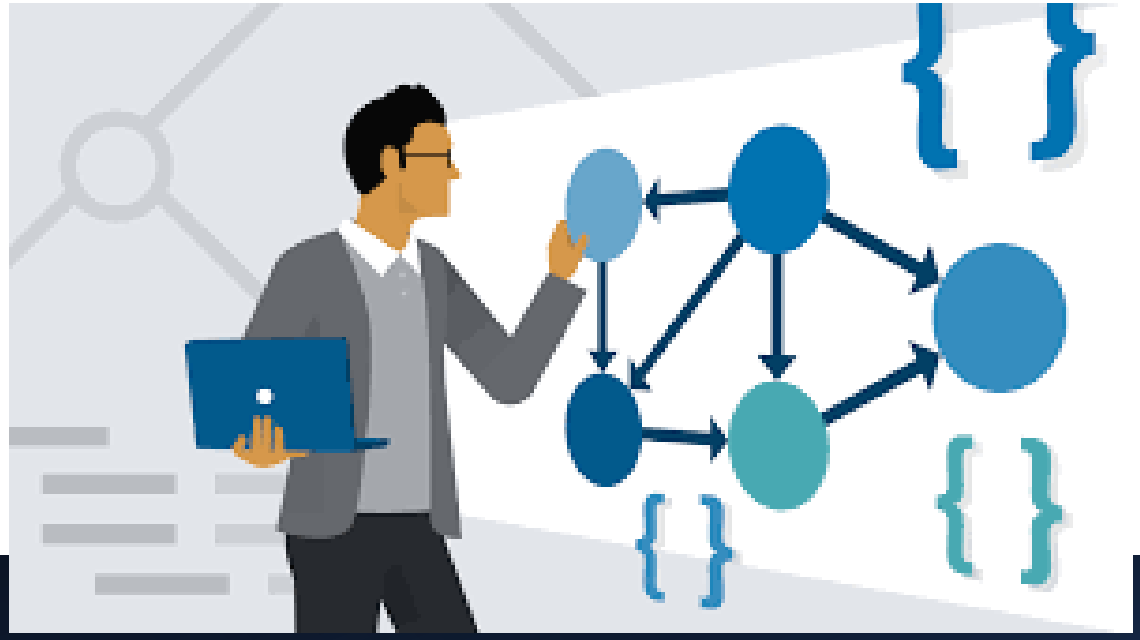
Algorithms

Lecture 10
Dynamic Programming



A. S. M. Sanwar Hosen

Email: sanwar@wsu.ac.kr

Date: 25 May, 2023

Dynamic Programming

- **Dynamic Programming (DP):** It is a problem solving technique that breaks down complex problems into overlapping subproblems and uses the solutions to those subproblems to build up the solution to the main problem.
- **Working Principle of DP**

✓ **Identifying the Optimal Substructure:** The problem must exhibit optimal substructure, which means that an optimal solution to the problem can be constructed from optimal solutions to its subproblems.

✓ **Defining the Recurrence Relation:** Once the optimal substructure is identified, the next step is to define a recurrence relation that expresses the solution to a larger problem in terms of solutions to its subproblems. The recurrence relation typically involves a recursive formulation.

✓ **Memoization or Tabulation:** It can be implemented using either Memoization or Tabulation.

a) Memoization: This involves storing the solutions to subproblems in a table or array. It uses top-down approach.

b) Tabulation: In tabulation, the solutions to subproblems are computed iteratively, starting from the smallest subproblems and building up to the larger problem. The solutions are stored in a table or array. It uses bottom-up approach.

✓ **Constructing the Solution:** Once the solutions to all necessary subproblems are computed and stored, the final solution to the original problem can be constructed by combining the solutions to the subproblems according to the recurrence relation.

✓ **Analyzing the Time and Space Complexity:** This analysis helps determine the efficiency of the DP algorithm and assess its feasibility for larger problem sizes.

