Grp-A Experiment -3

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

Objective:- Students should be able to understand and solve factorial Knapsack problems using greedy method.

Prerequisite:-
1. Basic Python or Java.
2. Concept of Greedy method.
3. Factorial Knapsack problem.

Theory:-

What is greedy method?
- A greedy algo^m 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 over all optimal result.

- The algo^m never ~~reverses~~ reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

- This algo^m may not produce the best result for all the problems. It is because it always goes for the local best choice to produce the global best ~~re~~ result.

Advantages:-

- The algo^m is easier to describe.
- This algo^m can perform better than other algo^m.

Drawback:-

- As mentioned earlier, the greedy algo^m doesn't always produce the optimal sol^n. This is major drawback.
- Example:- Suppose we want to find the longest path in graph below from root to leaf.

Greedy Algorithm:-

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

Knapsack problem :-
You are given following :-

- A knapsack ~~problem~~ with weight capacity.
- Four 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 max^{m}.
- And the weight limit of the knapsack does not exceed.

Knapsack variants :-
1. Fractional knapsack problem.
2. 0/1 knapsack problem.

Fractional knapsack problem :-
- As 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 greedy method.

It follows following steps :-
1. For each item, compute value/weight ratio.

2. Arrang all item in decreasing order of their value/weight ratio.
3. Start putting items into knapsack beginning from the item with highest ratio. Put as ~~much~~ many as you can into knapsack.

Problem –
For given set of items and knapsack capacity = 60kg, find optimal soln for fractional knapsack problem ~~usin~~ making use of greedy approach.

item	wt	value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	90

n = 5
w = 60 kg.
(w1, w2, w3, w4, w5) = (5, 10, 15, 22, 25)
(b1, b2, b3, b4, b5) = (30, 40, 45, 77, 99).

Soln :-

Step :- Compute value/wt ratio.

Items	Wt	value	ratio
1	5	30	6
2	10	40	4
3	15	45	3
4	22	77	3.5
5	25	90	3.6

Step 2 :-
Sort all items in decreasing order of value/wt ratio :-

I1 I2 I5 I4 I3
(6) (4) (3.6) (3.5) (3)

Step 3 :-
Start filling Knapsack by putting the items into it one by one.

Knapsack wt.	Items in Knapsack	Cost
60	∅	0
55	I1	30
45	I1, I2	70
20	I1, I2, I3	160

Now,
- Knapsack wt left to be filled is 20 kg but item 4 has wt of 22 kg.
- Since in fractional problem, even the fraction of any item can be taken.

- So, knapsack will contain items:-
  $\langle I_1, I_2, I_5, (20/22) I_4 \rangle$

Total cost of knapsack
= 160 + (20/22) x 77
= 160 + 70
= 230 units.

Time complexity:-
- Total time taken including sort is O(n log n)

code in python.

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
    def fractionalKnapsack(W, arr):
        # Sorting on the basis of ratio.
        arr.sort(key = lambda x: (x.value/x.weight), reverse = True)
        # result(value in Knapsack)
        final value = 0.0
        # Looping through all items
        for item in arr:
            # if adding Item won't overflow,
            # add it completely
            if item.weight <= W:
                W -= item.weight
                final value += item.value
```

```
# if we can't add current Item,
# add fractional part of it
else:
    finalvalue += item.value * W / item.weight
    break
# returning final value
return finalvalue

# Driver code
if __name__ == "__main__":
    W = 50
    arr = [Item(60, 10), Item(100,20), Item(120, 30)]
    # Function call
    max_val = fractionalKnapsack(W, arr)
    print(max_val)

