Parul® University

NAAC A++

Backtracking and Branch & Bound: Chapter-6

Mrs. Bhumi Shah
Assistant Professor
Department of Computer Science and Engineering

Parul® University

NAAC A++

Content

INDEX

Introduction to Backtracking

- Backtracking can be defined as a general algorithmic technique that considers **searching every possible combination** in order to solve an optimization problem.
- It is a recursive technique.
- It generates a state space tree for all possible solutions.
- It traverse the state space tree in the depth first order.
- So, in a backtracking we attempt solving a sub-problem, and if we don't reach the desired solution, then undo whatever we did for solving that sub-problem, and try solving another sub-problem.
- All the solutions require a set of constraints divided into two categories: explicit and implicit constraints.

Introduction to Branch & Bound

- The branch & bound approach is based on the principle that the total set of feasible solutions can be partitioned into smaller subsets of solutions.
- These smaller subsets can then be evaluated systematically until the best solution is found.
- Branch & bound is an algorithm design approach which is generally used for solving combinatorial optimization problems.
- These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.
- The Branch & Bound Algorithm technique solves these problems relatively quickly.

0/1 Knapsack Problem – Introduction

- Let us consider the 0/1 Knapsack problem to understand Branch & Bound.
- The Backtracking Solution can be optimized if we know a bound on best possible solution subtree rooted with every node.
- If the best in subtree is worse than current best, we can simply ignore this node and its subtrees.
- So, we compute bound (the best solution) for every node and compare the bound with current best solution before exploring the node.
- We are given a certain number of **objects** and a **knapsack.**
- Instead of supposing that we have n objects available, we shall suppose that we have **n types of object**, and that an adequate number of objects of each type are available.
- Our aim is to fill the knapsack in a way that **maximizes the value** of the included objects.
- We may take an object or leave behind, but we **may not take fraction** of an object.

0/1 Knapsack Problem using Branch & Bound

Input:
Weights: 1, 2, 3, 4
Profits: 10, 20, 25, 70
Maximum Weight Capacity: 7

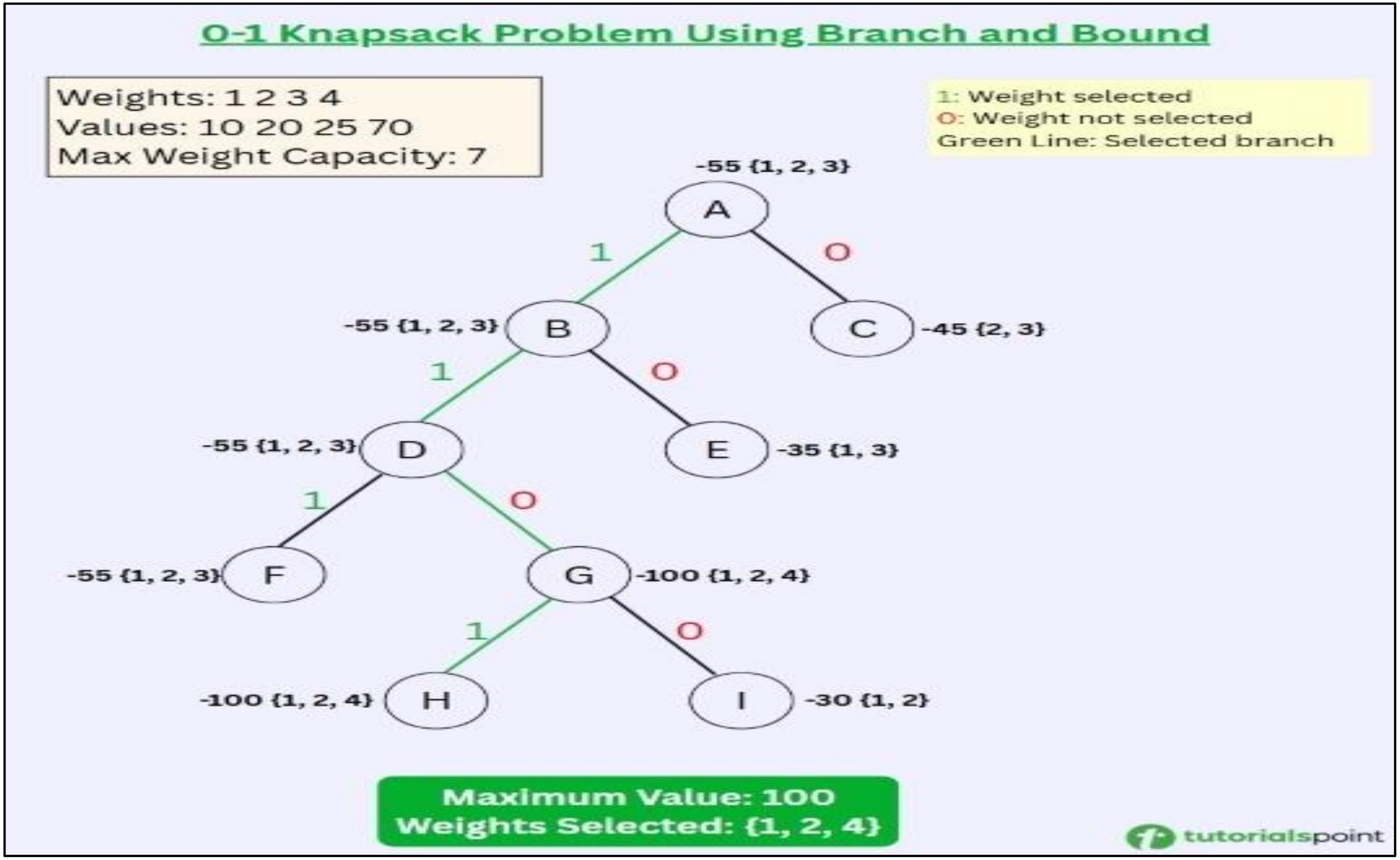
Output:
Maximum Profit = 100

0/1 Knapsack Problem using Branch & Bound

Input:
Weights: 1, 2, 3, 4
Profits: 10, 20, 25, 70
Maximum Weight Capacity: 7

Output:
Maximum Profit = 100

0/1 Knapsack Problem using Branch & Bound



0/1 Knapsack Algorithm

function backpack(i, r)

{Calculates the value of the best load that can be constructed using items of type i to n and whose total weight does not exceed r}

```
b ← 0
{Try each allowed kind of item in turn}
for k ← i to n do
    if w[k] ≤ r then
    b ← max(b, v[k] + backpack (k, r – w[k]))
return b
```

Parul® University

NAAC GRADE A++

https://paruluniversity.ac.in/

Parul® University

NAAC A++

Backtracking and Branch & Bound: Chapter-6

Mrs. Bhumi Shah

Assistant Professor

Department of Computer Science and Engineering

Content

1. Introduction to Backtracking, Introduction to Branch & Bound, 0/1 Knapsack Problem
2. N-Queens Problem, Travelling Salesman Problem

N-Queens Problem

- The N - queen is the problem of placing N chess queens on an $N \times N$ chessboard so that, no two queens attack each other.
- Two queens of same row, same column or the same diagonal can attack each other.
- K-Promising solution: A solution is called k-promising if it arranges the k - queens in such a way that, they can not threat each other.

