This volume provides a comprehensive and accessible introduction to the theory and practice of inventory control—a significant research area central to supply chain planning. The book outlines the foundations of inventory systems and surveys prescriptive analytical models for deterministic inventory control. It further discusses prescriptive analytical techniques for demand forecasting in inventory control and also examines prescriptive analytical models for stochastic inventory control.

Inventory Analytics is the first book of its kind to adopt a practical, Python-driven approach to illustrating theories and concepts via computational examples, with each model covered in the book accompanied by its Python code. Originating as a collection of self-contained lectures, this volume is an indispensable resource for practitioners, researchers, teachers, and students alike.

This is the author-approved edition of this Open Access title. As with all Open Book publications, this entire volume is available to read for free on the publisher’s website. Printed and digital editions, together with supplementary digital material, can also be found at https://www.openbookpublishers.com.

Deterministic Inventory Control
Introduction

In this chapter, we discuss inventory control in a deterministic setting. We first discuss the cost factors that should be considered, and we show how to model and simulate the system running costs. We finally introduce prescriptive analytics models to determine the economic lot size under a variety of settings.

 Topics

• Accounting for costs p. 39
• The Economic Order Quantity (EOQ) p. 41
• Sensitivity to variations of the EOQ p. 46
• Incorrect estimation of ordering and holding costs p. 47
• The EOQ under a production/delivery lag p. 48
• Powers-of-two policies for the EOQ p. 49
• The EOQ under order quantity discounts: all-units discounts and incremental discounts p. 50
• The EOQ under planned backorders p. 53
• The Economic Production Quantity (EPQ) p. 55
• The Economic Lot Scheduling (ELS) p. 56
• Joint Replenishments p. 59
• Time-varying demand: Dynamic Lot Sizing p. 61
• Planned backorders in Dynamic Lot Sizing p. 64
• Order capacity constraints in Dynamic Lot Sizing p. 66
Accounting for costs

The simplest lot sizing instance one may conceive includes two cost factors: a fixed ordering cost \(K\), which is charged every time an order is issued, and it is a cost that is independent of the size of the order; and a per unit inventory holding cost \(h\), which is charged for every unit carried forward in stock from one period to the next in the planning horizon. In the first instance, we will assume that all demand must be met, hence the per unit item purchase cost can be ignored for all practical purposes. The revised Warehouse class is shown in Listing 13.

```python
from collections import defaultdict

class Warehouse:
    def __init__(self, inventory_level, fixed_ordering_cost, holding_cost):
        self.i, self.K, self.h = inventory_level, fixed_ordering_cost, holding_cost
        self.o = 0 # outstanding orders
        self.period_costs = defaultdict(int) # a dictionary recording cost in each period

    def receive_order(self, Q, time):
        self.review_inventory(time)
        self.i, self.o = self.i + Q, self.o - Q
        self.review_inventory(time)

    def order(self, Q, time):
        self.review_inventory(time)
        self.period_costs[time] += self.K # incur ordering cost and store it in a dictionary
        self.o += Q
        self.review_inventory(time)

    def on_hand_inventory(self):
        return max(0, self.i)

    def issue(self, demand, time):
        self.review_inventory(time)
        self.i = self.i - demand

    def inventory_position(self):
        return self.o + self.i

    def review_inventory(self, time):
        try:
            self.levels.append([time, self.i])
            self.on_hand.append([time, self.on_hand_inventory()])
            self.positions.append([time, self.inventory_position()])
        except AttributeError:
            self.levels, self.on_hand = [[0, self.i]], [[0, self.on_hand_inventory()]]
            self.positions = [[0, self.inventory_position()]]

    def incur_holding_cost(self, time): # incur holding cost and store it in a dictionary
        self.period_costs[time] += self.on_hand_inventory() * self.h
```

Listing 13 The extended Warehouse class that models costs.

To account for costs incurred by carrying over inventory from one period to the next we need to define an EndOfPeriod event (Listing 14) that is scheduled for the first time at the end of the first period, and which reschedules itself to occur at the end of every subsequent period.
class EndOfPeriod:
    def __init__(self, des: DES, warehouse: Warehouse):
        self.w = warehouse  # the warehouse
        self.des = des  # the Discrete Event Simulation engine
        self.priority = 2  # denotes a low priority
    def end(self):
        self.w.incur_holding_cost(self.des.time)
        self.des.schedule(EventWrapper(EndOfPeriod(self.des, self.w)), 1)

Example 2. We simulate operations of a simple inventory system by leveraging the Python code in Listing 15. The warehouse initial inventory is 0 units. The customer demand rate is 10 unit per period. We simulate $N = 20$ periods. We order 50 units in periods 1, 5, 10, and 15; the delivery lead time is 0 periods (i.e. no lead time). The fixed ordering cost is 100, the per unit inventory holding cost is 1. After simulating the system, we find that the average cost per unit time is 40; costs incurred in each period are shown in Table 1.

Listing 15 Simulating the behaviour of a warehouse in Python: inventory level and inventory position at the end of each period $t \in \{1, 20\}$ when the initial inventory level is 0; orders are scheduled every 5 periods, and order the lead time is 0. The fixed ordering cost is 100, the per unit inventory holding cost is 1.

<table>
<thead>
<tr>
<th>Period</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>140</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>140</td>
</tr>
<tr>
<td>7</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>140</td>
</tr>
<tr>
<td>12</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>140</td>
</tr>
<tr>
<td>17</td>
<td>30</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>19</td>
<td>10</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1 Costs incurred in each period $t \in \{1, 20\}$ when the initial inventory level is 0, orders are scheduled every 5 periods, and order the lead time is 0. The fixed ordering cost is 100, the per unit inventory holding cost is 1.
The Economic Order Quantity

Consider an inventory system subject to a constant rate of $d$ units of demand per time period. We shall assume that inventory is continuously reviewed (continuous review) and that the ordering/production process is instantaneous, i.e. as soon as we order a product or a batch of products, we immediately receive it. The order quantity can take any nonnegative real value. There is a proportional holding cost $h$ per unit per time period for carrying items in stock. All demand must be met on time, i.e. it cannot be backordered.

In absence of fixed costs associated with issuing an order or with setting up production, since we face inventory holding costs, it is clear that the best strategy to meet demand is to order/produce a product as soon as demand for it materialises: a pure reactive and lean strategy. In practice, however, firms do face fixed production/setup costs. In this case, the optimal control strategy is less obvious.

The problem of determining the “economic” order quantity (EOQ) in presence of fixed and variable production costs as well as proportional inventory holding cost was first studied by Harris at the beginning of the last century. For a historical perspective see [Erlenkotter, 1990]. Harris’ original “manufacturing quantity curves” are shown in Fig. 16.

The elements of the problem are summarized in Listing 16.

```python
class eoq:
    def __init__(self, K: float, h: float, d: float, v: float):
        """
        Constructs an instance of the Economic Order Quantity problem.
        Arguments:
            K (float) -- the fixed ordering cost
            h (float) -- the proportional holding cost
            d (float) -- the demand per period
            v (float) -- the unit purchasing cost
            ...
        """
        self.K, self.h, self.d, self.v = K, h, d, v
```

In the EOQ, the demand is constant, we operate under continuous review, and backorders are not allowed; hence, the following property ensures one does not incur unnecessary holding costs.

**Lemma 1** (Zero inventory ordering). Given an order quantity $Q$ it is optimal to issue an order as soon as the inventory level is zero.

The inventory level as a function of time is shown in Fig. 17: as soon as inventory level hits zero, an order of size $Q$ is immediately received and inventory immediately starts decreasing at rate $d$ unit per period; the cycle repeats when inventory level hits zero again.

**Definition 1** (Replenishment cycle). A replenishment cycle is the time interval comprised within two consecutive orders.
Lemma 2 (Cycle length). The length of an EOQ replenishment cycle is $Q/d$.

This is also known as the demand “coverage.”

Consider a replenishment cycle of length $R$ periods, a demand rate of $d$ units/period and an order quantity $Q = dR$, which covers exactly the demand over $R$ periods.

