel amount, reflecting the cost of carrying inventory to the next station. Second, the effective price of $p_{2,i}$ fuel increases by $h(t)$ per unit. Third, as demonstrated in the following lemma, all MV values increase uniformly by the constant $h(t)$, which can be efficiently implemented as a lazy update operation on the binary search tree maintaining these values.

Lemma 8 (Holding-cost for MV values at station i)**.** *Fix station i and a linear holding cost h_i per unit of inventory at the end of period i. Implement the holding-cost update at station i by*

$$dp(i, x) \mapsto dp(i, x) + h_i x \quad \text{for all states } x, \qquad \text{and} \qquad p_{1,i},\, p_{2,i} \mapsto p_{1,i} + h_i,\ p_{2,i} + h_i.$$

Then, for every adjacent pair of segments (S_1, S_2) in $S(i)$ with explicit states $dp(i, f)$ and $dp(i, g)$, $g > f$,

1. *the* Regular MV *(boundary checkpoint $x = g$) satisfies*

$$MV'(S_1) \;=\; MV(S_1) \;+\; h_i,$$

2. *and any* Terminus MV *computed at a fixed checkpoint x_0 (e.g., $x_0 =$* $\text{terminus}(S_2)$*) also satisfies*

$$MV'_{\text{term}}(S_1) \;=\; MV_{\text{term}}(S_1) \;+\; h_i.$$

Consequently, the ordering of all MV values is preserved by the holding-cost update.

Proof. Regular MV. For adjacent states $\text{dp}(i, f)$ and $\text{dp}(i, g)$ with $g > f$, the boundary threshold is

$$MV(S_1) \;=\; \frac{\text{dp}(i,g) - \text{dp}(i,f)}{g - f}.$$

After the update,

$$MV'(S_1) = \frac{\big(\text{dp}(i,g) + h_i g\big) - \big(\text{dp}(i,f) + h_i f\big)}{g - f} = MV(S_1) + h_i.$$

Terminus MV. Any terminus-based comparison is performed at a *fixed checkpoint* x_0 (e.g., $x_0 = \text{terminus}(S_2)$), and yields an affine inequality of the form

$$C_1(x_0; c) \;\le\; C_2(x_0),$$

where c is the unit price used for period-i purchases by S_1 (so $c \in \{p_{1,i}, p_{2,i}\}$ in our algorithm). Under the update, both state values acquire the same additive term $h_i x_0$ and the period-i unit prices shift by $+h_i$, i.e., $c \mapsto c + h_i$. Therefore the new inequality reads

$$C_1(x_0; c + h_i) + h_i x_0 \;\le\; C_2(x_0) + h_i x_0,$$

which is equivalent to $C_1(x_0; c + h_i) \le C_2(x_0)$. Thus the new threshold is obtained by adding h_i to the old one, i.e., $MV'_{\text{term}}(S_1) = MV_{\text{term}}(S_1) + h_i$. □

6 The Algorithm

6.1 General Overview

Instead of computing individual states, our algorithm computes *segments* that compactly represent contiguous ranges of optimal states together with the information needed to regenerate any state they cover. These segments are maintained in an augmented balanced binary search tree (BST), keyed by their terminus, and augmented with the appropriate MV thresholds (Regular or Terminus) and lazy tags.

At station i, we construct an approximate version of $S(i)$ from $S_b(i)$ using two kinds of updates:

- **Individual updates (with $p_{1,i}$ or $p_{2,i}$):** we repeatedly compare the current price $c \in \{p_{1,i}, p_{2,i}\}$ to the *maximum* applicable MV threshold among adjacent segment pairs (Regular MV when both sides would use the same unit price at the boundary, Terminus MV when the right segment uses discounted fuel on its right). Whenever $c \le MV$, the left segment dominates the right one at the checkpoint; we remove the dominated segment and re-adjust only its neighbors in $O(\log n)$ time.
- **Bulk updates (with Q units at $p_{2,i}$):** all segments whose current right endpoint is strictly less than $d(i, i+1)$ (i.e., cannot reach station $i+1$) are gathered, and a lazy "$+Q$ at $p_{2,i}$" tag is applied to this subset. If a segment *straddles* $d(i, i+1)$ we cut it at that point (creating at most one additional segment).

The per-station flow is:

1. **Prune with $p_{1,i}$:** run individual updates comparing $p_{1,i}$ to the stored MV thresholds to remove any immediately dominated neighbors.
2. **Form $\mathbf{dp(i{+}1, 0)}$:** compare (i) the best carry-over that already reaches $d(i, i+1)$ (no purchase at i) with (ii) the cheapest "top-up" among the non-reaching segments that raises the state to exactly $d(i, i+1)$ (if the purchase at i is $< Q$, this top-up is at $p_{1,i}$; otherwise at $p_{2,i}$). Keep the cheaper as $\mathrm{dp}(i+1, 0)$.
3. **Split at $\boldsymbol{d(i, i+1)}$:** partition the BST into $X = \{$segments with right end $< d(i, i+1)\}$ and $R = \{$the rest$\}$; if a segment straddles $d(i, i+1)$, cut it.
4. **Bulk-update X at $\mathbf{p_{2,i}}$:** lazily add Q units to each segment in X (so each can qualify for $p_{2,i}$), then run individual updates *inside* X with $p_{2,i}$ to remove dominated segments. When both adjacent segments are bulk-updated, we convert any stored Terminus MV to the Regular MV at their boundary (both sides now use the same unit price at g).
5. **Line-merge in $\boldsymbol{[Q, 2Q]}$:** merge the bulk-updated segments from X with the portion of R that covers the $[Q, 2Q]$ band, using the line-merge lemma to discard suboptimal pieces (complete takeover or tail extension).
6. **Merge back and final prune:** join the structures, recompute the cross boundary MV where needed (Regular vs Terminus, per the fuel available on each side), and run a final individual-update pass with $p_{1,i}$ to eliminate any remaining dominated neighbors.

The BST stores, for each segment, just enough information to recreate any state within its coverage. Most operations are standard BST primitives (search, split, join, local re-balance) plus lazy tags for value/position shifts and price updates. Critically, states *between* two consecutive segments may remain undetermined; if a concrete value is needed at some x between their termini, we evaluate the two bounding segments at x in $O(1)$ and take the minimum after an $O(\log n)$ search, yielding overall $O(\log n)$ query time.

Algorithm 1: Per-station construction of $S(i)$ from $S_b(i)$

Input: BST T storing segments of $S_b(i)$ keyed by terminus; prices $p_{1,i}, p_{2,i}$; threshold Q; distance $d(i, i{+}1)$.

