Experiment 1:

Objective: To implement and analyze various sorting algorithms—Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort—and to study their efficiency and performance characteristics through code, input/output examples, and result discussion.

Theory:

Algorithm	Best	Average	Worst	Space
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)

Bubble Sort Algorithm:

1. Loop through the array.
2. For each pass, compare all adjacent elements.
3. If left > right, swap. Continue until entire array is sorted.

Code

```
def bubble_sort(arr):
    n=len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j]>arr[j+1]:
                arr[j], arr[j+1]= arr[j+1], arr[j]
                swapped = True
        if(swapped==False):
            break
arr=[19,8,12,9,5,7]
bubble_sort(arr)
print("Sorted Array: ")
for i in range(len(arr)):
    print("%d, "%arr[i],end="")
```

Selection Sort Algorithm:

1. Loop from start to end.
2. Find the minimum in the unsorted part and swap it with the current position.

Code:

```
def selection_sort(arr):
    n=len(arr)
    for i in range(n-1):
        min_idx = i
        for j in range(i+1, n):
            if arr[j]<arr[min_idx]:
                min_idx=j
        arr[i], arr[min_idx]= arr[min_idx], arr[i]
arr=[19,8,12,9,5,7]
selection_sort(arr)
print("Sorted Array: ")
for i in range(len(arr)):
    print("%d, "%arr[i],end="")
```

Insertion Sort Algorithm:

1. Start from the second element.
2. Insert it in the correct position among previously sorted elements.

Code:

```
def insertion_sort(arr):
    for i in range(1,len(arr)):#start from
        key = arr[i]
        j=i-1
        while j>=0 and key < arr [j]:
            arr[j+1]=arr[j]
            j-=1
        arr[j+1] = key
arr=[12,11,13,5,6]
insertion_sort(arr)
for i in range(len(arr)):
    print("%d, "%arr[i],end="")
```

Merge Sort Algorithm:

1. If the array has more than one element, split it into two halves.
2. Recursively sort each half using Merge Sort.
3. Merge the two sorted halves back together by comparing elements and placing them in order.

Code:

```
def mergeSort(arr):
    if len(arr)<=1:
        return arr
    mid = len(arr)//2
    left_half = mergeSort(arr[:mid])
    right_half = mergeSort(arr[mid:])
    return merge(left_half, right_half)
def merge(left, right):
    merged = []
    i=j=0
    while i<len(left) and j<len(right):
        if left[i]<=right[j]:
            merged.append(left[i])
            i+=1
        else:
            merged.append(right[j])
            j+=1
    merged.extend(left[i:])
    merged.extend(right[j:])
    return merged
arr=[38,27,43,10,19]
output = mergeSort(arr)
for i in output:
    print(i, end=" ")
```

Quick Sort Algorithm:

1. Choose pivot.
2. Partition elements into <pivot and >pivot.
3. Recursively quick sort partitions.

Code:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[-1]
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1]if x > pivot]
    return quick_sort(left) + [pivot] + quick_sort(right)
arr = [29, 10, 14, 37, 13]
print("Original Array:", arr)
sorted_arr = quick_sort(arr)
print("Sorted Array:", sorted_arr)
```

Heap Sort Algorithm:

1. Build a max-heap.
2. Swap the top with the last item, reduce the heap and heapify again.

Code:

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
def HeapSort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
def printArray(arr):
    for i in arr:
        print(i, end=" ")
    print()
arr = [9, 4, 3, 8, 10, 2, 5]
HeapSort(arr)
print("Sorted array is")
printArray(arr)
```

Output:

Algorithm	Output
Bubble Sort	5, 7, 8, 9, 12, 19
Selection Sort	5, 7, 8, 9, 12, 19
Insertion Sort	5, 6, 11, 12, 13
Merge Sort	10, 19, 27, 38, 43
Quick Sort	10, 13, 14, 29, 37
Heap Sort	2, 3, 4, 5, 8, 9, 10

Experiment 2:

Objective: To implement the binary search algorithm to efficiently find an element in a sorted array.

Theory

Binary search is an efficient searching algorithm for sorted arrays. It works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the search continues on the left half; otherwise, on the right half. The process repeats until the element is found or the interval is empty.

