Optimization Problem by Dynamic-Programming algorithm - Summative Assessment

Pietro Zafferani

BSc in Genomics, University of Bologna

May 24, 2021

1 Introduction

The problem we were asked to solve consisted in a so called *optimization problem*, in other words the goal was to find, among all the possible solutions, one for which the unknown's value was maximized.

Suppose to put yourself in the shoes of an employee of a pharmaceutical company selling a product. This product is sold in batches of fixed amounts (cannot be further decomposed in unities), each one with an associated worth, and in particular the company wants to sell n batches of the product. The company asks you to determine the selling plan for the n batches of the product to maximize the revenue.

In this report we proposed a Dynamic Programming-based algorithm that is able to return the optimal solution for any instance of this problem; the pseudocode we are going to show is thoroughly explained, moreover we propose also its Python implementation. Then we will proceed to define an upper bound to the computationally worst case in terms of *big O notation*, testing the algorithm on a series of different inputs to record its time performances and eventually graphically representing its behaviour to check whether our predictions will be confirmed.

2 Combinatorial Problem

In section 1 we saw a brief explanation of the problem's features, however these pieces of information must be processed before being useful for the

construction of the algorithm.

Abstracting the problem at the combinatorial level is essential in order to design a reliable algorithm, in fact we need to start from the clear and precise definition of inputs and outputs:

INPUT: a list $L = [0, 1, \ldots, n]$ of n non repeated elements (batches), where $n \in N$, any $0 \leq k \leq n$ element is associated with a monetary value p_k, and $p_0 = 0$.

OUTPUT: a list $Plan = [k_1, \ldots, k_i]$, in which $\sum k = n$ and for which the sum of the batches' values ($\sum p_k$) is maximum.

3 The Pseudocode

In this section we are going to define and consequently analyze the pseudocode of the algorithm.

We adopted a *Dynamic Programming* based approach, this was possible since the problem presents an 'optimal substructure', in other words it means that the solution to the given optimization problem can be obtained by the combination of overlapping optimal solutions for its sub-problems. The algorithm is able to exploit this property to calculate the optimized solution, in fact it computes the optimized solutions for all the sub-instances of the problem, then it builds on top of them the overall optimal solution.

DynamicSelling(L)

$OptimalSolutionsDictionary \leftarrow Dictionary(0^{th} element)$
for $subBbatch \leftarrow 1, \ldots, n$ **do**
  $OptimalPlan \leftarrow emptyList$
  $OptimalValue \leftarrow 0$
  for $PutativeBatch \leftarrow 1, \ldots, subBatch$ **do**
    if $PutativeBatchValue + OptimalValueofRemainingSubBatches \geq$
    $OptimalValue$ **then**
      $OptimalValue \leftarrow PutativeBatchValue + OptimalValueofRemainingSubBatches$
      $OptimalPlan \leftarrow PutativeBatch + OptimalRemainingSubBatches$
    end if
  end for
  $OptimalSolutionsDictionary(SubBatch) \leftarrow OptimalPlan$
end for
return $OptimalSolutionsDictionary(n)$

Our goal consists in selling the n batches for the highest amount of money possible and this includes the possibility of combining smaller batches in order

to obtain the biggest profit.

For instance, we can consider the example showed on the Assignment paper; in order to determine with certainty which is the best way to sell 4 batches of product we can decompose the problem in different combinations of smaller batches (or *sub-batches* as defined in the pseudocode above): 1 batch + (3 batches sold in the best way), 2 batches + (2 batches sold in the best way), 3 batches + (1 batch sold in the best way) or just 4 batches sold as unique stock. Computing the highest monetary value among these 4 possible solutions will return the final optimal solution.

We can notice how the same procedure must be repeated for all the other smaller batches in a recursively manner, namely we must decompose also the instance of 3 batches, 2 batches and 1 batch. This reasoning is effective because we will arrive at a point in which the optimal solution for a *sub-batch* is the value of the batch itself, after that we can start computing the optimal solutions for bigger batches until we reach the n^{th} batches' number.

The first line of the algorithm summarizes this last paragraph, in practice we created a dictionary that will store in memory the optimal solution of each number of batches, however we can fill it a priori with the optimal solution of the smallest batch, namely 0 itself (0 batches are sold for 0 money).

We have to point out that the pseudocode proposed is not implemented through a recursive algorithm even though the idea behind it is the same. We used an iterative approach that takes advantage of nested **for** loops, the first one goes through all the different number of batches, and for each round we instantiate a new plan that will be associated to the optimal solution for each number of batches.

