Assignment No. 1

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

Objectives:

- To find out Fibonacci numbers using recursive and non-recursive method.
- To understand how recursive and non-recursive algorithms works.
- To analyze time complexity of both the algorithm.

Outcome:

- Student will be able to implement Fibonacci numbers using recursive and non-recursive method.
- Student will be able to analyze time complexity of both the algorithm.

Theory:

In Maths, the **Fibonacci numbers** are the numbers ordered in a distinct **Fibonacci sequence**. These numbers were introduced to represent the positive numbers in a sequence, which follows a defined pattern. The list of the numbers in the **Fibonacci series** is represented by the recurrence relation: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ……..,∞.

A Fibonacci number is a series of numbers in which each Fibonacci number is obtained by adding the two preceding numbers. It means that the next number in the series is the addition of two previous numbers. Let the first two numbers in the series be taken as 0 and 1. By adding 0 and 1, we get the third number as 1. Then by adding the second and the third number (i.e) 1 and 1, we get the fourth number as 2, and similarly, the process goes on. Thus, we get the Fibonacci series as 0, 1, 1, 2, 3, 5, 8, ……. Hence, the obtained series is called the Fibonacci number series.

Fibonacci Numbers Formula: The sequence of Fibonacci numbers can be defined as:

$$F_n = F_{n-1} + F_{n-2}$$

Where F_n is the nth term or number

F_{n-1} is the (n-1)th term

F_{n-2} is the (n-2)th term

From the equation, we can summarize the definition as, the next number in the sequence, is the sum of the previous two numbers present in the sequence, starting from 0 and 1.

So, the first six terms of Fibonacci sequence is 0,1,1,2,3,5, 8, 13, 21, 34……

Implementation of Fibonacci numbers using non-recursive algorithm.

```
classFib_nonrec {

static void Fibonacci(int N)
   {
int num1 = 0, num2 = 1;
int counter = 10;
while (counter > 0) {

          System.out.print(num1 + " ");
          int num3 = num2 + num1;
num1 = num2;
             num2 = num3;
          counter= counter - 1;

     }   }
public static void main(String args[])
   {
int N = 10;
Fibonacci(N);
   }
}
```

Analysis of Algorithm:

- The time complexity of the Fibonacci series is T(N) i.e, linear. We have to find the sum of two terms and it is repeated n times depending on the value of n.
- The space complexity of the Fibonacci series using dynamic programming is O(1).

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily.
Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals,
A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems. Many more recursive calls can be generated as and when required. It is essential to know that we should provide a certain case in order to terminate this recursion process. So we can say that every time the function calls itself with a simpler version of the original problem.

Need of Recursion: Recursion is an amazing technique with the help of which we can reduce the length of our code and make it easier to read and write. It has certain advantages over the iteration technique which will be discussed later. A task that can be defined with its similar subtask, recursion is one of the best solutions for it.

Implementation of Fibonacci numbers using non recursive algorithm (Method-1).

```
public class FibonacciExample1{
  public static void main(String args[]) {
    int n1 = 0, n2 = 1, n3, i, max = 5;
    System.out.print(n1 + " " + n2);
    for (i = 2; i < max; ++i) {
      n3 = n1 + n2;
      System.out.print(" " + n3);
      n1 = n2;
      n2 = n3;
    }
  }
}
```

Analysis of Algorithm:

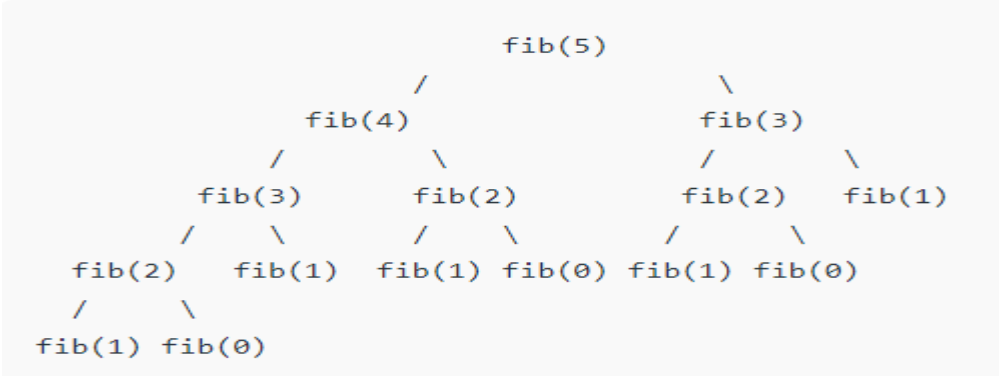
- The time complexity of the Fibonacci series is T(N) i.e, linear. We have to find the sum of two terms and it is repeated n times depending on the value of n.
- The space complexity of the Fibonacci series using dynamic programming is O(1).

Implementation of Fibonacci numbers using recursive algorithm (Method-2).

```
class FibonacciExample2{
staticint n1=0,n2=1,n3=0;
static void printFibonacci(int count){
if(count>0){
      n3 = n1 + n2;
      n1 = n2;
      n2 = n3;
System.out.print(" "+n3);
printFibonacci(count-1);
   }
 }
public static void main(String args[]){
int count=10;
System.out.print(n1+" "+n2);//printing 0 and 1
printFibonacci(count-2);//n-2 because 2 numbers are already printed
 }
}
```

Analysis of Algorithm:

Time Complexity: Exponential, as every function calls two other functions.If the original recursion tree were to be implemented then this would have been the tree but now for n times the recursion function is calledOriginal tree for recursion



The space complexity of the Fibonacci series using dynamic programming is O(1).

Conclusion: both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than the iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

It also shows that if recursion is not implemented properly then it leads to overhead and increase in time complexity.