Time Complexity:

- Best: $O(1)$ (if found at first mid)
- Average/Worst: $O(\log\ n)$

Algorithm:

1. Initialize left = 0 and right = last index of array.
2. While left <= right:
    - Compute mid = (left + right) // 2
    - If arr[mid] == target, return mid
    - If arr[mid] < target, set left = mid + 1
    - Else, set right = mid - 1
3. If not found, return -1

Code:

```
def binary_search(arr, target):
    left=0
    right = len(arr)-1
    while left <= right:
        mid = (left+right)//2
        if arr[mid]==target:
            return mid
        elif arr[mid] < target:
            left=mid+1
        else:
            right = mid -1
    return -1
arr = [10,20,30,40,50,60,70]
target = 60
result = binary_search(arr, target)
if result != -1:
    print(f"Element {target} found at index {result}")
else:
    print(f"Element {target} not found in the array")
```

Output:

Input:
Array: [10, 20, 30, 40, 50, 60, 70]
Target: 60

Output:
Element 60 found at index 5

Experiment 3:

Objective: To develop an algorithm for hashing and perform hash-based searching to efficiently retrieve data from a hash table.

Theory:

A **hash table** is a data structure that associates keys with values, using a **hash function** to transform each key to an index in an underlying array (the 'table'). This enables nearly constant-time (O (1)) lookup, insertion, and deletion operations on average.

Hash Function: Maps a key (e.g., a number or string) to an array index.

Collision Handling: If multiple keys map to the same index, a strategy such as chaining (storing key-value pairs in a list at each slot) or open addressing is used.\

Algorithm

1. **Initialization:** Create an array of 'buckets' (lists) of fixed size N.
2. **Hash Function:** For integer keys, index = key % N.
3. **Insertion:** Put (key, value) into the bucket at the computed index.
    - If the key already exists, update its value.
4. **Search:** Compute index using the hash function, search the list at that index for the key.
    - If found, return value.
    - If not found, return 'not found.'
5. **Deletion:** Locate the key as above and remove it from the chain.

Output:

Hash Table: [[], [(123, 'apple')], [], [(432, 'mango'), (213, 'orange')], [], [], [(654, 'guava')], [], [], []]

Search for 432: mango

After deleting 123: [[], [], [], [(432, 'mango'), (213, 'orange')], [], [], [(654, 'guava')], [], [], []]

Code:

```
hash_table = [[] for _ in range(10)]
def hash_function(key):
    return key % len(hash_table)
def insert(key, value):
    index = hash_function(key)
    for i, (k, v) in enumerate(hash_table[index]):
        if k == key:
            hash_table[index][i] = (key, value)
            return
    hash_table[index].append((key, value))
def search(key):
    index = hash_function(key)
    for k, v in hash_table[index]:
        if k == key:
            return v
    return None
def delete(key):
    index = hash_function(key)
    for i, (k, _) in enumerate(hash_table[index]):
        if k == key:
            del hash_table[index][i]
            return True
    return False
insert(123, "apple")
insert(432, "mango")
insert(213, "banana")
insert(654, "guava")
insert(213, "orange")
print("Hash Table:", hash_table)
print("Search for 432:", search(432))
delete(123)
print("After deleting 123:", hash_table)
```

Experiment 4:

Objective: Implement graph representations and perform Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms to explore all nodes in a graph.

Theory

- A **graph** consists of vertices and edges. Common representations are adjacency matrices and adjacency lists.
- **BFS** explores the graph level by level, visiting all immediate neighbours of a node before moving deeper. It uses a **queue** data structure.
- **DFS** explores as deep as possible along one branch before backtracking. It uses a **stack** (explicitly or via recursion).
- Both algorithms ensure all connected nodes are visited, but their traversal orders differ.

Algorithm: Breadth-First Search (BFS)

1. Mark all nodes as unvisited.
2. Enqueue the start node and mark it as visited.
3. While the queue is not empty, repeat:
    - Dequeue a node and visit it.
    - For every unvisited adjacent node, enqueue it and mark it as visited.