Lemma 3 (Average inventory level). The average inventory level over the cycle is $Q/2$.

Proof.

\[
\int_0^R (Q - dr) dr = \frac{d}{Q} \left[ Qd - \frac{d^2}{2} \right]_0^R = \frac{d}{Q} \left( \frac{Q^2}{2} - \frac{dQ^2}{2d^2} - (0Q - 0) \right) = Q/2.
\]

A key metric generally used to gauge inventory system performance is the so-called Implied Turnover Ratio.

Definition 2 (Implied Turnover Ratio). The Implied Turnover Ratio (ITR) represents the number of times inventory is sold or used in a time period; this is expressed as average demand over average inventory

\[
2d/Q.
\]

This information is important because it measures how fast a company is selling inventory and can be compared against industry benchmarks.

Cost analysis

The total cost of a strategy that issues an order of size $Q$ as soon as the inventory level reaches zero can be expressed in terms of ordering and holding cost per replenishment cycle

\[
C(Q) = \frac{K}{Q/d} + \frac{hQ}{2d} + dv
\]

(1)
Since we operate under an infinite horizon and we assume that all demand must be met, in our cost analysis we can safely ignore the variable purchasing cost $dv$, which is constant and independent of $Q$, and consider the total “relevant cost” $C_r(Q) = C(Q) - dv$. These concepts are implemented in Listing 17.

```python
class eoq:
    def cost(self, Q: float) -> float:
        return self.fixed_ordering_cost(Q) + self.variable.ordering_cost(Q) +
        self.holding_cost(Q)
    def relevant_cost(self, Q: float) -> float:
        return self.fixed_ordering_cost(Q) + self.holding_cost(Q)
    def fixed.ordering_cost(self, Q: float) -> float:
        K, d = self.K, self.d
        return K/(Q/d)
    def variable.ordering_cost(self, Q: float) -> float:
        d, v = self.d, self.v
        return d*v
    def holding.cost(self, Q: float) -> float:
        h = self.h
        return h*Q/2
```

In Fig. 18 we plot the different components that make up the EOQ cost function as well as $C_r(Q)$.

**Fig. 18** EOQ cost functions.

**Lemma 4** (Convexity of relevant cost). $C_r(Q)$ is convex.

**Proof.**

\[
\frac{d^2C_r(Q)}{dQ^2} = \frac{2kd}{Q^3} > 0.
\]
Optimal solution

Since $C(Q)$ is convex, its global minimum can be found via global optimisation approaches readily available in software libraries such as Python scipy. For instance, one may exploit Nelder-Mead\textsuperscript{13} algorithm as shown in Listing 18.

\begin{verbatim}
from scipy.optimize import minimize
class eoq:
    def compute_eoq(self) -> float:
        x0 = 1  # start from a positive EOQ
        res = minimize(self.relevant_cost, x0, method='nelder-mead',
                        options={'xtol': 1e-8, 'disp': False})
        return res.x[0]

Listing 18 Compute $Q^*$.
\end{verbatim}

The analytical closed-form optimal solution to the EOQ problem, the so-called Economic Order Quantity $Q^*$ is shown in the following Lemma.

Lemma 5 (Economic Order Quantity).

$$Q^* = \sqrt{\frac{2Kd}{h}}.$$ \hspace{1cm} (2)

Proof. By exploiting convexity of $C_r(Q)$, one sets its first derivative to zero

$$-\frac{Kd}{Q^2} + \frac{h}{2} = 0$$

and obtains a closed form for the optimal order quantity. \qed

The particular form of $Q^*$ allows us to make some observations: as $K$ increases we will issue larger orders; as $h$ increases holding inventory becomes more expensive and we order more frequently; finally, as $d$ increases the order quantity increases.

Lemma 6 (Relevant cost of ordering the Economic Order Quantity).

$$C_r(Q^*) = \sqrt{2Kdh}$$ \hspace{1cm} (3)

Proof. This is obtained by plugging Eq. 2 into $C_r(Q)$. \qed

Example 3. We consider the numerical example in Listing 19. Note that this is the same instance considered in Example 2. After running the code we obtain $Q^* = 44.72$ and $C_r(Q^*) = 44.72$. The replenishment cycle length is therefore $Q^*/d = 4.472$ periods.

\begin{verbatim}
instance = {'K': 100, 'h': 1, 'd': 10, 'v': 2}
pb = eoq(**instance)
Qopt = pb.compute_eoq()
print("Economic order quantity: "+\%2f\% Qopt)
print("Total relevant cost: "+\%2f\% pb.relevant_cost(Qopt))

Listing 19 Numerical example 3.
\end{verbatim}

The fact that $Q^* = C_r(Q^*)$ is a direct consequence of Lemma 6 and $h = 1$.  

Since we are operating under continuous review, although the instance parameters are the same, the cost obtained by applying the EOQ formula to the previous example is not directly comparable to that obtained via the simulation presented in Listing 15, which operates under periodic review. To address this issue, we need to adopt a finer discretisation of the planning horizon. This is shown in Listing 20 and in Listing 21. The cost of the simulated solution is now 44.78, which is equivalent to that obtained from the analytical solution. The behaviour of inventory over time is shown in Fig. 19.

Listing 20 Simulating the behaviour of a warehouse in Python: DES simulated EOQ solution under a finer discretisation of the simulation horizon (100 smaller period for each original period).

```python
instance = {'inventory_level': 0, 'fixed.ordering.cost': 100, 'holding.cost': 1.0/100}  # holding cost rescaled
w = Warehouse(**instance)

des = DES()

demand = 10/100  # discretise each period into 100 periods
Q = 44.72  # optimal EOQ order quantity
N = round(Q/demand)*10  # planning horizon length: simulate 10 replenishment cycles

des.schedule(EventWrapper(EndOfSimulation(des, w)), N)  # schedule EndOfSimulation

d = CustomerDemand(des, demand, w)
des.schedule(EventWrapper(d), 0)  # schedule a demand immediately

des.start()
print("Period costs: "+str([w.period_costs[e] for e in w.period_costs]))
print("Average cost per period: "+ '%.2f' % (100*sum([w.period_costs[e] for e in w.period_costs]) / len(w.period_costs))")

plot_inventory_finer(w.positions, "inventory position")
plot_inventory_finer(w.levels, "inventory level")
plt.legend(loc=1)
plt.show()
```

Listing 21 Method plot_inventory under a finer discretisation of the simulation horizon (100 smaller period for each original period).

```python
def plot_inventory(values, label):
    # data
df=pd.DataFrame({'x': np.array(values)[:,0], 'fx': np.array(values)[:,1]})
    # plot
    plt.xticks(range(0,len(values),200), range(0,len(values)//100,2))
    plt.xlabel("t")
    plt.ylabel("items")
    plt.plot('x', 'fx', data=df, linestyle='-', marker='', label=label)

plot_inventory(values, "items")
plt.xlabel("t")
plt.ylabel("items")
plt.plot('x', 'fx', data=df, linestyle='-', marker='', label=label)
```

Fig. 19 Simulating the behaviour of a warehouse in Python under a finer discretisation of the simulation horizon (100 smaller period for each original period).
Sensitivity to variations of $Q$ 

Suppose management decides to order a quantity $Q$ that differs from $Q^*$. The following Lemma is important in order to understand the sensitivity of the relevant cost to the choice of $Q$.

**Lemma 7** (Sensitivity to $Q^*$). Let $Q > 0$

\[ \frac{C_r(Q)}{C_r(Q^*)} = \frac{1}{2} \left( \frac{Q^*}{Q} + \frac{Q}{Q^*} \right) \]  

Proof.

\[
\frac{C_r(Q)}{C_r(Q^*)} = \frac{Kd}{Q\sqrt{2Kd/h}} + h\frac{Q}{2\sqrt{2Kd/h}} \\
= \frac{1}{2Q} \frac{2Kd}{h} + h \frac{Q}{2} \sqrt{\frac{h}{2Kd}} \\
= \frac{1}{2} \left( \frac{Q^*}{Q} + \frac{Q}{Q^*} \right).
\]

\[\text{(4)}\]

Sensitivity can be computed as shown in Listing 22.