Assignment No. 2

Title: Write a Programme to implement Hummfan's coding algorithm Using Greedy Method

Objectives:

- To find out Hummfan's coding tree using greedy method.
- To understand how greedy algorithms works.
- To analyze time complexity of the algorithm.

Outcome:

- Student will be able to implement Fibonacci numbers using recursive and non-recursive method.
- Student will be able to analyze time complexity of both the algorithm.
- Student will be able to implement an algorithm using Greedy algorithm design strategy.

Theory:

Greedy Method: Among all the algorithmic approaches, the simplest and straightforward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future. Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

Greedy algorithm is designed using the following steps –

1. Characterize the structure of an optimal solution.
2. Find the set of feasible solutions.
3. Find the locally optimal solutions.

Huffman's Tree: Huffman tree or Huffman coding tree defines as a full binary tree in which each leaf of the tree corresponds to a letter in the given alphabet. The Huffman tree is treated as the binary tree associated with minimum external path weight that means, the one associated with the minimum sum of weighted path lengths for the given set of leaves. So the goal is to construct a tree with the minimum external path weight.

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 most frequent character gets the smallest code and the least frequent character gets the largest code.

There are two to methods to Construct Huffman's Tree.

1. Fixed Length Method
2. Variable Length Method

Steps to Create Huffman's Tree.

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes.

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

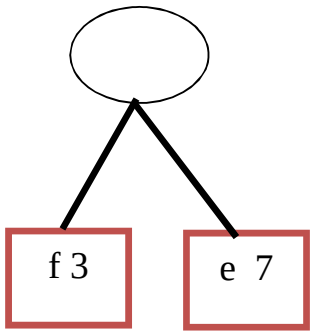
Example: Obtain the Huffman's Encoding for the following Data. And encode the code message for word "baba" Using Variable length method.

Letters	a	b	c	d	e	f
Frequency	39	10	9	25	7	3

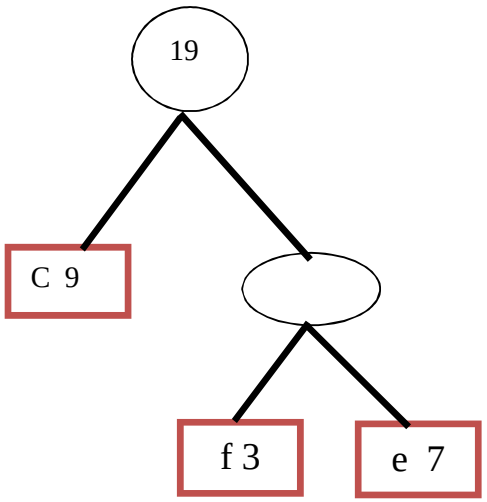
Solution:

Step 1: Arrange the letters in ascending order of frequency.

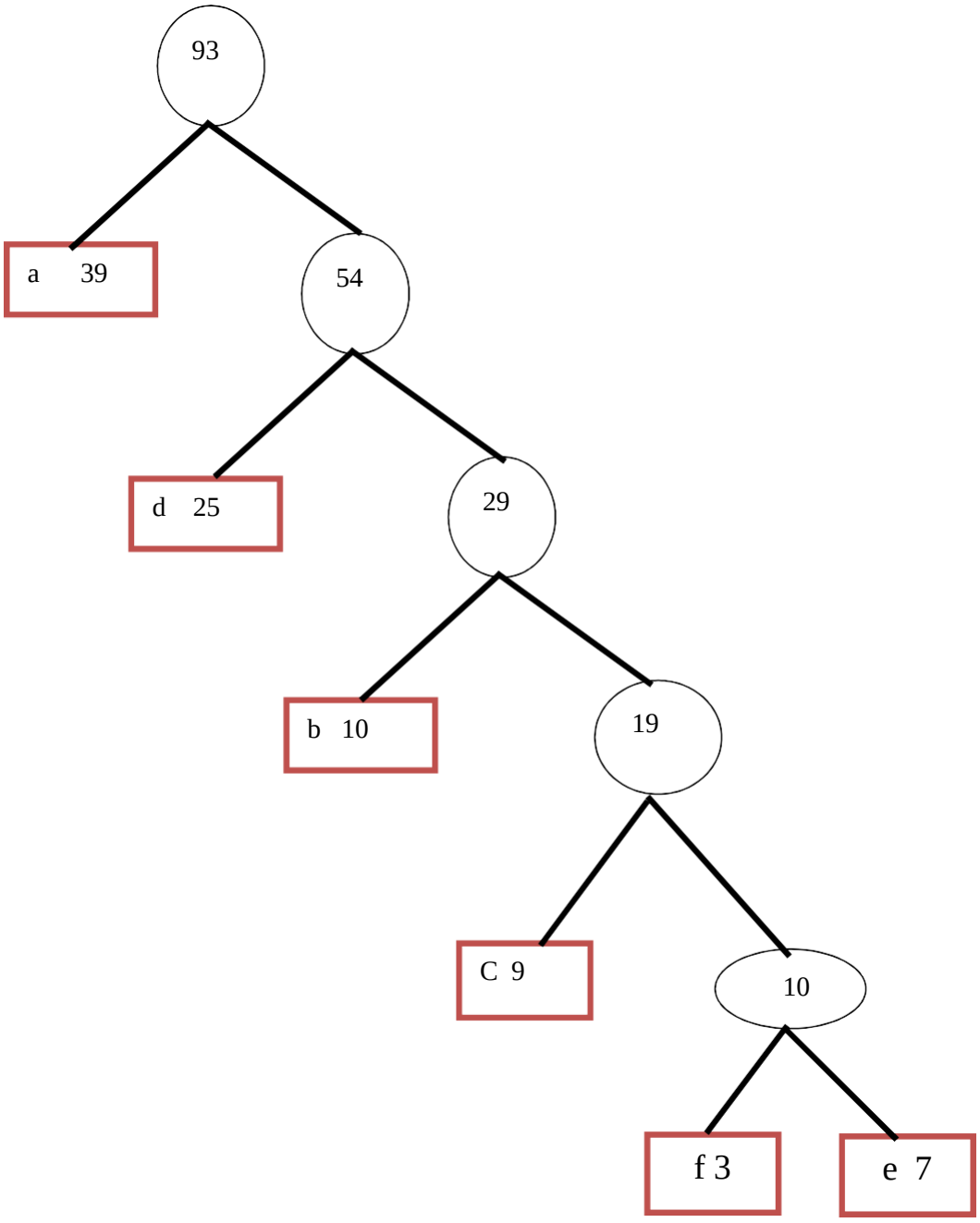
Step 2: Pick first two Object and marge them to generate parent node.



