Hands-on Activity 2.1 : Dynamic Programming

Objective(s):

This activity aims to demonstrate how to use dynamic programming to solve problems.

Intended Learning Outcomes (ILOs):

- Differentiate recursion method from dynamic programming to solve problems.
- Demonstrate how to solve real-world problems using dynamic programming

Resources:

- Jupyter Notebook

Procedures:

1. Create a code that demonstrate how to use recursion method to solve problem

1. Create a program codes that demonstrate how to use dynamic programming to solve the same problem

Question:

Explain the difference of using the recursion from dynamic programming using the given sample codes to solve the same problem

Type your answer here: Recursion and dynamic programming both solve the knapsack problem, but DP is much faster and more efficient.

In recursion, the function keeps calling itself for smaller cases until it reaches the base case. It checks all possible ways to pick items, but this leads to repeated calculations, making it very slow for large inputs. Since it keeps creating new function calls, it also uses a lot of memory, which can cause stack overflow if the input is too big.

Dynamic programming (DP), on the other hand, stores previous results in a table and builds the solution step by step. Instead of recalculating the same values, it reuses them, making it much faster.

In short, recursion is simple but slow, while DP is a better choice for large problems because it avoids unnecessary calculations.

1. Create a sample program codes to simulate bottom-up dynamic programming

1. Create a sample program codes that simulate tops-down dynamic programming

Question:

Explain the difference between bottom-up from top-down dynamic programming using the given sample codes

Type your answer here:

Bottom-up and top-down dynamic programming both solve the knapsack problem, but they do it in different ways.

Bottom-up (Tabulation) builds the solution step by step using a table. It starts from the smallest subproblems and fills up the table using loops. Since it doesn't use recursion, it is faster and uses less memory.

Top-down (Memoization) solves the problem using recursion and stores already computed values to avoid repeating work. It starts from the main problem and breaks it into smaller parts, storing results in a table. However, since it uses recursion, it needs more memory for function calls.

In short, bottom-up uses loops and is more memory-efficient, while top-down uses recursion and may take more memory. Both make the knapsack problem faster, but bottom-up is better for larger inputs.

0/1 Knapsack Problem

- **Analyze three different techniques to solve knapsacks problem**

1. **Recursion**
2. **Dynamic Programming**
3. **Memoization**

In [1]:

```
#sample code for knapsack problem using recursion
def rec_knapSack(w, wt, val, n):

  #base case
  #defined as nth item is empty;
  #or the capacity w is 0
  if n == 0 or w == 0:
    return 0

  #if weight of the nth item is more than
  #the capacity W, then this item cannot be included
  #as part of the optimal solution
  if(wt[n-1] > w):
    return rec_knapSack(w, wt, val, n-1)

  #return the maximum of the two cases:
  # (1) include the nth item
  # (2) don't include the nth item
  else:
    return max(
        val[n-1] + rec_knapSack(
            w-wt[n-1], wt, val, n-1),
            rec_knapSack(w, wt, val, n-1)
    )
```

In [2]:

```
#To test:
val = [60, 100, 120] #values for the items
wt = [10, 20, 30] #weight of the items
w = 50 #knapsack weight capacity
n = len(val) #number of items

rec_knapSack(w, wt, val, n)
```

Out[2]:

```
220
```

In [4]:

```
#Dynamic Programming for the Knapsack Problem
def DP_knapSack(w, wt, val, n):
  #create the table
  table = [[0 for x in range(w+1)] for x in range (n+1)]

  #populate the table in a bottom-up approach
  for i in range(n+1):
    for w in range(w+1):
      if i == 0 or w == 0:
        table[i][w] = 0
      elif wt[i-1] <= w:
        table[i][w] = max(val[i-1] + table[i-1][w-wt[i-1]],
                          table[i-1][w])
  return table[n][w]
```

In [5]:

```
#To test:
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)

DP_knapSack(w, wt, val, n)
```

Out[5]:

```
220
```

In [19]:

```
#Sample for top-down DP approach (memoization)
  #initialize the list of items
  val = [60, 100, 120]
  wt = [10, 20, 30]
  w = 50
  n = len(val)

  #initialize the container for the values that have to be stored
  #values are initialized to -1
  calc =[[-1 for i in range(w+1)] for j in range(n+1)]


  def mem_knapSack(wt, val, w, n):
    #base conditions
    if n == 0 or w == 0:
      return 0
    if calc[n][w] != -1:
      return calc[n][w]

    #compute for the other cases
    if wt[n-1] <= w:
      calc[n][w] = max(val[n-1] + mem_knapSack(wt, val, w-wt[n-1], n-1),
                       mem_knapSack(wt, val, w, n-1))
      return calc[n][w]
    elif wt[n-1] > w:
      calc[n][w] = mem_knapSack(wt, val, w, n-1)
      return calc[n][w]

  mem_knapSack(wt, val, w, n)
```

Out[19]:

```
220
```

Code Analysis

The recursion method is the most easy. It breaks the problem into smaller subproblems by either including or excluding an item. However, it repeats the same calculations many times, which makes it very slow when dealing with large inputs.

