Draft of September 24, 2014

Dynamic Programming (I)

Amrit Kumar
amrit.kumar@inria.fr

1 Recap and Beyond

In the previous lecture, we studied two *naive* recursive procedures namely: `Fibonacci` and `Binomial`. The former allowed us to compute the n^{th} number in the Fibonacci sequence, while the later computes $\binom{n}{k}$. These procedures entail a time complexity of $\Theta(2^n)$. The reason behind this exponential cost is the fact that many subproblems are solved multiple times, which is apparently visible from the recursion tree.

The general approach towards reducing the complexity of these procedures is called *Memoization*. The principle idea is to store or memorize the result of each subproblem, and if a previously solved subproblem is encountered later in the recursive calls, instead of solving the subproblem again, the previously stored result is retrieved and returned. This storage can be achieved using multiple means, in particular using an array or a hash table allowing the storage and retrieval in constant time.

Unlike Fibonacci and Binomial which inherently have a recursive definition, this lecture goes a step further and looks at *optimization problems*. At the first glance, these problems might not appear to have any recursive structure, but the motivation is that some of these optimization problems indeed have a recursive formulation. We see this through a couple of examples and study a generic approach called *Dynamic Programming* to solve some of these problems.

2 Optimization Problems

An optimization problem typically consists of an *objective function* $f : X^n \to X$, and some constraints $g_i : X^n \to X$ called inequality constraints and $h_i : X^n \to X$

referred to as equality constraints. Typically we would suppose that the space X is the set of real numbers $\mathbb{R}$ or the set of integers $\mathbb{Z}$. The goal of the problem is to find the *optimal value* (maximum or minimum) of f under the presented constraints. **The** optimal value is achieved with respect to **an** *optimal solution* i.e. x. Evidently, several optimal solutions might corresponding to the unique optimal value.

$$\begin{aligned}
&\underset{x}{\text{optimize}} && f(x) && (1)\\
&\text{subject to} && g_i(x) \leq c_i, \quad i = 1, \ldots, m && (2)\\
& && h_i(x) = d_i, \quad i = 1, \ldots, p && (3)
\end{aligned}$$

In the following sections, we study a couple of algorithmic problems which can be expressed as optimization problems.

3 Binary (0/1) Knapsack Problem

Informally, the problem is described using n items $\{x_1, \ldots, x_n\}$. Each item x_i has a value v_i units, and a weight w_i units associated with it. We further suppose that we have a knapsack with a capacity C, i.e. it can bear a maximum weight of C units. The goal of the problem is to choose the items to be put in the knapsack in a way that maximizes the total value and that the total weight of the chosen items respects the capacity of the knapsack.

The binary knapsack enforces the restriction where one can either take the item (1) or leave it (0). The items are hence indivisible and taking multiple copies of an item is not allowed.

3.1 Formal Description

Consider vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ of length n, representing the value and the weight of n items respectively. The couple (v_i, w_i) represents the value and the weight of the i^{th} item for $1 \leq i \leq n$. Consider a binary vector $\boldsymbol{s} \in \{0,1\}^n$ to denote the choices of the items. Hence the binary knapsack problem refers to the following optimization problem:

$$\underset{s}{\text{maximize}} \qquad \boldsymbol{s} \cdot \boldsymbol{v} \tag{4}$$

$$\text{subject to} \qquad \boldsymbol{x} \cdot \boldsymbol{w} \leq C \tag{5}$$

The operator $\cdot$ denotes the scalar product of two vectors.

3.2 Solving Knapsack

The first algorithm that pops up to solve the binary knapsack problem is the brute force search. The total number of possible choices for the items is 2^n i.e. the number of distinct vectors $\boldsymbol{s}$. The brute force algorithm would try each of the 2^n choices and find the optimal i.e. $\boldsymbol{s}$ that maximizes the total value. This leads to an algorithm of complexity $\Theta(2^n)$.

