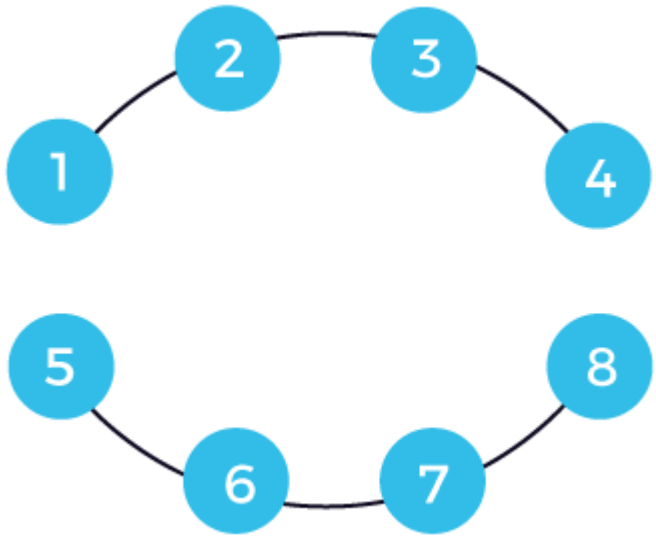
Unit – II

1. Disjoint set data structure

The disjoint set data structure is also known as union-find data structure and merge-find set. It is a data structure that contains a collection of disjoint or non-overlapping sets. The disjoint set means that when the set is partitioned into the disjoint subsets. The various operations can be performed on the disjoint subsets. In this case, we can add new sets, we can merge the sets, and we can also find the representative member of a set. It also allows to find out whether the two elements are in the same set or not efficiently.

The disjoint set can be defined as the subsets where there is no common element between the two sets. Let's understand the disjoint sets through an example.



s1 = {1, 2, 3, 4}

s2 = {5, 6, 7, 8}

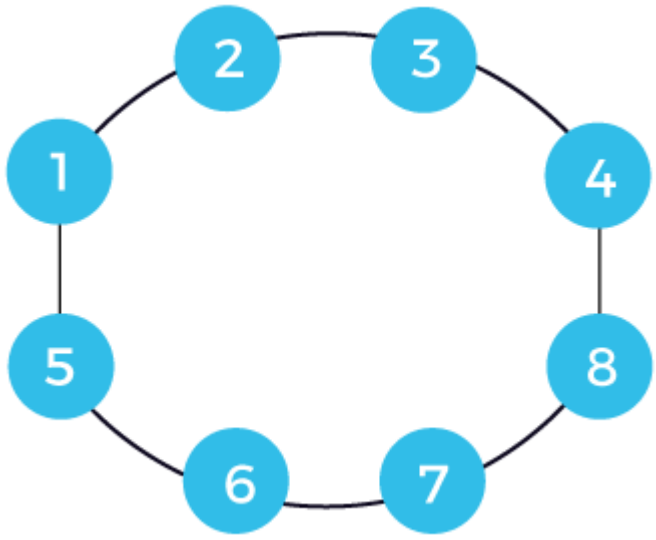
We have two subsets named s1 and s2. The s1 subset contains the elements 1, 2, 3, 4, while s2 contains the elements 5, 6, 7, 8. Since there is no common element between these two sets, we will not get anything if we consider the intersection between these two sets. This is also known as a disjoint set where no elements are common. Now the question arises how we can perform the operations on them. We can perform only two operations, i.e., find and union.

In the case of find operation, we have to check that the element is present in which set. There are two sets named s1 and s2 shown below:

Suppose we want to perform the union operation on these two sets. First, we have to check whether the elements on which we are performing the union operation

belong to different or same sets. If they belong to the different sets, then we can perform the union operation; otherwise, not. For example, we want to perform the union operation between 4 and 8. Since 4 and 8 belong to different sets, so we apply the union operation. Once the union operation is performed, the edge will be added between the 4 and 8 shown as below:

When the union operation is applied, the set would be represented as:



s1Us2 = {1, 2, 3, 4, 5, 6, 7, 8}

Suppose we add one more edge between 1 and 5. Now the final set can be represented as:

s3 = {1, 2, 3, 4, 5, 6, 7, 8}

If we consider any element from the above set, then all the elements belong to the same set; it means that the cycle exists in a graph.

In **pseudocode**, the find operation looks like this:
find(node):
if (parent(node)==node) return node;
else return find(parent(node));

Algorithm

When handling multiple disassociated sets, utilizing the disjoint data structure can prove useful. Each individual grouping is designated with a specific representative characterizing it. The starting point involves each component forming its own isolated set that corresponds with its respective representative (which also happens to be itself). The two primary operations performed on disjoint sets are union and find.

Union operation

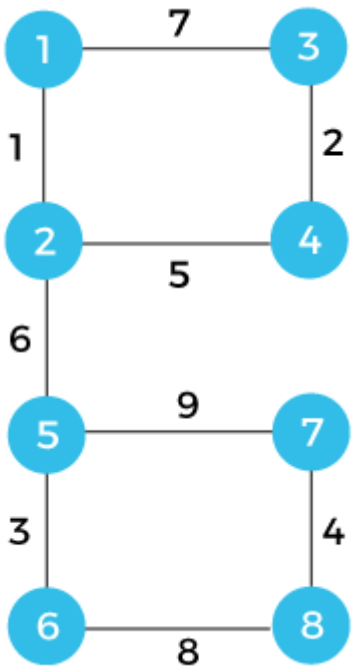
- Find the representatives of the two sets to be merged.
- If the representatives are different, make one representative point to the other, effectively merging the sets.
- If the representatives are the same, the sets are already merged, and no further action is needed.

Find operation

- Given an element, find the representative of the set it belongs to.
- Follow the parent pointers until reaching the representative.
- Return the representative as the result.

How can we detect a cycle in a graph?

We will understand this concept through an example. Consider the below example to detect a cycle with the help of using disjoint sets.



U = {1, 2, 3, 4, 5, 6, 7, 8}

Each vertex is labelled with some weight. There is a universal set with 8 vertices. We will consider each edge one by one and form the sets.

First, we consider vertices 1 and 2. Both belong to the universal set; we perform the union operation between elements 1 and 2. We will add the elements 1 and 2 in a set s1 and remove these two elements from the universal set shown below:

s1 = {1, 2}

The vertices that we consider now are 3 and 4. Both the vertices belong to the universal set; we perform the union operation between elements 3 and 4. We will form the set s3 having elements 3 and 4 and remove the elements from the universal set shown as below:

s2 = {3, 4}

The vertices that we consider now are 5 and 6. Both the vertices belong to the universal set, so we perform the union operation between elements 5 and 6. We will form the set s3 having elements 5 and 6 and will remove these elements from the universal set shown as below:

s3 = {5, 6}

The vertices that we consider now are 7 and 8. Both the vertices belong to the universal set, so we perform the union operation between elements 7 and 8. We will form the set s4 having elements 7 and 8 and will remove these elements from the universal set shown as below:

s4 = {7, 8}

The next edge that we take is (2, 4). The vertex 2 is in set 1, and vertex 4 is in set 2, so both the vertices are in different sets. When we apply the union operation, then it will form the new set shown as below:

s5 = {1, 2, 3, 4}

The next edge that we consider is (2, 5). The vertex 2 is in set 5, and the vertex 5 is in set s3, so both the vertices are in different sets. When we apply the union operation, then it will form the new set shown as below:

s6 = {1, 2, 3, 4, 5, 6}

Now, two sets are left which are given below:

s4 = {7, 8}

s6 = {1, 2, 3, 4, 5, 6}

The next edge is (1, 3). Since both the vertices, i.e.,1 and 3 belong to the same set, so it forms a cycle. We will not consider this vertex.

The next edge is (6, 8). Since both vertices 6 and 8 belong to the different vertices s4 and s6, we will perform the union operation. The union operation will form the new set shown as below:

s7 = {1, 2, 3, 4, 5, 6, 7, 8}

The last edge is left, which is (5, 7). Since both the vertices belong to the same set named s7, a cycle is formed.

How can we represent the sets graphically?

We have a universal set which is given below

U = {1, 2, 3, 4, 5, 6, 7, 8}

We will consider each edge one by one to represent graphically.

