Assignment: Dynamic Programming algorithm

Worked by: Amelia Kurti

May 2021

1 Introduction

The aim of this report is to describe a Dynamic Programming approach used to solve the given problem. The solution is provided in both modalities, pseudocode and Python code, to facilitate the understanding of the reader. In order to evaluate the entity of the provided solution, tests are performed, gathering data, to determine its efficiency and complexity. Predictions are compared to the collected data, visualised on a graph in order to grade the correctness of theoretically based predictions.

2 Dynamic programming

Dynamic programming is a programming approach, a collection of methods for solving sequential decision problems. The methods are based on decomposing a multistage problem into a sequence of interrelated sub-problems. The final solution is based on the collection of smaller sub-problem solutions, which are not solved independently. This property allows time saving during the execution since parts of the problems are already solved.

3 The problem to be solved

The given problem consists on determining the selling plan of a product. Several buyers can pay a specific price only for a certain number of product's units. Our task is to sell a given quantity of product to different buyers by creating a selling plan which maximises the revenue.

Input: An integer representing the number of batches to be sold and a registry which stores **"n batches : selling price"** key:value pairs.

Output: A list containing the selling plan of a given quantity of batches and the maximal revenue we can profit from this plan.The selling plan is represented as a list of tuples, each tuple showing **(batches number, selling price)**.

4 The Solution

In this section, both the pseudocode and the Python code of the Dynamic Programming algorithm will be shown, annotated using similar terms in order to be easily understandable, accompanied by detailed explanations.

```
 **SellingPlan**(n, SellingRegistry):
$OptimalSolutionsDictionary \leftarrow SolutionFor0Product$
for $Batches \leftarrow 1 \ldots n+1$ do
  $MaxRevenue \leftarrow 0$
  $FinalPlan \leftarrow []$
  for $UnsoldBatches \leftarrow 1 \ldots n+1$ do
    $BestPlan \leftarrow []$
    $Optimized \leftarrow Batches - UnsoldBatches$
    $UnsoldBatchesPrice \leftarrow SellingRegistry[UnsoldBatches]$
    $MaxRevenueOf(Optimized) \leftarrow OptimalSolutionsDictionary[Optimized][1]$
    $CurrentValue \leftarrow MaxRevenueOf(Optimized) + UnsoldBatchesPrice$
    if $CurrentValue > MaxRevenue$ then
      $MaxRevenue \leftarrow CurrentValue$
      $BestPlan \leftarrow [[UnsoldBatches, UnsoldBatchesPrice]]$
      $FinalPlan(Optimized) \leftarrow OptimalSolutionsDictionary[Optimized][0]$
      if $FinalPlan(Optimized) \neq (0,0)$ then
        $BestPlan+ = FinalPlan(Optimized)$
      end if
      $FinalPlan \leftarrow BestPlan$
    end if
  end for
  $OptimalSolutionsDictionary[Batches] \leftarrow [FinalPlan, MaxRevenue]$
end for
return $OptimalSolutionsDictionary[n]$
```

The SellingPlan() function takes as parameters, as described in the previous section, the number of product quantities to be sold **n** and the dictionary which contains quantity:price pairs, **SellingRegistry**. A new variable is instantiated, a dictionary **OptimalSolutionsDictionary**, where as the only key:value pair is given the best solution for selling 0 product. The key corresponds to the number of batches to be sold, the value contains the selling plan and the maximal revenue. In this dictionary for every number of batches, the maximal revenue plan will be stored.

The given optimization problem follows the definition of optimal substructure, meaning that an instance can be decomposed in several sub-instances (a bunch of Batches can be sold separately to different buyers fulfilling the condition to maximise the revenue). By the rules of Dynamic Programming, we know that in these cases, the optimized solution for a subproblem, can be extended to obtain an optimized solution of a super-problem. So, if we are required to plan the best selling of **n** batches, saying that we know the optimal selling plan of **n-k** batches differently named **Optimized**, for **k** from 1 to n, aka UnsoldBatches, we can exploit this already-known solution to sell **Optimized** to obtain the optimal result of our Batches quantity **n**.