// Output
240.0
```

Grp A:- Experiment - 4

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

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

Prerequisite:-
1. Basic Python and Java.
2. Concept of Dynamic programming.
3. 0/1 Knapsack problem.

Theory:-

What is dynamic programming?

- DP is also used in optimization problems. Like divide-and-conquer method, DP solves problems by combining the sol^n of subproblems

- DP algo^m solves each subproblem 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 given problem can be solved using DP. These properties are overlapping subproblems and optimal substructure.

- Example, Binary search does not have overlapping sub-problem. whereas recursive problem of Fibonacci numbers have many overlapping sub-problems.

Steps of DP approach:-

- Characterize the structure of optimal sol^n
- Recursively define the value of optimal sol^n.
- Compute the value of optimal sol^n, typically in bottom up fashion.
- Construct an optimal ~~approach~~ sol^n from computed info.

Knapsack problem:-

- A knapsack with limited weight capacity.
- Few items each having some wt 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 knapsack is max^m.
- And the wt limit of knapsack does not exceed.

Knapsack problem variants:-

Knapsack problem has 2 variants:-

1. Fractional Knapsack problem.
2. 0/1 Knapsack problem.

In 0/1 problem,

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

Using Greedy method:-

Consider:-

- Knapsack wt. capacity = w
- No. of items each having some wt and value = n.

0/1 Knapsack problem is solved using DP in following steps:-

Step 1:-

- Draw a table say 'T' with (n+1) no. of rows and (w+1) no. of columns.
- Fill all 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 2:-
Start filling table row wise top to bottom from left to right. Use formula :-

$$T(i,j) = \max\{T(i-1,j), value\,i + T(i-1, j-wt\,i)\}$$

Here, $T(i,j)$ = max^m value of selected items if we can take items 1 to i and have wt. restriction of j

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

Step 3: -
- To identify the items that must be put into knapsack to obtain that max^m profit
- Consider the last column of table.
- Start scanning the entries from bottom to top.
- After all the entries are scanned, the marked labels represents the items that must be put into the knapsack.

Problem:-
For the given set of items and knapsack capacity = 5 kg, find optimal soln for the 0/1 knapsack problem making use of dp approach.

item	wt	value
1	2	3
2	3	4
3	4	5
4	5	6

$n = 4$
$w = 5kg$
$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$
$(b_1, b_2, b_3, b_4) = (3, 4, 5, 6)$

Soln:-
Given:-
- Knapsack capacity (w) = 5kg
- No. of items (n) = 4.

Step 1:-
- Draw a table 'T' with $(n+1) = 4+1 = 5$ no of rows and $(w+1) = 5+1 = 6$ no of columns.
- Fill all boxes of 0th row and 0th column with 0

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

Step 2:-
Start filling the table row wise.

