1 Introduction

Dynamic programming is a powerful mathematical optimization technique that breaks the problem into subproblems of smaller dimensions, solves these subproblems only once, memorizes the result for each subproblem and then uses it to get the answer to subproblems of larger dimensions. Eventually, this leads to the solution of the initial problem. In order to get an optimal answer, each of the problems must be solved optimally.

There are two ways of how a problem can be solved with the help of dynamic programming.

Top-down approach. In this method, we break a problem into smaller subproblems, solve each subproblem and then combine the results to get the answer to the initial problem. The relation between the problem and its subproblems is represented by recurrent formulas.

Bottom-up approach. In this method, we find an answer to the current problem and then use its result to get answers to problems of larger dimensions. Our initial problem will be their subproblem.

In this homework, we will consider a top-down approach, although many problems can be solved in both ways.

Let’s show how dynamic programming works with an example.

Problem. You are given a knapsack of weight \( W \) and \( n \) items of weights \( w_1, w_2, ..., w_n \). You need to fill the knapsack in such a way that the total weight of all items in the knapsack is maximized and doesn’t exceed the capacity of the knapsack.

Solution. Let \( f_{i,k} \) be the optimal weight of all items in the knapsack of weight \( k \) when we can use frost \( i \) items. Here, we have defined the problem of the dimension \((i,k)\). These two parameters help us to define a problem (note that function \( f \) may also depend on variables that have no connection with the dimension of the problem).

Let’s assume that we know optimal answers to all subproblems of smaller sizes, i.e. to all problems \((i',k')\), such that \( i' \leq i \) and \( k' \leq k \) and \((i,k) \neq (i',k')\). Now we want to find an answer to problem \((i,k)\). Let’s look at item number \( i \). There are two possible actions that we can take: we can either include item \( i \) in the knapsack, if there is enough capacity for it, or we can ignore it. If we ignore item \( i \), this means that we have to fill the knapsack of the weight \( k \) optimally without using this item. Thus, we can only include items from 1 to \( i - 1 \). Therefore, the subproblem where we don’t include item \( i \) will be \((i-1,k)\). The answer to this subproblem is \( f_{i-1,k} \). Now, let’s consider the case where we try to put item \( i \) into the knapsack. It is only possible, if the capacity of the knapsack is not less than the weight of the item \( i \), i.e. \( k \geq w_i \). If this condition is true, then we put the item in the knapsack. As regards the rest of the space in the knapsack, we will try to fill it with items 1, 2, ..., \( i - 1 \) in the most optimal way. This subproblem will have dimension \((i-1,k-w_i)\), where \( k-w_i \) is available capacity left in the knapsack after we put item \( i \) in it. So, all smaller subproblems of problem \((i,k)\) can be divided into two sets: when we include item \( i \) in the knapsack and when we don’t. Using all these observations we can calculate the value \( f_{i,k} \):

\[
\begin{align*}
 f_{i,k} &= \begin{cases} 
 \max(f_{i-1,k-w_i} + w_i, f_{i-1,k}) & \text{if } k - w_i \geq 0 \\
 f_{i-1,k} & \text{otherwise}
\end{cases} 
\end{align*}
\]

This is a recurrent formula for dynamic programming. We can use recursion in order to compute it. An important thing to remember here is that we should calculate values for problems \( f_{i,k} \) only once. We can do this by memorizing all the computed values and just use them, if they are needed. This will improve time complexity dramatically.

Similar to mathematical induction, dynamic programming technique requires base.

As we can see, the recurrent formula above makes sense only when \( i > 0 \). So the base case will be \( f_{0,k} = 0 \), where \( k \in [0,W] \).

The correctness of such recurrent equations is proven in two steps: first, we prove the base case, then we prove the formula. When we prove the formula, we use the assumption that all subproblems are solved
optimally. For example, in this problem all possible valid combinations of the first \( i \) items can be divided into sets: when we include item \( i \) into the knapsack and when we don’t. We have optimal answers to both of these cases in our subproblems \((i - 1, k)\) and \((i, k - w_i)\), thus the answer to \((i, k)\) will also be optimal.

As each value \( f_{i,k} \) is calculated only once, then the time complexity for this problem is \( \Theta(nW) \).