The memoization (top-down DP) method improves recursion by storing previously calculated results in a table. This way, if the same subproblem appears again, it just looks up the answer instead of recalculating it. This makes it faster than pure recursion but still uses recursion, which can take up memory due to function calls.

The dynamic programming method is the most efficient. Instead of using recursion, it fills a table step by step from the smallest subproblems to the full problem. Since it doesn’t use function calls like recursion, it saves memory and runs much faster.

Seatwork 2.1

Task 1: Modify the three techniques to include additional criterion in the knapsack problems

In [23]:

```
#type your code here
#Recursion

def rec_knapSack(w, wt, val, n, max_items):
    # Base case: No items left, no weight left, or no items allowed
    if n == 0 or w == 0 or max_items == 0:
        return 0

    # If the current item is too heavy, skip it
    if wt[n-1] > w:
        return rec_knapSack(w, wt, val, n-1, max_items)

    # Try both choices: (1) Take the item, (2) Skip the item
    take = val[n-1] + rec_knapSack(w - wt[n-1], wt, val, n-1, max_items - 1)
    skip = rec_knapSack(w, wt, val, n-1, max_items)

    return max(take, skip)

# Example usage
val = [60, 100, 120]  # Item values
wt = [10, 20, 30]  # Item weights
w = 50  # Knapsack capacity
n = len(val)  # Number of items
max_items = 2  # Max number of items allowed

print(rec_knapSack(w, wt, val, n, max_items))



#Dynamic

def DP_knapSack(w, wt, val, n, max_items):
    # Create a 2D table where rows represent items (0 to n) and columns represent weight
(0 to w)
    table = [[0 for _ in range(w+1)] for _ in range(n+1)]

    # Populate the table using a bottom-up approach
    for i in range(1, n+1):  # Loop through each item
        for j in range(w, 0, -1):  # Loop through weight capacities (backward to avoid o
verwriting)
            if wt[i-1] <= j:  # If the current item's weight is within the capacity limi
t

                # Count how many items have been taken so far
                count_items = sum(1 for x in range(i) if table[x][j] != table[x][j-wt[i
-1]])

                if count_items < max_items:  # Ensure we do not exceed max_items
                    # Choose the max between taking or skipping the item
                    table[i][j] = max(val[i-1] + table[i-1][j-wt[i-1]], table[i-1][j])
                else:
                    # If max_items limit is reached, we cannot take the item
                    table[i][j] = table[i-1][j]
            else:
                # If the item is too heavy, we just inherit the previous value
                table[i][j] = table[i-1][j]

    # The last cell of the table contains the maximum value that can be obtained
    return table[n][w]


# Example usage
```

```
val = [60, 100, 120]  # Values of the items
wt = [10, 20, 30]      # Corresponding weights
w = 50                # Total weight capacity of the knapsack
n = len(val)          # Number of items
max_items = 2         # Maximum number of items allowed in the knapsack

# Call the function and print the result
print(DP_knapSack(w, wt, val, n, max_items))



#Memoization
# Initialize a memoization table (2D) with -1 (uncomputed values)
# Table size: (n+1) x (w+1) because we track items (0 to n) and weight (0 to w)
calc = [[-1 for _ in range(w+1)] for _ in range(n+1)]

def mem_knapSack(wt, val, w, n, max_items):

    #Base case: If no items left, no weight capacity left, or max_items limit reached
    if n == 0 or w == 0 or max_items == 0:
        return 0  # No value can be added

    #  Check if this state (n, w) has already been computed
    if calc[n][w] != -1:
        return calc[n][w]  # Return previously computed result



    # Case 1: If the current item's weight is MORE than the remaining capacity (w)
    # if We cannot take this item, so we move to the next item (n-1)
    if wt[n-1] > w:
        calc[n][w] = mem_knapSack(wt, val, w, n-1, max_items)

    else:
        #Case 2A: Take this item
        # - Add the item's value
        # - Reduce the remaining weight (w - wt[n-1])
        # - Decrease the number of remaining items we can take (max_items - 1)
        take = val[n-1] + mem_knapSack(wt, val, w - wt[n-1], n-1, max_items - 1)

        #Case 2B: Skip this item
        # - Keep the remaining weight and number of items unchanged
        skip = mem_knapSack(wt, val, w, n-1, max_items)

        # Choose the maximum value possible: either take it or skip it
        calc[n][w] = max(take, skip)

    # Store result in the memoization table and return it
    return calc[n][w]

# Testing the function
val = [60, 100, 120]  # Values of items
wt = [10, 20, 30]       # Weights of items
w = 50                 # Total weight capacity of the knapsack
n = len(val)           # Number of items available
max_items = 2          # Maximum number of items allowed to pick

#Compute and print the maximum value that can be obtained
print(mem_knapSack(wt, val, w, n, max_items))
```

```
220
220
220
```