Code:

```
from collections import deque
def bfs(graph, start):
    visited = set()
    q=deque([start])
    while q:
        vertex = q.popleft()
        if vertex not in visited:
            print(vertex,end=" ")
            visited.add(vertex)
            for n in graph[vertex]:
                if n not in visited:
                    q.append(n)
graph = {
    1:[2,3,4],
    2:[1,5,6],
    3:[1,7,8],
    4:[1,9,10],
    5:[2],
    6:[2],
    7:[3],
    8:[3],
    9:[4],
    10:[4]
}
bfs(graph,1)
```

Algorithm: Depth-First Search (DFS)

1. Mark all nodes as unvisited.
2. Starting from the given node:
    - Mark it as visited.
    - Visit each unvisited neighbour recursively.

```
def dfs(graph,start):
    visited = set()
    stack = [start]
    while stack:
        vertex=stack.pop()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            for n in reversed(graph[vertex]):
                if n not in visited:
                    stack.append(n)
graph = {
    1:[2,3,4],
    2:[1,5,6],
    3:[1,7,8],
    4:[1,9,10],
    5:[2],
    6:[2],
    7:[3],
    8:[3],
    9:[4],
    10:[4]
}
dfs(graph,1)
```

Output:

Algorithm	Traversal Order
BFS	[1, 2, 3, 4, 5, 6]
DFS	[1, 2, 4, 5, 6, 3]

Experiment 5:

Objective: Design and implement algorithms to solve shortest path problems using Dijkstra's algorithm (for graphs with non-negative weights) and Bellman-Ford algorithm (for graphs that may have negative weights/cycles).

Theory:

- **Shortest path algorithms** find the minimum cost/path from a source node to all other nodes in a weighted graph.
- **Dijkstra's Algorithm:** Efficient for graphs **with non-negative weights**; uses a greedy approach and priority queue to always extend from the currently closest node.
- **Bellman-Ford Algorithm:** Handles graphs **with negative weights** and can detect negative-weight cycles; uses edge relaxation and runs for |V|-1 passes (V = number of vertices).

Dijkstra's Algorithm

1. Set all node distances to infinity except the source, which is 0.
2. Use a priority queue to repeatedly select the node with the smallest tentative distance.
3. For each neighbour of the selected node, if a shorter path is found, update its distance and add it to the queue.
4. Repeat until the queue is empty.

Code:

```python
import heapq
def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    pq = [(0, start)]
    while pq:
        current_distance, current_node = heapq.heappop(pq)
        if current_distance > distances[current_node]:
            continue
        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
    return distances
graph = {
    0: [(1, 4), (2, 8)],
    1: [(4, 6)],
    2: [(3, 2)],
    3: [(4, 10)],
    4: []
}
source = 0
shortest_distances = dijkstra(graph, source)
print(f"Shortest distances from source vertex {source}:")
for node, distance in shortest_distances.items():
    print(f"Vertex {node} → Distance = {distance}")
```

Bellman-Ford Algorithm

1. Set all node distances to infinity except the source, which is 0.
2. For (V-1) times, go through all edges and "relax" each: if distance[u] + weight < distance[v], update distance[v].
3. For one more pass, check if any distance can be updated again. If so, a negative weight cycle exists.

   Code:

```
V = 4
edges = [
    (1, 2, 4),
    (1, 4, 5),
    (4, 3, 3),
    (3, 2, -10)
]
def bellman_ford(V, edges, src):
    dist = {i: float('inf') for i in range(1, V+1)}
    dist[src] = 0
    for _ in range(V - 1):
        for u, v, w in edges:
            if dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
    for u, v, w in edges:
        if dist[u] + w < dist[v]:
            print("Graph contains a negative weight cycle")
            return
    print("Vertex\tDistance from Source")
    for vertex in range(1, V + 1):
        print(f"{vertex}\t{dist[vertex]}")
bellman_ford(V, edges, src=1)
```

Output:

Dijkstra's Algorithm

Vertex	Distance
0	0
1	4
2	8
3	10
4	10

Bellman-Ford Algorithm

Vertex	Distance
1	0
2	-3
3	8
4	5

Experiment 6:

Objective: To efficiently connect all nodes in a weighted undirected graph by constructing a Minimum Spanning Tree (MST) using both Prim's and Kruskal's algorithms.