There are two key observations: the sensitivity of the relevant cost to the choice of $Q$ only depends on $Q$ and $Q^*$, not on the specific values of problem parameters $K$ and $h$; moreover, this sensitivity is low.

**Example 4.** In Fig. 20 we plot Eq. 4 for the numerical example presented in Listing 19. We can see that a difference of 10 units between $Q^* = 44.72$ and $Q = 34.72$ only leads to a 3.22% cost increase.

![EOQ sensitivity to variations of Q from Q*](image-url)
Incorrect estimation of fixed ordering and holding costs

Suppose we incorrectly estimate the fixed ordering cost $K$ as $K'$, by leveraging once more Eq. 4 we obtain

$$\frac{C_r(Q')}{{C_r(Q^*)}} = \frac{1}{2^e}\left(\sqrt{\frac{K'}{K}}\right)$$

which implies that the cost of overestimating $K$ is lower than that of underestimating it. A similar analysis can be carried out for the holding cost parameter; for which, however, the situation is reversed: $C_r(Q')/C_r(Q^*) = 0.5\sqrt{h/h'}$.

**Fig. 21** EOQ sensitivity to $K$.

Sensitivity to $K$ and $h$ can be computed as shown in Listing 23.

**Fig. 22** EOQ sensitivity to $h$.

**Example 5.** In the numerical example presented in Listing 19, if we underestimate $K$ by 40% we underestimate $C_r(Q^*)$ by $0.5e(\sqrt{0.6}) = 0.0328$, i.e. 3.28%; if we overestimate $K$ by 40% we overestimate $C_r(Q^*)$ by $0.5e(\sqrt{1.4}) = 0.0141$, i.e. 1.41% (Fig. 21). The analysis carried out on $h$ leads to Fig. 22.
Production/delivery lag (lead time)

A key assumption in the EOQ problem formulation is that orders are delivered immediately. We will now relax this assumption and assume that orders are subject to a production/delivery lag of \( L \) periods.

By observing the behaviour of the inventory curve in Fig. 23 it is easy to see that the optimal solution does not change. The only adjustment required is to place an order \( L \) periods before the inventory level reaches zero. To determine when it is time to issue an order it is convenient to introduce the following definition.

**Definition 3 (Reorder point).** The reorder point \( r \) is the amount of demand observed over the lead time \( L \)

\[
r = dL.
\]

**Example 6.** In the numerical example presented in Listing 19, assuming \( L = 0.5 \), the reorder point is 5, which means an order is issued as soon as inventory drops to 5 units. The behaviour of inventory over time is shown in Fig. 24.
Powers-of-two policies

The problem statement resembles an EOQ setting; however rather than choosing an arbitrary optimal cycle length \( T \), we are given a base planning period \( T_b \) and we must choose an optimal cycle length taking the form \( T_b 2^k \), where \( k \in \{0, \ldots, \infty \} \). This is particularly useful in settings in which we seek order coordination across a range of stock keeping units.

Recall that \( Q^* = dT^* \), where \( T^* \) denotes the optimal cycle length of the EOQ. By substituting in \( C_r(Q) \) we can express the relevant cost as a function \( F(T) \) of the replenishment cycle length \( T \):

\[
F(T) = \frac{K}{T} + \frac{hdT}{2}.
\]

**Lemma 8** (Powers-of-two policy). Let \( T_b 2^k \) be a powers-of-two policy with base planning period \( T_b \), the optimal \( k \) is the smallest integer \( k \) satisfying

\[
F(T_b 2^k) \leq F(T_b 2^{k+1}).
\]

**Proof.** From Lemma 4 it immediately follows that \( F(T) \) is convex.

**Lemma 9** (Powers-of-two bound). Let \( T_b \) be a base planning period, then

\[
\frac{F(T_b 2^k)}{F(T^*)} \leq \frac{3}{2\sqrt{2}} \approx 1.06.
\]

**Proof.** From Eq. 4 we obtain

\[
\frac{F(T)}{F(T^*)} = \frac{1}{2} \left( \frac{T^*}{T} + \frac{T}{T^*} \right) = \frac{1}{2} e \left( \frac{T^*}{T} \right)
\]

where \( e(x) = x + 1/x \). Since

\[
F(T_b 2^k) \leq F(T_b 2^{k+1}) \to \sqrt{2}^{-1} T^* \leq T_b 2^k = T
\]

\[
F(T_b 2^{k-1}) > F(T_b 2^k) \to T = T_b 2^k \leq \sqrt{2} T^*,
\]

therefore \( \frac{T^*}{T} \leq \sqrt{2} \) and

\[
\frac{F(T_b 2^k)}{F(T^*)} \leq \frac{1}{2} e \left( \frac{1}{\sqrt{2}} \right) = \frac{1}{2} e(\sqrt{2}) = \frac{3}{2\sqrt{2}} \approx 1.06.
\]

An optimal powers-of-two policy can be computed as shown in Listing 24.

**Example 7.** In the numerical example presented in Listing 19, given a base planning period \( T_b = 0.7 \), the ratio \( F(T_b 2^k) / F(T^*) = 1.025 \leq 1.06 \); hence the resulting powers-of-two policy is only 2.5% more expensive than the optimal one.

---

```python
from sympy import *

class eoq:
    def opt_powersoftwo_policy(self, T: float) -> float:
        K, d, h = self.K, self.d, self.h
        rc = lambda t : K/t + h*d*t/2
        k = 0
        while rc(T*2**(k+1)) < rc(T*2**k):
            k += 1
        return T*2**k
```

Listing 24 Computing an optimal powers-of-two policy.
Quantity discounts

In several practical situations it is common to offer a discount on the purchasing price when the order quantity is high enough. In this problem setting we are given breakpoints $b_0, \ldots, b_{T+1}$, where $b_0 = 0$ and $b_{T+1} = \infty$, and associated purchasing prices $v_k$ for $k = 1, \ldots, T + 1$, where purchasing price $v_k$ applies within the range $(b_{k-1}, b_k)$. The structure of an instance is illustrated in Listing 25.

```python
class eoq_discounts(eoq):
    def __init__(self, K: float, h: float, d: float, b: List[float], v: List[float]):
        ""
        Constructs an instance of the Economic Order Quantity problem.
        Arguments:
        K {float} -- the fixed ordering cost
        h {float} -- the proportional holding cost as a percentage of purchase cost
        d {float} -- the demand per period
        b {float} -- a list of purchasing cost breakpoints
        v {float} -- a list of decreasing unit purchasing costs where v[j] applies in (b[j],b[j-1])
        ""
        self.K, self.h, self.d, self.b, self.v = K, h, d, b, v
        self.b.insert(0, 0)
        self.b.append(float("inf"))

    def compute_eoq(self) -> float:
        ""
        Computes the Economic Order Quantity.
        Returns:
        float -- the Economic Order Quantity
        ""
        quantities = [minimize(self.cost,
                                self.b[j-1]+1,
                                bounds=((self.b[j-1],self.b[j]),),
                                method='SLSQP',
                                options={'ftol': 1e-8, 'disp': False}).x[0]
                    for j in range(1, len(self.b))]
        costs = [self.cost(k) for k in quantities]
        return quantities[costs.index(min(costs))]
```

We still observe the fixed ordering cost $K$ and, as discussed, item purchasing price takes different values depending on the size of the order. Holding cost $h$, however, is no longer absolute and now denotes a percentage of the purchasing price.

Listing 25 also embeds a method `compute_eoq` which computes the economic order quantity. The total cost function is convex within each interval $(b_{k-1}, b_k)$; `compute_eoq` analyses each individual interval $(b_{k-1}, b_k)$ separately and returns the optimal $Q$ that minimizes the total cost across all possible intervals. This solution method works for any possible discount structure.

There are two types of discount strategies typically applied: all-units discounts and incremental discounts.

In **all-units discounts** purchasing price $v_k$ applies to the entire order quantity if this falls within the range $(b_{k-1}, b_k)$.