In our algorithm, all these optimal known solutions, together with the maximal revenue profited from them, are stored in the **OptimalSolutionsDictionary** and can be easily accessed by indexing with their corresponding key **n-k**. To predict the most efficient placement of the other **k** batches, in **SellingRegistry** search for the value associated to **k**.

When we consider **n** batches, at least 1 up to **n** of them are not placed in the plan in an optimized way yet, named **UnsoldBatches**. The batches which are present in the selling plan in such way that ensures the maximal profit, are named **Optimized**. In the pseudocode, one can notice the variable **CurrentValue**, which stores the result of the maximal profit of the plan of Optimized and the value we can obtain by selling the **UnsoldBatches**, value stored in the **SellingRegistry**. Moreover the **BestPlan** list keeps track of the possible selling plan of the number of Batches, given that there is a certain number of **UnsoldBatches**, as well as the **CurrentValue**. For **n** Batches, different combinations are compared, involving different number of **UnsoldBatches**, and the combination whose **CurrentValue** is the highest, is set as the **FinalPlan**, while the **MaxValue** takes the value of **CurrentValue**. When this is finalised, in the dictionary **OptimalSolutionsDictionary**, a new pair **k:[FinalPlan, MaxValue]** is added. The process is repeated for every number of **UnsoldBatches**, from 1 to n. In the end, the function returns the plan for selling n Batches, by accessing the value associated to key=n in **OptimalSolutionsDictionary**.

4.1 Recurrence Relation

A recurrence relation is an equation that recursively defines a sequence, once one or more initial terms are given: each further term of the sequence is defined as a function of the preceding terms.

In the matrix shown below, the rows correspond to the number of **Batches** needed to be sold, while the columns to **UnsoldBatches** (batches whose optimal placement in the selling plan is not decided yet).

$$\begin{array}{ccccc}
- & 1 & 2 & 3 & 4 \\
1 & 1opt & - & - & - \\
2 & 1opt+1 & 2 & - & - \\
3 & 2opt+1 & 1opt+2 & 3 & - \\
4 & 3opt+1 & 2opt+2 & 1opt+3 & 4
\end{array}$$

The cells contain different combinations of filling the selling plan for a given number of Batches **n**. For every row, the combination which ensures the best maximal value of selling the given **n** Batches is selected and used on the computations for a larger number of Batches plan.

On every execution, the optimal placement for 1 batch is calculated. When 2 Batches are to be sold, there are two different ways on splitting them:

- sell the 2 batches together based on the registry of quantity of 2 : price.
- sell 1 Batch considering the optimal placement of one batch (calculated one row above) and the UnsoldBatch which is left based on the registry of quantity:price.

After all various ways of selling 2 Batches are considered, it is selected the combination with the highest revenue. It will be called **2opt** and will be used on the next steps of computing the diverse combinations of 3 and 4 batches. In this way the execution flows for any **n**.

4.2 The Python Program

The whole Python program to calculate and provide the solution of our problem is composed by diverse functions, around 50 lines in total, whose cooperation results in an well-explained selling plan together with the maximal revenue.

```
def create_registry(n):
    quantities = [n-i for i in range(0, n+1)][::-1]
    registry = {}
    for quantity in quantities:
        registry[quantity] = random.randint(quantity*4, quantity*6)
    return registry
```

