2007

TR-2007018: Peak Shaving through Resource Buffering

Amotz Bar-Noy

Matthew P. Johnson

Ou Liu

Peak Shaving Through Resource Buffering[*]

Amotz Bar-Noy[†] Matthew P. Johnson[‡] Ou Liu[†]

October 15, 2007

Abstract

We introduce and solve a new problem inspired by energy pricing schemes in which a client is billed for peak usage. Given is a sequence of n positive values $(d_1, d_2, \ldots, d_n)$ that represent resource demands (typically energy) over time. At each timeslot i, the system requests a certain amount r_i to meet the demand d_i. The added piece of infrastructure is the *battery*, which can store surplus resource for future use. The demands may represent required amounts of energy, water, or any other *tenable* resource which can be obtained in advance and held until needed. In a feasible solution, each demand must be supplied on time, through a combination of newly requested energy and energy withdrawn from the battery. The goal is to minimize the maximum request. We consider batteries with and without a bounded capacity, and with or without a percentage loss in charging due to inefficiency. In the online version of this problem, the algorithm must determine request r_i without knowledge of future demands, with the goal of maximizing the amount by which the peak is reduced.

We give efficient combinatorial algorithms for the offline problem, which are optimal for all four battery types. Central to our analysis is a mathematical property we call a *generalized average*. Our fastest offline algorithms for the lossy battery settings compute a series of generalized averages with the aid of balanced binary search trees. We also show how to find the optimal offline battery size, for the setting in which the final battery level must equal the initial battery level. In the online setting, we focus on lossless batteries, with and without a capacity bound. We prove that no purely online algorithm can have competitive ratio better than n, and that if the peak demand is revealed in advance, no online algorithm can have competitive ratio better than H_n. We give two simple online algorithms, the fastest one with $O(1)$ per-slot running-time, which meet this bound.

[*]This work was supported by grants from the National Science Foundation and the New York State Office of Science, Technology and Academic Research.

[†]amotz@sci.brooklyn.cuny.edu, City University of New York, Graduate Center, and Brooklyn College.

[‡]{mjohnson,oliu}@cs.gc.cuny.edu, City University of New York, Graduate Center.

1 Introduction

Power companies charge some high-consumption clients not just for the total amount of power consumed, but also for how quickly they consume it. Within the billing period (typically a month), the client is charged for the amount of energy used (*usage charge*, in kWh) and for the maximum amount requested over time (*peak charge*, in kW). If demands are given as a sequence $(d_1, d_2, \ldots, d_n)$, then the total bill is of the form $c_1 \sum_i d_i + c_2 \max_i\{d_i\}$, i.e., a weighted sum of the total usage and the maximum usage. (In practice, the discrete timeslots may be 30-minute averages [2].) This means a client who powers a 100kW piece of machinery for one hour and then uses no more energy for the rest of the month would be charged more than a client who uses a total of 100kWh spread evenly over the course of the month. Since the per-unit cost for peak charge may be on the order of 100 times the per-unit cost for total usage [3], this difference can be significant.

At least one start-up company [1] is currently marketing battery-based systems intended to reduce peak energy charges. In such a system, a battery is placed between the power company and a high-consumption client site, in order to smooth power requests and shave the peak. The client site will charge to the battery when demand is low and discharge when demand is high. Spikes in the demand curve can thus be consistent with a relatively flat level of supplied power. The result is a lower cost for the client and a more manageable request curve for the provider.

It is interesting to note that a battery system may actually *raise* energy usage, since there may be energy loss due to inefficiency in AC/DC conversion. Serving peak requests during periods of high demand is a difficult and expensive task for the power company, however, and the event of a black-out inflicts high societal costs. While a battery system may involve higher total energy requests, it may benefit the system as a whole by easing the strain of peak demands. Combined with alternative energy sources such as solar panels, the system could even lower the net commercial power usage. Alternative energy sources are typically low-cost but unreliable, since they depend on external events such as the weather. With a battery, this energy can be stored until needed.

We may generalize this problem of minimaxing the request to any resource which is *tenable* in the sense that it may be obtained early and stored until needed. For example, companies frequently face shortages of popular products: "Plentiful supply [of Xboxes] would be possible only if Microsoft made millions of consoles in advance and stored them without releasing them, or if it built vast production lines that only ran for a few weeks–both economically unwise strategies" [11]. A producer could smooth the product production curve by increasing production and warehousing supply until future sales. But when should the producer "charge" or "discharge"? A third application is scheduling of jobs that are composed of generic work-units that may be done in advance. Although the problem is very general, we will use the language of energy and batteries for concreteness.

In the online version of our problem, the essential choice faced at each timeslot is whether (and by how much) to invest in the future or to cash in a prior investment. The investment in our setting is a request for more energy than is needed at the time. If the algorithm only asks for the minimum required, then it is vulnerable to spikes in demand; if it asks for much more energy than it needs, then the greater request could itself become a new, higher peak. The strictness of the problem lies in the fact that the cost is not cumulative: we want *every* request to be low.

Background. There is a wide literature on commodity production, storage, warehousing, and supply-chain management (see e.g. [13, 18, 9, 15]). More specifically, there are a number of inventory problems based on the Economic Lot Sizing model [8], in which demand levels for a product vary over a discrete finite time-horizon and are known in advance. A feasible solution in these problems must obtain sufficient supply through production (sometimes construed as ordering) or through other methods, in order to meet each of the demands on time, while observing certain constraints. The solution quality may depend on multiple costs, which vary by formulation.

One such inventory problem is Single-Item Lot-Sizing, in which sufficient supplies must be ordered to satisfy each demand, while minimizing the total cost of ordering charges and holding charges. The ordering charge consists of a fixed charge per order plus a charge linear in order size. The holding charge for inventory is per-unit and per-timeslot. There is a tradeoff between these incentives since fixed ordering charges encourage large orders while holding charges discourage them. Wagner & Whitin [17] showed in 1958 that this problem can be solved in polynomial time. Under the assumption of *non-speculative costs*, in which case orders should always be placed as late as possible, the problem can be solved in linear time. Such "speculative" behavior, however, is the very motivation of our problem. There are a variety of lot-sizing variations, including constant-capacity models that limit the amount ordered per time period. (See [15] and references therein.) Our offline problem differs in that our objective is minimizing this constant capacity (for orders), subject to a bound on *inventory* size, and our inventory is free but *lossy*.

