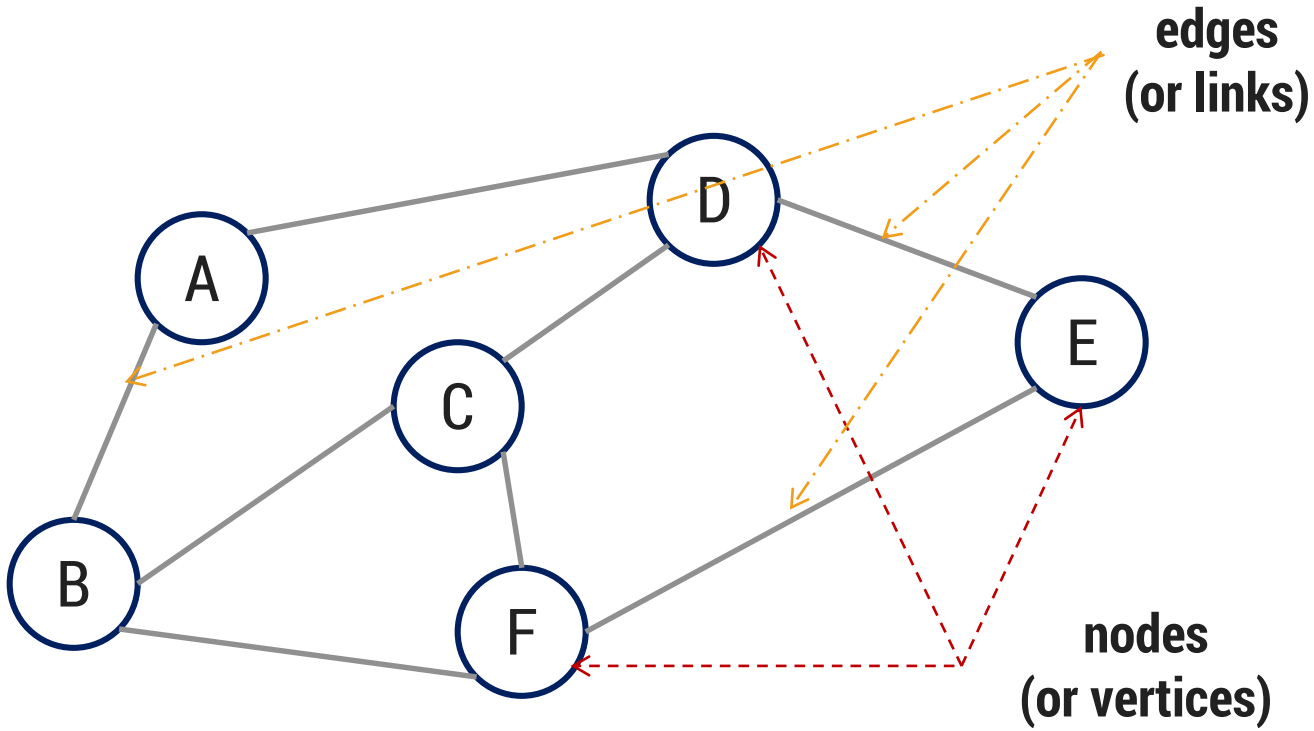
Unit 6 : Graph Algorithms

Outline

- An introduction using Graphs
- Undirected Graph
- Directed Graph
- Traversing Graphs
- Depth First Search (DFS)
- Breath First Search (BFS)
- Topological sort
- Back tracking

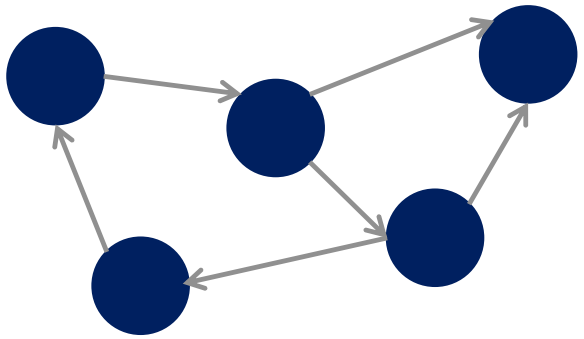
Graph - Definition

- A graph $G = \langle N,)(A \rangle$ consists of a non-empty set N called the set of nodes (vertices) of the graph, a set A called the set of edges that also represents a mapping from the set of edges A to a set of pairs of elements N.

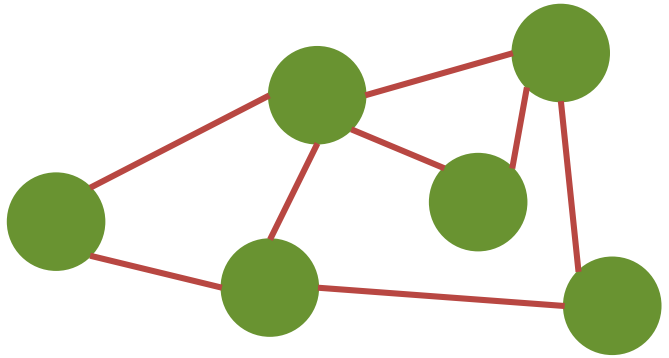


Directed & Undirected Graph

- **Directed Graph**: A graph in which **every edge is directed** from one node to another is called a directed graph or digraph.
- **Undirected Graph**: A graph in which **every edge is undirected and no direction is associated with them** is called an undirected graph.



Directed Graph

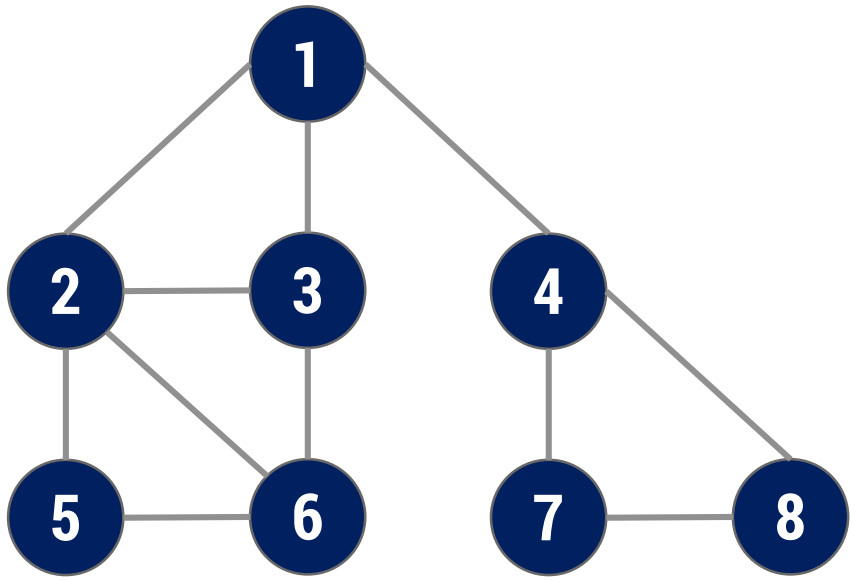


Undirected Graph

Traversing Graph/Tree

- **Preorder**
  i. Visit the **root**.
  ii. Traverse the **left sub tree** in preorder.
  iii. Traverse the **right sub tree** in preorder.
- **In order**
  i. Traverse the **left sub tree** in in order.
  ii. Visit the **root**.
  iii. Traverse the **right sub tree** in in order.
- **Post order**
  i. Traverse the **left sub tree** in post order.
  ii. Traverse the **right sub tree** in post order.
  iii. Visit the **root.**

