



AISSMS

COLLEGE OF ENGINEERING

ज्ञानम् सकलजनहिताय



Approved by AICTE, New Delhi, Recognized by Government of Maharashtra
Affiliated to Savitribai Phule Pune University and recognized 2(f) and 12(B) by UGC
(Id.No. PU/PN/Engg./093 (1992))

Accredited by NAAC with "A+" Grade | NBA - 7 UG Programmes

Department of Computer Engineering

“Design and Analysis of Algorithms Lab Manual”

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

In

COMPUTER ENGINEERING

Submitted By

Name of the Student: Piyusha Rajendra Supe

Roll No: 23CO315

Batch: C (BE-B)

Under the Guidance of

Prof. Ashish U. Khandait

**ALL INDIA SHRI SHIVAJI MEMORIAL SOCIETY'S COLLEGE OF
ENGINEERING PUNE-411001**

Academic Year: 2025-26 (Term-II)

Savitribai Phule Pune University

Practical 1

Aim: Write a program non-recursive and recursive program to calculate Fibonacci numbers and analyze their time and space complexity

Theory:

The Fibonacci sequence is one of the most fundamental and widely studied sequences in mathematics and computer science. It is a sequence of numbers where each number is the sum of the two preceding numbers. The sequence starts with 0 and 1, and the subsequent numbers are generated according to the formula:

$$F(n) = F(n - 1) + F(n - 2)$$

with initial values:

- $F(0) = 0$
- $F(1) = 1$

So, the first few terms of the Fibonacci sequence are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... and so on.

The Fibonacci sequence appears in many natural phenomena such as the branching of trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, and even in computer algorithms and data structures.

Approaches to Compute Fibonacci Numbers

There are two common approaches to compute Fibonacci numbers:

1. Recursive Approach

In the recursive approach, the function calls itself to compute smaller Fibonacci numbers until it reaches the base cases $F(0)$ or $F(1)$. The recursive formula directly matches the mathematical definition of the Fibonacci sequence.

Step-by-Step Explanation of Recursive Algorithm:

1. Start the program.
2. Define a recursive function `fib_recursive(n)` which will calculate the n th Fibonacci number.
3. Check the base condition:
 - If n is 0, return 0.
 - If n is 1, return 1.
4. Recursive call:
 - For $n > 1$, the function calls itself twice: once with $n - 1$ and once with $n - 2$.
 - Add the results of these two calls to get $F(n)$.
5. User Input: Read the value of n from the user.

6. Compute Fibonacci numbers: Call the recursive function for each value from 0 to n - 1 to display the Fibonacci sequence.
7. End the program.

Observation: The recursive approach is simple and elegant but inefficient for large values of n. This is because many Fibonacci numbers are recalculated multiple times, which increases computation time exponentially.

2. Non Recursive (Iterative) Approach

The iterative approach uses a simple loop to compute Fibonacci numbers sequentially. Instead of repeatedly calling a function, it stores only the last two computed values and uses them to generate the next number. This makes the iterative approach highly efficient.

Step-by-Step Explanation of Iterative Algorithm:

1. Start the program.
2. Read input from the user for the number of terms n to be displayed.
3. Initialize two variables: a = 0 (first Fibonacci number) and b = 1 (second Fibonacci number).
4. Check for special cases:
 - If n is less than or equal to 0, display an invalid input message.
 - If n is 1, display a only.
5. Loop from 2 to n - 1:
 - Calculate the next Fibonacci number as $c = a + b$.
 - Print c.
 - Update variables: assign b to a and c to b for the next iteration.
6. Continue the loop until all n terms are printed.
7. End the program.

Observation: The iterative approach avoids redundant calculations and is more memory-efficient. Only two variables are needed to keep track of the last two Fibonacci numbers at any time.

Comparison Between Recursive and Iterative Approach

Aspect	Recursive Approach	Iterative Approach
Definition	Uses function calls that refer to themselves to compute Fibonacci numbers.	Uses a loop to compute Fibonacci numbers sequentially.
Memory Usage	High, due to function call stack.	Low, only a few variables are used.

Time Efficiency	Low, exponential time complexity due to repeated calculations.	High, linear time complexity with no repeated calculations.
Ease of Understanding	Conceptually matches the mathematical formula.	Simple to implement and efficient.