Another related inventory problem is Capacity and Subcontracting with Inventory (CSI) [5], which incorporates trade-offs between production costs, subcontracting costs, holding costs, and the cost for maximum per-unit-timeslot production capacity. The goal in that problem is to choose a production capacity and a feasible production/subcontracting schedule that together minimize total cost. One known algorithm ("Algorithm 1" of [5]) simplifies to a linear-time algorithm for our simplest (lossless/unbounded) offline problem setting, but it is more complicated than our algorithm for that case and it does not support our novel extensions of (analogously) inventory capacity and inventory loss. Inventory problems have also been studied for probabilistic models. Scarf [16] solved the single-period newsvendor problem, in which a vendor decides how much stock to order in advance for sale during the timeslot, before learning the demand for that period. His method finds the stocking policy that maximizes the minimum expected profit, over all consistent distributions. See [14] for an example of recent work in this mode.

In the minimax work-scheduling problem [12], the goal is to minimize the maximum amount of work done in any timeslot over a finite time-horizon. Our online problem is related to a previously studied special case in which jobs with deadlines are assigned online. In that problem, all work must be done by deadline but cannot be begun until assigned. Subject to these restrictions, the goal is to minimize the maximum work done in any timeslot. While the optimization goal is the same, our problem differs in two respects. First, each job for us is due immediately when assigned. Second, we *are* allowed to do work (request and store energy) in advance. One online algorithm for the jobs-by-deadlines problem is the *Alpha policy* [12]: at each timeslot, the amount of work done is *alpha times the maximum per-unit-timeslot amount of work that OPT would have done, when running on the partial input received so far*. One of our online algorithms adopts a similar strategy. At each point, it chooses a request based on the peak request that the optimal algorithm would make when run on the demands received so far.

Contributions. We introduce a novel scheduling problem and solve several versions optimally with efficient combinatorial algorithms. We solve the offline problem for four kinds of batteries: lossless/unbounded battery in $O(n)$, lossless/bounded in $O(n^2)$, lossy/unbounded in $O(n \log n)$, and lossy/bounded in $O(n^2 \log n)$. To solve the lossy cases efficiently, we pose and solve a mathematical problem which is a *generalization* of the arithmetic mean of a set of numbers. We show that this generalized average (GA) can be computed in linear time. Using balanced BSTs, we show how to find the generalized average of *each prefix* of a sequence of n numbers in $O(n \log n)$ total time. Separately, we show how to find the optimal offline battery size, for the setting in which the final battery level must equal the initial battery level. This is the smallest battery size that achieves the optimal peak. The online problem we study is very strict. A meta-strategy in many online problems is to balance expensive periods with cheap ones, so that the overall cost stays low [6]. The difficulty in our problem lies in its non-cumulative nature: we optimize for the max, not for the average. We show that several versions of the online problem have no non-trivial competitive algorithm (with ratio better than n). Given advanced knowledge of the peak demand D, however, we give H_n-competitive algorithms for lossless batteries (bounded and unbounded). Our fastest algorithm has $O(1)$ per-slot running-time. H_n is the (optimal) competitive ratio for each.

Examples: Although there is no constant-ratio competitive algorithm for unbounded n, our intended application in fact presumes a fixed time-horizon. If the billing period is one month, and peak charges are computed as 30-minute averages, then for this setting H_n is approximately 7.84. If we assume that the battery can fully recharge at night, so that each day can be treated as a separate time period, then for a 12-hour daytime time-horizon H_n is to approximately 3.76.

2 Model and preliminaries

Definition 2.1 *The* demand curve *is the timeslot-indexed sequence of energy demands* $(d_1, ..., d_n)$*. The* request curve *is the timeslot-indexed sequence of energy requests* r_i*. Battery charge level* b_i *indicates the (non-negative) amount of energy present in the battery* at the start *of timeslot* i*.* D *is the maximum demand* $\max_i\{d_i\}$*, and* R *is the maximum request* $\max_i\{r_i\}$*.*

The demand curve (combined with battery information) is the problem instance; the request curve is the problem solution. In the absence of battery loss and overflow/underflow, the battery level at timeslot i is simply $b_i = b_{i-1} + r_{i-1} - d_{i-1}$. It is forbidden for b_i to ever fall below 0. That is, the request r_i and the battery level b_i must sum to at least the demand d_i at each timeslot i.

In the energy application, battery capacity is measured in kWh, while instantaneous request is measured in kW. By discretizing we assume wlog that battery level, demand, and request values are all expressed in the same units. Peak charges are based linearly on the max request. We optimize for the peak charge, not for total energy usage. There are several independent optional extensions, leading to many problem variants. The battery can have *maximum capacity* B or be unbounded; with some batteries, there is an automatic percentage *loss* $0 \le \ell \le 1$ in all charged energy, due to AC/DC conversion; the problem may be online, offline, or in between; we consider the setting in which the peak demand D is revealed in advance, perhaps predicted from historical information.

Threshold algorithms: For a particular snapshot (d_i, r_i, b_i), demand d_i must be supplied through a combination of the request r_i and a change in battery $b_i - b_{i-1}$. This means that are only three possible modes for each timestep: request exactly the demand, request more than the demand and *charge* the difference, or request less than the demand and *discharge* the difference. Our online algorithms and (most of) our offline algorithms are *threshold algorithms.* Let $T_1, T_2, ..., T_n$ be a sequence of values. Then the following algorithm uses these as request thresholds:

```
for each timeslot $i$
    if $d_i < T_i$
        charge $min(B - b_i, T_i - d_i)$
    else
        discharge $d_i - T_i$
```

The algorithm amounts to the rule: *at each timeslot* i*, request an amount as near to* T_i *as the battery constraints will allow.* Our offline algorithms are *constant threshold* algorithms, with a fixed T; our online algorithms compute T_i dynamically for each timeslot i. We may assume wlog that requests are monotonically increasing, since there is no incentive even to request a lower value than the max so far, except when prevented by overflow.

Definition 2.2 *Let* overflow *be the situation in which* $T_i - d_i > B - b_i$*, i.e., there is not enough room in the battery for the amount we want to charge. Let* underflow *be the situation in which*

$d_i - T_i > b_i$, *i.e., there is not enough energy in the battery for the amount we want to discharge. Call a threshold algorithm* feasible *if underflow never occurs (overflow merely wastes energy).*

Remark: A constant-threshold algorithm is specified by a single number. In the online setting, predicting the *exact* optimal threshold from historical data suffices to solve the online algorithm optimally. A small overestimate of the threshold will merely raise the peak cost correspondingly higher. Unfortunately, however, examples can be constructed in which even a small *underestimate* eventually depletes the battery before peak demand and thus produce no cost-savings at all.