Output: T updated to a $2Q$-approximate $S(i)$; state $\mathrm{dp}(i{+}1, 0)$.

for $i \leftarrow 1$ **to** n **do**

  `// (a) prune with` $p_{1,i}$

  IndividualUpdate$(T, 1)$

  `// (b) best cost to reach` $d(i, i{+}1)$ `(carry-over vs top-up)`

  $\mathsf{cand}_{\mathrm{tmp}} \leftarrow$

$$\min\left\{\text{carry-over to } d(i,i{+}1),\ \text{top-up to } d(i,i{+}1) \text{ using } \begin{cases} p_{1,i}, & \text{if added } < Q, \\ p_{2,i}, & \text{if added } \geq Q \end{cases}\right\}$$

  `// (c) split at` $d(i, i{+}1)$

  CutSegmentAtBoundaryAndPutTheResultInT$(T, d(i, i+1)$; $(X, R) \leftarrow$ SplitAt$\big(T, d(i, i{+}1)\big)$;

  `// (d) lazy bulk update in` X `with` Q `at` $p_{2,i}$

  BulkUpdateQAtP2$(X, Q, p_{2,i})$

  `// (e) Compute the cheapest cost among segments in` X `whose terminus` $< d(i, i{+}1)$ `after adding at station` i `the extra units priced at` $p_{2,i}$.

  `// The` Q`-block having been enabled in step (d) to reach` $d(i, i{+}1)$`; remove those segments from` X.

  $\mathsf{cand}_X \leftarrow$ cheapest among segments in X whose terminus $< d(i, i{+}1)$ after adding just enough $p_{2,i}$ to the segment(its endpoints) so the initial state of it to reach $d(i, i{+}1)$; remove those segments from X

  `// (f) finalize` $\mathrm{dp}(i{+}1, 0)$ `with integrated tie-handling ===comment the following is inadequate to update the structure for the dp(i+1,0), tree insertion operations are needed`

  if $\mathsf{cand}_{\mathrm{tmp}} < \mathsf{cand}_X$ **then**

    $\mathrm{dp}(i{+}1, 0) \leftarrow \mathsf{cand}_{\mathrm{tmp}}$

  else if $\mathsf{cand}_X < \mathsf{cand}_{\mathrm{tmp}}$ **then**

    $\mathrm{dp}(i{+}1, 0) \leftarrow \mathsf{cand}_X$

  else

    `// tie: prefer the one whose plan has cheaper legacy` p_2`; if still tied, use a fixed canonical rule`

    $\mathrm{dp}(i{+}1, 0) \leftarrow$ TieBreakByLegacyP2$(\mathsf{cand}_{\mathrm{tmp}}, \mathsf{cand}_X)$

  InsertOrReplaceAtZero$(T, \mathrm{dp}(i{+}1, 0))$

  `// (g) prune` X `with` $p_{2,i}$

  IndividualUpdate$(X, 2)$

  `// (h) line-merge in` $[Q, 2Q]$ `and reassemble`

  $(R', X') \leftarrow$ LineMergeBand$\big(R|_{[Q,2Q]}, X\big)$; $T \leftarrow$ Join(R', X')

  `// (i) cross-boundary dominance (keep Terminus MV at the join)`

  while CrossBoundaryDominates(T) **do**

    RemoveDominatedAndRecomputeBoundaryAndMV(T)

  `// (j) final prune with` $p_{1,i}$

  IndividualUpdate$(T, 1)$

  `// (k) Remove any state above` $B(i)$ `and cutting the state that covers the point at` $B(i)$

  Trim$(T, B(i))$

  AddHoldingCostsAndSubtractDemand$(T, h(i), d(i))$

Algorithm 2: IndividualUpdate(T, j)

Input: BST T; comparator index $j \in \{1, 2\}$ with price $p_{j,i}$.
Output: T with dominated adjacent segments removed under price $p_{j,i}$.

while $\exists$ *eligible MV in T for comparator j with value* $\geq p_{j,i}$ **do**
    $(S_1, S_2, g) \leftarrow$ ArgMaxEligibleMV(T, j) `// adjacent pair at boundary` g
    if IsTerminusMV(S_1, S_2) ***and*** BoundaryPricesEqualAt(g) **then**
      RecomputeAsRegularMV(S_1, S_2, g); **continue**
    RemoveDominatedRightNeighborAndRe-adjustLeft(S_1, S_2)
    RecomputeLocalMVs(left neighbor of S_1, S_1) and
      (S_1, new right neighbor of S_1)

Tie rule: if $p_{j,i} = MV$ exactly, prefer the left segment at g (canonical choice).

Algorithm 3: $\textsc{LineMergeBand}(R|_{[Q,2Q]}, X)$

Input: $R|_{[Q,2Q]}$: segments from $S_b(i)$ restricted to the band $[Q, 2Q]$ (ordered by start); X: bulk-updated segments (ordered by start).

Output: (R', X'): updated fragments after merging in $[Q, 2Q]$.

Function `RightmostCheaperPoint`$(l_R, l_X, [a, b])$**:**

  // $[a, b]$ is the (nonempty) overlap of the two segments, clipped to $[Q, 2Q]$

  Compute affine forms in $[a, b]$: $C_R(x) = \alpha_R + \beta_R x$, $C_X(x) = \alpha_X + \beta_X x$ using the local unit prices each segment actually uses in this subinterval.

  if $\beta_R = \beta_X$ **then**

    if $C_R(b) < C_X(b)$ **then**

      return b

    else

      return None

  else

    $x^\star \leftarrow (\alpha_X - \alpha_R)/(\beta_R - \beta_X)$

    if $a \le x^\star \le b$ ***and*** $C_R(x^\star) \le C_X(x^\star)$ **then**

      return $x^\star$

    else

      return None

$L_1 \leftarrow R|_{[Q,2Q]}$; $L_2 \leftarrow X$; $R' \leftarrow L_1$; $X' \leftarrow L_2$;

$l_1 \leftarrow$ leftmost seg. in R'; $l_2 \leftarrow$ leftmost seg. in X'.

while $l_1 \ne$ null ***and*** $l_2 \ne$ null **do**

  $I \leftarrow \text{overlap}\big(l_1.\text{range} \cap l_2.\text{range} \cap [Q, 2Q]\big)$

  if $I = \emptyset$ **then**

    // Advance the one that ends first in $[Q, 2Q]$

    if $l_1.end \le l_2.end$ **then**

      $l_1 \leftarrow$ next in R'

    else

      $l_2 \leftarrow$ next in X'

  else

    $(a, b) \leftarrow I$

    $x \leftarrow$ `RightmostCheaperPoint`$(l_1, l_2, [a, b])$

    if $x \ne$ None **then**

      // Remove the dominated prefix of l_2 up to x (inclusive); adjust or drop l_2

      if $x = b$ ***and*** $b = l_2.end$ **then**

        delete l_2 from X'; $l_2 \leftarrow$ next in X'

      else

        // shrink l_2 so it starts just right of x

        $l_2.\text{start} \leftarrow x$

        // if $x = b$ and $b = l_1$.end, advance l_1

        if $x = b$ ***and*** $b = l_1.end$ **then**

          $l_1 \leftarrow$ next in R'

    else

      // No point in I where l_1 beats l_2 $\Rightarrow$ complete domination or tail extension

      while $l_1 \ne$ null **do**

        delete l_1 from R'; $l_1 \leftarrow$ next in R'

      break

return (R', X')

6.2 Auxiliary procedures used by Algorithm 1

We summarize the auxiliary procedures used by Algorithm 1. Each procedure operates on the height-balanced BST of segments augmented with MV keys (Regular or Terminus) and standard lazy tags; asymptotic costs assume $O(\log n)$ BST edits and $O(\log n)$ priority updates.