Fibonacci Numbers

In [ ]:

```

```

Task 2: Create a sample program that find the nth number of Fibonacci Series using Dynamic Programming

Task 2: Create a sample program that find the nth number of Fibonacci Series using Dynamic Programming

In [16]:

```
#type your code here
def fibonacci(n):
    #Base cases: If n is 0 or 1, return n directly
    if n == 0:
        return 0
    if n == 1:
        return 1

    #Create a DP table to store Fibonacci numbers up to nth term
    fib_table = [0] * (n+1)   # Array of size (n+1), initialized with 0

    #Set base values:
    fib_table[0] = 0
    fib_table[1] = 1

    #Compute Fibonacci numbers iteratively
    for i in range(2, n+1):
        fib_table[i] = fib_table[i-1] + fib_table[i-2]

    #Return the nth Fibonacci number
    return fib_table[n]

# Example usage:
n = 10   # Find the 10th Fibonacci number
print(f"Fibonacci({n}) = {fibonacci(n)}")
```

```
Fibonacci(10) = 55
```

Supplementary Problem (HOA 2.1 Submission):

- **Choose a real-life problem**
- **Use recursion and dynamic programming to solve the problem**

In [17]:

```
# Recursive function to maximize enjoyment within a budget
def rec_travelBudget(budget, costs, fun_values, n):
    # Base case: No more budget or activities left
    if n == 0 or budget == 0:
        return 0

    # If the activity is too expensive, skip it
    if costs[n-1] > budget:
        return rec_travelBudget(budget, costs, fun_values, n-1)

    # Choose the maximum between including or skipping the activity
    else:
        return max(
            fun_values[n-1] + rec_travelBudget(budget - costs[n-1], costs, fun_values, n
-1),
            rec_travelBudget(budget, costs, fun_values, n-1)
        )

# Example usage
costs = [20, 50, 30]   # Cost of each activity
fun_values = [60, 100, 120]   # Enjoyment value
budget = 50   # Available budget
n = len(fun_values)

print("Maximum Enjoyment (Recursion):", rec_travelBudget(budget, costs, fun_values, n))
```

```
Maximum Enjoyment (Recursion): 180
```

In [13]:

```
def dp_travelBudget(budget, costs, fun_values, n):
    # Create a DP table
    table = [[0 for _ in range(budget + 1)] for _ in range(n + 1)]

    # Build the table bottom-up
    for i in range(n + 1):
        for b in range(budget + 1):
            if i == 0 or b == 0:
                table[i][b] = 0
            elif costs[i-1] <= b:
                table[i][b] = max(fun_values[i-1] + table[i-1][b - costs[i-1]], table[i-
1][b])
            else:
                table[i][b] = table[i-1][b]

    return table[n][budget]

costs = [20, 50, 30]  # Cost of each activity
fun_values = [60, 100, 120]  # Enjoyment value
budget = 50  # Available budget
n = len(fun_values)
# Example usage
print("Maximum Enjoyment (Dynamic Programming):", dp_travelBudget(budget, costs, fun_valu
es, n))
```

```
Maximum Enjoyment (Dynamic Programming): 180
```

Conclusion

In solving problems like the Knapsack Problem, we used three different methods which are Recursion, Memoization, and Dynamic Programming (DP). Recursion is the simplest because it just keeps breaking the problem into smaller parts, but it's very slow since it repeats calculations. Memoization improves recursion by saving answers in a table so we don't have to redo the same work, making it faster. Dynamic Programming (DP) takes a bottom-up approach, filling up a table step by step, which makes it the most efficient since it avoids recursion and extra memory use. In short, recursion is easy but slow, memoization is better but still uses recursion, and DP is the best for big problems because it's fast and memory-efficient

In [ ]:

```
#type your answer here
```

In [ ]:

```
```