EXPERIMENT - 01

Aim : Given N items with their corresponding weights and values, and a package of capacity C, choose either the entire item or fractional part of the item among these N unique items to fill the package such that the package has maximum value.

Description :

This problem is known as the "Knapsack Problem," which is a classic optimization problem in computer science and mathematics. The goal is to maximize the total value of the items placed in the knapsack while ensuring that the total weight does not exceed the knapsack's capacity.

Approach

The Knapsack Problem can be solved using a greedy approach by sorting the items based on their value-to-weight ratio in descending order. The items with higher value-to-weight ratios are prioritized for inclusion in the package.

In the context of the Knapsack Problem, the greedy approach involves sorting the items based on their value-to-weight ratio in descending order. The rationale behind this approach is to prioritize the inclusion of items that provide the highest value per unit weight. By making greedy choices at each step, the algorithm attempts to maximize the total value obtained within the given weight capacity.

Algorithm

- Consider all the items with their weights and profits mentioned respectively.
- Calculate P_i/W_i of all the items and sort the items in descending order based on their P_i/W_i values.
- Without exceeding the limit, add the items into the knapsack.
- If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.
- Hence, giving it the name fractional knapsack problem.

// Input: A list of items, each with a value and a weight. The capacity of the knapsack

// Output: The maximum profit made by filling the knapsack

Step 1: weight Taken = 0;

Step 2: profit = 0

Step 3: sort item I in descending order of v/w

Step 4: for each item i in the sorted list

Step 5: if (weight Taken + wi <= capacity)

Step 6: weight Taken = weight Taken + wi

Step 7: profit profit + vi

Step 8: else

Step 9: remaining = capacity - weight Taken

Step 10: profit = profit + remaining (v/w)

Step 11: break

Code :

```
def value_per_weight (item):
    return item[1]/item [0]

def fractional_Knapsack (items, capacity):
    items.sort(key=value_per_weight, reverse=True)
    max_value = 0
    for weight, value in items:
        if capacity == 0:
            break
        if weight <= capacity:
            max_value += value
            capacity -= weight
        else:
            max_value += (value/weight)*capacity
            break
```

```
    return max_value

num_items = int(input("Enter the number of items: "))
items = []

for i in range(num_items):
    weight = int(input(f"Enter the weight for item {i+1}: "))
    profit = int(input(f"Enter the value for the item {i+1}: "))
    items.append((weight, profit))

capacity = int(input("Enter the capacity of the Knapsack: "))
print("Maximum profit: ", fractional_Knapsack(items, capacity))
```

Output :

```
Enter the weight for item 2: 5
Enter the value for the item 2: 20
Enter the weight for item 3: 8
Enter the value for the item 3: 15
Enter the weight for item 4: 1
Enter the value for the item 4: 5
Enter the capacity of the Knapsack: 10
Maximum profit:  38.75
```

Analysis:

Time Complexity :
The maximum Value function first sorts a vector of pairs using the sort function and the compare function as the comparator. The time complexity of the sort operation is O(n log n), where n is the size of the vector. **Hence, the overall time complexity is : O(nlogn)**
Space Complexity :
The space complexity of the sorting algorithm used is **O(log n)** due to the auxiliary space used.

EXPERIMENT - 02

Aim:

Given a bunch of projects, where every project has a deadline and associated profit if the project is finished before the deadline. It is also given that every project takes one month duration, so the minimum possible deadline for any project is 1 month. In what way the total profits can be maximized if only one project can be scheduled at a time.

Description:

This problem can be solved using a greedy approach by scheduling the projects in descending order of their profit-to-deadline ratio. This approach ensures that the projects with higher profits and shorter deadlines are prioritized, thereby maximizing the total profit while meeting the deadlines.
The algorithm is similar to a knapsack problem and requires the use of additional space , sorting etc..