Q	

1 - Promising Solution

Q	
Q	

0 - Promising Solution

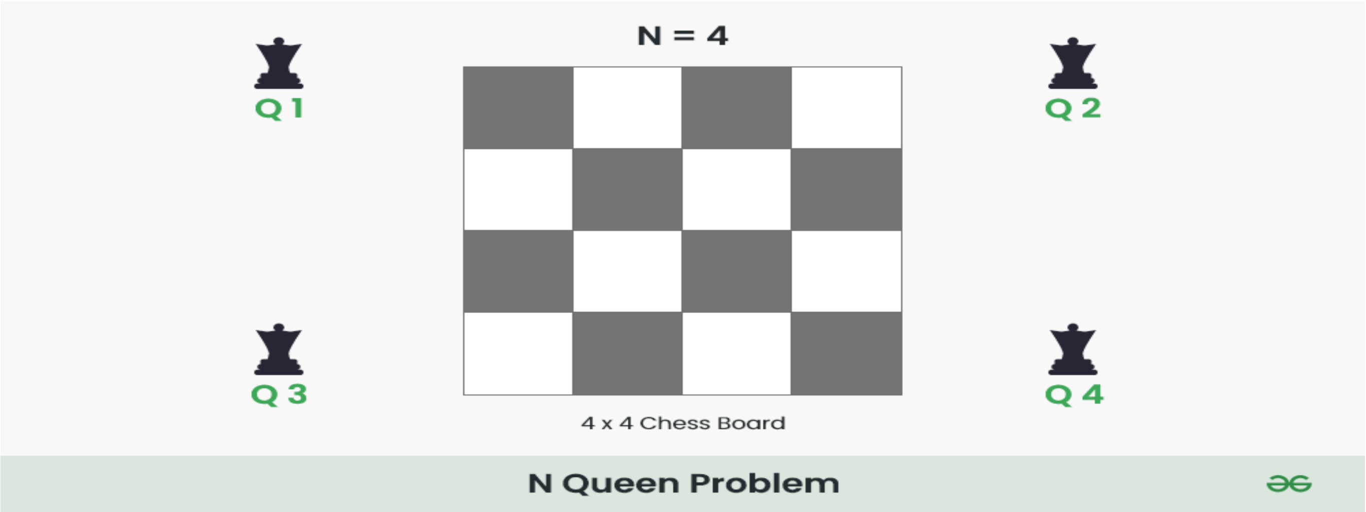
Q	
	Q

0 - Promising Solution

0 - Promising Solution

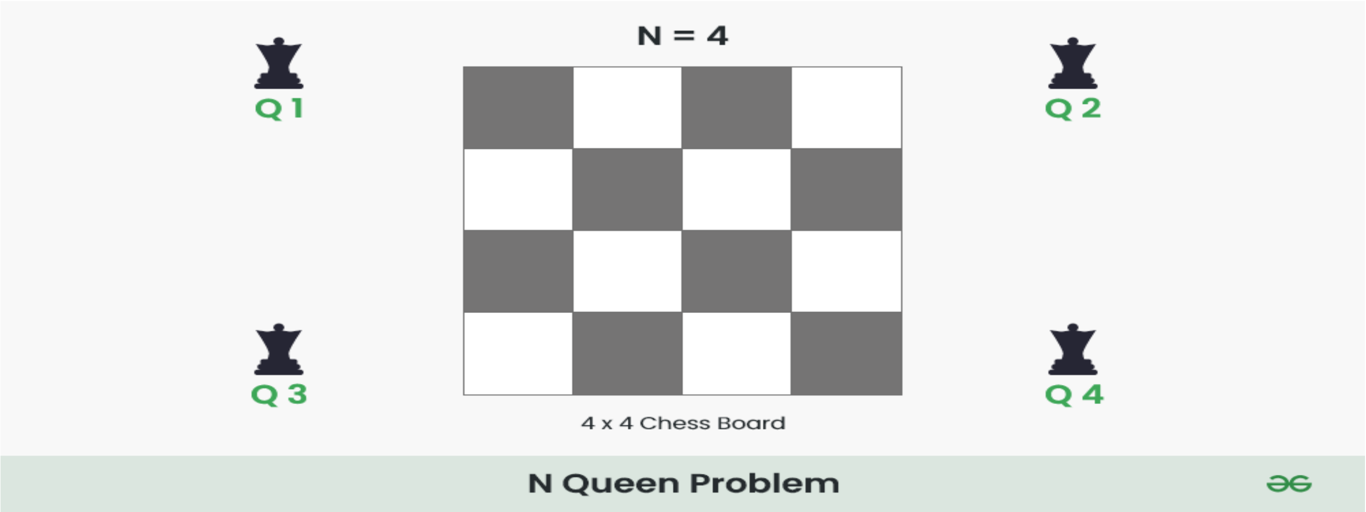
4-Queens Problem- Example

- The **4 Queens** Problem consists in placing four queens on a **4 x 4** chessboard so that no two queens attack each other. That is, no two queens are allowed to be placed on the **same row**, the **same column** or the **same diagonal**.
- We are going to look for the solution for n=4 on a 4 x 4 chessboard in this article.



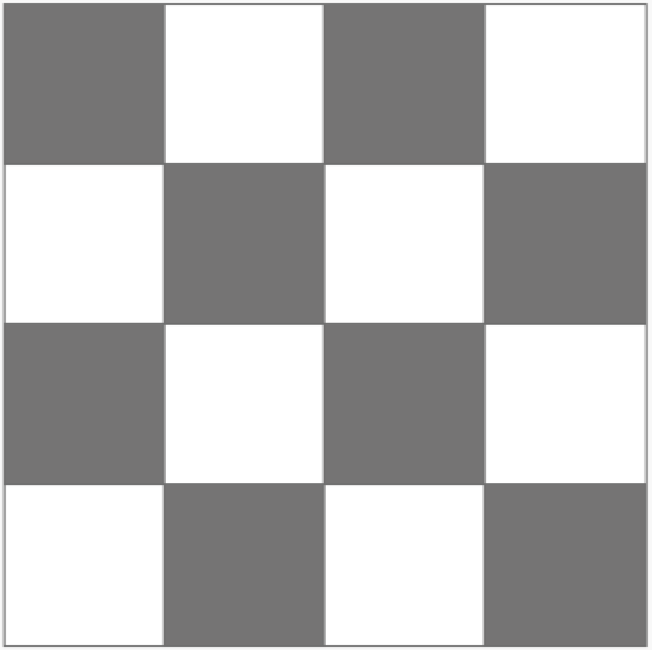
4-Queens Problem- Example

- The **4 Queens** Problem consists in placing four queens on a **4 x 4** chessboard so that no two queens attack each other. That is, no two queens are allowed to be placed on the **same row**, the **same column** or the **same diagonal**.
- We are going to look for the solution for n=4 on a 4 x 4 chessboard in this article.



4-Queens Problem- Example

Step 0: Initialize a 4×4 board.

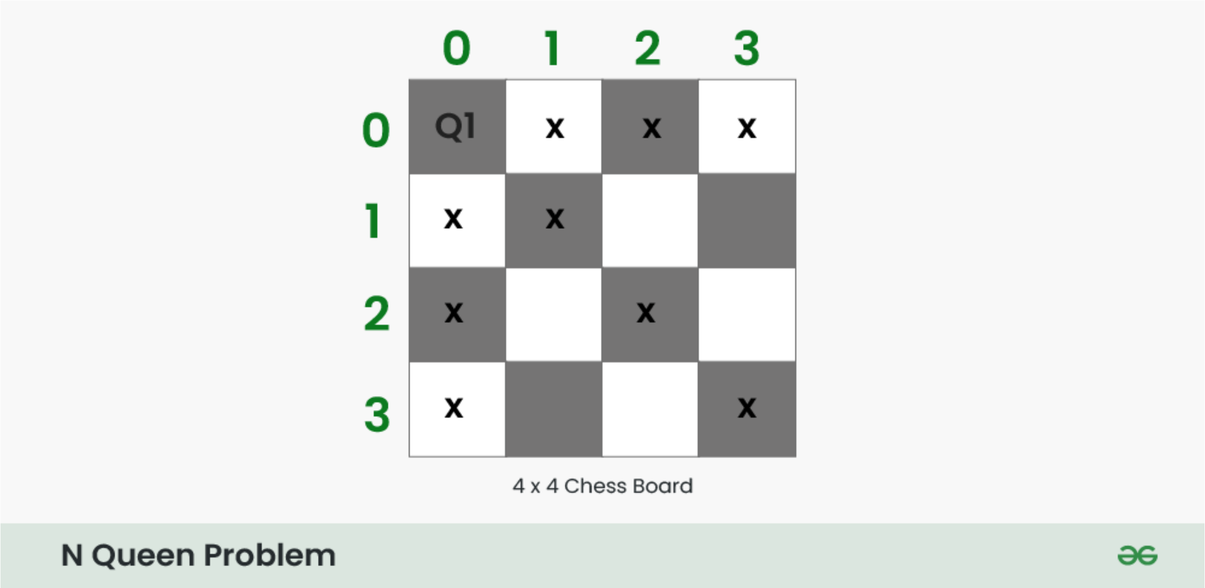


4 x 4 Chess Board

4-Queens Problem- Example

Step 1:

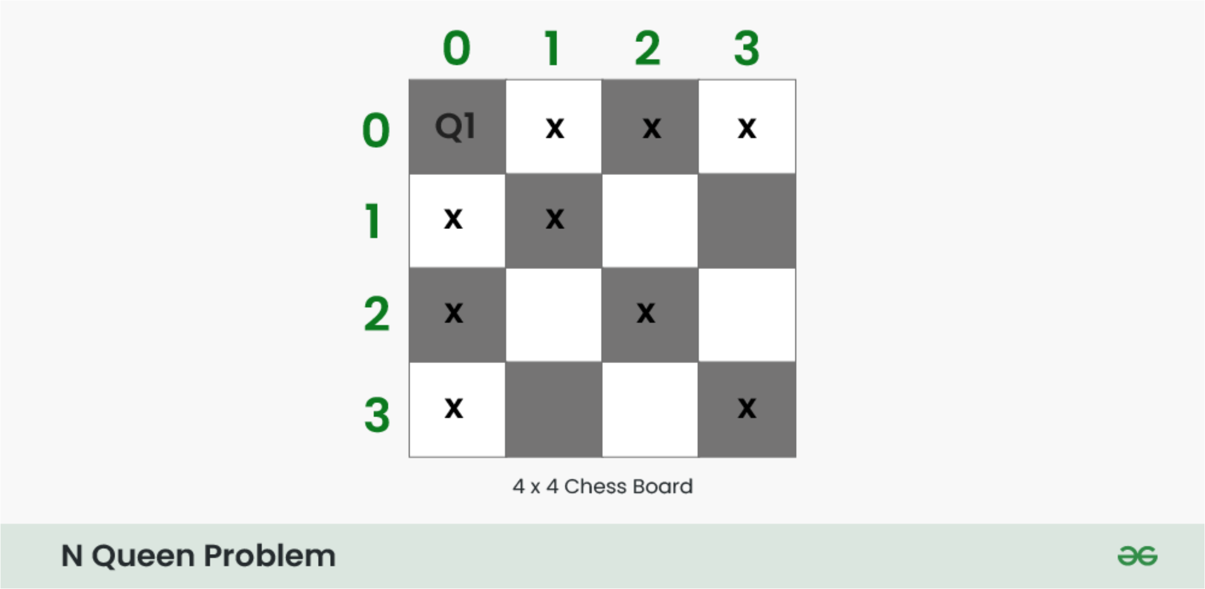
- Put our first Queen (Q1) in the (0,0) cell .
- ‘x‘ represents the cells which is not safe i.e. they are under attack by the Queen (Q1).
- After this move to the next row [0 -> 1].



4-Queens Problem- Example

Step 1:

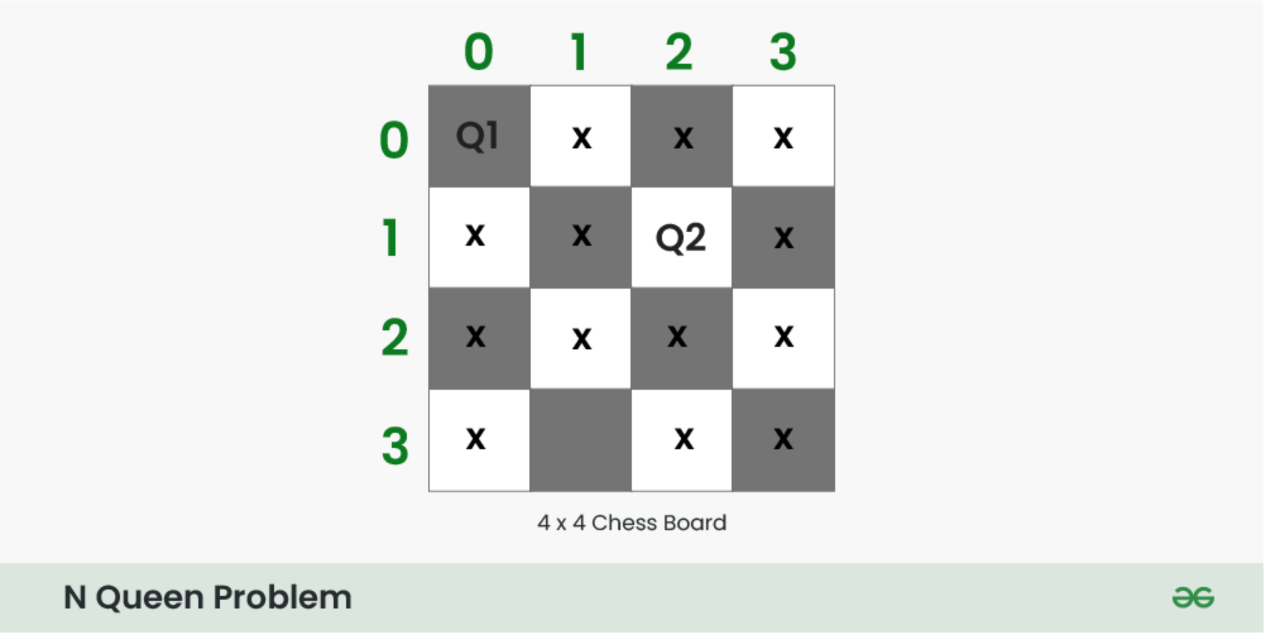
- Put our first Queen (Q1) in the (0,0) cell .
- ‘x‘ represents the cells which is not safe i.e. they are under attack by the Queen (Q1).
- After this move to the next row [0 -> 1].



4-Queens Problem- Example

Step 2:

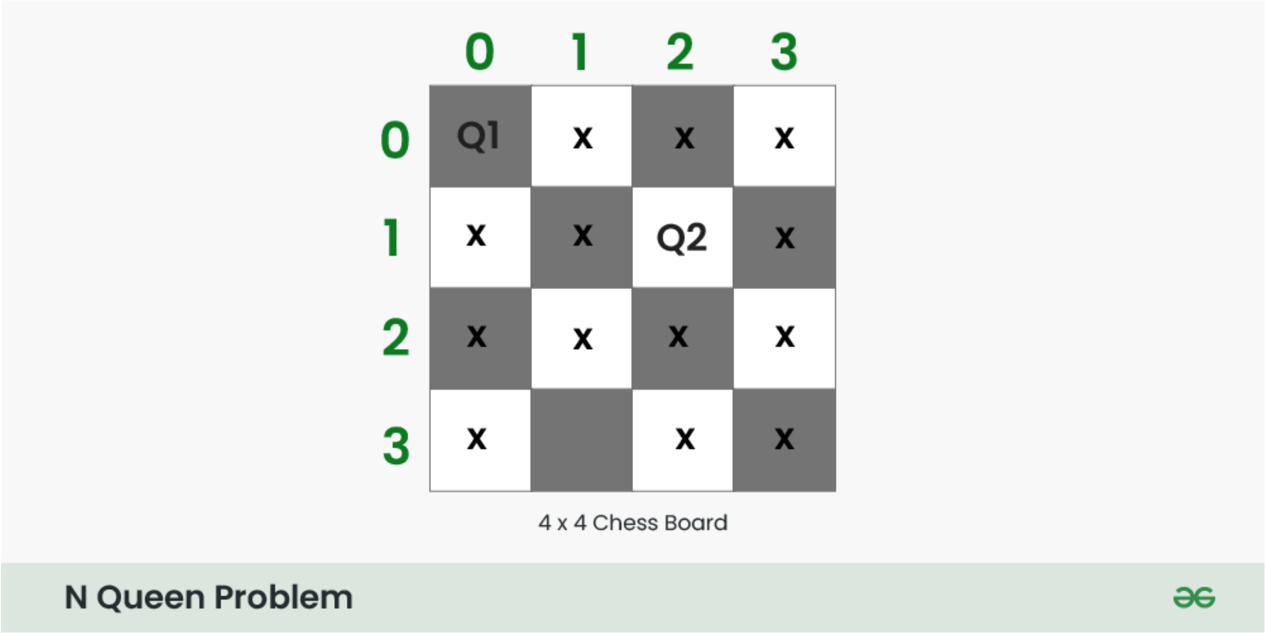
- Put our next Queen (**Q2**) in the **(1,2)** cell .
- After this move to the next row [1 -> 2].



