

IMMC 2025 Winter - When Watts Meet Bits

TEAM ID: 25961008

January 2025

§1 Summary

The utmost priority of a data center is to complete its requests on time. Yet, a limited budget, sudden decreases in energy supply and spikes in demand can derail the operation of a data center.

In the first task, a short-run optimization model is introduced. There is no setting up before a day, and every bit energy is purchased as the day goes on. Through defining deadlines and a priority index, the concept of urgency is concretely brought to life. Using a penalty function, the problem is reformulated into a graph problem.

However, future data is unknown to data centers in the real world. In our model, using existing demand, supply and price data, accurate curves are fitted with Fourier series, and used to predict future data points on the fly with a geometric mean. The data center is therefore able to dynamically adjust its scheduling. Combining the above, an algorithm is designed to produce semi-optimal solutions to minimize penalty using a heuristic ordering.

In the second task, to handle the additional factors of green energy proportion and cost optimization, we employ a greedy algorithm to calculate the minimum cost to attain a certain proportion of green energy. The process involves introducing an energy reserve and predicting price based on demand and supply. We generate random data using Poisson distributions and add random noise, followed by running Monte Carlo Simulations to compute the energy reserve needed and the cost over an extended duration of 1000 days.

At last, cost optimization in the general case is considered. A simplified algorithm is derived from the second task, without the need to differentiate between green and traditional energy. Using the same random set of data generated in the second task, we again obtain results regarding the minimum reserve needed to attain a minimum cost, while analyzing the effects different computational capacities have on the final penalty.

§2 Introduction

§2.1 Background

The operation of data centers is crucial for the seamless functioning of the modern digital age. With the increasing demand for computational power, servers face the challenge of balancing performance requirements with cost efficiency and environmental sustainability. In response to these challenges, data centers are integrating green energy sources along with traditional electricity to fulfill corporate social responsibility.

The coexistence of traditional and green electricity sources introduces more uncertainties and complexities in power management. Green energy sources such as solar power are intermittent in nature, leading to fluctuations in their availability. This poses a challenge for data centers in ensuring a stable power supply and reserve to meet dynamic computational demands with different degrees of urgency.

Computational tasks vary in terms of priority, resource requirements, and time sensitivity. Critical and high-demand tasks require immediate response to maintain service levels and consistency. While less urgent tasks can be flexibly scheduled based on the availability of resources.

This paper leverages mathematical modeling techniques to maximize performance, minimize energy costs, while upholding sustainable principles.

§3 Task 1: Scheduling Electricity To Minimize Penalty

§3.1 Definitions, Variables and Assumptions

Assumption 1 — Tasks are given at the start of each hour. Supply and price will not vary between the start and end of completion.

Justification: Even small data centers consume 1MW (1000 kW) of power and more. [1] Most tasks of reasonable scale (80 kWh for High Urgency Tasks) in the real world can be completed in minutes if resources are allocated to it. There is little impact of when a task is started within an hour.

Assumption 2 — Computational capacity is constant over various days.

Definition 1 (Deadlines)

Let the hour at which a task is defined be h , representing the hour-long interval $[h:00, (h + 1):00]$.

High Urgency Tasks are to be completed before $(h + 1):00$ (within the same hour).

Medium Urgency Tasks are to be completed before $(h + 2):00$ (within 2 hours).

Low Urgency Tasks are to be completed before $(h + 3):00$ (within 3 hours).

Therefore deadline DL_i of the i -th task $\in h + 1, h + 2, h + 3$ for High, Medium and Low Urgency Tasks respectively.

The time limit of a task is $lim_i \in 1, 2, 3$ for High, Medium and Low Urgency tasks respectively.

Justification:

1. High Urgency Tasks include real-time tasks, which undoubtedly should be processed immediately and will be done within the hour.
2. Due to technological advancements and the speed of computation, modern data centers are expected to have quick response times. To maximize consumer satisfaction, reduce "debt" (the postponing of tasks to future days), while ensuring that urgency of tasks are differentiated between, we prefer to use the 3 smallest positive integers, 1, 2 and 3.

Definition 2 (Priority Index)

Following the previous definition, if the current hour is h , and the deadline of a task is DL_i , the **priority index** P_i of a task is $h - DL_i$. The higher the value of the **priority index** (that is, as the deadline gets closer), the higher the priority of the task.

Note the difference between **priority** and **urgency**. Urgency is defined in the problem statement.

Important variables used in future definitions will not be written in this table, and introduced with the model's logical development.

Variable	Description
$C_{H,h}$	Number of High Urgency Tasks requested in the h -th hour.
$C_{M,h}$	Number of Medium Urgency Tasks requested in the h -th hour.
$C_{L,h}$	Number of Low Urgency Tasks requested in the h -th hour.
$D_{H,h}$	Demand for energy (in kWh) contributed by High Urgency Tasks requested in the h -th hour.
$D_{M,h}$	Demand for energy (in kWh) contributed by Medium Urgency Tasks requested in the h -th hour.
$D_{L,h}$	Demand for energy (in kWh) contributed by Low Urgency Tasks requested in the h -th hour.
D_h	Total demand for energy (in kWh) contributed by tasks requested in the h -th hour.
cap	Computational capacity, which is the maximum amount of energy (in kWh) that can be used within an hour.
T_i	The i -th task
DL_i	Deadline of the i -th task
P_i	Priority of the i -th task (at a certain hour which will be made clear in the context)
Q_i	Energy requirement of the i -th task
$time_i$	Time limit of the i -th task

We have the following relations from the Problem Statement, Appendix, Table 1:

$$Q_i = \begin{cases} 80 & T_i \text{ is High Urgency} \\ 50 & T_i \text{ is Medium Urgency} \\ 30 & T_i \text{ is Low Urgency} \end{cases}$$

$$D_{H,h} = 80 \times C_{H,h}$$

$$D_{M,h} = 50 \times C_{M,h}$$

$$D_{L,h} = 30 \times C_{L,h}$$

§3.2 Objective

Task 1 focuses on a 24-hour period. We assume the data center is working in the short run, without the time or need to prepare for its production scale beforehand (that is, all costs are variable). Therefore, some tasks may be uncompleted due to limited computational

capacity and energy reserve.

The different costs of various units of energy, which may be due to different time of day during its purchase or the type of energy, will be considered in later sections.

Definition 3 (Late Value)

Let fin_i be the hour h at which T_i is done. We can calculate the amount of hours which it is past the deadline with

$$late_i = \max(0, fin_i - DL_i + 1)$$

Assumption 3 — In Task 1, as the problem requires an analysis of a 24 hour period, **each day is independent** - there are no unfinished tasks from the day prior and no unused leftover energy. The demand and supply schedules will be such that there is no $fin_i > 23$, i.e. there is no task which completion is past the end of the day.

Justification: For ease of modelling, reduction of parameters and less ambiguity in the mathematical formulation of the problem, and by the requirements of the problem statement.

Definition 4 (Weight, Penalty Function)

We define the weight w_i of T_i as

$$w_i = \begin{cases} 4 & T_i \text{ is High Urgency} \\ 2 & T_i \text{ is Medium Urgency} \\ 1 & T_i \text{ is Low Urgency} \end{cases}$$

The penalty pen is defined as

$$\sum_i w_i \times \left(\frac{late_i + lim_i}{lim_i} \right)^2$$

Obviously, the lower the penalty, the better.

Justification:

1. It is safe to assume that more urgent tasks are more important. Therefore, the cost of delaying its completion past its time limit is higher, hence we impose a higher penalty for these tasks using a weighted sum.
2. Postponing tasks for too long due to computational restrictions is undesirable. For example, given 2 overdue tasks, 1 and 100 hours behind schedule respectively, it would be of a data center's priority to handle the latter, to ensure fairness (first come first serve), societal benefit and productivity (not to hold up other projects for too long), and maximize consumer satisfaction. Therefore, instead of using a linear measure, we square the ratio of time which a task is overdue for, so tasks that are long overdue will be more heavily penalized as it is quadratically scaled.

3. Consider a human worker instead of a data center, completing two different tasks which have $late_i = 1$. A high urgency task should be completed in 1 hour, but it has taken 2. A low urgency task should be completed in 3 hours, but it has taken 4. Informally speaking, the degree of incompetence of the worker is of a much larger magnitude in the former case, as $2/1 = 2$ while $4/3 = 1.33$ (the worker has taken twice as long in the former case). Therefore, it is natural and more accurate for us to consider the relative proportion of $late_i$ to $time_i$, instead of the absolute value of $late_i$.

§3.3 Mathematical Reformulation

We are now equipped to reformulate the problem:

Input: given the demand schedule $D_{H,0}, \dots, D_{H,23}, D_{M,0}, \dots, D_{M,23}, D_{L,0}, \dots, D_{L,23}$, the supply schedule S_0, \dots, S_{23} , where S_i is the increase in energy of the data center in the i -th hour.

States and Transitions: If at the end of the $(i-1)$ -th hour, the state is (E, H, M, L) - E representing the amount of leftover, unused energy (in kWh); H , M and L , representing the total energy demand (in kWh) contributed by unprocessed High, Medium and Low Urgency Tasks respectively. An integer tuple (h_i, m_i, l_i) may be chosen, representing the number of High, Medium and Low Urgency Tasks completed during the hour, satisfying the following constraints:

$$\left\{ \begin{array}{l} 0 \leq h_i \times 80 \leq D_{H,i} + H \\ 0 \leq m_i \times 50 \leq D_{M,i} + M \\ 0 \leq l_i \times 30 \leq D_{L,i} + L \\ 0 \leq h_i \times 80 + m_i \times 50 + l_i \times 30 \leq \min(E + S_i, cap) \end{array} \right.$$

The new state, at the end of the i -th hour, will be $(E + S_i - h_i \times 80 + m_i \times 50 + l_i \times 30, H + D_{H,i} - h_i \times 80, M + D_{M,i} - m_i \times 50, L + D_{L,i} - l_i)$.

Output: The minimum penalty pen , for all possible selections of the 24 tuples (h_i, m_i, l_i) for $0 \leq i \leq 23$.

Note: We may also model each state as a node and each transition as an edge in a directed acyclic graph, but it is so large that it is impossible to visualize.