Another possible algorithm would be to use a greedy approach. Greedy programming techniques are used in optimization problems and typically use some heuristic or common sense knowledge to generate a sequence of sub-optimum that hopefully converges to an optimum value. Possible greedy strategies to the (0/1) Knapsack problem could be to:

— Choose the item that has the maximum value from the remaining items; this increases the value of the knapsack as quickly as possible.
— Choose the lightest item from the remaining items which uses up the capacity as slowly as possible allowing more items to be stuffed in the knapsack.
— Choose the items with as high a value per weight as possible.

However, using counterexamples, it can be easily shown that for each of the possible strategy, one can find an instance of binary knapsack on which the strategy does not yield and optimal solution.

Complexity theory view point. We know that binary knapsack is **NP**-complete. Hence, there should not exist any generic algorithm in **P** (unless **P** = **NP**) to solve all binary knapsack instances. We hence would not search for a polynomial time algorithm for binary knapsack (unlike we did for Fibonacci and Binomial), and would content ourselves by doing any better than brute force.

3.3 Recursive Approach

The goal of recursive algorithms as in *Divide-and-Conquer* partitions the starting problem into subproblems, then solves the subproblems recursively and finally combines the solutions to solve the original problem. If the subproblems overlap i.e. subproblems share subsubproblems, then a divide and conquer algorithm repeatedly solves the common subproblems. Hence, it does more work than necessary. A better solution is provided by *Dynamic Programming.*

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. ("Programming" in this context refers to the British interpretation of programs i.e. optimization using tables). In contrast to the divide-and-conquer approach, where subproblems are disjoint, dynamic programming applies when the subproblems overlap. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem. We would look more precisely on the conditions that an optimization problem must satisfy in order to apply dynamic programming later in the course.

3.3.1 Identifying Recursion in Binary Knapsack

It might be difficult at the first glance to imagine the knapsack problem to have a recursive formulation. But, indeed the problem does consist of solving subproblems, and the solution to these subproblems can be combined to form the solution to the initial problem. We would say that the knapsack problem exhibits *optimal substructure*: an optimal solution to the problem contains within it the optimal solutions to subproblems.

Defining the problem. To start with, we define the following problem that formulates the binary knapsack problem: $\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w})$. It represents the problem of determining the optimal value for a knapsack of capacity C, **for the first n items**. The vector $\boldsymbol{v}, \boldsymbol{w}$ corresponds to the value vector and the weight vector respectively. For instance, given a C', $\texttt{Knapsack}(2, C', \boldsymbol{v}, \boldsymbol{w})$ defines a knapsack problem for a capacity C' and considers the first two items only. Clearly, if $C' \leq C$, then $\texttt{Knapsack}(2, C', \boldsymbol{v}, \boldsymbol{w})$ is a subproblem of $\texttt{Knapsack}(n \geq 2, C, \boldsymbol{v}, \boldsymbol{w})$.

Guessing. In order to identify the recursion and hence the dependence of a knapsack problem on subproblems, we guess a part of an optimal solution yielding the optimal value. Despite the fact that many optimal solutions may correspond to the unique optimal value. Our goal is to find only one of these optimal solutions. Hence, the solution we are looking for would be considered as "the" solution and not "a" solution.

A possible guess would be that i_0th item for $1 \leq i_0 \leq n$ is in the optimal solution (we are looking) for the problem $\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w})$. For example, given the problem $\texttt{Knapsack}(5, 50, \boldsymbol{v}, \boldsymbol{w})$, we may guess that the 2nd item is in the optimal solution. However, a better and more constructive guess would be to consider the last item instead of just any item. We would see why in the sequel. Hence **our guess would be that the nth item (also the last item) of the problem instance $\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w})$ is in the optimal solution.**

From problem to subproblem. Since not everyone of us would be lucky in his guess, we might end up in one of the following two situations:

1. ***Our guess were right.*** If our guess that the nth item (also the last item) of the problem instance $\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w})$ is in the optimal solution, were right, we would have reduced the initial problem to the subproblem $\texttt{Knapsack}(n-1, C-w_n, \boldsymbol{v}, \boldsymbol{w})$ which satisfies the following relation:

$$\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w}) = v_n + \texttt{Knapsack}(n-1, C-w_n, \boldsymbol{v}, \boldsymbol{w})$$

   This follows from the observation that, if our guess were right, the optimal solution contains the nth item. We hence are left with the remaining first $n-1$ items and the capacity of the knapsack has now reduced to $C - w_n$, since it now contains the nth item. Moreover, adding this item to the knapsack gives us the value v_n.