Depth-First Search / Traversal



Select any node $v \in N$ as starting point mark that node as visited

Select one of the unvisited adjacent of current node.
Make it new starting point and mark it as visited

If new node has no unvisited adjacent then move to parent and make it starting point

Visited : 1 2 3 6 5 4 7 8

DFS – Procedure

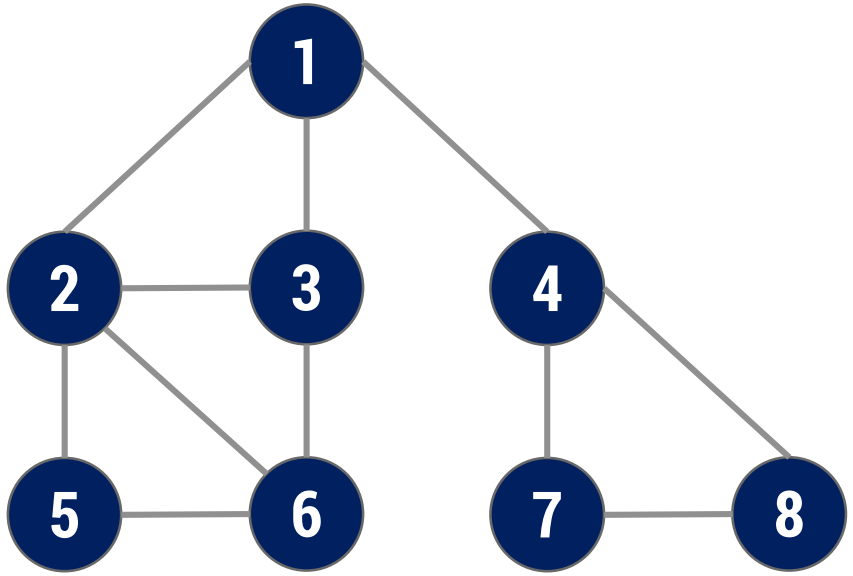
- Let $G = (N, A)$ be an undirected graph all of whose nodes we wish to visit.
- It is somehow possible **to mark a node** to show it has already been visited.
- To carry out a **depth-first traversal** of the graph, choose any node $v \in N$ as the starting point.
- Mark this node to show it has been **visited**.
- If there is a node adjacent to v that has not yet been visited, choose this node as a new starting point and call the depth-first search procedure **recursively**.
- When all the nodes adjacent to v **are marked**, the search starting at v is finished.
- If there remain any nodes of G that **have not been visited**, choose any one of them as a **new starting point**, and call the procedure again.

Depth-First Search Algorithm

```
procedure dfsearch(G)
    for each v ∈ N do
            mark[v] ← not-visited
    for each v ∈ N do
            if mark[v] ≠ visited
            then dfs(v)
procedure dfs(v)
    {Node v has not previously been visited}
    mark[v] ← visited
    for each node w adjacent to v do
            if mark[w] ≠ visited
            then dfs(w)
```

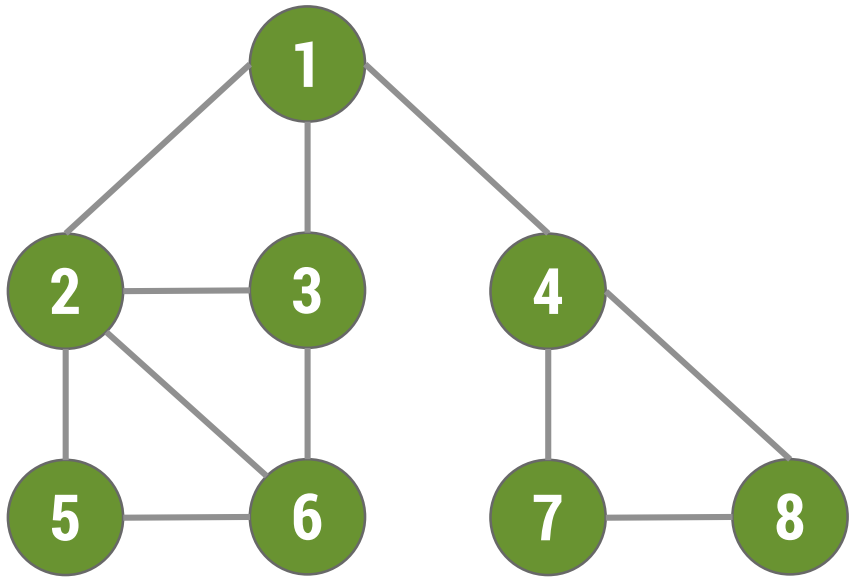
Depth-First Search - Algorithm

```
1. dfs(1)              Initial call
2.   dfs(2)            recursive call
3.     dfs(3)          recursive call
4.       dfs(6)        recursive call
5.         dfs(5)      recursive
   call; progress is blocked
6.   dfs(4)                  a
   neighbour of node 1 that has
   not been visited
7.     dfs(7)          recursive call
8.       dfs(8)        recursive call
9. There are no more nodes to
   visit
```




```
procedure dfs(v)
    mark[v] ← visited
    for each node w adjacent to v do
        if mark[w] ≠ visited
        then dfs(w)
```

Breadth First Search / Traversal



Select any node v ∈ N as starting point mark that node as visited.

Enqueue visited v node into queue Q

Dequeue a node from the front of queue.
Find it's all unvisited adjacent nodes, mark as visited, enqueue into queue

1	2	3	4	5	6	7	8

Queue Q

Visited : 1 2 3 4 5 6 7 8

Breadth First Search - Algorithm

```
procedure search(G)
    for each v ∈ N do
      mark[v] ← not visited
    for each v ∈ N do
      if mark[v] ≠ visited
      then bfs(v)
```

```
procedure bfs(v)
    Q ← empty-queue
    mark[v] ← visited
    enqueue v into Q
    while Q is not empty do
       u ← first(Q)
       dequeue u from Q
       for each node w adjacent to u do
             if mark[w] ≠ visited
             then mark[w] ← visited
        enqueue w into Q
```

Comparison of DFS and BFS

Depth First Search (DFS)	Breath First Search (BFS)
DFS traverses according to tree depth. DFS reaches up to the bottom of a subtree, then backtracks.	BFS traverses according to tree level. BFS finds the shortest path to the destination.
It uses a stack to keep track of the next location to visit.	It uses a queue to keep track of the next location to visit.
DFS requires less memory since only nodes on the current path are stored.	BFS guarantees that the space of possible moves is systematically examined; this search requires considerably more memory resources.
Does not guarantee to find solution. Backtracking is required if wrong path is selected.	If there is a solution, BFS is guaranteed to find it.