Our model is robust in the sense that it can adapt to various supply schedules. For a general algorithm on when and how much energy to purchase as the data centre to minimize costs, see the sections on Tasks 2 and 3.

§3.4 Estimating Demand and Supply

§3.4.1 Demand (base model)

The demand for server service has a high degree of seasonality [3], which means that the demand schedules for each day are similar to each other. To relate that to real-life context, we can expect a higher server demand during working hours than at midnight.

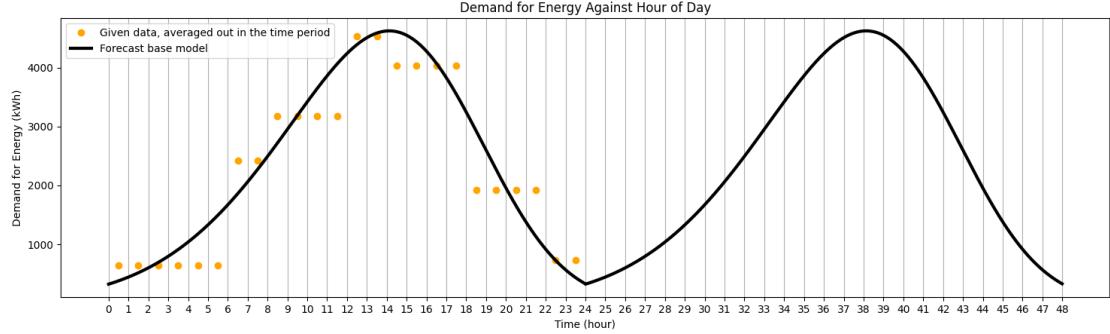


Figure 1: General Demand Schedule

This high degree of seasonality allows us to build a base model, and dynamically adjust the scaling factors in real-time to obtain a forecast of future demand; as well as adding random amplification to simulate sudden increase in demand.

We built four models to estimate demand. A general one to estimate the amount of energy required for a time period in hours; and three to map the demand for high-, medium- and low-urgency tasks respectively.

General demand schedule model (Fig. 1)

$$D_g(x) = 59000 \cdot \left[\frac{\exp\left(\frac{-1}{2} \cdot \left(\frac{x-18.1}{7.4}\right)^2\right)}{7.4\sqrt{2\pi}} \right] \cdot [1 + \text{erf}\left(\frac{-0.25 * (x - 18.1)}{\sqrt{2}}\right)]$$

General form:

$$D_g(x) = k \cdot \text{NormDist}\left[\frac{\exp\left(\frac{-1}{2} \cdot \left(\frac{x-pos}{sd}\right)^2\right)}{sd\sqrt{2\pi}}\right] \cdot \text{SkewFunc}\left[1 + \text{erf}\left(\frac{\text{SkewFactor} * (x - pos)}{\sqrt{2}}\right)\right]$$

We observe from both the data given in the problem statement and real-life small-scale data centres that the computational demand is skewed towards the afternoon/evening hours instead of perfectly distributed around 12 noon. We chose to model with the equation of skewed normal distribution. It is obvious that various factors, such as user activity and workload influence the demand for servers. According to the Central Limit Theorem, the sum of independent random variables tends toward a normal distribution regardless of the original distribution of the variables. Hence modelling the demand schedule with normal distribution is justified. Empirical studies have shown that in many cases, the demand for server services over time tends to exhibit a skewed bell-shaped curve similar to that of a normal distribution with a peak in the afternoon, while there are fluctuations, modelling the demand schedule with a skewed normal distribution gives a simpler and more convenient base model. [4]

We understand that the yellow points are not uniformly distributed through the time period, hence when we were fine-tuning the parameters, the cost function to minimise was the maximum squared difference between the areas under D_g and the amount of energy demanded for all the time periods specified in the problem statement. The parameters of the model, namely the scale (57500), standard deviation (5.31) and mean (13.6) came from an exhaustion to minimise the cost function.

High-urgency tasks demand schedule model (Fig. 2)

$$D_h(t) = \frac{1}{3} \max[10 + 2 \sum_{n=1}^1 \frac{24(1 + (-1)^{n+1})}{n\pi} \sin \frac{n\pi}{12}(t - 9), 0.2] +$$

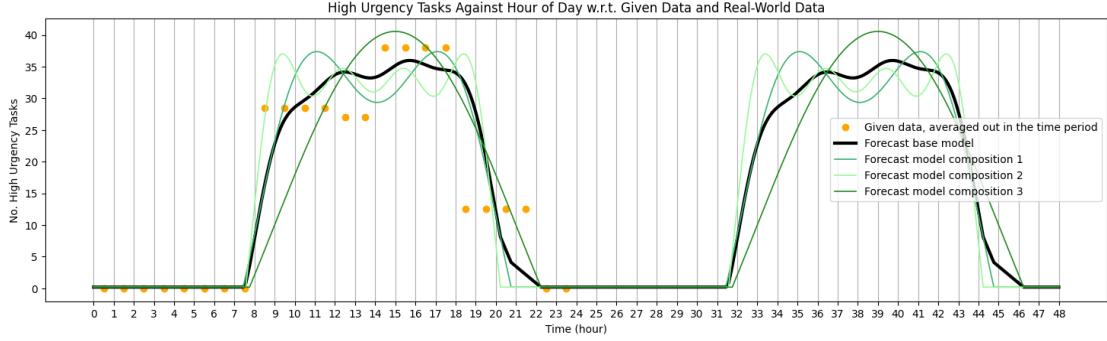


Figure 2: Number of high-urgency tasks schedule

$$\begin{aligned} & \frac{1}{3} \max[10 + 1.9 \sum_{n=1}^3 \frac{24(1 + (-1)^{n+1})}{n\pi} \sin \frac{n\pi}{12}(t - 8.1), 0.2] + \\ & \frac{1}{3} \max[10 + 1.9 \sum_{n=1}^7 \frac{24(1 + (-1)^{n+1})}{n\pi} \sin \frac{n\pi}{12}(t - 7.9), 0.2] \end{aligned}$$

Inspired by the principle behind dynamic harmonic regression and harmonic analysis, and the fact that any periodic function has a Fourier series representation. Periodic functions like sine and cosine are appropriate for demand estimation. We observe that the high-urgency tasks only come in the daytime and keep at a steady level with minor fluctuations or spikes, they closely resemble square waves, so we decided to use the Fourier series to find a regression model for high-urgency tasks.

The model consists of three Fourier series with different n , *mean* and *pos*, then take the average between the three to obtain $D_h(t)$. It is observed that there is no quantity demanded of high-urgency tasks during the night, therefore our model takes the maximum between the Fourier series and 0.2, so we can get a better estimation with our Fourier series where the data are non-zero. The $n = 3, 7$ Fourier series are there to introduce some fluctuations to the model, as well as take care of some earlier server requests. The last Fourier series term $n = 1$ is just a sine function, its purpose is to introduce higher demand in the afternoon, as well as ensure adequate demand during evening times.

A minimum value of 0.2 tasks is given as lower-bound aligns more with real-life demand and allows us to calculate the forecast of the demand more easily by preventing division by 0.

Medium-urgency tasks demand schedule model

$$D_m(t) = 13.3 + 5.9 \sin(\frac{\pi}{12}(t - 7.2))$$

It is observed that there are quite significant spikes from the usual demand from medium-urgency tasks, those are treated as random amplifications and will be handled by the forecasting model as randomized events. The base model medium-urgency tasks follows a sine curve with more demand during the daytime, which aligns with real-world data, hence a simple function is used for the base model.

Low-urgency tasks demand schedule model

$$D_l(t) = \max[5.44 - 0.413 \sum_{n=1}^3 \frac{24(1 + (-1)^{n+1})}{n\pi} \sin(\frac{n\pi}{12}(t - 6.57)), 0.2]$$

A Fourier series composed of two sine functions is used to model demand for low-urgency tasks. The given data has an obvious abnormality (35kWh per hour at hours

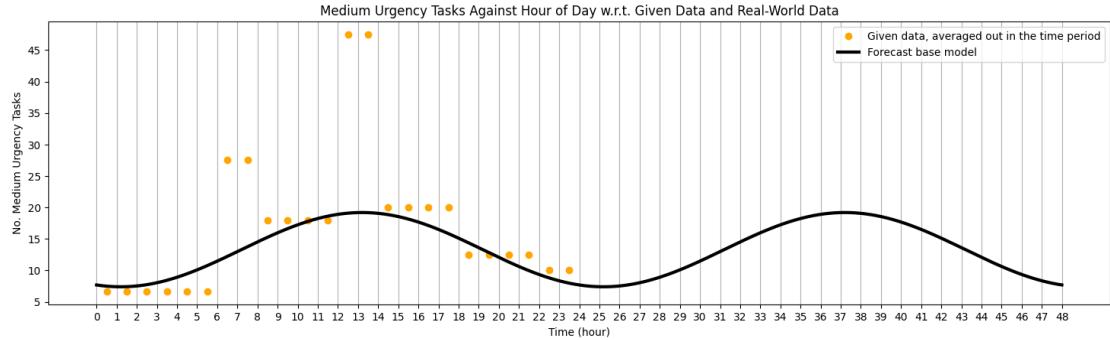


Figure 3: Number of medium-urgency tasks schedule

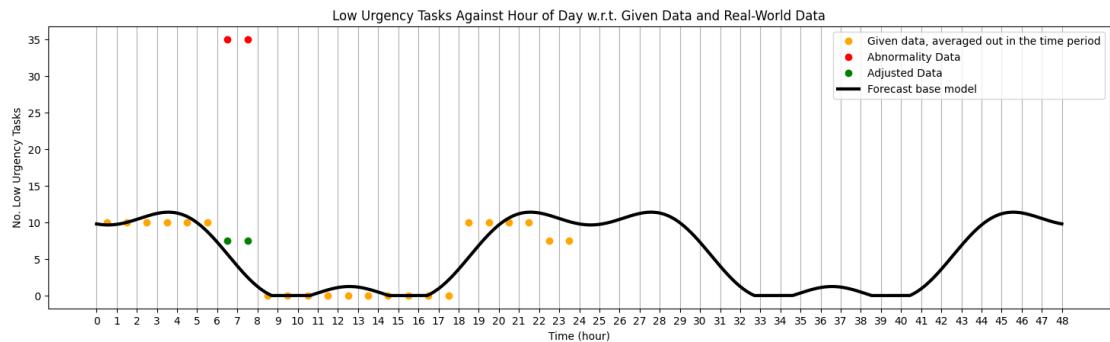


Figure 4: Number of low-urgency tasks schedule

6-8), more than three times the next highest value. We adjusted the data for those two hours and observed that the curve looked like a square wave with "on" and "off" values with similar amplitudes. The composition of the two sine waves with period $\frac{1}{12}$ and $\frac{1}{4}$ gives a smooth transition between the high and low demands to align with real data; creates a significant high and low pattern; and has some variation around the maximum energy level to align more with realistic demands.