Recursive Program:

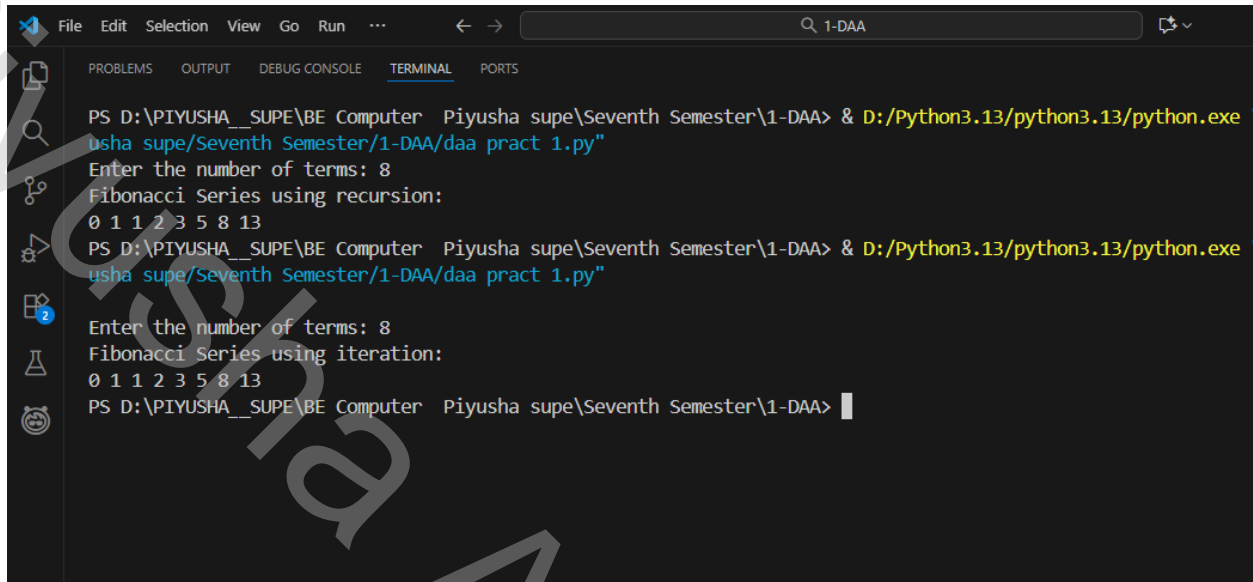
```
def fib_recursive(n):  
    if n <= 1:  
        return n  
    else:  
        return fib_recursive(n - 1) + fib_recursive(n - 2)  
  
# Main Code  
n = int(input("Enter the number of terms: "))  
print("Fibonacci Series using recursion:")  
for i in range(n):  
    print(fib_recursive(i), end=" ")
```

Non Recursive program

```
def fib_non_recursive(n):  
    a, b = 0, 1  
    print("Fibonacci Series using iteration:")  
    if n <= 0:  
        print("Invalid input")  
    elif n == 1:  
        print(a)  
    else:  
        print(a, b, end=" ")  
        for i in range(2, n):  
            c = a + b  
            print(c, end=" ")  
            a, b = b, c
```

Main Code

```
n = int(input("\nEnter the number of terms: "))  
fib_non_recursive(n)
```



```
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> & D:/Python3.13/python3.13/python.exe  
usha supe/Seventh Semester/1-DAA/daa pract 1.py"  
Enter the number of terms: 8  
Fibonacci Series using recursion:  
0 1 1 2 3 5 8 13  
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> & D:/Python3.13/python3.13/python.exe  
usha supe/Seventh Semester/1-DAA/daa pract 1.py"  
Enter the number of terms: 8  
Fibonacci Series using iteration:  
0 1 1 2 3 5 8 13  
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> |
```

Analysis:

Recursive Approach

- In recursion, each call to fib(n) makes two more calls to itself until the base condition is met.
- This leads to an **exponential growth** of function calls as the value of n increases.
- For instance, to calculate fib(5), the function computes fib(4) and fib(3). But to compute fib(4), it again calls fib(3) and fib(2).
- Thus, the same values are recomputed multiple times, making recursion inefficient for large inputs.

Non Recursive (Iterative) Approach

- The iterative version uses a **loop** to compute Fibonacci numbers sequentially without re-computation.
- Since each term depends only on the previous two, only a constant amount of memory is required to store the last two values.
- The iterative approach is therefore much more efficient and suitable for large values of n.

Time Complexity Analysis:

Method	Time Complexity	Explanation
Recursive	$O(2^n)$	Each function call results in two more recursive calls, causing exponential growth.
Non Recursive	$O(n)$	Only a single loop runs from 2 to n. Each iteration performs a constant amount of work.

Space Complexity Analysis:

Method	Space Complexity	Explanation
Recursive	$O(n)$	Due to function call stack during recursive calls.
Non Recursive	$O(1)$	Uses a fixed number of variables regardless of n.

Conclusion:

The Fibonacci sequence can be implemented in both recursive and non recursive ways. The recursive approach, although simple and closely aligned with the mathematical definition, becomes inefficient as it performs redundant calculations and consumes more memory due to deep recursion stacks. The non recursive or iterative approach, on the other hand, is far more efficient and suitable for practical use since it computes Fibonacci numbers in linear time with constant space. Thus, for real world applications where performance and memory efficiency matter, the non recursive approach is preferred.