2. ***Our guess were not right.*** If our guess were not right, which means that the nth item was not in the optimal solution to $\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w})$, we still have reduced the initial problem to the subproblem $\texttt{Knapsack}(n-1, C, \boldsymbol{v}, \boldsymbol{w})$, which satisfies:

$$\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w}) = \texttt{Knapsack}(n-1, C, \boldsymbol{v}, \boldsymbol{w})$$

This immediately ensues from the fact that, if the optimal solution does not contain the last item, we are only required to consider the first $n-1$ items with the same knapsack capacity.

We notice than choosing the last item instead of choosing any arbitrary item has the benefit that the new problems generated have the same form as that of the original problem.

Recursion. Well, since we are not GOD, we do not know before hand whether our guess would be right or wrong, hence we have to consider both the cases. Thus to obtain the optimal value and the optimal solution we would solve both the subproblems and choose the option that maximizes the total value. Hence we have for $\forall n$ st $C - w_n > 0$:

$$\texttt{Knapsack}(n, C, \boldsymbol{v}, \boldsymbol{w}) = \max\left(v_n + \texttt{Knapsack}(n-1, C-w_n, \boldsymbol{v}, \boldsymbol{w}), \texttt{Knapsack}(n-1, C, \boldsymbol{v}, \boldsymbol{w})\right) \tag{6}$$

The readers are cautioned to be slightly careful with Equation (6). If at any instance $C - w_n < 0$, the max should return the second value i.e. $\texttt{Knapsack}(n-1, C, \boldsymbol{v}, \boldsymbol{w})$. This is because in this case, we would not be able to put the n^{th} item. Hence, it would be an unintelligent guess to suppose that n^{th} item is in the optimal solution.

The base case for the recursion would be:

$$\texttt{Knapsack}(0, C, \boldsymbol{v}, \boldsymbol{w}) = \texttt{Knapsack}(n, 0, \boldsymbol{v}, \boldsymbol{w}) = 0 \tag{7}$$

The base cases refer to the condition that, when we have no items or when the knapsack is full, then the optimal value that we can obtain is 0.

Exercise 0: Neatly develop the full recurrence, with all the conditions.

Exercise 1: Exploiting (6) and (7), write a procedure in pseudocode to compute the optimal value for a knapsack instance.

Solution: Procedure 1 implements the above recursion. The procedure does nothing fancy and is direct translation of the recursion found in the previous exercise.

Exercise 2: Prove that the procedure found in Exercise 1 runs in exponential time. Present the recursion tree and comment.

Solution : Clearly, the procedure checks for every i satisfying $n \geq i \geq 1$ whether or not the i^{th} item which is the last item of the subproblem $\texttt{knapsack}(i, C, \boldsymbol{v}, \boldsymbol{w})$ is included in the optimal solution. Hence, the total search cost is $\Theta(2^n)$.

```
Procedure knapsack($n, C, v, w$)
1   if $n \leq 0$ then
        return 0
    end
2   if $C < w_n$ then
        withLastItem $\leftarrow -1$
    end
3   else
4       withLastItem $\leftarrow v_n +$ knapsack($n-1, C-w_n, v, w$)
    end
5   withoutLastItem $\leftarrow$ knapsack($n-1, C, v, w$)
6   return max(withLastItem, withoutLastItem)
```

Algorithm 1: (0/1) Knapsack naive approach.

Exercise 3: Modify the algorithm to exploit the observation made on the recursion tree in Exercise 2. This modification should use memoization.

