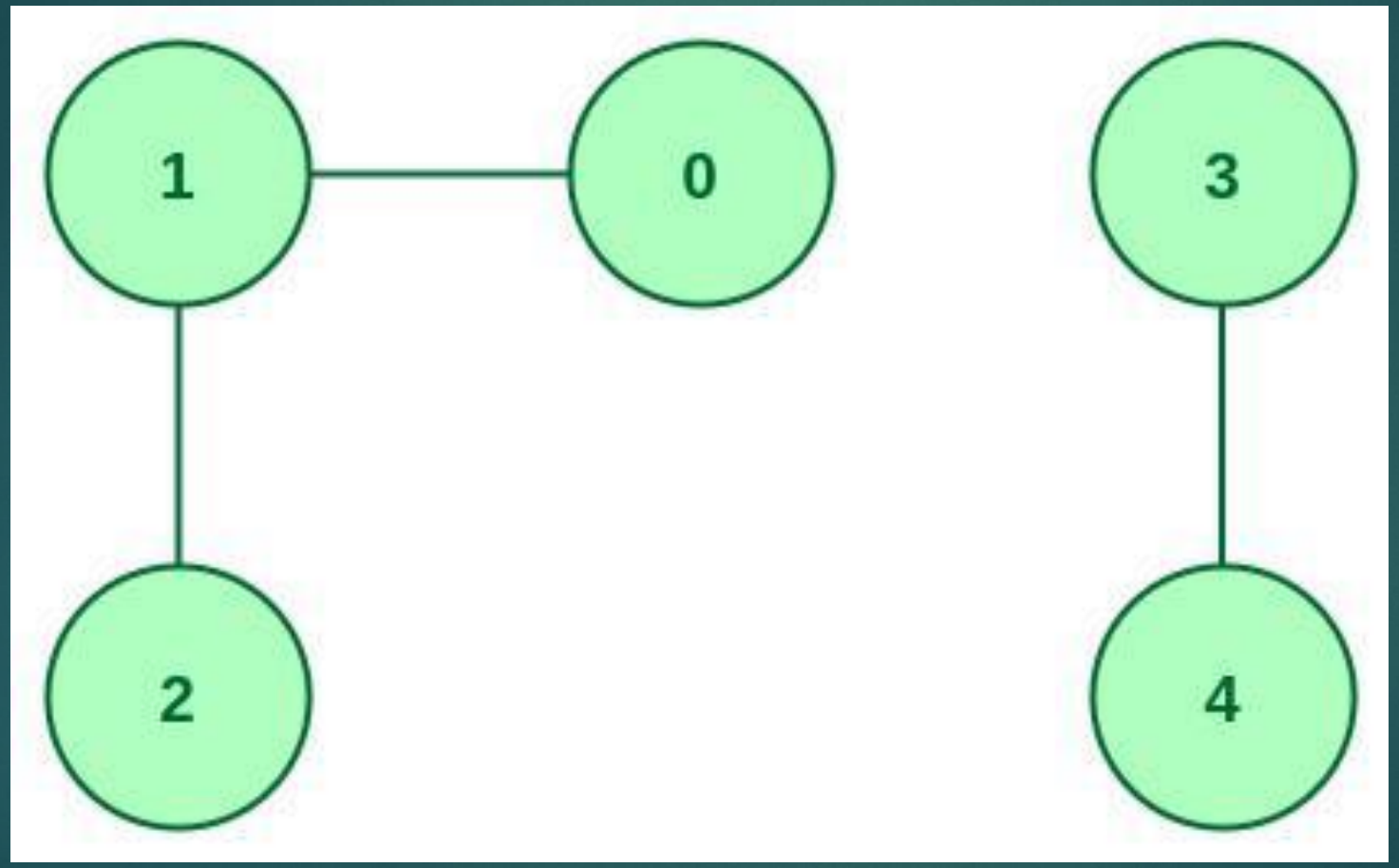
- *Connected component in an undirected graph refers to a group of vertices that are connected to each other through edges, but not connected to other vertices outside the group.*
- For example in the graph shown below, {0, 1, 2} form a connected component and {3, 4} form another connected component.