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

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

$T(1,3)$:–
we have,
$i=1$
$j=1$
$(value)_i = (value)_1 = 3$
$(weight)_i = (weight)_1 = 2$

Substituting the values,
$T(1,3) = \max\{T(1-1,3), 3+T(1-1,3-2)\}$
$T(1,3) = \max\{T(0,3), 3+T(0,1)\}$
$T(1,3) = \max\{0, 3+0\}$
$T(1,3) = 3$.

$T(1,4)$:–
we have,
$i=1, j=4$
$(value)_i = (value)_1 = 3$
$(weight)_i = (weight)_1 = 2$.

Substituting the values :–
$T(1,4) = \max\{T(1-1,4), 3+T(1-1,4-2)\}$
$T(1,4) = \max\{T(0,4), 3+T(0,2)\}$
$T(1,4) = \max\{0, 3+0\}$
$T(1,4) = 3$.

T(2,1):-
we have,
$i=2, j=1$
$(value)_i = (value)_2 = 4$
$(wt)_i = (wt)_2 = 3$

Substituting the values:-
$T(2,1) = max\{T(2-1,1), 4+T(2-1,1-3)\}$
$T(2,1) = max\{T(1,1), 4+T(1,-2)\}$
$T(2,1) = T(1,1)$ { Ignore $T(1,-2)$ }
$T(2,1) = 0$.

T(2,3):-
we have,
$i=2, j=3$
$(value)_i = (value)_2 = 4$
$(wt)_i = (wt)_2 = 3$

Substituting the values,
$T(1,3) = max\{T(2-1,3), 4+T(2-1,3-3)\}$
$T(2,3) = max\{T(1,3), 4+T(1,0)\}$
$T(2,3) = max\{3, 4+0\}$
$T(2,3) = 4$.

Similarly, compute all the entries.
After all the entries are computed and filled in table, we get:-

	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)

Identifying items to be put into Knapsack:-
Following step 4:-
- We mark rows labelled "1" and "2".
- Thus, items that must be put into the Knapsack to obtain the max^m value 7 are:-

Item-1 and Item-2

Time complexity :-

- Each entry of table requires constant time $\theta(1)$ for its computation.
- It takes $\theta(nw)$ time to fill $(n+1)(w+1)$ table entries.
- It takes $\theta(n)$ time for tracing the soln since tracing process traces the n rows.
- Thus, overall $\theta(nw)$ 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 ~~programming~~ approach.

Grp A Experiment - 5

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

Objective :- Students should be able to understand and solve n-Queen problem, and understand basics of Backtracking.

Theory :-

Introduction :-

- 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 info to know what to choose,
- Each decision leads to a new set of choices.
- Some sequence of choices may be a sol^n to your problem.

What is Back tracking :-

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, ~~to~~ 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 subsolution of problem and then check if we can proceed further with this sub-soln or not. If not, then we just come back and change it.

Applications:-
- N Queen Problem.
- Sum of Subsets problem.
- Graph coloring.
- Hamiltonian cycles.

N-Queens Problem:-
A classic combinational problem is the classical Example of backtracking. N-Queen problem is defined as, "given NXN chessboard, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal".

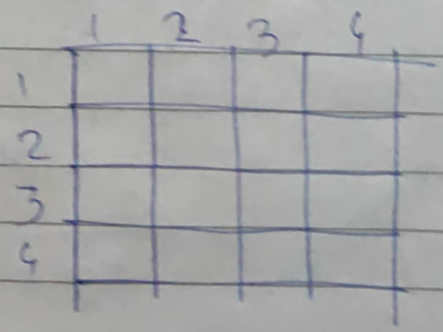
For N=1, this is a trivial case. For N=2 and N=3, a soln is not possible. So we start with N=4 and we will generalize it for N queens.