It is easy to solve the *offline* problem approximately, within additive error ϵ, through binary search for the minimum feasible constant threshold value T. Simply search the range $[0, D]$ for the largest value T for which the threshold algorithm exhibits no underflow, in total time $O(n \log \frac{D}{\epsilon})$. If the optimal peak reduction is $R - T$, then the algorithm's peak reduction will be at least $R - T - \epsilon$. It is straightforward to give a linear programming formulation of the *offline* problem; it can also be solved by generalized *parametric* max-flow [4]. Our interest, however, is in efficient combinatorial optimal algorithms. Indeed, our combinatorial offline algorithms are significantly faster than these general techniques and lead naturally to our competitive online algorithms. Online algorithms based on such general techniques would be intractable for fine-grain timeslots.

2.1 Generalized average

In Section 3 we will solve the offline problem for lossless and lossy batteries, each with or without a capacity bound. The algorithms for lossy batteries will be the same as for lossless, except computations of *average* will be replaced with *generalized average* (GA).

Definition 2.3 *Given n real values $(y_1, y_2, \ldots, y_n)$ and constants $0 \le r \le 1$ and $B \ge 0$, let the* ***generalized average*** *$GA(y_1, y_2, \ldots, y_n)$ be the value a satisfying $U(a) = B + r \cdot L(a)$, where: $U(a) = \sum_{i=1}^{n} \max(y_i - a, 0)$ and $L(a) = \sum_{i=1}^{n} \max(a - y_i, 0)$. We call $U(a)$ and $L(a)$ a's L/U* values *or a's* upper *and* lower*. They correspond to the area above a and below y (upper) and the area below a and above y (lower). For any value x, let x's* neighbors *be the data points (y_i, y_j) such that $y_i \le x \le y_j$ and for no other y_k do we have $y_i \le y_k \le y_j$. For convenience we allow $-\infty$ to be the smaller neighbor if x is less than all y_i. When the values being averaged are clear from the context, we will let $GA[i, j]$ refer to $GA(y_i, ..., y_j)$. Let the rank of y_i be $r(y_i) = |\{y_j : y_j < y_i\}|$ and the reverse rank of y_i be $r'(y_i) = |\{y_j : y_j > y_i\}|$.*

Note that a need not be one of the y_i values. When $r = 1$ and $B = 0$, the generalized average is simply the mean of the values y_i; when $r = 0$ and $B = 0$, the generalized average is the maximum. We now prove two short lemmas helpful in computing GA.

Lemma 2.1 *Let y^j be a node of rank j. If the U (respectively L) value of y^j is known, then the U (respectively L) value of y^{i+1} and y^{i-1} can be found in constant time.*

Proof: By viewing the sequence as carving out rectangles of area under a step function, we recognize the following identities: $L(y^{i+1}) = L(y^i) + i(y^{i+1} - y^i)$, $U(y^{i-1}) = U(y^i) + (y^i - y^{i-1})(n - i + 1)$. $\square$

Lemma 2.2 *Let $r(y_i) = (U(y_i) - B)/L(y_i)$. Let y_j be of rank j. Assume the neighbors of the GA a are (y_i, y_j) and that ranks $r(y_i)$ and $r(y_j)$ are known. Then a can be solved for in constant time.*

Proof: Since u and a are piecewise linear with breakpoints y_i, they are linear in the interval $[y_i, y_j]$. For abbreviation, let $x_0 = y_i$, and $x_1 = y_j$. Then let $L'(x)$ and $U(x)$ be the lines defined by pairs of points the $\{(x_0, r \cdot L(x_0) + B), (x_1, r \cdot L(x_1) + B)\}$ and $\{(x_0, U(x_0)), (x_1, U(x_1))\}$, with slopes $m_L = \frac{L'(x_1) - L'(x_0)}{x_1 - x_0} = rj$ and $m_U = \frac{U(x_1) - U(x_0)}{x_1 - x_0} = -(n - j)$, respectively, i.e., $L'(x) = m_L \cdot (x - x_0) + L'(x_0)$

and $U(x) = m_U \cdot (x - x_0) + U(x_0)$. These two lines meet at the point: $a = \frac{U(x_0) - L'(x_0)}{m_L - m_U} + x_0$. □

Computing a GA in $O(n \log n)$ is not difficult. First sort the values y_i. Next, compute $L(y_1) = 0$, since y_1 is the smallest y_i. For each subsequent i up to n, $L(y_i)$ can be computed in constant time (Lemma 2.1). Similarly, compute each $U(y_i)$, starting with $U(y_n)$. Once all the U/Ls are computed, a's neighbors (y_i, y_{i+1}) can be found by inspection, and then a can be computed (Lemma 2.2). Unlike the ordinary arithmetic mean, however, computing a GA in $O(n)$ requires more effort.

Our recursive algorithm invokes the well-known linear-time deterministic Selection Algorithm [7], at two levels. The bulk of the algorithm finds a's *neighbors*. Given these data points (and their upper and lower values), we can solve (Lemma 2.2) for the correct value a in constant time. (The cases when the solution a is among the data points, and when a is less than *all* the points can be checked as special cases.) The algorithm for finding the neighboring data points to a takes the set of points y_i as input. Let $0 \le r < 1$ and B be the parameters to the GA.

At a high level, the algorithm performs the Selection Algorithm on the set of data points. In each recursive call, it choose a pivot, performs the pivoting operation, and then recurses to one side or the other. The direction of recursion is based on the relationship of $U(p)$ relative to $r \cdot L(p) + B$, i.e., whether the GA is above or below p. Internally, the pivot is chosen by running the Selection Algorithm itself on the set of data points. The correct computation of p's U/L values is subtle and uses the passed-in areas and numbers of nodes from previous recursive calls. The first parameter to the algorithm is the set of values to be averaged; all other parameters to the first (non-recursive) call are set to 0. In recursive calls, S_U is the indices of the points known to be above the GA, and S_L the indices of the points known to be below. X_U is the area contributed by S_U, i.e., $\sum_{i \in S_U} (y_i - \max\{A\})$ and W_U is the upper width, i.e., $|S_U|$. X_L and W_L are defined similarly.

```
GenAvgNbrs($A[], X_U, X_L, W_U, W_L$) :
    **if** length(A) == 2
        **return** $A$;
    **else** p = Select-Median(A);                          (a)
        $(A_L, A_U)$ = Pivot(A,p);                          (b)
        $U_p$ = Upper(A,p); $L_p$ = Lower(A,p);             (c)
        $U = U_p + X_U + W_U \cdot (max(A) - p)$; $L = L_p + X_L + W_L \cdot (p - min(A))$;
        **if** $U < r \cdot L + B$
            **return** GenAvgNbrs($A_U \cup p, L, X_U, W_L + |A_L|, W_U$);
        **else if** $U > r \cdot L + B$
            **return** GenAvgNbrs($A_L \cup p, X_L, U, W_L, W_U + |A_U|$);
        **else**
            **return** p;
```

