Dynamic Programming Finale Course Project

Yuval Brunshtein

February 2025

1 Project Description and Definitions

This project is a part of the Algorithms of Dynamic Programming course in Ariel University given by Prof' Vadim Levit.
The project's algorithms will be applied using Python, the correctness of the algorithms will be proven as well as their complexities and the proof of the correctness of the complexities.

Let A be a sequence where $A = a_1, a_2, ..., a_n$ where $a_i \in R$ is an ascending sequence meaning $i_1 < i_2 < ... < i_k$ where $j = 1, 2, 3.., k$ then a $D = d_1, d_2, .., d_k$ is a subsequence where $a_{i_j} = d_j$.
A common subsequence is a subsequence D which is a subsequence of both A and B.
An increasing subsequence is a subsequence D of A such that there exists:
1. It satisfies the definition of a sub sequence above.
2. The values in the sub sequence satisfy: $a_{i_1} < a_{i_2} < a_{i_3} < .... < a_{i_k}$

Let A be a sequence where $A = \{a_1, a_2, a_3, .., a_n\}$ and $C = \{c_1, c_2, c_3, .., c_n\}, a_i \in Z, c_i \in \{0, 1\}$ then longest increasing subsequence(LIS) of A is a sequence which satisfies the following:
1. $(i_1 < i_2 < ... < i_k), a_{i_1} < a_{i_2} < a_{i_3} < .... < a_{i_k}$
2. $c_{i_1} = c_{i_2} = c_{i_3} = ... = c_{i_k} = 1$

1.1 Question 1

Let there be two sequence A,B where $A = \{a_1, a_2, a_3, .., a_n\}, B = \{b_1, b_2, b_3, .., b_m\}, |A| = n, |B| = m, n = m \vee n! = m, n, m, a_i, b_j \in Z\ \forall i, j, i = 1, 2, 3..., n, j = 1, 2, 3, ..., m$
1. Find the number of the subsequences which satisfy being the longest common subsequences(LCS).
2. Return the longest common subsequences. Limit the number of longest common subsequences which are able to return using a parameter θ
3. Return the longest common subsequences with no duplicates return at most θ subsequences.

1.2 Question 2

Let there be two sequences A, C where $A = \{a_1, a_2, a_3, .., a_n\}, C = \{c_1, c_2, c_3, .., c_n\}, |A| = |C| = n, a_i \in Z, c_i \in \{0, 1\}$ for both of the sequences the following should be achieved:
1. Finding the length of the longest increasing subsequence of A
2. Find the number of the longest increasing subsequences of A.
3. Returning all longest increasing subsequences of A. Defining a parameter θ which limits the amount of subsequences returned.
4. Returning the longest increasing subsequences of A with no duplicates as well as limiting the amount of subsequences returned up until θ.

2 Question 1 - LCS Algorithm and Backtracking

In order to solve this problem we will use an algorithm known as LCS and in addition use a backtracking algorithm to return the amount of subsequences which exist as well as the subsequences themselves. The algorithm used in the solution was applied using python in the file "LongestSubsequence.py". Using a dynamic programming approach this is efficient because we are able to break down the lists and slowly from breaking the problem into smaller problems able to solve the bigger problem.
Note: We will use the definitions here described in the first section of the paper, Project Description and Definitions specifically the definitions and in Question 1 and some of the ones beforehand .

2.1 Algorithm Explanation and Complexity

LCS Algorithm:
1. We create a dynamic programming matrix $(n + 1) \times (m + 1)$ which will be noted as DPM where each cell $DPM[i][j]$ will contain the length of the longest common subsequence up until the i element of A and the j element of B. For the base case we initialize: $DPM[0][j] = DPM[i][0] = 0, \forall i, j$.
2. Updating the matrix- for this two cases will be taken case 1: $A[i] = B[j]$ then $DPM[i][j] = DPM[i - 1][j - 1] + 1$
case 2: $A[i]! = B[j]$ then $DPM[i][j] = max(DPM[i - 1][j], DPM[i][j - 1])$.
We update this table using two loops going over the values of A, B.
3. The cell which will contain the size of the longest common sub sequence will be $DPM[n][m]$.
Note: i, j represents the indexes of A, B which are ran by the two loops.
Complexity: $O(n \times m)$

Backtrack Algorithm: We use the DPM matrix and trace back from $DPM[n][m]$ to reconstruct the longest common subsequences.

