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

2. 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:

-The dynamic programming uses range function or stores a results in table. On the other hand, recursion uses the function to call itself

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

4. 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:

0/1 Knapsack Problem

- Analyze three different techniques to solve knapsacks problem

1. Recursion
2. Dynamic Programming
3. Memoization

```
#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)
    )


#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)
```

```
220
```

```
#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]


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

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

```
220
```

Code Analysis

Type your answer here.

Seatwork 2.1

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

```
#type your code here
```

```
#Dynamic
```

```
#Memoization
```

```
#recursion
def rec_knapSack(w, wt, val, n, exp_date):

    # base case
    if n == 0 or w == 0:
        return 0

    if exp_date[n - 1] <= 0:
        return rec_knapSack(w, wt, val, n - 1, exp_date)

    # ignore if the weight of the current item is more than the capacity
    if wt[n - 1] > w:
        return rec_knapSack(w, wt, val, n - 1, exp_date)

    # include the current item or exclude it
    include = val[n - 1] + rec_knapSack(w - wt[n - 1], wt, val, n - 1, exp_date)
    exclude = rec_knapSack(w, wt, val, n - 1, exp_date)

    # choosing the option with the maximum valuetogether with expiration dates
    return max(include if exp_date[n - 1] > 0 else 0, exclude)


#test
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)
exp_date = [10, 5, 15]  #expiration dates in days

max_value = rec_knapSack(w, wt, val, n, exp_date)
print("Maximum value obtainable without expired items:", max_value)
```

```
Maximum value obtainable without expired items: 220
```

```
#dynamic
def DP_knapSack(w, wt, val, n, exp_date):
    # build the table
    table = [[0 for _ in range(w + 1)] for _ 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 and exp_date[i - 1] > 0:  # check exp date and weight
                table[i][w] = max(val[i - 1] + table[i - 1][w - wt[i - 1]],
                                 table[i - 1][w])
            else:
                table[i][w] = table[i - 1][w]  # remove expired or overweight items

    return table[n][w]


#test
val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)
exp_date = [10, 5, 15]  # dates in days

max_value = DP_knapSack(w, wt, val, n, exp_date)
print("Maximum value obtainable without expired items:", max_value)
```

```
Maximum value obtainable without expired items: 220
```

```
def mem_knapSack(wt, val, w, n, exp_date):
    calc = [[[-1 for _ in range(w + 1)] for _ in range(n + 1)] for _ in range(max(exp_date) + 1)]

    # base conditions
    for i in range(n + 1):
        for j in range(max(exp_date) + 1):
            calc[j][i][0] = 0
            calc[j][0][w] = 0

    # memoization table population
    for i in range(1, n + 1):
        for j in range(max(exp_date) + 1):  # iterate through possible expiration days
            for w in range(1, w + 1):
                if j + exp_date[i - 1] < 0 or j + exp_date[i - 1] >= max(exp_date) + 1 or wt[i - 1] > w:
                  calc[j][i][w] = calc[j][i - 1][w]
                else:
                    calc[j][i][w] = max(
                        val[i - 1] + calc[j + exp_date[i - 1]][i - 1][w - wt[i - 1]],
                        calc[j][i - 1][w],
                    )

    # maximum value
    return max(calc[j][n][w] for j in range(max(exp_date) + 1))


val = [60, 100, 120]
wt = [10, 20, 30]
w = 50
n = len(val)
exp_date = [10, 5, 15]  # dates in days

max_value = mem_knapSack(wt, val, w, n, exp_date)
print("Maximum value obtainable without expired items:", max_value)
```

```
Maximum value obtainable without expired items: 159
```

Fibonacci Numbers

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

```
#type your code here
def fibonacci(n):
  num = [-1]*(n+1)
  return val(n, num)

def val(n, num):
  if num[n] >= 0:
    return num[n]
  if (n==0 or n==1):
    nth = n
  else:
    nth = val(n -1, num) + val(n - 2, num)

  num[n] = nth
  return nth

n = int(input('input n: '))
nthnum = fibonacci(n)
print('The nth number is', nthnum)
```

```
input n: 5
The nth number is 5
```

Supplementary Problem (HOA 2.1 Submission):

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

I made a program that chooses the shortest route as possible and totals the shortest route.

```
#type your code here for recursion programming solution
def shortest_route(cities, distances, current_city, remaining_cities, total_distance):

    # base case
    if not remaining_cities:
        return total_distance, [city for city in cities]  # return the complete route

    shortest_route_found = None
    for next_city in remaining_cities:
        if distances[current_city][next_city] != 0:  # check if there is a connection
            new_remaining_cities = remaining_cities.copy()
            new_remaining_cities.remove(next_city)
            new_distance, new_route = shortest_route(
                cities, distances, next_city, new_remaining_cities, total_distance + distances[current_city][next_city]
            )
            if shortest_route_found is None or new_distance < shortest_route_found[0]:
                shortest_route_found = (new_distance, [current_city] + new_route)

    return shortest_route_found


#test
cities = ["A", "B", "C", "D"]

distances = [[0, 10, 15, 20],[10, 0, 35, 25],[15, 35, 0, 30],[20, 25, 30, 0]]
starting_city = 0
distance, route = shortest_route(cities, distances, starting_city, set(range(1, len(cities))), 0)

if distance is not None:
  print("Shortest route: ", route)
  print("Total distance: ", distance)
```

```
Shortest route:  [0, 1, 3, 'A', 'B', 'C', 'D']
Total distance:  65
```

```
#type your code here for dynamic programming solution
def shortest_route_dp(cities, distances, starting_city):
    n = len(cities)
    dp = [[float('inf')] * n for _ in range(1 << n)]  # initialize DP table
    dp[1 << starting_city][starting_city] = 0  # Mark start city with cost 0

    for mask in range(1, 1 << n):
        for city in range(n):
            if mask & (1 << city) > 0:  # city already visited in this mask
                for prev_city in range(n):
                    if mask & (1 << prev_city) > 0 and distances[prev_city][city] > 0:
                        # check previous city and connection exists
                        dp[mask][city] = min(dp[mask][city], dp[mask ^ (1 << city)][prev_city] + distances[prev_city][city])

    # find minimum distance and return the route
    min_dist = min(dp[-1])
    if min_dist == float('inf'):
        return ValueError("No route found.")

    visited_cities = [starting_city]
    for mask in range(1, 1 << n):
        if dp[-1][visited_cities[-1]] != dp[mask][visited_cities[-1]]:
            # find the city
            new_city = (mask ^ visited_cities[-1]) & ~(mask ^ (1 << n))
            visited_cities.append(new_city)

    return min_dist, visited_cities

print("Shortest route: ", route)
print("Total distance: ", distance)
```

```
Shortest route:  [0, 1, 3, 'A', 'B', 'C', 'D']
Total distance:  65
```

Conclusion

```
#type your answer here
```