SplitAt(T, x)**:** Split the segment tree at the absolute boundary $x = d(i, i{+}1)$ into X (right endpoint $< x$) and R (the rest). Time: $O(\log n)$.

CutSegmentAtBoundaryAndPutTheResultInT$(T, d(i, i{+}1))$**:** If some segment spans $x = d(i, i{+}1)$, cut it there and recompute the two new boundary MVs. Time: $O(\log n)$.

BulkUpdateQAtP2$(X, Q, p_{2,i})$**:** Lazily enable a Q-block at station i priced at $p_{2,i}$ for all segments in X. Time: $O(1)$ (lazy tag).

IndividualUpdate(T, j)**:** Repeatedly remove dominated right neighbors under the comparator price $p_{j,i}$ ($j \in \{1, 2\}$), using the maximum eligible MV at each step; converts Terminus$\to$Regular at equal boundary prices. Time: $O(\log n)$ per removal; $O(n \log n)$ total over the run.

InsertOrReplaceAtZero$(R, \mathrm{dp}(i{+}1, 0))$**:** Insert (or replace) the boundary state/segment at $x = d(i, i{+}1)$ in the *reaching* tree R, with canonical tie rule (cheaper legacy p_2, then fixed key). Time: $O(\log n)$.

LineMergeBand$(R|_{[Q,2Q]}, X)$**:** Merge bulk-updated X against the slice of R covering $[Q, 2Q]$ using the line-merge lemma (complete domination or tail extension); returns trimmed fragments to rejoin. Time: $O(\#\mathrm{edits} \cdot \log n)$, with $\#\mathrm{edits} = O(n)$ over the run.

CrossBoundaryDominates(T)**:** At the unique R–X join, test the correct MV (Terminus or Regular) against the appropriate comparator price; return true if the right neighbor is dominated. Time: $O(1)$.

RemoveDominatedAndRecomputeBoundaryMV(T)**:** Remove the dominated neighbor at the join; recompute the new cross-boundary MV (Regular if boundary prices are equal, else Terminus). Time: $O(\log n)$.

Join(R', X')**:** Join two trees with disjoint key ranges; recompute the unique cross-boundary MV created by the join. Time: $O(\log n)$.

Trim$(T, B(i))$**:** Enforce capacity by removing infeasible tails and cutting at $B(i)$. Time: $O(\log n)$.

AddHoldingCosts$(T, h(i))$**:** Apply end-of-period linear holding costs by a lazy value shift $+h(i) \cdot x$ and a uniform MV-key shift $+h(i)$ (holding-cost MV-shift lemma). Time: $O(1)$ (lazy).

6.3 Correctness of the Algorithm

We prove that Algorithm 1 produces, at every station i, a correct $2Q$-approximate solution set $S(i)$ and the correct boundary state $\mathrm{dp}(i{+}1, 0)$. The proof proceeds by induction over stations and is structured around an invariant that is preserved by each step of the algorithm.

Invariant ($\mathcal{I}_i$) (coverage, optimality, and consistency). After completing the loop for station i, the BST T encodes a $2Q$-approximate solution set $S(i)$ with the following properties:

(I1) **Coverage and optimality on $[0, 2Q]$.** For every inventory level $x \in [0, 2Q]$, the value represented implicitly or explicitly by T equals the true optimum $\mathrm{dp}(i, x)$. Each segment stores one explicit state and a valid linear equation to generate all covered states, and (by the continuity lemma in Section 5) every point covered by a segment is generated by its anchor without gaps for the chosen fuel type.In the segment overlapping sections the optimal value of a state is calculated by taking the appropriate min of the states generated by the two bounding segments.

(I2) **No removable domination at the maintained price.** No adjacent pair in T violates the dominance tests under $p_{1,i}$ (Regular or Terminus, as appropriate), i.e., the left segment is not strictly better at the boundary checkpoint when evaluated with the correct unit price; equivalently, all eligible MVs are $< p_{1,i}$ after the final prune.

(I3) **Correct MV type and value at every boundary.** If both sides buy at the same unit price at the boundary, the stored MV is Regular; otherwise it is Terminus and computed at the right segment's terminus checkpoint, per Section 5.1.

Base case ($i = 1$). At station 1 there is no legacy p_2; the algorithm constructs the reachable states using $p_{1,1}$ and (when applicable) $p_{2,1}$, applies IndividualUpdate$(T, 1)$, and forms the boundary state. The continuity lemma guarantees that each segment covers a single interval; the dominance and MV rules ensure (I2)–(I3). Thus ($\mathcal{I}_1$) holds.

Inductive step. Assume ($\mathcal{I}_i$) holds. We show that, after executing steps (a)–(k) of Algorithm 1, ($\mathcal{I}_i$) still holds for the constructed $S(i)$ and the boundary state $\mathrm{dp}(i{+}1, 0)$ is optimal. This yields ($\mathcal{I}_{i+1}$) in the next iteration once the demand $d(i, i{+}1)$ is subtracted (which is exactly what the boundary split/merge achieves).

Step (a): IndividualUpdate$(T, 1)$. Each iteration compares $p_{1,i}$ to the maximum eligible MV. If the MV is Terminus but both sides now buy at the same unit price at the boundary, the MV is converted to Regular (no structural change). Otherwise, when $p_{1,i}$ is not larger than the MV, the left segment is strictly cheaper at the checkpoint, and by the dominance-from-terminus test (Section 5) it dominates the right segment on its whole coverage; removing the dominated segment preserves (I1). The loop ends exactly when all eligible MVs are $< p_{1,i}$, establishing (I2).

Step (b): forming the boundary candidate $\mathsf{cand}_{\mathrm{tmp}}$. By the generalized Observation 1, a state with remaining fuel $x > d(i, i+1)$ must not buy at station i, because the same or cheaper fuel is available at station $i+1$ in the non-increasing price regime; hence the only optimal ways to be exactly at $d(i, i+1)$ are: (i) carry over a state that already reaches the boundary, or (ii) top up a non-reaching state at i. By the all-units discount, the top-up is either $< Q$ (priced at $p_{1,i}$) or $\geq Q$ (discounted, priced at $p_{2,i}$). Taking the minimum of these two possibilities therefore yields the optimal boundary candidate among the reaching side R, proving that $\mathsf{cand}_{\mathrm{tmp}}$ is correct.

Step (c): split at $d(i, i+1)$ and cut straddlers. These are structural edits that do not change any state value. The explicit boundary guarantees that subsequent tests compare exactly the intended checkpoints (either $x = g$ for Regular MVs or $x = t$ for Terminus MVs).