In **incremental discounts** purchasing price $v_k$ only applies to the fraction of order quantity that falls within within the range $(b_{k-1}, b_k)$. 

Listing 25 EOQ under quantity discounts.
All-units discounts

Variable ordering cost as a function of the ordering quantity $Q$ (unit_cost), as well as variable ordering cost (co_variable) and inventory holding cost (ch) per replenishment cycle can be computed as shown in Listing 26.

In this case, assuming $Q \in [b_j, b_{j+1})$, the total cost takes the form

$$C(Q) = \frac{Kd}{Q} + \frac{hv_j Q}{2} + v_j d$$

**Example 8.** We consider the numerical example in Listing 19. However, we consider all-units discounts with $b = \{0, 10, 20, 30, \infty\}$ and associated $v = \{5, 4, 3, 2\}$. The per unit purchasing cost as a function of the order quantity $Q$ is shown in Fig. 25. The total cost function is shown in Fig. 26. The economic order quantity is $Q^* = 31.6$ and the total cost is $C(Q^*) = 83.2$.

class eoq_all_units(eoq_discounts):
    def unit_cost(self, Q):
        j = set(filter(lambda j:
                        self.b[j-1] <= Q <
                        self.b[j],
                        range(1,len(self.b))))[0]
        return self.v[j-1]*Q
    def co_variable(self, Q):
        j = set(filter(lambda j:
                        self.b[j-1] <= Q <
                        self.b[j],
                        range(1,len(self.b))))[0]
        return self.v[j-1]*self.d
    def ch(self, Q: float) -> float:
        j = set(filter(lambda j:
                        self.b[j-1] <= Q <
                        self.b[j],
                        range(1,len(self.b))))[0]
        h = self.h*self.v[j-1]
        return h*Q/2

Listing 26 EOQ under all units quantity discounts.

Fig. 25 All units quantity discounts.

Fig. 26 EOQ total cost for all units quantity discounts.
**Incremental discounts**

Variable ordering cost as a function of the ordering quantity $Q$ (unit_cost), as well as variable ordering cost (co_variable) and inventory holding cost (ch) per replenishment cycle can be computed as shown in Listing 27.

In this case, the total cost takes the form

$$C(Q) = \frac{Kd}{Q} + \frac{hQ}{2} + \frac{c(Q)}{Q}d$$

where, assuming $Q \in [b_j, b_{j+1})$,

$$c(Q) = \sum_{i=0}^{j-1} v_i(b_{i+1} - b_i) + v_j(Q - b_j).$$

**Example 9.** We consider the numerical example in Listing 19. However, we consider all-units discounts with $b = \{0, 10, 20, 30, \infty\}$ and associated $v = \{5, 4, 3, 2\}$. The per unit purchasing cost as a function of the order quantity $Q$ is shown in Fig. 27. The total cost function is shown in Fig. 28. The economic order quantity is $Q^* = 39.9$ and the total cost is $C(Q^*) = 130$.
**Planned backorders in the EOQ**

In this section we still consider an EOQ setting, but we relax the assumption that all demand must be met on time. In other words, we will allow demand to be backordered and met when the successive replenishment arrives. The behaviour of the system is illustrated in Fig. 29. The system resembles the classical EOQ. However, the zero-inventory ordering property does not hold for this system. Instead, an order will be issued when inventory reaches $-S$, the planned backorder level. We therefore now have two decision to be made: how much to order ($Q$) and how much to backorder ($S$).

![EOQ inventory curve under planned backorders.](image)

**Cost analysis**

Incurring backorders must be expensive, otherwise the optimal policy would simply be to not order at all. More specifically, we will charge a penalty cost $p$ per unit backordered per period. The total relevant cost then becomes

$$C_b(Q, S) = \frac{Kd}{Q} + \frac{h(Q-S)^2}{2Q} + \frac{pS^2}{2Q}$$

**Lemma 10** (Holding cost reduction factor). *Allowing backorders is mathematically equivalent to reducing the holding cost rate by the factor $p/(p+h)$.*

**Proof.** To prove this, it is convenient to let $S = xQ$, where $x$ denotes the fraction of backordered demand in a replenishment cycle. Substitute in $C_b(Q, S)$, take partial derivatives w.r.t. $Q$ and $x$, and set both partial derivatives to zero. Interestingly,

$$\frac{dC_b(Q,S)}{dx} = -hQ(1-x) + pQx = 0$$

admits solution

$$x^* = \frac{h}{p+h} \quad (5)$$

which is independent of $Q$. If we plug $x^*$ into $C_b(Q, S)$, where $S = xQ$, we then obtain

$$C_b(Q) = \frac{hp}{h+p} \frac{Q}{2} + \frac{Kd}{Q} \quad (6)$$
which is the EOQ cost function in which the holding cost rate has been reduced by the factor $p/(p+h)$.

The planned backorder cost analysis can be implemented as shown in Listing 28.

```python
class eoq_planned_backorders:
        ""
        Constructs an instance of the Economic Order Quantity problem.
        Arguments:
        K {float} -- the fixed ordering cost
        h {float} -- the proportional holding cost
        d {float} -- the demand per period
        v {float} -- the unit purchasing cost
        p {float} -- the backordering penalty cost
        ""
        self.K, self.h, self.d, self.v, self.p = K, h, d, v, p

    def relevant_cost(self, Q: float) -> float:
        return self.co_fixed(Q)+self.ch(Q)+self.cp(Q)

    def cost(self, Q: float) -> float:
        return self.co_fixed(Q)+self.co_variable(Q)+self.ch(Q)+self.cp(Q)

    def co_fixed(self, Q: float) -> float:
        K, d = self.K, self.d
        return K/(Q/d)

    def co_variable(self, Q: float) -> float:
        d, v = self.d, self.v
        return d*v

    def ch(self, Q: float) -> float:
        h = self.h
        x = self.h/(self.p+self.h)
        return h*(Q-Q*x)**2/(2*Q)

    def cp(self, Q: float) -> float:
        p = self.p
        x = self.h/(self.p+self.h)
        return p*(Q*x)**2/(2*Q)
```

Listing 28 Planned backorder cost analysis.

**Optimal solution**

We next characterize the structure of the optimal solution by building upon Lemma 10.

**Lemma 11 (Optimal order quantity).** The optimal order quantity is

$$Q^* = \sqrt{\frac{2Kd(h+p)}{hp}}$$

(7)

**Lemma 12 (Optimal fraction of backordered demand).** The optimal fraction of backordered demand in a replenishment cycle $x^* = h/(p+h)$.

**Lemma 13 (Optimal cost).**

$$C_b^h(Q^*, x^*) = \sqrt{2Kdhp/(h+p)}$$

(8)

The computation is similar to that presented in Listing 18.

**Example 10.** In the numerical example presented in Listing 19 we consider a penalty cost $p = 5$; then $Q^* = 48.99$, $x^* = 0.16$, $C_b^h(Q^*, x^*) = 40.82$. 

Finite production rate: The Economic Production Quantity

In contrast to the previous section, we shall now relax the assumption that the whole replenishment quantity $Q$ is delivered at once at the beginning of the planning horizon. This problem is known as the Economic Production Quantity (EPQ).\(^{14}\) The quantity is instead delivered at a constant and finite production rate $p > d$, where $d$ is the demand rate. Taft’s original drawings are shown in Fig. 30. The behaviour of the system is illustrated in Fig. 31.

![Fig. 30 Taft’s inventory curve from [Taft, 1918] (courtesy of HathiTrust).](image)

![Fig. 31 EPQ inventory curve.](image)

Like in the classical EOQ, the zero inventory ordering property holds for this system. A replenishment occurs when inventory level is zero. Production runs until the whole replenishment quantity $Q$ is delivered. While the replenishment quantity is delivered, demand occurs at rate $d$, so inventory increases at a rate $p - d$ for $Q/p$ time periods until it reaches a maximum level $Q(1 - d/p)$, and then decreases at rate $d$ over the rest of the replenishment cycle.