1
0
3
2
4

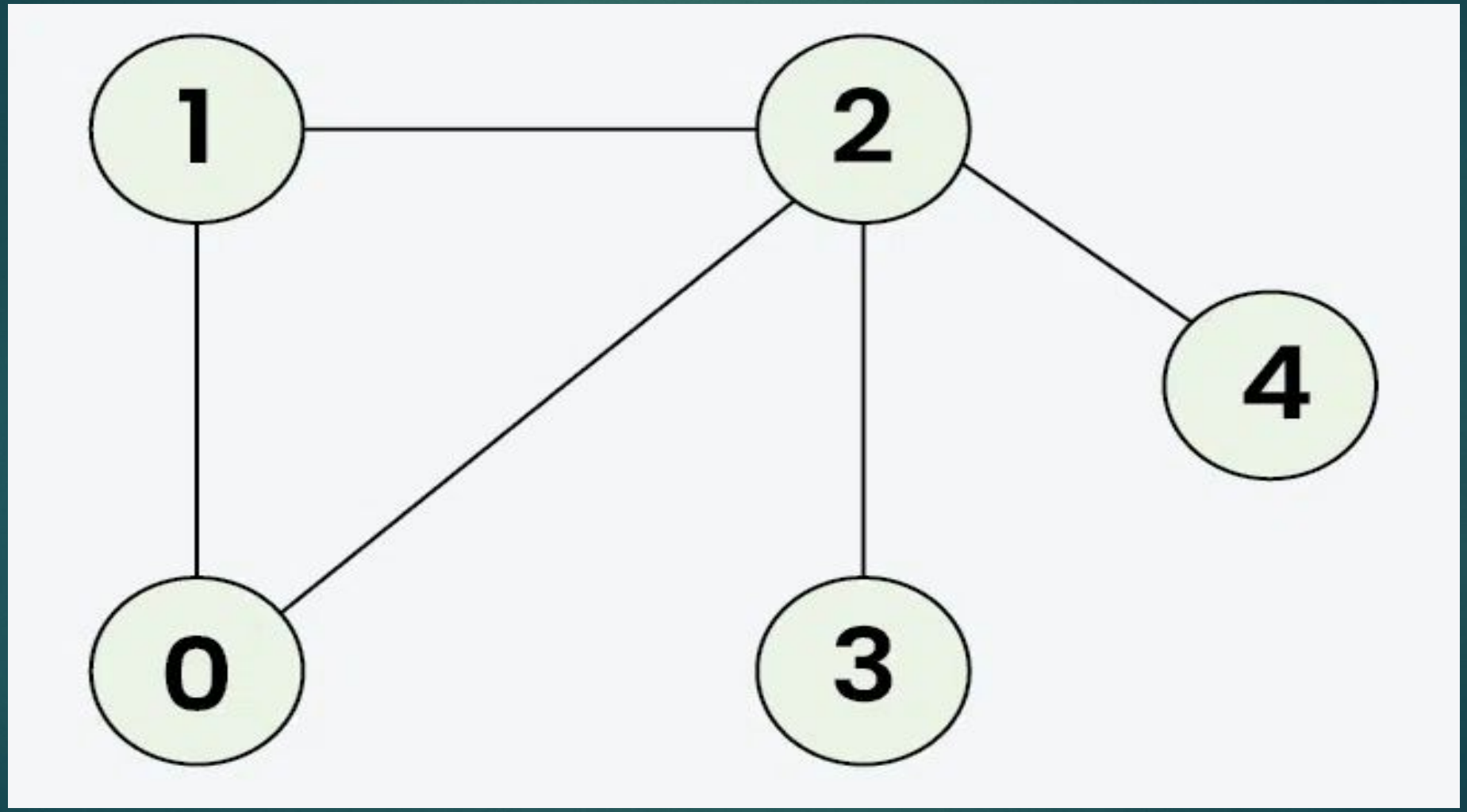
- **Characteristics of Connected Component:**
- A connected component is a set of vertices in a graph that are connected to each other.
- A graph can have multiple connected components.
- Inside a component, each vertex is reachable from every other vertex in that component.

- **How to identify Connected Component:**
- There are several algorithms to identify Connected Components in a graph. The most popular ones are:
- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Union-Find Algorithm (also known as Disjoint Set Union)

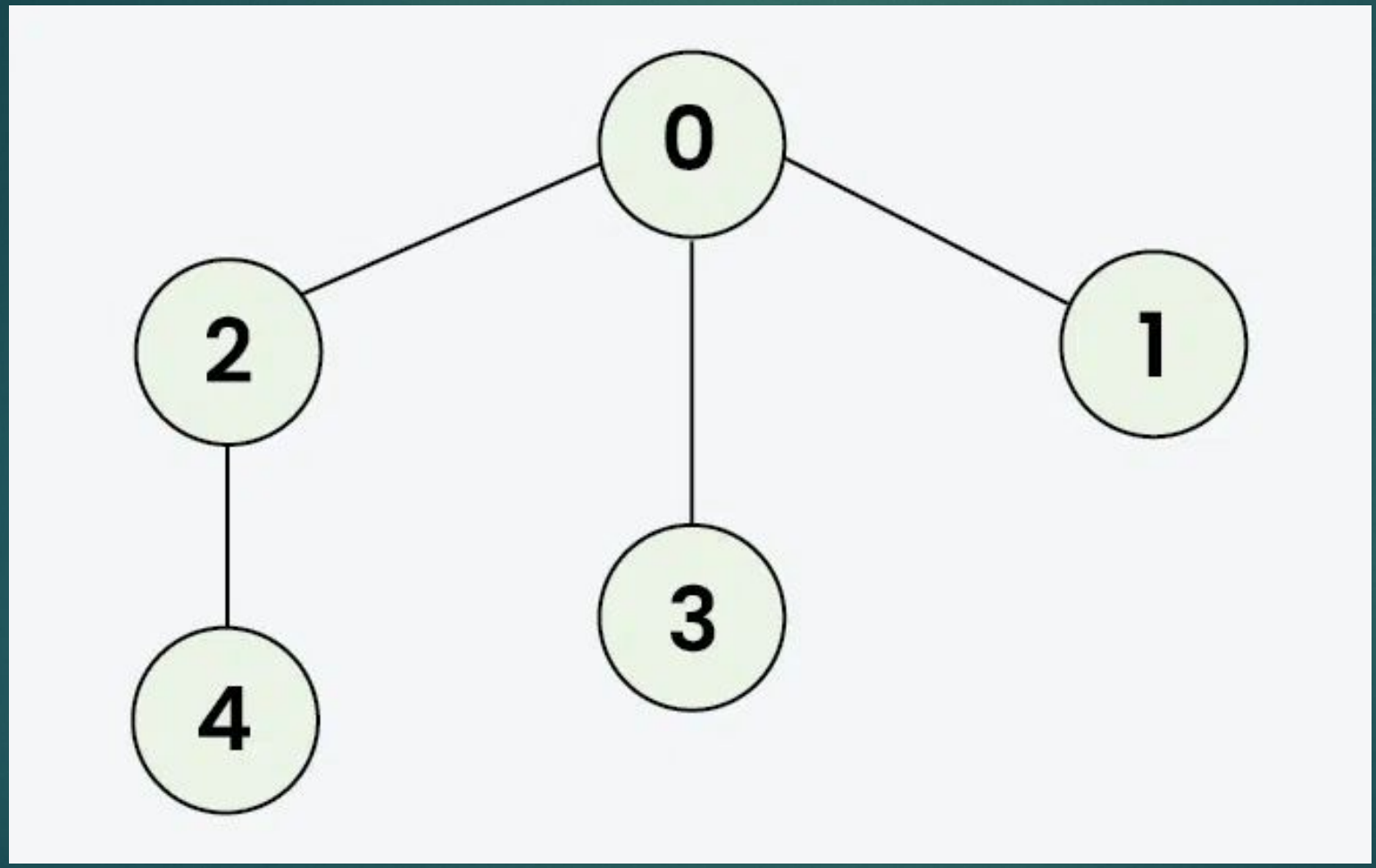
- **Applications of Connected Component:**
- Graph Theory: It is used to find subgraphs or clusters of nodes that are connected to each other.
- Computer Networks: It is used to discover clusters of nodes or devices that are linked and have similar qualities, such as bandwidth.
- Image Processing: Connected components also have usage in image processing.

