MET Bhujbal Knowledge City
Institute Of Engineering
Adgaon, Nashik - 422 003. Date :

Assignment No. 4

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

Objective of the Assignment:
Students should be able to understand & 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:
- Students 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 problem.
- 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 problem by combining the solutions of subproblems.

- Dynamic programming algorithm solves each sub-problem once & 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 & 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 ex, 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 4 steps -

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

Appl^n of Dynamic programming Approach :

- Matrix chain Multiplication
- Longest common sequence
- Travelling Salesman problem
- Knapsack problem.

You are given the following knapsack problem -

- A knapsack with limited weight capacity.
- few items each having some weight & value.

 Date :

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 take either an item completely or leave it completely.
- It is solved using a dynamic programming approach.

0/1 Knapsack Problem using Greedy Method:

- As the name suggests,

Consider -

- Knapsack weight capacity = w
- Number of items each having some weight & 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 & (w+1) number of columns.
- Fill all the boxes of 0^{th} rows & 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 ~~problem~~ formula -

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

Here, $T(i, j)$ = maximum value of the selected items if we can take items 1 to i & 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.

 Date :

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 & knapsack capacity = 5kg, find the optimal sol^n for the 0|1 knapsack problem making use of a dynamic programming approach.

Item	weight	value	
1	2	3	n = 4
2	3	4	w = 5kg
3	4	5	$(w_1, w_2, w_3, w_4) = (2, 3, 4, 5)$
4	5	6	$(b_1, b_2, b_3, b_4) = (3, 4, 5, 6)$

Solution :

Given : Knapsack capacity (w) = 5kg &

Number of items (n) = 4

Step 01 :

- Draw a table say "T" with (n+1) = 4+1 = 5 number of rows & (w+1) = 5+1 = 6 no. of columns.

• Fill all the boxes of 0th row & 0th column with 0

	0	1	2	3	4	5
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 -

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

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 $(0,-1)$}

$T(1,1) = 0$

finding T(1,2) -

we have

$i = 1$

$j = 2$

$(value)_i = (value)_1 = 3$

$(weight)_i = (weight)_1 = 2$

Substituting the value, 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

(weigW)$_i$ = (weigW)$_1$ = 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)$_1$~~

(weight)$_i$ = (weigW)$_1$ = 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

(weigW)$_i$ = (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)$_i$ = (weigW)$_2$ = 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 & 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" & "2".
- Thus, Items that must be put into the knapsack to obtain the maximum value V are - Item-1 & Item-2

Time Complexity -

- Each entry of the 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 approach.

Assignment No. 5

Title of the Assignment :

Design n-Queens matrix having first Queen placed. Using 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 back tracking 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 may be a solution to your problem.

What is backtracking?

Backtracking is finding the sol^n of a problem whereby the sol^n depends on the previous steps taken. For ex, 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 sol^n. In a maze problem, we first choose a path & continue moving along it. But once we understand that the particular path is incorrect then we just come back & change it.

In backtracking, we first take a step & then we see if this step taken is correct or not ie., whether it will give a correct ans or not. And if it doesn't then we just come back & 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 & then check if we can proceed further with this sub-sol^n or not. If not, then we just come back & change it.

Thus, the general steps of backtracking are:

- Start with a sub-solution
- Check if this sub-sol^n will lead to the sol^n or not.
- If not, then come back & change the sub-sol^n & continue again.

Applications of Backtracking:

- N Queen problem
- Sum of subsets problem
- Graph coloring
- Hamiltonin cycles.

N queens on N x N 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 sol^n of this problem is also attempted in a similar way. We first place the first queen anywhere arbitrarily & then place the next queen in any of the safe places. We continue this process until the number of unplaced queens becomes zero or no safe place is left. If no safe places is left, then we change the position of the previously placed queen.

N-Queen Problem:

A classic combinational problem is to place n queens on a n*n chess board so that no two attack ie. no two queens are on the same raw, column or diagonal.

What is the N-Queen Problem ?

N-Queen problem is the classical problem of backtracking. N-Queen problem is defined as, "given NxN chess board, arrange N-Queens in such way that no two queens attack each other by being in the same raw, coloumn, or diagonal."