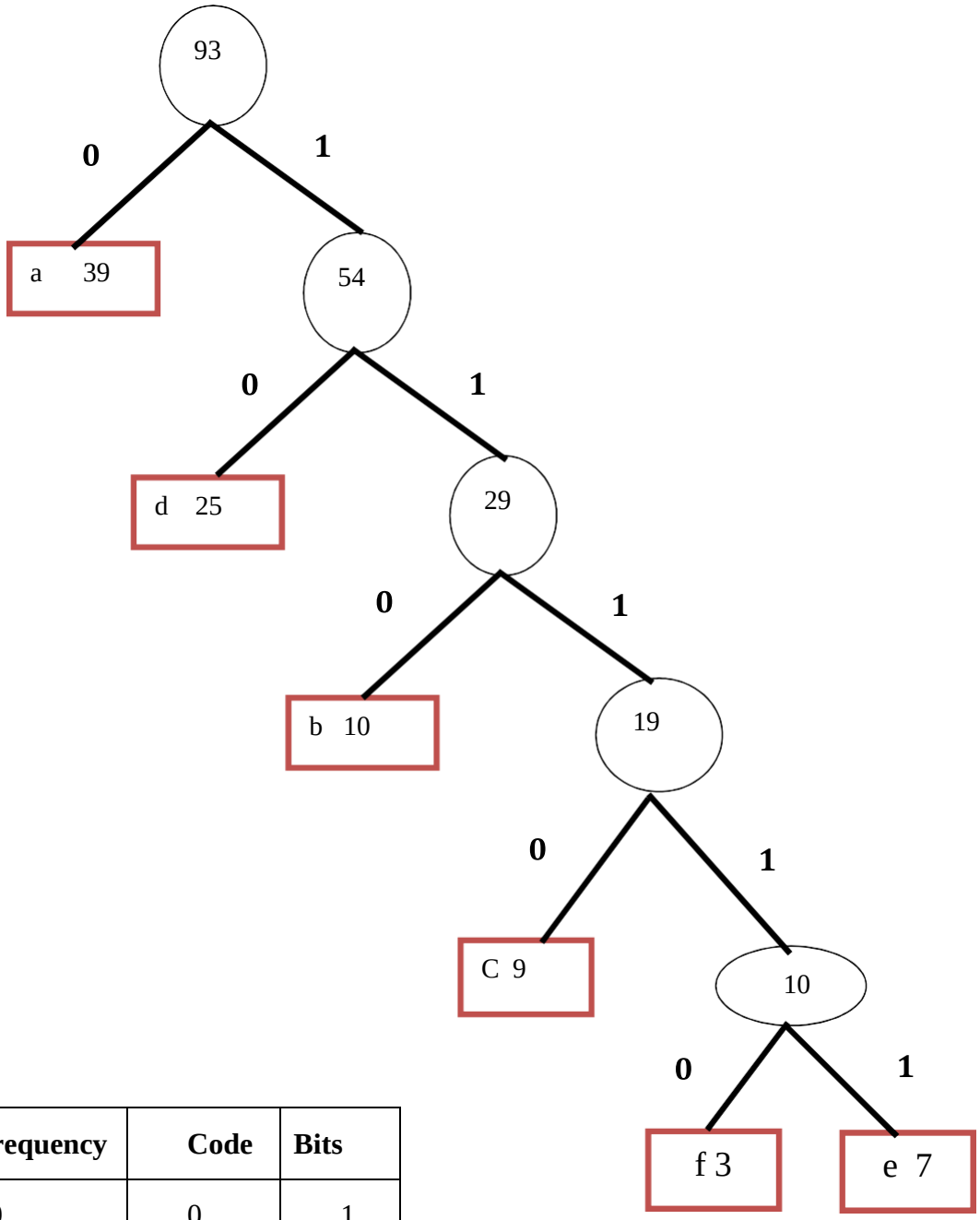
- Step 3: Pick next Object and marge them with newly generated node, it will be inserted in left side if its value is less than newly generated node's value else at right side.



Repeat the process till we get final tree



Step 5: Assign Huffman's Code



Symbol	Frequency	Code	Bits
a	39	0	1
b	10	110	3
c	9	1110	4
d	25	10	2
e	7	11111	5
f	3	11110	5

Encoding the word baba: **11001100**

Analysis of Algorithm:

- The time complexity of the Huffman encoding is **O(nlogn).** Where n is the number of characters in the given text.
- The child node holds the input character. It is assigned the code formed by subsequent 0s and 1s. The time complexity of decoding a string is O(n), where n is the length of the string.
- The space complexity of the Fibonacci series using dynamic programming is O(n).

Applications of Huffman Coding:

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.
3. Multimedia codecs like JPEG, PNG, and MP3 use Huffman encoding.

Conclusion: The Huffman algorithm is a greedy algorithm. Since at every stage the algorithm looks for the best available options. Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for Greedy. But still this method does not guarantee that the solution obtained is best optimal solution.