Cost analysis

By observing that the average inventory level is now $Q(1 - p/d)/2$ we obtain the following expression for the total relevant cost

$$C_p^r(Q) = \frac{dK}{Q} + \frac{hQ(1 - p/d)}{2}.$$ \hspace{1cm} (9)

The EPQ analysis can be computed as shown in Listing 29.

Optimal solution

As in the previous case, the optimal solution is simply a minor modification of the classical EOQ solution

**Lemma 14** (Economic Production Quantity).

$$Q^* = \sqrt{\frac{2Kd}{h(1 - d/p)}}$$ \hspace{1cm} (10)

**Lemma 15** (Optimal cost).

$$C_p^r(Q^*) = \sqrt{2Kdh(1 - d/p)}.$$ \hspace{1cm} (11)

The computation is similar to that presented in Listing 18.

**Example 11.** In the numerical example presented in Listing 19 we consider a production rate $p = 5$; then $Q^* = 63.24$ and $C_p^r(Q^*) = 31.62$.

Production on a single machine: The Economic Lot Scheduling

Consider an EPQ problem, for the sake of convenience, we shall divide a production cycle into two phases: the “ramp up” phase and the “depletion” phase. As we have seen in the previous section, a replenishment occurs when inventory level is zero. Production runs until the whole replenishment quantity $Q$ is delivered. While the replenishment quantity is delivered, demand occurs at rate $d$, so inventory increases at a rate $p - d$ for $Q/p$ time periods (“ramp up” phase) until it reaches a maximum level $Q(1 - d/p)$; and then decreases at rate $d$ over the rest of the replenishment cycle, for $Q(1 - p/d)/d$ time periods (“depletion” phase). Observe that, naturally, the cycle length is the sum of the length of these two phases: $Q(1 - d/p)/d + Q/p = Q/d$ (Fig. 32).

Assume now that the production facility requires a setup time $s$ (e.g. cleaning, maintenance) before a production run may start. Let $Q^*$ be the EPQ (Lemma 14). If $s \leq Q(1 - p/d)/d$, the solution is clearly feasible and optimal for the EPQ under setup time, since production facility setup can occur during the “depletion” phase, when the machine is idle, without affecting the “ramp up” phase.

Example 12. In the numerical example presented in Listing 19, consider a production rate $p = 5$ and recall that $d = 10$; then $Q^* = 63.24$, and $Q^*(1 - p/d)/d = 3.162$. If $s \leq 3.162$, $Q^*$ remains the EPQ.

If, however, $s > Q^*(1 - p/d)/d$, since the total relevant cost is convex, then the EPQ can be computed by enforcing the condition $s = Q(1 - p/d)/d$. In other words, the optimal cycle length turns out to be the cycle length that corresponds to a schedule in which the machine is never idle: it is either producing during the “ramp up” phase, or being setup during the “depletion” phase.

The general expression for the EPQ under setup time is then

**Lemma 16** (Economic Production Quantity under setup time).

$$Q^* = \max \left\{ \sqrt{\frac{2Kd}{h(1 - d/p)}} s/(1 - p/d) \right\}. \quad (12)$$

Having considered the production of a single item on a single machine with finite production capacity and setup time, we now generalise our analysis to the production of $n$ items on a single machine with finite production capacity and item dependent setup times.$^{15}$

Let $p_i > d_i$ be the constant and finite production rate of item $i$, where $d_i$ is the demand rate for item $i$. If we allow arbitrary production schedules, a feasible solution exists if and only if $\sum_{i=1}^n d_i/p_i < 1$; however, finding an optimal production schedule is NP-hard and there is no closed form expression or efficient algorithm readily available. We will therefore focus on determining the best production cycle that contains a single run of each item. This means the cycle lengths of the $n$ items have to be identical. Such a schedule is referred to as a rotation schedule.
Rotation Schedule

Finding a rotation schedule when item setup costs \( K_i \) are independent across items is no more difficult than solving the single item problem. Let \( h_i \) be the holding cost per period for item \( i \) and consider a cycle of length \( T \).

**Lemma 17.** The average inventory level of item \( i \) during a cycle is \( d_i T (1 - d_i / p_i) / 2 \).

**Proof.** The length of the production run of item \( i \) in a cycle is \( T d_i / p_i \). As we have seen, a production run for item \( i \) must start only when inventory of item \( i \) is zero. During production (the "ramp up" phase), the level increases at rate \( p_i - d_i \), until it reaches level \( T d_i (p_i - d_i) / p_i \). After production (the "depletion" phase), inventory decreases at a rate \( d_i \) until it reaches zero and a new production run starts.

**Lemma 18.** The total relevant cost per unit time is

\[
C^*_r(T) = \sum_{i=1}^{n} \left( \frac{1}{2} h_i d_i T (1 - d_i / p_i) + \frac{K_i}{T} \right).
\]

**Proof.** Follows from Lemma 17 and from the fact that the average cost per unit time due to setups for item \( i \) is \( K_i / T \).

**Lemma 19.** The optimal cycle length is

\[
T^* = \sqrt{\frac{\sum_{i=1}^{n} K_i}{\sum_{i=1}^{n} h_i d_i (p_i - d_i) / 2p_i}}.
\]

**Proof.** Take the derivative of \( C^*_r(T) \) and set it to zero.

**Example 13.** Consider the instance illustrated in Table 2, in Fig. 33 we plot \( C^*_r(T) \). The optimal cycle length is \( T^* = 1.78 \). The rotation schedule is illustrated in Fig. 34; and in Fig. 35 for positive setup times.

<table>
<thead>
<tr>
<th>Item</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_i )</td>
<td>50</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>( p_i )</td>
<td>400</td>
<td>400</td>
<td>500</td>
</tr>
<tr>
<td>( h_i )</td>
<td>20</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>( K_i )</td>
<td>2000</td>
<td>2500</td>
<td>800</td>
</tr>
</tbody>
</table>

The ELS analysis can be computed as shown in Listing 30.

If we include setup times \( s_i \) that are sequence independent, the problem remains easy since the sum of the setup times will not depend on the production sequence. If the sum of the setup times is less than the idle time in the rotation schedule, the rotation schedule obtained by ignoring setup times remains optimal. Otherwise, as in the single item case, since the total relevant cost is convex, the optimal cycle length can be found by forcing the idle time to be equal to the sum of the setup times (Lemma 20).
Fig. 33 Overall and item-wise, total relevant cost $C_r(T)$ of the ELS problem instance in Example 13.

Fig. 34 The optimal rotation schedule for the problem instance in Example 13; solid areas denote production time.

Fig. 35 The optimal rotation schedule for the problem instance in Example 13 assuming all item setup times are equal to 0.1; solid areas denote production time.

**Lemma 20.** If the sum of the setup times exceeds the idle time in the rotation schedule, then

$$T^* = \frac{\left(\sum_{i=1}^{n} s_i\right)}{1 - \sum_{i=1}^{n} \frac{d_i}{p_i}}.$$ 

*Proof.* Follows from the convexity of total relevant cost $C_r(T)$.  

\[\square\]
**Synchronising production: The Joint Replenishment Problem**

The Joint Replenishment Problem (JRP) occurs when it becomes necessary to synchronise production of multiple items.

Consider a continuous review inventory system comprising \( n \) items. Let \( d_i \) be the demand rate for item \( i \), and \( h_i \) be the holding cost per time period for item \( i \). There are two types of fixed setup costs: the major setup cost \( K_0 \) for the system, and a minor setup cost \( K_i \) for each item type. Essentially, every time production occurs, the major setup cost \( K_0 \) is incurred, regardless of how many types of items are produced. Conversely, the minor setup cost \( K_i \) is incurred at time \( t \) if and only if item type \( i \) is produced at that time. The aim is to minimise the total cost per period.

Two questions must be answered to address the JRP:

- What is the optimal time \( T_0 \) between major setups?
- What is the optimal production cycle length \( T_i \) for item \( i \)?

**Lemma 21** (Zero inventory ordering). It is optimal to produce item \( i \) at time \( t \) if and only if its inventory level is zero.

**Lemma 22.** The holding cost per time period for item \( i \) is \( H_i \triangleq h_i d_i / 2 \).

The JRP in its general form is an NP-hard problem\(^\text{16}\) and, therefore, it is unlikely that an efficient algorithm to solve this problem will be found.

