UNIT III

CHAPTER 6

Dynamic Programming

Syllabus

Introduction, What is Dynamic Programming Strategy? Properties of Dynamic Programming Strategy, Problems which can be solved using Dynamic Programming, Dynamic Programming Approaches, Examples of Dynamic Programming Algorithms, Understanding Dynamic Programming, Longest Common Subsequence.

Syllabus Topic : Introduction

6.1 Introduction

Invented in 1950 by Richard Bellman an U.S mathematician as a general method for optimizing multistage decision processes. It is a planning concept, which is a technique for solving problems with overlapping sub problems.

Syllabus Topic : What is Dynamic Programming Strategy ?

6.1.1 Dynamic Programming Strategy

- Dynamic Programming is a powerful technique that allows one to solve many different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time.
- It is used in optimization. Dynamic Programming solves problems by combining the solutions of different sub problem. Moreover, Dynamic Programming algorithm it is technique of reuse solution of problem which already solved.

Syllabus Topic : Properties of Dynamic Programming Strategy

6.1.2 Properties of Dynamic Programming Strategy

Overlapping sub-problems and **optimal substructure** are the two properties of problem tell us that given problem can be solved using dynamic programming.

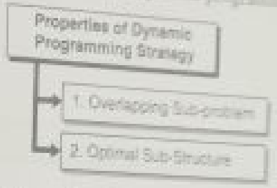
Properties of Dynamic Programming Strategy
1. Overlapping Sub-problem
2. Optimal Sub-Structure

Fig. C6.1 : Properties of dynamic programming strategy

➜ 1. Overlapping Sub-problem

Overlapping sub-problem means combines solutions to sub-problems. For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

➜ 2. Optimal Sub-Structure

Optimal Sub-Structure means, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

6.1.3 Ways to Solve Problem using dynamic Programming

There are two ways to solve problem using dynamic programming.

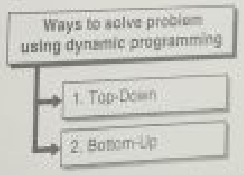

Fig. C6.2 : Ways to solve problem using dynamic programming

➜ 1. Top-Down

Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as **Memoization**.

➔ 2. Bottom-Up

- Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem.
- In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as **Dynamic Programming**.
- For example If a node **B** lies in the shortest path from a source node **A** to destination node **C**, then the shortest path from **A** to **C** is the combination of the shortest path from **A** to **B** , and the shortest path from **B** to **C**.

Syllabus Topic : Problems which can be solved using Dynamic Programming

6.2 Problems to Solved using Dynamic Programming

- Longest Common Subsequence
- Knapsack
- Matrix-chain multiplication.
- Bellman-Ford algorithm
- Floyd's All-Pairs shortest path algorithm
- Chain matrix multiplication
- Subset Sum
- Travelling salesman problem, and many more.

Syllabus Topic : Dynamic Programming Approaches

6.3 Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps –

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Syllabus Topic : Understanding Dynamic Programming

6.4 Understanding Dynamic Programming

To understand in better way let consider following Fibonacci series example. the Fibonacci sequence of integers: 1, 1, 2, 3, 5, 8, 13, 21, 33, 54,... following is recurrence relation for the same.

$$F(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F(n-1) + (n-2) & n > 1 \end{cases}$$

☞ **Program**

```
def fibonacci(n):
if(n <= 1):
return n
else:
return(fibonacci(n-1) + fibonacci(n-2))
n = int(input("Enter number of terms:"))
print("Fibonacci sequence:")
for i inrange(n):
printfibonacci(i)
```

Solving above recurrence give:

$$T(n) = T(n-1) + T(n-2) + \frac{1}{2} = (1+\sqrt{5})\pi = 2^n = O(2^n)$$

- Recall definition of Fibonacci numbers:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$
$$\text{Fib}(4) = \text{Fib}(3) + \text{Fib}(2)$$
$$= (\text{Fib}(2) + \text{Fib}(1)) + \text{Fib}(2)$$
$$\text{I"}> \;= ((\text{Fib}(1) + \text{Fib}(0)) + \text{Fib}(1)) + \text{Fib}(2)$$
$$= ((\text{Fib}(1) + \text{Fib}(0)) + \text{Fib}(1)) + (\text{Fib}(1) + \text{Fib}(0))$$

- Here, call to Fib(1) and Fib(0) is made multiple times.In the case of Fib(100) these calls would be count for million times.
- Hence there is lots of wastage of resources (CPU cycles and Memory for storing information on stack).