Solution: Procedure 2 is a memoized version of the previous procedure. This requires us to declare an array memo[1..n][1..C], to store the optimal values for the subproblems. Clearly, the problem knapsack has two parameters n and C, hence in total there are nC subproblems and hence the size of the memo table. We highlight that, memo[i][C′] stores the optimal value for the subproblem `Knapsack`(i, C', v, w) where $1 \leq i \leq n$ and $1 \leq C' \leq C$.

Exercise 4: Estimate the cost of the procedure obtained in Exercise 3 in terms of space and time.

Solution: Cost in terms of space is $\Theta(nC)$, i.e. the cost of storing the matrix. To estimate the cost in terms of time, we distinguish two different costs incurred by the procedure:

1. Non-memoized calls: There are $\mathcal{O}(nC)$ calls which is the cost of filling the memo table. Filling the table only requires $\mathcal{O}(1)$ cost for addition and finding the maximum.
2. Memoized calls: These calls only require consulting the memo array and are made as recursive calls of the first type. Since each recursive call makes $\mathcal{O}(1)$ of such memoized calls. The total cost of memoized calls over all recursive calls is $\mathcal{O}(nC)$.

```
memo[1..n][1..C]={$-\infty$};
Procedure knapsack_memo($n, C, v, w$)
1   if $n \leq 0$ then
        return 0
    end
2   if memo[n][C]! $= -\infty$ then
        return memo[n][C]
    end
3   if $C < w_n$ then
        withLastItem $\leftarrow -1$
    end
4   else
5       withLastItem $\leftarrow v_n +$ knapsack_memo($n-1, C-w_n, v, w$)
    end
6   withoutLastItem $\leftarrow$ knapsack_memo($n-1, C, v, w$)
7   memo[n][C]$\leftarrow$ max(withLastItem, withoutLastItem)
8   return memo[n][C]
```

Algorithm 2: Obtains the optimal value for (0/1) Knapsack using memoization.

Hence, the total cost is $\mathcal{O}(nC)$.

Exercise 5: The procedure described in Exercise 3 only computes the optimal value. Modify the algorithm in order to obtain the optimal solution i.e. the choice of the items in order to obtain the optimal value.

Solution: Procedure 3 stores the optimal choice for each subproblem. We note that once we have computed the max of the two arguments for a problem $\texttt{Knapsack}(n-1, C, v, w)$, we know whether we have to consider the n^{th} item (i.e the last one) for the subproblem. The array opt_choice stores this result for each subproblem. Hence, opt_choice[i][C′] stores 1, if for the subproblem $\texttt{Knapsack}(i, C', v, w)$ where $1 \leq i \leq n$ and $1 \leq C' \leq C$, the i^{th} item is in the optimal solution for the given subproblem.

Procedure 4 uses the afore-obtained table opt_choice[1..n][1..C] to obtain the optimal solution. The idea is that our initial problem was $\texttt{Knapsack}(n, C, v, w)$. So, we first check whether the last item for this problem is in the optimal solution, this is obtained from the entry opt_choice[n][C] in the array. If the entry at this

```
memo[1..n][1..C]={$-\infty$};
opt_choice[1..n][1..C]={0};
Procedure knapsack_memo($n, C, \boldsymbol{v}, \boldsymbol{w}$)
1   if $n \leq 0$ then
        return 0
    end
2   if memo[n][C]$! = -\infty$ then
        return memo[n][C]
    end
3   if $C < w_n$ then
        withLastItem $\leftarrow -1$
    end
4   else
5       withLastItem $\leftarrow v_n$+knapsack_memo($n-1, C-w_n, \boldsymbol{v}, \boldsymbol{w}$)
    end
6   withoutLastItem $\leftarrow$ knapsack_memo($n-1, C, \boldsymbol{v}, \boldsymbol{w}$)
7   memo[n][C]$\leftarrow$ max(withLastItem, withoutLastItem)
8   if memo[n][c]==withLastItem then
        opt_choice[n][C]$\leftarrow$ 1
    end
9   return memo[n][C]
```

Algorithm 3: Storing the optimal choices for the n^{th} item for each subproblem.

position is 1, then the n^{th} item is in the optimal solution for this problem. Hence the table opt_choice[n]=1. Now, in the next step, we have the following subproblem if the entry found in the opt_choice is 1: `Knapsack`$(n-1, C-w_n, \boldsymbol{v}, \boldsymbol{w})$, hence we now see whether the $n-1^{\text{th}}$ item is in the optimal solution for this subproblem problem, the place to look for is opt_choice[n-1][C-w_n]. We continue this way to fill the opt_sol array incrementally.

