

Dynamic Programming Project

C. Travis Johnson

Zac McClain

Brandon Parrish

November 3, 2015

1 Problem

Given x_1, x_2, \dots, 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, \dots, 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

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 processed $P(i, j)$ 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 calculations 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

```
# 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 partition.
    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

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

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 asymptotic 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

Listing 1: "Ruby Implementation"

```
1  # initialize_table will populate a 2D array (table) with an  
   initial row of 0 values  
2  def initialize_table(given, can)  
3    table = Array.new(can.length + 1)  
4      (0...(can.length + 1)).each { |x| table[x] = Array.new(given.  
      length) }  
5    (0...(given.length)).each { |x| table[0][x] = 0 }  
6    return table  
7  end  
8  
9  # output the structure of the table to the console  
10 def print_table table  
11   table.each do |x|  
12     if x != nil  
13       x.each { |y| print "#{y}_" }  
14     end  
15     puts  
16   end  
17 end  
18  
19 # return limiting factor (amount of data or server processing  
   power)  
20 def min(x, s)  
21   if x < s  
22     return x  
23   end  
24  
25   return s  
26 end
```

```

27
28 # return the maximum value of a specific column in a table
29 # linear search causes  $O(n)$ 
30 def column_max( table , column )
31     max = -9999999
32     (0...(column + 2)).each do |x|
33         if table[x][column] != nil
34             if table[x][column] > max
35                 max = table[x][column]
36             end
37         end
38     end
39
40     return max
41 end
42
43 # process the data sets X (given) and S (can)
44 # two iterative loops causes  $O(n^2)$ ?
45 def make_table( given , can )
46     table = initialize_table(given , can) # init row of 0s
47
48     # need one more row than columns (n + 1) rows
49     can = [0] + can
50
51     (0...(given.length)).each do |x|
52         (1...(can.length + 1)).each do |s|
53
54             # do not calculate min for lower diagonals
55             # cannot possibly process x_1 with the power of s_2, nor x_2
56             with s_3
57             if s - x < 2
58
59                 if x - 1 < 0 || s - 1 < 0
60                     table[s][x] = min( given[x] , can[s] ) # table[1][0] is
61                     min( x_1 , s_1 )
62
63                 elsif s - x >= 0
64                     table[s][x] = min( given[x] , can[s] ) + table[ s - 1 ][
65                         x - 1 ]
66                     # calculate sum of diagonals
67
68                 elsif s == 1
69
70                     table[s][x] = min( given[x] , can[s] ) + table[ s - 1 ][
71                         x - 1 ] + column_max(table , x - s - 1 )

```

```

68         # calculate sum of diagonals + max of the column
        immediately left of a reboot
69
70     else
71         table[s][x] = min( given[x], can[s] ) + table[ s - 1 ][
            x - 1]
72         # what does this do differently from the second
        condition?
73
74     end
75 end
76 end
77 end
78
79 print_table table # output the table
80 return table
81 end
82
83 def column_max_index( table , column )
84     max = -9999999
85     index = 0
86     (0...(column + 2)).each do |x|
87         if table[x][column] != nil
88             if table[x][column] > max
89                 max = table[x][column]
90                 index = x
91             end
92         end
93     end
94
95     return index
96 end
97
98 def trace_back(table)
99
100     reboots = Array.new()
101
102     column = table.length - 2
103     column = column - column_max_index(table , column) - 1
104
105     while column >= -1
106         reboots << (column + 1)
107
108         if column == -1
109             break

```

```

110     end
111
112     column = column - column_max_index(table, column) - 1
113
114     end
115
116     return reboots
117 end
118
119 table = make_table( [1,10], [10,1] )
120 puts "Max_amount_of_data_that_could_be_processed: #{column_max(
121   table, table.length-2)}"
p trace_back(table)

```

3.2 Small Example

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