Theory:

- **MST**: A subset of edges connecting all vertices with minimum total edge weight, no cycles.
- **Prim's Algorithm**: Grows MST from an initial node, always choosing the lowest-weight edge that extends the tree to a new vertex.
- **Kruskal's Algorithm**: Considers all graph edges in order of ascending weight, adding each that connects two previously unconnected sets, avoiding cycles.

Prim's Algorithm (Min-Heap Variant)

1. Start from any node, add all its edges to a min-heap.
2. While unvisited nodes remain:
    - Pop the minimum-weight edge from the heap.
    - If it leads to an unvisited node, add it to the MST and push new edges from this node.
3. Stop when all vertices are included.

Code:

```python
import heapq
def prims_min_heap(graph, start=0):
    visited = set()
    min_heap = [(0, start)]
    total_weight = 0
    mst_edges = []
    while min_heap:
        weight, node = heapq.heappop(min_heap)
        if node in visited:
            continue
        visited.add(node)
        total_weight += weight
        for neighbor, wt in graph[node]:
            if neighbor not in visited:
                heapq.heappush(min_heap, (wt, neighbor))
                mst_edges.append((node, neighbor, wt))
    print("MST Edges:", mst_edges)
    print("Total Minimum Cost:", total_weight)
graph = {
    0: [(1, 2), (3, 6)],
    1: [(0, 2), (2, 3), (3, 8), (4, 5)],
    2: [(1, 3), (4, 7)],
    3: [(0, 6), (1, 8)],
    4: [(1, 5), (2, 7)]
}
prims_min_heap(graph)
```

Kruskal's Algorithm (Simple Variant)

1. Sort all edges by weight (or heapify).
2. For each edge, if it connects two unvisited nodes (or two different sets):
    - Add it to MST; mark endpoints visited (or union their sets).
3. Stop when MST has $V - 1$ edges.

Code:

```python
import heapq
def kruskals_min_heap(V, edges):
    heapq.heapify(edges)
    mst = []
    total_weight = 0
    visited = set()
    while edges and len(mst) < V - 1:
        weight, u, v = heapq.heappop(edges)
        if u not in visited or v not in visited:
            mst.append((u, v, weight))
            total_weight += weight
            visited.add(u)
            visited.add(v)
    print("MST Edges:", mst)
    print("Total Minimum Cost:", total_weight)
edges = [
    (2, 0, 1),
    (3, 1, 2),
    (6, 0, 3),
    (8, 1, 3),
    (5, 1, 4),
    (7, 2, 4)
]
V = 5
kruskals_min_heap(V, edges)
```

Output:

Algorithm	MST Edges	Total Minimum Cost
Prim's	(0,1,2), (1,2,3), (1,4,5), (0,3,6)	16
Kruskal's	(0,1,2), (1,2,3), (1,4,5), (0,3,6)	16

Experiment 7:

Objective: Implement algorithms to generate a valid topological ordering of vertices in a directed acyclic graph using both Kahn's (BFS-based) algorithm and the DFS-based algorithm.

Theory:

- **Topological Sort** produces a linear ordering of the vertices such that for every directed edge from node u to node v, u comes before v in the ordering.
- Applicable only to **DAGs** (Directed Acyclic Graphs).
- **Kahn's Algorithm (BFS approach):** Uses a queue to repeatedly remove nodes with zero incoming edges (indegree), updating neighbours' indegrees accordingly.
- **DFS-based approach:** Performs DFS from each node, post-ordering nodes when recursion finishes, then reverses the result.

Kahn's Algorithm (BFS-based)

1. Calculate the indegree of all nodes.
2. Add all nodes with indegree zero to the queue.
3. While the queue is not empty:
    - Remove a node from the queue, add it to the topological order.
    - Decrease the indegree of its neighbours; add any that reach indegree zero.
4. Output topological order.