1. We start at $DPM[n][m]$

2. Two cases again: Case 1 $A[i] = B[j]$ meaning it must be added to the LCS and we move diagonally to $DPM[i-1][j-1]$ Case 2: $A[i]! = B[j]$ then $DPM[i][j]$ must be equal either to $DPM[i-1][j]$ or to $DPM[i][j-1]$. if $DPM[i][j] = DPM[i-1][j]$ then we move a cell up the matrix ignoring $A[i]$ else $DPM[i][j] = DPM[i][j-1]$ and we move a cell left in the matrix ignoring $B[j]$.
3. Follow the process until the stopping condition which is when either $i = 0 \vee j = 0$ which means we have traced back to the beginning of the matrix DPM.
We implement this algorithm using python and with a recursion technique with the stopping condition being arriving at the beginning of the matrix.
The complexity of this is $O(2^r)$ where r is the length of the longest common subsequence this is because at each step of the r steps we are taking we will have two optional paths to take along the matrix DPM.

2.2 Proof of Correctness of the Algorithm

Given two sequences A, B where $A = \{a_1, a_2, a_3, .., a_n\}, B = \{b_1, b_2, b_3, .., b_m\}, |A| = n, |B| = m, n = m \vee n! = m, n, m, a_i, b_j \in Z \ \forall i, j, i = 1, 2, 3..., n, j = 1, 2, 3, ..., m$ We will want to prove the algorithms below where LCS is responsible for extracting the size of the longest common subsequence and backtracking is responsible for extracting from the matrix used in LCS the common subsequences themselves.
LCS Algorithm:
I will prove the LCS algorithm by induction as follows:
First, $DPM[i][j]$ is defined as the length of the longest common subsequence of A up till i and B up till j.
For our base case we show that if $A = \varnothing$ then $DPM[0][j] = 0, \forall j$ or if $B = \varnothing$ then $DPM[i][0] = 0, \forall i$ which shows the correctness of the algorithm for the base case. Now I will assume correctness for all d, l where $d \leq i$ and $l \leq j$ and show that $DPM[i][j]$ is correct. Again we split this into two cases:
Case 1:
$A[i] = B[j]$ which means that: $DPM[i][j] = DPM[i-1][j-1] + 1$ which correctly preserves the relation ship because we assumed correctness for d, l meaning $DPM[i-1][j-1]$ is correctly computed meaning $DPM[i][j]$ is now also correctly computed.
Case 2:
$A[i]! = B[j]$ in this case we take $DPM[i][j] = max(DPM[i-1][j], DPM[i][j-1]$ which means we continue with the longest common subsequence computation of before which was determined correctly (by our assumption of the induction) meaning the next $DPM[i][j]$ which is determined in the way I explained above is also determined correctly.
■

Backtracking algorithm: To prove this algorithm we will take a similar approach as the previous algorithm, splitting into two cases. First we start at $DPM[n][m]$ where n, m are the sizes of A, B respectively.
Case 1: $A[i] = B[j]$ this means the numbers are equal and therefore are part of

the LCS therefore we will move diagonally across DPM to $DPM[i-1][j-1]$ and add $A[i]$ to the LCS being reconstructed. This is correct because we know that $DPM[i][j] = DPM[i-1][j-1]+1$ which is exactly the diagonal which we move on.
Case 2: $A[i]! = B[j]$ and from here we have two possible routes to take the first is move up $DPM[i-1][j]$ the second is to move left $DPM[i][j-1]$ according to the formula of $DPM[i][j] = max(DPM[i-1][j], DPM[i][j-1])$ which insures we retrace our steps. If they are both equal then we explore both paths to recover multiple longest common subsequences. We use a set and store the recovered longest common subsequences.
We stop when we reach either $DPM[0][j]$ or $DPM[i][0]$. This is correct because it follows the way the matrix was originally built in the LCS algorithm proof.
■

Note: When we actually code we will add parameter θ to limit the amount of longest common subsequences returned as well as a code which handles duplicates.

2.3 Complexity's Proof of Correctness

LCS Algorithm's Complexity:
In the LCS algorithm we construct the DPM matrix which size is $(n+1)\times(m+1)$ meaning we iterate over all $(n+1)\times(m+1)$ cells each cell requires $O(1)$ time to compute therefore the actual complexity is $O(n \times m)$.
■ **Backtracking Algorithm's Complexity:**
In the backtracking algorithm we have to explore all paths starting from $DPM[n][m]$ if we're at case 1(same case as above) we iterate over to one cell which means if it follows a single path of an LCS we achieve a complexity of $O(r)$ where r is the size of the longest common subsequence. However such as in life it is rarely this simple(:) (It's a smiley face)).Therefore if multiple paths exist we need to branch out and we find ourselves in a situation where we have to follow two paths therefore we find that the worst case performance is where at every cell inside the longest common subsequence we have the follow two paths which split into two more paths and those two into another two and so on and so forth which means that we achieve a complexity of $O(2^r)$. Therefore we achieve that the upper bound for the backtracking algorithm is $O(2^r)$.
■