Theorem 2.3 *$GA(y_1, ..., y_n)$ can be computed in $O(n)$ time.*

Proof: With $|A| = n$, lines a,b,c each take time $O(n)$ since *Select-Median* uses the Selection Algorithm, *Pivot* is the usual Quicksort pivoting algorithm, and Upper and Lower are computed directly. (Min and max can be passed in separately, but we omit them for simplicity.) The function makes one recursive call, whose input size is by construction half the original input size. Hence the total running time is $O(n)$. □

The bulk of the work done by our algorithms for lossy batteries is to compute the GA for a series of ranges $[i, j]$, as i stays fixed (as e.g. 1) and j increases iteratively (e.g. from 1 to n). It is straightforward to do this is in $O(n^2)$ time, by maintaining a sorted sublist of the previous elements, inserting each new y_j and computing the new GA in linear time. Unlike ordinary averages, $GA[i, j]$

and the value y_{j+1} do not together determine $GA[i, j + 1]$.[1] (The GA could also be computed separately for each region $[1, j]$.) This yields offline algorithms for the lossy unbounded and bounded settings, with running times $O(n^2)$ and $O(n^3)$. Through careful use of data structures, we obtain faster algorithms, with running times $O(n \log n)$ and $O(n^2 \log n)$, respectively.

Theorem 2.4 *The values $GA[1, j]$, as j ranges from 1 to n can be computed in $O(n \log n)$ total.*

Proof: (sketch) A balanced BST is used to store previous work so that going from $GA[i, j]$ to $GA[i, j + 1]$ is done in $O(\log n)$. Each tree node stores a y_i value plus other data (it's L/U, rank, etc.) used by GenAvgNbrs to run in $O(\log n)$. Each time a new data point y_i is inserted into the tree, its data must be computed (and the tree must be rebalanced). Unfortunately, each insertion partly corrupts *all other nodes' data*. Using a lazy evaluation strategy, we initially update only $O(\log n)$ values. After the insert, GenAvgNbrs is run on the tree's current set of data points, in $O(\log n)$ time, relying only the nodes' data guaranteed to be correct. Running on the BST, GenAvgNbrs's subroutines (Select-Median, Pivot, and selection of the subset to recurse on) now complete in $O(\log n)$, for a total of $O(n \log n)$. See Appendix A for full proof. □

```
create an empty BST $T$
for $j = 1$ to $n$
    insert $d_i$ into $T$
    update $T$ as needed
    compute the generalized density of $[1, j]$
end
```

Figure 1: GenAvgs-BST algorithm

3 Offline problem

In this section we find optimal algorithms for the four offline settings. For unbounded battery, we assume the battery starts empty; for bounded battery, we assume the battery starts with amount B. For both, we leave the final battery level unspecified. It can be shown that these assumptions are made without loss of generality. Because of its relationship to the offline algorithms, the same holds regarding initial values for the online algorithms (see Appendix B). A related problem is finding the optimal battery size for a given demand curve d_i, given that the battery starts and must end with a certain amount b (b can be seen as an amount *borrowed and repayed*.) The optimal peak request possible will be $\frac{1}{n} \sum_i^n d_i = m$, and goal is to find the smallest b that achieves peak m. This can be solved with no added complexity above the offline algorithms themselves (see Appendix B). The threshold functions (see Defs. 3.1, 3.2) for the four offline settings are shown in Table 1.

Alg.	lossy	bounded	threshold T_i	run-time
1.a	no	no	$mpd(n)$	$O(n)$
1.b	no	yes	$md(n)$	$O(n^2)$
1.c	yes	no	$mgpd(n)$	$O(n \log n)$
1.d	yes	yes	$mgd(n)$	$O(n^2 \log n)$

Table 1: Threshold functions used for offline algorithm settings.

3.1 Lossless batteries

One natural idea for solving the unbounded case is to take each high value in the demand curve and *push it back*, as flatly as possible, so that the needed energy amount is accumulated in time for the

[1] When $B = 10/\ell = .5$, $GA(5, 10, 15) = GA(3, 21, 3) = 7$, but $GA(5, 10, 15, 20) = 10.83 \neq GA(3, 21, 3, 20) = 11.33$.

request. This is easy to do in quadratic time, and will result in a request curve that is a decreasing step function. If we know the best possible peak request R_{opt}, then it is easy solve this problem in linear time by running the stack-based "Algorithm 1" (for CSI with constant capacity) from [5], with R_{opt} as the production capacity parameter. R_{opt} can be found in linear time, however, which immediately yields a simple linear-time threshold algorithm (see Section 2).

Definition 3.1 *Let $pd(j) = \frac{1}{j}\sum_{t=i}^{j} d_t$ be the density of the prefix region $[1, j]$, and let $mpd(n) = max_{1 \leq j \leq n} pd(j)$ be the maximum density among of the prefix regions up to n. Let $density(i, j) = \frac{-B+\sum_{t=i}^{j} d_t}{j-i+1}$ and $md(n) = max_{1 \leq i \leq j \leq n} density(i, j)$.*

Bounding capacity changes the character of the offline problem. It suffices, however, to find the peak request made by the optimal algorithm, R_{opt}. It is clear that $R_{opt} \geq D - B$, since the ideal case is that a width-one peak is reduced by size B. Of course, the peak region might be wider.

Theorem 3.1 *In the offline/lossless setting, Algorithm* 1.a *(threshold $T_i = mpd(n)$, for unbounded battery) and Algorithm* 1.b *(threshold $T_i = md(n)$, for bounded battery) are optimal, feasible, and run in times $O(n)$ and $O(n^2)$, respectively.*

Proof: Let the battery be unbounded. For any region $[1, j]$, the best we can hope for is that all demands $d_1, ..., d_j$ can be spread evenly over the first j timeslots. Therefore the optimal threshold cannot be lower than $pd(j)$. For feasibility, it suffices to show that after each time j, the battery level is non-negative. But by time j, the total output from the system will be exactly the total input: $\sum_{t=1}^{j} d_j$. For complexity, just note that $pd(j + 1) = \frac{j \cdot pd(j)+d_{j+1}}{j+1}$.