The very first function **create-registry()**, creates a registry of the clients, to whom we can sell a specific quantity of product, for a given price. An integer is the only parameter of the function, and it is used to create a list of integers which symbolises the quantity of the batches that each client can buy. The list of these quantities is obtained by exploiting generators and a variable i taking values from 0 to n. By the difference n-i, are obtained all the integers from 0 to n. The resulting list, named **quantities**, which will be in descending order, since we start from i = 0, and n-0=n, using [::-1] we invert it.The next step performed by the function, is creating an empty dictionary, called **registry**, which contains ('units of product : price offered') pairs. To generate the price offered by a given buyer for a quantity of product, **random.randint** function of random module is applied, in a range that the price of the batches quantity is not smaller than 4 times the quantity, neither bigger than 6 times the quantity. The price value is immediately assigned to the corresponding quantity in the registry. Registry is then returned in the end of the execution of the function.

```
def selling_plan(n, registry):
    optimal_solutions = {0:[(0, registry[0]), registry[0]]}

    for batches in range(1, n+1):
        Max_revenue = 0
        final_plan = []

        for unsold_batches in range(1, batches+1):
            best_plan = []
            optimized = batches-unsold_batches
            current_value = optimal_solutions[optimized][1] +
            registry[unsold_batches]

            if current_value > Max_revenue:
                Max_revenue = current_value
                best_plan.append((unsold_batches,
                registry[unsold_batches]))

                if optimal_solutions[optimized][0] != (0, 0):
                    best_plan +=optimal_solutions[optimized][0]

                final_plan = best_plan

        optimal_solutions[batches] = [final_plan, Max_revenue]

    return optimal_solutions[n]
```

This is the central function of the whole program, which provides a Dynamic Programming based result. The function takes as input the number of the batches planned to be sold and the registry we obtained in the **create-registry()** function, and its aim is to return a list which inside itself contains the list the selling plan and an integer recording the maximal revenue we can profit. Creation of **optimal-solutions** is the next step. This dictionary is supposed to store the best plan to sell any quantity of product in the format 'quantity of product : [(quantity, price offered), revenue]'. The plan for selling 0 product is already part of this dictionary for a certain reason: every time we compute the plan for a quantity of product n, we base the solution in the plan of selling n-k product for k from 1 to n. So overall, the whole problem, since it is solved using a Dynamic Programming approach, it will divide the problem being considered in small subproblems whose solutions are known and then used to compute the result of the main problem. Important local variables of the function are Max-revenue which stores the highest revenue we can obtain by selling batches and final-plan list which stores the final plan of selling n batches. For any number of batches, we want to compute the Max-revenue as well as the plan, while we already know the best plan for n-k batches. For facilitating the orientation of the reader the variable that keeps track of the number of batches which have been already sold in optimized way, is named **optimized**. This is the difference of the number of total batches quantity we have to sell, and the **unsold-batches**. For the number of optimized batches, we can search on the **optimal-solutions** dictionary, by using **optimized** as a key, to return the maximal revenue which can be obtained by their best selling plan. The plan to sell n batches, the variable named **current-value**, keeps track of the revenue of every plan. It consists on the sum of the money obtained by selling plan of **'optimized'** batches, with the price we can sell the **unsold-batches**, found associated to the **key = unsold-batches**, on the registry given as a parameter to the selling-plan() function. If the **current-value** is bigger than the **Max-revenue**, we update the **Max-revenue** value, assigning to it the value of **current-value**. At the same time, to the **best-plan**, which in the beginning of the for loop is an empty list, is appended as a tuple the number of the unsold-batches and the price we can sell them, found as a value in the registry. Setting the condition in the following line, does not allow the function to append on the solution list the plan to sell 0 batches, which has no purpose on the result, rather than computational importance. If the plan of unsold-batches does not involve selling (0 batches, zero money units), then the plan is added from the optimal-solutions, to the best plan. In the end of the scope, the **final-plan** is set to be the best-plan.

So every time that the second for loop is executed, the best plan is empty, if the

$$\textbf{current-value} \geq \textbf{Max-revenue}$$

, the **Max-revenue** gets updated, best-plan gets filled and the final-plan assigned another list.