Comparison of DFS and BFS

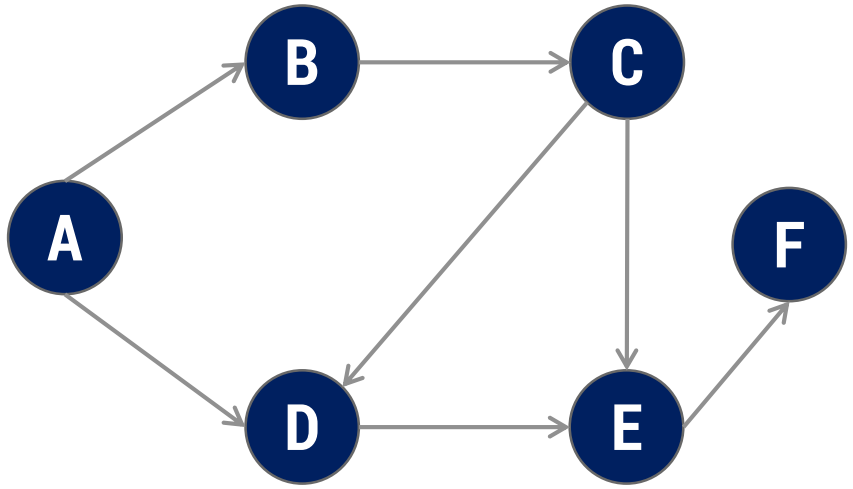
Depth First Search	Breath First Search
If the selected path does not reach to the solution node, DFS gets stuck or trapped into an infinite loops.	BFS will not get trapped exploring an infinite loops.

The Time complexity of both BFS and DFS will be O(V + E), where V is the number of vertices, and E is the number of Edges.

Topological Sorting

- A **topological sort** or **topological ordering** of a directed acyclic graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, the vertex u comes before the vertex v in the ordering.
- Topological Sorting for a graph is not possible if the graph is not a DAG.
- In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices.
- Few important applications of topological sort are-
  - Scheduling jobs from the given dependencies among jobs
  - Instruction Scheduling
  - Determining the order of compilation tasks to perform in makefiles

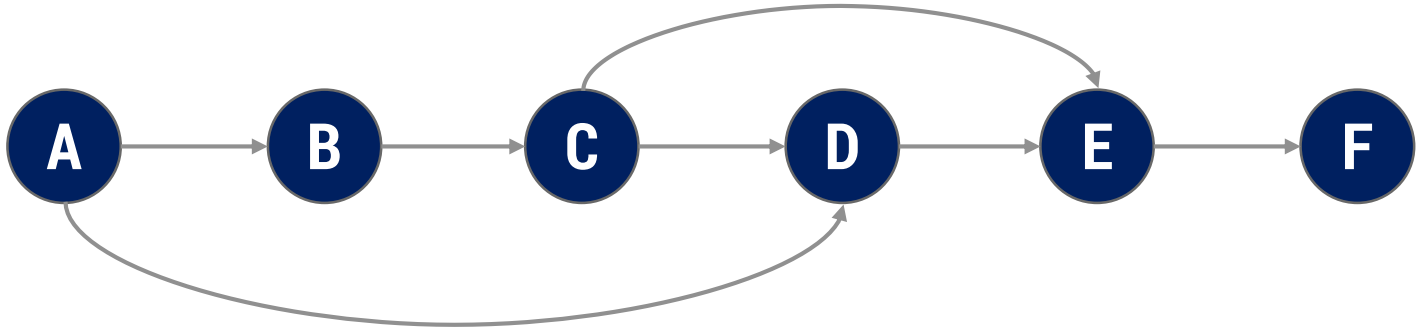
Topological Sorting – Example 1



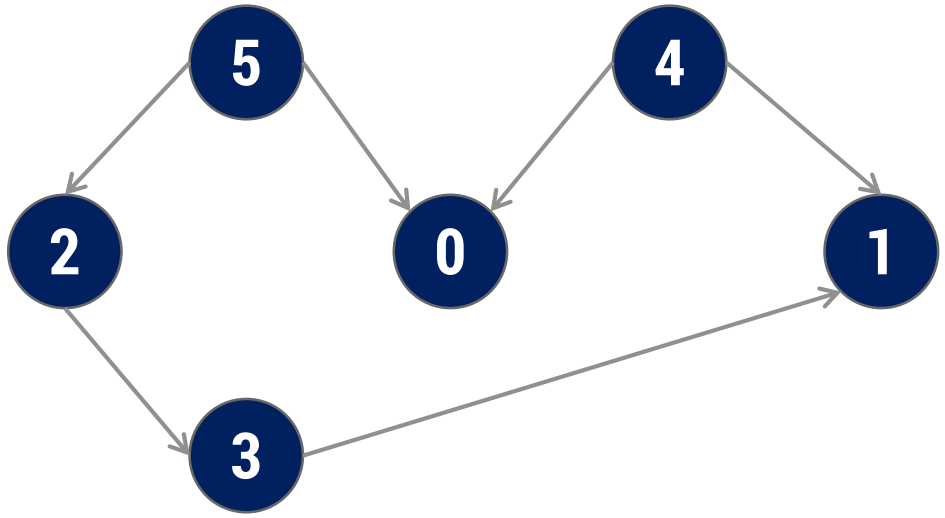
Identify nodes having in degree '0'

Select a node and delete it with its edges then add node to output

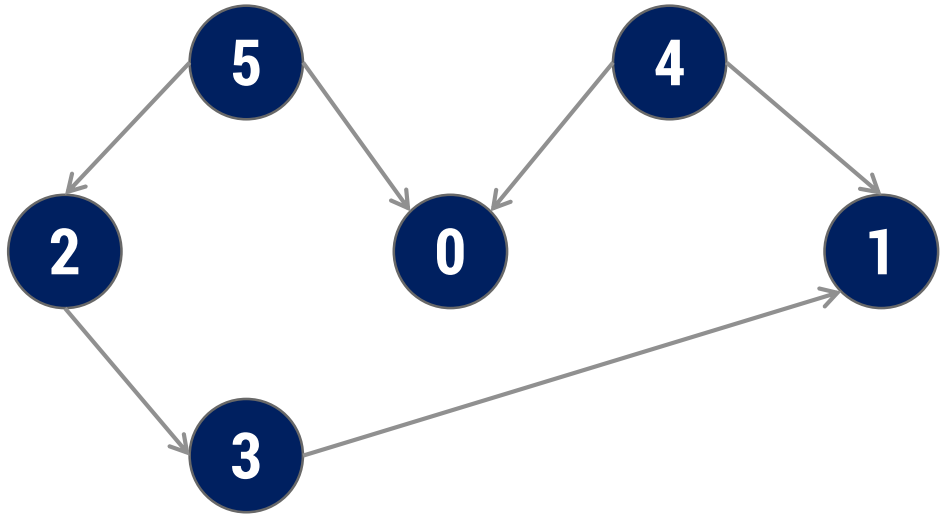
Output:



