Table of Contents

Question 1.

This problem asks us to find the maximum bytes processable given a schedule of data $x_1, x_2, \ldots x_n$ and computer efficiency $s_1, s_2, \ldots, s_n$ where s_i is the efficiency i days after a reboot. The subproblems that we will be using to solve for day n is the solutions for day $n-1$ or $n-2$. This is because on each day we have two options, we can continue from the previous day, or reboot the computer, not processing anything but resetting our efficiency. Therefore, we can either:

1. use the optimal solution for yesterday and continue processing today
2. we use the optimal solution for before yesterday, reset yesterday, and process today with optimal efficiency

This works because in an optimal processing sequence, we must have the optimal amount processed in sub sequences with respect to our choice of resetting or not. This is because if our previous options are not optimal, then we can find a better way to process the previous days, which would increase our overall bytes processed and give us a better solution.

$$\text{Opt}(i) = \begin{cases} (0,\ 0) & \text{if } i = 0 \\ \big(\min(x_1,\ s_1),\ 1\big) & \text{if } i = 1 \\ \Big(\begin{array}{l} \min(x_i,\ s_{\text{Opt}(i-1)[1]+1}) + \text{Opt}(i-1)[0], \\ \text{Opt}(i-1)[1] + 1 \end{array} \Big) & \text{if } \min(x_i,\ s_{\text{Opt}(i-1)[1]+1}) + \text{Opt}(i-1)[0] > \min(x_i,\ s_1) + \text{Opt}(i-2)[0] \\ \big(\min(x_i,\ s_1) + \text{Opt}(i-2)[0],\ 1\big) & \text{if } \min(x_i,\ s_{\text{Opt}(i-1)[1]+1}) + \text{Opt}(i-1)[0] \leq \min(x_i,\ s_1) + \text{Opt}(i-2)[0] \end{cases}$$

In the recurrence formula, we return 2 values represented as a tuple: the optimal total amount processed for day i, and how many days it has been since the last reset. This allows us to track resets by checking if the value for day i is 1, signifying there was a reset the day before. We have a base value of (0, 0) to represent that with an empty list we do nothing. Then on our first day, the best option would be to process the data, since resetting would mean processing 0 total data and as our data is non-negative processing it would give us a larger total. The following two options return either yesterdays optimal added to processing todays data with the next efficiency, or before yesterdays optimal added to processing today with the best (initial) efficiency depending on which is greater.

Proof of correctness:

Base Cases:

Let $n = 0$.

Since we have no days, we have no data, and therefore we can only return (0, 0).

Let $n = 1$.

Since we only have one day, our best option is to process data, as our only other option is to reboot, but then we would not be able to process any data and have a total amount of data processed of 0. Since the amount of data we can process is ≥ 0, processing data will be equal to if not better than rebooting, so we return $\big(\min(x_1, s_1), s_1\big)$

Induction Step:

Let $n \geq 1$.

Suppose Opt(i) holds whenever $0 \leq i < n$.

return

$$\max\Big(\big(\min\big(x_i, s_{Opt(i-1)[1]+1}\big) + Opt(i-1)[0],\ S_{opt(i-1)[1]+1}\big),\\ (\min(x_i, s_1) + Opt(i-2)[0],\ 1),\\ key = (e) => e[0]\Big)$$

We know Opt($i-1$), Opt($i-2$) return the optimal values for day $i-1$ and $i-2$ respectively by the induction hypothesis. Therefore, to increase our total amount of data processed, we have to process data today. Therefore we can either continue processing data without rebooting, or process with the best efficiency by rebooting yesterday. Since we pick the maximum of those two options, we return the maximum amount of data processed overall.

$\therefore$ By induction, Opt(i) holds for all $i \geq 0$. ■

Time complexity:

This algorithm is $O(n)$ as there is only one loop which is at line 15 which runs for the length of the input array + 1, and inside that loop we only do $O(1)$ operations line min, addition, array access and array pushes.

Space complexity:

This algorithm has space complexity $O(n)$, as the only variable that changes size with the input is the process array, and it holds $n + 1$ elements which is $O(n)$. All other variables are primitives or data structures with primitive attributes.

Since in our algorithm we return a tuple, we created a struct to represent that called ProcessProgress. The first element represents the optimal total amount processed for the corresponding day, and the second element represents the amount of days since last reset. On line 6 and 8 we have initial return statements because the answers to the base case of the recurrence relation are immediately knowable.