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

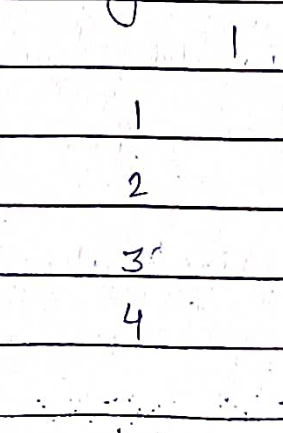
If we take n=4 then the problem is called the 4 queens Problem. If we take n=8 then the problem is called the 8 queen problem.

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 & recursively check if placing queen here leads to a sol^n.
b) If placing the queen in [row, column] leads to solution then return true
c) If placing queen doesn't leads to a solution then unmark this [row, column] (Backtrack) & go to step(a) to try other rows.
4) If all rows have been tried & nothing worked, return false to trigger backtracking.

4-Queen Problem:

Given 4x4 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.



4x4 Chessboard

THE MET LEAGUE OF COLLEGES
MET Bhujbal Knowledge City
AS SHARP AS YOU CAN GET
Institute Of Engineering
Adgaon, Nashik - 422 003.

Date :

- We have to arrange four queens, Q_1, Q_2, Q_3 & Q_4 in 4x4 chess board. we will put the queen in ith row. Let us start with position (1,1). Q_1 is the only queen, so there is no issue. Partial solution is <1>.
- We cannot place Q_2 at positions (2,1) or (2,2). Position (2,3) is acceptable. the partial sol^n is <1, 3>.
- Next, Q_3 cannot be placed in position (3,1) as Q_1 attacks her. And it cannot be placed at (3,2) (3,3) or (3,4) as Q_2 attacks her. There is no way to put Q_3 in the third row. Hence, the algorithm backtracks & goes back to previous sol^n & readjusts the position of queen Q_2 Q_2 is moved from position (2,3) to (2,4). Partial solution is <1, 4>
- Now, Q_3 can be placed at position (3,2) Partial solution is <1, 4, 3>
- Queen Q_4 cannot be placed anywhere in row four so again, backtrack to the previous sol^n & readjust the position of Q_3. Q_3 cannot be placed on (3,3) or (3,4). So the algorithm backtracks even further.
- All possible choices for Q_2 are already explored, hence the algorithm goes back to a partial solution <1> & moves the queen Q_1 from (1,1) to (1,2). And this process continues until a sol^n found.