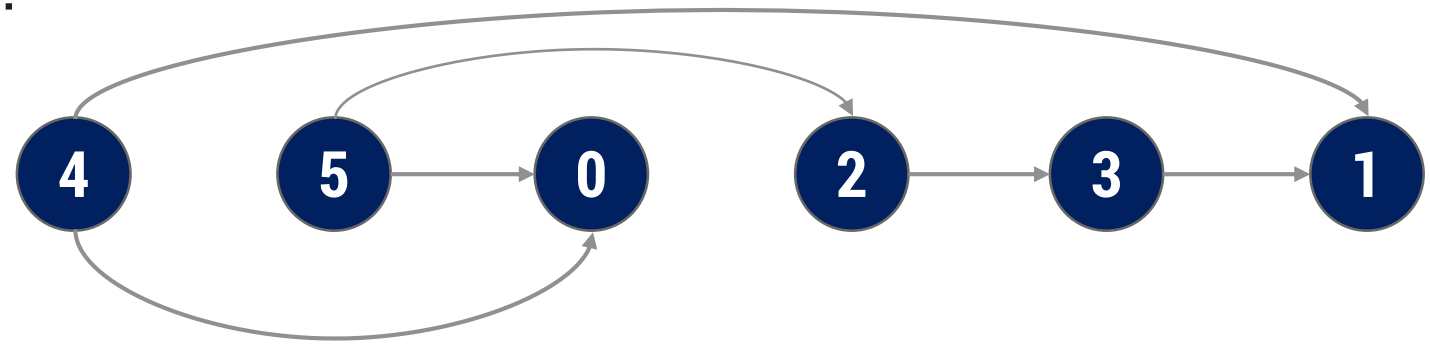
Topological Sorting – Example 2



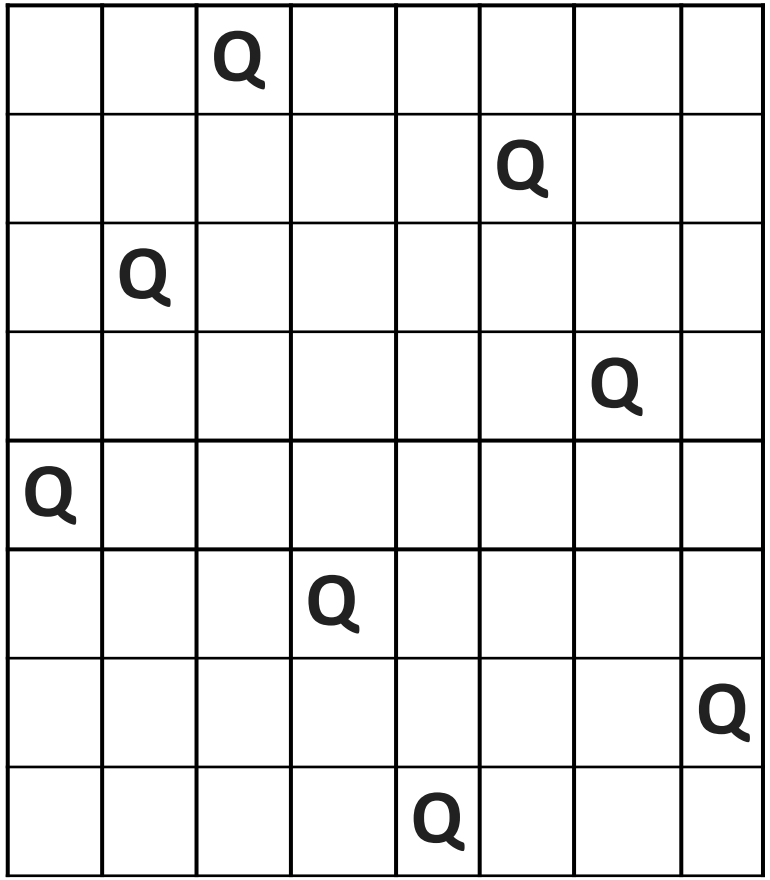
Topological Sorting – Example 2



Output:



N- Queen Problem using Backtracking



Backtracking

- Problems which deal with searching for a set of solution or which ask for an **optimal solution** satisfying some constraints can be solved using the backtracking formulation.
- In many applications of the backtrack method the desired solution is expressible as an **'n' tuple {x_1, x_2,… xn}**, where the **x_i** are chosen from some finite set **s_i** .
- Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a **criterion function p{ x_1, x_2,…,x_n }.**
- m_i = size of set s_i then there are m= {m_1, m_2,…,m_n } n-tuples that are possible candidates for satisfying the function p.
- Brute force approach would be to form all these 'n' tuples, evaluate each one with p, and save those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with **far fewer than m trials.**

Backtracking (cntd...)

- The major advantage of this method is this: if it is realize that the partial vector { x_1, x_2,…,x_i } can in no way lead to an optimal solution, then m_{i+1} …. $m_{n.}$ possible test vector can be ignored entirely.

- **For any problem constraints can be divided in two categories.**

1) Explicit : this constraints are rules that restrict each x_i to take on values only from a given set S_i.
2) Implicit : this constraints are rules that determine which of the tuple in the solution space of I satisfy the criterion function.

.

Applications

N-queen problem

Sum of subset

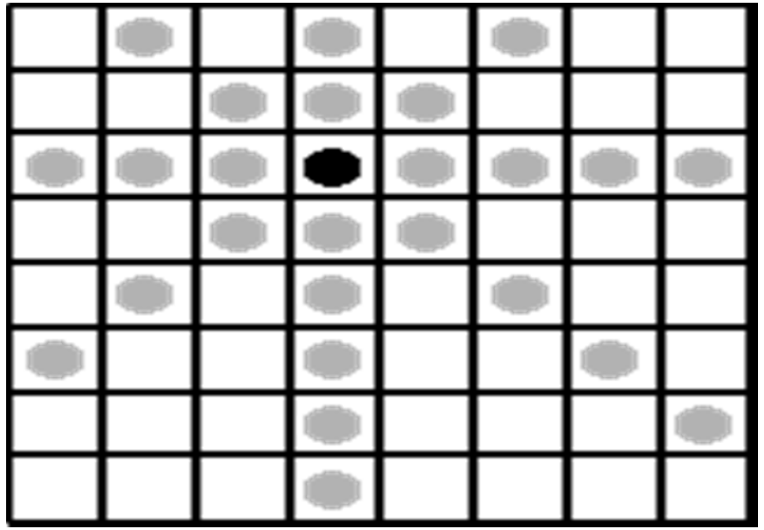
Graph coloring

Hamiltonian cycles

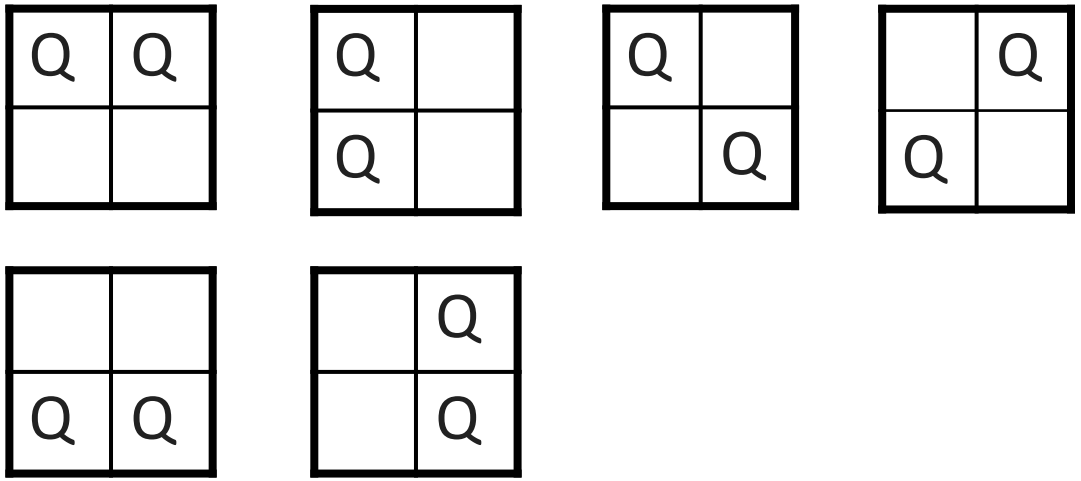
Knapsack problem

N-Queen Problem

- The **n queens puzzle** is the problem of putting ‘n’ chess queens on an n*n chessboard such that no two queens would be able to attack each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. where solutions exist only for $n = 1$ or $n \geq 4$.
- **Explicit** constraint is to take queen between 1 to n only and **implicit** constraint is that no two queen can attack each other.