Step (d): bulk-update X at $p_{2,i}$. Only segments with right endpoint $< d(i, i+1)$ (i.e., cannot reach $i+1$) are allowed to buy at station i, by Observation 1. Bulk-enabling a Q-block at $p_{2,i}$ for all of them is logically equivalent to exposing the cheapest available station-i unit price they can obtain when purchasing $\geq Q$. No state in R is altered at this step.

Step (e): still-below-boundary top-ups in X. Among segments in X whose terminus remains $< d(i, i+1)$ even after (d), the cheapest way to reach the boundary is to add the exact shortfall priced at $p_{2,i}$ (the Q-block having been enabled). Keeping the best of these candidates and removing the segments used to form them is safe: those segments cannot generate any state in $S(i)$ other than their contribution to the boundary state (again Observation 1), and the best such contribution is preserved as cand_X.

Step (f): finalize $\mathrm{dp}(i+1, 0)$ and insert into R. By the argument in steps (b) and (e), the optimal boundary state is the minimum of $\mathsf{cand}_{\mathrm{tmp}}$ and cand_X; the tie rule (prefer the one with the cheaper p_2, then a fixed canonical rule) preserves optimality and ensures determinism. Inserting this single boundary point into R maintains (I1) on the reaching side.

Step (g): IndividualUpdate$(X, 2)$. Now that purchases in X can be made at $p_{2,i}$, the same dominance logic as in step (a) applies with comparator $p_{2,i}$. Using the Regular or Terminus MV as appropriate, any right neighbor strictly dominated at the tested price is removed; by the dominance lemma, no optimal state is lost and (I1) is preserved inside X.

Step (h): line-merge in $[Q, 2Q]$. By the "bulk vs. pre-existing p_2" lemma (Section 5), merging the bulk-updated segments against R's $[Q, 2Q]$ slice yields exactly two patterns: complete takeover (once X is better at some point in the band, it remains better to the right because it uses a weakly cheaper unit price), or tail extension (if X never wins inside the band, it can only become optimal to the right of $2Q$). The line-merge performs exactly these trims, preserving (I1).

Step (i): cross-boundary dominance at the R–X join. At the unique join, the correct MV type is used: Terminus if the right segment still uses legacy p_2 up to its terminus, else Regular at the boundary. If the comparator price meets/exceeds the threshold, the right neighbor is dominated by the left and can be removed (dominance lemma). The loop stops exactly when the new boundary is non-dominated, establishing (I2) at the join.

Step (j): final IndividualUpdate$(T, 1)$. This is identical to step (a) on the reassembled structure, ensuring that no removable domination under $p_{1,i}$ remains anywhere in T (I2), while (I1) is preserved by the dominance lemma.

Capacity and holding costs. TRIM removes only infeasible states (those exceeding $B(i)$) and cuts a straddler at $B(i)$; feasibility is preserved and (I1) remains true on the allowed domain. ADDHOLDINGCOSTSANDSUBTRACTDEMAND adds $h(i) \cdot x$ lazily to every state and subtracts the appropriate demand. By the holding-cost MV-shift lemma (Section 5), every MV threshold increases by the same constant $h(i)$, so the ordering of thresholds and all dominance decisions are unchanged; hence (I2)–(I3) remain true.

Conclusion of the inductive step. Combining the above, every step either (i) performs a purely structural edit, (ii) removes a segment that is dominated everywhere on its coverage (hence cannot be part of an optimal solution), or (iii) inserts the unique optimal boundary state $\mathrm{dp}(i+1, 0)$. Therefore $(\mathcal{I}_i)$ holds for the constructed $S(i)$, and the boundary state is optimal. As $S_b(i+1)$ is obtained from $S(i)$ by subtracting $d(i, i+1)$ from remaining fuel, the same arguments yield $(\mathcal{I}_{i+1})$.

Theorem 1. *Assuming non-increasing unit prices over time, a single all-units discount threshold Q, and linear holding costs, Algorithm 1 correctly computes, for every station i, a $2Q$-approximate solution set $S(i)$ and the optimal boundary state $\mathrm{dp}(i+1, 0)$. Equivalently, for every $x \in [0, 2Q]$, the value represented in T equals $\mathrm{dp}(i, x)$, and the state $\mathrm{dp}(i+1, 0)$ is optimal.*

6.4 Time Complexity Analysis

Model of computation. We store segments in a height-balanced BST keyed by their terminus. Standard operations (search, insert, delete, split, join) take $O(\log n)$ time. The data structure maintains at most $O(n)$ MV keys (Regular or Terminus) ; reading or updating takes $O(\log n)$ time. Bulk and holding-cost updates are implemented lazily (constant-time tags applied at subtree roots or global MV-key shifts).

Global structural bound. Let m_i be the number of segments in $S(i)$. By the segment-count lemma (Section 5), at each station i we create *at most* two new segments (a split at $d(i, i+1)$ and the explicit zero-inventory state for station $i+1$); every dominated segment is removed permanently. Hence $m_i \leq 2i$ and the total number of segment insertions and deletions across the entire run is $O(n)$. The same $O(n)$ bound holds for the number of MV keys that are ever created and subsequently removed.

Per-station operations accounting. For a fixed station i, we analyze each step of Alg. 1.

- **(a) IndividualUpdate**$(T, 1)$**.** Each loop iteration removes one dominated segment or converts one Terminus MV to a Regular MV and continues. Across *all* stations, there are at most $O(n)$ removals (each segment is removed once) and at most $O(n)$ conversions of MV values (since at the end of each station there is at most one additional terminus MV). Each action costs $O(\log n)$ to update the BST, so the total over the run is $O(n \log n)$; per station this contributes $O(\log n)$ amortized.
- **(b) Boundary candidate** $\mathsf{cand}_{\text{tmp}}$**.** A search at $x = d(i, i+1)$ returns the at-most-two bounding segments; cost evaluations are $O(1)$. Total $O(\log n)$ per station.
- **(c) CutSegmentAtBoundaryAndPutTheResultInT + SplitAt.** Cutting a single segment (if any) and splitting the tree by the boundary are both $O(\log n)$.
- **(d) BulkUpdateQAtP2**$(X, \cdot)$**.** Lazy annotation on the subtree root is $O(1)$; pushing cost is charged to subsequent searches/updates.
- **(e) Still-below-boundary top-ups in** X**.** Let s_i be the number of segments in X after the bulk-update whose terminus lies $< d(i, i+1)$. For all of these segments we compute one top-up candidate in $O(1)$ and keep the segment with the best value at $d(i, i+1)$ and then delete the rest of the segments in $O(\log n)$. Since each segment can become "below boundary" and be removed at most once in the whole run, $\sum_i s_i = O(n)$, giving a total of $O(n \log n)$ across all stations (and $O(\log n)$ amortized per station).
- **(f) Insert** $\mathrm{dp}(i+1, 0)$ **into** R**.** One insertion/replacement at the boundary in the tree R costs $O(\log n)$.
- **(g) IndividualUpdate**$(X, 2)$**.** After the bulk update, internal adjacencies in X are eligible for pruning at $p_{2,i}$. Exactly as in step (a), over the whole run there are $O(n)$ removals/conversions; total $O(n \log n)$ and $O(\log n)$ amortized per station.
- **(h) LineMergeBand**$(R|_{[Q,2Q]}, X)$**.** The merge trims or shrinks segments in the $[Q, 2Q]$ band. Each edited fragment is trimmed at most once in the

run. The number of edits is at most $O(n)$, and each edit induces at most $O(\log n)$ work to update the BST. Hence the total cost is $O(n \log n)$ over the run and $O(\log n)$ amortized per station.

