University of Toronto at Scarborough

CSCC73 - Algorithm Design and Analysis

Assignment #5: Dynamic Programming 2

Due: Feb 25, 2023 at 11.59pm This exercise is worth 5% of your final grade.

Warning: Your electronic submission on MarkUs affirms that this exercise is your own work and no one else's, and is in accordance with the University of Toronto Code of Behaviour on Academic Matters, the Code of Student Conduct, and the guidelines for avoiding plagiarism in CSCC73. Late assignments will not be accepted. If you are working with a partner your partners' name must be listed on your assignment and you must sign up as a "group" on Gradescope. Recall you must not consult **any outside sources except your partner, textbook, TAs and instructor**.

1. Suppose that you have a computer that can process a certain amount of data each day and further that the more consecutive days that the computer runs, the worse it's performance. For each of n days, you are given x_i terabytes of data to process on day i. For each terabyte of of data processed you receive a fixed amount of revenue but any unprocessed data becomes unavailable at the end of the day (so you cannot process it the next day).

   Since you are limited by the capabilities of your computer which decreases with each passing day since the last reboot, you want to find a schedule to process the data and reboot the machine to maximize the revenue. For your computer you know the amount of data it can process j days after the last reboot. So for example, after a reboot, the next day it can process s_1 terabytes of data. On the second day after reboot, it can process s_2 terabytes of data etc. We assume that $s_1 > s_2 > \ldots > s_n > 0$. To return to a processing power of s_1 you can reboot, but on that day, no data will be processed.

   Example.

   Suppose $n = 4$ and the values of x_i and s_i are given below:

	Day 1	Day 2	Day 3	Day 4
x	10	1	7	7
s	8	4	2	1

   For this example, the best schedule would be to reboot on day 2 since then $8+0+7+4 = 19$ terabytes would be processed whereas if no reboot occurred then only $8+1+2+1 = 12$ terabytes would have been processed.

   Formally, given a schedule of data $x_1, x_2, \ldots, x_n$ for n days and efficiency of your computer $s_1, s_2, \ldots, s_n$ where s_i is the processing power j days *after* a reboot, give an efficient algorithm that takes the values and returns the number of terabytes processed by an optimal solution. Be sure to justify the correctness of your algorithm as discussed on piazza for dynamic problems and give the complexity of your algorithm.

   Solution.

   We have more than one possible way to approach this problem. One of them is the following:

   Let $Opt(i, j)$ denote the max amount of work that can be done starting from day i through to day n, given that the last reboot was j days prior, so the system was rebooted on day $i - j$.

   We know that on each day we have two choices, reboot or continue processing. If we:

- Reboot: then we don't process any data on day i but we reset the reboot count so the first day after a reboot is day $i+1$. This gives

$$Opt(i,j) = Opt(i+1,1)$$

- Keep Processing: which means we increase the number of days since last reboot and process the minimum of x_i and s_j,

$$Opt(i,j) = min\{x_i, s_j\} + Opt(i+1, j+1)$$

On the last day, there is no advantage gained in a reboot so $Opt(n,j) = min\{x_n, s_j\}$.

We can define our algorithm to compute $Opt(i,j)$ as:

```
for j: 1 .. n:
    $Opt(n,j) = min\{x_n, s_j\}$
for i: n-1 .. 1:
    for j: 1 .. i:
        $Opt(i,j) = max\{Opt(i+1,1), min\{x_i, s_k\} + Opt(i+1, j+1)\}$
return Opt(1, 1)
```

The running time is proportional to the number of values being calculated (since each value takes constant time). Since there are $O(n^2)$ values, the time is $O(n^2)$.

2. Question 15 from the text, given below.

Consider the problem faced by a stockbroker trying to sell a large number of shares of stock in a company whose stock price has been steadily falling in value. It is always hard to predict the right moment to sell stock but owning a lot of shares in a single company adds an extra complication: selling many shares in a single day will have an adverse effect on the price.