First, we consider the vertices 1 and 2, i.e., (1, 2) and represent them through graphically shown as below:

In the above figure, vertex 1 is the parent of vertex 2.

Now we consider the vertices 3 and 4, i.e., (3, 4) and represent them graphically shown as below:

In the above figure, vertex 3 is the parent of vertex 4.

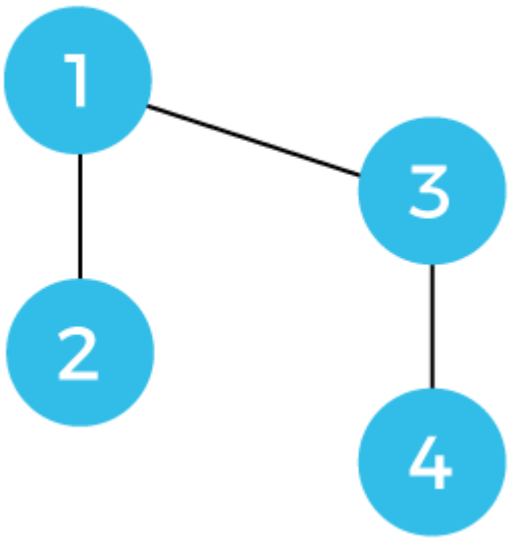
Consider the vertices 5 and 6, i.e., (5, 6) and represent them graphically shown as below:

In the above figure, vertex 5 is the parent of vertex 6.

Now, we consider the vertices 7 and 8, i.e., (7, 8) and represent them through graphically shown as below:

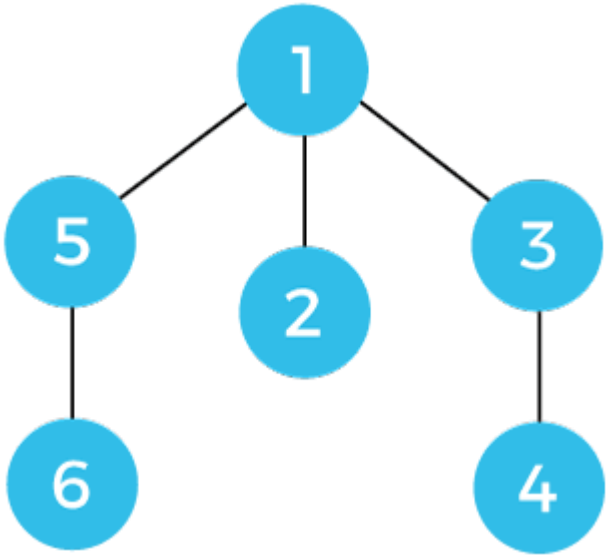
In the above figure, vertex 7 is the parent of vertex 8.

Now we consider the edge (2, 4). Since 2 and 4 belong to different sets, so we need to perform the union operation. In the above case, we observe that 1 is the parent of vertex 2 whereas vertex 3 is the parent of vertex 4. When we perform the union operation on the two sets, i.e., s1 and s2, then 1 vertex would be the parent of vertex 3 shown as below:



The next edge is (2, 5) having weight 6. Since 2 and 5 are in two different sets so we will perform the union operation. We make vertex 5 as a child of the vertex 1 shown as below:

We have chosen vertex 5 as a child of vertex 1 because the vertex of the graph having parent 1 is more than the graph having parent 5.



The next edge is (1, 3) having weight 7. Both vertices 1 and 3 are in the same set, so there is no need to perform any union operation. Since both the vertices belong to the same set; therefore, there is a cycle. We have detected a cycle, so we will consider the edges further.

How can we detect a cycle with the help of an array?

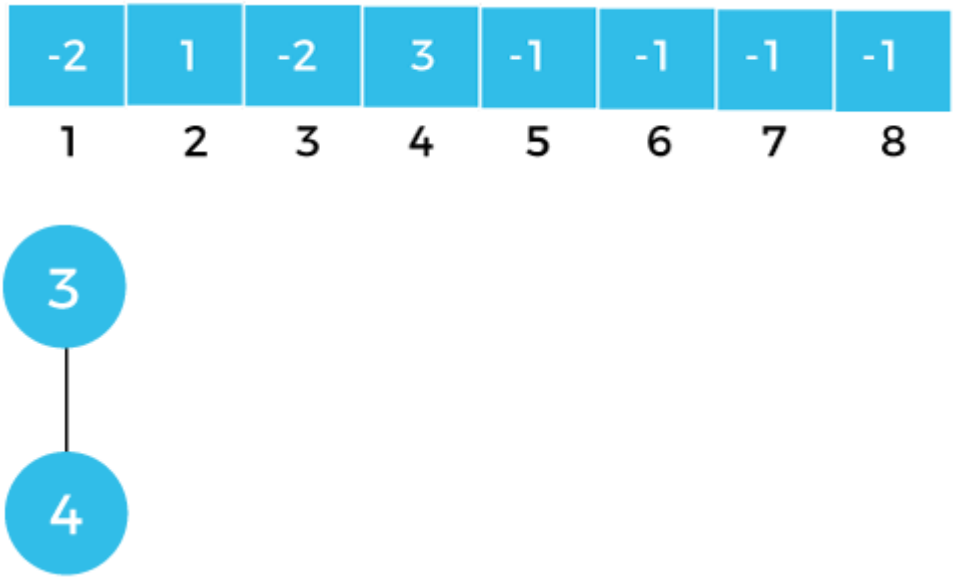
Consider the below graph:

The above graph contains the 8 vertices. So, we represent all these 8 vertices in a single array. Here, indices represent the 8 vertices. Each index contains a -1 value. The -1 value means the vertex is the parent of itself.

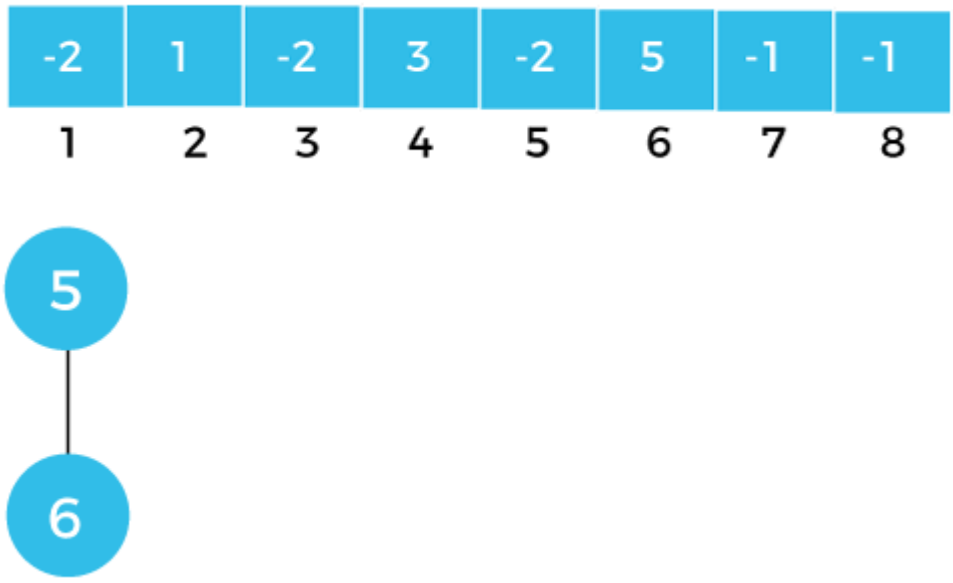
Now we will see that how we can represent the sets in an array.

First, we consider the edge (1, 2). When we find 1 in an array, we observe that 1 is the parent of itself. Similarly, vertex 2 is the parent of itself, so we make vertex 2 as the child of vertex 1. We add 1 at the index 2 as 2 is the child of 1. We add -2 at the index 1 where '-' sign that the vertex 1 is the parent of itself and 2 represents the number of vertices in a set.