- **(i) Cross-boundary dominance loop.** At the unique R–X join, the loop repeatedly removes a dominated neighbor and recomputes the new cross-boundary MV (Regular if boundary prices become equal, else Terminus). As each iteration deletes a segment, there can be at most $O(n)$ iterations in total; each iteration is $O(\log n)$. Thus $O(n \log n)$ overall and $O(\log n)$ amortized per station.
- **(j) Final IndividualUpdate$(T, 1)$.** Same accounting as (a): $O(n \log n)$ total, $O(\log n)$ amortized.
- **Capacity/holding updates.** TRIM$(T, B(i))$ performs at most one cut (and possibly deletes a tail of segments); across the run these deletions are covered by the global $O(n)$ removal bound, so $O(\log n)$ amortized per station. ADDHOLDINGCOSTSANDSUBTRACTDEMAND $(T, h(i), d(i))$ is a lazy value/MV-key shift and is $O(1)$ per station.

Putting it together. Every station performs a constant number of $O(\log n)$ structural operations (split/join/insert) and a constant number of dominance passes whose *total* number of removals/conversions across the run is $O(n)$. Since each removal/conversion/re-adjustment costs $O(\log n)$, the total cost over n stations is $O(n \log n)$.

Theorem 2. *Under the segment bound $m_i \leq 2i$ and with an augmented balanced-tree primitives, the per-station construction of $S(i)$ from $S_b(i)$ runs in total time $O(n \log n)$ and uses $O(n)$ space.*

6.5 An extension for $B(i) > 2Q$

This add-on is used *only* to answer $dp(i, x)$ for $x \leq B(i)$ using purchases at station i; it does *not* modify how we build $S_b(i{+}1)$. It is executed *after* the per-station routine has finished (i.e., after the final prune with $p_{1,i}$) and is fully reversed before processing station $i{+}1$.

Transient in-place expansion (no line-merge). Let each surviving segment S have anchor $dp(i, f)$, legacy discounted price $p(i, f)$ with residual $r(i, f)$, and terminus $t_S := f + r(i, f)$. Proceed as follows:

1. **Select eligible segments.** Mark the set

$$X^{\text{cap}} \;:=\; \{\, S \in T \;:\; t_S + Q \;<\; B(i) \,\},$$

i.e., exactly those segments that can legally accept a full Q-block at price $p_{2,i}$ without exceeding capacity.

2. **Apply a lazy bulk tag (in place).** On X^{cap} only, apply a transient lazy tag that grants a Q-block at unit price $p_{2,i}$ (all-units discount at station i). No *no line-merge* is performed.

3. **Local pruning inside X^{cap} and global reconnection.** Run IndividualUpdate limited to *internal* adjacencies of X^{cap} with comparator $p_{2,i}$. This improves query speed but must not remove or alter segments outside X^{cap} and must not use cross-boundary comparisons; the base frontier remains intact.

4. **Answer queries at station i.** To evaluate $dp(i, x)$ for any $x \leq B(i)$, perform the usual $O(\log n)$ search for the at-most-two bounding segments and compute their costs at x *with* the lazy tag in effect for any segment in X^{cap}. Take the minimum. Because the tag is transient and in-place, no explicit merge is needed.

5. **Reverse before $i{+}1$.** Remove the lazy tag from X^{cap} and discard any per-query caches so that T returns exactly to its post-step-(j) state. This prepares the tree for station $i{+}1$ and ensures the add-on is never used to construct $S_b(i{+}1)$.

Correctness and cost. The transient tag is never used to form $S_b(i{+}1)$, so *Observation 1* is respected. All comparisons during queries are pointwise and use the same piecewise-affine equations as in the core method; non-increasing prices guarantee that enabling $p_{2,i}$ is consistent with the dominance logic used inside X^{cap}. The add-on does not change the global $O(n \log n)$ bound; tagging and untagging are $O(1)$, and each query remains $O(\log n)$.

6.6 Keeping the optimal decisions

Keeping the optimal decisions can be done in two ways:

1. **Per-station operation log.** At each station, record every operation on the tree (individual updates, bulk updates) together with information about the *affected* elements. With this record, one can retrace every step for a given *solution* by *reconstructing the tree backward*. This takes $O(n \log n)$ time.

2. **Lazy branch tracking.** A more sophisticated approach lazily records, for each branch of the augmented tree T, the sequence of operations applied to that branch. This requires more careful pointer management to preserve operation order. When a sequence is pushed from a node to its children, keep in each child's list a pointer to a compact single-element proxy that references the original sequence; this lets the children's lists *share* the parent's sequence without copying.

7 Conclusion

We presented an $O(n \log n)$ algorithm for the single-item capacitated lot sizing problem with a 1-breakpoint all-units quantity discount in the case of non-increasing prices. Our algorithm works identically for integer and non-integer quantities because it computes (and prunes) the lower envelope of finitely many affine pieces using only slope comparisons. Our algorithm is an improvement over the previous $O(n^2)$ state-of-art algorithm for the same problem. Furthermore, we developed a rather general dynamic programming technique that may be applicable to similar problems in lot sizing or elsewhere. Finally an interesting avenue of research would be to investigate broad classes of cost and holding functions that preserve the validity of the lemmas of our algorithm.

Author's note on the use of AI. A generative AI assistant (ChatGPT, OpenAI GPT-5 Pro; October 2025) was used for language editing and presentation support—tightening phrasing, checking cross-references/section numbering, and proposing alternative wording for Section 2. The content of the appendix section such as the algorithms were generated by AI. All AI-assisted suggestions were manually reviewed and, where appropriate, rewritten before inclusion. The author takes full responsibility for the accuracy and integrity of the work; the AI tool is not an author.

References

[1] A. Atamtürk and D. S. Hochbaum. Capacity acquisition, subcontracting, and lot sizing. *Management Science*, 47(8):1081–1100, 2001.

[2] Peter Brass. *Advanced data structures*, volume 193. Cambridge University Press Cambridge, 2008.

[3] C.-S. Chung and C.-H. Lin. An O(T^2) algorithm for the NI/G/NI/ND capacitated lot size problem. *Management Science*, 34(3):420–426, 1988.

[4] P. Down, J. Smith, and R. Johnson. Linear cost algorithms for single-item lot sizing with quantity discounts. *Operations Research Letters*, 49(4):512–518, 2021.