Demand schedule overview

When adding together the individual demand schedule with different levels of urgency, and considering the random spikes we will include in the forecasting model on top of the base models included above, the total energy required closely aligns with our general model.

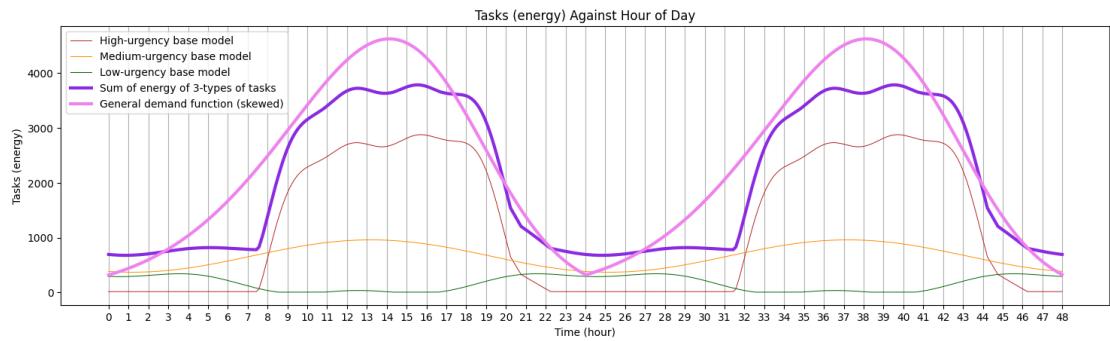


Figure 5: Overview of the demand schedules

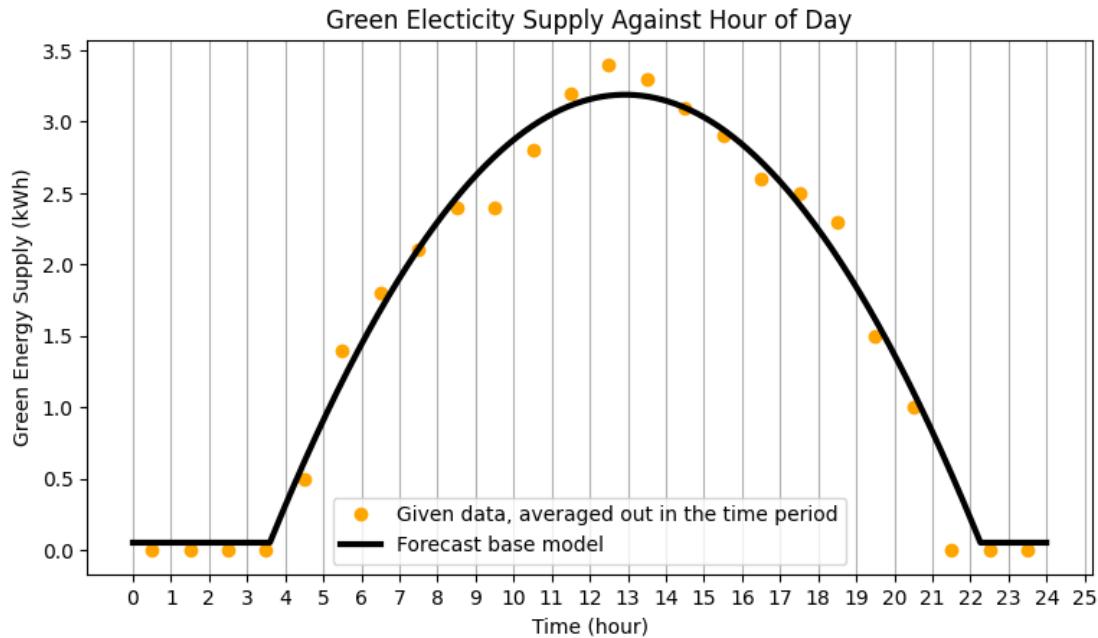


Figure 6: Supply of green electricity modelled against hour of day

§3.4.2 Supply

The supply of green electricity can also be modelled against time.

$$S(t) = \max(21.58476 \cdot \sin[0.0585206 \cdot (t \pmod{24}) + 0.814195] - 18.39532, 0.05)$$

Green electricity comes from natural sources, which has a very high seasonality every day. Therefore we decided to model supply with time.

Assumption 4 — Traditional energy has practically infinite supply for our data center's needs.

The supply of traditional energy is rather stable, and a large proportion of the energy produced today is traditional energy. Renewable energy takes up only 1% of CLP Hong Kong Limited's power output. [5]

§3.4.3 Price

"The electricity price fluctuates based on supply and demand." - Problem statement

By the problem statement, price is modelled by demand and supply. As demand and supply are functions of time, it is also reasonable for us to establish a price curve as a function of time.

$$\text{Price}(t) = 0.147863 \cdot \sin(0.312526(t \pmod{24}) + 1.38352) + 0.47101$$

§3.4.4 Predicting Future Demand, Supply and Price

The fitted curves represent a day with **average demand and supply**. Demand and supply may vary by day. Demand for data centre services is volatile, depending on the day of the week, economic activity (e.g. business cycle) and market environment. Supply of energy varies according to the environment (uncontrollable renewable energy resources

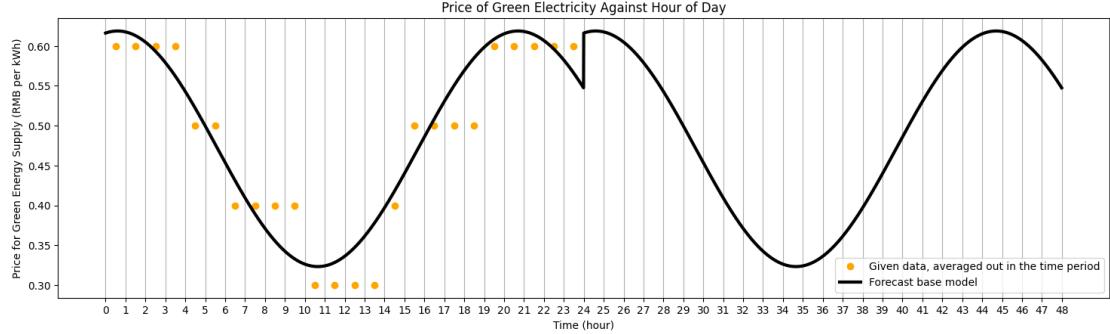


Figure 7: Price of Green Electricity against hour of day(RMB per kWh)

such as wind power and solar power depend heavily on the weather), long-term resource exhaustion (fossil fuels are becoming increasingly rare), and technological improvements (reducing the cost of production).

By definition, trends are continuous. For example, demand may steadily increase from Fridays to Sundays. The cost of energy production is gradually decreasing and hence the supply, in general, is gradually increasing. We take the fitted curve as a baseline, and **adjust the expected demand and expected supply** based on recent real-time data.

We take the past 24 hours of data for predictions of the upcoming hour/day's demand and supply. Let the scale factor of demand,

$$SF_{D,i} = \frac{D_i}{f(i)}$$

where f is the fitted demand curve. The scale factor of supply, $SF_{S,i}$, is defined similarly. If $SF_{D,i} > 1$, the expected value of demand is an underestimate, and an overestimate otherwise. A day busier than average will generally have $SF_{D,i} > 1$ for all i .

Our goal is to predict $SF_{D,h}$ and $SF_{S,h}$ for the upcoming hour. If k hours have elapsed since the recording of scale factor data, We take the **geometric mean** GM of the past $\min(k, 24)$ values of $SF_{D,i}$. We can now estimate

$$\mathbf{E}(D_h) = GM \times f(h)$$

. Estimations are done similarly for supply and price.

Justification for using the geometric mean:

- Geometric means are typically used to average proportions and rates, and our scale factor falls precisely into the two categories.
- We want to minimize the effect of outliers, while not completely eliminating them in our considerations. A geometric mean greatly reduces the magnitude of impact of a particularly huge data point by taking its n -th root.

§3.5 Responding to Real-Time Demand and Supply

Consider the following situation: as a data center, at midnight, we know there is a potential huge spike in High-Urgency Tasks coming at 6:00-8:00, but the increase in supply of energy, S_6 , is insufficient to handle all the new tasks.

Here, we use our predictive modeling of demand and supply to conserve a certain amount of energy, denoted by $cons$, for future hours.

Let our current energy deficit $ED_{cur} = (\max(0, Energy - D_{H,i} - D_{M,i} - D_{L,i}))$. The expected maximum **increase in energy deficit** in the near future, (we take it to be 3 hours), is

$$M = \max_{h < i \leq h+3} \sum_{j=h+1}^i (\mathbf{E}(D_j) - \mathbf{E}(S_j))$$

where $\mathbf{E}(X)$ denotes the expected value of X .

We conserve $\frac{M}{2}$ of energy (that is, after handling tasks in the current hour, there should be $\geq \frac{M}{2}$ energy left), in order to prepare for the foreseeable future. This is so spikes in tasks in the future can be quickly completed.

Justification:

In addition to preparing for the future, it is important to minimize penalty created by current tasks. Therefore, we only reserve half of the energy needed, to reduce the penalty created now. In addition, we only estimate the next 3 hours, to increase the accuracy of estimation (estimations require recent data), to remain flexible in our energy allocation. In addition, if we were to set aside a chunk of energy for more than 3 hours, it is very possible that current tasks would be delayed for more than 3 hours, making the conservation no longer worth it in terms of minimizing penalty.