2.4 Iterative Step by Step Example

Here I will show a step by step example which we will iterate manually through. Let A, B be sequences where $A = \{1,2,3,6,2\}, B = \{2,4,3,6,1,3\}$ we can already see without running the algorithm that the longest common subsequence is $\{2,3,6\}$ of size 3 We construct the DPM matrix and achieve the following:

A/B	–	2	4	3	6	1	3
–	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1
2	0	1	1	1	1	1	1
3	0	1	1	2	2	2	2
6	0	1	1	2	3	3	3
2	0	1	1	2	3	3	3

Iteration 1: We start at $DPM[5][6], A = 2, B = 3$ and move one cell up to $DPM[4][6]$
Iteration 2: We start at $DPM[4][6], A[4] = 6, B[6] = 3$ and we move to $DPM[4][5]$ **Iteration 3:** We start at $DPM[4][5], A[4] = 6, B[5] = 1$ we continue to $DPM[4][4]$.
Iteration 4: We start at $DPM[4][4], A[4] = 6, B[4] = 6$ and we move to $DPM[3][3]$. We save 6 as it's visible its part of the LCS.
Iteration 5: We start at $DPM[3][3], A[3] = 3, B[3] = 3$ We save 3 as it's visible its part of the LCS and we move to $DPM[2][2]$.
Iteration 6: We start at $DPM[2][2], A[2] = 2, B[2] = 4$ and we move to $DPM[2][1]$
Iteration 7: We start at $DPM[2][1], A[2] = 2, B[1] = 2$ and we move to $DPM[1][0]$. We save 2 as it's visible it's part of the LCS.
Iteration 8: We stop the algorithm as we reached $DPM[0][1]$ and obtain the following LCS $\{2, 3, 6\}$.
Note: in the code we obtain the sorting of the list using a built in sorting function with a complexity of $O(nlogn)$.

3 Question 2 - LIS Algorithm

Like the previous problem we take a dynamic programming approach in order to compute the length of Longest Increasing Subsequence and some sort of reconstruction algorithm to reconstruct the longest increasing subsequence with help from the dynamic programming methodology. Here again, dynamic programming helps us by reducing problems to smaller sub problems making the bigger problem easier to solve.
Note: We will use the definitions here described in the first section of the paper, Project Description and Definitions specifically we will use the definitions defined in Question 2 as well as the ones which are defined in the first section of the Project Description and Definitions.

3.1 Algorithm Explanation and Complexity

LIS Algorithm
1. We create two lists of length n (The length of A) which we will denote as DL and *count* where $DL[i]$ is the length of longest increasing sequence of A up until i and *count* stores the number of longest increasing subsequences of A up till i.

Each valid $A[i]$ where $C[i] = 1$ then $DL[i] = 1 = count[i]$ as an initialization.
2. For each element $A[i]$:
a.We iterate over previous elements $j < i$
b. Only consider elements where $C[j] = 1$.
c. Check if $A[j] < A[i]$ if true then
c.a. $DL[i] = DL[j] + 1$
c.b. Reset count $count[i] = count[j]$ (In case where we find a new Longest increasing subsequence)
c.c. If it's the same Longest Increasing subsequence then $count[i] + = count[j]$(Pardon me for the Programming notation).
3. The steps below are repeated until we iterate over all elements.
The maximum value of DL gives the length of the longest increasing subsequence and the sum of all $count[i]$ at places where $DL[i] = max(DL)$ gives us the amount of the longest increasing subsequences.
This is achieved in $O(n^2)$ complexity.

Extracting the LIS Algorithm:
1. We create a *paths* dictionary which stores all longest increasing subsequences where $paths[i]$ stores the LIS of A ending at i. We initialize by storing $paths[i] = \{A[i]\}, C[i] = 1$
2. If $A[j] < A[i], j < i$ and extends a longest increasing subsequence then $paths[i] = \{seq + [A[i]] \text{ for } seq \in paths[j]\}$. Else it maintains the longest increasing subsequence length then $paths[i] = paths[i] \bigcup \{seq + [A[i]] \text{ for } seq \in paths[j]\}$.
3. Continue doing the steps below until we finish iterating over the elements.
We then extract all longest increasing subsequences which satisfy being the size of $max(DL)$. I achieve to implement this algorithm in a complexity of $O(n^2)$ using an iterative approach rather than a recursive approach like in the previous algorithm.

