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

In [4]:
```
# In recursion we could use a helper function to maintain state across recur
def factorial(n):
    def helper(n, accumulator):
        if n == 0:
            return accumulator
        else:
            return helper(n - 1, n * accumulator)

    return helper(n, 1)
# Example usage:
print(factorial(5))  # Output: 120
```

```
120
```

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

In [5]:
```
# In iteration we can maintain state using loop variables.
def factorial_iterative(n):
    accumulator = 1
    for i in range(1, n + 1):
        accumulator *= i
    return accumulator
# Example usage:
print(factorial_iterative(5))  # Output: 120
```

```
120
```

Question:

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

The main difference is that **recursion is dumb repetition** while **dynamic programming is smart storage**.

Recursion keeps solving the same small problems over and over. Like if you're figuring out the best items for your backpack, it keeps recalculating "should I take item 2 and 3?" again and again even though you already solved that before. It's simple to write but wastes tons of time.

Dynamic programming writes down answers as it goes. It solves "should I take item 2 and 3?" once, writes the answer in a table, and just looks it up next time. No repeat work. Much faster but needs more memory for the table.

So basically:
Recursion = keep solving same puzzle pieces
DP = solve each piece once and remember it

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

In [6]:
```
def fib_bottom_up(n):
    if n < 0:
        raise ValueError("n must be >= 0")
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]

# Example usage:
print("First 10 Fibonacci numbers (bottom-up):", [fib_bottom_up(i) for i in
```

```
First 10 Fibonacci numbers (bottom-up): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

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

In [7]:
```
def fib_top_down(n, memo=None):
    if memo is None:
        memo = {}

    if n <= 1:
        return n
    if n in memo:
        return memo[n]

    memo[n] = fib_top_down(n - 1, memo) + fib_top_down(n - 2, memo)
    return memo[n]

# Example usage:
print("First 10 Fibonacci numbers (top-down):", [fib_top_down(i) for i in ra
```

```
First 10 Fibonacci numbers (top-down): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Question:

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

Type your answer here:

Between the two dynamic programming methods, bottom-up (tabulation) and top-down (memoization), each has its own strengths.

Bottom-up works iteratively, building a solution from the smallest subproblems upward, like in our fib_bottom_up function. It runs in linear time, uses either O(n) space for a full table or O(1) space if we optimize, and has no risk of recursion limits because it doesn't use recursion at all. This approach is best when you need all subproblem results or want predictable performance.

Top-down, on the other hand, starts from the main problem and works its way down recursively while storing answers along the way, as in fib_top_down. It also runs in linear time but uses extra space for both the memo dictionary and the recursion stack. The advantage is that it only computes what's actually needed, which is helpful if many possible states are unnecessary. It feels more natural if you're already thinking recursively, but it can run into recursion depth issues on very large inputs.

In practice, pick bottom-up for maximum speed and to avoid recursion limits, and choose top-down when the recursive logic is clearer or when you can skip many unnecessary computations.

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 [3]:
```
#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 [ ]:
```
#To test:
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)

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

```
Out[ ]: 220
```

```
In [ ]:
#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[ ]: 220
```

Code Analysis

1. Recursion Answer - This is the basic brute force way. It tries every possible combination by either taking or leaving each item. It's easy to write and understand - you just keep breaking the problem into smaller pieces until you reach the base case. The big problem is it's super slow for more than a few items because it calculates the same things over and over.

2. Dynamic Programming Answer - This is the optimized table method. Instead of recalculating, it builds up answers in a table from the bottom. You solve all the small problems first, then use those answers to solve bigger ones. It's much faster than recursion because each calculation only happens once. The downside is it uses more memory for the table.

3. Memoization Answer - This is like smart recursion. You still use the recursive approach, but you save answers as you go in a memo (like a cheat sheet). Before calculating anything, you check if you already solved it. This gives you the speed of dynamic programming with the simplicity of recursion. Best of both worlds for medium-sized problems.

Seatwork 2.1

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

In [8]:
```python
#type your code here

#Recursion (additional criterion: limit number of items K)
def rec_knapSack_limit(w, wt, val, n, k):
    if n == 0 or w == 0 or k == 0:
        return 0
    if wt[n-1] > w:
        return rec_knapSack_limit(w, wt, val, n-1, k)
    include = val[n-1] + rec_knapSack_limit(w-wt[n-1], wt, val, n-1, k-1)
    exclude = rec_knapSack_limit(w, wt, val, n-1, k)
    return max(include, exclude)

#Dynamic (bottom-up) with item-count limit K
def DP_knapSack_limit(w, wt, val, n, K):
    table = [[[0 for _ in range(K+1)] for _ in range(w+1)] for _ in range(n+
    for i in range(1, n+1):
        for cap in range(w+1):
            for used in range(0, K+1):
                table[i][cap][used] = table[i-1][cap][used]
                if used > 0 and wt[i-1] <= cap:
                    table[i][cap][used] = max(
                        table[i][cap][used],
                        val[i-1] + table[i-1][cap-wt[i-1]][used-1]
                    )
    return table[n][w][K]

#Memoization (top-down) with item-count limit K
def mem_knapSack_limit(wt, val, w, n, k, memo=None):
    if memo is None:
        memo = {}
    key = (n, w, k)
    if key in memo:
        return memo[key]
    if n == 0 or w == 0 or k == 0:
        memo[key] = 0
        return 0
    if wt[n-1] > w:
        memo[key] = mem_knapSack_limit(wt, val, w, n-1, k, memo)
    else:
        include = val[n-1] + mem_knapSack_limit(wt, val, w-wt[n-1], n-1, k-1
        exclude = mem_knapSack_limit(wt, val, w, n-1, k, memo)
        memo[key] = max(include, exclude)
    return memo[key]

# Example usage:
```

```
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)
K = 2

print("Recursion (<=K items):", rec_knapSack_limit(w, wt, val, n, K))
print("Dynamic (bottom-up, <=K items):", DP_knapSack_limit(w, wt, val, n, K)
print("Memoization (<=K items):", mem_knapSack_limit(wt, val, w, n, K))
```

```
Recursion (<=K items): 220
Dynamic (bottom-up, <=K items): 220
Memoization (<=K items): 220
```