§3.6 Allocating Energy: An Algorithm

We employ a **greedy algorithm** to complete the tasks in descending priority.

Assumption 5 — For a certain hour, after a certain supply of energy is purchased and consumed, there is insufficient remaining energy to complete the highest priority task, no task is started.

Justification: Tasks and unused energy may be carried over to the next day. Data centers prefer to complete tasks in whole - an uncompleted query cannot be returned to the person who requested it.

With this assumption, tasks must be completed in whole. It remains to decide on the order of the tasks.

A polynomial-time algorithm to determine the optimal order to minimize penalty is difficult (we are not sure of its existence) and unrealistic to implement. On the other hand, exhausting all possible orders, with a running time of at least $\mathcal{O}(2^n)$, is far too slow for about $n = 1000$ total tasks. Heuristic orderings that produce semi-optimal solutions are therefore employed.

Heuristic Orderings 1

Three orderings of tasks are proposed:

1. Order by descending P_i , tiebreak by decreasing urgency
2. Order by decreasing urgency, tiebreak by P_i
3. At hour h , order by $\delta = w_i \times (\max(0, (h+1) - DL_i + 1) - \max(0, h - DL_i + 1))$, the increase in penalty shall T_i be done a day later

Let A be a set of tasks (i, P_i) , each represented by an ordered pair of integers - its index i , and priority index P_i . Let (E, H, M, L) be the state at the ending of the $(h-1)$ -th hour, indicating that there are E kWh of unused energy leftover, H , M and L kWh of demand contributed by High, Medium, and Low Urgency Tasks respectively.

For hour h , we may use

$$cap' = \max(0, \min(cap, E + S_h - \frac{M}{2}))$$

where M is the future energy deficit as defined above.

In the beginning of hour h , S_h kWh of energy is added to E . We iterate over the set of tasks using one of the heuristic orderings.

If the amount of energy consumed in this hour is cur , and $cur + Q_i \leq cap'$ (so it does not exceed the adjusted computation capacity) we perform 3 updates to the variables we maintain:

1. $cur \leftarrow cur + Q_i$,
2. $late_i \leftarrow h$,
3. remove (i, P_i) from the set A .

This algorithm runs in $\mathcal{O}(n \log n)$ using a heap (`priority_queue` in C++) data structure, which is used to speed up the heuristic orderings, where n is the number of tasks. The top of the heap, which is the most prioritized task, the insertion/removal of tasks, can be achieved in $\mathcal{O}(\log n)$.

§4 Task 2: Balancing Going Green and Cutting Costs

§4.1 Preliminary Assumptions

Assumption 6 — Traditional energy has a stable price across days.

It follows from the last Assumption, that supply is stable. The demand is stable as the data center is not the only entity using energy in the world. By the law of large numbers, the total demand for energy will remain relatively constant between days and converges to the mean. When both demand and supply are stable, price is stable.

Assumption 7 — Power and electricity is supplied at the start of an hour. It can be used immediately for the new tasks requested in the same hour.

Assumption 8 — The data center is operating in the long run and it would like to minimize its **long run average cost** (LRAC).

The problem statements for Tasks 2 and 3 does not involve scheduling in a short 24-hour period.

Assumption 9 (Necessary Condition in Operation) — The data center completes **all** tasks eventually.

The problem statements for Tasks 2 and 3 focuses on cost optimization. We must impose conditions for cost optimization to hold any meaning, for we can achieve 0 cost by simply not completing any tasks. It is also obviously expected that data centers complete all its tasks. We therefore choose to impose this condition.

§4.2 A Few Critical Questions

We aim to answer a few main questions:

1. How many kWh of energy should be reserved?
2. What is the mean daily cost of production, under various proportions of green energy?
3. How well can our algorithm adapt to significant fluctuations in demand and supply?

Note that these questions will be answered in accordance to the data given in Tables 1-4 in the Problem Statement. However, we will still forecast random data below.

§4.3 General Strategy

For a fixed price schedule, to facilitate cost optimization, a natural way to approach buying energy for a day is to:

1. Estimate the energy usage in the coming day, E , to **complete all tasks**.
2. Look through the price schedule PS in advance, which can be represented by a list of ordered tuples of integers (t, s, p) , where s is the available supply of energy that can be purchased at the given price p in time interval t . Sort PS in ascending order of p . Traditional energy, which supply is effectively infinite for our data center's needs, will be taken as ∞ .
3. Iterate through each item in the sorted list in order, purchasing

$$bought = \max(0, E - s)$$

kWh of energy, paying a price of $bought \times p$ ($cost \leftarrow cost + bought \times p$), and decreasing the remaining energy required, E by $bought$ (formally, $E \leftarrow E - bought$).

This strategy purchases precisely enough energy to satisfy the demand, at the minimal cost. We can be certain the cost is at its global minimum.

However, putting aside potentially inaccurate estimations of energy usage and supply, this greedy strategy is still flawed: if the cheapest prices are in the second half of the day, and our greedy strategy decides to only purchase at such prices, there would be no energy for the data center to complete any tasks in the morning. To tackle this problem, we introduce an important assumption:

Assumption 10 (Energy Reserve) — The data center possesses an energy reserve of R kWh.

Justification:

- Such a reserve is necessary and realistic. Real world data centers have an energy reserve that they can feed into in case their power supply is cut off [2]. There is otherwise no way to adapt quickly to power supply outages.
- Such a reserve is bounded and practical. It is easy to see that R is smaller or equal to the maximum energy usage in a day. The setup cost of this strategy is very affordable, relative to the production scale of the data center.

Note that the purpose of energy reserve is not only to deal with sudden decreases in power supply. It also enables the General Strategy described above. If the time at which prices have reached a daily minimum has not been reached yet, and tasks need to be completed, energy may simply be retrieved from the energy reserve. Step 3 in the General Strategy will refill the reserve back up to a constant level of R kWh at the end of a day.

We can compute R in our simulations of the data center's operations with a few modifications. The center starts with no reserve ($R \leftarrow 0$), but **it may go into energy debt**. That is, the amount of energy it possesses, E , may be negative. By the end of each day, the energy level will be refilled back to 0. The simulation is ran over a prolonged period of time (1000 days). Every time a task is completed, and energy is consumed, R is updated by

$$R \leftarrow \max(R, -E)$$

This is because we must ensure $R + E \geq 0 \implies R \geq -E$ at all times.

Reserving energy prior to the beginning of operation is a fixed cost. Theoretically, it has **negligible impact** on the average daily cost of production, under a sufficiently long production period.

§4.4 Going Green

The previous section puts its sole focus on cost optimization.

Let $target$ be the target proportion (in percentage) of green energy to be used. In step 1 of our General Strategy, the energy usage in a day, E , is predicted. Therefore, $E_{green} = E * target$.

We apply the algorithm in the general strategy, except with total energy usage E_{green} , with price and supply data of green energy only.

§4.5 Running The Monte Carlo Simulations

The result of the model and algorithm (minimum reserve needed, penalty, etc.) is too complicated to solve analytically for any arbitrary set of data. We therefore run Monte Carlo Simulations on well-generated random data that aims to simulate real-world situations, to stress test our model and scheduling algorithm under significant fluctuations and so it can return accurate numerical values.

It is important to generate random data that:

- is noisy
- has periods where it is generally increasing (or decreasing), to simulate real life trends
- contains extreme cases, like power outages or sudden jumps in demand

For brevity, the process of generating demand data is described. Let F_i be a metric representing the factor at which data is scaled, compared to the baseline fitted curve, during a day. F_i is rather continuous - between all days i and $i + 1$, we have $|F_i - F_{i+1}| \leq 0.25$, and $F_i \in [0.5, 2.0]$. We also introduce a boolean variable $trend$, that has a 0.02 probability of toggling between increasing and decreasing. Shall $trend = \text{increasing}$, F will increase by a random real number in $U[0, 0.05]$ between days with 0.9 probability, and it will increase by a random real number in $U[0, 0.25]$ between days with 0.1 probability, where U represents a uniform distribution. The $trend = \text{decreasing}$ case is similar.

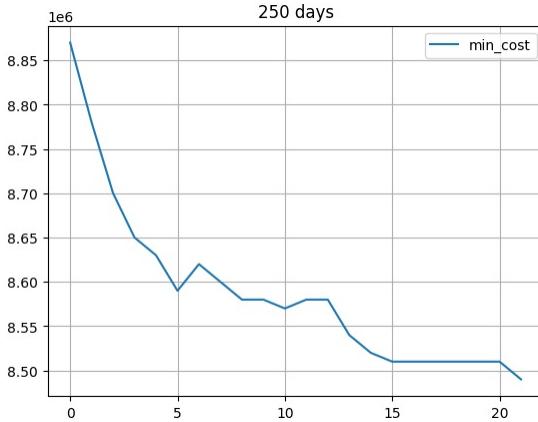


Figure 8:

We now generate D_h , the value of demand in each hour of the day. We let $\lambda = f(h)$, where f is the fitted, periodic demand curve. The mean value of demand at the h -th hour is λ . To ensure the curve of demand is not periodic and takes different shapes, we use a **Poisson distribution** with a mean of λ , to generate D_h .

In the end, to simulate sudden spikes in demand (such as the aforementioned spikes in Medium Urgency Tasks), $D_h \leftarrow D_h \times 3$ with a probability of $\frac{1}{100}$, and $D_h \leftarrow \frac{D_h}{3}$ with equal probability.

The case of supply is analogous, except instead of increasing demand at specific points in time, we significantly decrease supply to very small amounts (this is done by dividing supply by 100) to simulate power outages.

Assumption 11 — The demand, supply, and price can be treated as a small number if it is 0.

Justification:

1. This has negligible impact on the final calculation.
2. This is to prevent the generated data from having data points with value 0, which would cause the scale factor $SF_{D,i} = 0$. The geometric mean would be 0 and subsequent data would all be predicted to be 0.
3. We also take the values x at points on the fitted curves to be $\max(x, 0.2)$. This is to prevent division by zero errors when the scale factors are calculated.

See Appendix A for a concrete implementation of how the random data is generated in C++.