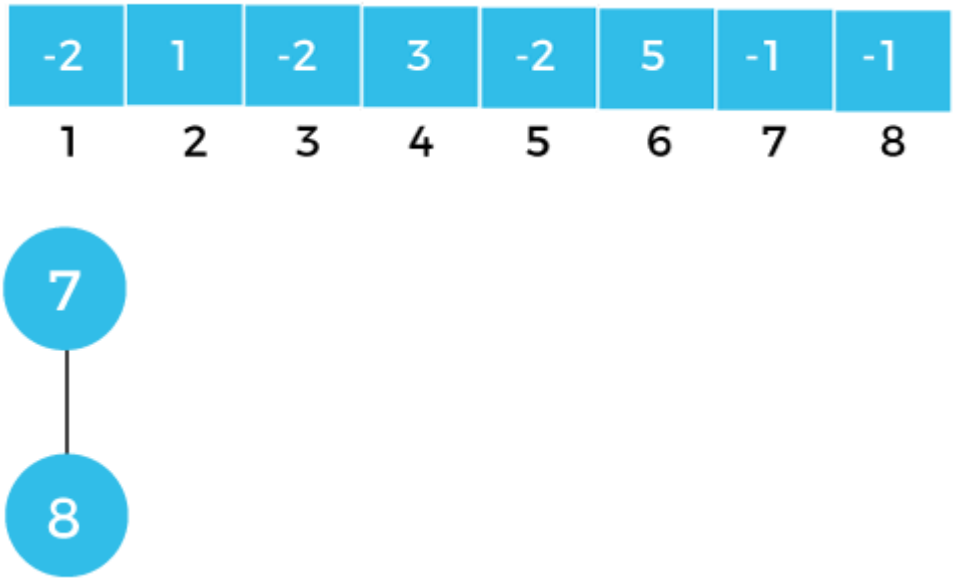
The next edge is (3, 4). When we find 3 and 4 in array; we observe that both the vertices are parent of itself. We make vertex 4 as the child of the vertex 3 so we add 3 at the index 4 in an array. We add -2 at the index 3 shown as below:



The next edge is (5, 6). When we find 5 and 6 in an array; we observe that both the vertices are parent of itself. We make 6 as the child of the vertex 5 so we add 5 at the index 6 in an array. We add -2 at the index 5 shown as below:

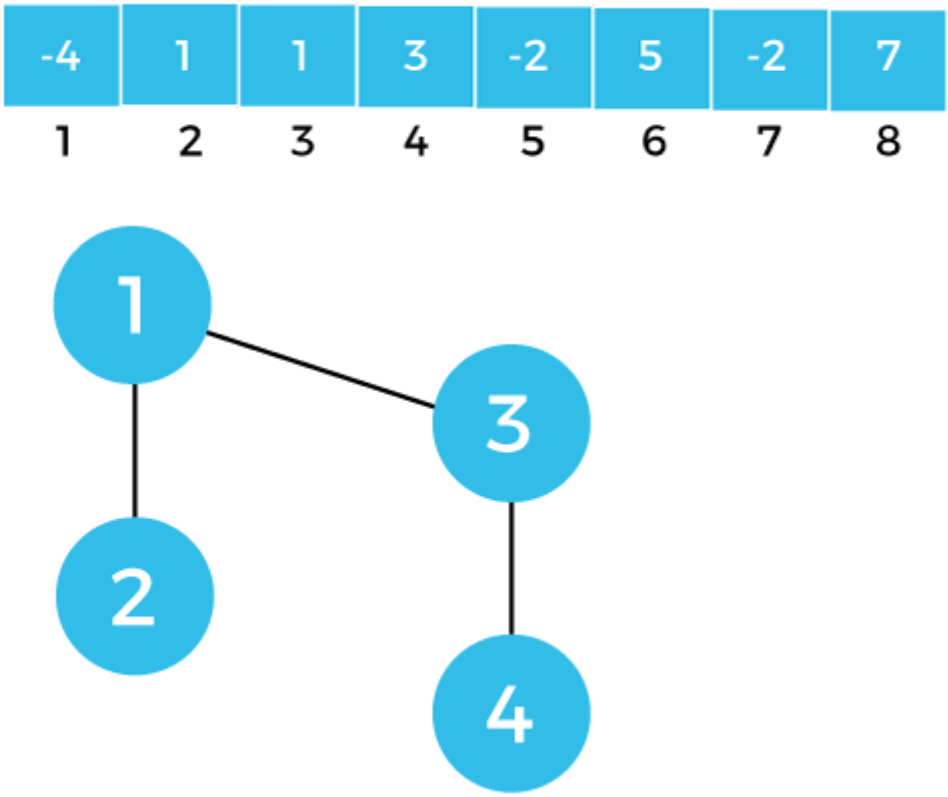


The next edge is (7, 8). Since both the vertices are parent of itself, so we make vertex 8 as the child of the vertex 7. We add 7 at the index 8 and -2 at the index 7 in an array shown as below:



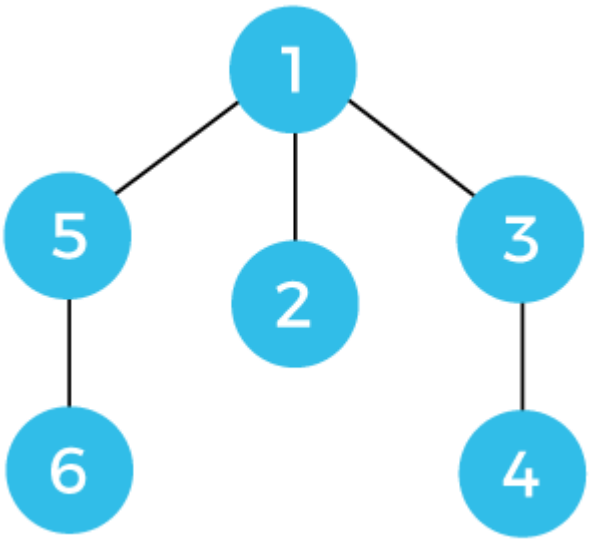
The next edge is (2, 4). The parent of the vertex 2 is 1 and the parent of the vertex is 3. Since both the vertices have different parent, so we make the vertex 3 as the child of vertex 1. We add 1 at the index 3. We add -4 at the index 1 as it contains 4 vertices.

Graphically, it can be represented as



The next edge is (2,5). When we find vertex 2 in an array, we observe that 1 is the parent of the vertex 2 and the vertex 1 is the parent of itself. When we find 5 in an array, we find -2 value which means vertex 5 is the parent of itself. Now we have to decide whether the vertex 1 or vertex 5 would become a parent. Since the weight of vertex 1, i.e., -4 is greater than the vertex of 5, i.e., -2, so when we apply the union operation then the vertex 5 would become a child of the vertex 1 shown as below:

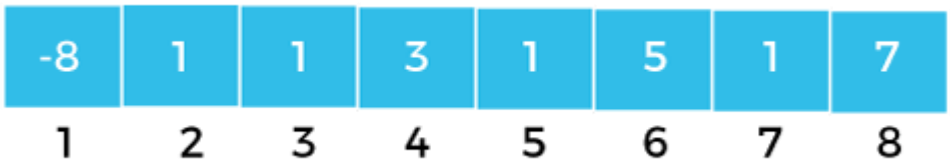
In an array, 1 would be added at the index 5 as the vertex 1 is now becomes a parent of vertex 5. We add -6 at the index 1 as two more nodes are added to the node 1.

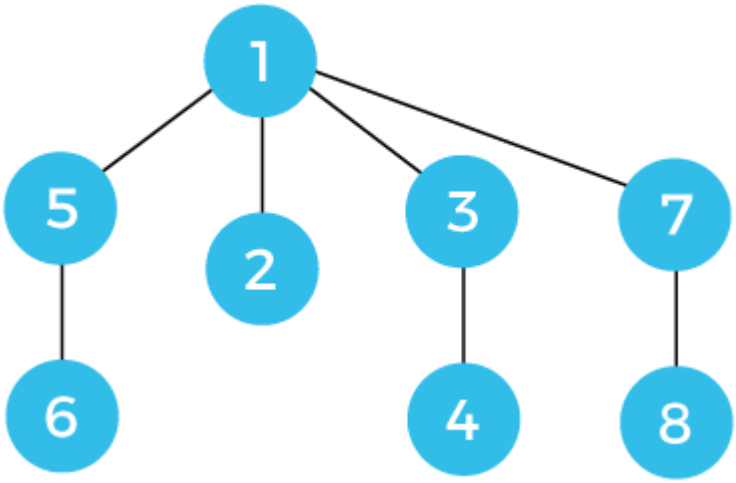


The next edge is (1,3). When we find vertex 1 in an array, we observe that 1 is the parent of itself. When we find 3 in an array, we observe that 1 is the parent of vertex 3. Therefore, the parent of both the vertices are same; so, we can say that there is a formation of cycle if we include the edge (1,3).