Assignment No. 3

Title: Write a Program to solve Fractional Knapsack Problem Using Greedy Method

Objectives:

- To Solve Fractional Knapsack Problem using greedy method.
- To understand how greedy algorithms works.
- To analyze time complexity of the algorithm.

Outcome:

- Student will be able to implement Fractional Knapsack Problem Using Greedy Method.
- Student will be able to analyze time complexity of the Fractional Knapsack algorithm.
- Student will be able to implement an algorithm using Greedy algorithm design strategy.

Theory:

What is a Greedy Method?

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

Drawback of Greedy Approach

- As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm
- For example, suppose we want to find the longest path in the graph below from root to leaf.

Greedy Algorithm

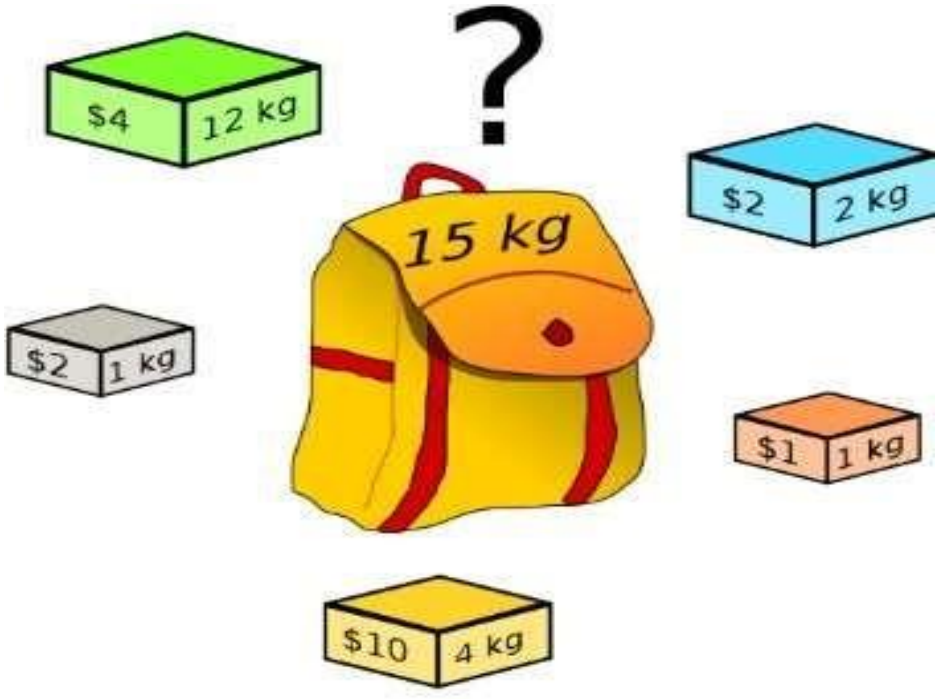
1. To begin with, the solution set (containing answers) is empty.
2. At each step, an item is added to the solution set until a solution is reached.
3. If the solution set is feasible, the current item is kept.
4. Else, the item is rejected and never considered again.

Knapsack Problem

The knapsack problem is a problem in combinatorial optimization, Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem Variants: Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

Fractional Knapsack Problem- In Fractional Knapsack Problem,

- As the name suggests, items are divisible here.
- We can even put the fraction of any item into the knapsack if taking the complete item is not Possible.
- It is solved using the Greedy Method.

Fractional Knapsack Problem Using Greedy Method-

Fractional knapsack problem is solved using greedy method in the following steps-

Step-01: For each item, compute its value / weight ratio.

Step-02: Arrange all the items in decreasing order of their value / weight ratio.

Step-03: Start putting the items into the knapsack beginning from the item with the highest ratio. Put as many items as you can into the knapsack.

Example: For the given instance of a knapsack problem find the optimal solution using greedy approach if n=3 and M=20.

Solution: To Solve the Problem using greedy approach perform following steps.

Find fractions for Descending order of Profit and weight ratio

i	**1**	**2**	**3**
Pi/Wi	30/18	21/15	18/10
Ratio	1.67	1.4	1.8

i	**3**	**1**	**2**
Pi	18	30	21
Wi	10	18	15
Ratio in Descending Order	1.8	1.67	1.4

- The first object with weight 10 will be placed completely so X_1 will be 1.

- The second object will no be t placed completely so find fraction X_2,

$$X_2 = (20 - 10) / 18 = 10 / 18 \qquad = 5/ 9$$

- The Third object will not be placed at all so fraction $X_3 = 0$.
- So the Fraction set is
- And now let us compute W_iP_i and W_iX_i

i	Fraction
X_1	1
X_2	5/9
X_3	0

So,

$$\sum (W_iP_i) = (30 * (5/ 9)) + (21 * 0) + (18 * 1) = 34.67$$

&

$$\sum (W_iX_i) = (18 * (5/ 9)) + (15 * 0) + (10 * 1) = 20$$

Hence the maximum Profit obtained is 34.67

Time Complexity-

- The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- And algorithm take O(n log n) time to solve problem if we use quick sort for sorting.

Conclusion-

A greedy algorithm is the most straightforward approach to solving the knapsack problem, in that it is a one-pass algorithm that constructs a single final solution. At each stage of the problem, the greedy algorithm picks the option that is locally optimal, meaning it looks like the most suitable option right now

Assignment Question

1. **What is Greedy Approach?**
2. **Explain concept of fractional knapsack**
3. **Difference between Fractional and 0/1 Knapsack**
4. **Solve one example based on Fractional knapsack(Other than Manual)**

Assignment No. 4

Title of the Assignment: Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

Objective of the Assignment: Students should be able to understand and solve 0-1 Knapsack problem using dynamic programming

Objectives:

- To Solve 0/1 Knapsack Problem using dynamic programming.
- To understand how dynamic programming works.
- To analyze time complexity of the algorithm.