Practical 2

Aim: Write a program to implement Huffman Encoding using a greedy strategy

Theory:

Huffman Encoding is a widely used lossless data compression algorithm that reduces the size of data without losing any information. It is particularly effective for compressing text and is commonly used in file compression, image compression, and communication systems.

The basic idea of Huffman Encoding is to assign **variable length codes** to input characters, with shorter codes assigned to more frequent characters and longer codes to less frequent characters. This ensures that the total number of bits required to represent the data is minimized.

Greedy Strategy in Huffman Encoding

Huffman Encoding uses a **greedy strategy**, meaning that at each step, it makes a locally optimal choice with the hope of finding the globally optimal solution. The steps are as follows:

1. **Count the frequency** of each character in the input data.
2. **Create a leaf node** for each character and build a priority queue or a min heap, with the frequency as the key.
3. **Repeat until there is only one node left in the heap:**
 - Extract the two nodes with the smallest frequencies from the heap.
 - Create a new internal node with these two nodes as children. The frequency of the new node is the sum of the two smallest frequencies.
 - Insert the new node back into the min heap.
4. The remaining node in the heap is the **root of the Huffman Tree**.
5. **Assign codes** to each character by traversing the tree:
 - Move left from a node → assign 0.
 - Move right from a node → assign 1.
 - Continue until all leaf nodes are assigned a unique Huffman code.

The greedy choice at each step ensures that the least frequent characters are placed deeper in the tree, which results in shorter codes for more frequent characters. This minimizes the total number of bits required for encoding.

Step-by-Step Algorithm

1. **Input the data:** Read the string or list of characters to encode.

2. **Calculate frequencies:** Determine how many times each character appears.
3. **Create a min heap:** Each element is a node with a character and its frequency.
4. **Build the Huffman Tree:**
 - Extract two nodes with the smallest frequencies.
 - Merge them into a new node with frequency equal to the sum of the two.
 - Insert the new node back into the heap.
 - Repeat until only one node remains.
5. **Generate Huffman Codes:** Traverse the tree from the root to each leaf. Assign 0 for left edges and 1 for right edges.
6. **Encode the input string:** Replace each character with its Huffman code.
7. **Display output:** Print Huffman codes for each character and the encoded string.

Node class for Huffman Tree

class Node:

def __init__(self, char, freq):

self.char = char

self.freq = freq

self.left = None

self.right = None

defining comparators for priority queue

def __lt__(self, other):

return self.freq < other.freq

Function to generate Huffman codes

def huffman_codes(root, code, codes_dict):

if root is None:

return

If leaf node, store the code

if root.char is not None:

codes_dict[root.char] = code

huffman_codes(root.left, code + "0", codes_dict)


```
huffman_codes(root.right, code + "1", codes_dict)
```

```
# Main Huffman Encoding Function
```

```
def huffman_encoding(data):
```

```
    # Count frequency of each character
```

```
    freq_dict = { }
```

```
    for char in data:
```

```
        if char in freq_dict:
```

```
            freq_dict[char] += 1
```

```
        else:
```

```
            freq_dict[char] = 1
```

```
# Create priority queue (min heap)
```

```
heap = [Node(char, freq) for char, freq in freq_dict.items()]
```

```
heapq.heapify(heap)
```

```
# Build Huffman Tree
```

```
while len(heap) > 1:
```

```
    left = heapq.heappop(heap)
```

```
    right = heapq.heappop(heap)
```

```
    merged = Node(None, left.freq + right.freq)
```

```
    merged.left = left
```

```
    merged.right = right
```

```
    heapq.heappush(heap, merged)
```

```
# Generate codes
```

```
root = heap[0]
```

```
codes_dict = { }
```