- **Depth First Traversal (or DFS)** for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

- ***Input:*** *V = 5, E = 5, edges = {{1, 2}, {1, 0}, {0, 2}, {2, 3}, {2, 4}},*
- *source = 1*


1
2
4
0
3

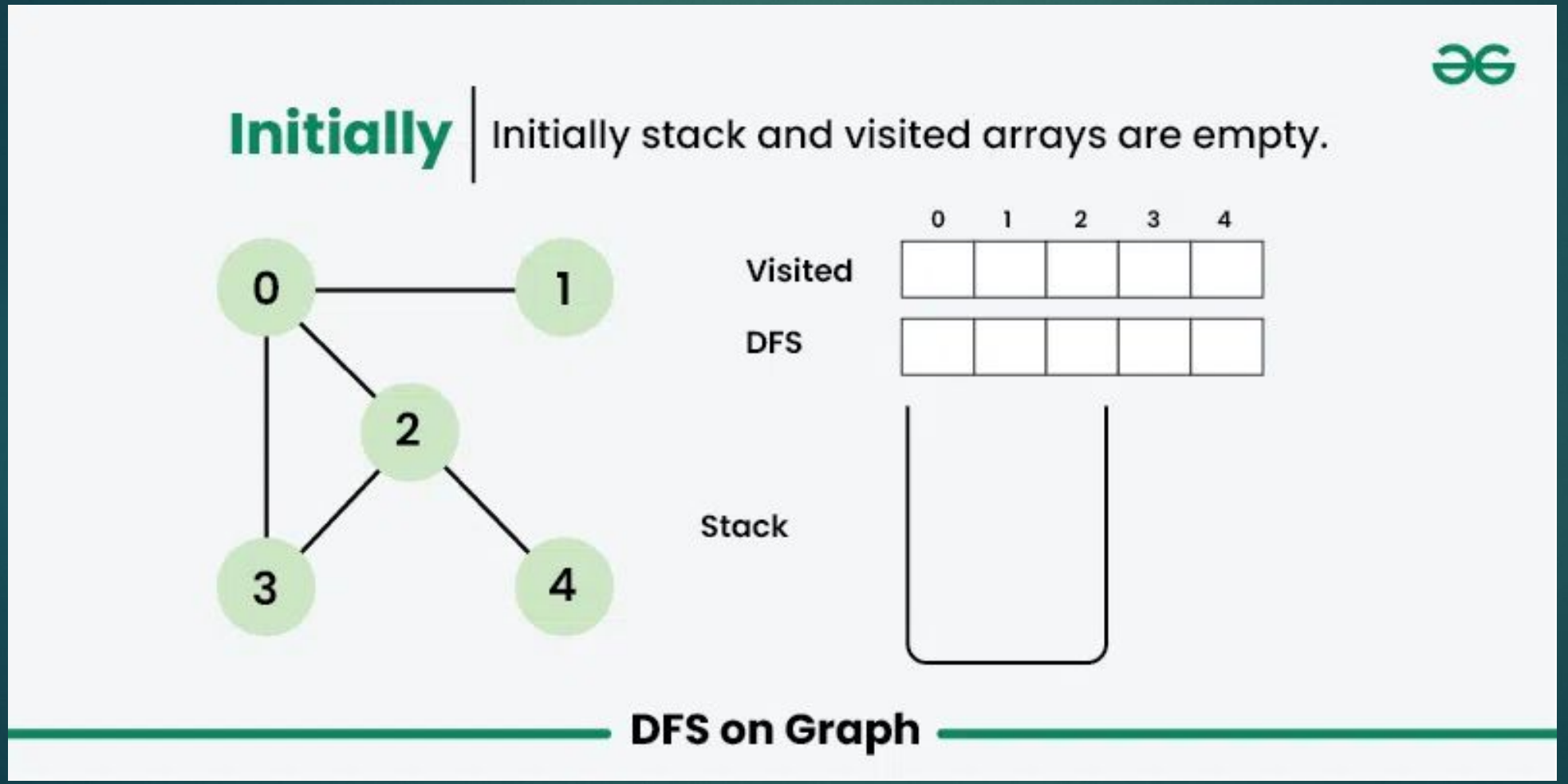
- ***Output:*** *1 2 0 3 4*
  Explanation: *DFS Steps:*
- *Start at 1: Mark as visited. Output: 1*
- *Move to 2: Mark as visited. Output: 2*
- *Move to 0: Mark as visited. Output: 0 (backtrack to 2)*
- *Move to 3: Mark as visited. Output: 3 (backtrack to 2)*
- *Move to 4: Mark as visited. Output: 4 (backtrack to 1)*