- In dynamic Programming all the subproblems are solved even those which are not needed, but in recursion only required subproblem are solved. So solution by dynamic programming should be properly framed to remove this ill-effect.
- For example, in combinatorics, C(n,m) = C(n-1,m) + C(n-1,m-1).

```
          1
         1 1
        1 2 1
       1 3 3 1
      1 4 6 4 1
    1 5 10 10 5 1
```

- In simple solution, one would have to construct the whole Pascal triangle to calculate C(5,4) but recursion could save a lot of time.
- Dynamic programming and recursion work in almost similar way in the case of non overlapping subproblem. In such problem other approaches could be used like "divide and conquer".

Syllabus Topic : Examples of Dynamic Programming Algorithms

6.5 Examples of Dynamic Programming Algorithms

6.5.1 Factorial Problem

- n! is the product of all integers between 11 and 1. The definition of recursive factorial can be give n as:

  n! = n × (n - 1)!

  1! = 1

  O! = 1

This definition can easily be converted to implementation. Here the problem is finding the value of n!, and the sub-problem is finding the value of (n - 1)!.

In the recursive case, when n is greater than 1, the function call itself to find the value of (n - 1)! and multiplies that with n. In the base case, when n is 0 or 1, the function simply returns 1.

Program

```
def factorial(n):
if n = = 0: return 1
return n*factorial(n-1)
print(factorial(6))
```

The recurrence for the above implementation can be given as: T(n) = n x T(n - 1) =O(n) Time Complexity: O(n). Space Complexity: O(n), recursive calls need a stack of size n.

- Time Complexity: O(n).
- Space Complexity: O(n).

Recursive calls need a stack of size n. In the above recurrence relation and implementation, for any n value, there are no repetitive calculations (no overlapping of sub problems) and the factorial function is not getting any benefits with dynamic programming. Now, let us say we want to compute a series of m! for some arbitrary value m. Using the above algorithm, for each such call we can compute it in O(m).

For example, to find both n! and m! , We can use the above approach, wherein the total complexity for finding n! and m! is O(m + n).

- Time Complexity: O(n + m)
- Space Complexity: O(max(m, n))
- Recursive calls need a stack of size equal to the maximum of m and n.

Improving

Now let us see how Dynamic problem reduces the complexity. From the above it can be seen that **fact(n)** is calculated from fact(n - 1) and n and nothing else. Rather than calling **fact(n)** every time, we can store the previous calculated values in a table and use these values to calculate a new value. This implementation can be given as:

Program

```
factTable = n
def factorial(n):
try: return factTable[n]
exceptKeyError:
if n = = 0:
factTable[0] = 1
return 1
else:
 factTable[n] = n * factorial(n− 1)
return factTable[n]
print(factorial( 10))
```

For simplicity, let us assume that we have already calculated n! and want to find m!.

For finding m! , we just need to see the table and use the existing entries if they are already computed. If m < n then we do not have to recalculate m!. If m > n then we can use n! and call the factorial on the remaining numbers only.

The above implementation clearly reduces the complexity to O(max(m, n)). This is because if the fact(n) is already there, then we are not recalculating the value again. If we fill these newly computed values, then the subsequent calls further reduce the complexity.

- Time Complexity: O(max(m, n)).
- Space Complexity: O(max(m, n)) for table.

6.5.2 Knapsack Problem

- In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.
- Hence, in case of 0-1 Knapsack, the value of xi can be either 0or 1, where other constraints remain the same.
- 0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

  The following examples will establish our statement.

Example 1

- Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

- Without considering the profit per unit weight (pi/wi), if we apply Greedy approach to solve this problem, first item A will be selected as it will contribute maximum profit among all the elements.
- After selecting item A, no more item will be selected. Hence, for this given set of items total profit is 24. Whereas, the optimal solution can be achieved by selecting items, B and C, where the total profit is 18 + 18 = 36.