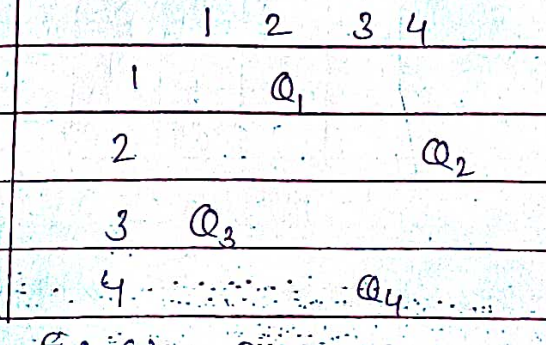
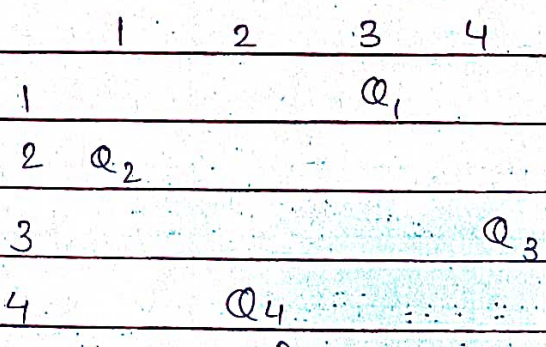
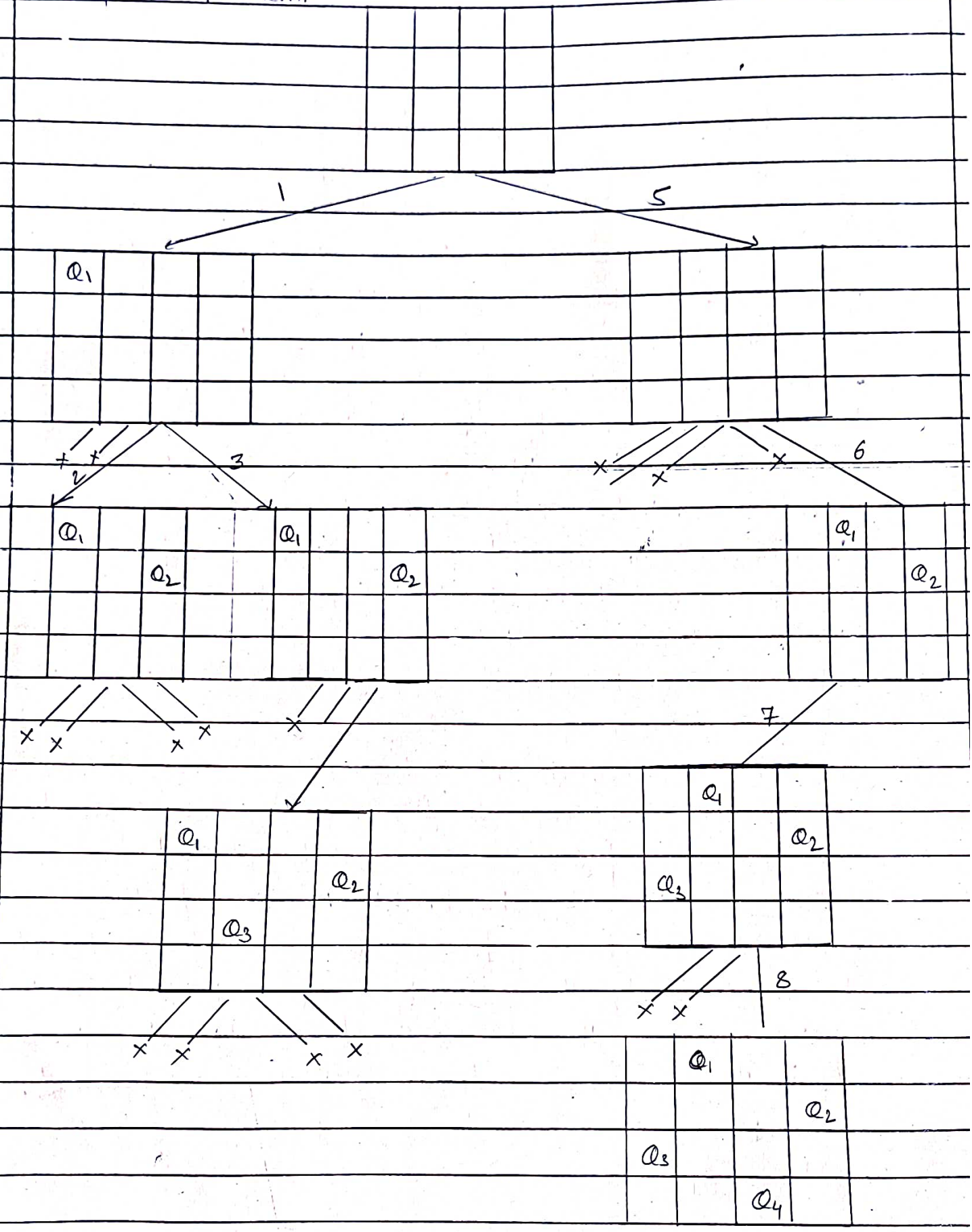


Fig. (a): sol^n -1



Fig(b): sol^n -2

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

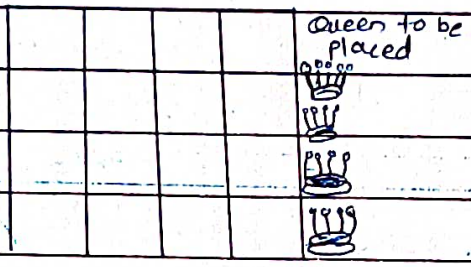


Institute Of Engineering
Adgaon, Nashik - 422 003.