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

Algorithm :-
1. Start in leftmost col^m.
2. If all queens are place return true.
3. Try all rows in current col^m.

In following for every tried row:-
a. If the queen can be placed safely in this row then mark this [row, col^{m}] as part of the sol^{n} and recursively check if placing queen here leads to a sol^{n}.
b. If placing the queen [row, col^{m}] leads to sol^{n} then return true.
c. If placing queen doesn't lead to a sol^{n} then unmark this [row, col^{m}] (Backtrack) and go to step(a) to try other rows
d. If all rows have been tried and nothing worked, return false to trigger backtracking.

4-Queen Problem:-
Problem 1:- Given 4x4 chessboard, arrange 4 queens in a way, such that no two queens attack each other. That is, no two queens are placed in same row, col^{m}, or diagonal.



We have to arrange 4 queens, Q_1, Q_2, Q_3, Q_4 in 4x4 chess board. We will put with queen in 1th row. Let us start with position(1, 1). Q_1 is only queen, so there is no issue. partial sol^{n} is <1>.

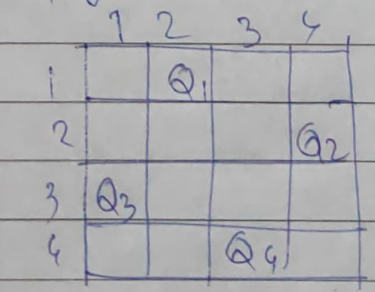
We cannot place Q_2 at positions (2,1) or (2,2). Position (2,3) is acceptable. the partial sol^n is $\langle 1,3 \rangle$.

Now Q_3 can be placed at position (3,2). Partial sol^n is $\langle 1,4,3 \rangle$.

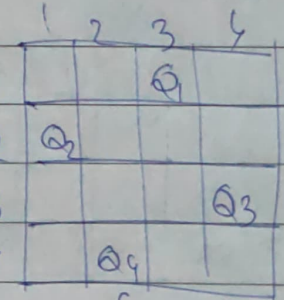
Q_4 cannot be placed anywhere in row 4. So again, backtrack to the previous sol^n and readjust the position of Q_3. Q_3 cannot be placed on (3,3) or (3,4). So algo^m backtracks even further.

All possible choices for Q_2 are already explored, hence the algo^m goes back to partial sol^n $\langle 1 \rangle$ and moves the queen Q_1 from (1,1) to (1,2). And this process continues until a sol^n is found.

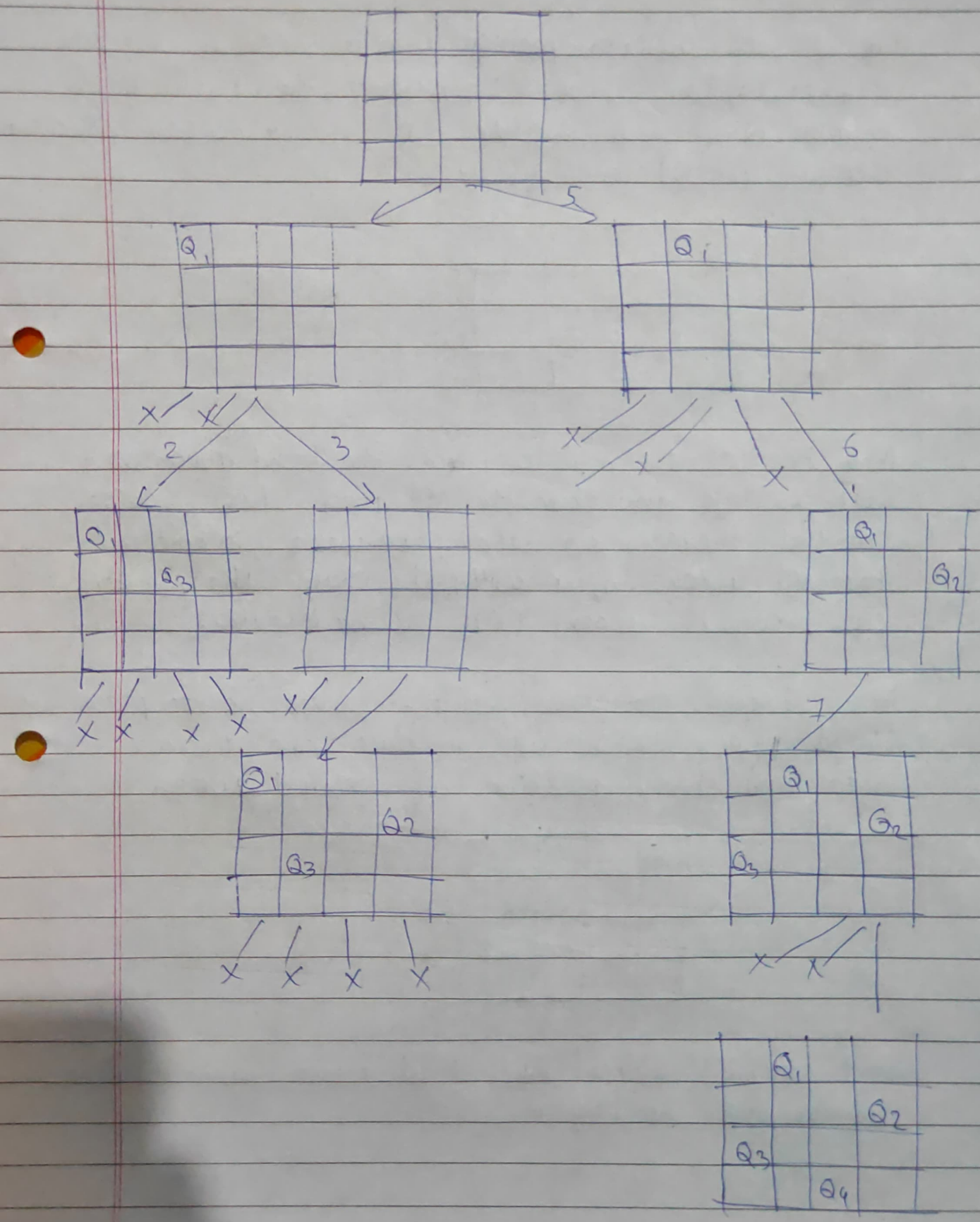
All possible sol^n for 4 queen are shown in fig(a) and (b)



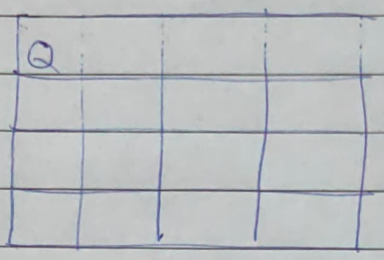
fig(a)



fig(b).

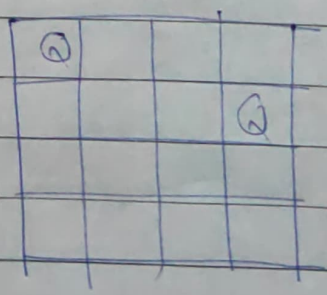

Q1
Q2
Q3
Q4
2
3
5
6
7

The soln of 4 Queen problem can be seen as 4 tuples (x_1, x_2, x_3, x_4), where x_i represents the Col^m no. of queen Q_i. Two possible soln for 4 Queen problem are (2,4,1,3) and (3,1,4,2).

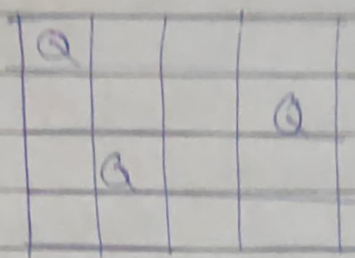


Now, the second step is to place the second queen in a safe position and then the 3rd queen. Now, you can see that there is no safe place where we can put the last queen. So, we will just chang the position of previous queen. This is backtracking.

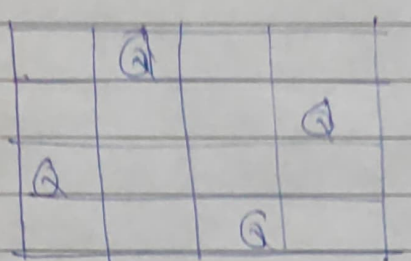
Also, there is no other position where we can place third queen so we will go back one more step and chang position of second queen.



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

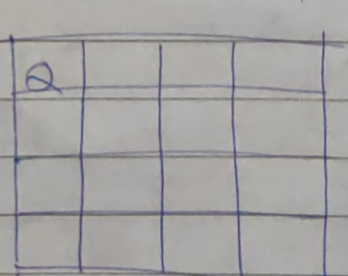


We will continue this process and finally we get the sol^n as below.



We need to check if a cell (i, j) is under attack or not. For that, we will pass these 2 in our fu^n along with the chessboard and its size - IS-ATTACK (i, j, board, N).