Code:

```
from collections import deque
graph = {
    0: [1, 2],
    1: [4],
    2: [5],
    5: [4],
    4: []
}
indegree = {node: 0 for node in graph}
for node in graph:
    for neigh in graph[node]:
        indegree[neigh] += 1
queue = deque([node for node in indegree if indegree[node] == 0])
topo_order = []
while queue:
    node = queue.popleft()
    topo_order.append(node)
    for neigh in graph[node]:
        indegree[neigh] -= 1
        if indegree[neigh] == 0:
            queue.append(neigh)
print("Topological Order (BFS / Kahn):", topo_order)
```

DFS-based Algorithm

1. For all nodes, do DFS if not visited:
    - For each neighbour, DFS recursively.
    - After visiting all descendants, add node to the result (stack).
2. Reverse the stack to get topological order.

    Code:

```
graph = {
    0: [1, 2],
    1: [4],
    2: [5],
    5: [4],
    4: []
}
visited = []
stack = []
def dfs(node):
    if node in visited:
        return
    visited.append(node)
    for neighbor in graph[node]:
        dfs(neighbor)
    stack.append(node)
for node in graph:
    dfs(node)
stack.reverse()
print("Topological Order (DFS):", stack)
```

Output:

Algorithm	**Topological Order**
Kahn's Algorithm	[0, 1, 2, 5, 4]
DFS-based Algorithm	[0, 2, 5, 1, 4]

Experiment 8:

Objective: Apply dynamic programming techniques including memorization and tabulation to solve optimization problems like the Longest Common Subsequence, Matrix Chain Multiplication, and Knapsack problem.

Longest Common Subsequence (LCS) Algorithm

Objective: Find the longest subsequence present in both sequences.

- Input: Two strings, e.g., X = "AGGTAB", Y = "GXTXAYB"
- Output: Length of LCS (and optionally the sequence)

Memoization Approach (Recursive + Cache)

1. Initialize memo dictionary: memo = {}

2. Call LCS(m, n) where m = len(X), n = len(Y)

3. In LCS(i, j):

   a. BASE CASE: if i == 0 or j == 0: return 0

   b. CACHE CHECK: if (i,j) in memo: return memo[(i,j)]

   c. RECURSE:

   - If X[i-1] == Y[j-1]: memo[(i,j)] = 1 + LCS(i-1,j-1)

   - Else: memo[(i,j)] = max(LCS(i-1,j), LCS(i,j-1))

   d. Return memo[(i,j)]

Code:

```python
def lcs_memo(X, Y):
    m, n = len(X), len(Y)
    memo = {}
    def solve(i, j):
        if i == m or j == n:
            return 0
        if (i, j) in memo:
            return memo[(i, j)]
        if X[i] == Y[j]:
            memo[(i, j)] = 1 + solve(i + 1, j + 1)
        else:
            memo[(i, j)] = max(
                solve(i + 1, j),
                solve(i, j + 1)
            )
        return memo[(i, j)]
    return solve(0, 0)
X = "ABCBDAB"
Y = "BDCABA"
print("LCS length =", lcs_memo(X, Y))
```

Tabulation Approach (DP Table)

1. Create DP table: dp[m+1][n+1] initialized to 0

2. Fill table iteratively:

FOR i = 1 to m:

FOR j = 1 to n:

IF X[i-1] == Y[j-1]:

dp[i][j] = dp[i-1][j-1] + 1

ELSE:

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

3. Return dp[m][n]

Code:

```
def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
                dp[i][j] = 1 + dp[i - 1][j - 1]
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])
    return dp[m][n]
X = "ABCBDAB"
Y = "BDCABA"
print("LCS length =", lcs(X, Y))
```

Matrix Chain Multiplication (MCM) Algorithm

Objective: Parenthesize products to minimize multiplication cost.

- Input: List of dimensions, e.g.,
- Output: Minimum cost (scalar multiplications)

Memoization Approach

1. Initialize memo = {}

2. Call MCM(1, n-1) where n = len(arr)

3. In MCM(i, j):

a. BASE CASE: if i == j: return 0

b. CACHE CHECK: if (i,j) in memo: return memo[(i,j)]

c. Initialize: memo[(i,j)] = ∞

d. FOR k = i to j-1:

cost = MCM(i,k) + MCM(k+1,j) + arr[i-1]×arr[k]×arr[j]

memo[(i,j)] = min(memo[(i,j)], cost)

e. Return memo[(i,j)]