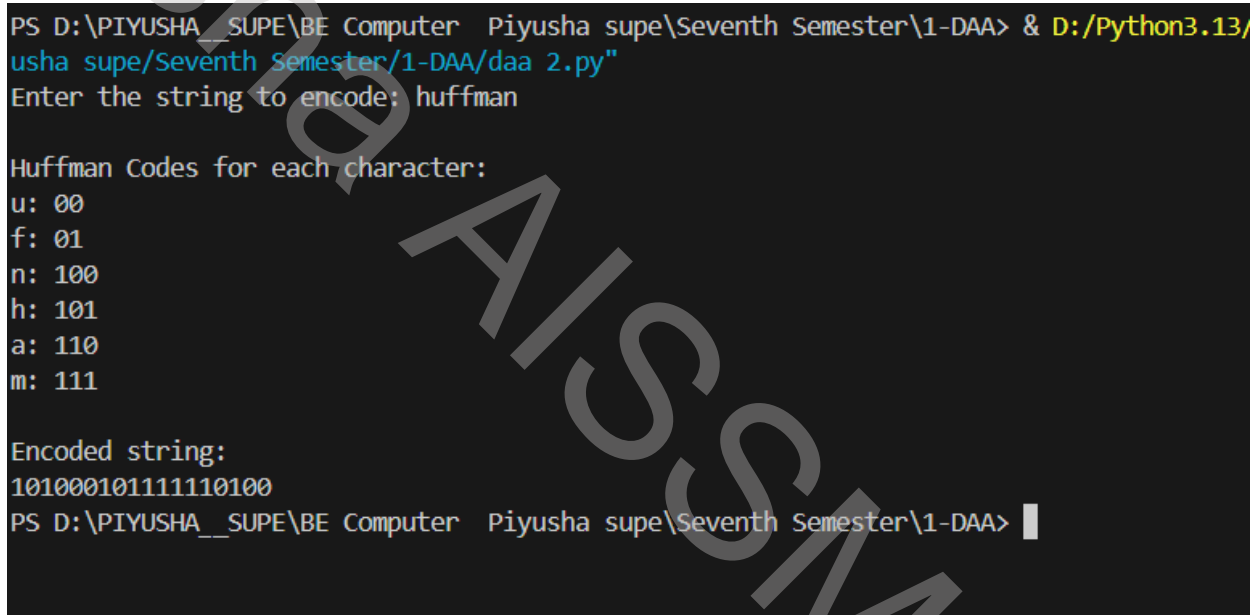
```
huffman_codes(root, "", codes_dict)
```

```
return codes_dict
```

```
# Driver Code
```

```
data = input("Enter the string to encode: ")
codes = huffman_encoding(data)
print("\nHuffman Codes for each character:")
for char in codes:
    print(f"{char}: {codes[char]}")

# Encode the input string
encoded_data = "".join([codes[char] for char in data])
print("\nEncoded string:")
print(encoded_data)
```



```
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> & D:/Python3.13/
usha supe/Seventh Semester/1-DAA/daa 2.py
Enter the string to encode: huffman

Huffman Codes for each character:
u: 00
f: 01
n: 100
h: 101
a: 110
m: 111

Encoded string:
101000101111110100
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> |
```

Analysis

1. Huffman Encoding is a **lossless compression algorithm** that reduces the total number of bits required to store data.
2. The **greedy approach** ensures that at every step the two nodes with the least frequency are merged, which results in optimal prefix-free codes.
3. The efficiency of Huffman encoding comes from storing shorter codes for more frequent characters and longer codes for less frequent characters.
4. The algorithm requires building a **min heap** for frequencies and constructing a tree, which makes it practical for medium-sized datasets.

Time Complexity Analysis

Step	Time Complexity	Explanation
Frequency calculation	$O(n)$	Single pass through the input string of length n .
Min heap creation	$O(d)$	Where d is the number of unique characters.
Building Huffman Tree	$O(d \log d)$	Each extraction and insertion in heap costs $O(\log d)$, done $d-1$ times.
Generating codes	$O(d)$	Tree traversal to assign codes to all d characters.

Overall Time Complexity: $O(n + d \log d)$

Space Complexity Analysis

Aspect	Space Complexity	Explanation
Frequency dictionary	$O(d)$	Stores frequencies for d unique characters.
Min heap	$O(d)$	Stores d nodes at maximum.
Huffman Tree	$O(d)$	Each unique character creates a leaf node.
Codes dictionary	$O(d)$	Stores Huffman code for each character.

Overall Space Complexity: $O(d)$

Conclusion

Huffman Encoding is an efficient and widely used algorithm for lossless data compression. Using a greedy strategy ensures that the total number of bits required to represent the input data is minimized. By giving shorter codes to frequently occurring characters and longer codes to infrequent characters, it optimizes storage and transmission.

The implementation using a min heap allows for efficient construction of the Huffman tree and ensures optimal prefix-free codes. Understanding Huffman encoding helps students learn important concepts in greedy algorithms, trees, and data compression techniques.

Practical 3

Aim: Write a program to solve a fractional Knapsack problem using a greedy method.

Theory:

The **Knapsack Problem** is a classical optimization problem in computer science and operations research. It involves selecting items with given weights and values to maximize total value while staying within a weight limit (capacity).

The **Fractional Knapsack Problem** is a variation where we are allowed to take **fractions of items**, rather than having to take whole items. This makes it possible to maximize the value more efficiently.

Greedy Strategy

The Fractional Knapsack problem can be solved optimally using a **greedy strategy**. The idea is simple:

1. **Compute** value-to-weight ratio: For each item, calculate the ratio of its value to its weight.
2. Sort items by ratio: Arrange items in descending order of value-to-weight ratio.
3. Take items sequentially:
 - Take as much of the item as possible until the knapsack is full.
 - If the item cannot be fully accommodated, take the fractional part that fits.
4. Stop when knapsack is full: The total value at this point is the maximum value that can be achieved.