Exercise 6: Estimate the cost of the two procedures in terms of space and time.

Solution: Space and Time: $\mathcal{O}(nC)$ for the first one storing the optimal choices, while $\mathcal{O}(n)$ in terms of space and time for the second procedure.

```
opt_sol[1..n]={0};
Procedure knapsack_optimalsol(opt_choice[1..n][1..C], $C, w$)
    for $i \leftarrow n$ to 1 do
        if opt_choice[n][C]==1 then
            opt_sol[i]=1
            $C = C - w_i$
        end
    end
```
Algorithm 4: Finding an optimal solution.

4 Elements of Dynamic Programming

Dynamic programming(DP) approach to an optimization problem can be considered to be an “intelligent brute force”. It basically consists in making a choice or a guess of a part of the optimal solution, this choice reduces the problem to solving another subproblem, which eventually leads to an optimal solution. However, there are two basic properties that the optimization problem should have in order for dynamic programming to be applied, namely:

Optimal substructure. An optimal solution to the original problem must incorporate optimal solution to the subproblems. As we have seen in the previous example of knapsack, a solution to the problem consists of making a choice i.e. which item to choose from the given set of items. Making this choice leaves one or more subproblems to be solved. We hence guess or choose a part of the optimal solution. Given this choice, we determine which subproblems ensue and how to best characterize the resulting space of subproblems. We should then verify that the solutions to the subproblems used within an optimal solution to the problem are themselves optimal.

Greedy approach also requires this property, however the returned solution is not guaranteed to be optimal in all cases.

Overlapping subproblems. When a recursive algorithm calls the same problem repeatedly, we say that the optimization problem has overlapping subproblems. In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion. Dynamic-programming algorithms typically take advantage of overlapping subproblems by leveraging on memoization.

The afore-described properties and the examples that we have seen so far leads us to characterize dynamic programming to be consisting of the following steps. These steps sometimes are very dependent in the sense that one step might already include the next one.

Step 1. Characterizing an optimal solution.
Implicitly requires to find a recursion. The idea to guess a part of the optimal solution. This often gives a recursion and leads to subproblems, which can be verified to exhibit the optimal substructure property.

Step 2. Solve the recursion to obtain optimal value using either memoization and recursion (top-to-down) or using iteration (bottom-to-up). This relies on the fact that subproblems overlap.

Step 3. Remembering the optimal choices.

Step 4. Reconstructing a solution.

5 Application: Selling Strawberries

We consider another optimization problem where a farmer or a distributor has n crates of strawberries and wishes to distribute them among k shops. The profit that a shop can make depends on the number of crates sold. Unfortunately, the profit made is not linear in the number of crates sold. Let $b_j(i)$ denote the profit made by the shop j by selling i crates ($1 \leq i \leq n$ and $1 \leq j \leq k$).

A simple instance is as follows for $n = 6$ and $k = 3$:

i	$b_1(i)$	$b_2(i)$	$b_3(i)$
0	0	0	0
1	3	6	5
2	7	12	10
3	12	16	15
4	17	20	20
5	26	22	25
6	35	24	30

We also define the marginal gain function for the j^{th} shop as $b_j(i) = b_j(i) - b_j(i-1)$, with $b_j(0) = 0$.