Algorithm:

```
MaximizeProfit(projects):
   sort projects in descending order of their deadlines
   total_profit = 0
   deadlines = set of all available deadlines (from len(projects) to 1)

   for each project (deadline, profit) in the sorted projects:
      find the latest available deadline <= current project deadline:
         latest_deadline = current project deadline
         while latest_deadline is not in deadlines and latest_deadline > 0:
            latest_deadline = latest_deadline - 1

      if a suitable latest_deadline is found (latest_deadline > 0):
         add profit to total_profit
         remove latest_deadline from deadlines

   return total_profit
```

The Algorithm works as follows:

1. Sort the projects in descending order of their deadlines. This is done to prioritize projects with tighter deadlines, as they have fewer choices for scheduling.

2. Initialize total_profit to 0, which will store the maximum total profit.

3. Create a set deadlines containing all the available deadlines, from len(projects) down to 1.

4. Iterate over each project in the sorted list of projects.

5. For each project (deadline, profit):

• Find the latest available deadline that is less than or equal to the current project's deadline. This is done by starting with the project's deadline and decrementing it until a deadline in the deadlines set is found or the deadline becomes 0.

• If a suitable deadline is found (i.e., latest_deadline > 0), add the project's profit to total profit and remove the latest deadline from the deadlines set to ensure that no other project is assigned the same deadline.

6. After iterating over all projects, return the total profit as the maximum total profit that can be achieved by selecting a subset of non-overlapping projects.

Code:

```
def max_profit(projects):
    def sort_key(project):
        return project[1]
    projects.sort(key = sort_key, reverse=True)
    total_profit = 0
    deadlines = set(range(len(projects), 0, -1))
    for deadline,profit in projects:
        while deadline not in deadlines and deadline:
            deadline -= 1
        if deadline>0:
            total_profit+=profit
            deadlines.remove(deadline)
    return total_profit

num_projects = int(input("Enter the number of projects: "))
projects=[]
```

```
for i in range(num_projects):
    deadline = int(input(f"Enter the deadline for project {i+1}: "))
    profit = int(input(f"Enter the profit for project {i+1}: "))
    projects.append((deadline,profit))
print("Maximum total profit: ",max_profit(projects))
```

Output:

```
Enter the profit for project 1: 20
Enter the deadline for project 2: 28
Enter the profit for project 2: 5
Enter the deadline for project 3: 14
Enter the profit for project 3: 16
Enter the deadline for project 4: 21
Enter the profit for project 4: 16
Maximum total profit:  57
```

Output Analysis:

The code outputs the maximum number of non-overlapping jobs that can be scheduled and the maximum profit obtainable from scheduling those jobs. The output is determined by the `jobScheduling` function, which sorts the jobs by profit and greedily schedules them without overlapping time slots.

Time Complexity Analysis:

1. Sorting jobs: O(n log n)
2. Finding max deadline: O(n)
3. Creating slots array: O(maxDeadline), which is O(n) in the worst case.
4. Scheduling jobs: O(n * maxDeadline), which is O(n^2) in the worst case.

Overall time complexity: O(n log n + n^2) = O(n^2)

Space Complexity Analysis:

1. Input jobs vector: O(n)
2. Sorted jobs vector: O(n)
3. Slots array: O(maxDeadline), which is O(n) in the worst case.

Overall space complexity: O(n)

In summary, the time complexity of the `jobScheduling` function is O(n^2), and its space complexity is O(n), where n is the number of jobs.

EXPERIMENT - 03

Aim: To find the maximum and minimum value from the given array of numbers using the divide and conquer approach.

Description:
The task is to find the maximum and minimum element in an array using the divide and conquer approach. The findMaxMin function is implemented to do the same . It achieves this by employing a divide-and-conquer strategy. The algorithm recursively breaks down the problem by splitting the array in half. It then delves deeper, finding the maximum and minimum elements in each half of the array through further recursion. Finally, it compares the maximum and minimum values obtained from both halves to determine the overall maximum and minimum elements for the entire array.