**Powers-of-two policies**

We shall here focus on a restricted version of the original problem: the JRP under a powers-of-two policy.\(^\text{17}\) For each item \( i \), rather than choosing an arbitrary optimal cycle length \( T_i \), we are given a base planning period \( T_b \) — which is assumed sufficiently small, and in particular, smaller than the cycle length of the most frequently ordered item — and we must choose an optimal cycle length taking the form \( T_b 2^k \), where \( k \in \{0, \ldots, \infty\} \). This leads to the following nonlinear programming model (problem \( Z \)).

\[
Z : \min \sum_{i=0}^{n} K_i / T_i + H_i T_i 
\]

Subject to,

\[
\begin{align*}
T_i &= M_i T_b & i &= 1, \ldots, n, \\
T_i &\geq T_0 & i &= 1, \ldots, n, \\
M_i &\in \{2^k | k = 0, 1, \ldots, \infty\},
\end{align*}
\]

where, for the sake of convenience, we let \( H_0 \triangleq 0 \).

Now, relax constraint \( \text{14} \) in problem \( Z \), and name the new problem obtained \( \hat{Z} \).

**Lemma 23.** \( \hat{Z} \) is a lower bound among all feasible policies.
Lemma 24. The solution to problem $Z$ is no more than 6% more expensive than the lower bound obtained via problem $\hat{Z}$.

Proof. Let $T^R_i, i = 0, \ldots, n$, be the optimal solution to the relaxed problem $\hat{Z}$. By following a line of reasoning similar to that presented in Lemma 9, one first proves that the powers-of-two restriction implies

$$\frac{1}{\sqrt{2}} \leq \frac{T^*_i}{T^R_i} \leq \sqrt{2}, \quad (17)$$

where $T^*_i$ is the optimal solution to the JRP under a powers-of-two policy (problem $Z$), $i = 0, \ldots, n$. The result then follows from Eq. 17 and from the convexity of the objective function of $\hat{Z}$. \hfill $\square$

The JRP can be modelled and solved by using ILOG CP Optimizer in Python as shown in Listing 31 and in Listing 32, the solution leverages Constraint Programming\textsuperscript{18} to deal with the nonlinear and discrete nature of the problem.

from docplex.cp.model import CpoModel
from typing import List

class jrp:
    def solve(self):
        mdl = CpoModel()
        M = mdl.integer_var_list(self.n+1, 0, self.U, "M")
        power = [2**i for i in range(0,self.U+1)]
        T = mdl.integer_var_list(self.n+1, 0, power[self.U], "T")
        mdl.add(mdl.element(power, M[i]) == T[i] for i in range(0,self.n+1))
        mdl.add(T[i] >= T[0] for i in range(0,self.n+1))
        mdl.minimize(mdl.sum(self.H[i]*T[i]/self.beta+self.K[i]/(T[i]/self.beta) for i in range(0,self.n+1)))
        print("Solving model....")
        msol = mdl.solve(TimeLimit=10, agent='local', execfile='/Applications/CPLEX_Studio1210/cpoptimizer/bin/x86-64_osx/cpoptimizer')
        msol.print_solution() if msol else print("no solution") # Print solution

Example 14. Consider a base planning period $T_b = 1/52$ (e.g. a planning on a weekly basis), and the problem parameters in Table 3. The total cost of a powers-of-two policy is 25.3, the optimal solution is to order items 1, 2, 3, 4 every $2^7 = 128$ weeks, and to order item 5 every 256 weeks. The lower bound obtained by solving the relaxed problem $\hat{Z}$ is 24.9, then $24.9 + 6\% = 26.4$; as expected, the total cost of a powers-of-two policy falls within the bounds $25.3 \in (24.9, 26.4)$.

<table>
<thead>
<tr>
<th>Item</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_j$</td>
<td>-2 2 2 2 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$h_i$</td>
<td>0 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$K_i$</td>
<td>5 1 2 4 6 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3 Problem parameters for the JRP problem instance (yearly rates).
Time-varying demand: Dynamic Lot Sizing

While exploring variants of the EOQ problem we maintained the assumption that demand rate is known and constant and that inventory is reviewed continuously. Both these assumptions may result unrealistic in practice. In fact, it is often the case that decision maker operate under a “periodic review” setting in which inventory can be reviewed — and orders issued — only at certain points in time. This leads to a discretization of the planning horizon into periods. Moreover, demand rate in practice is often not constant and varies from period to period.

In their 1958 seminal work Wagner and Whitin explored the Dynamic Version of the Economic Lot Size Model. The so-called Wagner-Whitin problem setting considers a finite planning horizon comprising \( T \) periods. Demand \( d_t \) may vary from one period \( t \) to another. Unlike the EOQ inventory can only be reviewed — an orders issued — at the beginning of each period. Like in the Economic Order Quantity orders are received immediately after being placed, there is a fixed cost \( K \) as well as variable cost \( v \) for placing an order. There is a proportional cost \( h \) for carrying one unit of inventory from one period to the next. Finally, all demand must be met on time and the initial inventory is assumed to be zero. It is safe to disregard the proportional ordering cost because the planning horizon is finite and all demand must be met, therefore this is in fact a constant. The Wagner-Whitin problem can be modelled in Python as shown in Listing 33.

As in the EOQ, we leverage the concept of replenishment cycle.

**Lemma 25.** The cost associated with a replenishment cycle starting in period \( i \) and ending in period \( j \) (included) is

\[
c(i,j) = K + h \sum_{k=i}^{j} (k-i)d_k.
\]

Costs \( c(i,j) \) can be computed as shown in Listing 34.

We can then represent the problem as a Directed Acyclic Graph (DAG) in which arcs represent all possible replenishment cycles that can take place within our \( T \)-period planning horizon (Fig. 38) and in which the cost associated with arc \( (i,j) \) is \( c(i,j-1) \).

---

The traditional Wagner-Whitin shortest path algorithm can be implemented in Python as shown in Listing 35.

```python
from typing import List
import networkx as nx
import itertools

class WagnerWhitinDP(WagnerWhitin):
    
    Implement the traditional Wagner-Whitin shortest path algorithm.

    def __init__(self, K: float, h: float, d: List[float]):
        super().__init__(K, h, d, 0)
        self.graph = nx.DiGraph()
        for i in range(0, len(self.d)):
            for j in range(i+1, len(self.d)):
                self.graph.add_edge(i, j, weight=self.cycle_cost(i, j-1))
```

Listing 35 Wagner-Whitin dynamic programming problem setup.

It can be shown that determining the cost of an optimal plan is equivalent to finding the shortest path in the aforementioned DAG. This can be done efficiently, for instance by leveraging Dijkstra's algorithm.\(^{20}\)

The cost of an optimal plan and associated order quantities can be retrieved as shown in Listing 36.

```python
class WagnerWhitinDP:
    def optimal_cost(self) -> float:
        
        Compute the cost of an optimal solution to the Wagner-Whitin problem

        T, cost, g = len(self.d), 0, self.graph
        path = nx.dijkstra_path(g, 0, T-1)
        path.append(len(self.d))
        cost += self.cycle_cost(path[t-1], path[t]-1)
        return cost

    def order_quantities(self) -> List[float]:
        
        Compute optimal Wagner-Whitin order quantities

        T, g = len(self.d), self.graph
        path = nx.dijkstra_path(g, 0, T-1)
        path.append(len(self.d))
        qty = [0 for k in range(0, T)]
        for t in range(1, len(path)):
            qty[path[t]-1] = sum([self.d[k] for k in range(path[t-1], path[t])])
        return qty
```

Listing 36 Wagner-Whitin problem solution cost retrieval.

**Example 15.** We now consider the Wagner-Whitin problem shown in Listing 37. The cost of an optimal plan is \(110\), and associated order quantities in each period are \(\{30, 0, 30, 40\}\). The optimal plan is visualised as a shortest path in Fig. 39.

```python
instance = {"K": 30, "h": 1, "d":[10,20,30,40]}
ww = WagnerWhitinDP(**instance)
print("Cost of an optimal plan: ", ww.optimal_cost())
print("Optimal order quantities: ", ww.order_quantities())
```

Listing 37 A Wagner-Whitin instance.