Dynamic Programming

❑ Problem Solving Techniques of DP

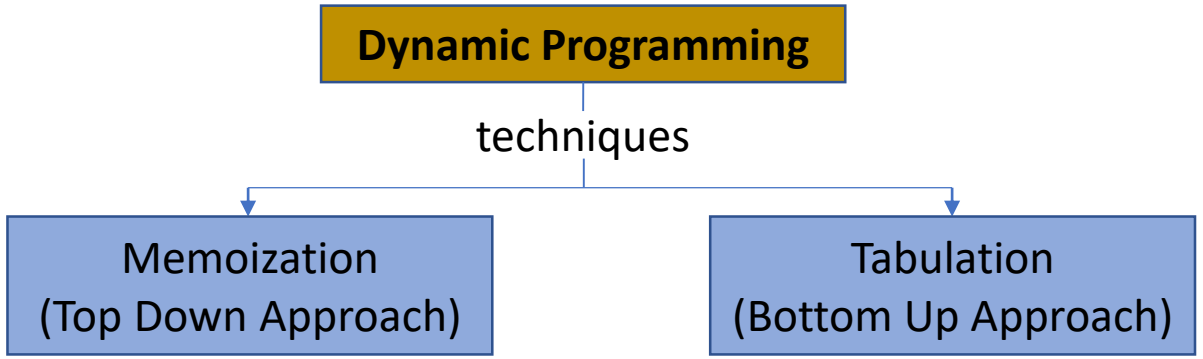


Fig. Dynamic Programming techniques to solve a dynamic problem.

- ✓ **Memoization:** The algorithm first checks if the solution is already available in the table. If so, it retrieves the solution. Otherwise, it computes the solution and stores it for future use. Memoization avoids redundant computations and improves the efficiency of DP algorithms. It uses top-down approach.
- ✓ **Tabulation:** The algorithm starts by solving the smallest subproblems and progressively computes solutions to larger subproblems until the final solution is reached. Tabulation eliminates recursion and relies on iteration, making it more space-efficient compared to Memorization. It uses bottom-up approach.

Dynamic Programming

❑ How to Solve a DP Problem?

To solve the problem dynamically, the following conditions are necessarily checked:

- ✓ **Overlapping Subproblems:** The presence of overlapping subproblems arises when the same subproblems need to be solved repeatedly in order to solve the main problem.
- ✓ **Optimal Substructure Property:** It is present in a problem when its optimal solution can be derived by combining the optimal solutions of its subproblems.

❑ Steps to Solve a DP Problem

Step 1: Determine whether the problem can be classified as a DP problem.

Step 2: Choose a concise state representation with minimal parameters.

Step 3: Create the state representation and define the relationship between states in the problem formulation.

Step 4: Apply the techniques of Tabulation or Memoization to optimize the solution.

Dynamic Programming

- **Step 1: How to Classify a Problem as a DP Problem?**

✓ DP is applicable to problems involving optimization, counting arrangements, and probability, where maximizing or minimizing quantities are required, providing efficient solutions.

✓ DP problems exhibit the overlapping subproblems property universally, while the majority of classic DP problems also fulfill the optimal substructure property. Identifying these properties in a problem ensures its suitability for a DP solution.

- **Step 2: Deciding the State:** DP problems primarily revolve around the state and its transition. The initial phase, involving careful selection of the state definition, is crucial as the state transition heavily relies on it.
- **Step 3: Formulating a Relation among the States:** The most challenging aspect of a DP problem lies in this step, which demands intuition, keen observation, and extensive practice.

❑ Step 3: Formulating a Relation among the States

Example: For a given set of three numbers {1, 3, 5}, the objective is to determine the total count of distinct ways in which the number N can be formed by summing the given numbers, considering repetitions and different arrangements.