The divide and conquer approach is generally used for sorting and searching and it reduces the time complexity from O(n^2) and O(n) to O(nlogn) and O(logn) by eliminating one half of the array in each pass.

Algorithm :
Algorithm FindMaxMin(arr, low, high):

Step 1: If low is equal to high, it means the subarray contains only one element. Return this element as both the maximum and minimum value.

```
if low == high:
    return arr[low], arr[low]
```

Step 2: If high is one more than low, it means the subarray contains two elements. Return the maximum and minimum of these two elements.

```
if high == low + 1:
    return max(arr[low], arr[high]), min(arr[low], arr[high])
```

Step 3: Compute the middle index of the subarray.

```
mid = (low + high) // 2
```

Step 4: Recursively find the maximum and minimum elements in the left subarray (from low to mid) by calling FindMaxMin(arr, low, mid).

```
max_left, min_left = FindMaxMin(arr, low, mid)
```

Step 5: Recursively find the maximum and minimum elements in the right subarray (from mid + 1 to high) by calling FindMaxMin(arr, mid + 1, high).

max_right, min_right = FindMaxMin(arr, mid + 1, high)

Step 6: Combine the maximum and minimum values obtained from the left and right subarrays.

maximum = max(max_left, max_right)
minimum = min(min_left, min_right)

Step 7: Return the combined maximum and minimum values as a tuple.

return maximum, minimum

Step 8: In the main function:

a. Get the size of the array (n) from the user.
b. Get the array elements from the user and store them in a list (arr).
c. Call FindMaxMin(arr, 0, n - 1) to find the maximum and minimum elements of the entire array.
d. Print the maximum and minimum elements.

Code:

```
def find_max_min(arr, low, high):
    if low == high:
        return arr[low], arr[low]

    if high == low + 1:
        return (max(arr[low], arr[high]), min(arr[low], arr[high]))

    mid = (low + high) // 2
    max_left, min_left = find_max_min(arr, low, mid)
    max_right, min_right = find_max_min(arr, mid + 1, high)

    return (max(max_left, max_right), min(min_left, min_right))

if __name__ == "__main__":
    n = int(input("Enter the Size of the Array: "))
    arr = list(map(int, input("Enter the Array Elements: ").split()))

    max_val, min_val = find_max_min(arr, 0, n - 1)

    print(f"Maximum Element: {max_val}")
```

```
print(f"Minimum Element: {min_val}")
```

Output:

```
Enter the Size of the Array: 7
Enter the Array Elements: 25 0 -9 100 -1 47 1
Maximum Element: 100
Minimum Element: -9
```

Output Analysis:

Time Complexity: O(n log n)

The algorithm uses a divide-and-conquer approach, breaking the problem into subproblems and recombining the results. The recursive calls contribute to the logarithmic factor, while the work done per level (finding middle element and comparisons) results in the linear factor.

Space Complexity: O(log n)

The recursive nature of the algorithm leads to function call stack overhead. The call stack depth grows logarithmically with the input size, resulting in logarithmic space complexity.

EXPERIMENT - 04

Aim: Apply 0/1 knapsack problem to find the maximum profit using dynamic approach and analyse its time complexity.

Description:
Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

- Initialization: Create a 2D array to store maximum values for different capacities, initializing the first row and column to zeros.
- Dynamic Programming: Iterate over items and capacities.
- For each item: If the item's weight is greater than the current capacity, skip it. Otherwise, choose the maximum of including or excluding the item.
- Output: The value in the bottom-right cell of the array represents the maximum value that can be obtained.