Code:

```
import sys
def matrix_chain_memo(p, n):
    dp = [[-1 for _ in range(n)] for _ in range(n)]
    return solve(p, 1, n - 1, dp)
def solve(p, i, j, dp):
    if i == j:
        return 0
    if dp[i][j] != -1:
        return dp[i][j]
    dp[i][j] = sys.maxsize
    for k in range(i, j):
        cost = (
            solve(p, i, k, dp) +
            solve(p, k + 1, j, dp) +
            p[i - 1] * p[k] * p[j]
        )
        dp[i][j] = min(dp[i][j], cost)
    return dp[i][j]
p = [10, 20, 30, 40, 30]
n = len(p)
print(matrix_chain_memo(p, n))
```

Tabulation Approach

1. Create DP table: dp[n][n] initialized to 0

2. FOR chain_length = 2 to n-1:

FOR i = 1 to n-chain_length:

j = i + chain_length - 1

dp[i][j] = ∞

FOR k = i to j-1:

cost = dp[i][k] + dp[k+1][j] +

arr[i-1]×arr[k]×arr[j]

dp[i][j] = min(dp[i][j], cost)

4. Return dp[1][n-1]

Code:

```
import sys
def matrix_chain_tab(p, n):
    dp = [[0 for _ in range(n)] for _ in range(n)]
    for L in range(2, n):
        for i in range(1, n - L + 1):
            j = i + L - 1
            dp[i][j] = sys.maxsize
            for k in range(i, j):
                cost = (
                    dp[i][k] +
                    dp[k + 1][j] +
                    p[i - 1] * p[k] * p[j]
                )
                dp[i][j] = min(dp[i][j], cost)
    return dp[1][n - 1]
p = [10, 20, 30, 40, 30]
n = len(p)
print(matrix_chain_tab(p, n))
```

Knapsack Problem (0/1) Algorithm

Objective: Select items to maximize value without exceeding capacity.

- Input: Item weights/values, knapsack limit (e.g., weights = , values = , W = 5)
- Output: Maximum attainable value.

Memoization Approach

1. Initialize memo = {}

2. Call Knap(n, W) where n = len(weights)

3. In Knap(i, w):

   a. BASE CASE: if i == 0 or w == 0: return 0

   b. CACHE CHECK: if (i,w) in memo: return memo[(i,w)]

   c. IF wt[i-1] > w:

      memo[(i,w)] = Knap(i-1, w)

   d. ELSE:

      memo[(i,w)] = max(

         val[i-1] + Knap(i-1, w-wt[i-1]),

         Knap(i-1, w))

      e. Return memo[(i,w)]

Code:

```
def knapsack_memo(W, wt, val, n, memo):
    if n == 0 or W == 0:
        return 0
    if (n, W) in memo:
        return memo[(n, W)]
    if wt[n-1] > W:
        memo[(n, W)] = knapsack_memo(W, wt, val, n-1, memo)
    else:
        memo[(n, W)] = max(val[n-1] + knapsack_memo(W - wt[n-1], wt, val, n-1, memo),
                           knapsack_memo(W, wt, val, n-1, memo))
    return memo[(n, W)]
val = [60, 100, 120]
wt = [1, 2, 3]
W = 5
memo = {}
print(knapsack_memo(W, wt, val, len(val), memo))
```

Tabulation Approach

1. Create DP table: dp[n+1][W+1] initialized to 0

2. FOR i = 1 to n:

   FOR w = 0 to W:

      IF wt[i-1] <= w:

         dp[i][w] = max(

            val[i-1] + dp[i-1][w-wt[i-1]],

            dp[i-1][w])

      ELSE:

         dp[i][w] = dp[i-1][w]

4. Return dp[n][W]

Code:

```
def knapsack_tabulation(W, wt, val, n):
    dp = [[0] * (W + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(W + 1):
            if wt[i - 1] <= w:
                dp[i][w] = max(
                    val[i - 1] + dp[i - 1][w - wt[i - 1]],
                    dp[i - 1][w]
                )
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][W]
W = 50
wt = [10, 20, 30]
val = [60, 100, 120]
print(knapsack_tabulation(W, wt, val, len(val)))
```