The goal of the problem is to find a vector $\boldsymbol{q} = \{q_1, \ldots, q_k\} \in \mathbb{N}^k$, where q_i for $1 \leq i \leq k$ denotes the number of crates distributed to the k^{th} shop which maximizes the total benefit:

$$\underset{\boldsymbol{q}}{\text{maximize}} \quad \sum_{j=1}^{k} b_j(q_j) \tag{8}$$

$$\text{subject to} \quad \sum_{j=1}^{k} q_j = n \tag{9}$$

5.1 Initial Approaches

To start with, the first naive solution could be to perform exhaustive search. In Exercise 7, we show that such an approach would certainly lead to an exponential-time algorithm.

Exercise 7: Find the total number of possible solutions.

Solution: The number of possible solutions is equal to the number of ways one can create k packets out of n items. This would require to place $k-1$ separators among the n items. Consider the case when $n = 3, k = 2$.

$$\clubsuit\,\clubsuit\,|\clubsuit\,\clubsuit\,\clubsuit\,|\clubsuit\,\clubsuit\,\clubsuit$$

Hence, the total number of possible ways is the number of ways to choose the position of $k-1$ separators among $n+k-1$ total items. This is equal to $\binom{n+k-1}{k-1}$. As seen in the previous lecture, this is exponential in n for large enough k.

As in the case of binary knapsack, one may also be tempted to employ greedy approach. Exercise 8,9 asks to prove that one of the possible strategies does work for certain instances, however they do not return optimal value in the generic case.

Exercise 8: Let us suppose that for each shop j, the marginal gain function g_j is decreasing. Propose an algorithm which finds the optimal distribution.

Solution: Greedy Algorithm is indeed optimal (Prove !). It consists in distributing the crates one by one. The i^{th} crate is given to the shop j with the maximum marginal gain $g_j(i)$ considering the number of crates it already has.

Exercise 9: Give a counter example to show that the above algorithm does not work in the general case.

Solution: Consider the following scenario with $n = k = 2$ with the follow profit table:

i	$b_1(i)$	$b_2(i)$
0	0	0
1	4	5
2	11	10

In this scenario, $g_1(1) = 4, g_1(2) = 7$ and $g_2(1) = g_2(2) = 5$. The greedy algorithm gives both of the crates to the 2nd shop to obtain a profit of 10, while, if we give both the crates to the first shop, we obtain a benefit of 11.

5.2 A DP solution

Defining the problem. Let us suppose that `maxprofit(n,k)` denotes the optimal profit obtained for n crates to be distributed among the **first** k **shops**. As in the knapsack problem, we may also look for subproblems to the above problem. For instance, given n′, `maxprofit(n',2)` is a problem and if n′≤ n, `maxprofit(n',2)` is a subproblem to `maxprofit(n,k≥2)`.

Guessing. As earlier, we guess a part of the optimal solution we are looking for. Different possible guesses could be made, for instance how many crates does the i_0th shop gets in the optimal solution to the problem instance `maxprofit(n,k)`, where $1 \leq i_0 \leq k$. As earlier, for the sake of the format of the problem, let us guess the number of crates given to the kth shop (the last shop) in the problem `maxprofit(n,k)`. We guess that in an optimal solution x crates of strawberry is distributed to the kth shop.

From problem to subproblem. If our guess were right, we reduce the initial problem `maxprofit(n,k)` to the subproblem `maxprofit(n-x, k-1)` using the following recursive relation:

$$\texttt{maxprofit(n,k)} = b_k(x) + \texttt{maxprofit(n-x, k-1)}$$

We highlight that, if our guess were correct, x crates out of n are given to the last i.e. k^{th} shop. Hence, we are left with $n - x$ crates and the remaining first $k - 1$ shops. This leads to the subproblem `maxprofit(n-x,k)`. Moreover, giving x crates to the k^{th} shop yields $b_k(x)$ benefit.