0
2
1
3
4

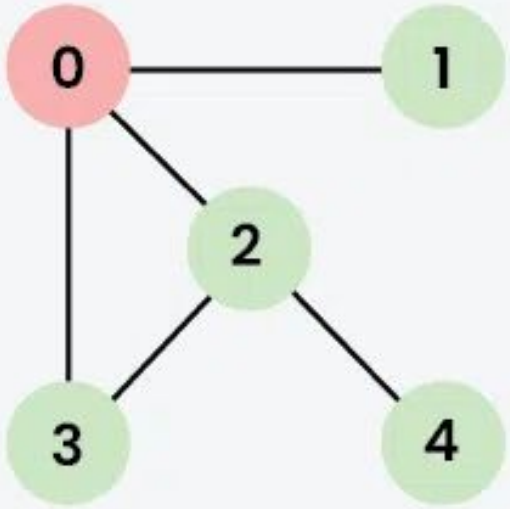
- ***Input:*** *V = 5, E = 4, edges = {{0, 2}, {0, 3}, {0, 1}, {2, 4}}, source = 0*

- ***Output:*** *0 2 4 3 1*
  Explanation: *DFS Steps:*
- *Start at 0: Mark as visited. Output: 0*
- *Move to 2: Mark as visited. Output: 2*
- *Move to 4: Mark as visited. Output: 4 (backtrack to 2, then backtrack to 0)*
- *Move to 3: Mark as visited. Output: 3 (backtrack to 0)*
- *Move to 1: Mark as visited. Output: 1*