Inside the scope of this loop there is another **for** loop, its role is to consider all the smaller sub-batches for a given batch, then the algorithm (**if** statement) computes the highest profit given by the sum of each putative sub-batch and the value produced by the optimal solution of the remaining number of batches (the sum of their labels must be equal to batch considered by the first for loop).

Every time an optimal solution for a sub-batch is found it is saved in the dictionary. Eventually the algorithm returns only the value in the dictionary associated to the key that represents the n^{th} batch.

Since we start from the smallest batch and we proceed towards the biggest one, every time we consider a batch of number k, we have already stored in memory the optimal solution for the *k-1* batch and thus we can use it to compute the optimal solution of k. This feature is what gives our algorithm the properties of a Dynamic Programming approach.

3.1 The Recurrence Relation

As described in section 3 our algorithm exploits the optimal substructure of the problem in order to built, layer after layer, the optimal solution of the original instance.

In our case the *recurrence relation* is the best value that can be obtained when selling any number of batches, so we have to consider the different ways in which we can split a batch in sub-optimal selling plans. However, this procedure can result difficult to abstract so we are going to use a visual tool to make it more comprehensible:

	1	2	3	4	recurrence relation
1	1	-	-	-	Max(row1) = 1 opt
2	1 + 1 opt	2	-	-	Max(row2) = 2 opt
3	1 + 2 opt	2 + 1 opt	3	-	Max(row3) = 3 opt
4	1 + 3 opt	2 + 2 opt	3 + 1 opt	4	Max(row4) = 4 opt

In the table above, each **row** represents the optimal way of selling k batches, instead each **column** defines which sub-batch to insert in the hypothetical optimal solution for a certain batch. Our algorithm starts iterating from the first row and at the end of each one it stores in memory the solution with the highest payoff, such solution will be considered as the optimal selling plan for the current batch and it will be directly reused for computing the optimal solution for the successive batches (rows). This allowed us to dynamically fetch the optimal solution for each number of batches in every round.

As described before the first (and smallest) element considered is already the optimal solution of itself and thus it can be used as a base from which we can start building in sequence the 'bigger' optimal selling plans.

Note: Both the 0 batch and 1 batch can be considered as the smallest sub-instances of the problem, however in the table above we showed only the 1 batch for simplicity.

4 The Implementation

In this section we are going to show the Python implementation of all the functions involved in the assignment's solution, then we will describe in details the implementation of the Dynamic Programming algorithm that followed faithfully the pseudocode as presented in section 3.

The function below is dedicated to the randomly creation of the list L containing the n batches and their respective monetary value. It takes as

unique argument the number of desired batches, moreover we exploited the *random* library to assign a realistic value to each batch.

```
1  import random
2
3  def Instances(n: int) -> list:
4      L = []
5      for i in range(n + 1):
6
7          # the per-unity price is drawn in a range similar to
       the one used for the assignment's example (4-8), then
       multiplied for the numbers of unities in the given batch
8
9          batch = [i, i * random.randint(4, 8)]
10         L.append(batch)
11     return L
12
13 #Example: create a list of size 5
14
15
16 print(Instances(5))
17
18 >> [[0, 0], [1, 7], [2, 10], [3, 12], [4, 32], [5, 20]]
```

The function below is the core of the Dynamic Programming algorithm itself, it takes as parameter the list of n batches and returns the best selling plan for the n^{th} one. It faithfully follows the pseudocode of section 3, moreover we added comments for each line of code in order to give a detailed description of its features.

```
1  def DynamicSelling(L: list) -> tuple:
2
3      # this dictionary stores the optimal solution for each
       sub-instance of n elements and their associated value
4      OptimalDict = {}
5      # we know a priori that the optimal solutions for the 0
       batch is the batch itself so we can directly save them in
       the dictionary with their associated value
6      OptimalDict[0] = [L[0]], L[0][1]
7
8      # we iterate over each stock of batches (starting from 1)
       in order to find the optimal solution for each one
```

```
 9      for M in range(1, len(L)):
10          bestPlan = []
11          bestMoney = 0
12          # for each stock of batches we use their optimal sub-
        solutions to compute the respective optimal solution
13          for f in range(1, M + 1):  # we consider only the
        batches smaller or equal than the current one
14              currentPlan = []  # instantiate an empty selling
        plan that will be compared to the optimal one
15              OptRest = M - f  # number of optimal batch to add
         to hypothetical one in each round
16
17              # check if the money associated to the
        hypothetical batch 'f' + the optimal solution of the
        remaining batches is better than the previous one
18              if L[f][1] + OptimalDict[OptRest][1] >=bestMoney:
19                  # substitute the highest amount of money with
         the current one
20                  bestMoney = L[f][1] + OptimalDict[OptRest][1]
21                  # substitute the optimal selling plan with
        the current one
22                  currentPlan.append(L[f])
23                  currentPlan += OptimalDict[OptRest][0]
24                  bestPlan = currentPlan
25
26          # save in the dictionary the optimal solution for the
         specific 'M' batch and the respective value
27          OptimalDict[M] = bestPlan, bestMoney
28
29      # return the optimal solution of the bigger batch 'n' in
        the dictionary
30      return OptimalDict[len(L) - 1]
31
32
33  #Example: solve the problem's instance proposed in the
        assignment
34
35  # Given list in which n = 4
36  L=[ [0,0], [1,5], [2,9], [3,18], [4,21]]
37
38  #Algorithm's solution:
```

```
39 print(DynamicSelling(L))
40
41 >> ([[3, 18], [1, 5], [0, 0]], 23)
```