- Example: 2 Queens problem is not solvable.



4-queen problem

- Lets take a look at the simple problem of placing queens 4 queens on a 4x4 board
- The brute-force solution is to place the first queen, then the second, third, and forth
  - After all are placed we determine if they are placed legally
- There are 16 spots for the first queen, 15 for the second, etc.
  - Leading to 16*15*14*13 = 43,680 different combinations
- Obviously this isn't a good way to solve the problem

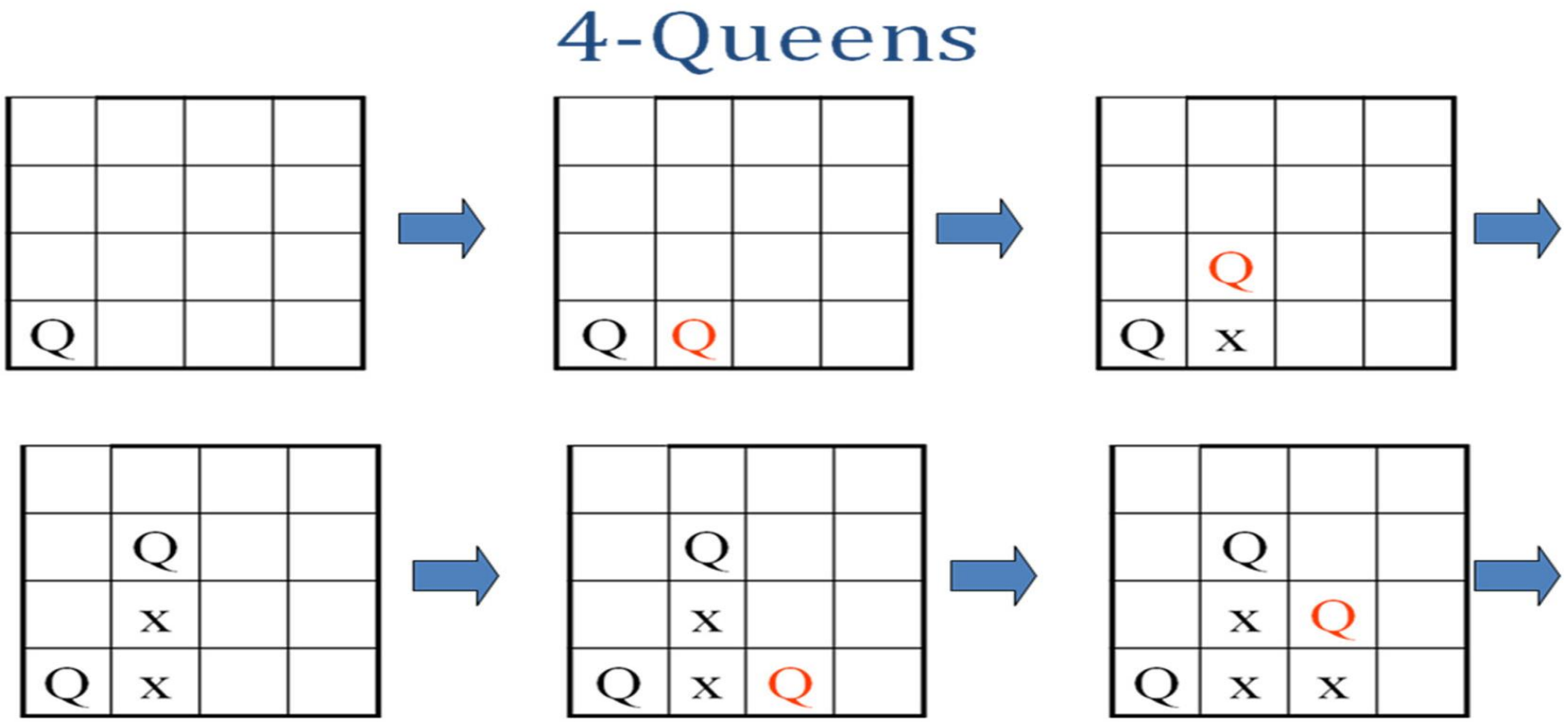
- First lets use the fact that no two queens can be in the same column to help us
  - That means we get to place a queen in each column
- So we can place the first queen into the first column, the second into the second, etc.
- This cuts down on the amount of work
  - Now there are 4 spots for the first queen, 4 spots for the second, etc.

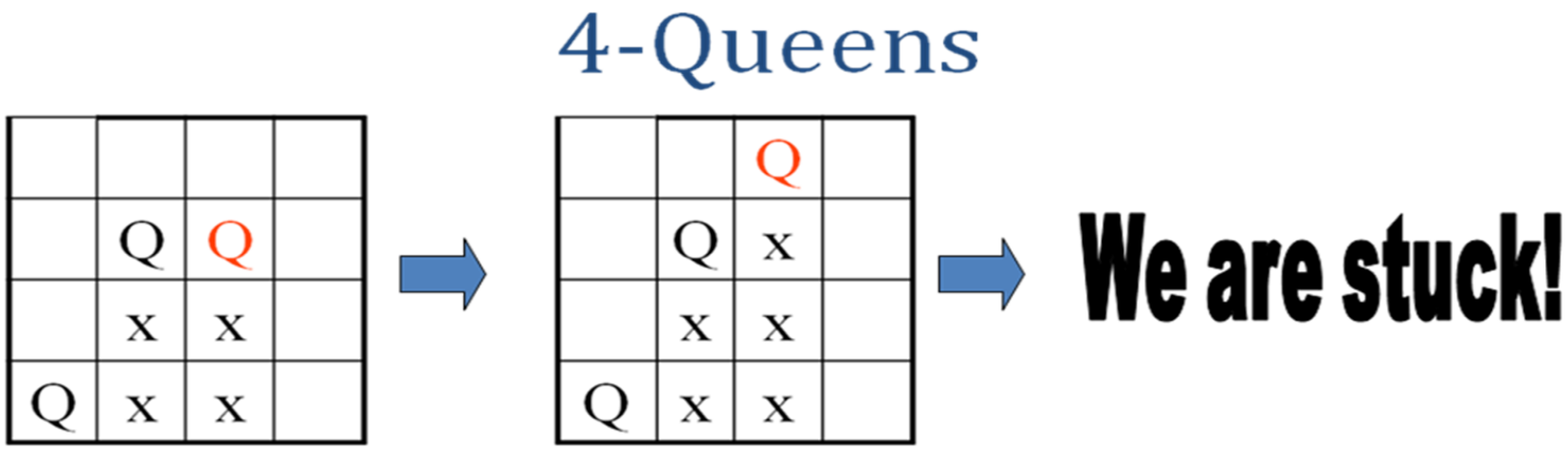
    4*4*4*4 = 256 different combinations

