Week 5 Solutions-

The king’s treasure hunt

```
def knapsack(n, W, items):
    # Initialize DP table
    dp = [[0] * (W + 1) for _ in range(n + 1)]

    # Fill the DP table
    for i in range(1, n + 1):
        wi, vi = items[i - 1]
        for w in range(W + 1):
            if wi <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - wi] + vi)
            else:
                dp[i][w] = dp[i - 1][w]

    # The bottom-right corner of the DP table will hold the maximum value
    return dp[n][W]

# Test cases

test_cases = [
    (4, 5, [(2, 3), (3, 4), (4, 5), (5, 6)]),  # Test Case 1
    (3, 10, [(3, 4), (6, 5), (7, 6)]),  # Test Case 2
    (5, 7, [(3, 4), (2, 3), (4, 5), (5, 8), (6, 7)]),  # Test Case 3
    (3, 4, [(1, 2), (2, 3), (3, 4)]),  # Test Case 4
    (6, 8, [(1, 1), (2, 3), (3, 4), (2, 5), (5, 8), (6, 7)]),  # Test Case 5
    (4, 3, [(2, 1), (2, 2), (1, 3), (2, 4)])  # Test Case 6
]
```

```
# Running test cases

for i, (n, W, items) in enumerate(test_cases, 1):

    result = knapsack(n, W, items)

    print(f"Test Case {i}: {result}")
```

The Lord Abhay's Coin Collection

```
def coinChange(coins, target):
# Initialize the dp array, size of target + 1, filled with infinity
dp = [float('inf')] * (target + 1)
dp[0] = 0 # Base case: 0 coins needed to make 0

# For each coin, try to update dp array
for coin in coins:
for amount in range(coin, target + 1):
dp[amount] = min(dp[amount], dp[amount - coin] + 1)

# If dp[target] is still infinity, it's impossible to make the target
return dp[target] if dp[target] != float('inf') else -1

# Test cases

test_cases = [
([1, 2, 5], 11), # Test Case 1
([2, 5], 3), # Test Case 2
([1, 3, 4, 5], 7), # Test Case 3
([1, 3, 4], 6), # Test Case 4
([2], 9), # Test Case 5
([1, 3, 5], 12) # Test Case 6
]

# Running test cases
for i, (coins, target) in enumerate(test_cases, 1):
result = coinChange(coins, target)
print(f"Test Case {i}: {result}")
```

The Princess's Quest

```
import heapq
```

```
def shortestPath(n, m, paths, start, end):
# Create adjacency list
adj = {i: [] for i in range(1, n + 1)}

for u, v, d in paths:
adj[u].append((v, d))
adj[v].append((u, d)) # since the graph is undirected

# Distance array to store minimum difficulty to reach each node
dist = [float('inf')] * (n + 1)
dist[start] = 0

# Min-heap priority queue for Dijkstra's algorithm
pq = [(0, start)] # (distance, node)

while pq:
current_dist, node = heapq.heappop(pq)

if current_dist > dist[node]:
continue

# Check neighbors
for neighbor, weight in adj[node]:
new_dist = current_dist + weight
if new_dist < dist[neighbor]:
dist[neighbor] = new_dist
heapq.heappush(pq, (new_dist, neighbor))

# Return the shortest path to the destination node
return dist[end] if dist[end] != float('inf') else -1

# Test cases

test_cases = [
(4, 5, [(1, 2, 2), (1, 3, 1), (2, 3, 3), (2, 4, 1), (3, 4, 5)], 1, 4), # Test Case 1
(3, 3, [(1, 2, 4), (2, 3, 1), (1, 3, 5)], 1, 3), # Test Case 2
(4, 4, [(1, 2, 2), (2, 3, 3), (3, 4, 4), (1, 4, 8)], 1, 4), # Test Case 3
(3, 3, [(1, 2, 5), (2, 3, 7), (1, 3, 10)], 1, 3), # Test Case 4
(4, 3, [(1, 2, 2), (2, 3, 3), (1, 4, 6)], 1, 3), # Test Case 5
(4, 3, [(1, 2, 1), (2, 3, 2), (3, 4, 1)], 1, 4) # Test Case 6
]

# Running test cases
for i, (n, m, paths, start, end) in enumerate(test_cases, 1):
result = shortestPath(n, m, paths, start, end)
print(f"Test Case {i}: {result}")
```

The Ancient Library’s Code

```
def fibonacci_sum(n):
# Base case for the first Fibonacci number
if n == 1:
return 0

# Initialize the first two Fibonacci numbers
a, b = 0, 1
total_sum = a + b

# Loop to compute Fibonacci numbers and sum them
for i in range(2, n):
a, b = b, a + b
total_sum += b

return total_sum

# Test cases

test_cases = [
5, # Test Case 1
7, # Test Case 2
3, # Test Case 3
10, # Test Case 4
1 # Test Case 5
]

# Running test cases
for i, n in enumerate(test_cases, 1):
result = fibonacci_sum(n)
print(f"Test Case {i}: {result}")
```

The Secret Letters of the Lost Kingdom

```
def longestCommonSubsequence(str1, str2):
m = len(str1)
n = len(str2)

# Initialize a 2D DP table with zeros
dp = [[0] * (n + 1) for _ in range(m + 1)]

# Build the DP table
```

```
for i in range(1, m + 1):
for j in range(1, n + 1):
if str1[i - 1] == str2[j - 1]:
dp[i][j] = dp[i - 1][j - 1] + 1
else:
dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

# The length of the longest common subsequence is at the bottom-right corner of the table
return dp[m][n]

# Test cases

test_cases = [
("ABCBDAB", "BDCAB"), # Test Case 1
("AXYT", "AYZX"), # Test Case 2
("AGGTAB", "GXTXAYB") # Test Case 3
]

# Running test cases
for i, (str1, str2) in enumerate(test_cases, 1):
result = longestCommonSubsequence(str1, str2)
print(f"Test Case {i}: {result}")
```