Algorithm :

```
Dynamic-0-1-knapsack (v, w, n, W)
  for w = 0 to W do
    c[0, w] = 0
  for i = 1 to n do
    c[i, 0] = 0
    for w = 1 to W do
      if wi ≤ w then
        if vi + c[i-1, w-wi] then
          c[i, w] = vi + c[i-1, w-wi]
        else c[i, w] = c[i-1, w]
      else
        c[i, w] = c[i-1, w]
```

Code :

```
def knapSack(W, wt, val, n):

    if n == 0 or W == 0:
        return 0
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    else:
        return max(
            val[n-1] + knapSack(
                W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))
if __name__ == '__main__':
    weight = [int(x) for x in input("Enter weights separated by space:
").split()]
    profit = [int(x) for x in input("Enter values separated by space:
").split()]
    W = int(input("Enter the capacity of knapsack: "))
    n = len(profit)
    print("Max Profit: ", knapSack(W, weight, profit, n) )
```

Output:

Enter weights separated by space: 10 20 30
Enter values separated by space: 60 100 120
Enter the capacity of knapsack: 50
Max Profit: 220

Analysis:

Time Complexity: O(n×W)

n: number of items
W: capacity of the knapsack
The time complexity arises from the nested loops in the dynamic programming solution, where we iterate through each item and each possible weight capacity of the knapsack.

Space Complexity: O(n×W)

n: number of items
W: capacity of the knapsack
We use a 2D array of size (n+1) x (W+1) to store the solutions to subproblems. Therefore, the space complexity is proportional to the product of the number of items and the capacity of the knapsack.

EXPERIMENT - 05

Aim: Apply Huffman coding to find out the code of each word, percentage of maximum profit and analyze its time complexity.

Description:
Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

1. Frequency Analysis: Determine the frequency of each symbol in the input data.
2. Build Huffman Tree: Create a binary tree where more frequent symbols have shorter binary codes.
3. Generate Huffman Codes: Assign binary codes to each symbol based on their position in the Huffman tree.
4. Encode Data: Replace symbols in the input data with their corresponding Huffman codes.
5. Output Encoding: Include necessary information for decoding, such as the Huffman tree structure.
6. Decoding: Reconstruct the original data using the Huffman tree. Compression Ratio: Compare the size of the original data with the compressed data to measure efficiency.
7. Implementation: Implement Huffman coding in your preferred programming language, using libraries or coding from scratch.

Algorithm :

Step 1: create a priority queue Q consisting of each unique character.
Step 2: sort then in ascending order of their frequencies.
Step 3: for all the unique characters:
Step 4: create a newNode
Step 5: extract minimum value from Q and assign it to leftChild of newNode
Step 6: extract minimum value from Q and assign it to rightChild of newNode
Step 7: calculate the sum of these two minimum values and assign it to the value of

newNode
Step 8: insert this newNode into the tree
Step 9: return rootNode

Algorithm Huffman (c)
{
n= |c|
Q = c
for i<-1 to n-1
do
{
temp <- get node ()
left (temp] Get_min (Q) right [temp] Get Min (Q)
a = left [templ b = right [temp]
F [temp]<- f[a] + [b]
insert (Q, temp)
}
return Get_min (0)
}

Code :

```
string = input("Enter the String: ")

class NodeTree(object):

    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self):
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)
```

```
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d

freq = {}
for c in string:
    if c in freq:
        freq[c] += 1
    else:
        freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

nodes = freq

while len(nodes) > 1:
    (key1, c1) = nodes[-1]
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))

    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])

print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
    print(' %-4r |%12s' % (char, huffmanCode[char]))
```

Output:

```
Enter the String: BCCABBDDAECCBBAEDDCC
 Char | Huffman code
----------------------
 'C' |        11
 'B' |        10
 'D' |        00
 'A' |       011
 'E' |       010
```

Analysis:

Time Complexity:

- Building the Huffman tree: O(nlogn)
- Creating the initial frequency table: O(n)
- Constructing the priority queue: O(n)
- Merging nodes and updating the priority queue: O(nlogn)
- Generating Huffman codes: O(n) (traversing the Huffman tree to assign codes)
- Encoding: O(m) (where m is the size of the input data)
- Decoding: O(m) (where m is the size of the encoded data)

Space Complexity:

- Priority queue: O(n) (to store the nodes during tree construction)
- Huffman tree: O(n) (assuming each node in the tree requires constant space)
- Encoding table: O(n) (to store the mapping of symbols to Huffman codes)
- Encoded data: O(m) (where m is the size of the compressed data)
- Decoded data: O(m) (same as the size of the encoded data, as it needs to store the original symbols)

EXPERIMENT - 06

Aim: Apply Matrix Chain Multiplication to find Maximum Profit and to analyse the Time and Space Complexity.