- However, we can still do better because as we place each queen we can look at the previous queens we have placed to make sure our new queen is not in the same row or diagonal as a previously place queen

- Then we could use a Greedy-like strategy to select the next valid position for each column

Example of backtrack solution for the 4 queen problem

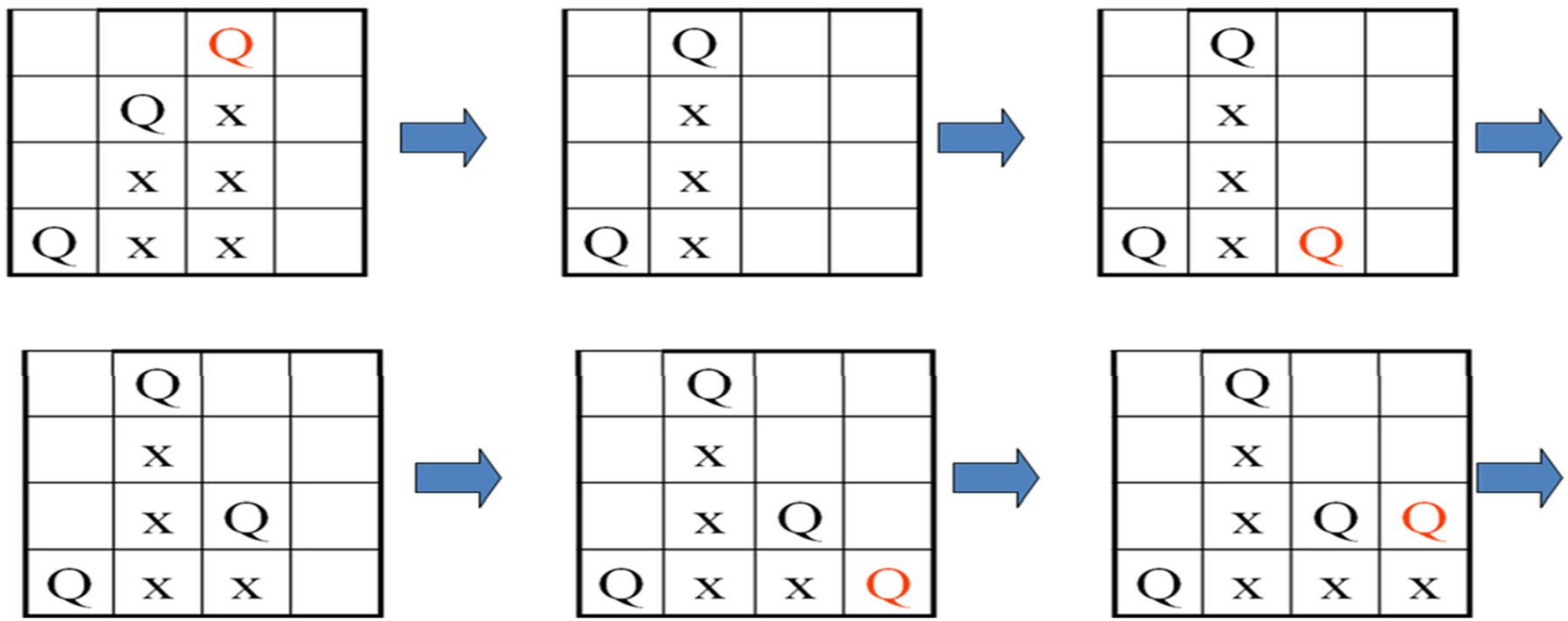




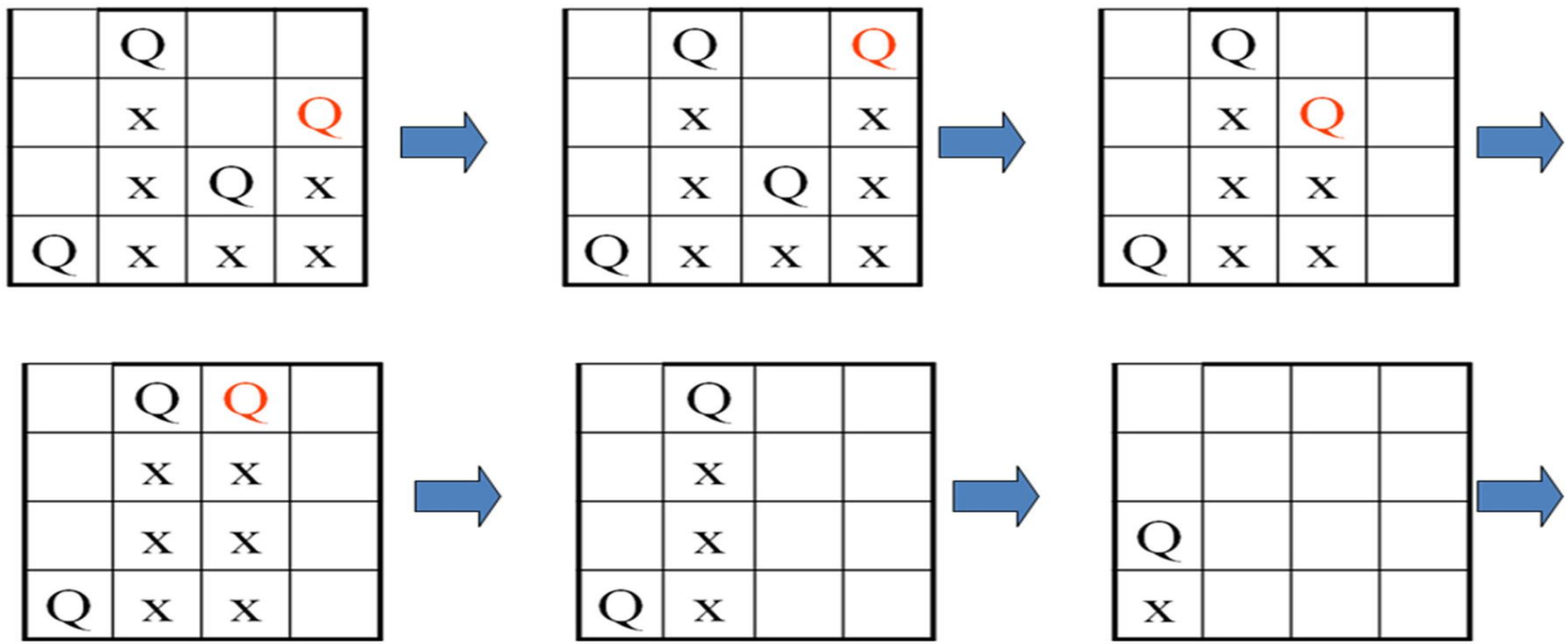
If one of your choices leads to a dead end, you need to back up to the last choice you made and take a different route