3.2 Proof of Correctness of the Algorithm

LIS Algorithm: For the base case we take a look at how $DL[i] = 1, count[i] = 1$ this is true because every single element on his own is the longest increasing subsequence by itself. We will continue the proof by induction considering the base case.
We assume correctness for some $j < i$ and so we see that if $A[j] < A[i]$ and if $DL[j] + 1 > DL[i]$ then we have found a longest increasing subsequence and so we update $DL[i] = DL[j] + 1$ this update which at every step depends on the updates beforehand and since the base case is correct and we assume correctness up till j then we achieve that $DL[i]$ must also be correct due to $Dl[i]$ being updated using $DL[j]$ which is correct. Thus when we keep updating we achieve that $max(DL)$ is indeed the size of longest increasing subsequence of A. The *count* list's correctness is observed due to the way it's being updated if we have found a new longest increasing subsequence (meaning $DL[j] + 1 > DL[i]$ then we say that $count[i] = count[j]$ and this is true because we keep in mind how

the base case of *count* is also correct. We assume correctness up til j and so the second case where we obtain another longest increasing subsequence of A but of the same length of another increasing subsequence of A and so now we need to add $count[j]$ to $count[i]$ so we obtain $count[i] += count[j]$ therefore this also preserves the correctness of the solution.
■

Recovering the LIS Algorithm:
Here the proof is trivial the way *paths* is defined is as a dictionary which keeps adding into it increasing subsequences of A therefore the subsequence which is the maximum size in *paths* is also the longest increasing subsequence of A.(It satisfies $max(DL)$).
■

3.3 Complexity's Proof of Correctness

LIS Algorithm: Each iteration takes $O(1)$ time and when we run the algorithm we iteratively iterate over all elements of A while comparing them to past elements of A meaning we iterate using an outer loop and an inner loop. The inner loop runs at most i times while the outer loop runs n times meaning the inner loop for worst case runs n iterations which means we achieve a complexity of $O(n^2)$
■

Reconstruction the LIS Algorithm:
Here we achieve similar results because we run similarly like explained above however we in addition save the increasing subsequences therefore we achieve a similar complexity of $O(n^2)$.
■

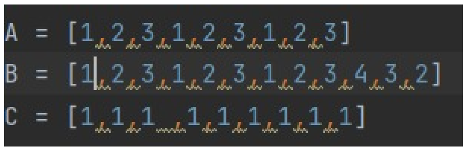
3.4 Iterative Step by Step Example

Let A be a sequence of numbers and C be an array of $0,1$ of the following:
$A = \{3, 10, 2, 1, 20\}, C = \{1, 1, 1, 1, 1\}$ and so we run the algorithm to find the longest increasing subsequence of A:
Iteration 1:
$i = 0 : A[i] = 3, C[i] = 1, DL = [1, 1, 1, 1, 1], count = [1, 1, 1, 1, 1], paths = \{0 : \{3\}\}, max(DL) = 1$
Iteration 2:
$i = 1 : A[i] = 10, 10 > 3, DL[i] = DL[j] + 1 = 2, count = [1, 1, 1, 1, 1], max(DL) = 2, paths = \{0 : \{3\}, 1 : \{3, 10\}\}$
Iteration 3:
$i = 2 : A[i] = 2, 10 > 2, DL[i] = 1, count = [1, 1, 1, 1, 1], max(DL) = 2, paths = \{0 : \{3\}, 1 : \{3, 10\}, 2 : \{2\}\}$
Iteration 4:
$i = 3 : A[i] = 1, 2 > 1, DL[i] = 1, count = [1, 1, 1, 1, 1], max(DL) = 2, paths = \{0 :$

$\{3\}, 1 : \{3, 10\}, 2 : \{2\}, 3 : \{1\}\}$
Iteration 5:
$i = 4 : A[i] = 20, 20 > 10 > 3 > 2 > 1, DL[i] = DL[1]+1 = 3, count = [1, 1, 1, 1, 1], max(DL) = 3, paths = \{0 : \{3\}, 1 : \{3, 10\}, 2 : \{2\}, 3 : \{1\}, 4 : \{3, 10, 20\}\}$.
We achieve the LIS of $\{3, 10, 20\}$ and the size of it being $max(DL) = 3$ and out of *count* we see that there is online one longest increasing subsequence of A.

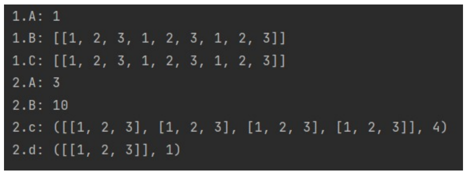
4 Running Examples.