The next edge is (6,8). When we find vertex 6 in an array, we observe that vertex 5 is the parent of vertex 6 and vertex 1 is the parent of vertex 5. When we find 8 in an array, we observe that vertex 7 is the parent of the vertex 8 and 7 is the parent of itself. Since the weight of vertex 1, i.e., -6 is greater than the vertex 7, i.e., -2, so we make the vertex 7 as the child of the vertex and can be represented graphically as shown as below:

We add 1 at the index 7 because 7 becomes a child of the vertex 1. We add -8 at the index 1 as the weight of the graph now becomes 8.





The last edge to be included is (5, 7). When we find vertex 5 in an array, we observe that vertex 1 is the parent of the vertex 5. Similarly, when we find vertex 7 in an array, we observe that vertex 1 is the parent of vertex 7. Therefore, we can say that the parent of both the vertices is same, i.e., 1. It means that the inclusion (5,7) edge would form a cycle.

2. Backtracking:

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

When to use a Backtracking algorithm?

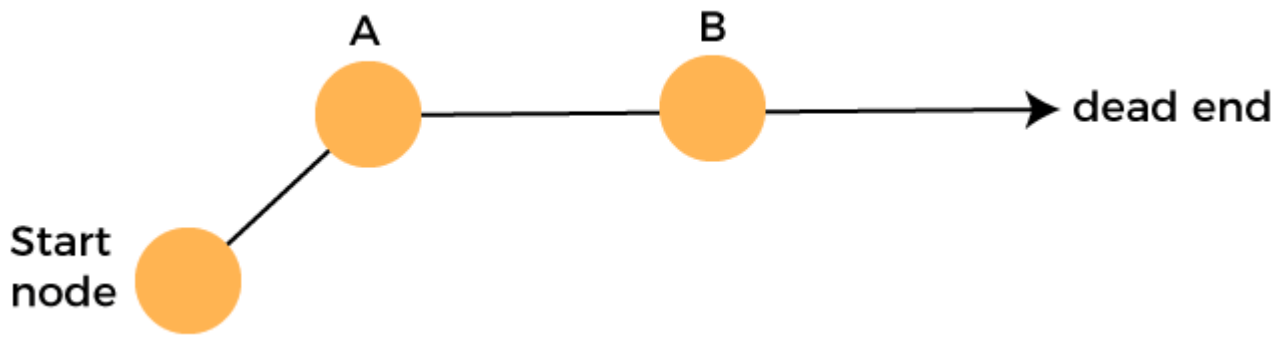
When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

- A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
- Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

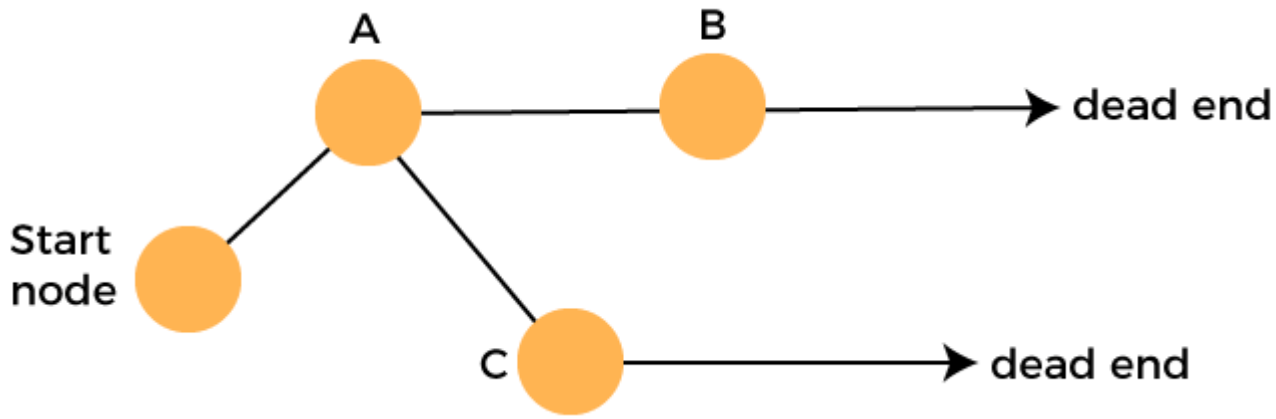
How does Backtracking work?

Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.

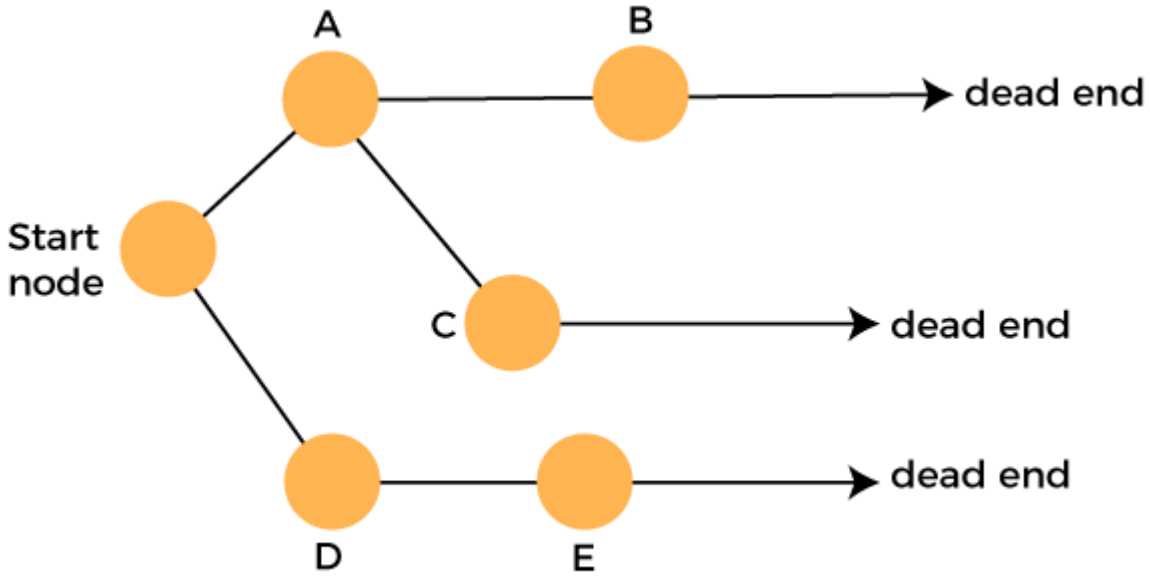


Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.