2 Problem 1

Problem: You are given a sequence of \( n \) integers \( a_1, a_2, ..., a_n \). Find the longest increasing subsequence \( a_{k_1}, a_{k_2}, ..., a_{k_m} \), such that \( k_i \in \{1, 2, ..., n\} \) for \( i \in \{1, 2, ..., m\} \), \( k_1 < k_2 < ... < k_m \) and \( a_{k_1} \leq a_{k_2} \leq ... \leq a_{k_m} \).

1. Find a dynamic programming formula to solve the problem.

2. Prove its correctness.

3. Show and prove its time complexity.

Solution:

1. Let \( f_i \) be the length of the longest increasing subsequence, that ends with element on position \( i \). Then:

   Base case:
   
   \( f_0 = 0 \)

   For \( i > 0 \):
   
   \( f_i = max(f_j + 1) \) for all \( j: 0 \leq j < i \) and \( a_j \leq a_i \)

   The answer to the initial problem: \( f_n \)

2. When there are no elements in the array, then the length of the longest increasing subsequence (LIS) is zero. The base case is correct. Now, let’s assume that values for \( f_1, f_2, ..., f_{i-1} \) are calculated correctly. We need to prove that \( f_i \) is also correct. If the LIS that ends with element \( a_i \) has the length one, then this means that \( a_i \) is the smallest element in \( a_1, a_2, ..., a_i \). In this case, our solution will give \( f_i = 1 \), which is correct. If the LIS that ends with \( a_i \) consists of more than one element, then some element \( a_j, j < i \) and \( a_j < a_i \) will come right before \( a_i \) in the optimal answer. As we iterate through all valid \( a_j \), and \( f_j \) is the longest increasing subsequence that ends with \( a_j \), then \( f_i \) will also be optimal.

3. We need \( O(n) \) to calculate \( f_i \). The overall time complexity will be \( O(n^2) \).

3 Problem 2

Problem: Find all binary sequences of the length \( n \), such that no sequence has two consecutive zeros.

1. Find a dynamic programming formula to solve the problem.

2. Prove its correctness.

3. Show and prove its time complexity.

Solution:

1. Let \( f_i \) be the number of all binary sequences of the length \( i \)

   Base case:
   
   \( f_0 = 1, f_1 = 2 \)
For $i > 1$:

$$f_i = f_{i-1} + f_{i-2}$$

The answer to the initial problem: $f_n$

2. The number of empty sequences is zero. Valid sequences of length 1: "0", "1". The base case is correct. Now, let’s assume that values $f_2, f_3, ..., f_{i-1}$ are calculated correctly. Let’s show that $f_i$ is also correct. Let’s consider all valid sequences of the length $i$. They can be divided into two groups: those that end with 1 and those that end with 0. The number of sequences of the length $i$ that end with 1 equals $f_{i-1}$. Indeed, if the last element is 1, then it doesn’t affect the previous elements anyhow, they only need to be valid. If the last element in the sequence is zero, then the previous element can be only 1. This means, that all sequences of length $i$ that end with 0 will end with 10. The number of such sequences is $f_{i-2}$ (we have just shown this).

3. $f_i$ is calculated in $O(1)$ time. Thus, the overall complexity is $O(n)$.

4 Problem 3

*Problem:* You are given a knapsack of weight $W$ and $n$ items with integer weights $w_1, w_2, ..., w_n$. The items have their cost $c_1, c_2, ..., c_n$. You can take each item only once. Your goal is to fill the knapsack with the items, such that their cost is maximized.

1. Find a dynamic programming formula to solve the problem.

2. Prove its correctness.

3. Show and prove its time complexity.

*Solution:*

1. Let $f_{i,k}$ be the optimal cost of the items in the knapsack of the weight $k$, when we have only the first $i$ items. Then:

   Base case:

   $$f_{0,k} = 0 \text{ for all } k \in [0, W]$$

   For other values:

   $$f_{i,k} = \begin{cases} 
   \max(f_{i-1,k-w_i} + c_i, f_{i-1,k}) & \text{if } k - w_i \geq 0 \\
   f_{i-1,k} & \text{otherwise}
   \end{cases}$$