Outcome:

- Student will be able to implement 0/1 Knapsack Problem Using dynamic programming.
- Student will be able to analyze time complexity of the 0/1 Knapsack algorithm.
- Student will be able to implement an algorithm using dynamic programming design strategy.

Theory:

What is Dynamic Programming?

- Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems.
- Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.
- Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

- For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

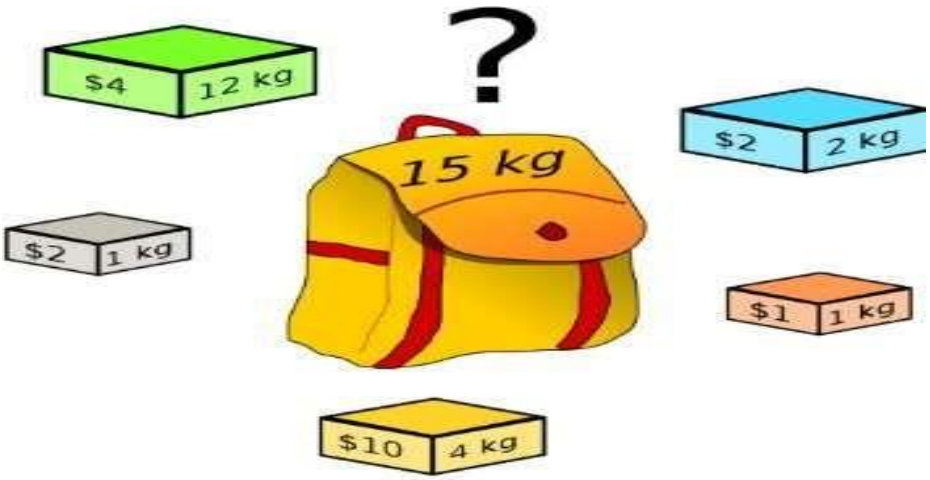
- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem
- Knapsack Problem

You are given the following for Knapsack Problem-

- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and value.

The problem states which items should be placed into the knapsack such that-

- The value or profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed.



Knapsack Problem Variants: Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible here.
- We can not take a fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using a dynamic programming approach.

0/1 Knapsack Problem Using Greedy Method-

Consider:

- Knapsack weight capacity = w
- Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01:

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with zeroes as shown-

	0	1	2	3	.....	W
0	0	0	0	0	.....	0
1	0					
2	0					
.....	.....					
n	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right.

Use the following formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.

- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-03:

- To identify the items that must be put into the knapsack to obtain that maximum profit, consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack

Problem-. For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of a dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

n = 4

w = 5 kg

(w1, w2, w3, w4) = (2, 3, 4, 5)

(b1, b2, b3, b4) = (3, 4, 5, 6)

Solution-

Given: Knapsack capacity (w) = 5 kg and Number of items (n) = 4

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with 0.

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

Finding T(1,1)-

We have,

- i = 1
- j = 1
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,1) = \max \{ T(1-1, 1), 3 + T(1-1, 1-2) \}$

$T(1,1) = \max \{ T(0,1), 3 + T(0,-1) \}$

$T(1,1) = T(0,1)$ { Ignore T(0,-1) }

$T(1,1) = 0$

Finding T(1,2)-

We have,

- i = 1
- j = 2
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,2) = \max \{ T(1-1, 2), 3 + T(1-1, 2-2) \}$

$T(1,2) = \max \{ T(0,2), 3 + T(0,0) \}$

$T(1,2) = \max \{0, 3+0\}$

$T(1,2) = 3$

Finding T(1,3)-

We have,

- i = 1
- j = 3
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,3) = max { T(1-1 , 3) , 3 + T(1-1 , 3-2) }

T(1,3) = max { T(0,3) , 3 + T(0,1) }

T(1,3) = max {0 , 3+0}

T(1,3) = 3

Finding T(1,4)-

We have,

- i = 1
- j = 4
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) }

T(1,4) = max { T(0,4) , 3 + T(0,2) }

T(1,4) = max {0 , 3+0}

T(1,4) = 3

Finding T(1,5)-

We have,

- i = 1
- j = 5
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }

T(1,5) = max { T(0,5) , 3 + T(0,3) }

T(1,5) = max {0 , 3+0}

T(1,5) = 3

Finding T(2,1)-

We have,

- i = 2
- j = 1
- $(value)_i = (value)_2 = 4$
- (weight)i = (weight)2 = 3

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1) { Ignore T(1,-2) }

T(2,1) = 0

Finding T(2,2)-

We have,

- i = 2
- j = 2
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2) { Ignore T(1,-1) }

T(2,2) = 3

Finding T(2,3)-

We have,

- i = 2
- j = 3
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }

T(2,3) = max { T(1,3) , 4 + T(1,0) }

T(2,3) = max { 3 , 4+0 }

T(2,3) = 4

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

Identifying Items to Be Put Into Knapsack

Following Step-04,

- We mark the rows labelled “1” and “2”.
- Thus, items that must be put into the knapsack to obtain the maximum value 7 are- Item-1 and Item-2

Time Complexity-

- Each entry of the table requires constant time θ(1) for its computation.
- It takes θ(nw) time to fill (n+1)(w+1) table entries.
- It takes θ(n) time for tracing the solution since tracing process traces the n rows.
- Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming

Conclusion-In this way we have explored Concept of 0/1 Knapsack using Dynamic approch

Oral Questions

1. **What is Dynamic Approach?**
2. **Explain concept of 0/1 knapsack**
3. **Difference between Dynamic and Branch and Bound Approach.Which is best?**
4. **Solve one example based on 0/1 knapsack (Other than Manual)**

Assignment No. 5

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

Objectives:

- To N-Queen Problem using backtracking.
- To understand how backtracking works.
- To analyze time complexity of the algorithm.

Outcome:

- Student will be able to implement N-Queen Problem using backtracking.
- Student will be able to analyze time complexity of the N-Queen Problem algorithm.
- Student will be able to implement an algorithm using backtracking design strategy.