The examples above are examples ran by my code for the project including inputs of the code and outputs. The code is the applied algorithms which were explained and proven in this document.



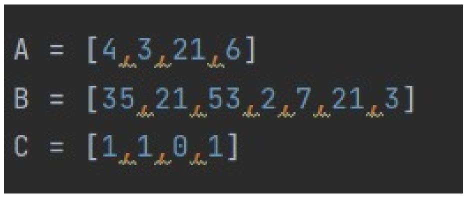
```
A = [1,2,3,1,2,3,1,2,3]
B = [1,2,3,1,2,3,1,2,3,4,3,2]
C = [1,1,1  ,1,1,1,1,1,1]
```

Figure 1.1: Example 1 Input



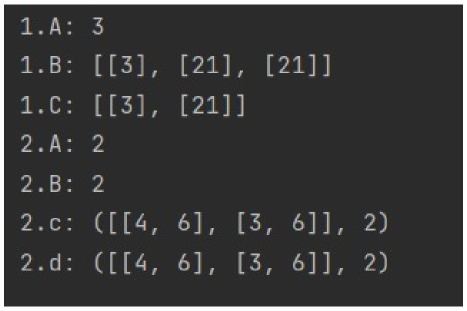
```
1.A: 1
1.B: [[1, 2, 3, 1, 2, 3, 1, 2, 3]]
1.C: [[1, 2, 3, 1, 2, 3, 1, 2, 3]]
2.A: 3
2.B: 10
2.c: ([[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]], 4)
2.d: ([[1, 2, 3]], 1)
```

Figure 1.2: Example 1 Output



```
A = [4,3,21,6]
B = [35,21,53,2,7,21,3]
C = [1,1,0,1]
```

Figure 1.3: Example 2 Input



```
1.A: 3
1.B: [[3], [21], [21]]
1.C: [[3], [21]]
2.A: 2
2.B: 2
2.c: ([[4, 6], [3, 6]], 2)
2.d: ([[4, 6], [3, 6]], 2)
```

Figure 1.4: Example 2 Output

5 Brief Code Explanations

In this section I will go briefly over the code with simple explanation of what each function does.

5.1 Question 1- LCS Algorithm

5.1.1 1.A

```
def number_of_lcs(A,B):
   **Creation of the DPM matrix**
   def backtrack(i,j,path,seen,indices):
   **Back tracking algorithm**

   return length(lcs_sequences)
```

5.1.2 1.B

We use the same function as above but we now return the sequences themselves with a θ parameter activation.

```
def all_lcs(A,B,Theta):
   **Creation of the DPM matrix**
   def backtrack(i,j,path,seen,indices):
     **Back tracking algorithm**
   **Theta Parameter Usage**
   **Sorting Algorithm applied on the sequences
   return (lcs_sequences)
```

5.1.3 1.C

As before we use the same algorithm/function only now we return the sequences themselves without duplicates and with a θ parameter usage.

```
def all_unique_lcs(A,B,Theta):
   **Creation of the DPM matrix**
   def backtrack(i,j,path,seen,indices):
     **Back tracking algorithm**
   **Theta Parameter Usage**
   **Removal of Duplicates
   **Sorting Algorithm applied on the sequences
   return (lcs_sequences)
```

5.2 Question 2- LIS Algorithm

5.2.1 2.A

```
def length_of_lis(A,C):
**Creation of the DL List & Updating it **
return max(DL)
```

5.2.2 2.B

Here we take the same approach as before but we add the *count* list in order to count the number of subsequences of A which match being the Longest increasing subsequences of A.

```
    def number_of_lis(A,C):
**Creation of the DL List & Updating it **
**Creation of the Count List & Updating it**
**Summing over the cells in count where the indexes match that DL[i] = max(DL)**
return (Sum of relevant_count)
```

5.2.3 2.C

We do the same here but activate a parameter θ to limit the amount of subsequences returned as well as use *paths* to reconstruct the longest increasing subsequences.

```
def all_lis(A,C,Theta)
**Creation of the DL List & Updating it**
**Creation of the paths dictionary & updating it**
**Theta parameter activation**
**Sorting Algorithm applied**
return lis_sequences
```

5.2.4 2.D

We do the same here as well but activate a parameter θ to limit the amount of subsequences returned as well as use *paths* to reconstruct the longest increasing subsequences and remove duplicates.

```
def all_unique_lis(A,C,Theta)
**Creation of the DL List & Updating it**
**Creation of the paths dictionary & updating it**
**Theta parameter activation**
**Removal of duplicates**
**Sorting Algorithm applied**
return lis_sequences
```

This is a brief explanation of the code of the project. The algorithms are applied in Python.