4-Queens Problem- Example

Step 2:

- Put our next Queen (**Q2**) in the **(1,2)** cell .
- After this move to the next row [1 -> 2].



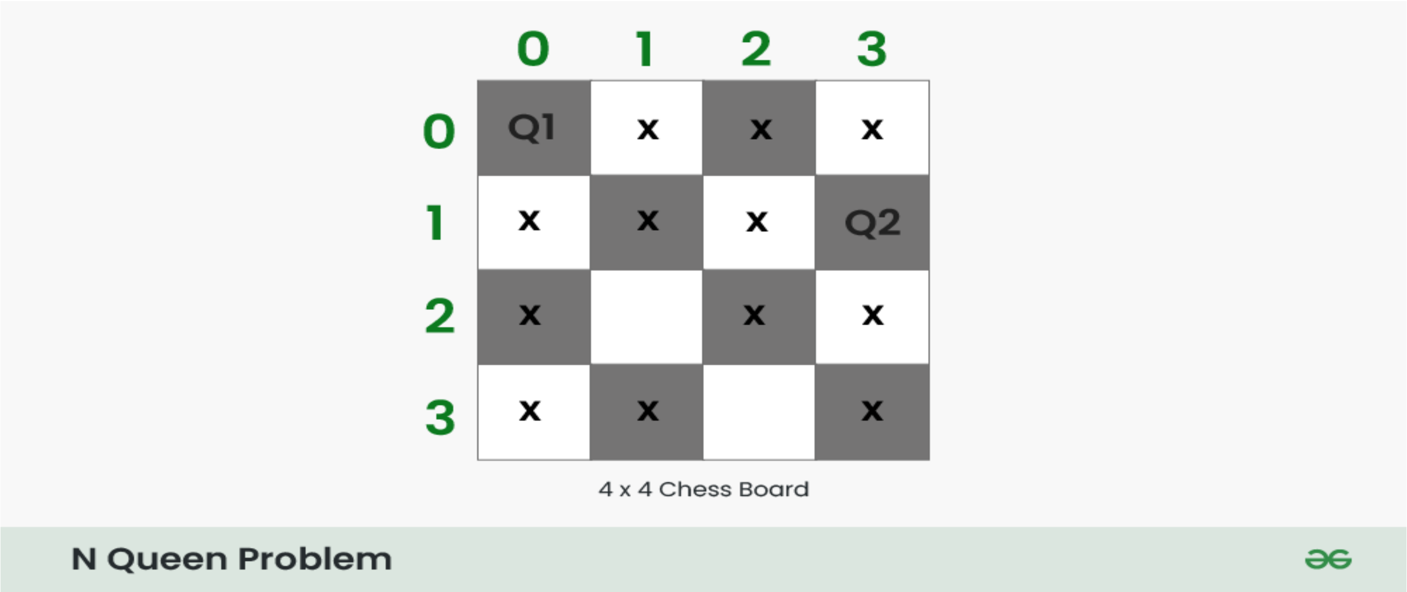
4-Queens Problem- Example

Step 3:

At row 2 there is no cell which are safe to place Queen (Q3) . So, backtrack and remove queen Q2 queen from cell (1, 2) .

Step 4:

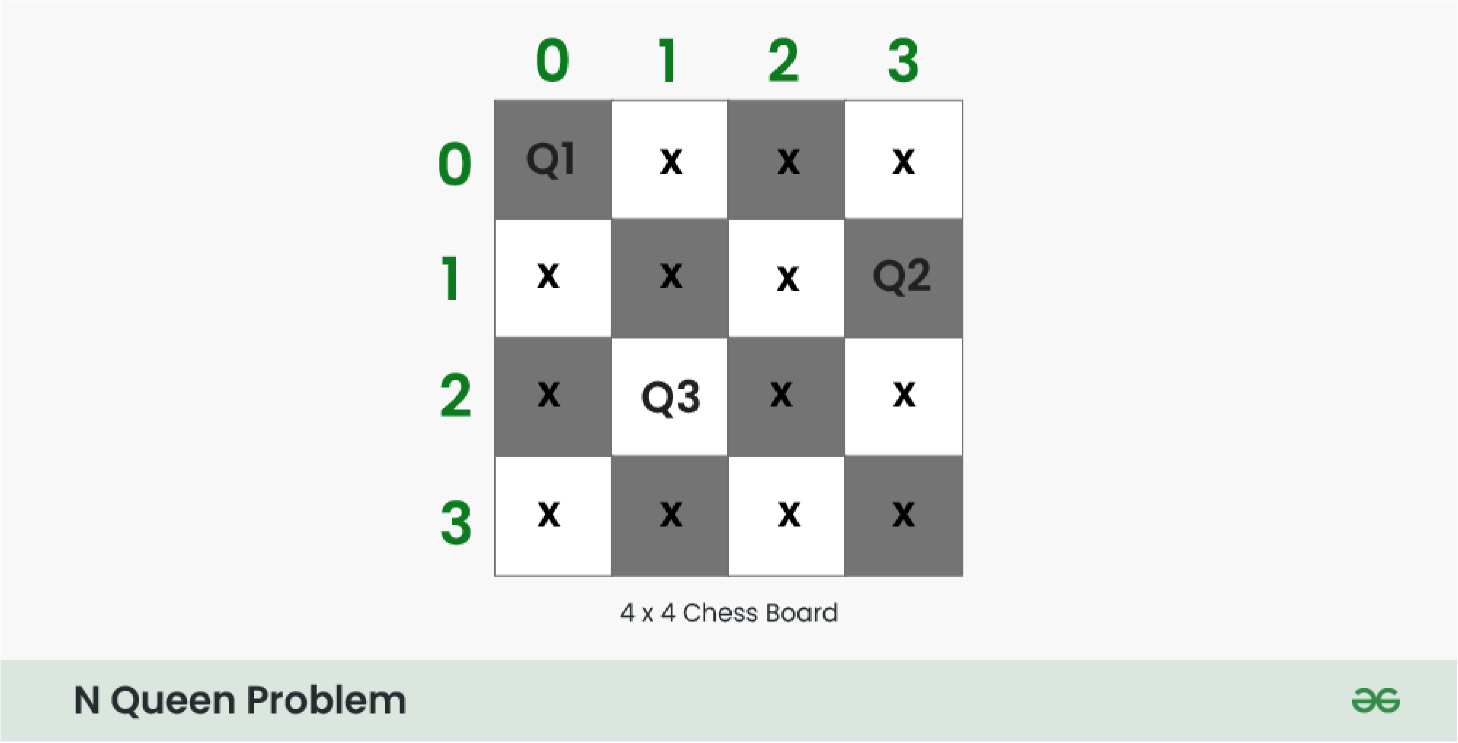
There is still a safe cell in the row 1 i.e. cell (1, 3).Put Queen (Q2) at cell (1, 3).