   The answer to the initial problem: $f_{n,W}$

2. If there are no items, then the total cost is always zero. The base case is correct. Now let’s assume that all $f_{i',k'}$, where $(i', k') \in (i, k)$ are calculated correctly. Let’s show that $f_{i,k}$ is also correct. Let’s consider two possibilities: when item $i$ is included in the knapsack and when it is not. If item $i$ is not in the knapsack, then we need to fill the knapsack of capacity $k$ with items 1, 2, ..., $i-1$, such that their cost is maximized. But we already know the answer to this problem. It is $f_{i-1,k}$. Now, let’s assume we include item $i$ in the knapsack. This means that there is now only $k - w_i$ space left. Let’s fill this space with items 1, 2, ..., $i - 1$, such that their total cost is maximized. The optimal value for this problem is $f_{i-1,k-w_i}$. At the end, we just need to take the one which has the larger value. As both problems are solved optimally (give the largest total cost), $f_{i,k}$ will also be optimal.

3. We need $O(1)$ to find $f_{i,k}$. The overall complexity will be $O(nW)$. 

3
5 Problem 4

Problem: You are given a knapsack of the weight \( W \) and \( n \) items with integer weights \( w_1, w_2, \ldots, w_n \). Each item has its cost \( c_1, c_2, \ldots, c_n \). You can take each item as many times as you want. Your goal is to fill the knapsack with items, such that their cost is maximized.

1. Find a dynamic programming formula to solve the problem.
2. Prove its correctness.
3. Show and prove its time complexity.

Solution:

1. Let \( f_{i,k} \) be the optimal cost of items in the knapsack of the weight \( k \), when we have only first \( i \) items. Then:
   Base case:
   \[
   f_{0,k} = 0 \text{ for all } k \in [0,W]
   \]
   For other values:
   \[
   f_{i,k} = \begin{cases} 
   \max(f_{i,k-w_i} + c_i, f_{i-1,k}) & \text{if } k - w_i \geq 0 \\
   f_{i-1,k} & \text{otherwise}
   \end{cases}
   \]
   The answer to the initial problem: \( f_{n,W} \)

2. The proof is similar to the previous one with just one exception. Now we can take each item as many times as we want. So in the case, when we include item \( i \) in the knapsack, we now need to fill the knapsack of the weight \( k \) with all items \( 1, 2, \ldots, i \). In the previous problem, we couldn’t include item \( i \), as it was already taken. But now we can. So the optimal answer to this subproblem will be \( f_{i,k-w_i} \).

3. We need \( \mathcal{O}(1) \) to find \( f_{i,k} \). The overall complexity will be \( \mathcal{O}(nW) \).

6 Problem 5

Problem: You are given a knapsack of the weight \( W \) and \( n \) items with integer weights \( w_1, w_2, \ldots, w_n \). Each item has its cost \( c_1, c_2, \ldots, c_n \). You can take item \( i \) only \( b_i \) times, where \( b_i \) is a positive integer number for \( i \in \{1, 2, \ldots, n\} \). Your goal is to fill the knapsack with items, such that their cost is maximized.

1. Find a dynamic programming formula to solve the problem.
2. Prove its correctness.
3. Show and prove its time complexity.

Solution:

1. Let \( f_{i,k} \) be the optimal cost of the items in the knapsack of the weight \( k \), when we have only first \( i \) items. Then:
   Base case:
   \[
   f_{0,k} = 0 \text{ for all } k \in [0,W]
   \]
   For other values:
\[ f_{i,k} = \begin{cases} 
\max(f_{i,k-l*W_i} + l*c_i, f_{i-1,k}) & \text{for all } l: \ l*W_i \geq w_i \text{ and } 1 \leq l \leq b_i \\
 f_{i-1,k} & \text{otherwise} 
\end{cases} \]

The answer to the initial problem: \( f_{n,W} \)

2. The proof for this problem is similar to proofs for the previous two. In this case, we do not consider the two possibilities of whether we put item \( i \) in the knapsack or not. Instead, we explore how many times we take item \( i \). For example, 0, 1, 2, ..., \( b_i \) times. We look through each case separately. The optimal substructure of subproblems will lead to the optimal structure of the initial problem.

3. It takes \( O(W) \) time to find \( f_{i,k} \). Because in the worst case, the weight of the item will be one, and \( b_i \) will be no less than \( W \). The overall complexity will be \( O(nW^2) \).

### 7 Problem 6

**Problem:** You are given two strings \( s = s_1s_2...s_n \) and \( t = t_1t_2...t_m \). You may perform three type of operations on a string: delete a symbol, insert a symbol or replace existing symbol with any other possible symbol. Assume that all symbols are lowercase English letters. What is the minimum number of operations you need to transform string \( s \) to string \( t \)?