[5] A. Federgruen and C.-Y. Lee. The dynamic lot size model with quantity discount. *Naval Research Logistics*, 37(5):707–713, 1990.

[6] Samir Khuller, Azarakhsh Malekian, and Julián Mestre. To fill or not to fill: The gas station problem. *ACM Transactions on Algorithms (TALG)*, 7(3):1–16, 2011.

[7] E. Koca, H. Yaman, and M. S. Aktürk. Lot sizing with piecewise concave production costs. *INFORMS Journal on Computing*, 26(4):767–779, 2014.

[8] C.-L. Li, J. Ou, and V. N. Hsu. Dynamic lot sizing with all-units discount and resales. *Naval Research Logistics*, 59(3-4):230–243, 2012.

[9] Y. Malekian, S. H. Mirmohammadi, and M. Bijari. Polynomial-time algorithms to solve the single-item capacitated lot sizing problem with a 1-breakpoint all-units quantity discount. *Computers & Operations Research*, 134:105373, 2021.

[10] S. H. Mirmohammadi and K. Eshghi. A hybrid polynomial-time algorithm for the dynamic quantity discount lot size model with resale. *Computers & Operations Research*, 39(7):1771–1778, 2012.

[11] J. Ou. Improved exact algorithms to economic lot-sizing with piecewise linear production costs. *European Journal of Operational Research*, 256(3):777–784, 2017.

8 Appendix

We will first define the operations required by our algorithm and then explain how they can be used to construct the functions in the pseudocode provided in the previous section and provide the time complexity for each operation. For these operations we use an augmented BST, where each of its internal nodes keeps additional information about the maximum MV values of their children.

A state segment is a 7-tuple containing the leftmost state(position and value) of the segment, its terminus position,its MV and its type(regular or terminus),its last used p_2 and the additional amount of that fuel it can use after its terminus point. For deletions and insertions, what matters for a state is the terminus point of a segment and the MV value, the rest of the information is just carried with it. The data structure needs for every operation to also apply each lazy tag on the visited nodes and then push it downward to its children. A data structure that supports the following basic operations in $\mathcal{O}(\log n)$ time is needed in order to implement the functions of our main algorithm:

Insert(x, w, v)**:** Inserts an segment (whose rightmost position/terminus is x) at position x along a one-dimensional axis. The element is given an MV of w and a value v. Its position is its terminus point.

Remove(e)**:** Given a pointer e, it deletes the element e from the structure.

Search(x)**:** Identifies the optimal segment t that covers the point x. The operation returns a reference to all the segments(2 at most) that overlap with x on the one-dimensional axis. This is done by searching for the element with the lowest terminus higher than x and the element with the highest terminus less than x. The optimal of the elements spans across x. We can find the optimal segment by generating the state using each of the two bounding segments.

ChangeMV(w, e)**:** Changes the MV to w of a specified segment e, identified by a reference e to the element.

FindMax: Locates the segment with the highest MV among all elements in the structure.

RemoveMax: Deletes the segment with the highest MV from the structure and reconfigures the position and MV of its adjacent segments

IncreaseMV(x, d)**:** Increases the MV of all elements from the element with the lowest position in the structure up to, but not including, the one that starts at or after position x. Each of these elements receives an increment of d units in MV.

IncreasePosition(x, d)**:** Increases the position by d of all elements starting from the element with the lowest position in the structure up to, but not including, the one that starts at or after position x.If the segment starts before x but ends after it we cut it at that position. Each of these elements receives an increment of d units in position value.

IncreaseValue(x, d)**:** Increases the value by d of all segments from the segment with the lowest position in the structure up to, but not including, the one that starts at or after position x. Each of these elements receives an increment of d units in value.

ChangePrice(x, a)**:** Changes the price of p_2 by a of all segments from the segment with the lowest position in the structure up to, but not including, the one that starts at or after position x. Each of these elements has the price of its cheapest fuel set to a.

Split(x)**:** Cuts the tree into two trees L and R, with L containing all the elements with positions lower or equal to x and R containing the rest of the elements.

Join(L, R)**:** Merges the trees L and R into a balanced binary search tree given that all the elements of one of the trees are at higher positions than the elements of the other tree.

8.1 Data Structure Implementation

The operations described above are implemented using an **augmented balanced binary search tree** (such as an AVL tree or Red-Black tree) where nodes are ordered by their terminus positions.

8.1.1 Position Coordinate System

We use an absolute coordinate system for inventory positions:

- Position 0 corresponds to zero inventory at the start of station 1

- For any station i, the position at zero inventory is $\sum_{j=1}^{i-1} d_j$ (cumulative distance)
- We maintain a prefix sum array: $D[i] = d(1, i-1) = \sum_{j=1}^{i-1} d_j$ for efficient position calculations
- A state with inventory f at station i has absolute position $D[i] + f$
- We can easily convert the absolute position system to a relative one which starts with zero at any station i by subtracting from all states the cumulative sum of distances up to that station.

This absolute positioning system allows us to:

- Compare positions across different stations uniformly
- Efficiently determine which segments can reach the next period
- Apply bulk updates by shifting absolute positions

8.1.2 Node Structure

Each node in the BST stores:

1. **Primary data:** The complete 7-tuple segment representation
2. **Auxiliary data for efficiency:**
    - `max_MV`: Maximum MV value in this node's subtree
    - `max_MV_node`: Pointer to the node with maximum MV in subtree
    - `has_terminus_MV`: Boolean indicating if subtree contains terminus-type MVs
3. **Lazy propagation fields:**
    - `position_delta`: Pending position increase for entire subtree
    - `value_delta`: Pending dp value increase for entire subtree
    - `MV_delta`: Pending MV increase for entire subtree (for holding costs)
    - p_2: Pending p_2 fuel price of fuel accessible for the segments of the entire subtree

8.1.3 Derived Attributes

Some segment attributes are computed dynamically as needed rather than stored:

- Once the latest p_2 price is pushed to a segment then actual $p(i, f)$ price is changed and stored in the segment tuple. Some attributes of a segment can be derived from the index of the station where it last received a bulk-update.

- The remaining capacity $r(i, f)$ is computed as:

$$r(i, f) = \min(\text{segment.r\_fuel}, B(i') - d(i', i)$$

  where i' is the period where the segment last received bulk update

- The current optimal fuel price for a segment considers both its stored p_2 price and the current period's $p_{1,i}$

8.1.4 Maintaining Auxiliary Information

After each modification (insertion, deletion, or lazy update), we update auxiliary fields bottom-up:

- `max_MV` = max(node.MV, left.max_MV, right.max_MV)
- `max_MV_node` points to the node achieving this maximum
- `has_terminus_MV` = node.is_terminus

This augmentation enables $\mathcal{O}(logn)$ access to the maximum MV and $\mathcal{O}(\log n)$ updates while maintaining the tree's balance properties.