Description:
Given the dimension of a sequence of matrices in an array arr[], where the dimension of the ith matrix is (arr[i-1] * arr[i]), the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.

- Input: Given a sequence of matrices with dimensions.
- Dynamic Programming: Create a table to store minimum multiplication costs. Iterate over chain lengths and starting indices to fill the table
- Output: Minimum number of multiplications required. Optimal parenthesization.

Algorithm :

MATRIX-CHAIN-ORDER (p)

1. n length[p]-1
2. for i ← 1 to n
3. do m [i, i] ← 0
4. for l ← 2 to n // l is the chain length
5. do for i ← 1 to n-l + 1
6. do j ← i+ l -1
7. m[i,j] ← ∞
8. for k ← i to j-1
9. do q ← m [i, k] + m [k + 1, j] + p_{i-1} p_k p_j
10. If q < m [i,j]
11. then m [i,j] ← q
12. s [i,j] ← k
13. return m and s.

Pseudocode to find the lowest cost of all the possible parenthesizations −

```
MATRIX-CHAIN-MULTIPLICATION(p)
   n = p.length — 1
   let m[1…n, 1…n] and s[1…n — 1, 2…n] be new matrices
   for i = 1 to n
      m[i, i] = 0
   for l = 2 to n // l is the chain length
      for i = 1 to n - l + 1
         j = i + l - 1
         m[i, j] = ∞
         for k = i to j - 1
            q = m[i, k] + m[k + 1, j] + pi-1pkpj
            if q < m[i, j]
               m[i, j] = q
               s[i, j] = k
return m and s
```

Pseudocode to print the optimal output parenthesization −

```
PRINT-OPTIMAL-OUTPUT(s, i, j )
if i == j
print “A”i
else print “(”
PRINT-OPTIMAL-OUTPUT(s, i, s[i, j])
PRINT-OPTIMAL-OUTPUT(s, s[i, j] + 1, j)
print “)”
```

Code :

```
def matrix_chain_order(p):
    n = len(p) - 1
    m = [[0] * n for _ in range(n)]
    s = [[0] * n for _ in range(n)]

    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            m[i][j] = float('inf')
            for k in range(i, j):
                q = m[i][k] + m[k+1][j] + p[i] * p[k+1] * p[j+1]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k

    return m, s

def print_optimal_parens(s, i, j):
    if i == j:
        print("A" + str(i), end="")
    else:
        print("(", end="")
        print_optimal_parens(s, i, s[i][j])
        print_optimal_parens(s, s[i][j]+1, j)
        print(")", end="")

# Get matrix dimensions from user input
n = int(input("Enter the number of matrices: "))
matrix_dimensions = []
for i in range(n + 1):
    dim = int(input(f"Enter dimension {i + 1}: "))
    matrix_dimensions.append(dim)

print("Matrix dimensions:", matrix_dimensions)
m, s = matrix_chain_order(matrix_dimensions)
print("Minimum number of scalar multiplications:", m[0][n - 1])
print("Optimal parenthesization:", end="")
print_optimal_parens(s, 0, n - 1)
```

Output:

Enter the number of matrices: 4
Enter dimension 1: 5
Enter dimension 2: 4
Enter dimension 3: 6
Enter dimension 4: 2
Enter dimension 5: 7
Matrix dimensions: [5, 4, 6, 2, 7]
Minimum number of scalar multiplications: 158
Optimal parenthesization:((A0(A1A2))A3)

Analysis:

- **Time Complexity:** O(n 3), where n is the number of matrices.
- **Building DP Table:** O(n 3) due to three nested loops.
- **Backtracking:** Typically O(n) for tracing back optimal parenthesization.
- **Space Complexity:** O(n 2), where n is the number of matrices.
- **DP Table:** O(n 2).
- **Optimal Parenthesization Table**: O(n 2).
- **Additional Space:** O(n) for storing matrix dimensions.