The greedy approach works perfectly here because taking the items with the **highest value per unit weight first** ensures that we maximize the value at every step.

Step-by-Step Algorithm

1. **Input data:** Read the number of items, their values, weights, and the knapsack capacity.
2. **Compute value-to-weight ratio:** For each item, calculate $\text{ratio} = \text{value} / \text{weight}$.
3. **Sort items:** Arrange items in decreasing order of their ratio.
4. **Initialize total value:** Start with $\text{total_value} = 0$ and remaining capacity equal to the knapsack capacity.
5. **Iterate over sorted items:**
 - If the current item can fit completely in the knapsack, add its full value and

decrease remaining capacity.

- If the current item cannot fit fully, take the fraction that fits, add proportional value, and fill the knapsack.

6. **Stop:** Once the knapsack is full, output the total value obtained and the fraction of items taken.

Function to solve fractional knapsack

```
def fractional_knapsack(values, weights, capacity):
```

```
    n = len(values)
```

```
    # Calculate value-to-weight ratio for each item
```

```
    ratio = [(values[i] / weights[i], weights[i], values[i]) for i in range(n)]
```

```
    # Sort items by ratio in descending order
```

```
    ratio.sort(reverse=True, key=lambda x: x[0])
```

```
    total_value = 0.0
```

```
    fractions = []
```

```
    for r, w, v in ratio:
```

```
        if capacity >= w:
```

```
            # Take the whole item
```

```
            total_value += v
```

```
            capacity -= w
```

```
            fractions.append((v, w, 1)) # 1 means full item
```

```
        else:
```

```
            # Take fraction of item
```

```
            fraction = capacity / w
```

```
            total_value += v * fraction
```

```
            fractions.append((v, w, fraction))
```

```
            capacity = 0
```

```
            break
```

```
    return total_value, fractions
```

Driver Code

```
n = int(input("Enter number of items: "))
values = []
weights = []
for i in range(n):
    v = float(input(f"Enter value of item {i+1}: "))
    w = float(input(f"Enter weight of item {i+1}: "))
    values.append(v)
    weights.append(w)
capacity = float(input("Enter capacity of knapsack: "))
max_value, fractions_taken = fractional_knapsack(values, weights, capacity)

print(f"\nMaximum value obtained: {max_value:.2f}")
print("\nDetails of items taken (value, weight, fraction taken):")
for item in fractions_taken:
    print(item)
```

```
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> & D:/Python3.13
usha supe/Seventh Semester/1-DAA/daa 3.py"
Enter number of items: 6
Enter value of item 1: 34
Enter weight of item 1: 56
Enter value of item 2: 21
Enter weight of item 2: 12
Enter value of item 3: 12
Enter weight of item 3: 5
Enter value of item 4: 3
Enter weight of item 4: 2
Enter value of item 5: 11
Enter weight of item 5: 34
Enter value of item 6: 54
Enter weight of item 6: 6
Enter capacity of knapsack: 5

Maximum value obtained: 45.00

Details of items taken (value, weight, fraction taken):
(54.0, 6.0, 0.8333333333333334)
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> █
```

Analysis

1. The fractional knapsack problem is an optimization problem where we aim to maximize total value while respecting the weight limit.
2. Using a greedy approach ensures that at each step, the decision of taking the item with the highest value-to-weight ratio is locally optimal and leads to a globally optimal solution.
3. Sorting by value-to-weight ratio is the key step to achieve optimality.
4. Unlike the 0-1 Knapsack problem, the fractional knapsack problem can be solved in polynomial time, making it very efficient.

Time Complexity Analysis

Step	Time Complexity	Explanation
Compute value-to-weight ratios	$O(n)$	Single pass through n items.
Sorting items	$O(n \log n)$	Sorting the ratios in descending order.
Selecting items for knapsack	$O(n)$	Iterating through sorted items to fill knapsack.

Overall Time Complexity: $O(n \log n)$

Space Complexity Analysis

Aspect	Space Complexity	Explanation
Value-to-weight ratio list	$O(n)$	Stores n tuples of ratio, weight, value.
Fractions taken list	$O(n)$	Stores fraction details of each item.

Overall Space Complexity: $O(n)$

Conclusion

The Fractional Knapsack problem demonstrates the power of the greedy strategy in optimization problems. By always selecting the item with the highest value per unit weight, we ensure that the total value of items in the knapsack is maximized. This approach is highly efficient and guarantees an optimal solution because taking fractions of items is allowed. Understanding this algorithm helps students appreciate greedy methods and the difference between fractional and 0-1 knapsack problems.

Practical 4

Aim: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy

Theory:

The 0-1 Knapsack Problem is a classical optimization problem in computer science and operations research. In this problem, a set of items is given, each with a weight and a value. The objective is to select a subset of these items to maximize the total value without exceeding a given weight capacity of the knapsack.