That is, you need to change one of your earlier selections

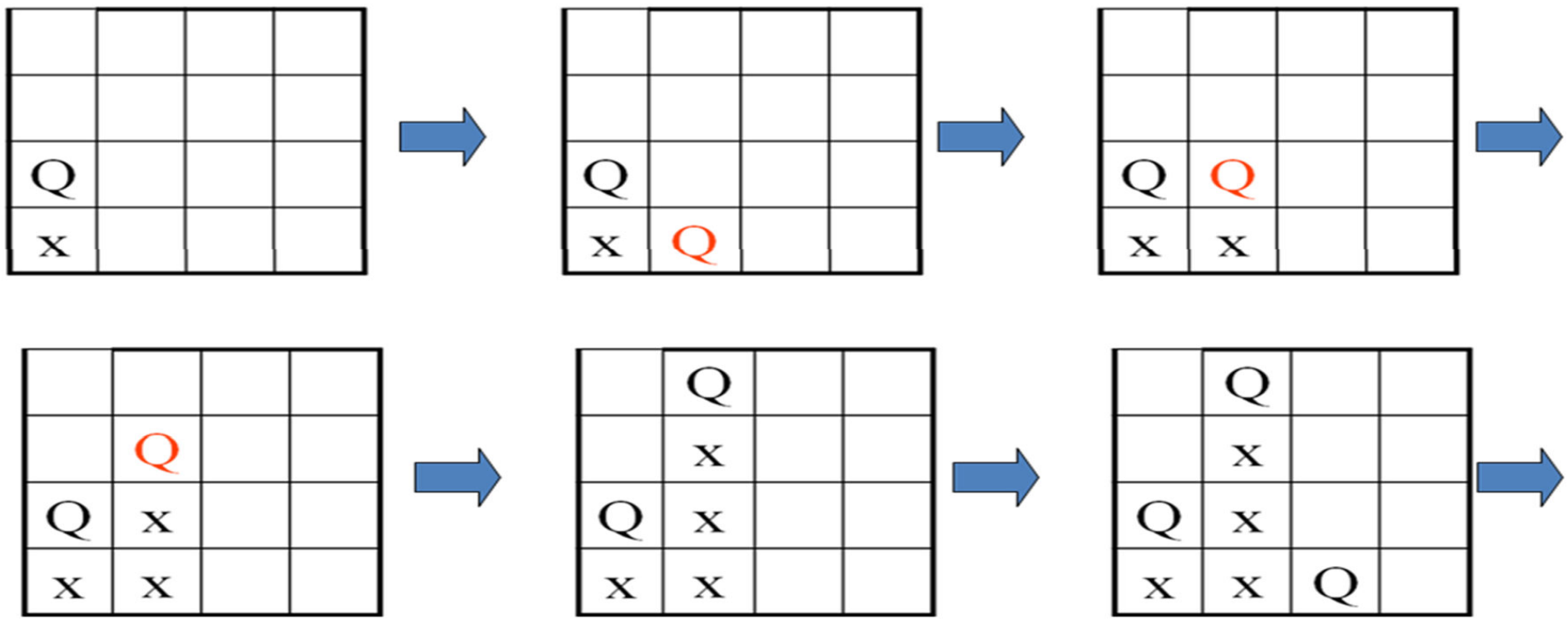
4-Queens



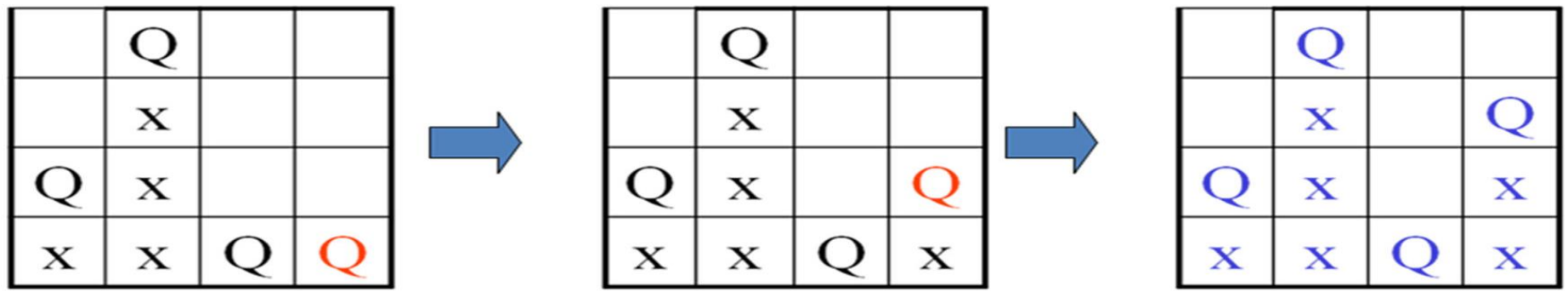
4-Queens

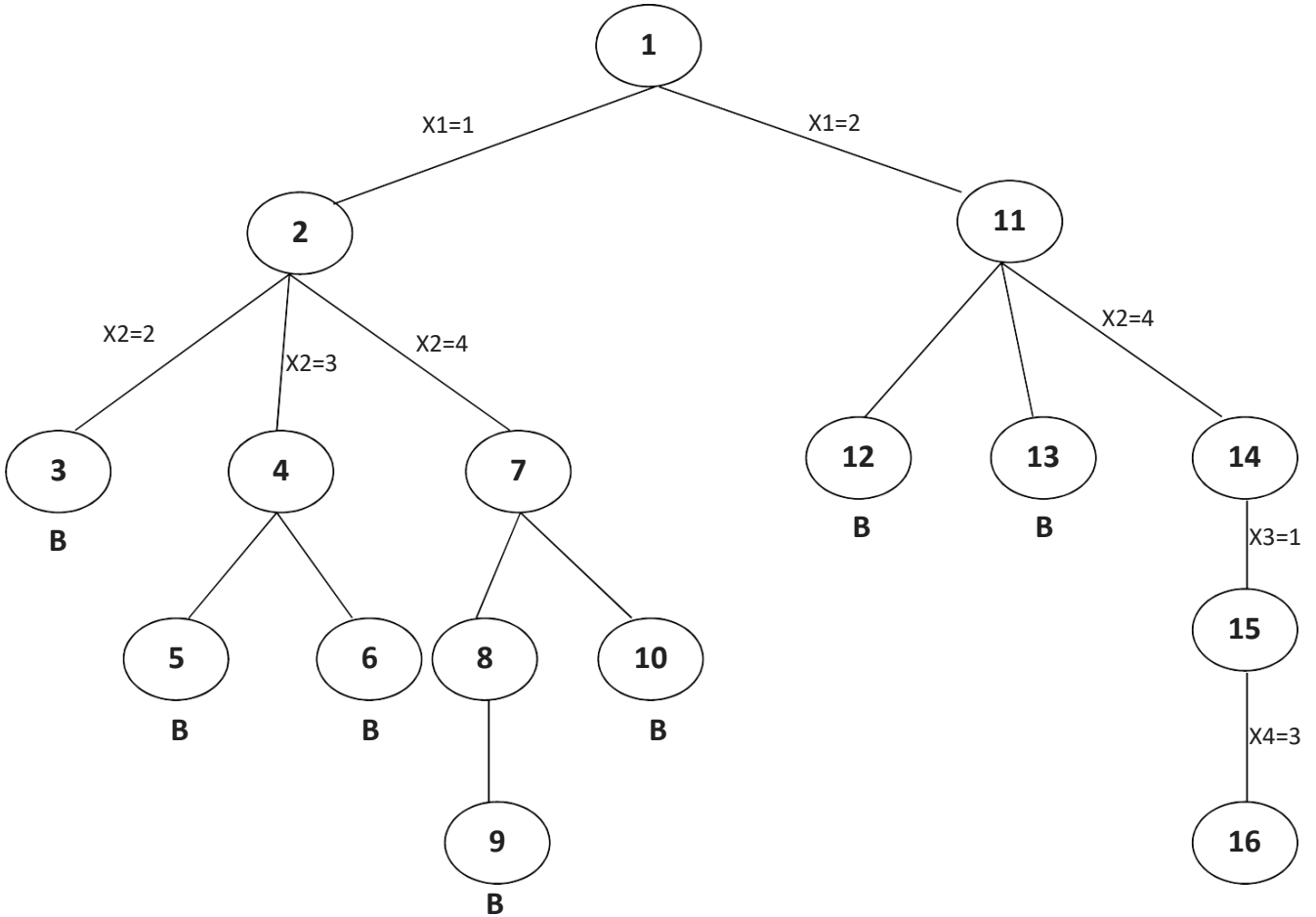

Q
x
Q
x
x
Q
x
Q
x
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x
Q
x
x
x

4-Queens



4-Queens




1
X1=1
X1=2
2
11
X2=2
X2=3
X2=4
X2=4
3
4
7
12
13
14
B
B
B
B
X3=1
5
6
8
10
15
B
B
B
X4=3
9
16
B

New queen can be placed or not???

```
1  Algorithm place(k,i)

2//Returns true if a queen can be placed in kth row and ith column. 3  //Otherwise it returns
false.x[] is a global array whose first (k-1)  4  //values have been  set.Abs(r) returns the
absolute value of r.

 5  {
 6     For j:=1 to k-1 do
 7           If ((x[j]=i)//two in the same column
8                        Or (Abs(x[j]-i) = Abs(j-k))) // or in the same diagonal
9           Then return false;
10Return true; 11 }
```

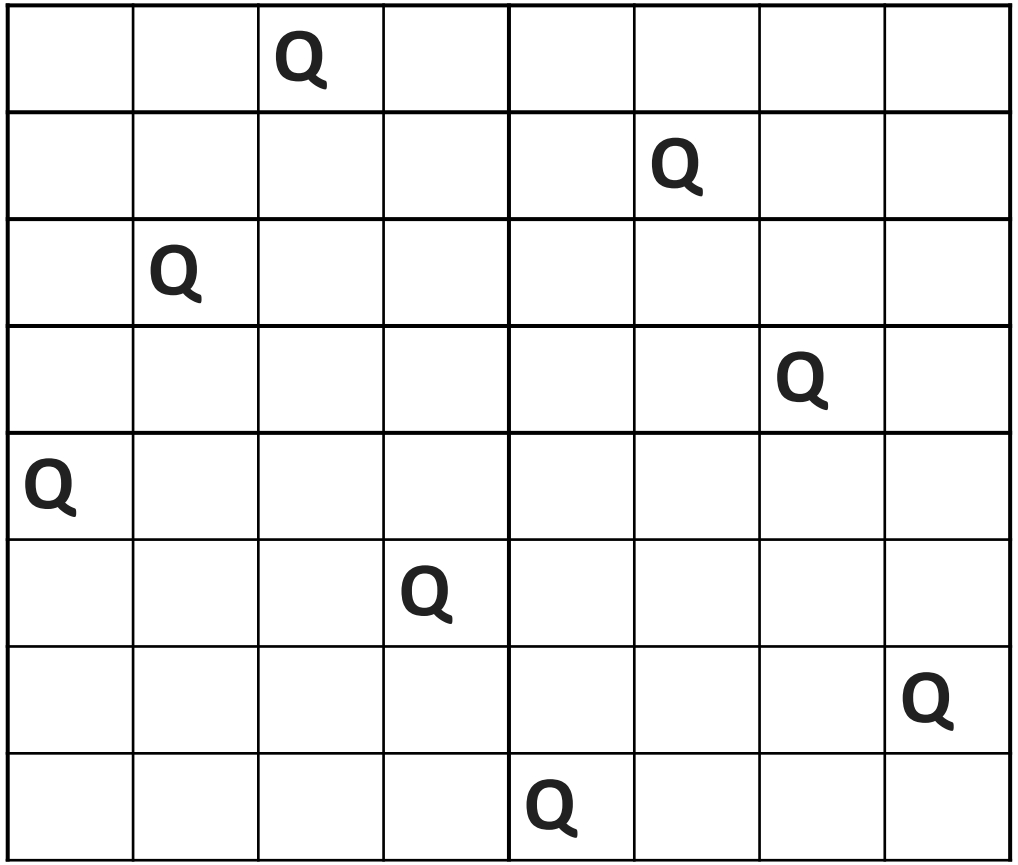
- Place(k,i) returns a Boolean value that is true if the k^{th} queen can be placed in column i.
- It tests both whether i is distinct from all previous values x[1],…,x[k-1] and whether there is no other queen on the same diagonal.
- Its computing time is O(k-1).

```
1  Algorithm NQueens(k,n)

2//Using backtracking, this procedure prints all possible placements of 3  //n queens on an n
    n chessboard so that they are nonattacking.

4  {
5     For i:=1 to n do 6{