4-Queens Problem- Example

Step 5:

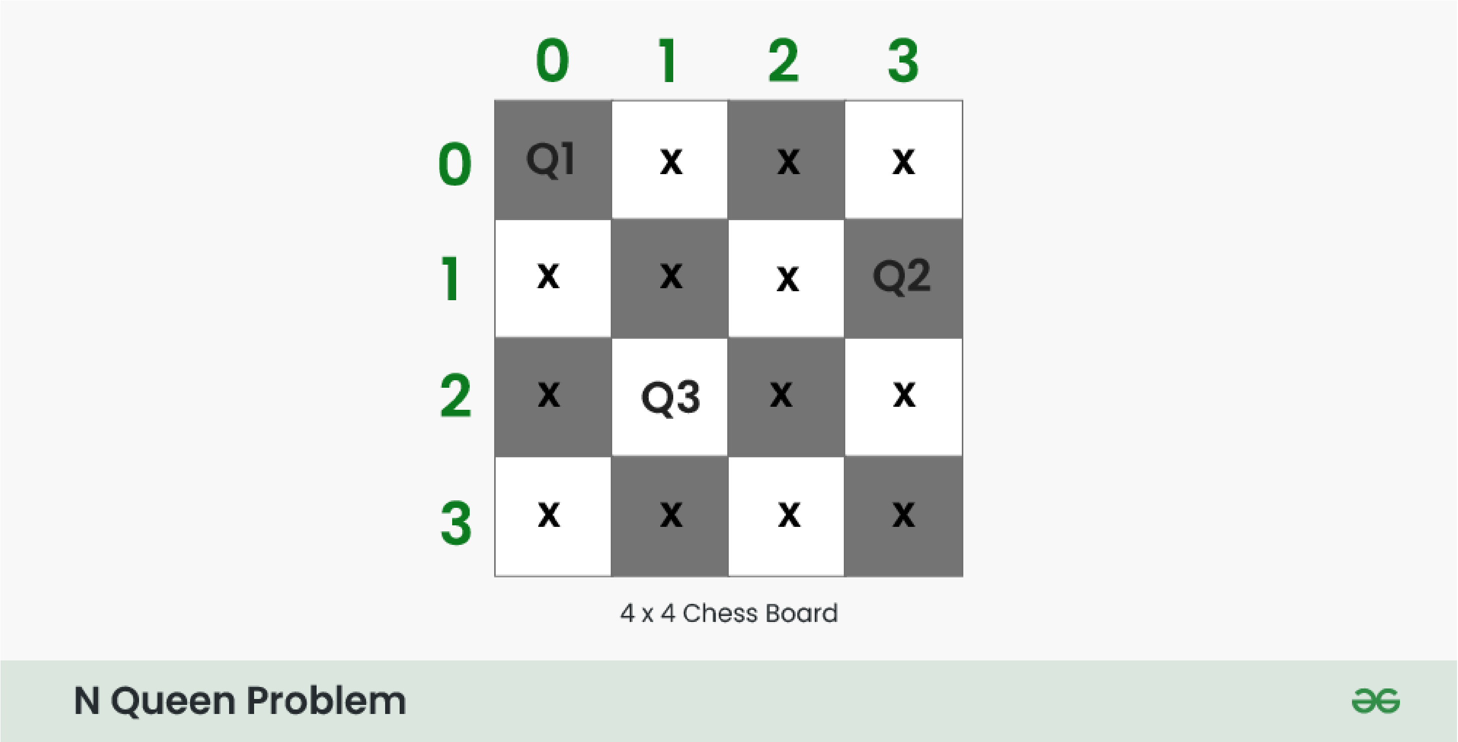
•Put queen **(Q3)** at cell (2, 1).



4-Queens Problem- Example

Step 5:

- Put queen **(Q3)** at cell (2, 1).



4-Queens Problem- Example

Step 6:

•There is no any cell to place Queen (**Q4**) at row 3.

•Backtrack and remove Queen (**Q3)** from row 2.

•Again there is no other safe cell in row 2, So backtrack again and remove queen (**Q2**) from row 1.

•Queen (**Q1)** will be remove from cell **(0,0)** and move to next safe cell i.e. **(0 , 1)**.

4-Queens Problem- Example

Step 6:

•There is no any cell to place Queen (**Q4**) at row 3.

•Backtrack and remove Queen (**Q3)** from row 2.

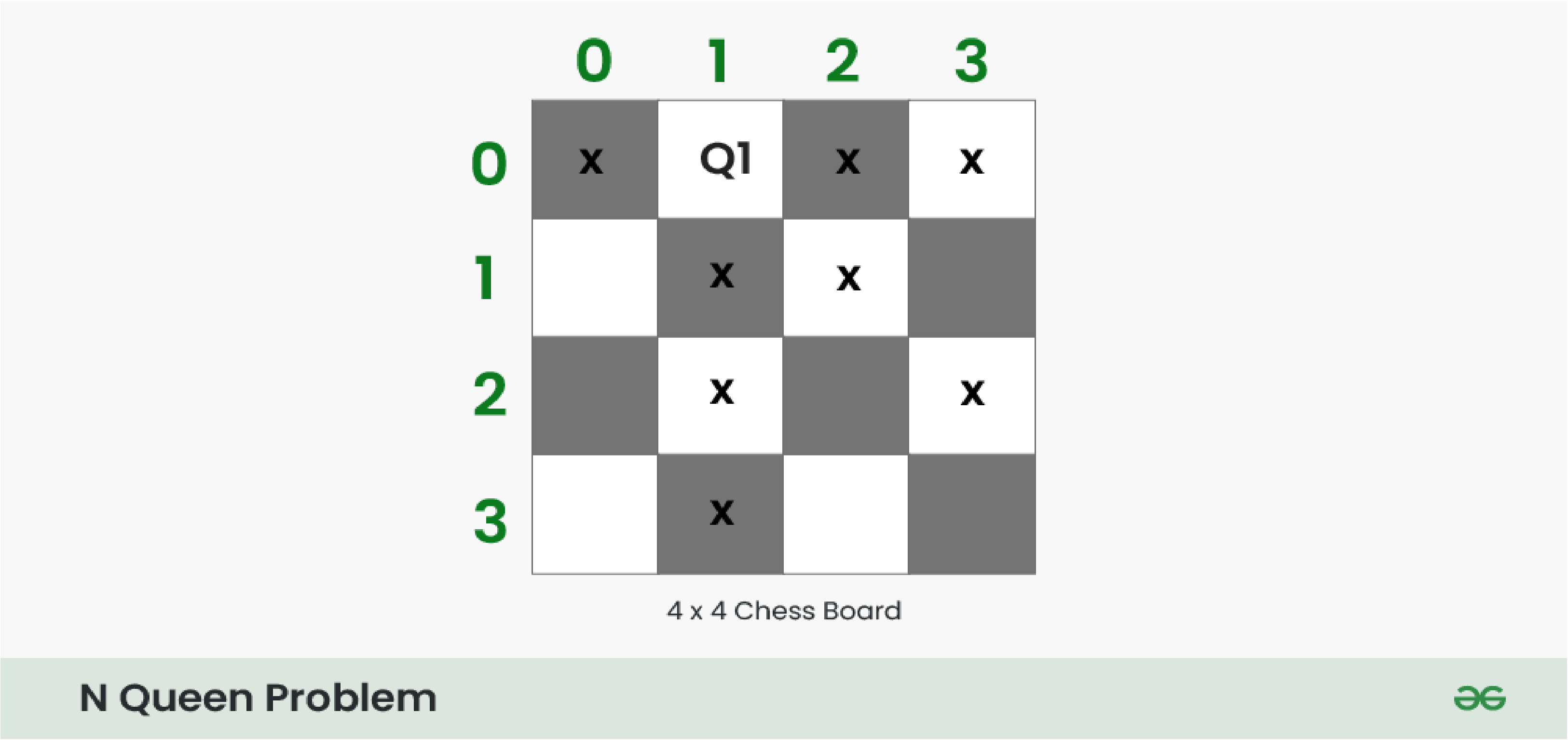
•Again there is no other safe cell in row 2, So backtrack again and remove queen (**Q2**) from row 1.

•Queen (**Q1)** will be remove from cell **(0,0)** and move to next safe cell i.e. **(0 , 1)**.