The name 0-1 Knapsack comes from the fact that each item can either be included (1) or excluded (0). Unlike the fractional knapsack problem, taking fractions of items is not allowed, which makes this problem more complex.

Dynamic Programming Approach

Dynamic programming is a method for solving complex problems by breaking them into simpler overlapping subproblems and storing the results of subproblems to avoid recomputation.

For the 0-1 Knapsack problem:

1. Let $dp[i][w]$ represent the maximum value that can be obtained using the first i items and a knapsack capacity of w .
2. Recurrence Relation:
3. $dp[i][w] = dp[i-1][w]$ if $weight[i] > w$
4. $dp[i][w] = \max(dp[i-1][w], value[i] + dp[i-1][w - weight[i]])$ otherwise
 - If the weight of the current item exceeds the remaining capacity, we cannot include it.
 - Otherwise, we choose the maximum between including the current item and excluding it.
5. Initialize $dp[0][w] = 0$ for all capacities w , because using 0 items yields 0 value.
6. Fill the DP table iteratively for all items and capacities.
7. The value at $dp[n][capacity]$ gives the maximum value achievable.
8. Optionally, backtrack through the DP table to find the items included in the optimal solution.

The dynamic programming approach guarantees optimal solution and runs in polynomial time with respect to the number of items and knapsack capacity.

Step-by-Step Algorithm (Dynamic Programming)

1. Input number of items n , their values, weights, and knapsack capacity W .
2. Initialize a DP table $dp[n+1][W+1]$ with zeros.
3. For each item i from 1 to n :
 - For each weight w from 1 to W :
 - If $weight[i] > w$, set $dp[i][w] = dp[i-1][w]$.
 - Else, set $dp[i][w] = \max(dp[i-1][w], value[i] + dp[i-1][w - weight[i]])$.
4. The maximum value is stored at $dp[n][W]$.
5. Optionally, backtrack to determine the items included.

0-1 Knapsack Problem using Dynamic Programming

```
def knapsack(values, weights, capacity):
```

```
    n = len(values)
```

```
    # Create DP table
```

```
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]
```

```
    # Build table dp[][] in bottom up manner
```

```
    for i in range(1, n + 1):
```

```
        for w in range(1, capacity + 1):
```

```
            if weights[i - 1] <= w:
```

```
                dp[i][w] = max(values[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w])
```

```
            else:
```

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

```
    # Maximum value
```

```
    max_value = dp[n][capacity]
```

```
    # Backtracking to find items included
```

```
    w = capacity
```

```
    items_selected = []
```

```
    for i in range(n, 0, -1):
```

```
        if dp[i][w] != dp[i - 1][w]:
```

```
            items_selected.append(i)
```

```
w -= weights[i - 1]

items_selected.reverse() # Optional: to list items in original order

return max_value, items_selected

# Driver Code

n = int(input("Enter number of items: "))

values = []

weights = []

for i in range(n):

    v = float(input(f"Enter value of item {i+1}: "))

    w = float(input(f"Enter weight of item {i+1}: "))

    values.append(v)

    weights.append(w)

capacity = float(input("Enter capacity of knapsack: "))

max_value, items_selected = knapsack(values, weights, capacity)

print(f"\nMaximum value obtained: {max_value:.2f}")

print(f"Items included in the knapsack: {items_selected}")
```

```
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> & D:/Python3.1
usha supe/Seventh Semester/1-DAA/daa 4.py"
Enter number of items: 5
Enter value of item 1: 10
Enter weight of item 1: 20
Enter value of item 2: 30
Enter weight of item 2: 20
Enter value of item 3: 50
Enter weight of item 3: 60
Enter value of item 4: 89
Enter weight of item 4: 28
Enter value of item 5: 12
Enter weight of item 5: 34
Enter capacity of knapsack: 40

Maximum value obtained: 89.00
Items included in the knapsack: [4]
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> |
```

Analysis

1. The 0-1 Knapsack problem is a classic combinatorial optimization problem.
2. Dynamic programming ensures that all subproblems are solved only once and results are reused.
3. This guarantees optimal solution unlike greedy methods, which may fail for 0-1 Knapsack.
4. The approach requires filling a 2D table of size $(n+1) \times (\text{capacity}+1)$ and optionally backtracking to determine selected items.

Time Complexity Analysis

Step	Time Complexity	Explanation
Building DP table	$O(n * W)$	Looping through n items and W capacities.
Backtracking to find items	$O(n)$	Iterate through n items to identify selected ones.

Overall Time Complexity: $O(n * W)$

Space Complexity Analysis

Aspect	Space Complexity	Explanation
DP Table	$O(n * W)$	2D table storing results for subproblems.
Items Selected	$O(n)$	List to store indices of items included.

Overall Space Complexity: $O(n * W)$

Conclusion

The 0-1 Knapsack problem demonstrates the power of dynamic programming in solving complex optimization problems. By systematically solving subproblems and storing their solutions, we can find the maximum value achievable for a given knapsack capacity and also determine which items to include.