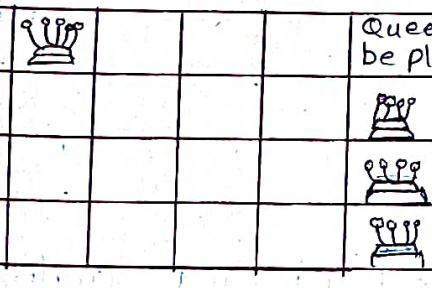
Date :

The soln for 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 soln for the 4-queen problems are (2,4,1,3) & (3,1,4,2)

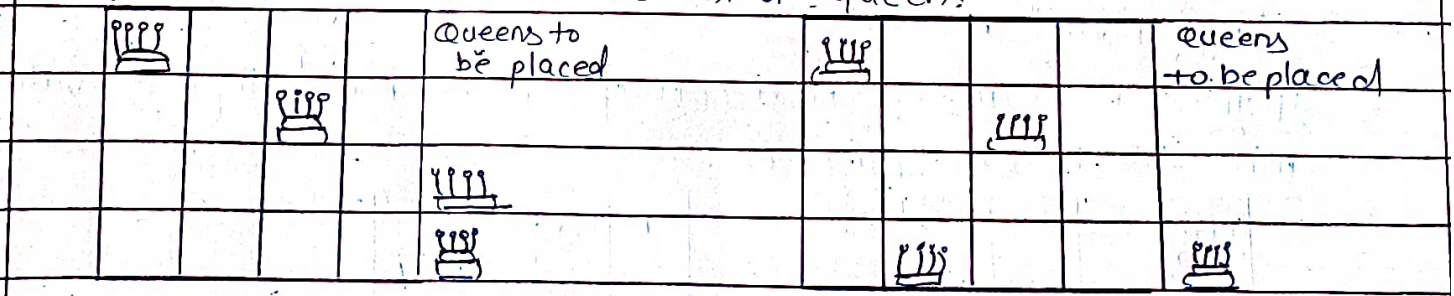
Explanation



The above pictures shows an NXN chessboard & we have to place N queens on it. So, we will start by placing the first queen.

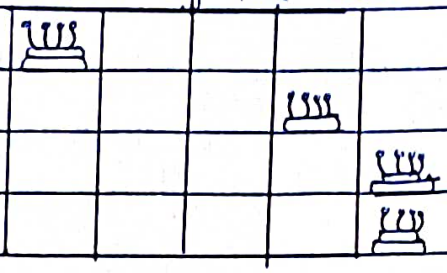


Now, the second step is to place the second queen in a safe position & then the third queen.

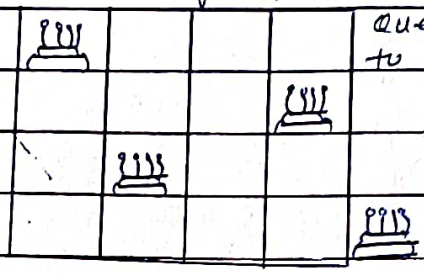


Now, you can see that there is no safe place where we can put the last queen, so we will just change the position

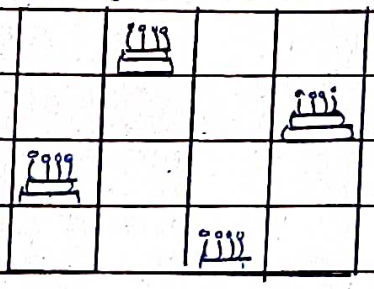
position of the previous queen. And this is backtracking Also, there is no other position where we can place the third queen so we will go back one more step & change the position of the second queen.



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

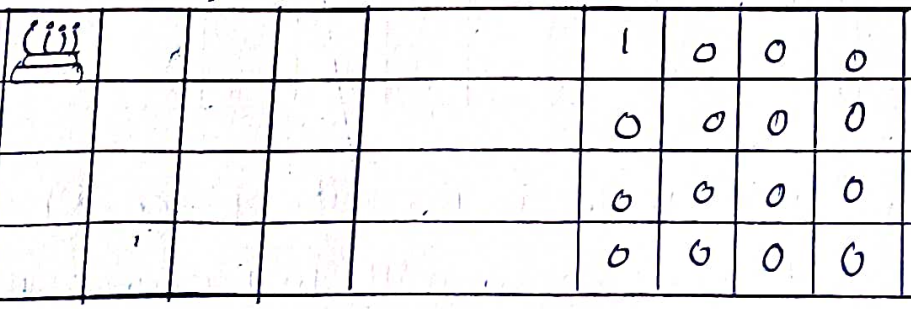


we will continue this process & finally, we will get the solution as shown below.



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 & its size. IS-ATTACK (i, j, board, N).