Now let the battery be bounded. Although for this setting there are simpler proofs, optimality and feasibility will be corollaries of the corresponding results for the unbounded case. The densest region can be found in $O(n^2)$, using our previous results for timeslots $[1, t]$ to find a densest subsequence in timeslots $[1, t+1]$ This is because the densest region in within $[1, t+1]$ will either include timeslot $t + 1$ or not: $T_{n+1} = max(T_n, max_{i \leq n+1}\{density(i, n + 1)\})$. □

3.2 Lossy batteries

Definition 3.2 *Let $gd(i, j)$ be the generalized average of the demands over region $[i, j]$ and $gpd(n) = gd(1, n)$. Let $mgd(1, n) = \max_{1 \leq i \leq j \leq n} gd(i, j)$ and $mgpd(n) = \max_{j=1}^{n} gpd(j)$.*

Theorem 3.2 *For the offline/lossy setting, Algorithm* 1.c *($T_i = mgpd(n)$, for unbounded battery) and Algorithm* 1.d *($T_i = mgd(n)$, for unbounded battery) are optimal, feasible, and run in times $O(n \log n)$ and $O(n^2 \log n)$, respectively.*

Proof: Let the battery be unbounded. Within any region $[1, j]$, the battery may be able to lower the local peak by sometimes charging and sometimes discharging. With battery loss percentage ℓ, the total discharged from the battery can be at most $(1 - \ell)$ times the total charged. The optimal threshold over this region cannot be less than $GA(d_1, ..., d_j)$ with $r = 1 - \ell$ and $B = 0$.

The threshold is $T_{opt} = mgpd(n)$. It suffices to show that the battery will be nonnegative after each time j. But after time j, the total charged in region $[1, j]$ is exactly $U(T) = \sum_{t=1}^{j} \max(T - d_t, 0)$ and the total discharged will be $L(T) = \sum_{t=1}^{j} \max(d_t - T, 0)$. The amount of energy available for discharge over the entire period is $r \cdot L(T)$. Overflow at time j means $U(T) > r \cdot L(T)$, but this contradicts the definition of T. The running time is simply the time to compute each $GA[1, j]$ iteratively, using Algorithm 2.1.

Now let the battery be bounded. Within any region $[i, j]$, the battery may help us in two ways. First, the battery may be able to lower the local peak by sometimes charging and sometimes discharging. Second, the battery in the best case would start with charge B at timestep i. With battery loss percentage l, the total amount discharged from the battery over this period can be at most B plus $(1-\ell)$ times the total amount charged. The optimal threshold over this region cannot be less than $GA(d_i, ..., d_j)$ with $r = 1-\ell$ and $B = B$.

The threshold used is $T = mgd(1, n)$. It suffices to show that the battery will be nonnegative after each time j. Suppose j is the first time underflow occurs. Let $i-1$ be the last timestep prior j with a full battery (or 0 if this has never occurred). Then there is no underflow *or* overflow in $[i, j)$, so the total charged in region $[i, j]$ is exactly $U(T) = \sum_{t=i}^{j} \max(T - d_t, 0)$ and the total discharged will be $L(T) = \sum_{t=i}^{j} \max(d_t - T, 0)$. The amount of energy available for discharge over the entire period is $B + r \cdot L(T)$. Overflow at time j means $U(T) > B + r \cdot L(T)$, but this contradicts the definition of T. To compute the thresholds, compute $GA[i, j]$ iteratively for each value i, using Algorithm 2.1. Each call takes $O(n \log n)$. □

4 Online problem

We consider two natural choices of objective function for the online problem. One option is to compare the peak requests, so that if ALG is the peak request of the online algorithm and OPT is that of the optimal offline algorithm, then a c-competitive algorithm for $c \geq 1$ must satisfy $c \geq \frac{ALG}{OPT}$ for every demand sequence. Although this may be the most natural candidate, it is uninteresting for many settings. If the peak demand is a factor k larger than the battery capacity, for example, then the trivial online algorithm that *does not use* the battery would be $(1+1/(k-1))$-competitive. Without the assumption that B is small compared to D, however, no competitive ratio better than n is possible, *even if D is revealed in advance*. For suppose that $B = D$ and the demand curve is $(B, 0, ..., 0, ?)$, where the last demand is either B or 0. ALG must discharge B at time 1, since $OPT = 0$ when $d_n = 0$. Thus ALG's battery is empty at time 2. If ALG requests nothing between times 2 and $n-1$ and $d_n = B$, we have $OPT = B/n$ and $ALG = n$; if ALG requests some $a > 0$ during any of those timeslots and $d_n = 0$, we have $OPT = 0$ and $ALG = a$.

Instead, we compare the *peak shaving amount*, i.e., $D - R$. For a given input, let OPT be peak savings of the optimal algorithm, and let ALG be the peak savings of the online algorithm. Then an online algorithm is c-competitive for $c \geq 1$ if $c \geq \frac{OPT}{ALG}$ for every problem instance. For this setting, we obtain the online algorithms described below.

Definition 4.1 *Let T_i^{opt} be the T_i used by the appropriate optimal algorithm. At time i during the online computation, let s_i be one greater than the timeslot of the most recent overflow prior to timeslot i, or to 1 if no overflow has yet occurred.*

Alg.	lossy	bounded	threshold T_i	per-slot time
2.a	no	both	$D - \frac{D-T_i^{opt}}{H_n}$	$O(n)$
2.b	no	both	$D - \frac{D-density(s_i,i)}{H_{n-s_{i+1}}}$	$O(1)$

Table 2: Threshold functions used for online algorithms.

4.1 Lower bounds

It can be shown (under light assumptions) that the competitiveness of our algorithms holds for initial battery levels other than the base cases, by slightly modifying the proof of Theorem 4.5 below, and by invoking the appropriately modified offline algorithm or density calculation (see Appendix B). For lower-bound arguments, therefore, we assume particular initial charges wlog.

Proposition 4.1 *With peak demand D unknown and finite time horizon n, there is no online algorithm 1) with constant competitive ratio for unbounded battery or 2) with competitive ratio* *better than n for bounded battery (even with $n = 2$).*

Proof: For part 1), assume $b_0 = 0$, and suppose $d_1 = 0$. Then if ALG requests $r_1 = 0$ and we have $d_2 = D$, then $OPT = D/2$ and $ALG = 0$; if ALG requests $r_1 = a$ and we have $d_2 = a$, then $OPT = a/2$ and $ALG = 0$. For part 2), let $b_0 = B$, and assume ALG is c-competitive. Consider the demand curve $(B, 0, 0, \ldots, 0)$. Then OPT clearly discharges B at time 1 (decreasing the peak by B). For ALG to be c-competitive, it must discharge at least $\frac{B}{c}$ in the first slot. Now consider curve $(B, 2B, 0, 0, \ldots, 0)$. At time 2, OPT discharges B, decreasing the peak by B. At time 2, ALG must discharge at least $\frac{B}{c}$. (At time 1, ALG already had to discharge $\frac{B}{c}$.) Similarly, at time i for $(B, 2B, 3B, ..., iB, ..., 0, \ldots, 0)$, ALG must discharge $\frac{B}{c}$. Total discharging by ALG is then at least: $\Sigma_{i=1}^{n} \frac{B}{c} = \frac{nB}{c}$. Since we must have $\frac{B}{c} n \leq B$, it follows that $c \geq n$. □

Remark: The trivial algorithm that discharges amount B/n at each of the n timesteps and *never charges* is n-competitive (since $OPT \leq B$) and so matches the lower bound.