7           If place (k,i) then
8           {
9                    X[k]:=I;
10                   If(k=n) then write (x[1:n]);
11Else NQueens(k+1,n); 12        }
13    }
14  }
```

Solution for 8-queen problem



n-Queens	First Lexicographic Solution
1	**{1}**
2	No Solution Exists
3	No Solution Exists
4	{2 4 1 3}
5	{1 3 5 2 4}
6	{2 4 6 1 3 5}
7	{1 3 5 7 2 4 6}
8	{1 5 8 6 3 7 2 4}
9	{1 3 6 8 2 4 9 7 5}
10	{1 3 6 8 10 5 9 2 4 7}
11	{1 3 5 7 9 11 2 4 6 8 10}
12	{1 3 5 8 10 12 6 11 2 7 9 4}
13	{1 3 5 2 9 12 10 13 4 6 8 11 7}
14	{1 3 5 7 12 10 13 4 14 9 2 6 8 11}
15	{1 3 5 2 10 12 14 4 13 9 6 15 7 11 8}
16	{1 3 5 2 13 9 14 12 15 6 16 7 4 11 8 10}
17	{1 3 5 2 8 11 15 7 16 14 17 4 6 9 12 10 13}
18	{1 3 5 2 8 15 12 16 13 17 6 18 7 4 11 9 14 10}
19	{1 3 5 2 4 9 13 15 17 19 7 16 18 11 6 8 10 12 14}
20	{1 3 5 2 4 13 15 12 18 20 17 9 16 19 8 10 7 14 6 11}
21	{1 3 5 2 4 9 11 15 21 18 20 17 19 7 12 10 8 6 14 16 13}
22	{1 3 5 2 4 10 14 17 20 13 19 22 18 8 21 12 9 6 16 7 11 15}
23	{1 3 5 2 4 9 11 13 18 20 22 19 21 10 8 6 23 7 16 12 15 17 14}
24	{1 3 5 2 4 9 11 14 18 22 19 23 20 24 10 21 6 8 12 16 13 7 17 15}
25	{1 3 5 2 4 9 11 13 15 19 21 24 20 25 23 6 8 10 7 14 16 18 12 17 22}

Counting solution

- The following table gives the number of solutions for placing *n* queens on an *n***n* board, both unique and distinct.

N:	1	2	3	4	5	6	7	8	9	10
Unique:	1	0	0	1	2	1	6	12	46	92
Distinct	1	0	0	2	10	4	40	92	352	724