✓ **The problem can be solved by following the steps:**

1. Select an appropriate state for the given problem.
2. The state will be determined by the variable N, as it enables the identification of all subproblems.
3. The DP state will be represented as state (N), where state (N) signifies the total number of arrangements needed to construct N using the elements 1, 3, and 5. Identify the appropriate state expression for the problem.
4. Next, we need to compute the state (N).

The number of ways to form the sum 6 using the given set is 8:

$1+1+1+1+1+1$
$1+1+1+3$
$1+1+3+1$
$1+3+1+1$
$3+1+1+1$
$3+3$
$1+5$
$5+1$

Dynamic Programming

❑ How to Compute the State?

Since we can only utilize the numbers 1, 3, or 5 to construct a given number N, let's assume that we have the results for $N = 1, 2, 3, 4, 5,$ and 6.

Assuming we already have the solution for: *state* (n=1), *state* (n=2), *state* (n=3), … , $state\ (n = 6)$

Now, let's determine the result for $state\ (n = 7)$. Since we can only add the numbers 1, 3, and 5, there are 3 distinct ways to achieve a total sum of 7:

1. By adding 1 to all possible combinations of $state(n = 6)$, we explore the different ways to obtain $state(n = 7)$:

$[1 + 1 + 1 + 1 + 1 + 1] + 1$
$[1 + 1 + 1 + 3] + 1$
$[1 + 1 + 3 + 1] + 1$
$[1 + 3 + 1 + 1] + 1$
$[3 + 1 + 1 + 1] + 1$
$[3 + 3] + 1$
$[1 + 5] + 1$
$[5 + 1] + 1$

2. By adding 3 to all possible combinations of $state(n = 4)$, we explore the different ways to obtain $state(n = 7)$:

$[1 + 1 + 1 + 1] + 3$
$[1 + 3] + 3$
$[3 + 1] + 3$

3. By adding 5 to all possible combinations of $state(n = 2)$, we explore the different ways to obtain $state(n = 7)$:

$[1 + 1] + 5$

Take a moment to carefully analyze and verify that the three cases mentioned above encompass all possible ways to achieve a sum total of 7. Hence, we can conclude that the result for:

$$state(7) = state(6) + state(4) + state(2)$$

or alternatively, $state(7) = state(6) + state(7 - 3) + state(7 - 5)$.

In general, the $state(n) = state(n - 1) + state(n - 3) + state(n - 5)$.

Dynamic Programming Implementation in Python (1)

- **Python Program to Returns the Number of Arrangements to form 'n'**

```
# Implementation of the number arrangement to make sum 'n' problem in Python
# Define the function named numberOfArrangement
def numberOfArrangement(n): # n is the sum of the arrangement
    if(n < 0): # sum is less than 0
        return 0
    if(n == 0): # sum is == to 0
        return 1
    return numberOfArrangement(n-1)+numberOfArrangement(n-3)+numberOfArrangement(n-5) # Otehrwise

print('Output:\n')
print('The total number of arrangements to make the sum 7 is:')
# call the function here for any given sum
numberOfArrangement(7)
```

```
Output:

The total number of arrangements to make the sum 7 is:

12
```

Dynamic Programming Implementation in Python (2)

- **Step 4: Adding Memoization or Tabulation for the State:** The most straightforward aspect of a DP solution is the ability to store the state's solution in memory, enabling easy access whenever that state is required again.

```python
# Implementation of number of arrangements to form 'n' in Python
def numberOfArrangement(n):
    dp = [0] * (n + 1)
    dp[0] = 1

    for i in range(1, n + 1):
        for num in [1, 3, 5]:
            if i >= num:
                dp[i] += dp[i - num]

    return dp[n]

# Test the program
n = 7
distinct_ways = numberOfArrangement(n)
print('Output:\n')
print("Distinct ways to form", n, "is:", distinct_ways)
```

```
Output:

Distinct ways to form 7 is: 12
```

Dynamic Programming Examples

❑ Few More Examples of DP Problems

1. Fibonacci Sequence: Computing the nth Fibonacci number can be efficiently solved using DP. By defining the recurrence relation $F(n) = F(n-1) + F(n-2)$, with base cases $F(0) = 0$ and $F(1) = 1$, the Fibonacci sequence can be calculated iteratively or recursively with Memoization.