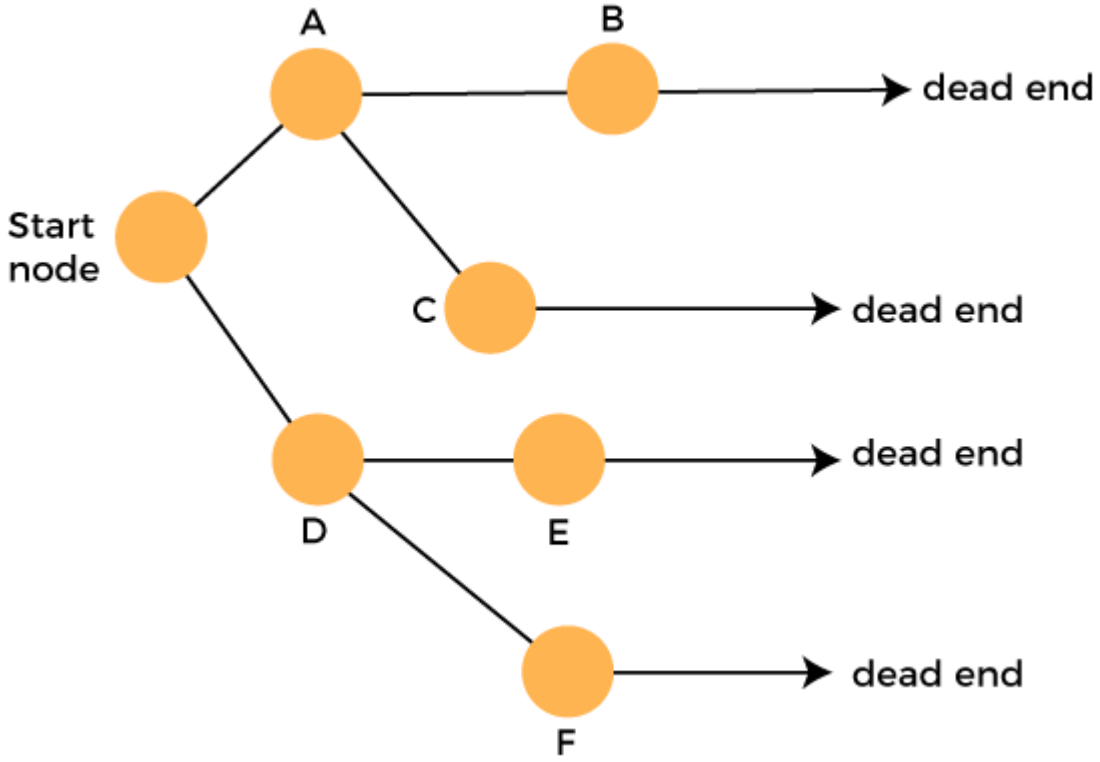


Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D

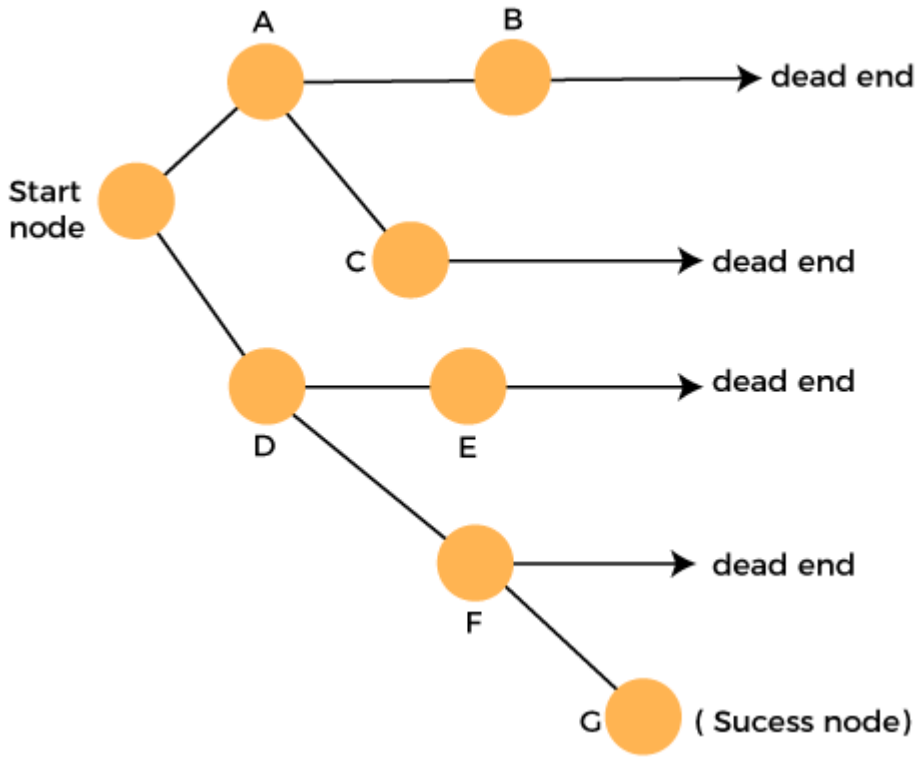
to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



The terms related to the backtracking are:

- **Live node:** The nodes that can be further generated are known as live nodes.
- **E node:** The nodes whose children are being generated and become a success node.
- **Success node:** The node is said to be a success node if it provides a feasible solution.
- **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

- **Implicit constraint:** It is a rule in which how each element in a tuple is related.
- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Applications of Backtracking

- N-queen problem

- Sum of subset problem
- Graph coloring
- Hamiliton cycle

3. N-Queens Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3.

N Queen problem demands us to place N queens on a N x N chessboard so that no queen can attack any other queen directly.

Problem Statement: We need to find out all the possible arrangements in which N queens can be seated in each row and each column so that all queens are safe. The queen moves in 8 directions and can directly attack in these 8 directions only.

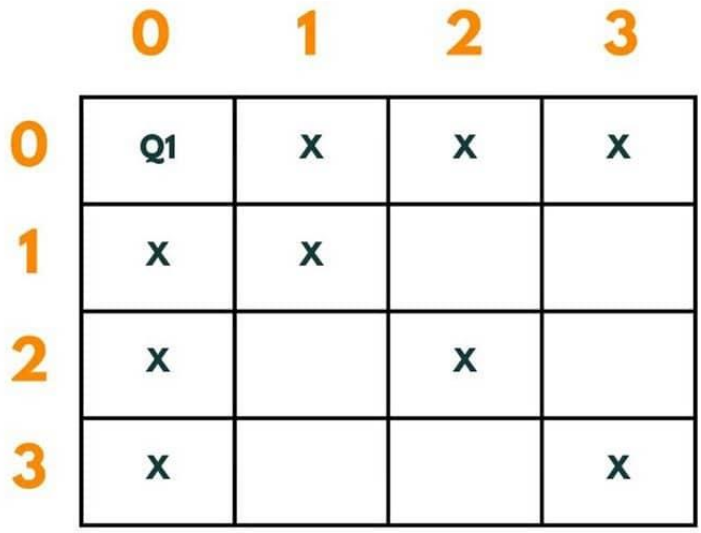
Before starting with the actual N queen problem, let's shorten it down to 4 - Queen problem, and then we will generalize it for N.

4 - Queen Problem

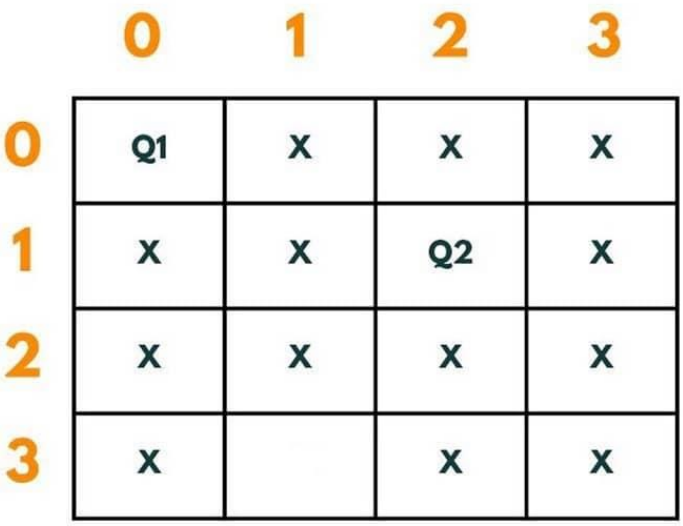
This problem demands us to put 4 queens on 4 X 4 chessboard in such a way that one queen is present in each row and column and no queen can attack any other queen directly. this means no 2 or more queens can be placed in the same diagonal or row or column.