working of **Depth First Search**



01 Step | Visit 0 and put its adjacent nodes which are not visited yet into the stack.

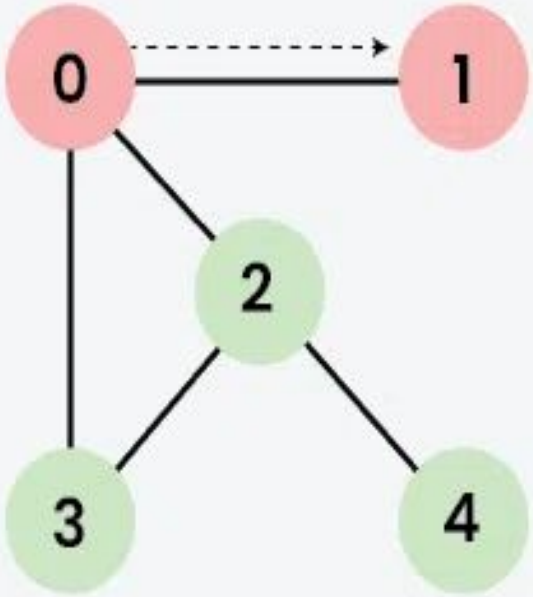


	0	1	2	3	4
Visited	T	T	T	T	
DFS	0				

Stack

DFS on Graph

02 Step | Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

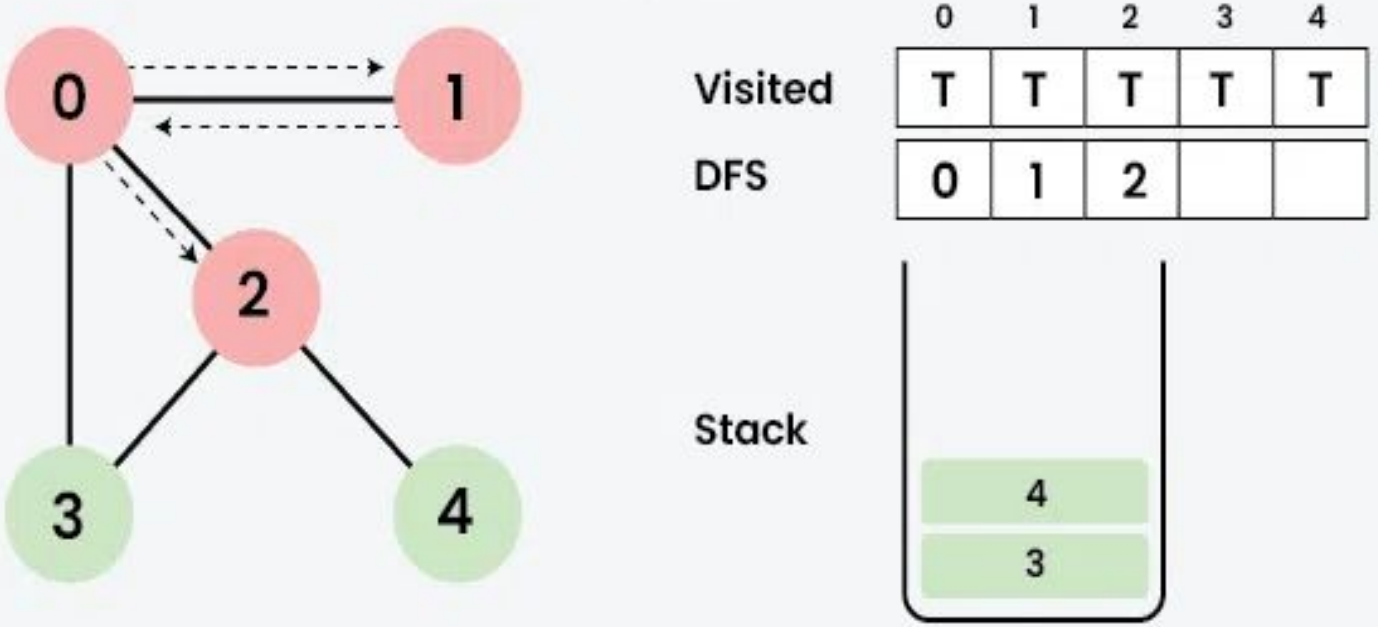


	0	1	2	3	4
Visited	T	T	T	T	
DFS	0	1			

Stack

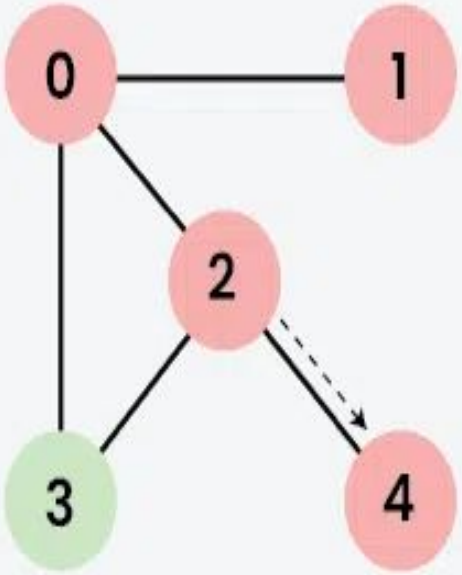
DFS on Graph

03 Step | Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



DFS on Graph

04 Step | Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

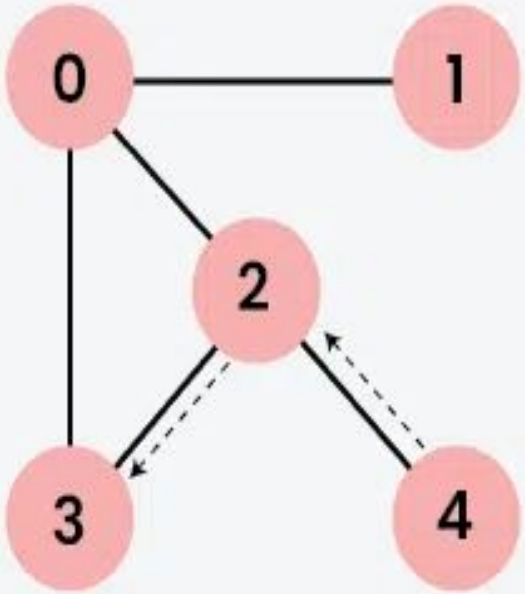


	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	