EXPERIMENT - 07

Aim: Apply travelling sales man problem to find maximum profit and analyse the time and space complexity.

Description:
Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

- **Input:** Given a set of cities and the distances between them. Represent cities as nodes in a weighted graph.
- **Approaches:** For small n, use a brute force approach to check all permutations and find the shortest route.
- **Dynamic Programming:** Use memorization to efficiently compute shortest paths.
- **Branch and Bound:** Explore search space, pruning branches with higher costs.
- **Approximation Algorithms:** Use algorithms like nearest neighbor or minimum
- spanning tree.
- **Output:** Return the sequence of cities forming the optimal tour. Calculate the total distance traveled along the tour.
- **Implementation:** Implement the chosen approach in your preferred programming language, utilizing appropriate data structures.

Algorithm :

```
function tsp(graph, start):
    n = number of cities
    memo = 2^n x n matrix filled with infinity

    function dfs(node, visited):
        if visited == all cities visited:
            return distance from node to start

        if (node, visited) in memo:
            return memo[(node, visited)]

        min_distance = infinity
        for next_node in unvisited cities:
```

distance = graph[node][next_node] + dfs(next_node, visited | (1 << next_node))
min_distance = min(min_distance, distance)

memo[(node, visited)] = min_distance
return min_distance

return dfs(start, 1 << start)

Code :

```
def tsp(graph, start):
    n = len(graph)
    max_distance = float('inf')
    memo = {}
    shortest_path = []

    def dfs(node, visited, path):
        if visited == (1 << n) - 1:
            path.append(start)
            return graph[node][start], path.copy()

        if (node, visited) in memo:
            return memo[(node, visited)]

        min_distance = max_distance
        optimal_path = []

        for next_node in range(n):
            if not visited & (1 << next_node):
                distance, temp_path = dfs(next_node, visited | (1 << next_node), path +
[next_node])
                distance += graph[node][next_node]

                if distance < min_distance:
                    min_distance = distance
                    optimal_path = temp_path

        memo[(node, visited)] = (min_distance, optimal_path)
        return min_distance, optimal_path

    distance, path = dfs(start, 1 << start, [start])

    return distance, path
```

```
n = int(input("Enter the number of cities: "))

print("Enter the distances between cities (use space-separated values):")
graph = []
for _ in range(n):
    row = list(map(int, input().split()))
    graph.append(row)

start_node = int(input("Enter the start city (0-indexed): "))

print("\nDistances between cities:")
for row in graph:
    print(row)

distance, path = tsp(graph, start_node)

print("\nMinimum distance for TSP:", distance)
print("Optimal path:", path)
```

Output:

Enter the number of cities: 4
Enter the distances between cities (use space-separated values):
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
Enter the start city (0-indexed): 1

Distances between cities:
[0, 10, 15, 20]
[10, 0, 35, 25]
[15, 35, 0, 30]
[20, 25, 30, 0]

Minimum distance for TSP: 80
Optimal path: [1, 0, 2, 3, 1]

Analysis:

Time Complexity:

- **Time Complexity**: $O(n\text{^}_2\times 2\text{^}_n)$
  - **n**: Number of cities
  - We need to fill an $n\times 2\text{^}$ nmemoization table, and for each entry, we consider n possible previous cities to compute the optimal distance.

Space Complexity:

- **Space Complexity**:$O(n\times 2\text{^}_n)$
  - **n**: Number of cities
  - We need to store an $n\times 2\text{^}_n$ memoization table.