The recurrence part works bottom up by filling in the values of the table from the start and then returning the last index's value. We store all the values in the progress array. We appropriately fill in the base cases on lines 12 and 13. Since we have an 'extra day' (empty list day), we loop from 2 (past our base case) to the length of terrabytes array + 1 and index the terrabytes with $i - 1$ since it is 0 indexed unlike our recurrence relation data which is 1 indexed. On line 16 we calculate the option where we reboot yesterday, and on line 20 we calculate the option where we continue processing without a reboot. In lines 25-28 we push the larger of the two options to the progress array. Finally we return the the final index's max attribute, which is the optimal total amount processed for the entire sequence.

Algorithm

Process_Terabytes (`terabytes`: array of terabytes to process, `decay`: array of computation power)

```
1: struct ProcessProgress {
2:     max: int,
3:     runtime: int
4: }
5:
6: if terabytes.len == 0 then
7:     return 0
8: if terabytes.len == 1 then
9:     return min(terabytes[0], decay[0])
10:
11: progress = []
12: progress[0] = ProcessProgress {max:0, runtime:0}
13: progress[1] = ProcessProgress {max:min(terabytes[0], decay[0]), runtime:1}
14:
15: for i in 2..terabytes.len()+1 do
16:     skip_yesterday = ProcessProgress {
17:         max:  progress[i-2].max + min(terabytes[i-1], decay[0]),
18:         runtime:  1
19:     }
20:     continue = ProcessProgress {
21:         max:  progress[i-1].max + min(terabytes[i-1], decay[progress[i-1].runtime]),
22:         runtime:  progress[i-1].runtime + 1
23:     }
24:
25:     if skip_yesterday.max > continue.max then
26:         progress.push(skip_yesterday)
27:     else
28:         progress.push(continue)
29:
30: return progress[-1].max
```

Question 2.

This problem asks us to find the optimal amount of stocks to sell on each day given a certain amount of stocks, the prices over the days, and the decay function for how much the prices decrease by for each stock sold. The subproblems to solve this algorithm are how much stock needs to be sold on previous days. This is because for each stock added, we can either sell it today or on a previous day. And for each day added, if we have the optimal profit if a certain amount of stocks are sold on a certain day, we can check if we can sell them better using the extra day or not. The optimal profit for all the stocks using all the days, is calculated using the optimal amount of stocks sold on each day.

$\text{Opt}(\texttt{day}, \texttt{stock}) =$

$$\begin{cases} (0,\ 0,\ 0) & \text{if } \texttt{day} = 0 \\ & \text{or } \texttt{stock} = 0 \\ \Big(& \\ \quad \max\limits_{0 \le \texttt{best} \le \texttt{stock}} & \text{Opt}(\texttt{day} - 1,\ \texttt{stock} - \texttt{best})\texttt{[0]} + \\ & \texttt{best} \times \big(p_{\texttt{day}} - \text{Opt}(\texttt{day} - 1,\ \texttt{stock} - \texttt{best})\texttt{[1]} - f(\texttt{best})\big), \\ & \text{Opt}(\texttt{day} - 1,\ \texttt{stock} - \texttt{best})\texttt{[1]} + f(\texttt{best}), \\ & \texttt{best}, \\ \Big) & \end{cases} \quad \text{otherwise}$$

The recurrence relation uses a tuple: the first value is the optimal profit possible for selling the stocks given the number of days. This is calculated by finding the best value to sell on that day, having sold the rest on a previous day. So we can partition the shares into the ones sold today and the ones sold before. This allows us to use the sub problems to find the best amount to sell by finding the amount that would maximize our profit from today's sale added to the optimal profit from selling the rest of the shares on the previous days. The second value is the total amount our price has been reduced by because of all the shares sold so far. The third value is the amount of shares that would give us the optimal value in the first position of the tuple.

Proof of correctness:

Base Cases:

Let $\texttt{day} = 0$.

Since we have no days, we cannot sell any of our stocks so our profit will be 0 so we return (0, 0, 0).

Let $\texttt{stock} = 0$.

Since we have no stocks, we cannot sell anything on any day so our profit will be 0 so we return (0, 0, 0).

Induction Step:

Let $\texttt{day} \geq 0,\ \texttt{stock} \geq 0$.