Since future market prices, and the effect of large sales on these prices, are very hard to predict, brokerage firms use models of the market to help them make such decisions. In this problem, we will consider the following simple model. Suppose we need to sell x shares of stock in a company, and suppose that we have an accurate model of the market: it predicts that the stock price will take the values $p_1, p_2, \ldots, p_n$ over the next n days. Moreover, there is a function $f(\cdot)$ that predicts the effect of large sales: if we sell y shares on a single day, it will permanently decrease the price by $f(y)$ from that day onward. So, if we sell y_1 shares on day 1, we obtain a price per share of $p_1 - f(y_1)$, for a total income of $y_1 \cdot (p_1 - f(y_1))$. Having sold y_1 shares on day 1, we can then sell y_2 shares on day 2 for a price per share of $p_2 - f(y_1) - f(y_2)$. This yields an additional income of of $y_2 \cdot (p_2 - f(y_1) - f(y_2))$. This process continues over all n days.

Design an algorithm that takes the prices $p_1, \ldots, p_n$ and the function $f(\cdot)$ (written as a list of values $f(1), f(2), \ldots, f(x)$) and determines the best way to sell x shares by day n. In other words, find natural numbers $y_1, y_2, \ldots, y_n$ so that $x = y_1 + \cdots + y_n$ and selling y_i on day i, $1 \le i \le n$, maximizes the total income achievable. You should assume that the share value p_i is monotone decreasing and $f(\cdot)$ is monotone increasing, ie., selling a larger number of shares causes a larger drop in the price. Your algorithm's running time can have a polynomial dependence on n (the number of days), x (number of shares) and p_1 (peak price of the stock). Be sure to justify the correctness of your algorithm as discussed on piazza for dynamic problems and give the complexity of your algorithm.

Example.

Let $n = 3$, $(p_1, p_2, p_3) = (90, 80, 40)$ and $f(y) = 1$ for $y \le 40{,}000$ and $f(y) = 20$ for $y > 40{,}000$. Assume you start with $x = 100{,}000$ shares. Selling all of them on day 1 would yield a price of 70 per share. For a total income of $7{,}000{,}000$. On the other hand, selling $40{,}000$ shares on day 1 yields a price of 89 per share, and selling the remaining 60,000 shares on day 2 results in a price of 59 per share for a total income of 7,100,000.

Sample Solution.

The key to this problem is noticing that the subproblems are defined by looking at the amount of stock, residual price for the *remaining* days (as opposed to the days completed).

We define the subproblems to be the profit generated with k stocks starting from day i onwards. This works from the following observations:

1. From the i^{th} day on, the selling strategy does not depend on how selling was done on the previous days because each of the remaining k stocks will incur the same constant penalty in profit due to the accumulated value of the function f from previous sales. This means our total profit will just decrease by k times that constant.

2. If we sell y stocks on the i^{th} day then all the k stocks will incur a profit loss of $f(y)$ when they are sold (even in the future).

Using these two observations, we can define the following recurrence:

$$Opt(i,k) = \begin{cases} 0 & k = 0 \\ max_{y \leq k}\{p_i y - kf(y) + Opt(i+1, k-y)\} & k \neq 0 \end{cases}$$

$p_i y$ is the profit incurred from selling y stocks with the price p_i, $kf(y)$ is the loss in price due the quantity y sold which is applied to *all* k stocks. The last term is the recursive optimal solution to selling the left over stock starting from day $i+1$.

To implement this we define a two dimensional table M, whose rows would stand for the days and the columns for the stocks to sell. We fill the matrix in by using the above recurrence from teh bottom row to the top one. The resulting algorithm is cubic. Our desired goal is $Opt(1,x)$. We also store amount of stock sold on the i^{th} day in our optimal solution giving the desired output.

The running time is dependent on x the total number of stocks. This is polynomial in the input size, the values of the function f are given as a set of x values $f(1), \ldots, f(x)$.