Introduction to Backtracking

- Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

Suppose we have to make a series of decisions among various choices, where

- We don't have enough information to know what to choose
- Each decision leads to a new set of choices.
- Some sequence of choices (more than one choices) may be a solution to your problem.

What is backtracking?

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it.

In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

- start with a sub-solution
- check if this sub-solution will lead to the solution or not
- If not, then come back and change the sub-solution and continue again

Applications of Backtracking:

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

N queens on NxN chessboard

One of the most common examples of the backtracking is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally. The solution to this problem is also attempted in a similar way. We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places. We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left. If no safe place is left, then we change the position of the previously placed queen.

N-Queens Problem:

A classic combinational problem is to place n queens on a n*n chess board so that no two attack, i.,e

no two queens are on the same row, column or diagonal.

What is the N Queen Problem?

N Queen problem is the classical Example of backtracking. N-Queen problem is defined as, "given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in the same row, column or diagonal".

- For N = 1, this is a trivial case. For N = 2 and N = 3, a solution is not possible. So we start with N = 4 and we will generalize it for N queens.

If we take n=4then the problem is called the 4 queens problem. If we take n=8 then the problem is called the 8 queens problem. **Algorithm**

1) Start in the leftmost column

2) If all queens are place return true

3) Try all rows in the current column.

Do following for every tried row.

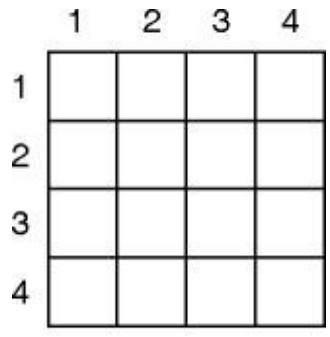
a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.

b) If placing the queen in [row, column] leads to a solution then return true.

c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

4) If all rows have been tried and nothing worked,return false to trigger backtracking.

4- Queen Problem: Given 4 x 4 chessboard, arrange four queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column, or diagonal.



4 x 4 Chessboard

- We have to arrange four queens, Q1, Q2, Q3 and Q4 in 4 x 4 chess board. We will put with queen in ith row. Let us start with position (1, 1). Q1 is the only queen, so there is no issue. partial solution is <1>

- We cannot place Q2 at positions (2, 1) or (2, 2). Position (2, 3) is acceptable. the partial solution is <1, 3>.
- Next, Q3 cannot be placed in position (3, 1) as Q1 attacks her. And it cannot be placed at (3, 2), (3, 3) or (3, 4) as Q2 attacks her. There is no way to put Q3 in the third row. Hence, the algorithm backtracks and goes back to the previous solution and readjusts the position of queen Q2. Q2 is moved from positions (2, 3) to
  (2, 4). Partial solution is <1, 4>
- Now, Q3 can be placed at position (3, 2). Partial solution is <1, 4, 3>.
- Queen Q4 cannot be placed anywhere in row four. So again, backtrack to the previous solution and readjust the position of Q3. Q3 cannot be placed on (3, 3) or(3, 4). So the algorithm backtracks even further.
- All possible choices for Q2 are already explored, hence the algorithm goes back to partial solution <1> and moves the queen Q1 from (1, 1) to (1, 2). And this process continues until a solution is found.

All possible solutions for 4-queen are shown in fig (a) & fig. (b)

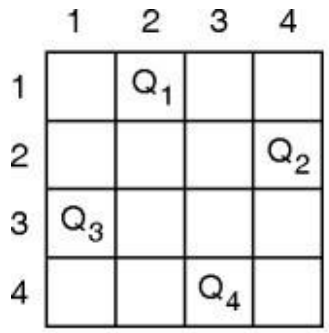


Fig. (a): Solution – 1

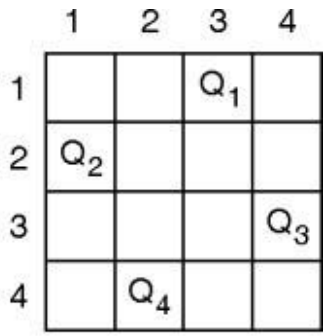
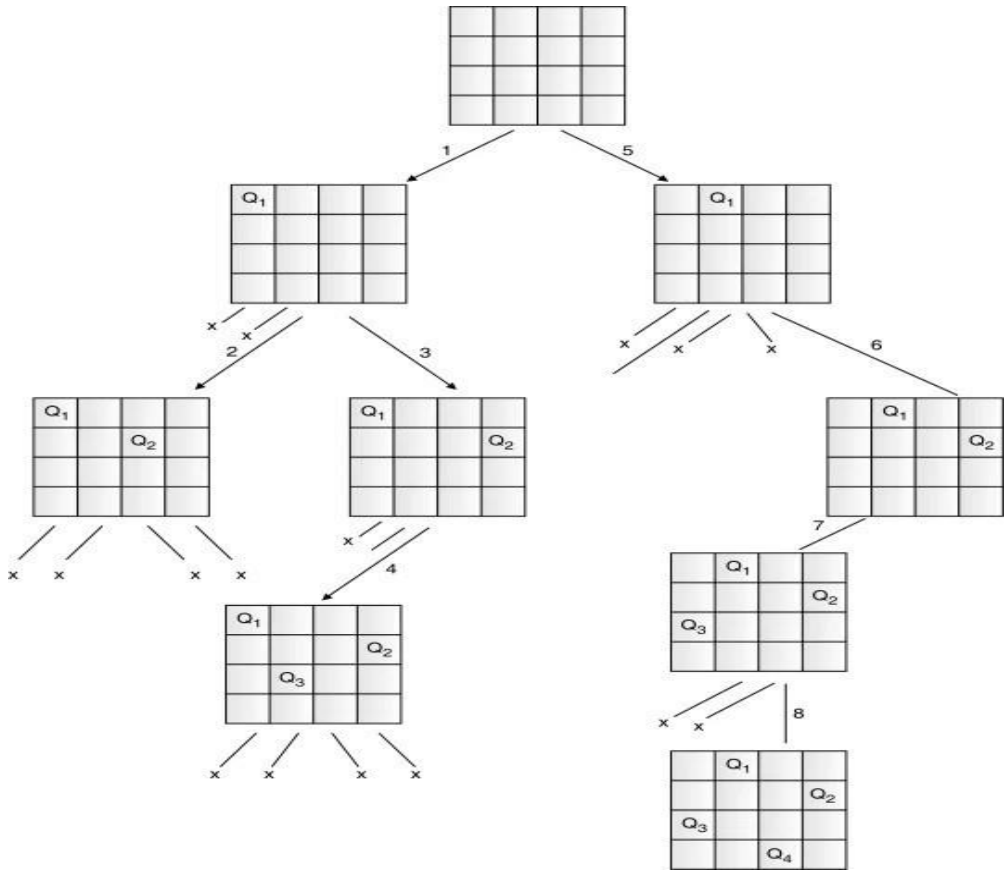