Since these lazy updates always affect the entirety of the affected tree (either because they are applied to a subtree cut from the main tree or because they are applied as a holding cost across the whole tree), they are always inserted at the root. For a discussion of a more general version of lazy updates, see *Purely Functional Data Structures* by Chris Okasaki.

The operations split and join are by far the hardest from the operations of the augmented tree structure. These operations are designed to accommodate bulk updates. The states affected by a bulk update are first split into a separate tree on which the update is applied lazily. After the update, the resulting tree is reconnected to the remainder of the tree, and the MV value between the last non-bulk-updated element and the first bulk-updated element is recalculated. We include a slightly modified version of a standard join operation of balanced trees below:

Algorithm 4: (compact, strict lazy): Join Height-Balanced Trees

```
Input: T1, T2 AVL with all keys in T1 < all keys in T2
Output: AVL tree with all keys of T1 ∪ T2
if T1 = ∅ then
    return T2
if T2 = ∅ then
    return T1
PushDown(T1); Update(T1); PushDown(T2); Update(T2)
h1 ← height(T1); h2 ← height(T2)
if |h1 − h2| ≤ 1 then
    if h1 ≤ h2 then
        (x, T2') ← ExtractMin(T2)
        PushDown(T1); Update(T1)
        r ← new node; r.key ← x; r.left ← T1; r.right ← T2'; Update(r); return r
    else
        (x, T1') ← ExtractMax(T1)
        PushDown(T2); Update(T2)
        r ← new node; r.key ← x; r.left ← T1'; r.right ← T2; Update(r); return r
else
    if h1 ≤ h2 − 2 then
        (x, T2') ← ExtractMin(T2)
        u ← root(T2')
        while true do
            PushDown(u); Update(u)
            if u.left = ∅ then
                break
            PushDown(u.left); Update(u.left)
            if height(u.left) ≤ h1 then
                break
            u ← u.left
        v ← u.left; w ← new node; w.key ← x; w.left ← T1; w.right ← v
        if w.left ≠ ∅ then
            PushDown(w.left); Update(w.left)
        if w.right ≠ ∅ then
            PushDown(w.right); Update(w.right)
        Update(w); u.left ← w; Update(u); RebalanceUp(u)
        return root(T2')
    else
        (x, T1') ← ExtractMax(T1)
        u ← root(T1')
        while true do
            PushDown(u); Update(u)
            if u.right = ∅ then
                break
            PushDown(u.right); Update(u.right)
            if height(u.right) ≤ h2 then
                break
            u ← u.right
        v ← u.right; w ← new node; w.key ← x; w.left ← v; w.right ← T2
        if w.left ≠ ∅ then
            PushDown(w.left); Update(w.left)
        PushDown(w.right); Update(w.right); Update(w)
        u.right ← w; Update(u); RebalanceUp(u)
        return root(T1')
```

Algorithm 5: Utility: ExtractMax on AVL (removes and returns the maximum key)

Input: AVL tree T
Output: Pair (x, T') where $x = \max(T)$ and T' is T without x

```
if T.right = ∅ then
    return (T.key, T.left);
else
    PushDown(T);
    (x, R') ← ExtractMax(T.right);
    T.right ← R';
    RebalanceUp(T);
    return (x, T);
```

Algorithm 6: Utility: RebalanceUp (standard AVL maintenance with aggregates)

Input: Node u where local structure/children may have changed
Output: Tree remains AVL; heights and aggregates (MV, maxMV, costs, ...) are correct

```
while u ≠ ∅ do
    Update(u);    // recompute height and all augmented fields
    if balance(u) = 2 then
        if balance(u.left) < 0 then
            u.left ← rotateLeft(u.left)
        u ← rotateRight(u);
    else
        if balance(u) = −2 then
            if balance(u.right) > 0 then
                u.right ← rotateRight(u.right)
            u ← rotateLeft(u);
    u ← u.parent;
```

Algorithm 7: Utility: Singleton (one-node AVL tree)

Input: Key k
Output: A one-node AVL tree containing only k

```
r ← new node; r.key ← k; r.left ← ∅; r.right ← ∅;
Update(r);
return r;
```

Algorithm 8: (list-based): Split Height-Balanced Tree at k_{split}

Input: Augmented AVL tree T, split key k_{split}
Output: $(T_{\text{left}}, T_{\text{right}})$ with all keys in $T_{\text{left}} < k_{\text{split}} \leq$ all keys in T_{right}

$L_{\text{left}} \leftarrow$ empty list of pairs $(tree, key)$; $L_{\text{right}} \leftarrow$ empty list of pairs $(tree, key)$;
$node \leftarrow \text{root}(T)$;
while $node \neq \varnothing$ **do**
    `PushDown`($node$); `// apply the lazy updates (prices, holding costs, etc.) before using children`
    if $k_{split} \leq node.key$ **then**
        `// send` $node.key$ `to RIGHT (note the` $\leq$ `matches the spec)`
        prepend $(node.right,\ node.key)$ to L_{right};
        $node \leftarrow node.left$;
    else
        prepend $(node.left,\ node.key)$ to L_{left};
        $node \leftarrow node.right$;

`// Join LEFT trees | every saved key is preserved (no discards)`
$T_{\text{left}} \leftarrow \varnothing$;
foreach $(tree, key)$ ***in*** L_{left} ***from front to back*** **do**
    $mid \leftarrow$ `Singleton`(key);
    $A \leftarrow$ `Join`($tree$, mid);   `//` $\forall a \in tree < key$
    $T_{\text{left}} \leftarrow$ `Join`(A, T_{left});   `//` $\forall a \in A < \forall b \in T_{\texttt{left}}$

`// Join RIGHT trees | every saved key is preserved (no discards)`
$T_{\text{right}} \leftarrow \varnothing$;
foreach $(tree, key)$ ***in*** L_{right} ***from front to back*** **do**
    $mid \leftarrow$ `Singleton`(key);
    $T_{\text{right}} \leftarrow$ `Join`(T_{right}, mid);
    $T_{\text{right}} \leftarrow$ `Join`(T_{right}, $tree$);
return $(T_{\text{left}},\ T_{\text{right}})$;

Helper Functions (Specifications)

Data model and conventions. Each node stores: *key*, pointers *left*, *right*, optionally *parent*, cached *height*, an optional lazy tag *lazy*, and any application-specific aggregates (e.g., MV, maxMV, costs). We use $\text{height}(\varnothing) = -1$ so a leaf has height 0. All aggregates must be recomputed by `Update` after structural changes or rotations. If you do not keep `parent` pointers, whenever a routine says "returns the (sub)tree root," use the pointer the routine returns.

`height(x)` **Input:** node pointer x (possibly $\varnothing$).   **Output:** cached height of x with $\text{height}(\varnothing) = -1$.   **Time:** $O(1)$.

`balance(u)` **Input:** node u. **Output:** height(u.left)−height(u.right). **Time:** $O(1)$.

`root(x)` **Purpose:** ascend parent links to obtain the root of x's tree. **Input:** node x. **Output:** tree root (or just x if parents are not stored). **Time:** $O(h)$.