2. Coin Change Problem: Given a set of coins with different denominations and a target amount, determine the minimum number of coins needed to make the target amount. This problem can be solved using DP by defining the recurrence relation as the minimum of $(1 + minCoins(target - coin))$ over all coins. The DP table stores the minimum number of coins for each target amount.

3. 0/1 Knapsack Problem: Given a knapsack with a limited capacity and a set of items with their weights and values, determine the maximum value that can be obtained by selecting items to fit into the knapsack without exceeding its capacity. This problem can be solved using DP by building a 2D table where each cell represents the maximum value that can be obtained with a specific weight and a subset of items.

Dynamic Programming Implementation in Python (3)

❑ Finding nth Fibonacci Number Implementation in Python

Computing the nth Fibonacci number can be efficiently solved using DP. By defining the recurrence relation $F(n)=F(n-1)+F(n-2)$, with base cases $F(0)=0$ and $F(1)=1$, the Fibonacci sequence can be calculated iteratively or recursively with Memoization.

```
# Computing the nth Fibonacci number in Python
# Define the function named fibonacci
def fibonacci(n):
    if (n <= 1): # Base case
        return n
    # Otherwise
    x = fibonacci(n - 1)
    y = fibonacci(n - 2)
    return x + y

# Define the value of n
n = 6;

# call the function
print('Output:\n')
print("The nth Fibonacci number is:",fibonacci(n))
```

```
Output:

The nth Fibonacci number is: 8
```

Dynamic Programming Implementation in Python (4)

❑ Coin Change Problem Implementation in Python

Given a set of coins with different denominations and a target amount, determine the minimum number of coins needed to make the target amount. This problem can be solved using DP by defining the recurrence relation as the minimum of (1+$minCoins(target - coin)$) over all coins. The DP table stores the minimum number of coins for each target amount.

```python
# Minimum number of coins to make a sum implementation in Python
# Define the function named minCoins
def minCoins(coins, target):
    # Create a DP table to store the minimum number of coins for each target amount
    dp = [float('inf')] * (target + 1)
    dp[0] = 0

    # Compute the minimum number of coins for each target amount
    for i in range(1, target + 1):
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], 1 + dp[i - coin])

    return dp[target] if dp[target] != float('inf') else -1

# Test the program
coins = [1, 3, 5]
target = 7
min_num_coins = minCoins(coins, target)
print('Output:\n')
print("Minimum number of coins needed to make", target, "is:", min_num_coins)
```

```
Output:

Minimum number of coins needed to make 7 is: 3
```

Dynamic Programming Implementation in Python (5)

❑ 1/0 Knapsack Problem Implementation in Python

Given a knapsack with a limited capacity and a set of items with their weights and values, determine the maximum value that can be obtained by selecting items to fit into the knapsack without exceeding its capacity. This problem can be solved using DP by building a 2D table where each cell represents the maximum value that can be obtained with a specific weight and a subset of items.

```python
# Knapsack problem implementation in Python
# Defined the function named knapsack
def knapsack(items, capacity):
    n = len(items)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        weight, value = items[i - 1]
        for w in range(1, capacity + 1):
            if weight > w:
                dp[i][w] = dp[i - 1][w]
            else:
                dp[i][w] = max(dp[i - 1][w], value + dp[i - 1][w - weight])
    return dp[n][capacity]

# Test the program
items = [(2, 3), (3, 4), (4, 5), (5, 8), (9, 10)]
capacity = 10
max_value = knapsack(items, capacity)
print('Output:\n')
print("Maximum value that can be obtained:", max_value)
```

```
Output:

Maximum value that can be obtained: 15
```