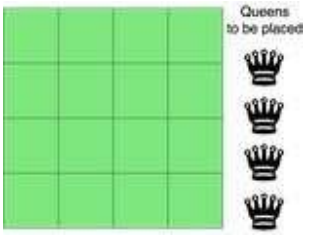
Fig. (b): Solution – 2

Fig. (d) describes the backtracking sequence for the 4-queen problem.

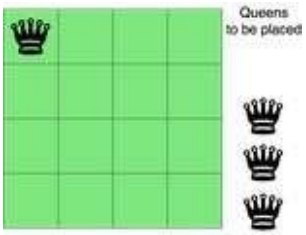


The solution of the 4-queen problem can be seen as four tuples (x_1, x_2, x_3, x_4), where x_i represents the column number of queen Q_i. Two possible solutions for the 4-queen problem are (2, 4, 1, 3) and (3, 1, 4, 2).

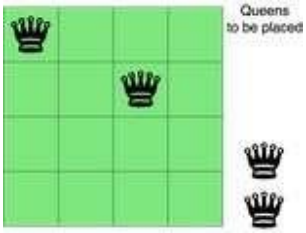
Explanation :

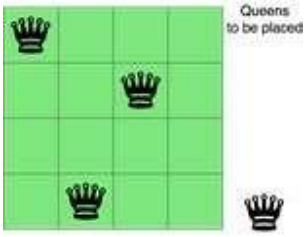


The above picture shows an NxN chessboard and we have to place N queens on it. So, we will start by placing the first queen.



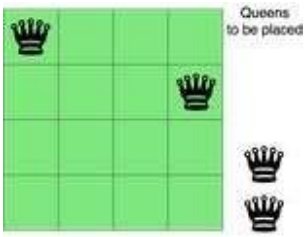
Now, the second step is to place the second queen in a safe position and then the third queen.



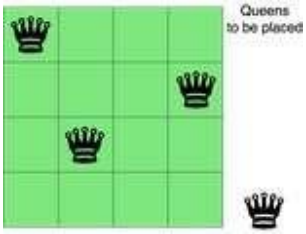


Now, you can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen. And this is backtracking.

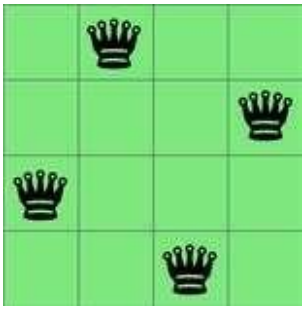
Also, there is no other position where we can place the third queen so we will go back one more step and change the position of the second queen.



And now we will place the third queen again in a safe position until we find a solution.

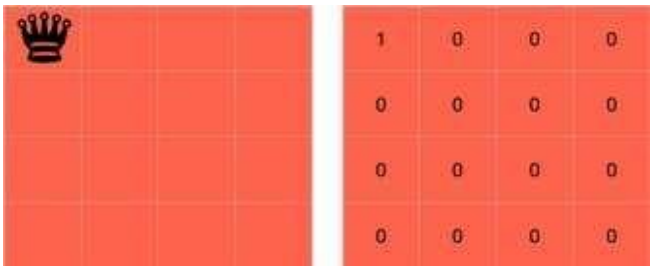


We will continue this process and finally, we will get the solution as shown below.

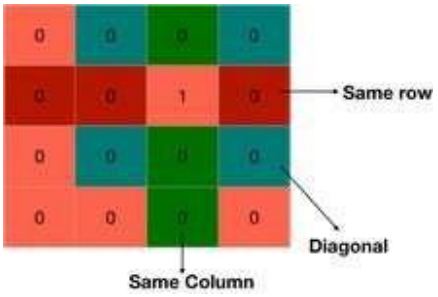


We need to check if a cell (i, j) is under attack or not. For that, we will pass these two in our function along with the chessboard and its size - IS-ATTACK(i, j, board, N).

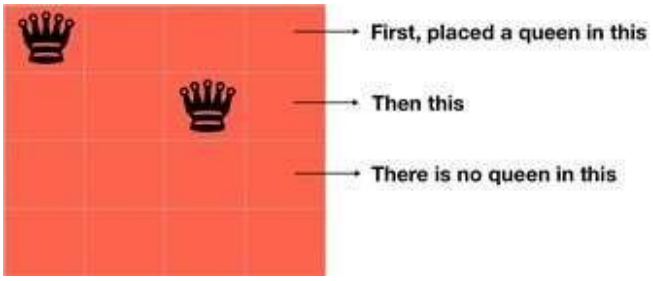
If there is a queen in a cell of the chessboard, then its value will be 1, otherwise, 0.