![Fig. 39. Wagner-Whitin instance.](image-url)
Positive initial inventory

Accounting for a positive initial inventory \( I_0 \) only requires a small modification to the DAG structure. Essentially, we must compute cycle costs as follows:

\[
c(i, j) = \begin{cases} 
  hI_0 + h \sum_{k=i}^{j} (k-i)d_k & \text{if } i = 1, \sum_{k=1}^{j} d_k \leq I_0; \\
  \infty & \text{if } i > 1, \sum_{k=1}^{j} d_k \leq I_0; \\
  K + h \sum_{k=i}^{j} (k-i)d_k & \text{otherwise.}
\end{cases}
\]

Moreover, while retrieving order quantities, we should bear in mind that we should issue an order in period \( t \), only if \( I_0 < \sum_{k=0}^{t} d_k \). Note that if \( I_0 \) exceeds the total demand over the planning horizon, then clearly it is optimal to never place any order.

A Wagner-Whitin instance with positive initial inventory is shown in Listing 38. The amended Python code is presented in Listing 39.

```python
instance = {"K": 30, "h": 1, "d": [10, 20, 30, 40], "I0": 40}
w = WagnerWhitinDP(**instance)
print("Cost of an optimal plan: ", w.optimal_cost())
print("Optimal order quantities: ", w.order_quantities())
```

Listing 38 A Wagner-Whitin instance.

Listing 39 Wagner-Whitin problem with positive initial inventory.
Planned backorders in Dynamic Lot Sizing

We consider an extension of the Wagner-Whitin problem setting in which demand can be backordered from one period to the next.

Consider a finite planning horizon comprising $T$ periods. Demand $d_t$ may vary from one period $t$ to another. Inventory can only be reviewed — an orders issued — at the beginning of each period. Orders are received immediately after being placed, there is a fixed cost $K$ as well as variable cost $v$ for placing an order. There is a proportional cost $h$ for carrying one unit of inventory from one period to the next. There is a proportional backorder/penalty cost $p$ for every unit that is backordered at the end of a period. The initial inventory is assumed to be equal to $I_0$. This problem can be modelled as follows.

$$\min \sum_{t \in T} \delta_t K + vQ_t + hI_t^+ + pI_t^-$$ (18)

Subject to,

$$Q_t \leq M\delta_t \quad t = 1, \ldots, T$$ (19)

$$I_0 + \sum_{k=0}^{t} (Q_k - d_k) = I_t \quad t = 1, \ldots, T$$ (20)

$$I_t = I_t^+ - I_t^- \quad t = 1, \ldots, T$$ (21)

$$Q_t, I_t^+, I_t^- \geq 0 \quad t = 1, \ldots, T$$ (22)

where $M = \sum_{t \in T} d_t$.

The Python code implementing this mathematical programming model is presented in Listing 40 and in Listing 41.

Listing 40 Wagner-Whitin problem with planned backorders, problem instance.

---

Planned backorders in Dynamic Lot Sizing


---
class WagnerWhitinPlannedBackordersCPLEX(WagnerWhitinPlannedBackorders):
    """
    Model and solve the Wagner-Whitin problem as an MILP via CPLEX
    """
    def model(self):
        model = Model("Wagner Whitin planned backorders")
        T, M = len(self.d), sum(self.d)
        idx = [t for t in range(0,T)]

        # Decision variables
        self.Q = model.continuous_var_dict(idx, name="Q")
        I = model.continuous_var_dict(idx, lb=-M, name="I")
        Ip = model.continuous_var_dict(idx, name="I^+")
        Im = model.continuous_var_dict(idx, name="I^-")
        delta = model.binary_var_dict(idx, name="delta")

        # Constraints
        for t in range(0,T):
            model.add_constraint(self.Q[t] <= delta[t]*M) # Eq. 14
            model.add_constraint(self.I0 + model.sum(self.Q[k] - self.d[k] for k in range(0,t+1)) == I[t]) # Eq. 15
            model.add_constraint(I[t] == Ip[t]-Im[t]) # Eq. 16
            model.add_constraint(self.Q[t] >= 0) # Eq. 17a
            model.add_constraint(Ip[t] >= 0) # Eq. 17b
            model.add_constraint(Im[t] >= 0) # Eq. 17c
        model.minimize(model.sum(delta[t] * self.K + self.Q[t] * self.v + self.h * Ip[t] + self.p * Im[t] for t in range(0,T))) # Eq. 13

        model.print_information()
        self.msol = model.solve()
        if self.msol:
            model.print_solution()
        else:
            print("Solve status: " + self.msol.get_solve_status() + "\n")

    def order_quantities(self) -> List[float]:
        """
        Compute optimal Wagner-Whitin order quantities
        """
        return [self.msol.get_var_value(self.Q[t]) for t in range(0,len(self.d))]

    def optimal_cost(self) -> float:
        """
        Compute the cost of an optimal solution to the Wagner-Whitin problem
        """
        return self.msol.get_objective_value()

instance = {'K': 40, 'v': 1, 'h': 1, 'p': 2, 'd':[10, 20, 30, 40], 'I0': 0}
p = WagnerWhitinPlannedBackordersCPLEX(**instance)
p.model()

The optimal ordering plan for the instance in Listing 42 is illustrated in Fig. 40.

![Diagram](image.png)
**Order quantity capacity constraints in Dynamic Lot Sizing**

We consider an extension of the Wagner-Whitin problem setting in which capacity constraints are imposed on the order quantity in each period.

Consider a finite planning horizon comprising $T$ periods. Demand $d_t$ may vary from one period $t$ to another. Inventory can only be reviewed — an orders issued — at the beginning of each period. The maximum order quantity in each period is $C$. Orders are received immediately after being placed. There is a fixed cost $K$ as well as variable cost $v$ for placing an order. There is a proportional cost $h$ for carrying one unit of inventory from one period to the next. Finally, all demand must be met on time and the initial inventory is assumed to be equal to $I_0$. This problem can be modelled as follows.

$$\min \sum_{t \in T} \delta_t K + vQ_t + hI_t$$

Subject to,

$$Q_t \leq C\delta_t \quad t = 1, \ldots, T$$

$$I_0 + \sum_{k=0}^{t} (Q_k - d_k) = I_t \quad t = 1, \ldots, T$$

$$Q_t, I_t \geq 0 \quad t = 1, \ldots, T$$

The Python code implementing this mathematical programming model is presented in Listing 43 and in Listing 44.

```python
from typing import List
class CapacitatedLotSizing:
    """
    A capacitated lot sizing problem under capacity constraints.
    Deterministic production planning: Algorithms and complexity.
    """
    def __init__(self, K: float, v: float, h: float, d: List[float], I0: float, C: float):
        """
        Create an instance of the capacitated lot sizing problem.
        Arguments:
        K (float) -- the fixed ordering cost
        v (float) -- the per unit ordering cost
        h (float) -- the per unit holding cost
        d (List[float]) -- the demand in each period
        I0 (float) -- the initial inventory level
        C (float) -- the order capacity
        """
        self.K, self.v, self.h, self.d, self.I0, self.C = K, v, h, d, I0, C
```