Stack

DFS on Graph

05 Step | Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



	0	1	2	3	4
Visited	T	T	T	T	T
DFS	0	1	2	4	3

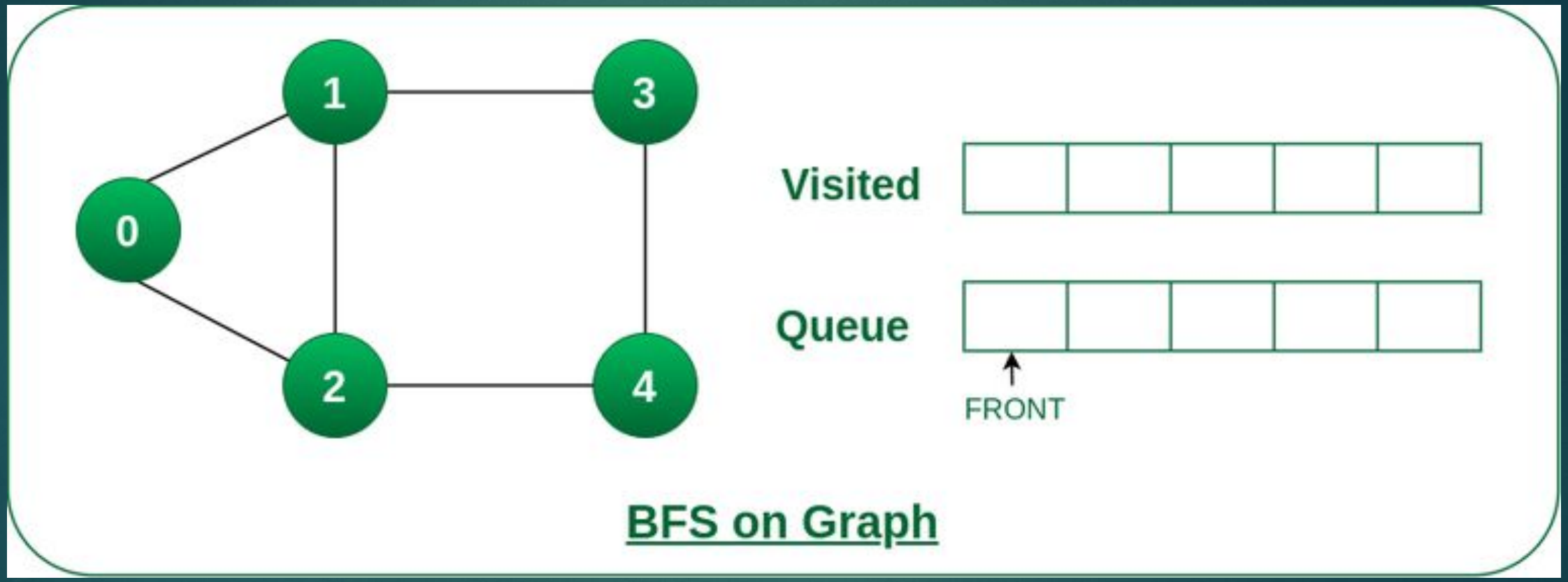
Stack

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

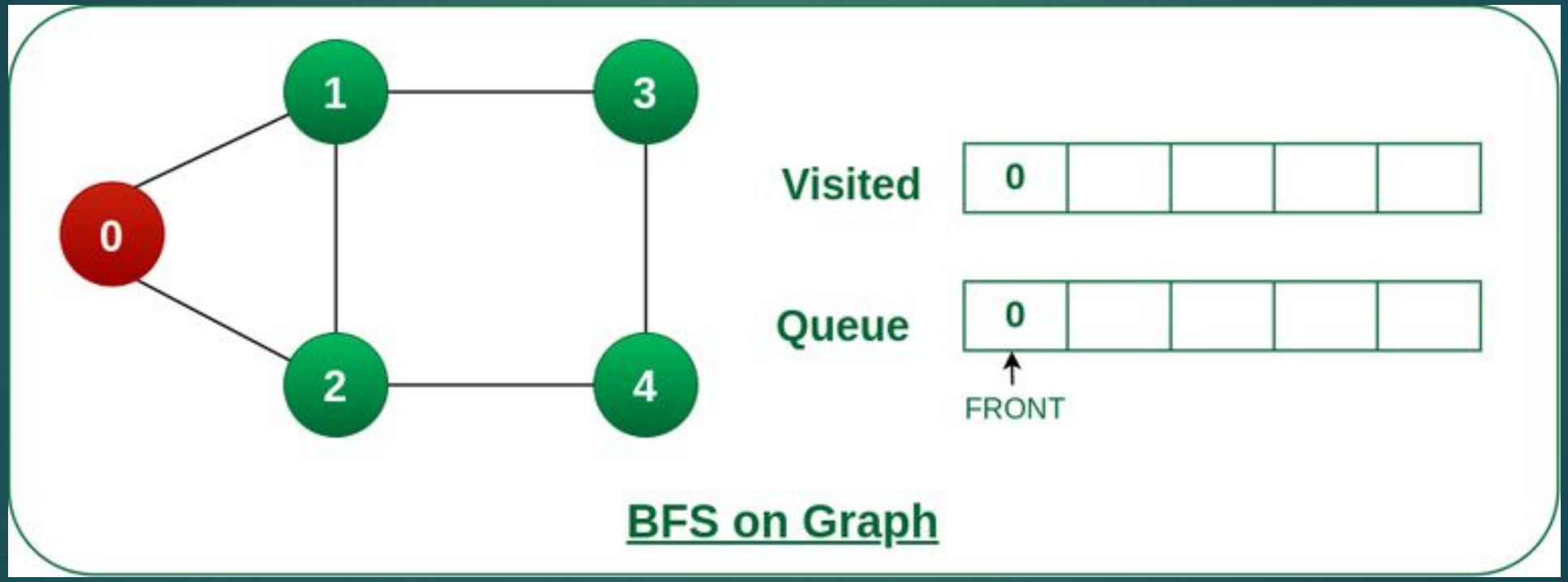
Breadth First Search

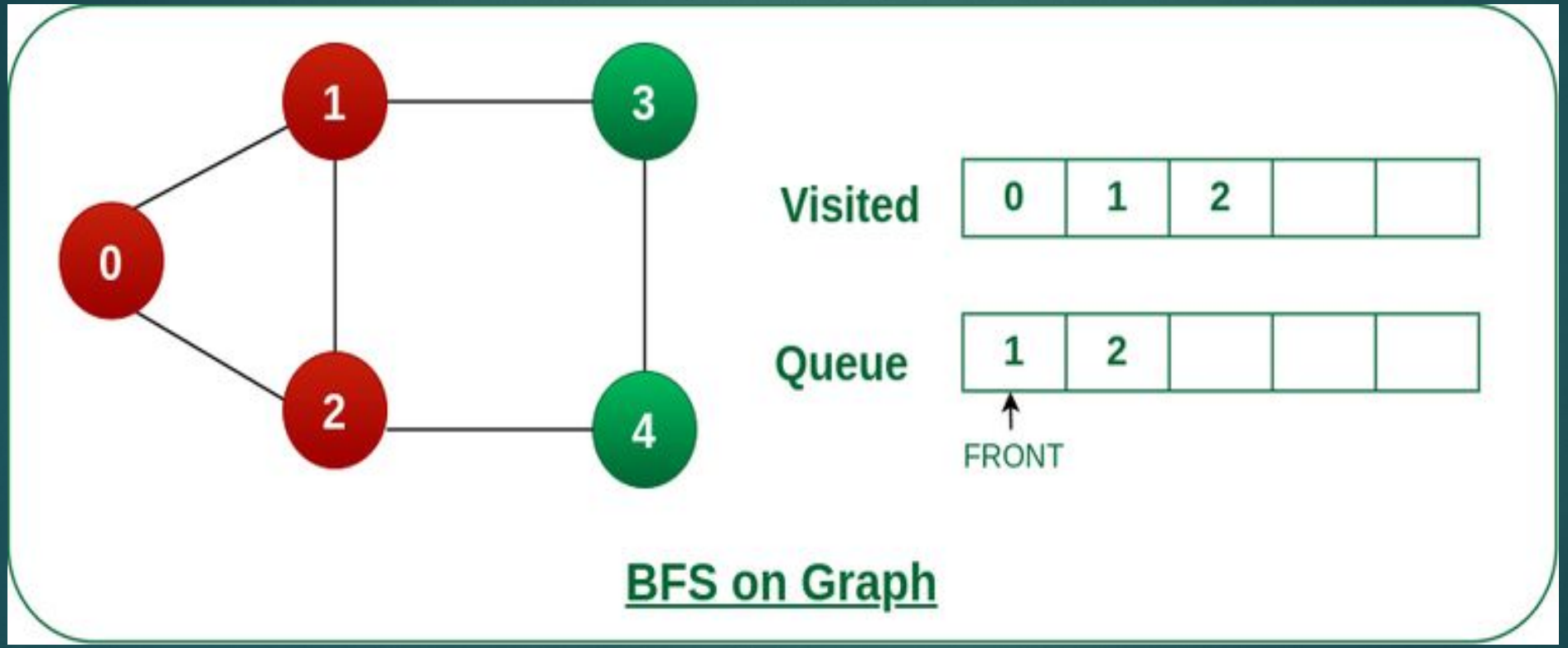
- **Breadth First Search (BFS)** is a fundamental **graph traversal algorithm.** It begins with a node, then first traverses all its adjacent.
- We mainly traverse vertices level by level. A lot of popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS.
- BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