Let's try to put queens Q1, Q2, Q3, and Q4 in the above present chessboard. The first queen i.e. Q1 can be put anywhere on the chessboard as there is no other queen present on the board and hence no restrictions. Therefore putting Q1 at position (0,0). So the path so far is|

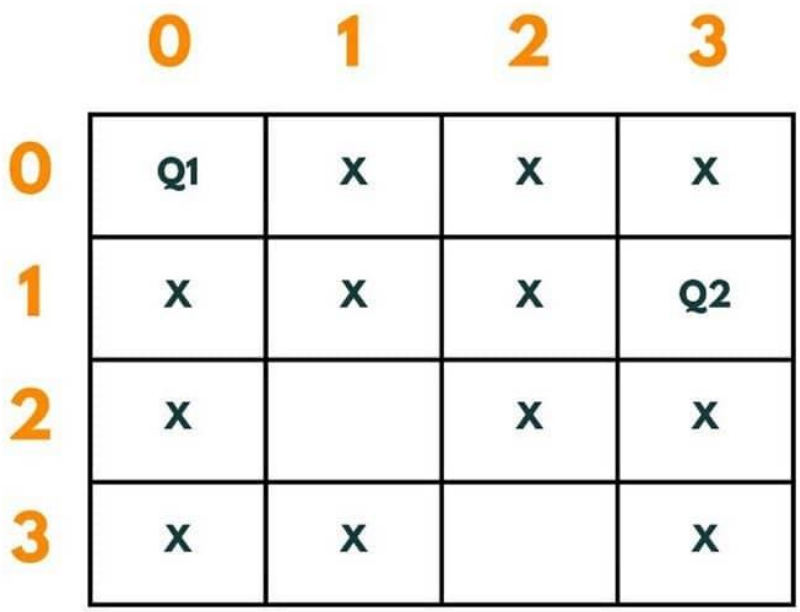
(



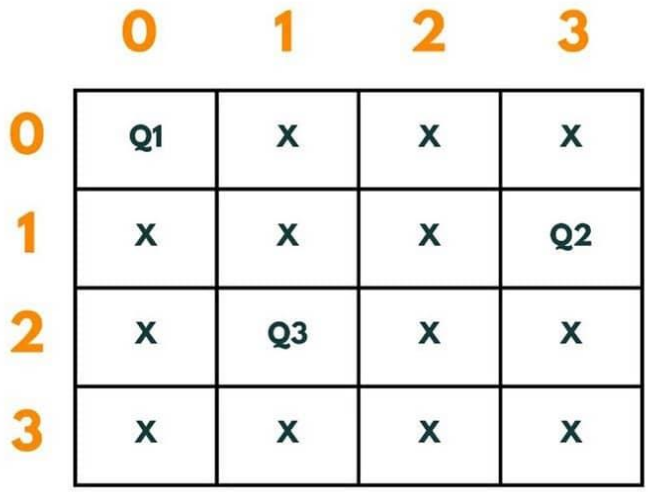
When Q1 has been placed there are some places where the next queens can't be placed to fulfill given conditions. So to put queen Q2 in the second row we have positions - (1,2) and (1,3). Let's put it at (1,2).The path so far is | (0,0) -> (1,2)|



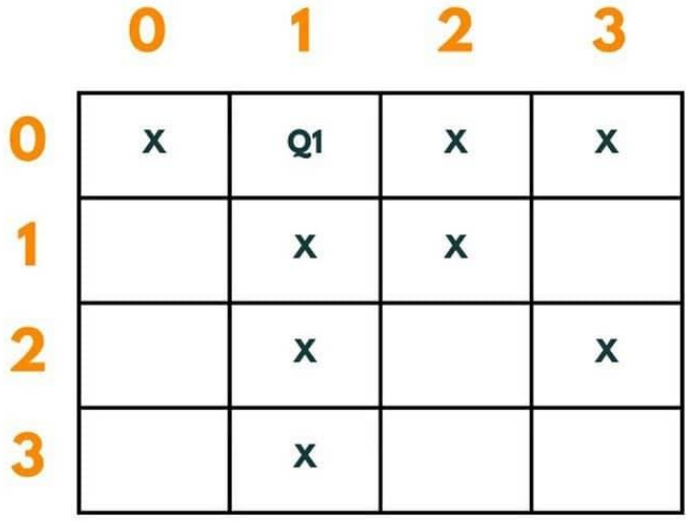
Now this placement of Q2 blocks all the boxes of row 3 and hence there is no way to put Q3. If we put it at (2,0) or (2,2), Q1 will attack it, and at (2,1) and (2,3) Q2 attacks it. Therefore we backtrack from here and revisit the previous solution by readjusting the position of Q2. So instead of putting it at (1,2), we put it at (1,3). The path so far is | (0,0) -> (1,3)|



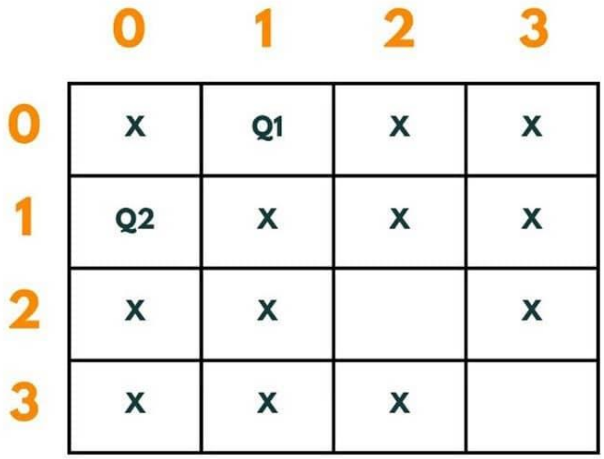
We put Q3 at (2,1). Hence, the path so far is | (0,0) -> (1,3) -> (2,1)|.



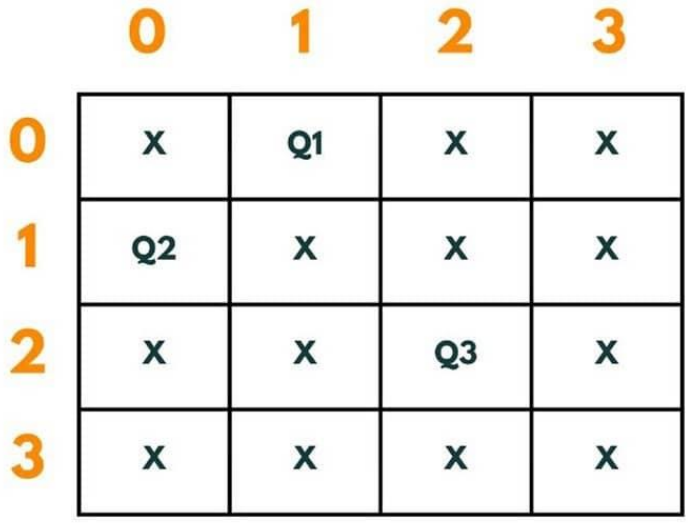
Now again the same problem occurs, there left no box to place Q4. There was only 1 way to place Q3 and all placements of Q2 have been explored, so now we come to Q1 for re-adjustment. We move it from (0,0) to (0,1). The path so far is | (0,1)|.



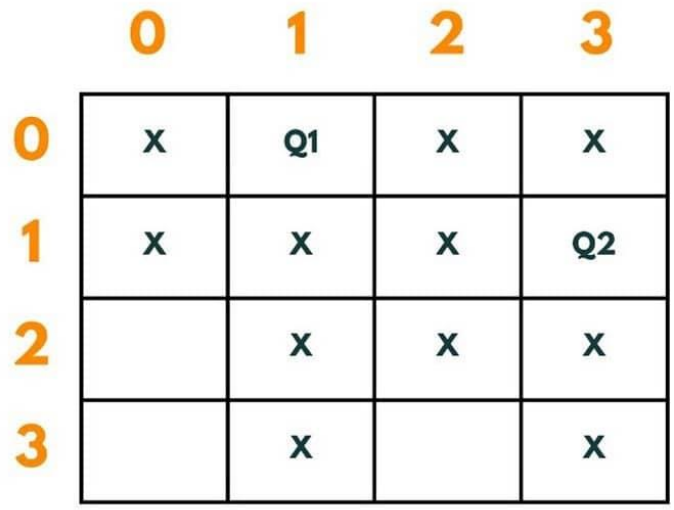
We put Q2 at (1,0). The path so far is | (0,1) -> (1,0)|.



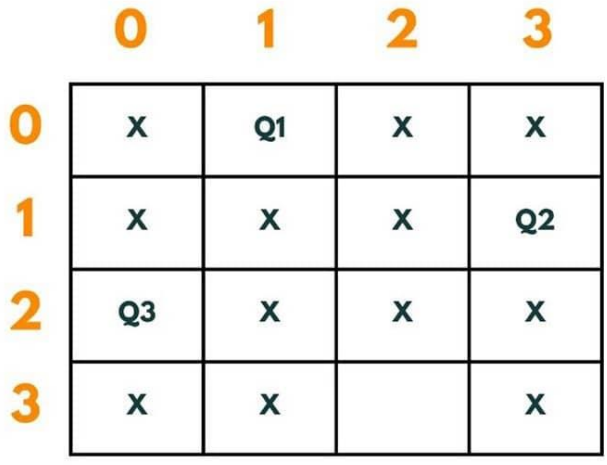
Q3 is put at (2,2). The path so far is | (0,1) -> (1,0) -> (2,2)|.