In the end of the execution of this second loop, the **optimal-solution** dictionary will have a new key:value pair ('batches number:[final-plan, Max-revenue]) format. All in all, for every number of batches from 0 to n, we store their best selling plan in the optimal-solutions dictionary, so they can be easily accessed when needed for computation. The function will return the value of this dictionary associated to the number of batches to be sold 'n', meaning the best selling plan for this quantity. Its results for running two different times for **n**=10 look as given below:

```
>>[[(3, 15), (7, 42)], 57]

>>[[(1, 6), (1, 6), (1, 6), (1, 6), (1, 6),
(1, 6), (1, 6), (1, 6), (1, 6), (1, 6)], 60]
```

```
def count_occurrences(given_list):
    occurrences_tracker = {}
    for element in given_list:
        if element not in occurrences_tracker:
            occurrences_tracker[element] = 1
        else:
            occurrences_tracker[element]+=1
    return occurrences_tracker
```

The **count-occurrences()** auxiliary function takes a list as a parameter and returns a dictionary, **occurrences-tracker**, counting the occurrence of every element of the list. The dictionary contains 'element:occurrence' pairs. The function considers every element of the list, if it is not in the dictionary already, it puts it as a key and assigns to it the value 1. Otherwise, it if is already a key of the dictionary, it will increase the values associated to it 1. The importance of this function is that it will help in the fancy and correct display of the selling plan. For every element (quantity, price offered), it will count its occurrence and create key:value pairs. The value will tell us how many stocks of this quantity are sold.

```
def present_plan(plan : list, n, occurrences_tracker):
    print('The selling plan in order to maximise the revenue
    for selling ',n, 'units of products, is: ')

    for occurrences in occurrences_tracker:

        if occurrences_tracker[occurrences]!=1:

            if occurrences[0]==1:
                print(occurrences_tracker[occurrences], 'stocks of',
                occurrences[0], 'batch each, to be sold for',
                occurrences[1], 'units of money.' )

            else:
                print(occurrences_tracker[occurrences], 'stocks of',
                occurrences[0], 'batches, to be sold for',
                      occurrences[1], 'units of money.')
        else:

            if occurrences[0] == 1:
                print(occurrences_tracker[occurrences], 'stock of',
                occurrences[0], 'batch each, to be sold for',
                occurrences[1],
                  'units of money each.')
            else:
                print(occurrences_tracker[occurrences], 'stock of',
                occurrences[0], 'batches, to be sold for',
                      occurrences[1], 'units of money.')

    print('The revenue: ', plan[1])
```

This function needed for the final representation of the plan. As an input takes the list of selling plan, the quantity to be sold and the **occurrences-tracker** dictionary. Rather than just printing the sentences to be displayed, the function considers the occurrence of every (quantity, price offered) and based on that and the quantity (i.e quantity singular 1 or plural), it prints the grammatically correct sentence. Moreover, the selling plan is indexed [1], so it returns which is the final revenue for the quantity to be sold.

It will present the plan for selling 10 batches in this way:

```
The selling plan in order to maximise the revenue for selling
10 units of products, is:
1 stock of 1 batch each, to be sold for 5 units of money each.
3 stocks of 3 batches, to be sold for 18 units of money.
The revenue:  59
```

4.3 The Algorithm Predicted Complexity

Taking a look at the structure of the proposed Dynamic Programming algorithm, what eyes notice are two ***for*** loops. Theoretically, loops represent iteration through all the elements of a given range. In our case, we consider a quantity of batches that varies from 1 to **n**. For each quantity of Batches, 1 up to **n** of these Batches are unsold. The algorithm first iterates though the quantity of Batches which corresponds to a time complexity of ***O(n)*** , then through the quantity of the at max **n** UnsoldBatches again a maximal time complexity ***O(n)***. Since the loops are nested, their respective time complexities multiply. Consequently, for a number of Batches quantity that varies from 1 to **n**, in the worst case of computation, the complexity of the algorithm reaches $O(n^2)$.