Recursion. Again, we are not GOD, hence we do not know beforehand the value of x. The solution is to try all possible values and find the x that maximizes the profit for the problem `maxprofit(n,k)`. Hence, the recurrence relation would be :

$$\texttt{maxprofit(n,k)} = \max_{0 \le 0 \le n} \left(b_k(x) + \texttt{maxprofit(n-x, k-1)}\right) \tag{10}$$

The base case would be:

$$\texttt{maxprofit(n,1)} = b_1(n) \tag{11}$$

Exercise 10: Prove that `maxprofit(n,k)` satisfies the optimal substructure property.
Solution: Left as an exercise.
Exercise 11: Write a procedure for selling strawberries using (10) and (11).
Solution: Procedure 5 implements the recursive functionality.

```
Procedure maxprofit(n, k)
1   if k==1 then
        return $b_1(n)$
    end
2   $q \leftarrow -\infty$
3   for $i \leftarrow 0$ to $n$ do
4       $q \leftarrow max(q, b_k(i) +$ maxprofit(n-i, k-1)$)$
    end
5   return $q$
```

Algorithm 5: Solves the strawberry selling problem.

Exercise 12: Prove that the procedure in Exercise 11 entails an exponential time for computing the optimal value.
Solution: Left as an exercise.

Exercise 12 bis: With the help of the recursion tree, convince yourself that redundant computations are performed. Conclude that, sacrificing space for time would lead to a better algorithm.

Solution: Left as an exercise.

Exercise 13: Transform the procedure obtained in Exercise 11 to obtain a DP algorithm that computes the optimal benefit.

Solution: Procedure 7 presents the required DP algorithm.

```
memo[0..n][1..k]={$-\infty$};
Procedure DPmaxprofit($n,k$)
1   if $k == 1$ then
        return $b_1(n)$
    end
2   if $memo[n][k]! = -\infty$ then
        return memo[n][k]
    end
3   $q \leftarrow -\infty$
4   for $i \leftarrow 0$ to $n$ do
5       $q \leftarrow max\,(q, b_k(i) +$ DPmaxprofit(n-i, k-1)$)$
    end
6   memo[n][k]$\leftarrow q$
7   return memo[n][k]
```

Algorithm 6: DP algorithm to solve the strawberry selling problem.

Exercise 14: Compute the cost in terms of time and space of the procedure presented in Exercise 13.

Solution: Clearly, the procedure consumes $\mathcal{O}(nk)$ space. For the cost in terms of time, we distinguish two different costs incurred:

1. Non-memoized calls: There are at most $\mathcal{O}(nk)$ such calls required to fill the table. Each such call requires the computation of the maximum of n values. Hence the cost is $\mathcal{O}(n^2k)$.
2. Memoized-calls: Such calls are made by calls of the previous type. Once these calls are made, the maximum of the values is computed. Since there are $\mathcal{O}(nk)$ calls of the previous type. The total cost in terms of time is $\mathcal{O}(n^2k)$.

Exercise 15: Modify the DP procedure to obtain the optimal solution.

Solution: Procedure 7 stores the optimal choice for each subproblem and Procedure 8 find an optimal solution.

```
memo[0..n][1..k]={$-\infty$};
opt_choice[0..n][1..k]={0};
Procedure DPmaxprofit($n,k$)
1   if $k == 1$ then
        return $b_1(n)$
    end
2   if $memo[n][k]! = -\infty$ then
        return memo[n][k]
    end
3   $q \leftarrow -\infty$
4   for $i \leftarrow 0$ to $n$ do
5       $q \leftarrow max\,(q, b_k(i) +$ DPmaxprofit(n-i, k-1)$)$
    end
6   memo[n][k]$\leftarrow q$
7   opt_choice[n][k]$\leftarrow i_{opt}$ // i for which max is obtained
8   return memo[n][k]
```

Algorithm 7: DP algorithm to store the optimal choices for the strawberry selling problem.

```
opt_sol[1..k]={0};
Procedure DPmaxprofit_optsol(opt_choice[0..n][1..k], n)
    for $i \leftarrow k$ to 1 do
1       opt_sol[i]$\leftarrow$ opt_choice[n][k]
2       $n \leftarrow n-$opt_choice[n][k]
    end
```

Algorithm 8: Finding the optimal solution.