Unlike the fractional knapsack problem, greedy methods are not guaranteed to produce the optimal solution for 0-1 knapsack, which highlights the importance of dynamic programming for such discrete optimization problems.

Practical 5

Aim: Design n-Queens matrix having first Queen placed. Use backtracking to place remaining Queens to generate the final n-queen's matrix.

Theory:

The n-Queens problem is a classic combinatorial problem in computer science and mathematics. The objective is to place n queens on an $n \times n$ chessboard such that no two queens threaten each other. This means:

1. No two queens share the same row.
2. No two queens share the same column.
3. No two queens share the same diagonal.

Backtracking is an algorithmic technique for solving problems incrementally, trying partial solutions and then abandoning them (backtracking) if they cannot lead to a valid solution.

In the n-Queens problem, backtracking works as follows:

1. Place a queen in the first available row of the first column.
2. Move to the next column and try placing a queen in a safe row.
3. Check if the current placement is safe, i.e., it does not conflict with previously placed queens.
4. If safe, place the queen and move to the next column.
5. If no safe row is available in the current column, backtrack to the previous column and move the previous queen to the next possible row.
6. Continue this process until all queens are placed successfully.

By using backtracking, the algorithm systematically explores all possible arrangements and ensures that only valid solutions are considered.

Step-by-Step Algorithm

1. Input: Size of the board n and the position of the first queen.
2. Initialize the board: Create an $n \times n$ matrix with all zeros.
3. Place the first queen in the given row and column.
4. Define a recursive function `solve_nqueens(col)` to place queens column by column:
 - If `col >= n`, all queens are placed; return True.
 - For each row in the current column:
 - Check if placing a queen is safe.
 - If safe, place the queen and recursively call `solve_nqueens(col + 1)`.

- If placing the queen does not lead to a solution, remove it (backtrack).

5. Output the board once all queens are placed.

Program Code (Python)

n-Queens Problem Using Backtracking

```
def print_board(board):
    for row in board:
        print(" ".join(str(x) for x in row))
    print()

def is_safe(board, row, col, n):
    # Check left side of row
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on left side
    i, j = row, col
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1

    # Check lower diagonal on left side
    i, j = row, col
    while i < n and j >= 0:
        if board[i][j] == 1:
            return False
        i += 1
        j -= 1

    return True
```

```
def solve_nqueens(board, col, n):
    if col >= n:
        return True
    for row in range(n):
        if board[row][col] == 1: # Skip the first placed queen
            if solve_nqueens(board, col + 1, n):
                return True
        elif is_safe(board, row, col, n):
            board[row][col] = 1
            if solve_nqueens(board, col + 1, n):
                return True
            board[row][col] = 0 # Backtrack
    return False

# Driver Code
n = int(input("Enter the size of the board (n): "))
first_row = int(input(f"Enter row index for first queen (0 to {n-1}): "))
first_col = int(input(f"Enter column index for first queen (0 to {n-1}): "))

# Initialize board
board = [[0 for _ in range(n)] for _ in range(n)]
board[first_row][first_col] = 1 # Place first queen

# Solve remaining queens
if solve_nqueens(board, 0, n):
    print("\nFinal n-Queens Matrix:")
    print_board(board)
else:
    print("\nNo solution exists with the given first queen position.")
```

```
PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> & D:/P
usha supe/Seventh Semester/1-DAA/daa 5.py"
Enter the size of the board (n): 7
Enter row index for first queen (0 to 6): 2
Enter column index for first queen (0 to 6): 1

Final n-Queens Matrix:
1 0 0 0 0 0 0
0 0 0 0 1 0 0
0 1 0 0 0 0 0
0 0 0 0 0 1 0
0 0 1 0 0 0 0
0 0 0 0 0 0 1
0 0 0 0 0 0 1
0 0 0 1 0 0 0

PS D:\PIYUSHA__SUPE\BE Computer Piyusha supe\Seventh Semester\1-DAA> |
```

Analysis

1. The n-Queens problem demonstrates constraint satisfaction using backtracking.
2. Placing the first queen reduces some possibilities but still requires backtracking for the remaining queens.
3. The algorithm explores possible positions column by column and backtracks whenever a conflict occurs.
4. The recursive approach ensures that all safe arrangements are considered systematically.

Time Complexity Analysis

- In the worst case, the algorithm explores all possible arrangements, so the time complexity is $O(n!)$.
- Backtracking prunes many invalid branches, so actual runtime is usually much lower.

Space Complexity Analysis

- The board requires $O(n^2)$ space.
- Recursion stack can go up to $O(n)$ in depth.