Proposition 4.2 *With peak demand D known in advance and finite time horizon n, no online algorithm can have competitive ratio better than H_n, with battery either 1) bounded or 2) unbounded.*

Proof: For part 1), suppose ALG is c-competitive. Consider the curve $(D, 0, 0, \ldots, 0)$, with $D > B$. Then OPT clearly discharges B at time 1 (decreasing the peak by B). For ALG to be c-competitive, it must discharge at least $\frac{B}{c}$. Now consider curve $(D, D, 0, 0, \ldots, 0)$. At times 1 and 2, OPT discharges $\frac{B}{2}$, decreasing the peak by $\frac{B}{2}$. At time 2, ALG will have to discharge at least $\frac{B/2}{c} = \frac{B}{2c}$. Similarly, at time i for $(D, D, D, , ..., iD, ..., 0, 0, \ldots, 0)$, ALG must discharge $\frac{B}{ic}$. Total discharging by ALG is then at least: $\Sigma_{i=1}^{n} \frac{B}{ic} = \frac{B}{cH_n}$. Since we must have $\frac{B}{c} Hn \leq B$, it follows that $c \geq H_n$. For part 2), assume $b_0 > 0$. Then we can apply the proof of part 1), plugging in b_0 for B. Although the battery is unbounded, ALG is forced never to discharge, and so we obtain the same result. □

4.2 Bounded/lossless battery

Our first algorithm bases its threshold at time i on a computation of the optimal offline threshold T_i^{opt} for the demands $d_1, ..., d_i$.

Theorem 4.3 *Algorithm* 2.a *is H_n-competitive, if it is feasible, and is $O(n)$* per timeslot*.*

Proof: Since T_i^{opt} is the lowest possible threshold up to time i, $D - T_i^{opt}$ is the highest possible peak savings as of time i. Since the algorithm always saves a $1/H_n$ fraction of this, it is H_n-competitive by construction. Recall from the proof for offline setting b) that the optimal offline thresholds T_i^{opt} for each prefix $[1, i]$ can be computed in $O(n^2)$. Thus a is linear per-slot. □

Next we consider a faster algorithm which is less intuitive but still H_n-competitive.

Corollary 4.4 *Algorithm* 2.b *is H_n-competitive,* if it is feasible*, and is $O(1)$* per timeslot*.*

Proof: For competitiveness, simply notice that $\frac{D - density(s_i, i)}{H_{n-s_i+1}} \geq \frac{D - T_i^{opt}}{H_n}$. For complexity, recall that offline algorithm a takes $O(n)$. The work to compute $density(s_i, i)$ is the same, except that whenever overflow occurs we start over, rather than extending the previous $density(s_{i-1}, i-1)$. □

Theorem 4.5 *Algorithm* 2.a *is feasible.*

Proof: Assume for contradiction that underflow occurs. Let t be the first time it does, and let the last overflow before t the underflow occur at timeslot $s - 1$. We now show that it is impossible for the battery to underflow in time range $[s, t]$. Since any underflow that occurs can be extended to the end of sequence by replacing $d_t, d_{t+1}, ..., d_n$ with $D, D, \ldots, D$ and since $T_i \geq D - \frac{1}{H_n}(D - (\sum_{k=s}^{i} d_k - B)/(i - s + 1))$ in both algorithms, we can assume wlog that $s = 1$ and $t = n$. We now show that it is impossible for the battery to fall below 0 at time t, by upperbounding the *net discharge* over this region. Let $\Delta b_i = b_i - b_{i+1} = d_i - T_i$ be the amount the battery discharges at step i. (Δb_i will be negative when the battery charges.) We will show that $\sum_{1 \le i \le n} \Delta b_i \le B$. Expanding the definition of our algorithm's threshold, we have:

$$\Delta b_i = d_i - T_i = d_i - \frac{1}{H_n}(D - density(1,i)) = d_i - \Big(D - \frac{1}{H_n}\Big(D - \frac{1}{i}\Big(\sum_{k=1}^{i} d_k - B\Big)\Big)\Big) \qquad (1)$$

By summing Eq. 1 for each i, we obtain:

$$\sum_{i=1}^{n} \Delta b_i \le \sum_{i=1}^{n} \Big(d_i - \Big(D - \frac{D - (\sum_{k=1}^{i} d_k - B)/i}{H_n}\Big)\Big) \qquad (2)$$

It now suffices to prove that the RHS of Eq. 2 is at most B. After some algebra, we can rewrite the desired inequality the following two ways:

$$\sum_{i=1}^{n} \Big(d_i - (D - \frac{D - \sum_{k=1}^{i} d_k/i}{H_n})\Big) \le 0 \iff \sum_{i=1}^{n} H_n d_i - \sum_{i=1}^{n}\sum_{k=1}^{i} \frac{d_k}{i} \le n(H_n - 1)D \qquad (3)$$

Because of the following derivation:

$$\sum_{i=1}^{n}\sum_{k=1}^{i} \frac{d_k}{i} = \sum_{i=1}^{n} \frac{1}{i}\Big(\sum_{k=1}^{n} d_k - \sum_{k=i+1}^{n} d_k\Big) = \sum_{i=1}^{n}\sum_{k=1}^{n} \frac{d_k}{i} - \sum_{i=1}^{n}\sum_{k=i+1}^{n} \frac{d_k}{i}$$

$$= \sum_{k=1}^{n}\sum_{i=1}^{n} \frac{d_k}{i} - \sum_{k=1}^{n}\sum_{i=1}^{k-1} \frac{d_k}{i} = \sum_{k=1}^{n} H_n d_k - \sum_{k=1}^{n} H_{k-1} d_k$$

we can rewrite Eq. 3.b as: $\sum_{i=1}^{n} H_{i-1} d_i \le n(H_n - 1)D$. Since $d_i \le D$, it suffices to show that $\sum_{i=1}^{n} H_{i-1} \le n(H_n - 1)$. In fact, this holds with equality (see [10], Eq. (2.36)). □

Replacing the threshold for algorithm a with that of b in Eq. 1, we also obtain:

Corollary 4.6 *Algorithm* 2.b *is feasible.*

4.3 Unbounded/lossless battery

Both online algorithms also work for the unbounded battery setting. The algorithms are feasible in this setting since $density(1,i) \ge T_i^{opt}$ still holds, where T_i^{opt} is now the optimal threshold for the unbounded battery setting. (Recall that offline algorithm a can "greedily" run in linear total time.) The algorithm is H_n-competitive by construction, as before.