As we can notice the algorithm returns a tuple in which both the optimal selling plan ([[3, 18], [1, 5], [0, 0]]) and the total amount of money attributed to such plan (23) are stored.

The following function instead takes as parameter the result obtained by the Dynamic Programming algorithm and it prints it in a more comprehensible way.

```
1  def printResults(results: tuple) -> print:
2      totMoney = results[1]
3      print('Best selling plan:')
4      for batch in results[0]:
5          if batch[0] != 0:
6              print('stock of ' + str(batch[0]) + ' batches
       sold for ' + str(batch[1]))
7      print('Total amount of money: ' + str(totMoney))
8
9
10 #Example: print the result of the assignment's instance
11
12 #save the result given by the algorithm
13 results=([[3, 18], [1, 5], [0, 0]], 23)
14
15 printResults(results)
16
17
18 >> Best selling plan:
19 stock of 3 batches sold for 18
20 stock of 1 batches sold for 5
21 Total amount of money: 23
```

Note: the 0 batch is not displayed in the final results since it is a trivial finding.

5 The Complexity

After analyzing both the pseudocode and the Python implementation, we are now able to define the upper boundary to the worst-computationally

case in terms of big O notation. The following list explains which features of the Dynamic Programming algorithm contribute the most to its overall complexity when the input's size increases towards infinity:

Note that the reference lines come from the Python implementation of section 4.

- **line 9**: the first **for** loop, as described in section 3, iterates over the n number of batches so its complexity takes liner time *O(n)*.
- **line 13**: the nested **for** loop, considers only the sub-batches smaller or equal than the current one, however when the total number of batches tends to infinite also the number of batches considered by this nested loop increases linearly to infinite, resulting in a time complexity of *O(n)*.

All the other lines in the Python implementation can be condensed in the constant c since they do not contribute directly on the input's size *O(n)*.

By joining each single complexity present in the algorithm we obtain the formula:

$$n * n + c$$

The complexities of the two loops are multiplied and not summed since they are nested, however this can be understood also by looking at the table of section 3.1. In fact if we consider a square with edges of length n, its area is equal to n^2.

Given these considerations, the overall complexity of the computationally-worst case can be represented as:

$$0(n^2)$$

6 The Algorithm's Behaviour

In this section we propose the functions used to test the algorithm's time-performances on a series of different input sizes, moreover we are going to check whether the predictions formulated in section 5 are supported by empirical data.

The first function proposed has the role of calculating the Dynamic Programming algorithm's average time of execution (calculated over 1000 trials) for a given input size.

```
def Test(n: int, repetitions=1000) -> float:
    # number of trials
    for i in range(repetitions):
        # create the list of batches
        Input = Instances(n)

        # record the running time
        with Timer(name='DynamicSelling', logger=None):
            DynamicSelling(Input)

    # return the average
    return round(Timer.timers.mean('DynamicSelling'), 9)


#example: record the average time of execution for n=5


print(Test(5))

>> 3.0302e-05
```

The following function takes as argument a list of input sizes, then it returns another list containing the average execution time for each of them.

```
def Time_sets(input_array: list) -> list:

    Res = list()
    # iterating over the list of input sizes
    for Input in input_array:
        # test the running time for each input size
        result = Test(Input)
        Res.append(result)
    # return the an array of running times
    return Res


#Example: compute the average running time for this array
Sizes=[10,20,30,40,50,60,70,80,90,100]

print(Time_sets(Sizes))