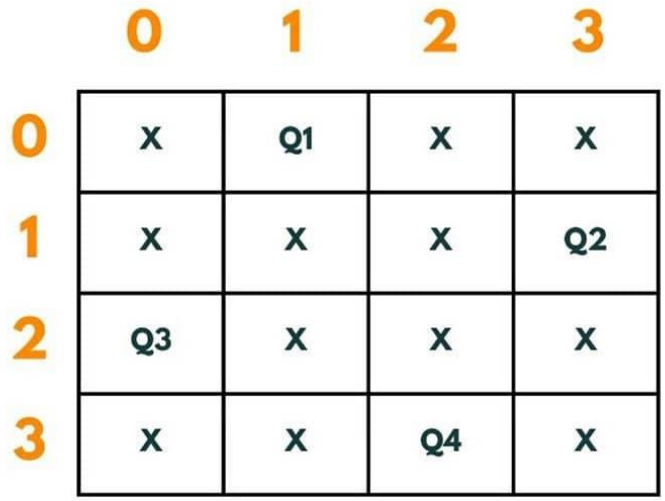
Now again there is no space left for placement of Q4 in row 4. Therefore we again backtrack and readjust position of Q2 from (1,0) to (1,3).The path so far is | (0,1) -> (1,3)|.



Q3 is put at (2,0). The path so far is | (0,1) -> (1,0) -> (2,0)|.

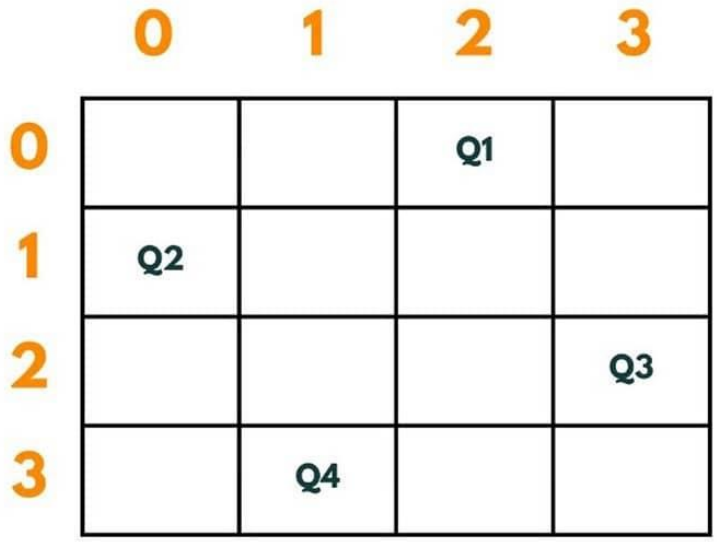


We put Q4 at (3,2). The path so far is | (0,1) -> (1,0) -> (2,0) -> (3,2)|.



Therefore through backtracking, we reached a solution where 4 queens are put in each row and column so that no queen is attacking any other on a 4 X 4 chessboard.

Another solution can be:



The Easiest Backtracking Approach

The most intuitive approach to solve this problem could be to find out all the possibilities until we find the appropriate arrangement. So whenever we will place any queen on the board we will check if any other queen has been placed in that row/column/diagonal.

N Queen Problem Algorithm

1. We create a board of N x N size that stores characters. It will store 'Q' if the queen has been placed at that position else '.'
2. We will create a recursive function called "solve" that takes board and column and all Boards (that stores all the possible arrangements) as arguments. We will pass the column as 0 so that we can start exploring the arrangements from column 1.
3. In solve function we will go row by row for each column and will check if that particular cell is safe or not for the placement of the queen, we will do so with the help of isSafe() function.
4. For each possible cell where the queen is going to be placed, we will first check isSafe() function.
5. If the cell is safe, we put 'Q' in that row and column of the board and again call the solve function by incrementing the column by 1.
6. Whenever we reach a position where the column becomes equal to board length, this implies that all the columns and possible arrangements have been explored, and so we return.
7. Coming on to the boolean isSafe() function, we check if a queen is already present in that row/ column/upper left diagonal/lower left diagonal/upper right diagonal /lower right diagonal. If the queen is present in any of the directions, we return false. Else we put board[row][col] = 'Q' and return true.

4. Sum of Subset Problem:

Given a **set[]** of non-negative integers and a value **sum**, the task is to print the subset of the given set whose sum is equal to the given **sum**.

Examples:

Input: *set[] = {1,2,1}, sum = 3*
Output: *[1,2],[2,1]*
Explanation: *There are subsets [1,2],[2,1] with sum 3.*
Input: *set[] = {3, 34, 4, 12, 5, 2}, sum = 30*
Output: *[]*
Explanation: *There is no subset that add up to 30.*

Problem statement:

Let, S = {S1 …. Sn} be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.It is always convenient to sort the set's elements in ascending order. That is, S1 ≤ S2 ≤…. ≤ Sn

Algorithm:

Let, S is a set of elements and m is the expected sum of subsets. Then:

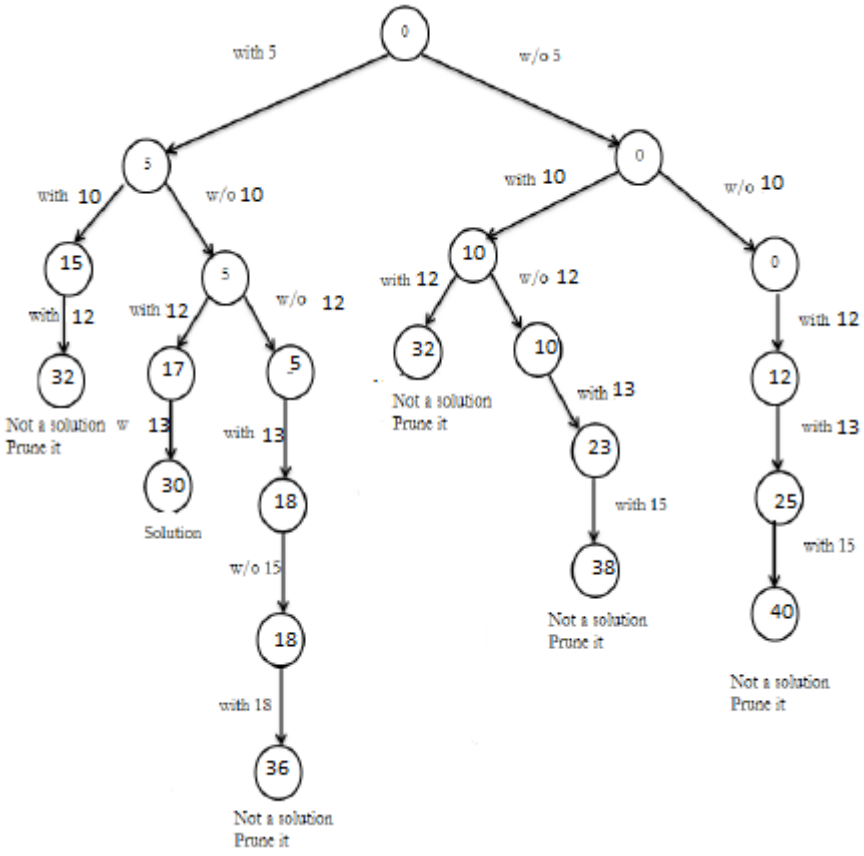
1. Start with an empty set.
2. Add to the subset, the next element from the list.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible then repeat step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Example: Solve following problem and draw portion of state space tree M=30,W ={5, 10, 12, 13, 15, 18}

Solution:

Initially subset = {}	**Sum = 0**	**Description**
5	5	Then add next element.
5, 10	15 i.e. 15 < 30	Add next element.
5, 10, 12	27 i.e. 27 < 30	Add next element.
5, 10, 12, 13	40 i.e. 40 < 30	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 15	42	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 18	45	Sum exceeds M = 30. Hence backtrack.
5, 12, 13	30	Solution obtained as M = 30

The state space tree is shown as below in figure. {5, 10, 12, 13, 15, 18}

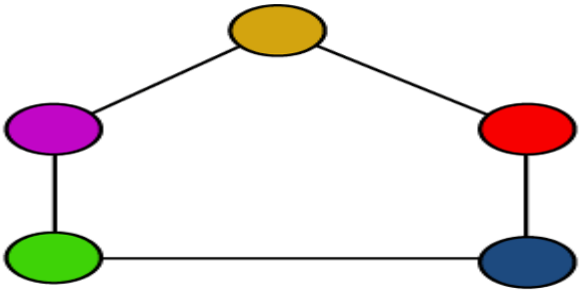


5. Graph coloring

Graph coloring can be described as a process of assigning colors to the vertices of a graph. In this, the same color should not be used to fill the two adjacent vertices. We can also call graph coloring as Vertex Coloring. In graph coloring, we have to take care that a graph must not contain any edge whose end vertices are colored by the same color. This type of graph is known as the Properly colored graph.