§4.6 Results

In reality, the algorithm cannot know for sure the future prices and the actual energy usage. All of this is done by estimation, hence there is a small error.

The algorithm is ran on two datasets: one of 250 days and another of 1000 days.

Our results indicate that there is little correlation between the period of production and the maximum reserve needed. This means a reserve of about **80000 kWh** (the maximum reserve is 74962.6 kWh across both simulations) is capable of facilitating optimal purchasing energy and any potential power outages of reasonable durations, for

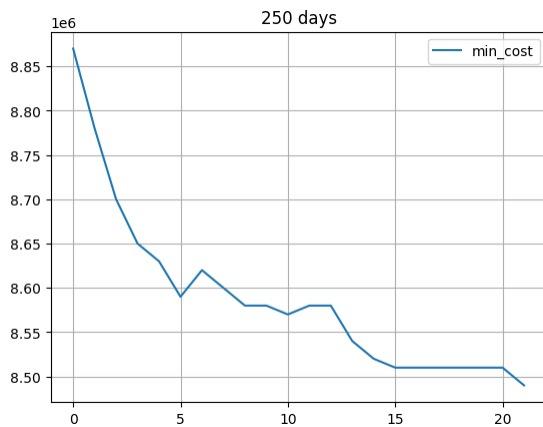


Figure 9:

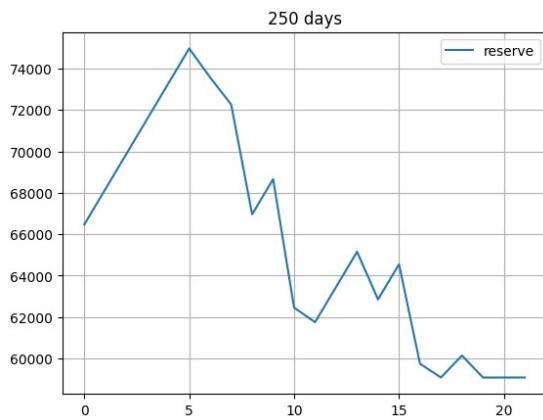


Figure 10:

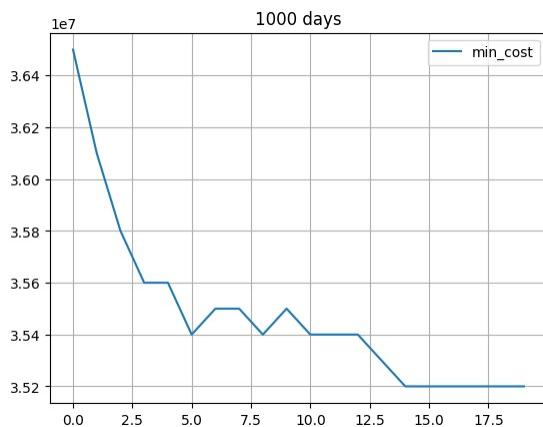


Figure 11:

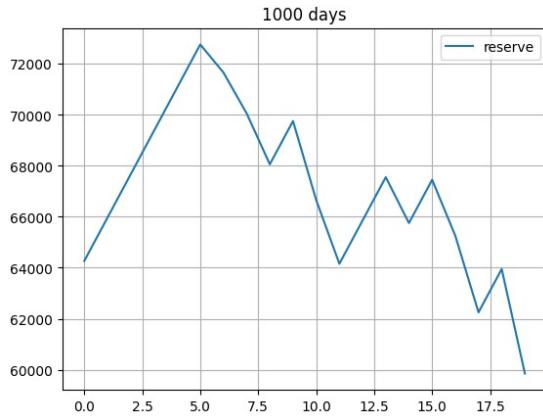


Figure 12:

any duration of production. The total energy requirement in Table 1 is 55750 kWh. This means roughly 1.43 (2 d.p.) days worth of energy need to be reserved. This is reasonable when we take into account the fact that there are sharp increases in demand in our random data.

The setup cost is therefore very small compared to the production costs. It is a very practical measure for data centers.

It is observed that cost decreases as the proportion of green energy increases. This is reasonable as in the sample data, the price of green energy ≤ 0.5 when there is supply, while the minimum price of traditional energy is also 0.5. After adding noise and predicting demand, supply, and price, green energy is not much more expensive than the minimum price of traditional energy. This means that green energy is mostly cheaper than traditional energy [6], hence as its proportion increases, cost decreases.

Therefore it is natural to consider the largest proportion of green energy possible. At 21% of green energy, the average cost is \$33947 (for 250 days), and at 19% of green energy, the average cost is \$35224 (for 1000 days). The discrepancy in the average cost may be explained by:

- Maximum daily demand across all days is larger and minimum green energy supply across all days is smaller when more days are simulated. The maximum possible proportion of green energy hence decreases as more days are simulated.

§5 Task 3

§5.1 A Few Critical Questions

We aim to answer a few main questions:

1. How many kWh of energy should be reserved?
2. When energy is sufficient, what is the minimum penalty that can be attained under a certain computational capacity cap ?
3. What is the mean daily cost of production?
4. How well can our algorithm adapt to significant fluctuations in demand and supply?

§5.2 The Algorithm

A slightly different algorithm is employed than our General Strategy in Task 2, as now there is no concern regarding differentiating between green and traditional energy.

Assumption 12 — Green energy is insufficient to handle demands. The data center must always purchase traditional energy.

Justification:

Similar to Assumption 5, renewable energy takes up only 1% of CLP Hong Kong Limited's power output.

1. Predict the price of traditional energy, p_i in the next 24 hours.
2. Iterate through the hours of the day in order. If there exists an hour h in which p_h is cheaper than the current price of green energy, we do not buy any green energy. Otherwise, we buy the entirety of the supply of the current hour's green energy.
3. At the end of a day, if the energy reserve $E < R$, purchase $R - E$ kWh of traditional energy so energy reserve can reattain the initial level R at the start of the second day.

The algorithm is efficient: it executes in $\mathcal{O}(n \log n)$ time using similar ideas that were previously mentioned.

§5.3 Results

For the 250-day simulation, a reserve of 101893 kWh is needed, and the average daily cost is \$29440.

For the 1000-day simulation, a reserve of 102236 kWh is needed, and the average daily cost is \$30627.

With some testing, it is concluded that the first Heuristic Ordering yields the minimum penalty for the same computational capacity cap . Below is a table listing out capacities and their corresponding penalties:

Capacity	Penalty	Remaining Tasks
0	0.000000	1004272
500	51540769098948.156250	815300
1000	59766102766038.921875	624245
1500	43062825203118.789062	430983
2000	16049138230152.605469	228603
2500	503774261617.490295	35980
3000	5437175839.887891	3093
3500	498379665.388723	1098
4000	26921220.111097	459
4500	9906893.111113	328
5000	4927205.444446	222
5500	2220046.000000	52
6000	776551.444444	0
6500	151848.277778	0
7000	11965.666667	0
7500	4663.055556	0
8000	3425.555556	0
8500	2672.277778	0
9000	2183.000000	0
9500	1781.000000	0
10000	1421.000000	0
10500	1184.000000	0
11000	992.000000	0
11500	800.000000	0
12000	576.000000	0
12500	384.000000	0
13000	240.000000	0
13500	144.000000	0
14000	32.000000	0
14500	0.000000	0

A computational capacity of 14500 (cor. to 3 sig. fig.) kW is the minimum capacity such that the data center can process all tasks, on time, with no penalty.

§6 Aftermath

§6.1 Strengths

1. The algorithms in Task 2 and Task 3 produce the minimum cost possible based on its predictions.
2. Imposing an energy reserve is very versatile. We can deal with large fluctuations in demand and supply by simply increasing the reserve. It greatly reduces our concerns with energy, making the limiting factor of operation the computational capacity.

§6.2 Weaknesses and Limitations

1. Our predictions of demand, supply, and price are based on a limited set of data.

2. An energy reserve cannot solve the short-run problem where there cannot be any setup.

§6.3 Things to Improve

1. More data from the real world is needed to eliminate the effects of outliers on our fitted curves.
2. We may use more advanced models, algorithms and ideas such as Network Flows (e.g. Successive Shortest Path to solve the Min Cost Max Flow Problem) or Dynamic Programming to evaluate the minimum cost to replace our Heuristic Ordering.

Note 1 (AI Report). LLMs such as ChatGPT were not used, and played no role in the process of creating the mathematical model and this solution paper.

§7 References

1. URL: <https://www.datacenterknowledge.com/energy-power-supply/data-center-power-fueling-the-digital-revolution>
2. URL: <https://www.m-q.ch/en/how-data-centers-handle-lack-of-power-and-security-of-supply/>
3. URL: <https://www.researchgate.net/publication/255215593>
4. URL: <https://www.researchgate.net/publication/324181793>
5. URL: <https://www.clp.com.hk/en/about-clp/power-generation>
6. URL: <https://www.theecoexperts.co.uk/news/is-renewable-energy-cheaper-than-fossil-fuels>

§8 Appendix

§8.1 Appendix A: Data Generation

```
#include <bits/stdc++.h>
using namespace std;

const int DAY_COUNT = 1000;

mt19937 rng(12345); // fixed seed so results can be easily reproduced

double rnd(double u, double v) {
    return uniform_real_distribution<> (u, v) (rng);
}

const double scale_1 = 1;
const double scale_2 = 147000;
const double shape = -0.29; // # should be negative in the final eq
const double pos = 17.6;
```

```

const double sd_skew = 10.3;
const double mean_skew = 8.9;

double square(double x) {
    return x * x;
}

double normal(double x) {
    return exp(-0.5 * square((x - mean_skew) / sd_skew)) / sd_skew / sqrt(2.0 * acos(-1));
}

double skew(double x) {
    return 0.5 * (1.0 + erf(x / sqrt(2.0)));
}

double demand_func_skewed_normal (int x) {
    x %= 24;
    return scale_2 * (2.0 / scale_1) * normal((x - pos) / scale_1) * skew(shape * ((x - pos) / scale_1));
}

double expect_supply(int cur_time){
    cur_time %= 24;
    return (21.58476*sin(0.0585206*cur_time+0.814195)-18.39532);
}

double expect_new_en_price(int cur_time){
    cur_time %= 24;
    return max((double) 0.2, 0.147863 * sin(0.312526 * cur_time + 1.38352) + 0.47101);
}

int main() {
    ios::sync_with_stdio(0); cin.tie(0); // fast input and output
    freopen("algo/in.txt", "w", stdout);
    cout << 1 << '\n';
    int trend = rnd(0, 1) < 0.5 ? 0 : 1; // 0: decreasing, 1: increasing
    double F = 1;
    for (int day = 0; day < DAY_COUNT; day++) {
        for (int h = 0; h < 24; h++) {
            if (rnd(0, 1) < 0.2) trend ^= 1; // change trend with probability 2%
            if (!trend) {
                if (rnd(0, 1) < 0.9) F -= rnd(0, 0.05);
                else F -= rnd(0, 0.25);
            } else {
                if (rnd(0, 1) < 0.9) F += rnd(0, 0.05);
                else F += rnd(0, 0.25);
            }
            F = max(F, 0.5);
            F = min(F, 2.0);
            poisson_distribution<> d(demand_func_skewed_normal(h) * F);
        }
    }
}