Corollary 4.7 *Algorithms* 2.a *and* 2.b *are feasible in the unbounded battery setting.*

Proof: The proof is similar to that of Theorem 4.5, except that b_0 (which may be 0) is plugged in for B, and overflow is no longer a concern. □

5 Conclusion

In this paper, we formulated a novel peak-shaving problem, gave efficient optimal offline algorithms and optimally competitive online algorithms. In work in progress, we are testing our online algorithms on actual client data from Gaia [1]. There are several interesting extensions to the theoretical problem and open questions that we plan to address, such as adapting the online algorithm to lossy batteries. We note that in that setting, the online algorithm corresponding to Algorithm *2.b* would benefit from the efficient generalized average computation of Section 2.1 and run in only $O(\log n)$ per timeslot rather than $O(n)$.

Acknowledgements: We thank Ib Olson of Gaia Power Technologies and Ted Brown of CUNY and the Institute for Software Development and Design for posing this problem to us. Thanks also to Deniz Sarioz for useful discussions and for suggesting the $O(n \log n)$ single-GA computation.

References

[1] Gaia Power Technologies. gaiapowertech.com.

[2] ConEd electricity rates document. www.coned.com/documents/elec/043-059h.pdf.

[3] Orlando Utilities Commission website. www.ouc.com/account/rates/electric-comm.htm.

[4] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows.* Prentice Hall, 1993.

[5] A. Atamturk and D.S. Hochbaum. Capacity acquisition, subcontracting, and lot sizing. *Management Science*, Vol. 47, No. 8, 2001.

[6] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[8] M. Florian, J.K. Lenstra, and A.H.G. Rinnooy Kan. Deterministic production planning: algorithms and complexity. *Management Science*, Vol. 26, 1980.

[9] M. Goh, O. Jihong, and T. Chung-Piaw. Warehouse sizing to minimize inventory and storage costs. *Naval Research Logistics*, Vol. 48, Issue 4, 3 Apr 2001.

[10] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science, 2nd Edition.* Addison-Wesley Professional, 1994.

[11] T. Harford. The great Xbox shortage of 2005. *Slate*, page www.slate.com/id/2132071/, Dec. 15, 2005.

[12] B. Hunsaker, A.J. Kleywegt, M.W. P. Savelsbergh, and C.A. Tovey. Optimal online algorithms for minimax resource scheduling. *SIAM J. Discrete Math.*, 2003.

[13] M.-K. Lee and E.A. Elsayed. Optimization of warehouse storage capacity under a dedicated storage policy. *Int J Prod Res*, Vol. 43, No. 9, 2005.

[14] Retsef Levi, Robin Roundy, and David B. Shmoys. Provably near-optimal sampling-based algorithms for stochastic inventory control models. *STOC 2006.*

[15] Y. Pochet and L.A. Wolsey. *Production Planning by Mixed Integer Programming.* Springer, 2006.

[16] H. Scarf. A min-max solution to an inventory problem. In *K.J. Arrow, S. Karlin, and H. Scarf, editors, Studies in the mathematical theory of inventory and production*, pages 201–209. Stanford University Press, 1958.

[17] H.M. Wagner and T.M. Whitin. Dynamic version of the economic lot size model. *Management Science*, Vol. 5, 1958.

[18] Y.-W. Zhou. A multi-warehouse inventory model for items with time-varying demand and shortages. *Computers and Operations Research*, Vol. 30, Issue 14, December 2003.

A Lemmas and proof for Theorem 2.4

Theorem 2.4: Values $GA[1, j]$, as j ranges can 1 to n can be computed in $O(n \log n)$ total time.

Proof: We compute the GA for each region $[1, j + 1]$ by using the work retained from the computation done for the previous region $[1, j]$. This is done by storing the data points y, along with supplementary information, in a balanced binary search tree (such as an AVL tree or Red-Black tree [7]). Each time a new data point y is inserted into the tree, GenAvgNbrs is run, on the tree's current set of data points, in $O(\log n)$ time. This speed-up is made possible by the balanced tree and *lazy evaluation*. In addition to its key value y_i, a node will contain its own L/U values; the min and max values of the subtree induced by that node; the rank, reverse-rank and L/U values of its parent. We assume for simplicity that all values are unique. Duplicates could be handled by adding a counter value to each node. We prove four essential properties with separate lemmas.

First, we can ensure that if the root's L/U values are correct, we can correct the L/U values of the nodes lying on any path from the root, as we traverse it. This is possible because given the correct L/U values of a node, we can correct the L/U values of its child in constant time (Lemma A.1). Note that for a tree with a single node, its L/U values are simply 0. Second, after we insert a node y, we can compute its L/U values in constant time (Lemma A.2) and (third) correct all the corrupted L/U values for the nodes lying on the path from y to the root, in $O(height)$ time (Lemma A.3). Finally, the necessary AVL rotations can be done without disturbing the supplementary information (Lemma A.4), so we can assume the tree height is logarithmic.

The work for inserting a single node y will thus consist of a) correcting all the L/Us on the nodes lying on the path to *y's future location*, b) doing the insert, c) correcting all the L/U values on the nodes lying on the path from y to the root (which may be corrupted by y's insertion), and d) performing AVL rotations.

Each call to GenAvgNbrs is now essentially a single search of the BST. The two neighbors will be the largest value we recurse right on and the smallest value we recurse left on. Given the neighbors and their L/Us, the solution can be found in constant time (Lemma 2.2). Of course, it is possible that we find the solution among the y values themselves. The final possibility is that the solution is less than *all* the y values. This is also easy to check for: maintain the sum of all y values in the tree. If B is greater than this sum, then the peak request is 0.

We now discuss efficiency. Based on the root's L/U values, the algorithm recurses on one subtree or the other, correcting the L/U values of that subtree's root *just in time* (Lemma A.1). In the original (array) implementation of GenAvgNbrs, the (linear) time-consuming tasks of a single recursive step are choosing the pivot (line 4), doing the pivot operation (line 5), and computing the pivot's L/U values (line 7). (The min and max can be passed in recursively, but we omit these parameters for simplicity.) With the BST, these steps are all performed in constant time. □

Lemma A.1 *Assuming that a node p's L/Us are correct, we can update the L/Us of either of p's children. Therefore the correct L/U values can be computed for all the nodes lying on a path from the root, in $O(height)$ time.*

Proof: Since each node stores its own mirror copies of its parent's rank, reverse rank, and L/U values, we can update the node's L/U values based on the change in its parent's information. To see how, suppose that a node y's L/U values have become incorrect. This happens when a new node has been inserted into the tree, without y's knowledge. Suppose, for example, that y's U value is wrong. This must have been caused by the insertion of at least one node z larger than y, i.e., $y < z$. Now there are several possibilities for the value of p. First, notice that z may not lie between y and p, since this would imply that z was inserted as a descendent of y. In either of these two cases, y's L/U values would have been correctly updated during the insert (see Lemma A.3).

