Victoria (Tori) Girkins
Clara Tran
Stacia Near
11/2/17

Algorithms Dynamic Programming Project

(a) [40 pts] Theory: Devise a dynamic programming algorithm that optimizes the number of terabytes processed. Demonstrate each of the following dynamic programming steps:

i. [20 pts] Describe a recursive solution to the problem. This can take the form of a mathematical expression or a recursive algorithm, but should be accompanied by an explanation.

Equation:

$$OptData(k) = \max_{i=k+1 \to n} [\sum_{j=k+1}^{i-1} (\min(x[j], s[j]) + OptData(i))]$$

Explanation:

We use recursion to solve the problem of when to reboot. In the problem statement, we are given two vectors as input.

Vector x gives the number of data processed each day, where index 0 is day 1, and so on.

Vector s gives the number of data that can be processed on each day after a reboot, so its indexing is a bit different. Instead of index 0 just being day 1, it actually refers to the 1st day after a reboot. So the pointer for the "current" index on vector s must be reset every reboot. On the other hand, vector x will just increment by one day at a time no matter what.

To create our recursive equation, we visualized the problem as pictured below. R means reboot. Day 0 is always a reboot, but we will store it in the table for reasons you will see below.

k is defined as the day of the last reboot.

Day 0	1	…	k	…	n
R			R		

We realized we could save space by storing the optimal results after a given day k, assuming all reboots before that are optional, and recursing back through the equation until we know all optimal reboots.

For example, lets say that after day 5, which we will designate as k, the best possible results you can get is 56 points of data processed:

Day 0	1	2	3	4	5	k=6	…
R	R?	??				56→	##

This answer could be relevant for many different situations, so we should store it in the table because whatever reboot configuration is set up to the RIGHT of day k (after day k) will be optimal no matter what happened earlier.

For instance, if we rebooted with this pattern:

Day 0	1	2	3	4	5	k=6	…
R		R			R	56	##

Or if we rebooted with this pattern:

Day 0	1	2	3	4	5	k=6	…
R	R		R			56	##

Even though the reboot configuration before k is different, everything after k remains the same.

This allows us to simplify the problem, only focusing on one k at a time. Using our recursive equation, we will find the optimal series of reboots, assuming we reboot on day k. The resulting amount of data processed will be the optimal number. We will store this in our table of intermediate results.

The intermediate results will be formatted in a table as follows:

k	0	1	2	3	4	…	n
Optimal amt data processed	56						
Day of next reboot	2		4		10		

The table is two dimensional – two vectors stacked on top of each other. The top row will simply be the indices of the array. The number of columns will equal the number of days, n.

In the example above, everything would be filled with numbers by the end, but for clarity I only show the relevant parts.

By the time the program is done running, it comes up with 56 as the optimal amount of data processed. This will always be displayed in the slot for day 0, because day 0 is always the first reboot. Since the definition for that cell is “the optimal amount of data processed assuming the first reboot is on day 0”, that must be the overall answer.

Let us run this algorithm on the example by hand, to see how it works in depth.

Here is the equation again:

$$OptData(k) = \max_{i=k+1 \to n} [\sum_{j=k+1}^{i-1} (\min(x[j], s[j]) + OptData(i))]$$

As explained above, this equation will provide a recursive solution to which days are best to reboot on, by simplifying the problem.

The equation finds the maximum amount of data processed after the last reboot k. So, from the next day k+1 to the last day n.

This means the equation will start on the left, but when we get to the bottom of the recursion we will be evaluating back from right to left as we step back up the chain of function calls.

When OptData(k) is called, nothing to the left of k is considered. We only care how much data is processed after that reboot k, and what the sequence of reboots after k was.

To get the maximum amount of data processed, we must first evaluate the right hand side of the equation, where the sum is. The sum is simply going over all the days "j" and summing the amount of data processed on each day.

We take the min of x and s on that day, because the amount of data we process is only as much as our "weakest link." Either we don't have enough data to reach the upper limit s(j) so we process x(j) units of data, or we have too much data, so we are forced to be limited by our processing ability s(j). Remember that in the code, we will have to reset which s(j) we are considering at a given time, because after a reboot our upper limit will increase to the s(j) we get on day 1.