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



1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

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

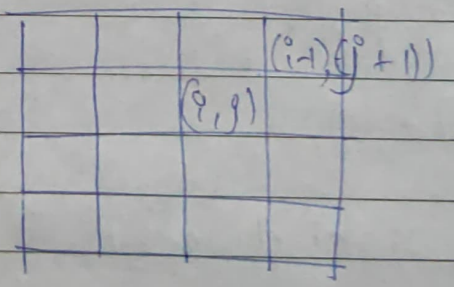
We can check for column j by changing K from 1 to i-1 in board[K][j] because only the rows from 1 to i-1 are filled.

for K in 1 to i-1.

if board[K][j]==1 return TRUE

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

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

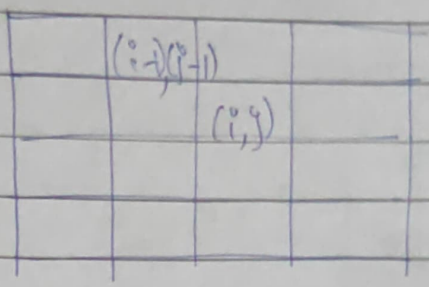


K = i-1 l = j+1

while K>=1 and l<=N if board[K][l]==1

return TRUE

K = K-1 l = l+1



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 return true is not returned by above statements and call(i,j) is safe.

We can write code as:-
IS-ATTACK(i,j, board,N)
// checking in colm j
for k in 1 to i-1
if board[K][j]==1.
return TRUE
// checking upper right diagonal.
K=i-1
I=j+1
while K>=1 an I<=N
if board[K][I]==1
return TRUE
K=K+1
I=I+1

//checking upperleft diagonal.
K = i - 1
J = j - 1
while K >= 1 and J >= 1
if board[K][J] == 1
return TRUE
K = K - 1
J = J - 1.
return FALSE.

Now, let's write real code involving backtracking to solve the N Queen problem. Our fun will take the row, no. of queens, size of board and the board itself
N - Queen(row, n, N, board).

If no. of queens is 0, then we have already placed all queens. If n == 0 return TRUE.

Otherwise, we will iterate over each cell of the board in row passed the fun 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 I to N.
if ! IS-ATTACK(row, j, board, N) board[row][j] = 1.
After placing the queen in cell, we will check if we are able to place the next queen with the 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.
```

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.
return FALSE.
```

// code :-

```
global N
N = 4

def printsol(board):
    for i in range(N):
        for j in range(N):
            print(board[i][j], end=" ")
        print()

def issafe(board, row, col):
    for i in range(col):
        if board[row][i] == 1:
            return False
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    for i, j in zip(range(row, N, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False
    return TRUE

def solNQUtil(board, col):
    if col >= N:
        return True
    for i in range(N):
        if issafe(board, i, col):
```

```
        board[i][col] = 1
        if SolveNQUtil(board, col+1) == TRUE:
            return True
        board[i][col] = 0
    return False

def SolveNQ():
    board = [ [0,0,0,0],
              [0,0,0,0],
              [0,0,0,0],
              [0,0,0,0] ]
    if SolveNQUtil(board,0) == False:
        print("Soln does not exist")
        return False
    printSol(board)
    return True

# Driver code
SolveNQ()
```

//Output:-

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

Conclusion:- In this way we have explored concept of backtracking method and solve n-Queen problem using backtracking method.