Output:

Problem	Technique	Sample Input	Output
Longest Common Subsequence	Memorization	X="AGGTAB", Y="GXTXAYB"	Length of LCS is 4
	Tabulation	X="AGGTAB", Y="GXTXAYB"	Length of LCS is 4
Matrix Chain Multiplication	Memorization	Arr = [10, 20, 30, 40, 30]	Minimum Cost is 30000
	Tabulation	Arr = [10, 20, 30, 40, 30]	Minimum Cost is 30000
Knapsack (0/1)	Memorization	W = 5, Wt = [1, 2, 3] Val = [60, 100, 120]	The maximum value in the knapsack is 220
	Tabulation	W = 50 Wt = [10, 20, 30] Val = [60, 100, 120]	The maximum value in the knapsack is 220

Experiment 9:

Objective: To understand the limits of algorithmic efficiency by studying advanced topics in algorithm design, specifically NP-completeness and approximation algorithms, and to implement a simple approximation algorithm for an NP-hard problem to observe how near-optimal solutions can be obtained in polynomial time.

Exact Algorithm: TSP using Brute Force

- **Goal:** Find the *exact* shortest tour that visits every city once and returns to the start.
- **Step 1 – Fix a start city:**
  Choose one city as the starting point (to avoid counting rotations as different tours).
- **Step 2 – Generate all permutations:**
  List all possible permutations of the remaining $n - 1$ cities.
  Each permutation represents one possible tour.
- **Step 3 – Compute tour distance:**
  For each permutation:
  - Start from the fixed city.
  - Visit the cities in the order of the permutation.
  - Return to the starting city.
  - Sum all edge weights to get the total tour cost.
- **Step 4 – Select best route:**
  Compare the total costs of all tours.
  Choose the permutation with the **minimum** total distance.

Code:

```
from itertools import permutations
def tsp_brute_force(distance_matrix):
    n = len(distance_matrix)
    min_path = None
    min_cost = float('inf')
    cities = range(n)
    for perm in permutations(cities):
        cost = sum(distance_matrix[perm[i]][perm[(i+1)%n]] for i in range(n))
        if cost < min_cost:
            min_cost = cost
            min_path = perm
    return min_path, min_cost
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
path, cost = tsp_brute_force(distance_matrix)
print("Optimal path:", path)
print("Optimal cost:", cost)
```

Approximation Algorithm: TSP using MST Heuristic (Metric TSP)

- **Assumption:** Distances satisfy **triangle inequality** (metric TSP).
- **Goal:** Find a tour **close to optimal** in polynomial time.
- **Step 1 – Build MST:**
  - Consider the complete graph of cities with edge weights = distances.
  - Construct a **Minimum Spanning Tree (MST)** using, e.g., Prim's or Kruskal's algorithm.
  - MST connects all cities with minimum total edge cost, but it is *not* a tour.
- **Step 2 – Pre-order traversal:**
  - Choose any node as root of the MST.
  - Perform a **pre-order depth-first traversal** of the MST.
  - Record the order in which cities are first visited.
- **Step 3 – Form the tour:**
  - Visit the cities in the recorded pre-order sequence.
  - After the last city, return to the starting city.
  - This gives a Hamiltonian tour.
- **Step 4 – Output:**
  - Return this tour as the **approximate** solution to TSP.

Code:

```
import networkx as nx
def tsp_mst_approx(distance_matrix):
    n = len(distance_matrix)
    G = nx.Graph()
    for i in range(n):
        for j in range(i+1, n):
            G.add_edge(i, j, weight=distance_matrix[i][j])
    mst = nx.minimum_spanning_tree(G)
    pre_order = list(nx.dfs_preorder_nodes(mst, source=0))
    approx_cost = sum(distance_matrix[pre_order[i]][pre_order[(i+1)%n]] for i in range(n))
    return pre_order, approx_cost
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
path, cost = tsp_mst_approx(distance_matrix)
print("Approximate path:", path)
print("Approximate cost:", cost)
```

Output:

Optimal path: (0, 1, 3, 2)

Optimal cost: 80

Approximate path: [0, 1, 2, 3]

Approximate cost: 95