4-Queens Problem- Example

Step 7:

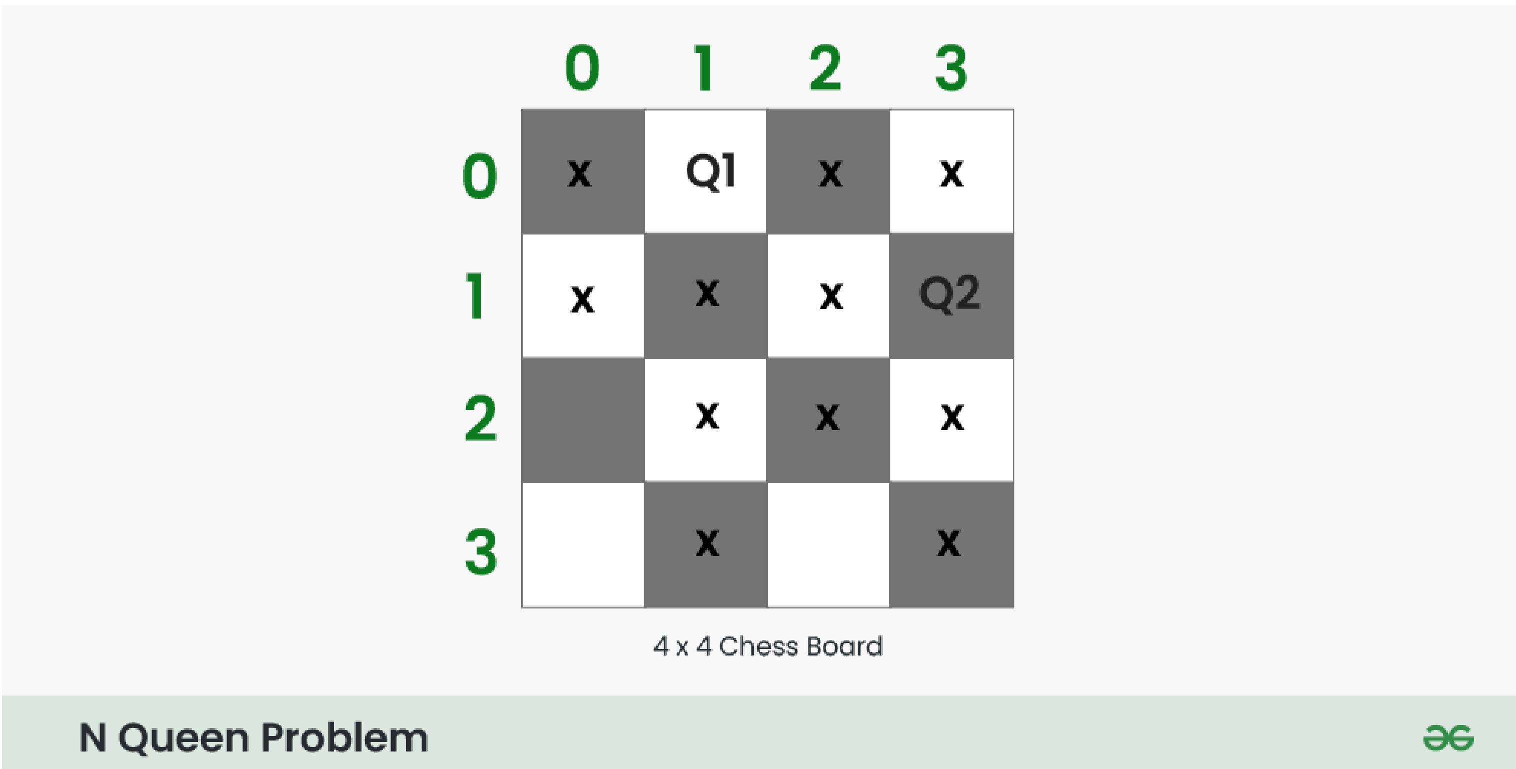
- Place Queen Q1 at cell (0 , 1), and move to next row.



4-Queens Problem- Example

Step 8:

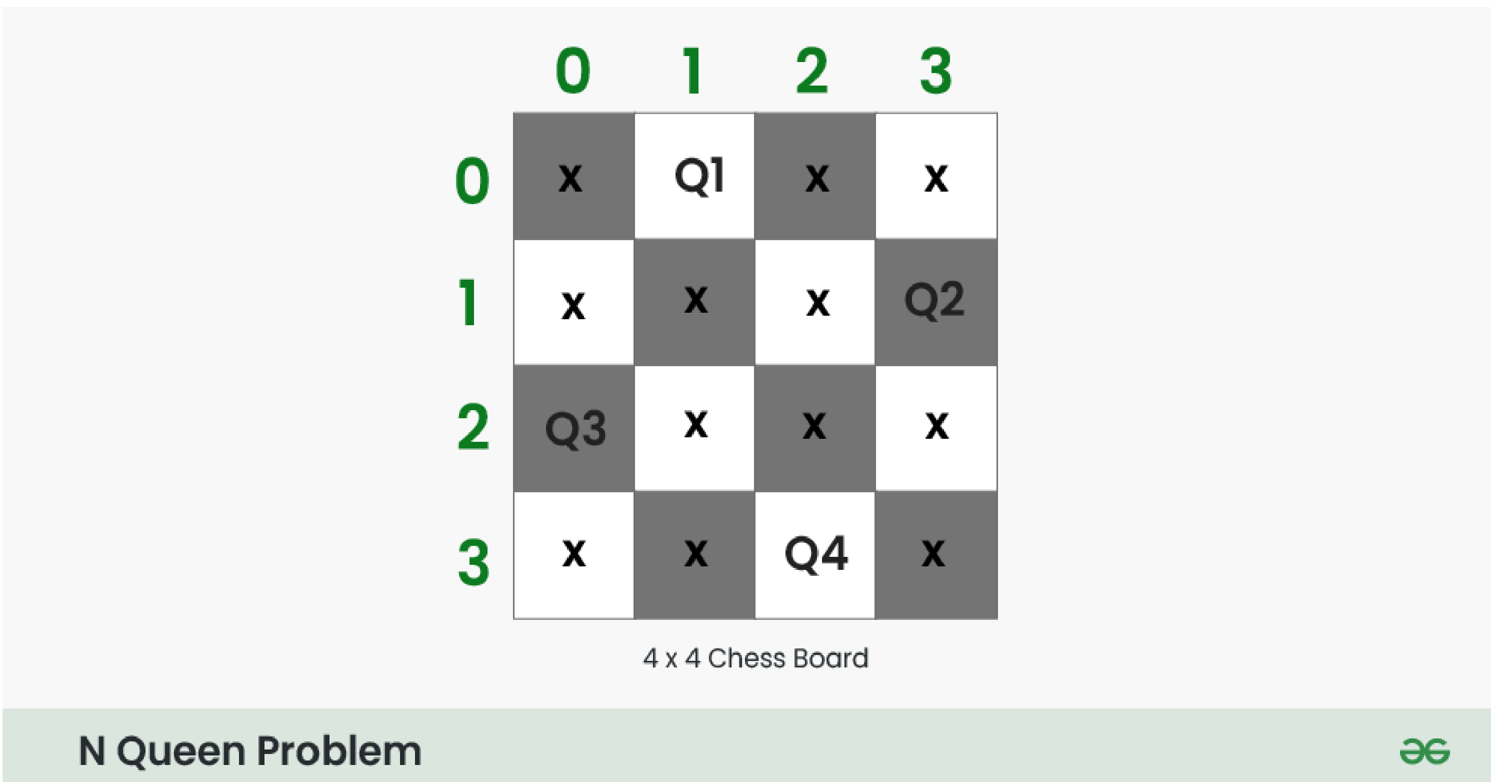
Place Queen Q2 at cell (1 , 3), and move to next row.