```

```
>> [3.7167e-05, 7.314e-05, 0.000136664, 0.000201648,
   0.000261989, 0.000337778, 0.000443923, 0.000538388,
   0.000662466, 0.000785579]
```

The data just obtained will be merged by the following function into a series of coordinates.

```
def printPairs(sizes: list, time_data: list) -> print:
    for (n, t) in zip(sizes, time_data):
        print((n, t), end=' ')

#Example: create the coordinates

printPairs([10,20,30,40,50,60,70,80,90,100], [3.7167e-05,
   7.314e-05, 0.000136664, 0.000201648, 0.000261989,
   0.000337778, 0.000443923, 0.000538388, 0.000662466,
   0.000785579])

#set of coordinates
>>  (10, 3.7167e-05) (20, 7.314e-05) (30, 0.000136) (40,
   0.000202) (50, 0.000261) (60, 0.000337) (70, 0.000444)
   (80, 0.000538) (90, 0.000662) (100, 0.000785)
```

The upper bound to the worst case complexity predicted in section 5 was $0(n^2)$, thanks to an auxiliary function we will fit the experimental data around a mathematical curve shaped as the the big O notation of the algorithm (polynomial). As previously mentioned, the goal is to check whether our prediction is supported by the empirical data.

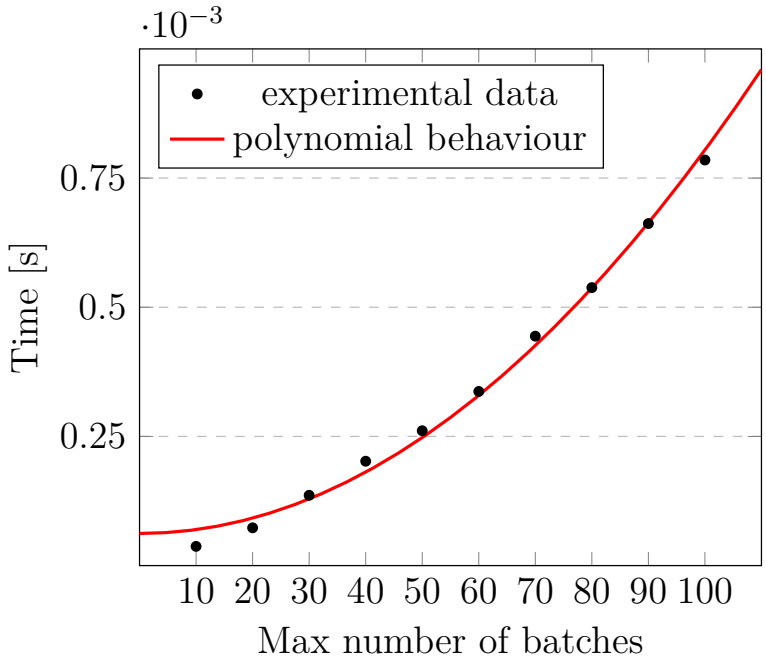
```
from codetiming import Timer
import scipy.optimize as optimization

def fit_curve(sizes: list, times: list):
    # define the mathematical function: quadratic
    def curve(x, a, b):
        return a * (x ** 2) + b

    # return the coefficients that shape the curve in order
   to fit the data
    # sizes represents the X-axis coordinates
    # times represents the Y-xis coordinates
```

```
12      return optimization.curve_fit(curve, sizes, times)
13
14  #Example: returns the curve's coefficients of a * x^2 + b
15
16  print(fit_curve([10,20,30,40,50,60,70,80,90,100], [3.7167e
        -05, 7.314e-05, 0.000136664, 0.000201648, 0.000261989,
        0.000337778, 0.000443923, 0.000538388, 0.000662466,
        0.000785579]))
17
18
19  >>  (array([7.42028847e-08, 6.21930942e-05]), array([[
        3.38587630e-18, -1.30356237e-14],
20          [-1.30356237e-14,  8.57744052e-11]]))
21
22  #a = 7.42028847e-08
23  #b = 6.21930942e-05 -> b is needed to make the curve more
        realistic
```

Now we are able to compare the experimental findings with the predicted outcomes:



As highlighted by the plot above, the predictions match with the experimental data, so we can safely confirm that the upper bound to the time complexity

of the Dynamics Programming algorithm we proposed is $O(n^2)$.

The table below represents the coordinates coming from the empirical findings.

Time [s]	3.717e-05	7.314e-05	0.000136	0.000202	0.000262	0.000337
Batches	10	20	30	40	50	60

Time [s]	0.00044	0.000538	0.000662	0.000785
Batches	70	80	90	100