5 Testing the Algorithm

In order to practically validate the complexity of our algorithm, testing is performed to collect running time data and later map them into a graph.

```
def testDP(n: int, repetitions=1000) -> tuple:

    for i in range(repetitions):
        input = create_registry(n)
        with Timer(name='selling_plan', logger=None):

            selling_plan(n, input)

    return round(Timer.timers.mean('selling_plan'), 9)
```

The function test the selling-plan() function on a given value 1000 times and return the average running time of all the tryouts exploiting **codetiming** module. As the input of the selling-plan(), it is created a registry by create-registry()

function based on the number of input **n** the code is being tested with.

```
def Time_setsDP(input_list:list)-> list:
    l=list()

    for input in input_list:
        result=testDP(input)
        l.append(result)

    return l
```

For testing purposes, **Time-setsDP()** function takes a list of integers (inputs sizes), and it calls the testDP() for each of these inputs.The results of these runs, (the average running time for each input size) is stored in a list which is then returned by the function.

```
testing_list = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55,
60, 65, 70, 75, 80, 85, 90, 95, 100]
time_list = Time_setsDP(testing_list)

def print_pairs(value_list, time_list):
    for (n, t) in zip(value_list, time_list):
        print((n,t), end='')
```

```
>>(0, 3.485e-06)(5, 1.9943e-05)(10, 3.1964e-05)(15, 4.6908e-05)
(20, 6.4903e-05)(25, 8.5914e-05)(30, 0.000110196)(35, 0.000136788)
(40, 0.000166901)(45, 0.000199671)(50, 0.000236598)(55, 0.000277235)
(60, 0.00032223)(65, 0.000374069)(70, 0.000424959)(75, 0.000478548)
(80, 0.000536127)(85, 0.000593817)(90, 0.000655108)(95, 0.000720457)
(100, 0.000790681)
```

This function takes the input list given to Time-setsDP functions, as well as its result, to print pairs of (input size, average running time), providing so the coordinates to be inserted in the graph.

Once the coordinates of points for the graphical visualization are obtained, they must be compared to the curve of a function which corresponds to the predicted complexity. So this function will be Quadratic, but how can we predict its coefficient, if any?

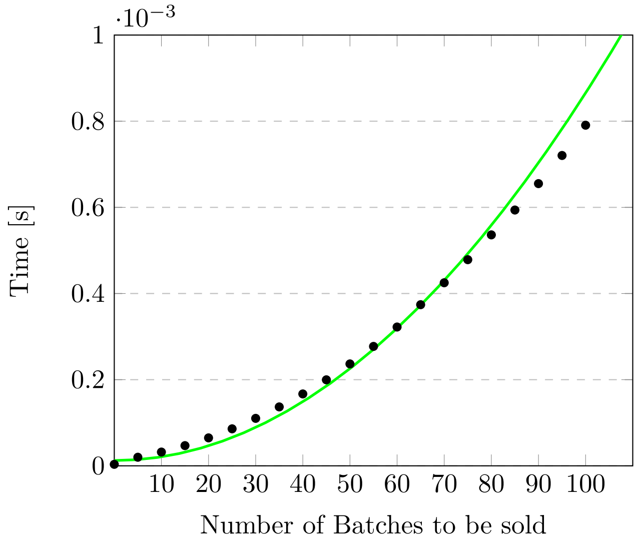
```
def polynomial(x, a, b):
    return a * numpy.power(x, 2) + b

print(optimization.curve_fit(polynomial, testing_list, time_list))
```

```
>>(array([8.52150536e-08, 3.70701950e-05]),
array([[ 1.19022216e-18,-4.26992202e-15],
[-4.26992202e-15,   2.68791585e-11]]))
```

It is one of the testing functions, which considers the timing-list and value-list to define the coefficients of the proposed function representing the Dynamic Programming based function, selling-plan(). The **b** coefficient is needed because, although the number of batches to be sold is 0, there will still be computation steps to be performed and this requires time.

From the results of testing, the curve in which our computation (input, time) pairs are supposed to map, is:

$$y = 8.52150536e - 08 * x^2 + 1.23341398e - 05$$



In the graph one can notice that the dots which represents the values obtained by the testing, align almost perfectly to the polynomial function with coefficients provided by Python implemented function **polynomial(x, a, b)** represented on a green motif. Hence we verify that the complexity of the Dynamic Programming function is $O(n^2)$ as predicted in the previous section.