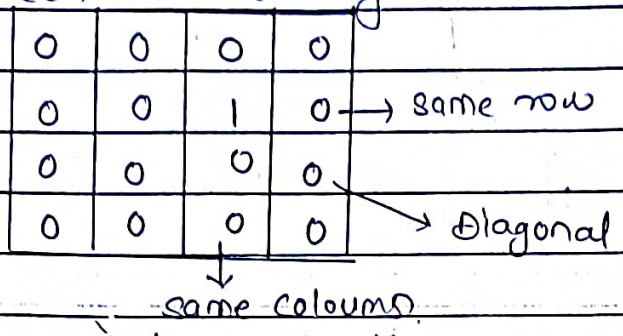
If there is a queen in a cell of the chessboard, then its value will be 1, otherwise 0.



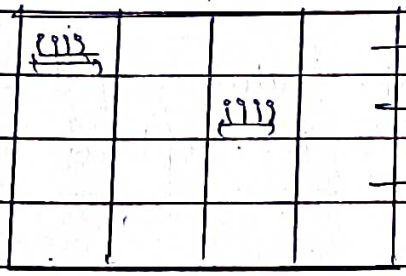
THE MET LEAGUE OF COLLEGES
MET Bhujbal Knowledge City
AS SHARP AS YOU CAN GET
Institute Of Engineering
Adgaon, Nashik - 422 003.

Date :

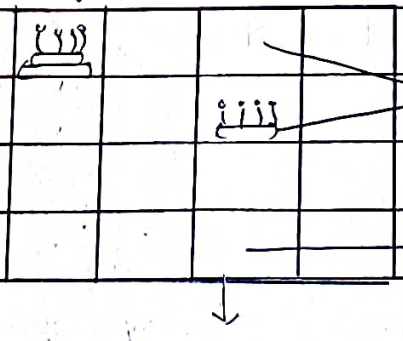
The cell (i,j) will be under attack in three condition - if there is any other queen in row i, if there is any other queen in the coloumn j or if there is any queen in the diagonals.



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 & thus, there is no need to check for row i.



We can check for the ~~qu~~ 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.

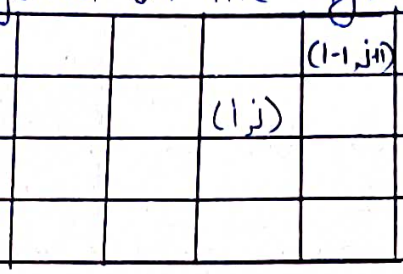



```
For k in 1 to i-1
if board[k][j] == 1
```

return TRUE

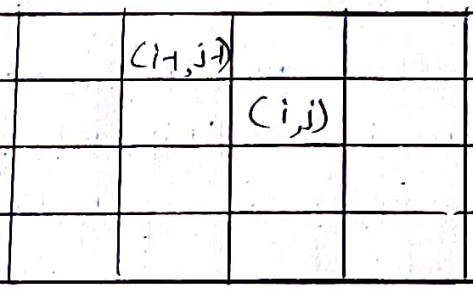
Now, we need to check for the diagonal. We know that all the rows below the row i are empty, so we need to check only for the diagonal elements which above the row i.

If we are on the cell (i,i), then decreasing the value of i & 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 & l <= N
if board [k][l] == 1 return TRUE
k = k - 1
l = l + 1
```

Also if we reduce both the values of i & 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 & l >= 1
    if board [k][l] == 1
    return TRUE
    k = k - 1
    l = l - 1
```

THE MET LEAGUE OF COLLEGES
MET Bhujbal Knowledge City
AS SHARP AS YOU CAN GET
Institute Of Engineering,
Adgaon, Nashik - 422 003.

Date :

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

We can write 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 & l<=N
    if board [k][l] == 1
      return TRUE
    k = k+1
    l = l+1
  k = i-1 .....
  l = j-1
  while k>=1 & 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, no of queens, size of the board & board itself. - N-queen (row, n, N, board)

If the no 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 & 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 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+, 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 will start from 'row+1'. If this returns true, then we are successful in placing all the queens, if not, then we have to change the position of our current queen so, we are leaving the current cell board [row] [j] = 0 & then iteration will find another place for the queen & 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 & this will return true

At last, if true is not returned, then we didn't find any way, so 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 we have explored Concept of Backtracking method & solve n-queen problem using backtracking method.