1. Find a dynamic programming formula to solve the problem.
2. Prove its correctness.
3. Show and prove its time complexity.

**Solution:**

1. Let \( f_{i,k} \) be the minimum number of operations needed to transform string \( s_1s_2...s_i \) to string \( t_1t_2...t_k \).

   **Base case:**
   
   \[ f_{0,k} = k \text{ for all } k : 1 \leq k \leq m f_{i,0} = i \text{ for all } i : 1 \leq i \leq n \]

   For other values:

   \[ f_{i,k} = \begin{cases} 
   \min(f_{i-1,k}, f_{i,k-1}, f_{i-1,k-1}) + 1 & \text{if } s_i \neq t_k \\
   f_{i-1,k-1} & \text{otherwise} 
   \end{cases} \]

   The answer to the initial problem: \( f_{n,m} \)

2. We need to add \( k \) symbols to turn an empty string to string \( t_1t_2...t_k \) and we need \( i \) operations to turn string \( s_1s_2...s_i \) to an empty string. The base case is correct. Now, let’s assume that all values \( f_{i',k'} \) for \( (i',k') \in (i,k) \) are calculated correctly. Let’s show that \( f_{i,k} \) is also correct.

   Let’s look into three possibilities:

   - At some step we have erased symbol \( s_i \). Let’s assume, we have erased it during the first step. It won’t change the result. Then, after erasing \( s_i \), we need to find a way to turn \( s_1s_2...s_{i-1} \) into \( t_1t_2...t_k \). The minimum number of operations to do so equals \( f_{i-1,k} + 1 \).

   - At some step, we have added symbol \( b_k \) to string \( s_1s_2...s_i \). As we can add symbols anywhere, we may assume that it was the last operation. So before adding \( b_k \), we need to turn \( s_1s_2...s_i \) into \( t_1t_2...t_{k-1} \). Then, we add \( b_k \). The minimum number of operations to do so equals \( f_{i,k-1} + 1 \).
• Now, let’s look at all other possibilities. This means that we do not erase \( a_i \) and we don’t add \( b_k \). First, let’s show that in this case we do not need to add any symbol to the right side of \( a_i \). Let’s say we added some symbols to the right of \( a_i \). None of them can be equal to \( b \). Then, some of them need to be changed to \( b \). But this is not optimal, because instead of adding some symbol and changing it to \( b \), we can just add \( b \) in the first place. Thus, \( a_i \) will be the last symbol in our string. If \( a_i = b_k \), then we don’t need to do anything with \( a_i \). The answer to this subproblem is \( f_{i-1,k-1} \). If \( a_i \neq b_k \), then we need to change \( a_i \) to \( b_k \). The answer to this subproblem will be \( f_{i-1,k-1} + 1 \).

3. \( f_{i,k} \) requires \( O(1) \) to be computed. Thus, the overall complexity is \( O(nm) \).

8 Problem 7

Problem: You have a board of a dimension \( 1 \times n \). There is a robot on cell 1. The robot can make moves from cell \( i \) to cells \( i + 1, i + 2, ..., i + k \). Moving to cell \( i \) costs \( c_i \) resources (\( c_1 = 0 \)). Find the path from cell 1 to cell \( n \) for a robot that requires a minimum amount of resources.

1. Find a dynamic programming formula to solve the problem.
2. Prove its correctness.
3. Show and prove its time complexity.

Solution:

1. Let \( f_i \) be a minimum amount of resources required for a robot to get to cell \( i \).
   
   Base case:
   
   \[ f_0 = 0 \]
   
   For \( i > 0 \):
   
   \[ f_i = \min(f_{i-j}) + c_i \text{ for all } j : i - j \geq 0 \text{ and } 1 \leq j \leq k \]
   
   The answer to the initial problem: \( f_n \)

2. If the board is empty, then the amount of resources gained is zero. The base case is correct. Let’s assume that subproblems \( f_1, f_2, ..., f_{i-1} \) are solved optimally. Let’s show that \( f_i \) is also correct. We can get to cell \( i \) from cells \( i - 1, i - 2, i - k \) (as long as these cells are on the board). We know the value of the optimal way to get to each of these cells and that all paths to cell \( i \) from 1 go through them. As all subproblems are solved optimally, the answer to \( f_i \) also will be optimal.