Suppose $\text{Opt}(i,\ j)$ holds whenever $0 \leq i < \texttt{day},\ 0 \leq j < \texttt{stock}$.

return

$$
\begin{pmatrix}
\max\limits_{0 \leq \texttt{best} \leq \texttt{stock}} & \text{Opt}(\texttt{day} - 1,\ \texttt{stock} - \texttt{best})\texttt{[0]} + \\
& \texttt{best} \times \big(p_{\texttt{day}} - \text{Opt}(\texttt{day} - 1,\ \texttt{stock} - \texttt{best})\texttt{[1]} - f(\texttt{best})\big), \\
\text{Opt}(\texttt{day} - 1,\ \texttt{stock} - \texttt{best})\texttt{[1]} + f(\texttt{best}), \\
\texttt{best}, \\
\end{pmatrix}
$$

If we have i days and j stocks, the best amount to sell on day i is the amount that maximizes our profit if we sell it today and have sold the remaining on a previous day. Therefore, we can partition our stocks into those sold today and those sold on another day. The best amount to sell today is the amount so that the profit from today's sale and the previous sales of all the stocks are maximized. Since we return this value, we return the optimal profit possible on i days and j stocks.

$\therefore$ By induction, $\text{Opt}(i, j)$ holds for all $i \geq 0,\ j \geq 0$. ■

Time complexity:
Opt($i,\ j$) is $O(ij^2)$ where i is the number of days and j is the number of stocks. The algorithm has 2 for loops on line 9 and 25. The loop on line 25 runs for the amount of days so its $O(i)$. The loop on line 9 is a triple for loop. The outer loop runs for the amount of days, the middle loop runs for the amount of stocks, and the inner loop runs as much as the index of the middle loop. The middle and inner loop together run $\sum_{i}^{n} i = \frac{n(n+1)}{2} \in O(n^2)$. Combined with the outer loop, the total is $O(ij^2)$. Inside the inner loop only $O(1)$ operations are done like array accesses, addition and comparisons. Since the decay function is given as a list of values, we also know that calling it is $O(1)$ so the total of the inner loop is $O(1)$. Therefore the time complexity is $O(i) + O(ij^2) \in O(ij^2)$.

Space complexity:
Opt($i,\ j$) uses $O(ij)$ space where i is the number of days and j is the number of stocks. This is because all the variables used are primitives or data structures with primitive attributes except the `profit_table` array and `optimal_sold_per_day` array. The `profit_table` array stores the data structure of the optimal profit for selling j shares on up to i days, which can be used to find out how much shares should be sold on which day for optimal profit. Since this is a 2d array with dimensions $i + 1$ and $j + 1$ so it uses $O(ij)$ space. The `optimal_sold_per_day` array backtraces through the `profit_table` and stores the amount optimal amount of shares sold per day. It is $i + 1$ elements long and therefore uses $O(i)$ space. Therefore the algorithm uses $O(ij) + O(i) \in O(ij)$ space.

Algorithm

Sell_Stocks (`stocks`: number of stocks, `model`: model of stock market, `stock_decay`: function)

```
1: struct OptimalStockInfo {
2:     optimal_profit: int,
3:     stock_decay: int
4:     stocks_sold: int
5: }
6:
7: profit_table = [[OptimalStockInfo {0, 0, 0}× (stocks + 1)] × model.len() + 1]
8:
9: for day in 1..=model.len() do
10:     for max_stock in 0..=stocks do
11:         for stock in 0..=max_stock do
12:             price_drop = profit_table[day-1][max_stock-stock] + stock_decay(stock)
13:             profit = profit_table[day-1][max_stock-stock] +
14:                             stock×(model[day-1] - price_drop)
15:
16:             if profit > profit_table[day][max_stock].optimal_profit then
17:                 profit_table[day][max_stock] = OptimalStockInfo {
18:                     optimal_profit: profit,
19:                     stock_decay: price_drop,
20:                     stocks_sold: stock
21:                 }
22:
23: stocks_left = stocks
24: optimal_sold_per_day = [0 * model.len() + 1]
25: for day in 0..model.len() do
26:     optimal_sold_per_day[model.len() - day] =
27:                         profit_table[model.len() - day][stocks_left].stocks_sold
28:     stocks_left −= optimal_sold_per_day[model.len() - day]
29:
30: return optimal_sold_per_day
```