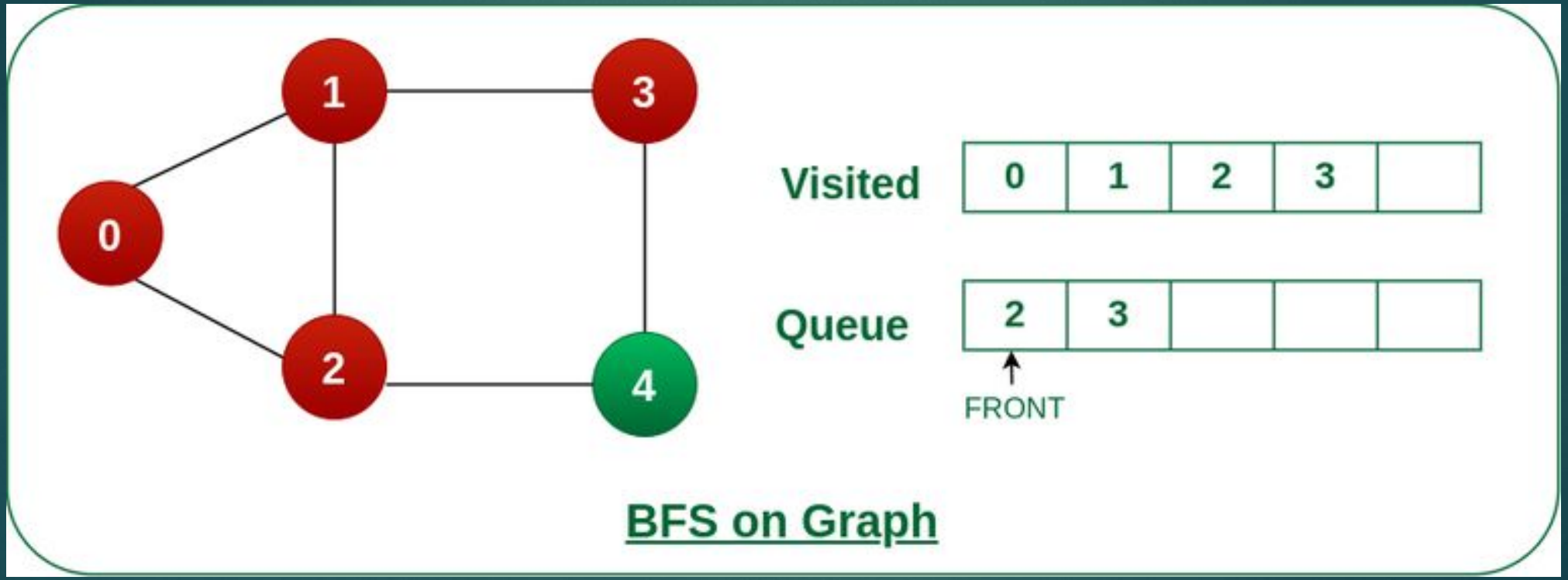
How Does the BFS Algorithm Work?