```

```

        double noises = 1;
        if (rnd(0, 1) < 1.0/10000) noises = 3;
        else if (rnd(0, 1) < 1.0/10000) noises = 1.0/3;
        double res = d(rng) * noises;
        res = max(res, (double) 0);
        cout << res << ' ';
    }
}
cout << '\n';
for (int day = 0; day < DAY_COUNT; day++) {
    for (int h = 0; h < 24; h++) cout << "0 ";
}
cout << '\n';
for (int day = 0; day < DAY_COUNT; day++) {
    for (int h = 0; h < 24; h++) cout << "0 ";
}
cout << '\n';
F = 1;
trend = rnd(0, 1) < 0.5 ? 0 : 1;
for (int day = 0; day < DAY_COUNT; day++) {
    for (int h = 0; h < 24; h++) {
        if (rnd(0, 1) < 0.2) trend ^= 1; // change trend with probability 2%
        if (!trend) {
            if (rnd(0, 1) < 0.9) F -= rnd(0, 0.05);
            else F -= rnd(0, 0.25);
        } else {
            if (rnd(0, 1) < 0.9) F += rnd(0, 0.05);
            else F += rnd(0, 0.25);
        }
        F = max(F, 0.5);
        F = min(F, 2.0);
        poisson_distribution<> d(expect_supply(h) * F * 10);
        double noises = 1;
        if (rnd(0, 1) < 1.0/10000) noises = 2;
        else if (rnd(0, 1) < 1.0/10000) noises = 0.01;
        double meh = d(rng);
        double res = meh * noises / 10;
        res = max(res, (double) 0.2);
        cout << res << ' ';
    }
}
cout << '\n';
F = 1;
trend = rnd(0, 1) < 0.5 ? 0 : 1;
for (int day = 0; day < DAY_COUNT; day++) {
    for (int h = 0; h < 24; h++) {
        double price = expect_new_en_price(h);
        double res = price * rnd(0.8, 1.2);
        res = max(res, (double) 0.2);
        cout << res << ' ';
    }
}

```

```

        }
    }
    cout << '\n';
}

§8.2 Appendix B: Task 2

#include <bits/stdc++.h>
using namespace std;

#define int long long
#define L(i,j,k) for(int i=(int)(j);i<(int)(k);i++)
#define db long double

const double threshhold=0;

double eps = 1e-9;
vector<int>green_energy_percentages={};
int timer = 0;
double cost = 0;
vector<double>lnprice,lndemand,lnsupply;
bool buy_new_energy[24];
double sumlnd=0, sumlnprice=0, sumlnsupply=0;
double traditional_energy_costs[24]={0.5,0.5,0.5,0.5,0.6,0.7,0.8,0.9,1,1.2,1.3,1.3,1.2,1
const int maxt = 1000;
int DH[24*maxt], DM[24*maxt], DL[24*maxt];
double RGPS[24*maxt], RGPP[24*maxt]; // real green power supplied
double global_minimum=0;

struct info{
    double price,supply; int idx;
    bool operator<(const info& I) const{
        return price<I.price;
    }
};

void clear_data(){
    global_minimum=sumlnd=sumlnprice=sumlnsupply=0;
    lnprice.clear(); lndemand.clear(); lnsupply.clear();
    timer=0; cost=0;
    for(int i=0;i<24;i++)buy_new_energy[i]=0;
}

int conv(char urgency) {
    if (urgency == 'H') return 1;
    if (urgency == 'M') return 2;
    if (urgency == 'L') return 3;
    assert(false);
    return 42;
}

struct node {

```

```

        double price,reserve; int per;
    };

// double geometric_mean(vector<double>& data, double& ln_sum) {
//     if (data.empty()) return 1;
//     double sum_log = 0;
//     ln_sum +=
//     return exp(sum_log / (int) data.size());
// }

/*
    All energy calculated in the code is based on kWh
    RGPS in real green power supplied
*/
void init() {
    timer = cost = 0;
}

void input() {
    L(i,0,24*maxt){
        cin>>DH[i];
    }
    L(i,0,24*maxt){
        cin>>DM[i];
    }
    L(i,0,24*maxt){
        cin>>DL[i];
    }
    L(i,0,24*maxt){
        cin>>RGPS[i];
    }
    L(i,0,24*maxt){
        cin>>RGPP[i];
    }
}
double square(double x) {
    return x * x;
}

double expect_new_en_price(int cur_time){
    cur_time %= 24;
    return max((double) 0.2, 0.147863 * sin(0.312526 * cur_time + 1.38352) +0.47101);
}

const double scale_1 = 1;
const double scale_2 = 147000;
const double shape = -0.29; // # should be negative in the final eq
const double pos = 17.6;
const double sd_skew = 10.3;

```

```

const double mean_skew = 8.9;

double normal(double x) {
    return exp(-0.5 * square((x - mean_skew) / sd_skew)) / sd_skew / sqrt(2.0 * acos(-1));
}

double skew(double x) {
    return 0.5 * (1.0 + erf(x / sqrt(2.0)));
}

double expect_en_demand (int x) {
    x %= 24;
    return scale_2 * (2.0 / scale_1) * normal((x - pos) / scale_1) * skew(shape * ((x - pos) / scale_1));
}

double expect_supply(int cur_time){
    cur_time %= 24;
    return max((double) 0.2,(21.58476*sin(0.0585206*cur_time+0.814195)-18.39532));
}

double predict_en_demand(int cur_time){
    // this is a function that predicts the energy demand
    double expected_demand = expect_en_demand((cur_time) % 24);
    // assert(expected_demand > eps);
    if((int)lndemand.size()==0) return expected_demand;
    double demand = expected_demand * exp(sumlnd / (int)lndemand.size());
    return demand;
}

double predict_en_supply(int cur_time){
    // this is a function that predicts the energy demand
    double expected_supply = expect_supply((cur_time) % 24);
    // assert(expected_supply > eps);
    if((int)lnsupply.size()==0) return expected_supply;
    double supply = expected_supply * exp(sumlnsupply / (int)lnsupply.size());
    return supply;
}

double predict_demand_for_whole_day(int cur_time){
    double total_demand=0;
    for(int i=cur_time+1;i<=cur_time+24;i++){
        total_demand+=predict_en_demand(i);
    }
    return total_demand;
}

double predict_supply_for_whole_day(int cur_time){
    double total_supply=0;
    for(int i=cur_time+1;i<=cur_time+24;i++){
        total_supply+=predict_en_supply(i);
    }
}

```

```

    }
    return total_supply;
}

double predict_new_en_price(int cur_time){
    double expected_price = expect_new_en_price((cur_time) % 24);
    // assert(expected_demand > eps);
    if((int)lnprice.size()==0) return expected_price;
    double price = expected_price * exp(sumlnprice / (int)lnprice.size());
    return price;
}

void new_energy_cost_buying_predictions(int cur_time, int energy_required){
    for(int i=0;i<24;i++)buy_new_energy[i]=0;
    vector<info>new_energy_information;
    for(int i=1;i<=24;i++){
        new_energy_information.push_back({expect_new_en_price((cur_time+i)%24),expect_supply[i]});
    }
    sort(new_energy_information.begin(),new_energy_information.end());
    // supply represented in unit: MW
    for(auto [price,supply,idx]: new_energy_information){
        int energy_can_get=supply*1000;
        energy_required-=energy_can_get;
        buy_new_energy[idx]=1;
        if(energy_required<=0) break;
    }
    return;
}
double trad_en_price(int cur_time){
    // use original
    return traditional_energy_costs[cur_time];
}

void update_price(int cur_time, double real_price){
    double expected_price=expect_new_en_price(cur_time);
    lnprice.push_back(log(real_price/expected_price));
    sumlnprice+=lnprice.back();
    if(lnprice.size()>24){
        sumlnprice-=lnprice.front();
        lnprice.erase(lnprice.begin());
    }
}

void update_demand(int cur_time, double real_demand){
    double expected_demand=expect_en_demand(cur_time);
    lndemand.push_back(log(real_demand/expected_demand));
    sumlnd+=lndemand.back();
    if(lndemand.size()>24){
        sumlnd-=lndemand.front();
    }
}