Example of Graph coloring

In this graph, we are showing the properly colored graph, which is described as follows:



The above graph contains some points, which are described as follows:

- The same color cannot be used to color the two adjacent vertices.
- Hence, we can call it as a properly colored graph.

Applications of Graph coloring

There are various applications of graph coloring. Some of their important applications are described as follows:

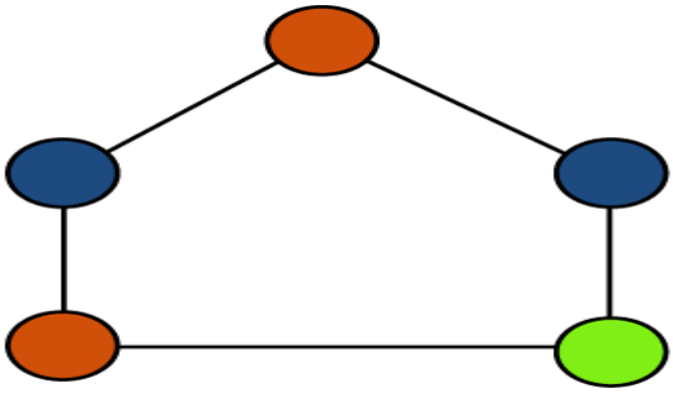
- Assignment
- Map coloring
- Scheduling the tasks
- Sudoku
- Prepare time table
- Conflict resolution

Chromatic Number

The chromatic number can be described as the minimum number of colors required to properly color any graph. In other words, the chromatic number can be described as a minimum number of colors that are needed to color any graph in such a way that no two adjacent vertices of a graph will be assigned the same color.

Example of Chromatic number:

To understand the chromatic number, we will consider a graph, which is described as follows:



The above graph contains some points, which are described as follows:

- The same color is not used to color the two adjacent vertices.
- The minimum number of colors of this graph is 3, which is needed to properly color the vertices.

- Hence, in this graph, the chromatic number = 3
- If we want to properly color this graph, in this case, we are required at least 3 colors.

Approach 1: Brute Force

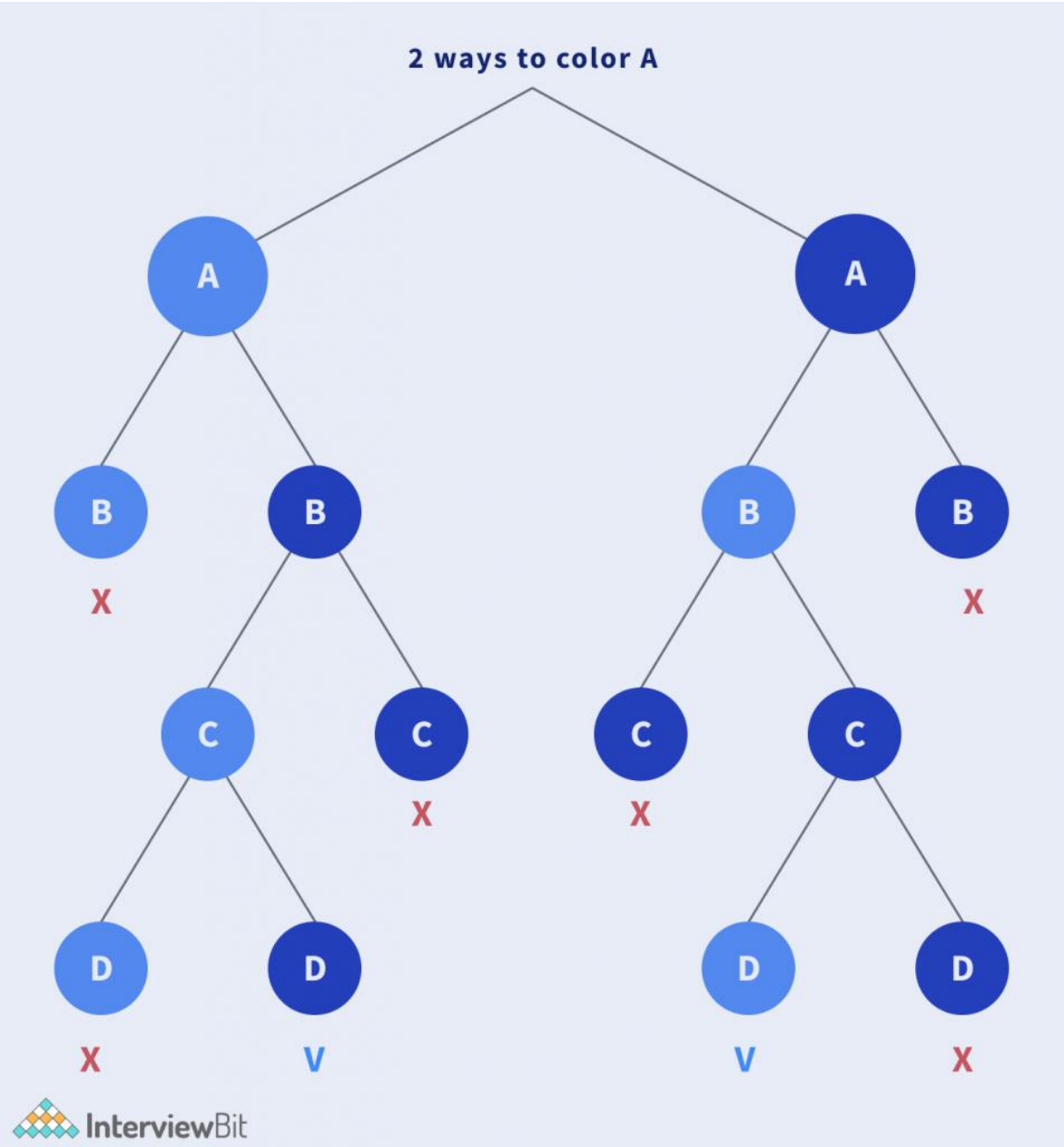
- The simplest approach to solve this problem would be to generate all possible combinations (or configurations) of colours.
- After generating a configuration, check if the adjacent vertices have the same colour or not. If the conditions are met, add the combination to the result and break the loop.
- Since each node can be coloured by using any of the **M** colours, the total number of possible colour configurations are mV. The complexity is exponential which is very huge.

- **Time Complexity:** O(M^V), where M is the total colours needed and V is the total vertices
- **Space Complexity:** O(V), as extra space is used for colouring vertices.

Approach 2: Backtracking

In the previous approach, trying and checking every possible combination was tedious and had an exponential time complexity.
Some of the permutation calculations were unnecessary but were calculated again and again. Therefore, the idea is to use a **backtracking** approach to solve the problem.

In this approach, the idea is to color a vertex and while coloring any adjacent vertex, choose a different color. Similarly, color every possible vertex following the restrictions, till any further vertex is left coloring. In any case, if all adjacent vertices for a given vertex are colored, then backtrack and change color.



If after coloring, if we return back to the same vertex that was started with and all colors are used, then more colors are needed. Hence, return **False**.

Algorithm

- Consider a color and check if it is valid i.e. from the given vertex check whether its adjacent vertices have been coloured with the same color.
- If true, pick a different colour, else continue colouring the vertices.
- If no other color is left un-used, then backtrack.