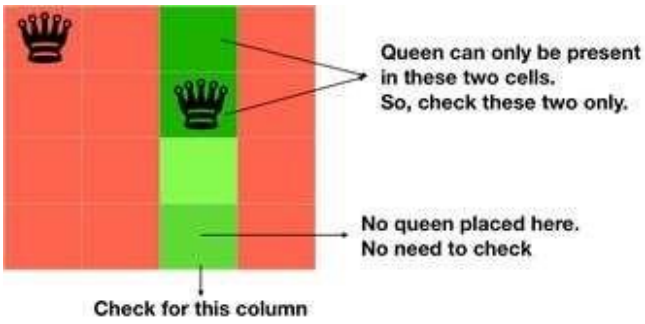
The cell (i,j) will be under attack in three condition - if there is any other queen in row i, if there is any other queen in the column j or if there is any queen in the diagonals.



We are already proceeding row-wise, so we know that all the rows above the current row(i) are filled but not the current row and thus, there is no need to check for row i.



We can check for the column j by changing k from 1 to i-1 in board[k][j] because only the rows from 1 to i-1 are filled.



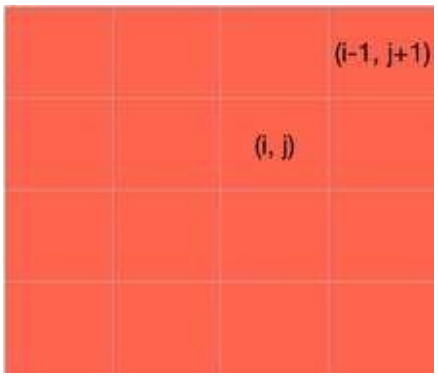
for k in 1 to i-1

if board[k][j]==1

return TRUE

Now, we need to check for the diagonal. We know that all the rows below the row i are empty, so we need to check only for the diagonal elements which above the row i.

If we are on the cell (i, j), then decreasing the value of i and increasing the value of j will make us traverse over the diagonal on the right side, above the row i.



k = i-1

l = j+1

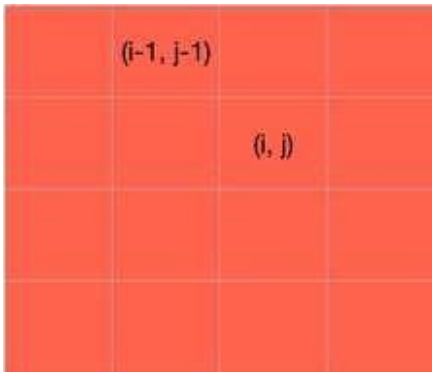
while k>=1 and l<=N

if board[k][l] == 1 return TRUE

k=k-1

l=l+1

Also if we reduce both the values of i and j of cell (i, j) by 1, we will traverse over the left diagonal, above the row i.




```
k = i-1
l = j-1
while k>=1 and l>=1
  if board[k][l] == 1
  return TRUE
  k=k-1
  l=l-1
```

At last, we will return false as it will be return true is not returned by the above statements and the cell (i,j) is safe.

We can write the entire code as:

```
  IS-ATTACK(i, j, board, N)
    // checking in the column j
    for k in 1 to i-1
      if board[k][j]==1
          return TRUE
            // checking upper right diagonal
    k = i-1
    l = j+1
    while k>=1 and l<=N
      if board[k][l] == 1
          return TRUE
      k=k+1
      l=l+1
```

```
// checking upper left diagonal
k = i-1
l = j-1
while k>=1 and l>=1
  if board[k][l] == 1
      return TRUE
  k=k-1
  l=l-1
return FALSE
```

Now, let's write the real code involving backtracking to solve the N Queen problem.

Our function will take the row, number of queens, size of the board and the board itself - N-QUEEN(row, n, N, board).

If the number of queens is 0, then we have already placed all the
queens. if n==0
  return TRUE

Otherwise, we will iterate over each cell of the board in the row passed to the function and for each cell, we will check if we can place the queen in that cell or not. We can't place the queen in a cell if it is under attack.

```
for j in 1 to N

  if !IS-ATTACK(row, j, board,

    N) board[row][j] = 1
```

After placing the queen in the cell, we will check if we are able to place the next queen with this arrangement or not. If not, then we will choose a different position for the current queen.

```
for j in 1 to N
    if N-QUEEN(row+1, n-1, N,
      board) return TRUE
    board[row][j] = 0
```

if N-QUEEN(row+1, n-1, N, board) - We are placing the rest of the queens with the current arrangement. Also, since all the rows up to 'row' are occupied, so we will start from 'row+1'. If this returns true, then we are successful in placing all the queen, if not, then we have to change the position of our current queen. So, we are leaving the current cell board[row][j] = 0 and then iteration will find another place for the queen and this is backtracking.

Take a note that we have already covered the base case - if n==0 → return TRUE. It means when all queens will be placed correctly, then N-QUEEN(row, 0, N, board) will be called and this will return true.

At last, if true is not returned, then we didn't find any way, so we will return
false. N-QUEEN(row, n, N, board)

```
 return FALSE
```

```
 N-QUEEN(row, n, N, board)
   if n==0
     return TRUE
   for j in 1 to N
     if !IS-ATTACK(row, j, board, N)
        board[row][j] = 1
        if N-QUEEN(row+1, n-1, N, board)
        return TRUE
        board[row][j] = 0 //backtracking, changing current decision
   return FALSE
```

Conclusion- In this way we have explored Concept of Backtracking method and solve n-Queen problem using backtracking method

Assignment Question

1. **What is backtracking? Give the general Procedure.**
2. **Give the problem statement of the n-queens problem. Explain the solution**
3. **Write an algorithm for N-queens problem using backtracking?**
4. **Why it is applicable to N=4 and N=8 only?**