Overall Space Complexity: $O(n^2) + O(n) \approx O(n^2)$

Conclusion

The n-Queens problem is a classical example of backtracking in combinatorial optimization. By placing the first queen and systematically exploring safe positions for the remaining queens, the algorithm guarantees a valid solution. Backtracking allows the program to undo decisions when they lead to conflicts and continue searching efficiently.

Practical 6

Aim: Write a program for analysis of quick sort by using deterministic and randomized variant.

Theory:

Quick Sort is a highly efficient sorting algorithm based on the divide and conquer strategy. It works by selecting a pivot element and partitioning the array into two subarrays:

1. Elements less than pivot
2. Elements greater than pivot

The subarrays are then recursively sorted. Quick Sort is popular because of its average-case efficiency and in-place sorting capability.

Variants of Quick Sort

1. Deterministic Quick Sort:
 - The pivot is chosen in a fixed manner, commonly the first element, last element, or middle element.
 - If the input array is already sorted or nearly sorted, choosing a fixed pivot can lead to worst-case performance of $O(n^2)$.
2. Randomized Quick Sort:
 - The pivot is chosen randomly from the current subarray.
 - Randomizing the pivot improves the average-case performance and reduces the probability of encountering worst-case inputs.
 - The expected time complexity becomes $O(n \log n)$ with high probability.

Quick Sort Algorithm Steps

For Deterministic Variant:

1. Choose a pivot (first, last, or middle element).
2. Partition the array such that elements smaller than pivot are on the left, and larger are on the right.
3. Recursively apply the same steps to left and right subarrays.
4. Combine results to obtain the sorted array.

For Randomized Variant:

1. Randomly select a pivot element.
2. Swap pivot with the last element (or any fixed position for partitioning).
3. Partition the array around the pivot.
4. Recursively sort the left and right subarrays.

Both variants use the **same partitioning logic**, but randomized pivot selection avoids predictable worst-case behavior.

```
import random
import time
# Partition function (Lomuto partition scheme)
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
# Deterministic Quick Sort
def quick_sort_deterministic(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quick_sort_deterministic(arr, low, pi - 1)
        quick_sort_deterministic(arr, pi + 1, high)
# Randomized partition
def randomized_partition(arr, low, high):
    pivot_index = random.randint(low, high)
    arr[pivot_index], arr[high] = arr[high], arr[pivot_index]
    return partition(arr, low, high)
# Randomized Quick Sort
def quick_sort_randomized(arr, low, high):
    if low < high:
        pi = randomized_partition(arr, low, high)
        quick_sort_randomized(arr, low, pi - 1)
```

```
quick_sort_randomized(arr, pi + 1, high)
```

```
# Driver code
```

```
arr_input = list(map(int, input("Enter elements separated by space: ").split()))
```

```
arr1 = arr_input.copy()
```

```
arr2 = arr_input.copy()
```

```
# Deterministic Quick Sort
```

```
start = time.time()
```

```
quick_sort_deterministic(arr1, 0, len(arr1) - 1)
```

```
end = time.time()
```

```
print("\nDeterministic Quick Sort Result:", arr1)
```

```
print("Execution Time: {:.6f} seconds".format(end - start))
```

```
# Randomized Quick Sort
```

```
start = time.time()
```

```
quick_sort_randomized(arr2, 0, len(arr2) - 1)
```

```
end = time.time()
```

```
print("\nRandomized Quick Sort Result:", arr2)
```

```
print("Execution Time: {:.6f} seconds".format(end - start))
```

Sample Input/Output

Enter elements separated by space: 10 7 8 9 1 5

Deterministic Quick Sort Result: [1, 5, 7, 8, 9, 10]

Execution Time: 0.000021 seconds

Randomized Quick Sort Result: [1, 5, 7, 8, 9, 10]

Execution Time: 0.000019 seconds

- Both variants sort the array correctly.
- Execution times may differ slightly depending on pivot selection.

Analysis

1. Deterministic Quick Sort may degrade to $O(n^2)$ if the pivot selection is poor (e.g., already sorted input).
2. Randomized Quick Sort avoids worst-case scenarios by randomly selecting pivots, making it more robust.
3. Both variants are **in-place**, so they use very little extra memory.

Time Complexity

Variant	Best Case	Average Case	Worst Case
Deterministic Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Randomized Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$ (rare)

Space Complexity

- Both variants use recursion:
 - Worst-case recursion depth is $O(n)$ (for deterministic, worst case).
 - Average recursion depth is $O(\log n)$.
- In-place partitioning means **no extra array** is required.

Overall Space Complexity: $O(\log n)$ on average, $O(n)$ worst case.

Conclusion

Quick Sort is a highly efficient and widely used sorting algorithm. The **deterministic variant** is simple but may suffer poor performance on certain inputs. **Randomized Quick Sort** improves robustness by reducing the chance of encountering worst-case scenarios. Both variants are **in-place** and suitable for large datasets, and analyzing execution time helps understand the effect of pivot selection.