Example 2

- Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio pi/wi. Let us consider that the capacity of the knapsack is W = 30 and the items are as shown in the following table.

Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

- Using the Greedy approach, first item A is selected. Then, the next item B is chosen. Hence, the total profit is 100 + 280 = 380. However, the optimal solution of this instance can be achieved by selecting items, B and C, where the total profit is 280 + 120 = 400.
- Hence, it can be concluded that Greedy approach may not give an optimal solution.
- To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a maximal weight of Winto his knapsack. There are n items and weight of ith item is w_i and the profit of selecting this item is p_i. What items should the thief take?

Dynamic-Programming Approach

- Let *i* be the highest-numbered item in an optimal solution S for W dollars. Then S = S - {i} is an optimal solution for W - w_i dollars and the value to the solution S is V_i plus the value of the sub-problem.
- We can express this fact in the following formula: define c[i, w] to be the solution for items 1,2, … , i and the maximum weight w.

The algorithm takes the following inputs

- The maximum weight W
- The number of items n
- The two sequences v = <v_1, v_2, …, v_n> and w = <w_1, w_2, …, w_n>

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
c[0, w] = 0
for i = 1 to n do
c[i, 0] = 0
for w = 1 to W do
if wi ≤ w then
if vi + c[i-1, w-wi] then
c[i, w] = vi + c[i-1, w-wi]
else c[i, w] = c[i-1, w]
else
c[i, w] = c[i-1, w]
```

- The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.
- If c[i, w] = c[i-1, w], then item i is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item i is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes $\theta(n, w)$ times as table c has $(n + 1).(w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

Syllabus Topic : Longest Common Subsequence Problems

6.6 Longest Common Subsequence Problems

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence

Let us consider a sequence $S = <s_1, s_2, s_3, s_4, \ldots, s_n>$.

A sequence $Z = <z_1, z_2, z_3, z_4, \ldots, z_m>$ over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, *X* and *Y* are two sequences over a finite set of elements. We can say that *Z* is a common subsequence of *X* and *Y*, if *Z* is a subsequence of both *X* and *Y*.

Longest Common Subsequence

- If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.
- The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics. It is also widely used by revision control systems, such as SVN and Git, for reconciling multiple changes made to a revision-controlled collection of files.

6.6.1 Naive Method

- Let X be a sequence of length m and Y a sequence of length n. Check for every subsequence of X whether it is a subsequence of Y, and return the longest common subsequence found.
- There are 2m subsequences of X. Testing sequences whether or not it is a subsequence of Y takes O(n) time. Thus, the naive algorithm would take O(n2m) time.

6.6.2 Dynamic Programming

- Let $X = < x_1, x_2, x_3, \ldots, x_m >$ and $Y = < y_1, y_2, y_3, \ldots, y_n >$ be the sequences. To compute the length of an element the following algorithm is used.

- In this procedure, table **C[m,n]** is computed in row major order and another table **B[m,n]** is computed to construct optimal solution.

Algorithm : LCS-Length-Table-Formulation (X, Y)

```
m := length(X)
n := length(Y)
for i = 1 to m do
C[i, 0] := 0
for j = 1 to n do
C[0, j] := 0
for i = 1 to m do
for j = 1 to n do
if x_i = y_j
C[i, j] := C[i - 1, j - 1] + 1
B[i, j] := 'D'
else
if C[i -1, j] ≥ C[i, j -1]
C[i, j] := C[i - 1, j] + 1
B[i, j] := 'U'
else
C[i, j] := C[i, j - 1] + 1
B[i, j] := 'L'
return C and B
Algorithm: Print-LCS (B, X, i, j)
if i = 0 and j = 0
return
if B[i, j] = 'D'
 Print-LCS(B, X, i-1, j-1)
Print(x_i)
else if B[i, j] = 'U'
 Print-LCS(B, X, i-1, j)
else
 Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of **X** and **Y**.

6.6.3 Analysis

To populate the table, the outer **for** loop iterates *m* times and the inner **for** loop iterates *n* times. Hence, the complexity of the algorithm is $O(m, n)$, where *m* and *n* are the length of two strings.