Fibonacci Numbers

In [9]:
```
# Task 2: Fibonacci numbers — recursion, bottom-up DP, memoization

# Naive recursion (exponential time)
def fib_recursive(n):
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)

# Bottom-up dynamic programming (iterative, linear time & space)
def fib_bottom_up(n):
    if n < 0:
        raise ValueError("n must be >= 0")
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Top-down with memoization (linear time with recursion)
def fib_memo(n, memo=None):
    if memo is None:
        memo = {0: 0, 1: 1}
    if n in memo:
        return memo[n]
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]

# Example usage:
n = 10
print("fib_recursive({}):".format(n), fib_recursive(n))
print("fib_bottom_up({}):".format(n), fib_bottom_up(n))
print("fib_memo({}):".format(n), fib_memo(n))
print("First 10 Fibonacci numbers (bottom-up):", [fib_bottom_up(i) for i in
```

```
fib_recursive(10): 55
fib_bottom_up(10): 55
fib_memo(10): 55
First 10 Fibonacci numbers (bottom-up): [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

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

```
In [10]: def fib_dp(n):
             if n < 0:
                 raise ValueError("n must be >= 0")
             if n <= 1:
                 return n
             a, b = 0, 1
             for _ in range(2, n + 1):
                 a, b = b, a + b
             return b

         def fib_sequence(n):
             if n < 0:
                 raise ValueError("n must be >= 0")
             if n == 0:
                 return [0]
             seq = [0, 1]
             for i in range(2, n + 1):
                 seq.append(seq[-1] + seq[-2])
             return seq

         # Example usage:
         n = 10
         print(f"fib_dp({n}) =", fib_dp(n))
         print("Sequence up to n:", fib_sequence(n))
```

```
fib_dp(10) = 55
Sequence up to n: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Supplementary Problem (HOA 2.1 Submission):

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

```
In [14]: # Recursion
         coins = [1, 3, 4]
         amount = 6
         def rec_min_coins(amount, coins, n):
             if amount == 0:
                 return 0
             if amount < 0 or n == 0:
                 return float('inf')
             include = 1 + rec_min_coins(amount - coins[n-1], coins, n)
             exclude = rec_min_coins(amount, coins, n-1)
             return min(include, exclude)

         def rec_min_coins_wrapper(amount, coins):
             res = rec_min_coins(amount, coins, len(coins))
             return res if res != float('inf') else -1
         print("Recursion (min coins) ->", rec_min_coins_wrapper(amount, coins))
```

```
Recursion (min coins) -> 2
```

In [12]:
```
# Dynamic Programming (bottom-up)
def dp_min_coins(amount, coins):
    # dp[x] = min coins needed to make amount x
    INF = float('inf')
    dp = [INF] * (amount + 1)
    dp[0] = 0
    for coin in coins:
        for x in range(coin, amount + 1):
            if dp[x - coin] + 1 < dp[x]:
                dp[x] = dp[x - coin] + 1
    return dp[amount] if dp[amount] != INF else -1

# Example usage:
coins = [1, 3, 4]
amount = 6
print("DP bottom-up (min coins) ->", dp_min_coins(amount, coins))
```

```
DP bottom-up (min coins) -> 2
```

Conclusion

In this activity, I learned about two main ways to solve problems: recursion and dynamic programming. I tried recursion, which is like breaking a big problem into smaller pieces that you solve the same way. It's simple to write and makes sense, but it can get really slow because it keeps solving the same small problems over and over again. Like with Fibonacci numbers, the regular recursive way takes forever for big numbers. Then I learned dynamic programming, which is basically a smarter version. Instead of doing the same work again and again, you save your answers as you go. There are two styles: bottom-up (where you start small and build up) and top-down (where you start with the big problem and remember answers as you go). I used both on the knapsack problem, coin change, and Fibonacci. The DP versions were way faster than plain recursion. For example, Fibonacci went from being super slow to almost instant! The big lesson is to me if your problem can be broken into smaller pieces that repeat, use dynamic programming. Choose bottom-up if you want to be efficient and avoid recursion limits, or choose top-down if it's easier to think about starting from the main problem. Both are much better than plain recursion for bigger problems.