Dynamic Programming Project

C. Travis Johnson
Zac McClain
Brandon Parrish

November 3, 2015

1 Problem

Given $x_1, x_2, \ldots, x_n$ TB of available data for the next n days and given the amount of data a server can process $s_1, s_2, \ldots, s_n$ for n days after a fresh reboot (in TB).

Goal Choose the days on which you are going to reboot so as to maximize the total amount of data you process.

2 Dynamic Programming Algorithm

2.1 Main Idea (20 pts)

Breaking Problem Into Sub-problems For days 0 to n, choose to restart on day d such that $max(f(d))$ where $f(d) = P(0, d-1) + P(d+1, n)$. Find the amount of data proceesed $P(i, j)$ for a range of days $[i, j]$ provided you start with a fresh server (s_1) on day i. For the right partition (days $d+1$ to n), we only need to keep track of the optimal value of data processed and the day d on which we choose to reboot. Now for the left partition (days 0 to $d-1$), repeat the same process of choosing the ideal day to reboot. This process will end when the length of the left partition, the number of days of data left to process is 1 or 0 (when rebooting will not increase the amount of data processed).

Calculating $P(i, j)$, the Amount of Data Processed without rebooting from days i to j On each day, decide whether the amount of data the server processes is limited by the amount of available data or the processing capability of the server. Return the sum of the limiting factors (available data or power) across days i through j as $P(i, j)$.

Don't Repeat Calculations, The Essence of Dynamic Programming As we process the values of $P(i, j)$ for various $[i, j]$, we will populate a results matrix (two-dimensional array) to avoid repetitive calulations of both $P(i, j)$ and $P(a, b)$ where $i = a$ and $j = b$. This matrix will need to be of size $n+1$ rows by n columns and will be initialized with a row of zero values which represent the amount of data processed on day d if we reboot on day d.

2.2 Pseudocode (10 pts)

```
# X represents the sequence of x_i values indexed from 1.
# S represents the sequence of s_j values indexed from 1.
def main():
    readInput() # populate X and S lists from given input
    # P (results matrix) will be of size n+1 by n (rows by columns)
    row_init()  # initialize zeroth row of P (results matrix) with 0s
    GetMaximumProcessed(X, S)

def GetMaximumProcessed(X, S):
    # Begin Populating table P
    # Days 0 to d represents the left partition.
    # Day d + 1 represents the partition day.
    # Day d + 2 to n (length of X) represents the right parition.
    for d in range(length of X):
        for j in range(length of S):
            # ensure do not waste time computing values...
            # ...that have already been computed
            # ...that are impossible to use (i.e. x_1 data processed using s_2)
            P[d][j] = Min(X[d], S[j])
            # P[d][j] depends on P[j - 1][d - 1]
            # ...because we build table P from left to right
            if P[j - 1][d - 1] is a valid cell with value 0,
                # the max of the left partition is in the d - j - 1 column
                then P[d][j] += the maximum of the d - j - 1 column.
            else if  P[j - 1][d - 1] is a valid cell
                # there are columns remaining in the left portion of the table P
                then P[d][j] += P[j - 1][d - 1]
    return max value of last column in P as optimum amount of data processed
```

2.3 Traceback Algorithm (10 pts)

Report Path to Optimum Find the goal cell (the maximum value in the results table P) which will be in the right-most column. From the right-most column in P, get the row index of the max value in that column. Using the indices of the max value, the day that will have caused that reboot (the day on which we will partition) is the index of the column subtracted by the index of the row (e.g. $column - row$). Add the number of the day that will have caused that reboot to a set tracking the days we will reboot the server. Repeat this process for the columns on the left of the column of the last reboot (the left partition) until we do not have any more days on which to try and reboot. When there are no more days left to make a decision, we report the set of days to reboot and on all other days- we will decide to process the data.

2.4 Time Complexity (10 pts)

To create the results matrix (P), we must build a table of size $n+1$ by n which takes $O(n^2)$ time: we will ignore populating some of these cell values, but this will not directly impact the asymptotic complexity of the algorithm. In addition to building the table, we will occasionally do a linear scan of a column to find its maximum value: this will take $O(n)$ time which again, does not directly impact the aymptotic complexity of the algorithm. Other operations should take constant time, so our **algorithm runs with quadratic time complexity-** $O(n^2)$.

3 Implementation

3.1 Code (15 pts)

Listing 1: "Ruby Implementation"

```ruby
# initialize_table will populate a 2D array (table) with an initial row of 0 values
def initialize_table(given, can)
  table = Array.new(can.length + 1)
    (0...(can.length + 1)).each { |x| table[x] = Array.new(given.length) }
  (0...(given.length)).each { |x| table[0][x] = 0 }
  return table
end

# output the structure of the table to the console
def print_table table
  table.each do |x|
    if x != nil
      x.each { |y| print "#{y} " }
    end
    puts
  end
end

# return limiting factor (amount of data or server processing power)
def min(x, s)
  if x < s
    return x
  end

  return s
end
```

```
# return the maximum value of a specific column in a table
# linear search causes O(n)
def column_max( table, column )
  max = -9999999
  (0...(column + 2)).each do |x|
    if table[x][column] != nil
      if  table[x][column] > max
        max = table[x][column]
      end
    end
  end

  return max
end

# process the data sets X (given) and S (can)
# two iterative loops causes O(n^2)?
def make_table( given, can )
  table = initialize_table(given, can)  # init row of 0s

  # need one more row than columns (n + 1) rows
  can = [0] + can

  (0...(given.length)).each do |x|
    (1...(can.length + 1)).each do |s|

      # do not calculate min for lower diagonals
      # cannot possibly process x_1 with the power of s_2, nor x_2 with s_3
      if s - x < 2

        if x - 1 < 0 || s - 1 < 0
          table[s][x] = min( given[x], can[s] ) # table[1][0] is min( x_1, s_1 )

        elsif s - x >=  0
          table[s][x] = min( given[x], can[s] ) + table[ s - 1 ][ x - 1 ]
          # calculate sum of diagonals

        elsif s == 1

          table[s][x] = min( given[x], can[s] ) + table[ s - 1 ][ x - 1 ] + column_max(table, x - s - 1 )
```

```
            # calculate sum of diagonals + max of the column immediately left of a reboot

          else
            table[s][x] = min( given[x], can[s] ) + table[ s - 1 ][ x - 1]
            # what does this do differently from the second condition?

          end
        end
      end
    end

    print_table table # output the table
    return table
end

make_table( [10, 3, 1, 8, 6], [6, 4, 3, 2, 1] )
```

3.2 Small Example (10 pts)

Given: $X = 10, 3, 1, 8, 6$ and $S = 6, 4, 3, 2, 1$
Construct the table and identify the maximum cell value (19). Because the maximum value (19) is in row index 2, we know **we reboot on day number 3 only** (or the 2nd indexed day counting from 0).

-	10	3	1	8	6
-	0	0	0	0	0
6	6	3	7	15	16
4	-	9	4	11	**<u>19</u>**
3	-	-	10	7	14
2	-	-	-	12	9
1	-	-	-	-	13