`Update(u)` **Purpose:** recompute cached fields of u from children.

$$u.\text{height} \leftarrow 1 + \max\big(\text{height}(u.\text{left}),\ \text{height}(u.\text{right})\big).$$

Also recompute all augmented fields deterministically from u.left, u's local value, and u.right (e.g., sums/mins/maxes, MV/maxMV). **Call after:** attaching/detaching children, finishing a rotation, or after children's lazies were pushed. **Time:** $O(1)$.

`PushDown(u)` **Purpose:** apply u's pending lazy tag to its children before reading them. If $u.\text{lazy} \neq I$ (identity):

- For each child $c \in \{u.\text{left}, u.\text{right}\}$, set $c.\text{lazy} \leftarrow \text{compose}(c.\text{lazy}, u.\text{lazy})$ and *update c's cached aggregates in place* to reflect the tag.
- If the tag semantics may *swap* children (e.g., "reverse subtree"), perform the swap at u now so future reads see the real left/right.
- Finally set $u.\text{lazy} \leftarrow I$.

Call before: any read of u.key, u.left/right, or a child's height/aggregates, and before following a child pointer. **Time:** $O(1)$.

`rotateLeft(u)` **Purpose:** single left rotation around u (RR/Right-heavy fix). **Pre:** $u.\text{right} \neq \varnothing$. If lazies can affect structure, it is safe to `PushDown` u and u.right first. **Effect:** promote $y = u.\text{right}$; $\beta = y.\text{left}$ becomes u.right; u becomes y.left; repair parents. **Post:** call `Update` on u then on y. **Output:** new local root y. **Time:** $O(1)$.

`rotateRight(u)` **Purpose:** single right rotation around u (LL/Left-heavy fix). **Pre/Effect/Post/Time:** symmetric to `rotateLeft`.

`RebalanceUp(u)` **Purpose:** restore AVL balance and recompute aggregates from u up to the root after a local change. **Loop:** while $u \neq \varnothing$:

1. `Update`(u).
2. If balance(u) = 2, do LL/LR (rotate right, possibly after left on u.left).
3. If balance(u) = −2, do RR/RL (rotate left, possibly after right on u.right).
4. Move to parent: $u \leftarrow u.\text{parent}$ (or caller-tracked ancestor).

Note: If your lazy tags can change structure/keys, `PushDown` the pivot child before a double rotation. **Time:** $O(h)$, with at most $O(1)$ rotations per level.

`ExtractMin(T)` **Purpose:** remove the minimum key of AVL tree T. **Method:** descend left; at each step `PushDown` the current node (and optionally the child), then go to .left. When the left child is $\varnothing$, return (T.key, T.right). On unwind, reattach, `Update` current node, and `RebalanceUp` from it. **Output:** pair (x, T') with $x = \min(T)$ and T' AVL without x. **Time:** $O(h)$.

`ExtractMax(T)` **Purpose/Method/Output/Time:** symmetric to `ExtractMin`, descending right.

`Singleton(k)` **Purpose:** create a one-node AVL tree. **Effect:** new node r with r.key $= k$, r.left $= r$.right $= \varnothing$, r.height $= 0$, r.lazy $= I$; initialize aggregates and call `Update`(r). **Time:** $O(1)$.

`compose(a,b)` **(lazy tags)** **Purpose:** combine two lazy tags so that applying a then b equals applying compose(a, b) once. Must be associative with identity I. **Examples:** range-add: numeric addition with $I = 0$; range-affine ($x \mapsto ax + b$): matrix/tuple composition with identity $(1, 0)$.

`apply_lazy_in_place(node, tag)` **Purpose:** reflect the lazy operation's effect in the node's cached aggregates *without* touching descendants immediately. **Examples:** for range-add δ, update sum $\leftarrow$ sum $+\delta\cdot$size, max $\leftarrow$ max $+\delta$, etc.

Call-order cheat sheet (for correctness with lazies).

1. **Before reading** a node's *key/left/right/height* or any child's height/aggregates: call `PushDown(node)` then `Update(node)`.
2. **Before comparing** a child's height to a threshold (e.g., in join's spine descent): `PushDown(child)` then `Update(child)`.
3. **After changing** a node's child pointer: `Update(node)` and then `RebalanceUp(node)`.
4. **Before any rotation:** if lazies can reorder structure, `PushDown` both the pivot and the child on which you rotate.
5. **When rebuilding in split:** before each `Join`, `PushDown+Update` on both inputs' roots.

Correctness invariants. (1) *BST order:* in-order traversal strictly increases by key. (2) *AVL balance:* for every node u, $\left|\text{height}(u.\text{left}) - \text{height}(u.\text{right})\right| \le 1$. (3) *Lazy safety:* all reads see fully applied parent tags thanks to `PushDown` before descent; all caches are up-to-date thanks to `Update` after writes/rotations. (4)

Aggregates: every augmented field is a pure function of children and local values, recomputed in `Update`.

Time Complexity: $O(\log n)$ - Following search path takes $O(\log n)$, and joining trees in increasing height order creates a geometric sum bounded by $O(\log n)$.

For more details on how the split and join operations on a balanced search tree can be performed in $\mathcal{O}(\log n)$ time, see [2] (pages 145–147). In essence, the split operation identifies and separates all subtrees containing elements with a value lower than x (with x being the terminus in our case) from the rest, and then reconnects them into two balanced trees. Additionally, all lazy updates along the affected path must be applied and pushed downward to the children of the affected nodes before any cutting or merging of subtrees occurs. In the merge operation, the maximum MV value must also be updated along the newly formed path from the roots of the connected subtrees upward toward the root of the main tree.

Algorithm 9

$\mathrm{dp}(t, i)$ represents the minimum cost for reaching time station t with i fuel in the fuel tank when ready to leave station t.

For every period $t \in \{1, \ldots, n\}$ and every feasible fuel level $i \leq B(t)$:

$$\mathrm{dp}(t, i) = \min_{\substack{0 \leq x_t \in \mathbb{N} \\ i' = i + d_t - x_t \\ 0 \leq i' \leq B(t-1)}} (\mathrm{dp}(t-1, i') + p_t(x_t) + i h_t)$$

with initial conditions:

$$\mathrm{dp}(0, 0) = 0, \quad \mathrm{dp}(0, i \neq 0) = +\infty$$

where the pricing function is:

$$p_t(x) = \begin{cases} p_{1,t} \cdot x, & x < Q \\ p_{2,t} \cdot x, & x \geq Q \quad (p_{2,t} \leq p_{1,t}) \end{cases}$$

A concrete ordering decision (i', x_t) is characterized by the station t and the fuel price $\alpha \in \{1 \text{ (using } p_{1,t}), 2 \text{ (using } p_{2,t})\}$.

Note: The relationship $i' = i + d_t - x_t$ represents that to have i items after satisfying demand d_t, we must have started with i' items and ordered x_t items, where $i' + x_t - d_t = i$.