4-Queens Problem- Example

Step 9:

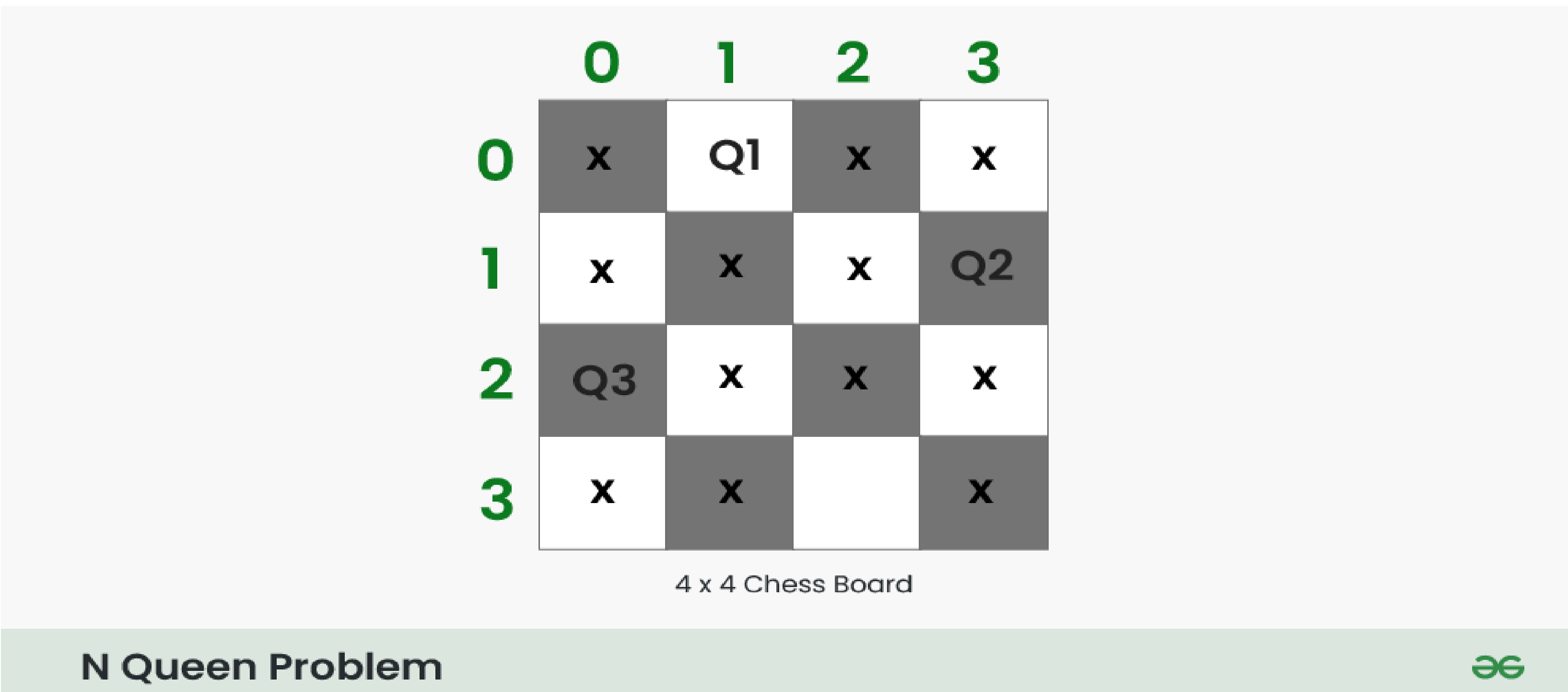
Place Queen Q3 at cell (2 , 0), and move to next row.



4-Queens Problem- Example

Step 10:

Place Queen Q4 at cell (3 , 2), and move to next row. This is one possible configuration of solution



N-Queens Problem- Algorithm

procedure queens (k, col, diag45, diag135)

{sol[1..k] is k-promising,

col = {sol[i] | 1≤i≤k},

diag45 = {sol[i]–i+1 | 1≤i≤k}, and

diag135 = {sol[i]+i–1 | 1≤i≤k}}

if k = 4 **then**

write sol

else

for j ← 1 **to** 4 **do**

if j ∉ col **and** j – k ∉ diag45 **and** j + k ∉ diag135

then sol[k+1]← j

queens(k + 1, col U {j}, diag45 U {j - k}, diag135 U {j + k})

N-Queens Problem- Algorithm

procedure queens (k, col, diag45, diag135)

{sol[1..k] is k-promising,

col = {sol[i] | 1≤i≤k},

diag45 = {sol[i]–i+1 | 1≤i≤k}, and

diag135 = {sol[i]+i–1 | 1≤i≤k}}

if k = 8 **then** {an 8-promising vector is a solution}

write sol

else {explore (k+1)-promising extensions of sol }

for j ← 1 **to** 8 **do**

if j ∉ col **and** j – k ∉ diag45 **and** j + k ∉ diag135 ∉ sol[k+1] ← j

then sol[k+1]← j

{sol[1..k+1] is (k+1)-promising}

queens(k + 1, col U {j}, diag45 U {j - k}, diag135 U {j + k})

Travelling Salesman Problem

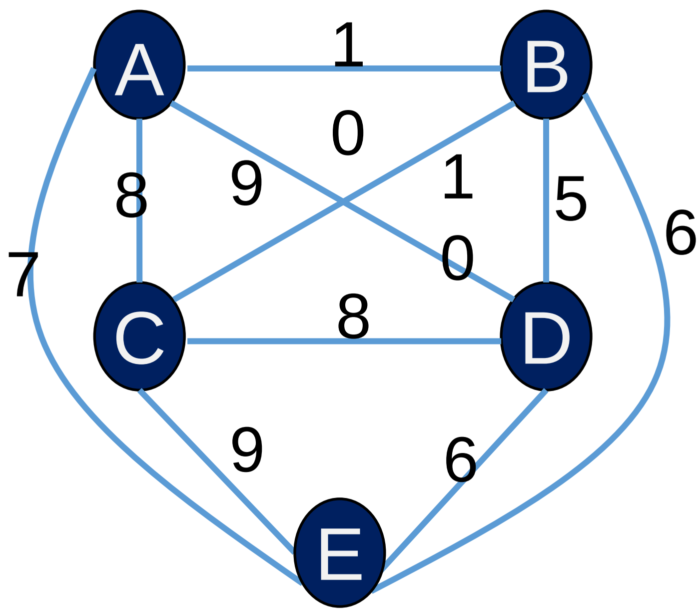
- A traveler needs to visit all the cities from a list, where distances between all the cities are known and each city should be visited just once.
- So, the problem is to find the shortest possible route that visits each city exactly once and returns to the starting point.
- Solution:
    - Consider city 1 as the starting and ending point.
    - Generate all (n-1)! Permutations of cities.
    - Calculate cost of every permutation and keep track of minimum cost permutation.
    - Return the permutation with minimum cost.
- Time Complexity is $\boldsymbol{\Theta}(\boldsymbol{n}!)$

Travelling Salesman Problem

- Travelling Salesman Problem (TSP) – Introduction

#cities	#tours
5	12
6	60
7	360
8	2,520
9	20,160
10	181,440

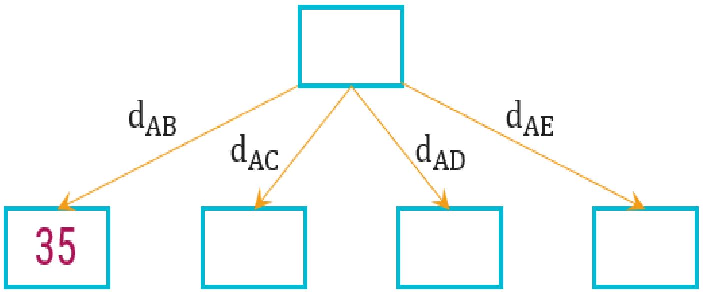
TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

- Here, total minimum distance = sum of row/column minimum = 31
- The upper bound = A ⟶B⟶C⟶D⟶E⟶A =
- Solution : [31…41]

TSP using Branch & Bound



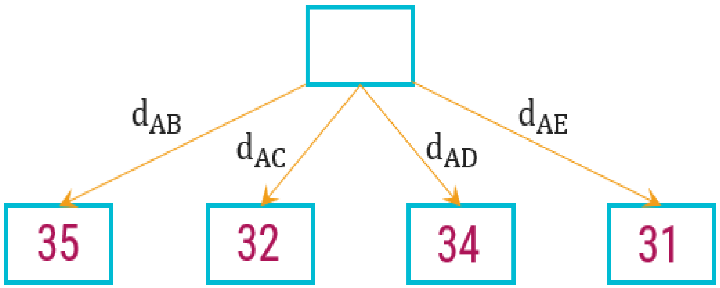
	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