```

```

        lndemand.erase(lndemand.begin());
    }
}

void update_supply(int cur_time, double real_supply){
    double expected_supply=expect_supply(cur_time);
    lnsupply.push_back(log(real_supply/expected_supply));
    sumlnsupply+=lnsupply.back();
    if(lnsupply.size()>24){
        sumlnsupply-=lnsupply.front();
        lnsupply.erase(lnsupply.begin());
    }
}
/*
1. Predict demand and supply and total needed energy for a day the respective costs v
2. Strategy:
- Reserve a certain amount of energy
- Always buy traditional energy at its lowest cost, based on prediction results
- Exhaust green energy % and run simulation cost
- Buy the respective amount of green energy at times with lowest cost
*/
pair<double,double> test_green_energy_percentage(int percent){
    // vector<double> t;
    // vector<double> lns_d, lns_s;
    // double ln_sum_d = 0, ln_sum_s = 0;
    // for (int i = 0; i < 24 * maxt; i++) {
    //     for (int j = 1; j <= 24; j++) {
    //         double expected_demand = ED((i+j) % 24);
    //         assert(expected_demand > eps);
    //         double demand = expected_demand * exp(ln_sum_d / lns_d.size());
    //         double expected_supply = ES((i+j) % 24);
    //         double supply = expected_supply * exp(ln_sum_s / lns_s.size());
    //     }
    //     double expected_demand = ED(i % 24);
    //     double total_real_demand = DH[i] + DM[i] + DL[i];
    //     double expected_supply = ED(i % 24);
    //     double total_real_supply = S[i];
    //     lns_d.push_back(log(total_real_demand / expected_demand));
    //     ln_sum_d += lns_d.back();
    //     while (lns_d.size() > 48) ln_sum_d -= lns_d.front(), lns_d.erase(lns_d.begin());
    // }
    int trad_energy_percent=100-percent;
    double initial_energy=0;
    for(int i=0;i<24*maxt;i++){
        double real_energy_needed=DH[i]+DM[i]+DL[i];
        double real_green_energy_supplied=RGPS[i];
        real_green_energy_supplied*=1000;
        double real_green_energy_price=RGPP[i];
        if(buy_new_energy[i%24]){
            initial_energy+=real_green_energy_supplied;
            cost+=real_green_energy_supplied*real_green_energy_price;
        }
    }
}

```

```

    }
    initial_energy-=real_energy_needed;
    update_price(i,real_green_energy_price);
    update_demand(i,real_energy_needed);
    update_supply(i,real_green_energy_supplied);
    global_minimum=min(global_minimum,initial_energy);
    if(i%24==23){
        double dmd=predict_demand_for_whole_day(i);
        double sply=predict_supply_for_whole_day(i);
        // cout << dmd << " " << sply << '\n';
        double tradition_energy_required=dmd*trad_energy_percent/100;
        cost+=tradition_energy_required*traditional_energy_costs[0];
        initial_energy+=tradition_energy_required;
        double green_energy_required=dmd*percent/100;
        if(sply+threshhold<green_energy_required){
            return {-1,-1};
        }
        new_energy_cost_buying_predictions(i,green_energy_required);
    }
}
return {cost,-global_minimum};
}
void solve() {
    vector<node>genergy;
    for(auto& per: green_energy_percentages){
        pair<double,double>tester=test_green_energy_percentage(per);
        clear_data();
        genergy.push_back({tester.first,tester.second,per});
    }
    for(auto& [price,reserve,per]: genergy){
        cout<<"For the percentage "<<per<<": \n";
        if(price==-1){
            cout<<"It is impossible to achieve this percentaage usage of green energy.\n";
        }
        else{
            cout<<"It is possible to use this percentage of green energy, with a minimum ";
        }
    }
}

int32_t main() {
    ios::sync_with_stdio(0); cin.tie(0);
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    for(int i=0;i<100;i++){
        green_energy_percentages.push_back(i);
    }
    int test = 1;
    cin >> test;
}

```

```

        while (test--) {
            init();
            input();
            solve();
        }
    }
}

```

§8.3 Appendix C: Task 3

```

#include <bits/stdc++.h>
using namespace std;

#define int long long
#define L(i,j,k) for(int i=(int)(j);i<(int)(k);i++)
#define db long double

const double threshhold=0;

double eps = 1e-9;
vector<int>green_energy_percentages={};
int timer = 0;
double cost = 0;
vector<double>lnprice,lndemand,lnsupply;
bool buy_new_energy[24];
double sumlnd=0, sumlnprice=0, sumlnsupply=0;
double traditional_energy_costs[24]={0.5,0.5,0.5,0.5,0.6,0.7,0.8,0.9,1,1.2,1.3,1.3,1.2,1
const int maxt = 1000;
int DH[24*maxt], DM[24*maxt], DL[24*maxt];
double RGPS[24*maxt], RGPP[24*maxt]; // real green power supplied
double global_minimum=0;

struct info{
    double price,supply; int idx;
    bool operator<(const info& I) const{
        return price<I.price;
    }
};

void clear_data(){
    global_minimum=sumlnd=sumlnprice=sumlnsupply=0;
    lnprice.clear(); lndemand.clear(); lnsupply.clear();
    timer=0; cost=0;
    for(int i=0;i<24;i++)buy_new_energy[i]=0;
}

int conv(char urgency) {
    if (urgency == 'H') return 1;
    if (urgency == 'M') return 2;
    if (urgency == 'L') return 3;
    assert(false);
    return 42;
}
}

```

```

struct node {
    double price,reserve; int per;
};

/*
    All energy calculated in the code is based on kWh
    RGPS in real green power supplied
*/
void init() {
    timer = cost = 0;
}

void input() {
    L(i,0,24*maxt){
        cin>>DH[i];
    }
    L(i,0,24*maxt){
        cin>>DM[i];
    }
    L(i,0,24*maxt){
        cin>>DL[i];
    }
    L(i,0,24*maxt){
        cin>>RGPS[i];
    }
    L(i,0,24*maxt){
        cin>>RGPP[i];
    }
}
double square(double x) {
    return x * x;
}

double expect_new_en_price(int cur_time){
    cur_time %= 24;
    return max((double) 0.2, 0.147863 * sin(0.312526 * cur_time + 1.38352) +0.47101);
}

const double scale_1 = 1;
const double scale_2 = 147000;
const double shape = -0.29; // # should be negative in the final eq
const double pos = 17.6;
const double sd_skew = 10.3;
const double mean_skew = 8.9;

double normal(double x) {
    return exp(-0.5 * square((x - mean_skew) / sd_skew)) / sd_skew / sqrt(2.0 * acos(-1));
}

```

```

double skew(double x) {
    return 0.5 * (1.0 + erf(x / sqrt(2.0)));
}

double expect_en_demand (int x) {
    x %= 24;
    return scale_2 * (2.0 / scale_1) * normal((x - pos) / scale_1) * skew(shape * ((x - pos) / scale_1));
}

double expect_supply(int cur_time){
    cur_time %= 24;
    return max((double) 0.2,(21.58476*sin(0.0585206*cur_time+0.814195)-18.39532));
}

double predict_en_demand(int cur_time){
    // this is a function that predicts the energy demand
    double expected_demand = expect_en_demand((cur_time) % 24);
    // assert(expected_demand > eps);
    if((int)lndemand.size()==0) return expected_demand;
    double demand = expected_demand * exp(sumlnd / (int)lndemand.size());
    return demand;
}

double predict_en_supply(int cur_time){
    // this is a function that predicts the energy demand
    double expected_supply = expect_supply((cur_time) % 24);
    // assert(expected_supply > eps);
    if((int)lnsupply.size()==0) return expected_supply;
    double supply = expected_supply * exp(sumlnsupply / (int)lnsupply.size());
    return supply;
}

double predict_demand_for_whole_day(int cur_time){
    double total_demand=0;
    for(int i=cur_time+1;i<=cur_time+24;i++){
        total_demand+=predict_en_demand(i);
    }
    return total_demand;
}

double predict_supply_for_whole_day(int cur_time){
    double total_supply=0;
    for(int i=cur_time+1;i<=cur_time+24;i++){
        total_supply+=predict_en_supply(i);
    }
    return total_supply;
}

double predict_new_en_price(int cur_time){

```

```

        double expected_price = expect_new_en_price((cur_time) % 24);
        // assert(expected_demand > eps);
        if((int)lnprice.size()==0) return expected_price;
        double price = expected_price * exp(sumlnprice / (int)lnprice.size());
        return price;
    }

void new_energy_cost_buying_predictions(int cur_time, int energy_required){
    for(int i=0;i<24;i++)buy_new_energy[i]=0;
    vector<info>new_energy_information;
    for(int i=1;i<=24;i++){
        new_energy_information.push_back({expect_new_en_price((cur_time+i)%24),expect_supply((cur_time+i)%24)});
    }
    sort(new_energy_information.begin(),new_energy_information.end());
    // supply represented in unit: MW
    for(auto [price,supply,idx]: new_energy_information){
        int energy_can_get=supply*1000;
        energy_required-=energy_can_get;
        buy_new_energy[idx]=1;
        if(energy_required<=0) break;
    }
    return;
}
double trad_en_price(int cur_time){
    // use original
    return traditional_energy_costs[cur_time];
}

void update_price(int cur_time, double real_price){
    double expected_price=expect_new_en_price(cur_time);
    lnprice.push_back(log(real_price/expected_price));
    sumlnprice+=lnprice.back();
    if(lnprice.size()>24){
        sumlnprice-=lnprice.front();
        lnprice.erase(lnprice.begin());
    }
}

void update_demand(int cur_time, double real_demand){
    double expected_demand=expect_en_demand(cur_time);
    lndemand.push_back(log(real_demand/expected_demand));
    sumlnd+=lndemand.back();
    if(lndemand.size()>24){
        sumlnd-=lndemand.front();
        lndemand.erase(lndemand.begin());
    }
}

void update_supply(int cur_time, double real_supply){
    double expected_supply=expect_supply(cur_time);
    lnsupply.push_back(log(real_supply/expected_supply));
}

```

```

sumlnsupply+=lnsupply.back();
if(lnsupply.size()>24){
    sumlnsupply-=lnsupply.front();
    lnsupply.erase(lnsupply.begin());
}
}

bool buy_green(double& real_green_energy){
    // determine whether we should buy the green energy
    return real_green_energy<traditional_energy_costs[0];
}

void buy(double& initi, double& delta_energy, double& pri){
    // simulate the action of buying green energy
    initi+=(delta_energy);
    cost+=pri*delta_energy;
}
/*
Notes:
1. In the same period of time in different days, assume that the price of traditional
*/
void solve() {
    double initial_energy=0;
    for(int i=0;i<maxt*24;i++){
        double real_energy_needed=DH[i]+DM[i]+DL[i];
        double real_green_energy_supplied=RGPS[i];
        real_green_energy_supplied*=1000; // Compute the amount of green energy supplied
        double real_green_energy_price=RGPP[i];
        if(buy_green(real_green_energy_price)){
            buy(initial_energy,real_green_energy_supplied, real_green_energy_price);
            // buy if green energy is cheaper than the cheapest traditional energy
        }
        initial_energy-=real_energy_needed; // consume energy
        global_minimum=min(global_minimum,initial_energy); // Update the minimum energy w
        if(i%24==23){
            initial_energy=-initial_energy;
            initial_energy*=(0.5);
            cost+=initial_energy;
            initial_energy=0; // This is to buy energy until the initial_energy is back to
        }
    }
    cout<<fixed<<setprecision(6)<<"Reserve "<<-global_minimum<<'\n'; // Reserve a little
    cout<<fixed<<setprecision(6)<<"Cost "<<cost<<'\n';
}

int32_t main() {
    ios::sync_with_stdio(0); cin.tie(0);
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    int test = 1;
    cin >> test;
}