In order to perform the recursion, we now add to that minimum for our current day by calling OptData(i) again. This will find the max stats for the day after k. And then, on the next call, for the day after that!

So, in the end, we will eventually work our way all the way to the right, we can find the optimal data processed for the base case once we hit the end, and then we can work our way back up, storing the optimal case for every single possible k.

However, in the end, we will have to figure out which days k we rebooted on the get the optimal solution. This is actually quite easy because of how our table is structured. We simply perform a traceback based on the answer we get for Day 0.

Using the example above again, we have the following table. The values will all be filled in at the end.

However, we will not use some of them. We are only interested in the days k that represent the optimal reboot days that we actually used to get the best and final answer.

k	0	1	2	3	4	…	n
Optimal amt data processed	56	45	20	17	15		##
Day of next reboot	2	3	4	5	10		##

This can be achieved very simply. We just look at the day of the next reboot in the table under day 0. This means that to get the optimal amount of data processed, 56, we have to have our next reboot on day 2.

We can actually go through the entire table just like this. Now that we know our next reboot is day 2, we simply check the cell for day 2. The next optimal reboot is day 4! If we continue we will get the sequence {2,4,10...etc} as our sequence for what days to reboot on.

This concludes the explanation of the equation. For more thorough understanding, we have included the pseudo code below.

ii. *[10 pts] Write pseudocode for a dynamic programming algorithm. As discussed in class, this can use loops or recursion. Either way, the idea is that you are storing intermediate results (describe the data structure you are using to store these).*

DpTable is an integer array with two rows: the first is the optimal amount of data we can process after this day (index) assuming optimal
strategy, and the second is the next day on which to reboot

```
OptData(dayOfLastReboot, DpTable)
    If DpTable has an entry at index k, return this.
    Else if dayOfLastReboot is the third to last day
        -Check to see if we would be better off rebooting or not
        rebooting on the second to last day (never reboot on last day)
        -Store our decision and the amount of data processed as a result
        in DpTable
    Else
        For every remaining day after dayOfLastReboot
            -Add up the data we can process between
            dayOfLastReboot and this day, and OptData(this day,
            DpTable)
            -If this is the maximum make note of this day and the
            amount of data
        -Also see how much data you could process if you didn't reboot
        anymore and make note if this is the maximum
        -Store the max data amount we found into DpTable, and the next
        day on which to reboot (or NULL if don't reboot any more)
```

We call this function by passing it 0 for dayOfLastReboot and an empty table of size 2 by (n+1) where n is the number of days.

iii. *[10 pts] Develop a traceback algorithm that returns the decision on each day (i.e.,process vs reboot).*

(b) [10 pts] Theory: Derive the complexity of your algorithm in terms of n.
The problem has been adapted from the text by Kleinberg and Tardos
The algorithm is O(n). It uses a 2 by (n + 1) table of entries, which is O(n). To generate the entries, the algorithm recurses further and further down until it gets to the base case, at which point it calculates this entry in constant time (see base case in pseudocode above). It then starts backing up the stack trace and calculating entries further and further back in the table, which once you know the table are just sums which run in constant time. So in short, we have O(n) entries that can each be calculated in O(1) time for an overall O(n) algorithm.

(c) [15 pts] Implementation: Implement your algorithm and include your (well-written and documented) code. If you were unable to get your code to compile/run, please state this clearly. We may choose a few groups randomly and ask them to demonstrate that their code works. (It would look pretty bad if you claim your code runs, but it doesn't!)

See main.cpp
(d) [10 pts] Implementation: Demonstrate that your code works correctly by showing its results on the following instance.

Day 1 Day 2 Day 3 Day 4 Day 5 Day 6 Day 7 Day 8 Day 9 Day 10
x 20 80 20 60 20 60 80 10 40 10
s 100 90 50 45 40 35 20 15 10 5

Your output should consist of two lines. The first line gives the total amount of data processed and the second line lists the amount of data processed on each day (with a 0 for a reset). For the small example given earlier, the output would be
19
8 0 7 4

365
20 80 20 45 0 60 80 10 40 10