	A	C	D	E
B	--	10	5	6
C	8	--	8	9
D	9	8	--	6
E	7	9	6	--

$$d_{AB} = 10 + 5 + 8 + 6 + 6 = 35$$

Distance from A to B

TSP using Branch & Bound



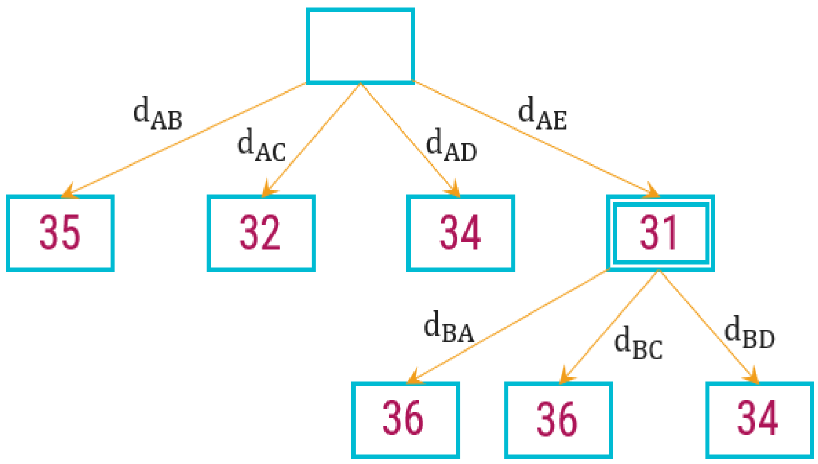
	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

	A	C	D	E
B	10	--	5	6
C	--	10	8	9
D	9	5	--	6
E	7	6	6	--

$$d_{AC} = 8 + 5 + 8 + 5 + 6 = 32$$

Distance from A to C

TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10		10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

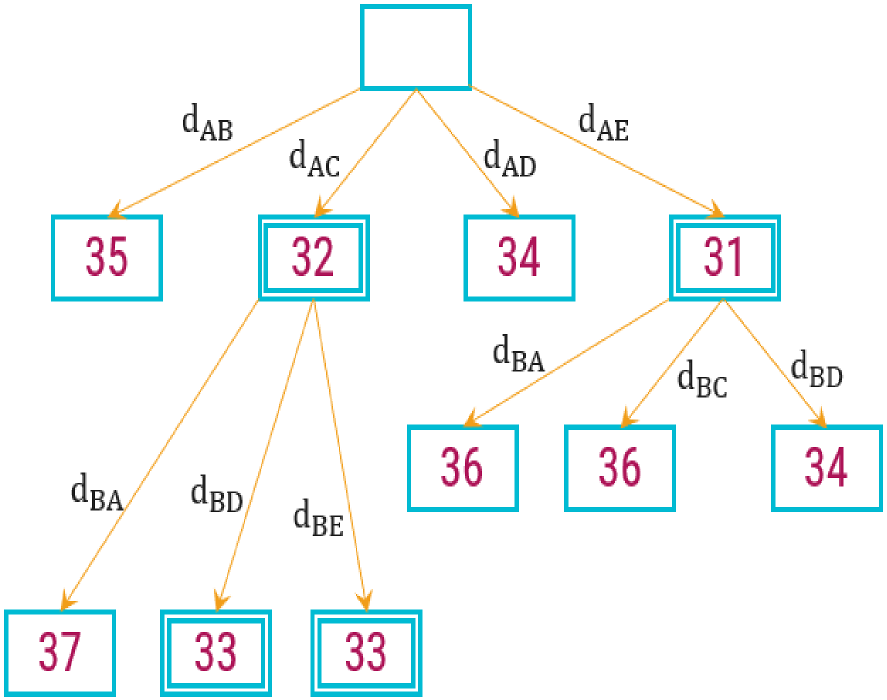
	A	B	D
C	8	--	8
D	9	5	--
E	7	6	6

For d_{AE} and $d_{BC} = 7 + 10 + 8 + 5 + 6 = 36$

Dista from

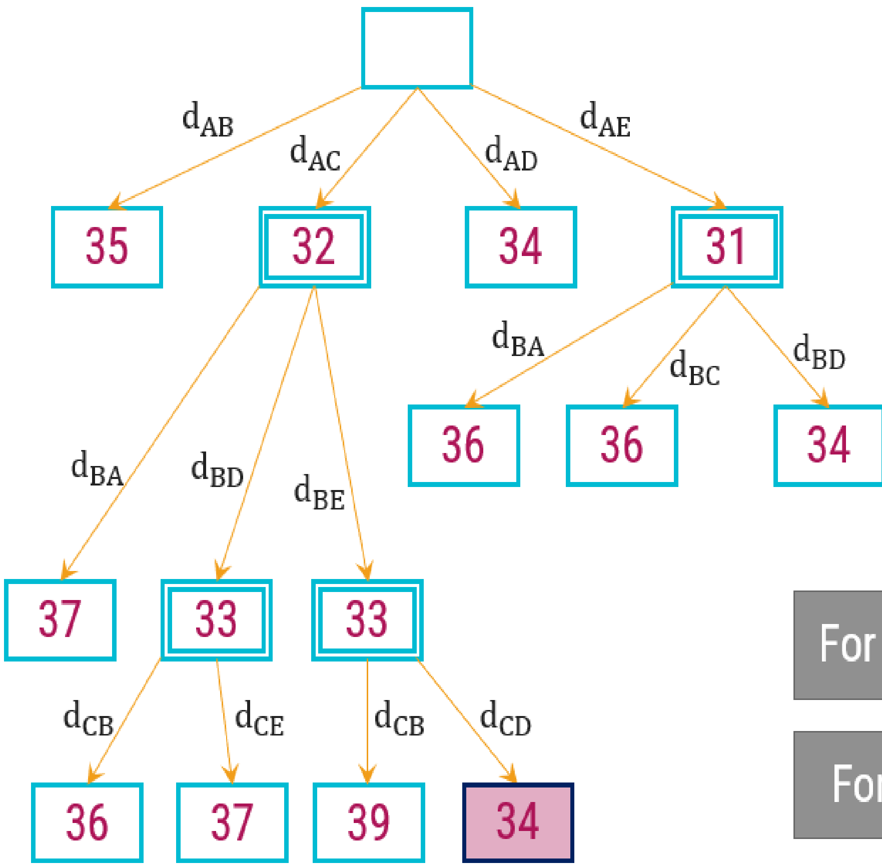
Distance from B to C

TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

TSP using Branch & Bound



	A	B	C	D	E
A	--	10	8	9	7
B	10	--	10	5	6
C	8	10	--	8	9
D	9	5	8	--	6
E	7	6	9	6	--

For $d_{AC} + d_{BE} + d_{CB} = 8 + 6 + 10 + 9 + 6 = 39$

For $d_{AC} + d_{BE} + d_{CD} = 8 + 6 + 8 + 5 + 7 = 34$

The optimal route is A – C – D – B – E – A with total cost = 34

Difference between Branch & Bound and Backtracking

Branch & Bound	Backtracking
Branch-and-Bound is used to solve optimization problems.	Backtracking is a general algorithm for finding all or some solutions to the computational problems
A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions. The set of candidate solutions is thought of as forming a rooted tree, the algorithm explores branches of this tree, which represent the subsets of the solution set.	It incrementally builds candidates to the solutions, and backtracks as soon as it determines that the candidate cannot possibly be completed to a valid solution.
Branch-and-Bound traverse the tree in any manner, DFS or BFS.	It traverses the state space tree by DFS(Depth First Search) manner.

Difference between Branch & Bound and Backtracking

Branch & Bound	Backtracking
Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.	It is an algorithmic-technique for solving problems using recursive approach by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.
Branch-and-Bound involves a bounding function	Backtracking involves feasibility function.
Branch-and-Bound is less efficient.	Backtracking is more efficient.

Parul® University | NAAC GRADE A++

https://paruluniversity.ac.in/