3. \( f_i \) is computed in \( O(k) \) time. Thus, the overall complexity equals \( O(nk) \).

9 Problem 8

Problem: You are given a board of a dimension \( N \times M \). There is a robot on cell \( (x_0, y_0) \) on the board. At each step, a robot may go from cell \( (x, y) \) to cells \( (x - 1, y), (x, y - 1), (x + 1, y) \) and \( (x, y + 1) \) (all cells must be in table). Find the number of ways in which the robot may get from the initial cell to the final cell \( (x_1, y_1) \) in \( k \) steps.

1. Find a dynamic programming formula to solve the problem.
2. Prove its correctness.
3. Show and prove its time complexity.
Solution:

1. Let $f_{x,y,k}$ be the number of ways for the robot to get to cell $(x,y)$ in $k$ steps from cell $(x_0,y_0)$. And let:

$$g_{i,j,k} = \begin{cases} f_{i,j,k} & \text{if } 1 \leq i \leq N \text{ and } 1 \leq j \leq M \\ 0 & \text{otherwise} \end{cases}$$

Base case:

$$f_{x,y,0} = 0 \text{ for all } (x,y) : 1 \leq x \leq N \text{ and } 1 \leq y \leq M$$

For $k > 1$:

$$f_{i,j,k} = g_{i-1,j,k-1} + g_{i,j-1,k-1} + g_{i+1,j,k-1} + g_{i,j+1,k-1}$$

The answer to the initial problem: $f_{x_1,y_1,k}$

2. The proof is similar to the proof for the previous problem. We can get to cell $(x,y)$ on step $k$ from cells $(x-1,y), (x,y-1), (x+1,y)$ and $(x,y+1)$ on step $k-1$. As all the subproblems are solved optimally, it leads to the optimality of the initial problem.

3. The time complexity for calculating $f_{x,y,k}$ is $O(1)$. Thus, the overall complexity is $O(NMk)$.

10 Problem 9

Problem: You are given a sequence of $n$ binary digits $a_1, a_2, ..., a_n$ ($a_i \in \{0,1\}$ for $i \in \{1,2, ..., n\}$). Find the number of subsequences $a_{k_1}, a_{k_2}, ..., a_{k_m}$ such that $k_i \in \{1,2, ..., n\}$ for $i \in \{1,2, ..., m\}$, $k_1 < k_2 < ... < k_m$ and the sum of all digits in the subsequence is even.

1. Find a dynamic programming formula to solve the problem.
2. Prove its correctness.
3. Show and prove its time complexity.

Solution:

1. Let $f_i$ and $g_i$ be numbers of subsequences of a sequence $a_1, a_2, ..., a_i$ with even and odd sums respectively. And let:

$$\sigma(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

Base case:

$$f_0 = g_0 = 0$$

For $i > 0$:

$$f_i = \begin{cases} f_{i-1} + g_{i-1} & \text{if } \sigma(a_i) = 0 \\ f_{i-1} & \text{otherwise} \end{cases}$$

$$g_i = \begin{cases} g_{i-1} & \text{if } \sigma(a_i) = 0 \\ f_{i-1} + g_{i-1} & \text{otherwise} \end{cases}$$

The answer to the initial problem: $f_n$
2. The correctness of the base case is obvious. Now let’s assume that all subproblems $f_1$, $g_1$, ..., $f_{i-1}$, $g_{i-1}$ are computed correctly. We will show that $f_i$ and $g_i$ are also correct.

Let’s consider sequence $a_1, a_2, ..., a_i$. We know the number of subsequences with even and odd sums of the sequence $a_1, a_2, ..., a_{i-1}$. All these subsequences are also part of $a_1, a_2, ..., a_i$. The remaining subsequences of $a_1, a_2, ..., a_i$ are the ones which include $a_i$. So let’s look at $a_i$. If $a_i$ is odd, then we can take any odd (even) subsequence of $a_1, a_2, ..., a_i$ and turn it into even (odd) by adding $a_i$ to it. We can do the same if $a_i$ is even. As all subproblems are computed correctly, the answer to the initial problem will also be correct.

3. It takes $O(1)$ time to find $f_i$ and $g_i$. Thus, the overall time complexity will be $O(n)$. 