The optimal ordering plan for the instance in Listing 45 is illustrated in Fig. 41.
from docplex.mp.model import Model
import sys
sys.path.insert(0, './Applications/CPLEX_Studio128/cplex/Python/3.6/x86-64_osx')

class CapacitatedLotSizingCPLEX(CapacitatedLotSizing):
    ""
    Solves the capacitated lot sizing problem as an MILP.
    ""

def __init__(self, K: float, v: float, h: float, d: List[float], I0, C: float):
    ""
    Create an instance of the capacitated lot sizing problem.
    Arguments:
        K {float} -- the fixed ordering cost
        v {float} -- the per unit ordering cost
        h {float} -- the per unit holding cost
        d {List[float]} -- the demand in each period
        I0 {float} -- the initial inventory level
    ""
    super().__init__(K, v, h, d, I0, C)
    self.model()

def model(self):
    ""
    Model and solve the capacitated lot sizing problem via CPLEX
    ""
    model = Model("Capacitated lot sizing")
    T = len(self.d)
    idx = [t for t in range(0,T)]
    self.Q = model.continuous_var_dict(idx, name="Q")
    I = model.continuous_var_dict(idx, lb=0, name="I")
    delta = model.binary_var_dict(idx, name="delta")
    for t in range(0,T):
        model.add_constraint(self.Q[t] <= delta[t]*self.C)
        model.add_constraint(self.I0 + model.sum(self.Q[k] - self.d[k] for k in
                range(0,t+1)) == I[t])
        model.add_constraint(self.Q[t] >= 0)
        model.add_constraint(I[t] >= 0)
    model.minimize(model.sum(delta[t] * self.K + self.Q[t] * self.v + self.h *
                I[t] for t in range(0,T)))
    model.print_information()
    self.msol = model.solve()
    if self.msol:
        model.print_solution()
    else:
        print("Solve status: " + self.msol.get_solve_status() + ":\n"

def order_quantities(self) -> List[float]:
    ""
    Compute optimal capacitated lot sizing order quantities
    ""
    return [self.msol.get_var_value(self.Q[t]) for t in range(0,len(self.d))]

def optimal_cost(self) -> float:
    ""
    Compute the cost of an optimal solution to the capacitated lot sizing problem
    ""
    return self.msol.get_objective_value()

instance = {"K": 40, "v": 1, "h": 1, "d":[10,20,30,40], "I0": 0, "C": 30}
CapacitatedLotSizingCPLEX(**instance)
Computational complexity

The capacitated lot sizing problem is known to be NP-hard. This means that it is unlikely we will ever find an efficient solution method to compute optimal replenishment plans. Apart from the mathematical programming model presented in the previous section, the problem can also be solved via dynamic programming.

Dynamic programming formulation

Consider the capacitated lot sizing problem,

- $T$ is the number of periods;
- the state $s$ is the initial inventory level in period $t$; therefore $S_t \triangleq \{0, \ldots, D\}$, for $t = 1, \ldots, T$, where $D = \sum_{t=1}^{T} d_t$;
- the action $a$ is the order quantity in period $t$; therefore $A_s \triangleq \{0, \ldots, C\}$ for any state $s$;
- the state transition function is simply $g_t(s, a) \triangleq s + a - d_t$;
- the immediate cost if action $a \in A_s$ is taken in state $s \in S_t$, is

$$c(s, a) \triangleq \begin{cases} K + av + h \max(s + a - d_t, 0) + M \max(d_t - s + a, 0) & a > 0, \\ h \max(s + a - d_t, 0) + M \max(d_t - s + a, 0) & \text{otherwise}, \end{cases}$$

where $M$ is a large number;
- the functional equation is

$$f_t(s) = \min_{a \in A_s} c(s, a) + f_{t+1}(g_t(s, a))$$  \hspace{1cm} (27)$$

for which the boundary condition is $f_{T+1}(s) \triangleq 0$ for all $s \in S_{T+1}$.

The goal is to determine $f_t(s)$, where $s$ denotes the initial inventory level in the first period.

We next show how to model and solve the capacitated lot sizing problem via dynamic programming in Python.

The State class (Listing 46), is used to capture a state of the system. Note that we implement function __eq__ to ensure to states can be compared with each other.

---


22 For more information on dynamic programming refer to the Appendix.
from typing import List

class State:
    
    The state of the inventory system.
    
    def __init__(self, t: int, I: float):
        
        Instantiate a state

        Arguments:
            t {int} -- the time period
            I {float} -- the initial inventory

        self.t, self.I = t, I
        
    def __eq__(self, other):
        return self.__dict__ == other.__dict__
        
    def __str__(self):
        return str(self.t) + " " + str(self.I)
        
    def __hash__(self):
        return hash(str(self))

class CapacitatedLotSizingSDP(CapacitatedLotSizing):
    
    Solves the capacitated lot sizing problem as an SDP.

    def __init__(self, K: float, v: float, h: float, d: List[float], I0, C: float):
        
        Create an instance of the capacitated lot sizing problem.

        Arguments:
            K {float} -- the fixed ordering cost
            v {float} -- the per unit ordering cost
            h {float} -- the per unit holding cost
            d {List[float]} -- the demand in each period
            I0 {float} -- the initial inventory level

        super().__init__(K, v, h, d, I0, C)
        
        # initialize instance variables
        self.T, min_inv, max_inv, M = len(d), 0, sum(d), 100000
        
        # lambdas
        self.ag = lambda s: [x for x in range(0, min(max_inv-s.I, self.C+1))] # action generator
        self.st = lambda s, a, d: State(s.t+1, s.I+a-d) # state transition
        L = lambda i,a,d : self.h*max(i+a-d, 0) + M*max(d-i-a, 0) # immediate holding/penalty cost
        self.iv = lambda s, a, d: (self.K+v*a if a > 0 else 0) + L(s.I, a, d) # immediate value function
        self.cache_actions = {} # cache with optimal state/action pairs
        
        print("Total cost: " + str(self.f(self.I0)))
        print("Order quantities: " + str([Q for Q in self.order_quantities()]))

    def _compute_order_quantities(self):
        
        Compute optimal capacitated lot sizing order quantities

        I = self.I0
        for t in range(len(self.d)):
            Q = self.q(t, I)
            I += Q - self.d[t]
            yield Q

    def order_quantities(self) -> List[float]:
        return [Q for Q in self._compute_order_quantities()]

    def optimal_cost(self) -> float:
        
        Compute the cost of an optimal solution to the capacitated lot sizing problem

        return self.f(self.I0)
To model this problem, we adopted a trick: it is clear that, since inventory cannot be negative, given a period \( t \) and an initial inventory level, some actions may be infeasible, therefore in general the space of possible action \( A_s \) may not be equal to \{0, \ldots, C\}. It may be complex to determine what values \( a \in \{0, \ldots, C\} \) are feasible for a given state \( s \). To overcome this difficulty, we allowed potentially infeasible actions, but we associated a very high cost (the large number \( M \)) to infeasible states in the immediate cost function \( c(s,a) \).
In dynamic programming, an optimal solution can be obtained via forward recursion or backward recursion. In Listing 47 and Listing 48 we present a solution based on forward recursion.

The action generator function, the state transition function, and the immediate value function are conveniently captured in Python via lambda expressions. A generic dynamic programming forward recursion that leverages these lambda expressions is implemented in function \( f \); this function is a direct implementation of Eq. 27.

Finally, the memoize class (Listing 49) is a decorator\(^{23}\) used to tabulate function \( f \) and make sure that, if \( f(s) \) has been already computed for a given state \( s \), this computation does not happen twice. This function leverages the \( \_\_hash\_ \) function of class State to store and retrieve states stored in the cache.

```python
import functools
class memoize(object):
    
    # Memoization utility
    
    def __init__(self, func):
        self.func, self.memoized, self.method_cache = func, {}, {}
    def __call__(self, *args):
        return self.cache_get(self.memoized, args, lambda: self.func(*args))
    def __get__(self, obj, objtype):
        return self.cache_get(self.method_cache, obj,
            lambda: self.__class__(functools.partial(self.func, obj)))
    def cache_get(self, cache, key, func):
        try:
            return cache[key]
        except KeyError:
            cache[key] = func()
            return cache[key]
    def reset(self):
        self.memoized, self.method_cache = {}, {}
```

Listing 49 Memoization utility.

A sample instance is presented in Listing 50. The solution to this instance is of course the same already illustrated in Fig. 41.

```python
instance = {"K": 40, "v": 1, "h": 1, "d":[10,20,30,40], "I0": 0, "C": 30}
CapacitatedLotSizingSDP(**instance)
```

Listing 50 Capacitated lot sizing, sample instance.

When the capacity is the same in every period, the problem has polynomial \( O(T^4) \) complexity.\(^{24}\)