Given this and that $y < z$, suppose $y < p < z$. Then if z is to blame for changing p's U, $U(p)$ will have increased by $z - p$ and $U(y)$ should increase by $z - p + p - y$. It may be more complicated than this, however, since *multiple* new values $z_1, ..., z_k$ may have caused the change δ in $U(p)$. The correction can still be done, though, since $U(y)$ should increase by $\delta + k \cdot (p - y)$. The number k of new problem nodes can be found by comparing p's current reverse-rank to y's stored copy of p's reverse-rank. (We can always determine the rank and reverse-rank of a node during the process of traversing its path from the root, since each node stores the number of nodes in its induced subtree.) Finally, it could be that $p < y < z$. If k new z nodes are the cause, then $U(y)$ should increase by $\delta - k \cdot (p - y)$. Updating a node's L value is similar, using the parent's L and rank. □

Lemma A.2 *We can correctly compute the L/Us of a newly inserted node y in constant time.*

Proof: The (current) L/U values of a node y can be computed in constant time given the L/Us of its (current) predecessor $pr(y)$ and successor $s(y)$, and the rank $r(y)$ of y within the set: $L(s(y)) = L(y) + (s(y) - y)r(s(y))$ and $U(pr(y))) = U(y) + (y - pr(y))(n - r(y) + 1)$. Since we know the node-count of each induced subtree, we can calculate the rank of y as we traverse a path down the tree in order to insert it. Note also that y's (current) predecessor and successor nodes (if they exist) will lie on y's path to the root, so these can be found during the insertion process. Therefore y's L/U values can be computed as it is inserted. □

Lemma A.3 *After each insertion of a node y, we can ensure in $O(height)$ time that all the L/Us on y's path to the root are correct.*

Proof: The insertion of a node y will initially corrupt (some of) the L/U values of the nodes on the path to y (as well as others'). For each ancestor a (up to the root) of y, $L(a)$ is updated by incrementing it by $a - y$ if $a > y$, and $U(a)$ is updated by incrementing it by $y - a$ if $a < y$. The opposite L/Us on the path will not be affected. (Note that the L/U values only increase.) The parent L/U values of the nodes on this path, as well as the subtree node-counts, are updated simultaneously. □

Lemma A.4 *The necessary AVL rotations can be done without disturbing the supplementary information.*

Proof: An AVL rotation will only occur for us in response to an insert. When a single rotation occurs (repeat this argument for double rotations), it will necessarily be performed on one of the nodes lying on the path of the newly inserted node y. After the insert but *before the rotation*, all these nodes' L/U values will be updated. When any rotation occurs, it will involve one node c on y's path taking the place of its parent p. Suppose this a left rotation. Then consider the left child g of c, which shifts to become the right child of p. g's stored parent information will no longer be correct because *it has changed parents*. If we update g's parent information, then we are obliged to update g's L/U values. This is possible in the usual way, using g's old parent c. Therefore AVL rotations will not be harmful, as long as we correct the L/U values of any shifting child node before performing the rotation. Finally, the affected min/max and node-count values can easily be updated after the rotation. □

B Initial charge and final charge

In the unbounded/lossless battery case of the offline problem, the algorithm assumes the battery starts empty, and it leaves the battery in an indeterminate state when it completes. The algorithm for the bounded battery case assumes the battery starts full to capacity B, and it leaves the battery at an indeterminate level of charge when it completes.

A more constrained version of the problem will require that $b_0 = \beta_0$ and $b_n = \beta_n$, i.e., the battery begins and ends at some charge levels specified by parameters β_0 and β_n. We argue here that these requirements are not significant algorithmically, since by pre- and postprocessing, we can reduce to the default cases for both the unbounded and bounded versions.

First, consider the unbounded setting. In order to enforce that $b_0 = \beta_0$ and $b_n = \beta_n$, run the usual algorithm on the sequence $(d_1 - \beta_0, d_2, ..., d_{n-1}, d_n + \beta_n)$. Then b_n will be at least β_n larger than d_n. To correct for the surplus, manually delete a total of $b_n - \beta_n$ from the final requests. For the bounded setting, the default case is $b_0 = B$ and b_n indeterminate. To support $b_0 = \beta_0 \neq B$ and $b_n = \beta_n$, modify the demand sequence as above, except with $d_1 - (B - \beta_0)$ as the first demand and then do postprocessing for any surplus. The lossy cases are similar, but we omit the details.

Finally, we can allow specified $b_0 = \beta_0$ (but not $b_n = \beta_n$) for the online cases in essentially the same way as in the offline cases, on the assumption that modifying d_1 does not change the revealed peak. The online algorithms call the offline algorithm (or a similar density function) as a subroutine, which can be modified as above to support specified b_0.

Optimal battery size. We close this appendix by considering the optimal battery size for a given demand curve d_i. The setting we consider is that the battery will start and end with a certain amount b, so that b can be seen as an amount *borrowed and repayed.* Without receiving the initial battery charge for free, the optimal peak request possible will be $\frac{1}{n}\sum_i^n d_i = m$. Our question is then, what is the smallest initial/final charge b that achieves peak m?

First, assume unbounded/lossless battery. Then just do the following: given the demand curve with mean demand m, run the offline algorithm to find the optimal threshold T for $b_0 = 0$. Given this, it is easy to find an optimal request curve in the form of a decreasing step function. Once we know the max-prefix-mean m, simply divide nm by T to get a three-step function: $T, T, ..., T, t, 0, ..., 0$. ($t = nm \mod T$.) Now draw the step-function request curve and the line $Y = m$. Then b_0 and b_n are just $U(m)$ and $L(m)$ (which are of course equal): the area below $Y = m$ and above the step function, and vice versa. The lossy/unbounded case is essentially the same, except that the optimal step-function curve is computed with the lossy optimal algorithm, and the value m is the generalized average of the step-function curve.

Second, assume a bounded/lossless battery. Then we really want two things, the best $b_0(= b_n)$ and bound B that together yield a flat curve. By initially setting $B = \infty$, we can find the best b_0. Once we fix b_0, we can set B to be just the maximum b_i that results when we perform the offline algorithm runs. This yields the best B and b_0, though it may happen that $B > b_0$. Is it possible to shrink B (in exchange for a larger b_0)? Apparently not: with b_0 held fixed, shrinking B would at some point make the battery overflow earlier and eventually raise a (now non-constant) request. Increasing b_0 would *exacerbate* this problem; decreasing b_0 is impossible.

The lossy/bounded case is similar.