```

```

        while (test--) {
            init();
            input();
            solve();
            clear_data();
        }
    }
}

```

§8.4 Appendix D: Computing Penalties under Various Capacities

```

#include <bits/stdc++.h>
using namespace std;

#define int long long
#define L(i,j,k) for(int i=(int)(j);i<(int)(k);i++)
#define db long double
double capacity=5000;
const double threshhold=0;

struct Heruristic{
    int p,urge; double energy;
    bool operator<(const Heruristic& h) const{
        if(p!=h.p) return p>h.p;
        return urge<h.urge;
    }
};
priority_queue<Heruristic>pq;

int ret(int x){
    if(x==1) return 3;
    if(x==2) return 2;
    if(x==4) return 1;
    assert(false);
    return 42;
}
double cost_func(double lateness, int urg){
    double ans=urg;
    double fn=lateness+ret(urg);
    fn/=ret(urg); fn*=fn;
    ans*=fn; return ans;
}
double eps = 1e-9;
vector<int>green_energy_percentages={};
double penalty=0;
int timer = 0;
double cost = 0;
vector<double>lnprice,lndemand,lnsupply;
bool buy_new_energy[24];
double sumlnd=0, sumlnprice=0, sumlnsupply=0;

```

```

double traditional_energy_costs[24]={0.5,0.5,0.5,0.5,0.5,0.6,0.7,0.8,0.9,1,1.2,1.3,1.3,1.2,1
const int maxt = 1000;
double DH[24*maxt], DM[24*maxt], DL[24*maxt];
double RGPS[24*maxt], RGPP[24*maxt]; // real green power supplied
double global_minimum=0;

struct info{
    double price,supply; int idx;
    bool operator<(const info& I) const{
        return price<I.price;
    }
};

void clear_data(){
    global_minimum=sumlnd=sumlnprice=sumlnsupply=0;
    lnpice.clear(); lndemand.clear(); lnsupply.clear();
    timer=0; cost=0;
    for(int i=0;i<24;i++)buy_new_energy[i]=0;
    penalty=0;
    while(pq.size())pq.pop();
}

int conv(char urgency) {
    if (urgency == 'H') return 1;
    if (urgency == 'M') return 2;
    if (urgency == 'L') return 3;
    assert(false);
    return 42;
}

struct node {
    double price,reserve; int per;
};

/*
    All energy calculated in the code is based on kWh
    RGPS in real green power supplied
*/
void init() {
    timer = cost = 0;
}

void input() {
    L(i,0,24*maxt){
        cin>>DH[i]>>DM[i]>>DL[i];
    }
    L(i,0,24*maxt){
        cin>>RGPS[i];
    }
    L(i,0,24*maxt){
        cin>>RGPP[i];
    }
}

```

```

}

double square(double x) {
    return x * x;
}

double expect_new_en_price(int cur_time){
    cur_time %= 24;
    return max((double) 0.2, 0.147863 * sin(0.312526 * cur_time + 1.38352) + 0.47101);
}

const double scale_1 = 1;
const double scale_2 = 147000;
const double shape = -0.29; // # should be negative in the final eq
const double pos = 17.6;
const double sd_skew = 10.3;
const double mean_skew = 8.9;

double normal(double x) {
    return exp(-0.5 * square((x - mean_skew) / sd_skew)) / sd_skew / sqrt(2.0 * acos(-1));
}

double skew(double x) {
    return 0.5 * (1.0 + erf(x / sqrt(2.0)));
}

double expect_en_demand (int x) {
    x %= 24;
    return scale_2 * (2.0 / scale_1) * normal((x - pos) / scale_1) * skew(shape * ((x - pos) / scale_1));
}

double expect_supply(int cur_time){
    cur_time %= 24;
    return max((double) 0.2,(21.58476*sin(0.0585206*cur_time+0.814195)-18.39532));
}

double predict_en_demand(int cur_time){
    // this is a function that predicts the energy demand
    double expected_demand = expect_en_demand((cur_time) % 24);
    // assert(expected_demand > eps);
    if((int)lndemand.size()==0) return expected_demand;
    double demand = expected_demand * exp(sumlnd / (int)lndemand.size());
    return demand;
}

double predict_en_supply(int cur_time){
    // this is a function that predicts the energy demand
    double expected_supply = expect_supply((cur_time) % 24);
    // assert(expected_supply > eps);
    if((int)lnsupply.size()==0) return expected_supply;
    double supply = expected_supply * exp(sumlnsupply / (int)lnsupply.size());
    return supply;
}

```

```

}

double predict_demand_for_whole_day(int cur_time){
    double total_demand=0;
    for(int i=cur_time+1;i<=cur_time+24;i++){
        total_demand+=predict_en_demand(cur_time);
    }
    return total_demand;
}

double predict_supply_for_whole_day(int cur_time){
    double total_supply=0;
    for(int i=cur_time+1;i<=cur_time+24;i++){
        total_supply+=predict_en_supply(cur_time);
    }
    return total_supply;
}

double predict_new_en_price(int cur_time){
    double expected_price = expect_new_en_price((cur_time) % 24);
    // assert(expected_demand > eps);
    if((int)lnprice.size()==0) return expected_price;
    double price = expected_price * exp(sumlnprice / (int)lnprice.size());
    return price;
}

void new_energy_cost_buying_predictions(int cur_time, int energy_required){
    for(int i=0;i<24;i++)buy_new_energy[i]=0;
    vector<info>new_energy_information;
    for(int i=1;i<=24;i++){
        new_energy_information.push_back({expect_new_en_price((cur_time+i)%24),expect_supply((cur_time+i)%24)});
    }
    sort(new_energy_information.begin(),new_energy_information.end());
    // supply represented in unit: MW
    for(auto [price,supply,idx]: new_energy_information){
        int energy_can_get=supply*1000;
        energy_required-=energy_can_get;
        buy_new_energy[idx]=1;
        if(energy_required<=0) break;
    }
    return;
}
double trad_en_price(int cur_time){
    // use original
    return traditional_energy_costs[cur_time];
}

void update_price(int cur_time, double real_price){
    double expected_price=expect_new_en_price(cur_time);

```

```

lnprice.push_back(log(real_price/expected_price));
sumlnprice+=lnprice.back();
if(lnprice.size()>24){
    sumlnprice-=lnprice.front();
    lnprice.erase(lnprice.begin());
}
}

void update_demand(int cur_time, double real_demand){
    double expected_demand=expect_new_en_price(cur_time);
    lndemand.push_back(log(real_demand/expected_demand));
    sumlnd+=lndemand.back();
    if(lndemand.size()>24){
        sumlnd-=lndemand.front();
        lndemand.erase(lndemand.begin());
    }
}
void update_supply(int cur_time, double real_supply){
    double expected_supply=expect_supply(cur_time);
    lnsupply.push_back(log(real_supply/expected_supply));
    sumlnsupply+=lnsupply.back();
    if(lnsupply.size()>24){
        sumlnsupply-=lnsupply.front();
        lnsupply.erase(lnsupply.begin());
    }
}

bool buy_green(double& real_green_energy){
    // determine whether we should buy the green energy
    return real_green_energy<traditional_energy_costs[0];
}

void buy(double& init, double& delta_energy){
    // simulate the action of buying green energy
    init+=(delta_energy);
}

double calculate(int cur_time, Heruristic h){
    if(cur_time<h.p) return 0;
    double late=cur_time-h.p+1;
    return cost_func(late,h.urge);
}
/*
Notes:
1. In the same period of time in different days, assume that the price of traditional
*/
void solve() {
    double initial_energy=0;
    for(int i=0;i<maxt*24;i++){
        double real_energy_needed=DH[i]+DM[i]+DL[i];

```

```

        double real_green_energy_supplied=RGPS[i];
        real_green_energy_supplied*=1000; // Compute the amount of green energy supplied
        double real_green_energy_price=RGPP[i];
        if(buy_green(real_green_energy_price)){
            buy(initial_energy,real_green_energy_supplied);
            // buy if green energy is cheaper than the cheapest traditional energy
        }
        for(int j=1;j<=DH[i]/80;j++)pq.push({i+1,4,80});
        for(int j=1;j<=DM[i]/50;j++)pq.push({i+2,2,50});
        for(int j=1;j<=DL[i]/30;j++)pq.push({i+3,1,30});
        double rem=capacity;
        while(pq.size()){
            Heruristic cur=pq.top();
            if(cur.energy<=rem){
                rem-=cur.energy;
                penalty+=calculate(i,cur); pq.pop();
            }
            else{
                break;
            }
        }
        initial_energy-=real_energy_needed; // consume energy
        global_minimum=min(global_minimum,initial_energy); // Update the minimum energy
        if(i%24==23){
            initial_energy=0; // This is to buy energy until the initial_energy is back to zero
        }
        // cout<<fixed<<setprecision(6)<<-global_minimum*1.5<<'\\n'; // Reserve a little bit more
        cout<<fixed<<setprecision(6);
        string latex=" \\\\";
        cout<<penalty<<" & "<<pq.size()<<latex<<'\\n';
        cout<<"\\hline\\n";
    }

int32_t main() {
    ios::sync_with_stdio(0); cin.tie(0);
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    int test = 1;
    cin >> test;
    while (test--) {
        init();
        input();
        for(int i=0;i<=14500;i+=500){
            capacity=i;
            cout<<i<<" & ";
            solve();
            clear_data();
        }
    }
}

```

}