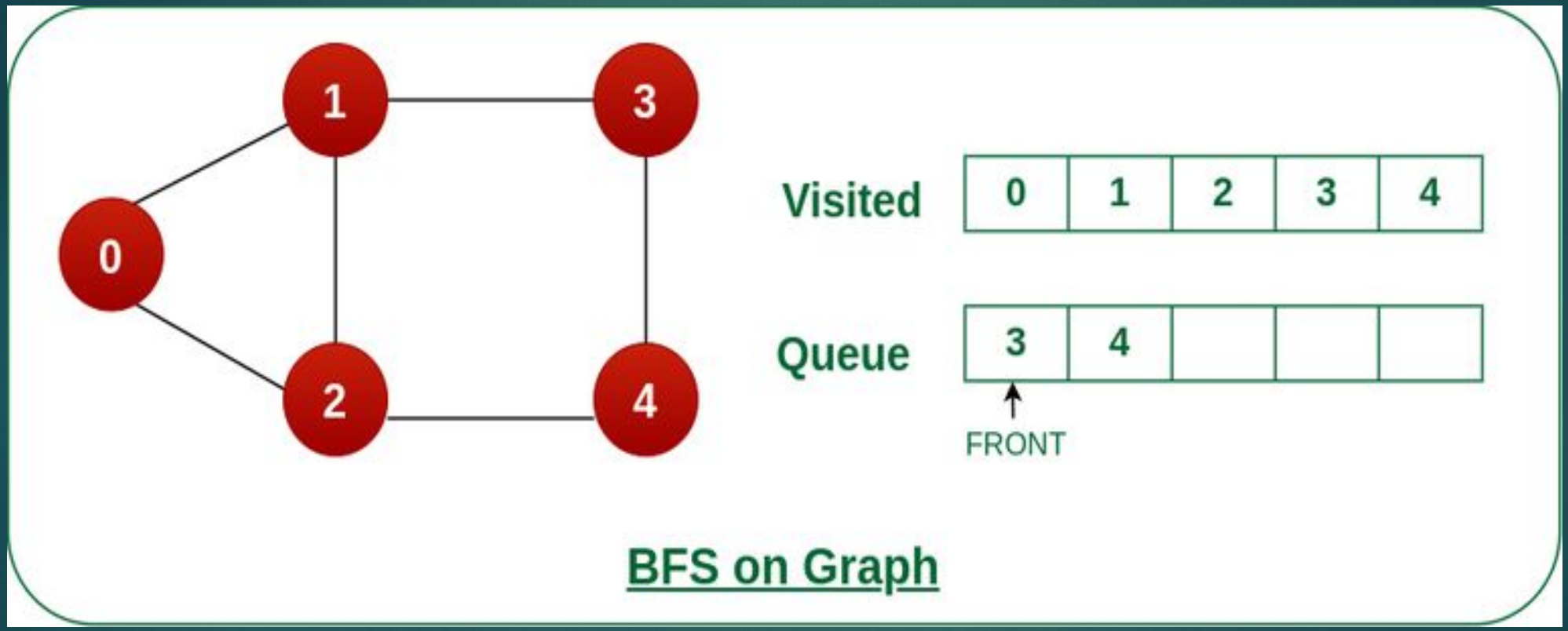


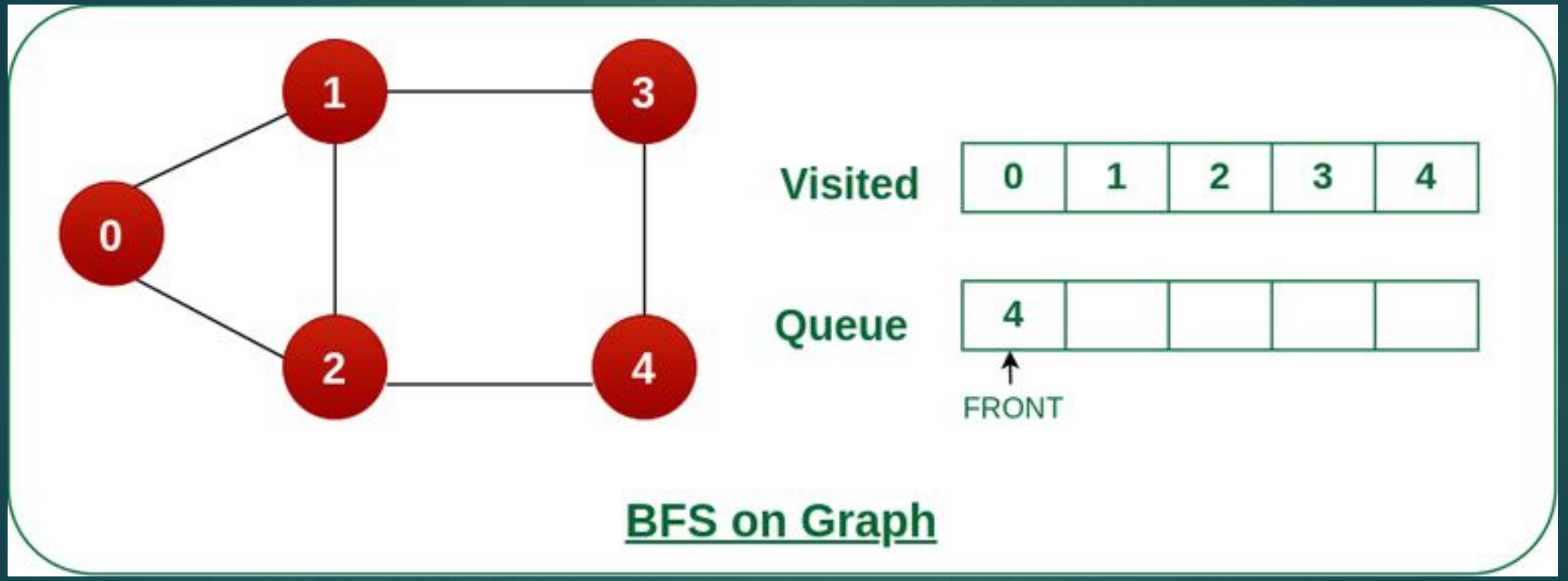
- ***Step1:*** *Initially queue and visited arrays are empty.*
- ***Step2:*** *Push 0 into queue and mark it visited.*

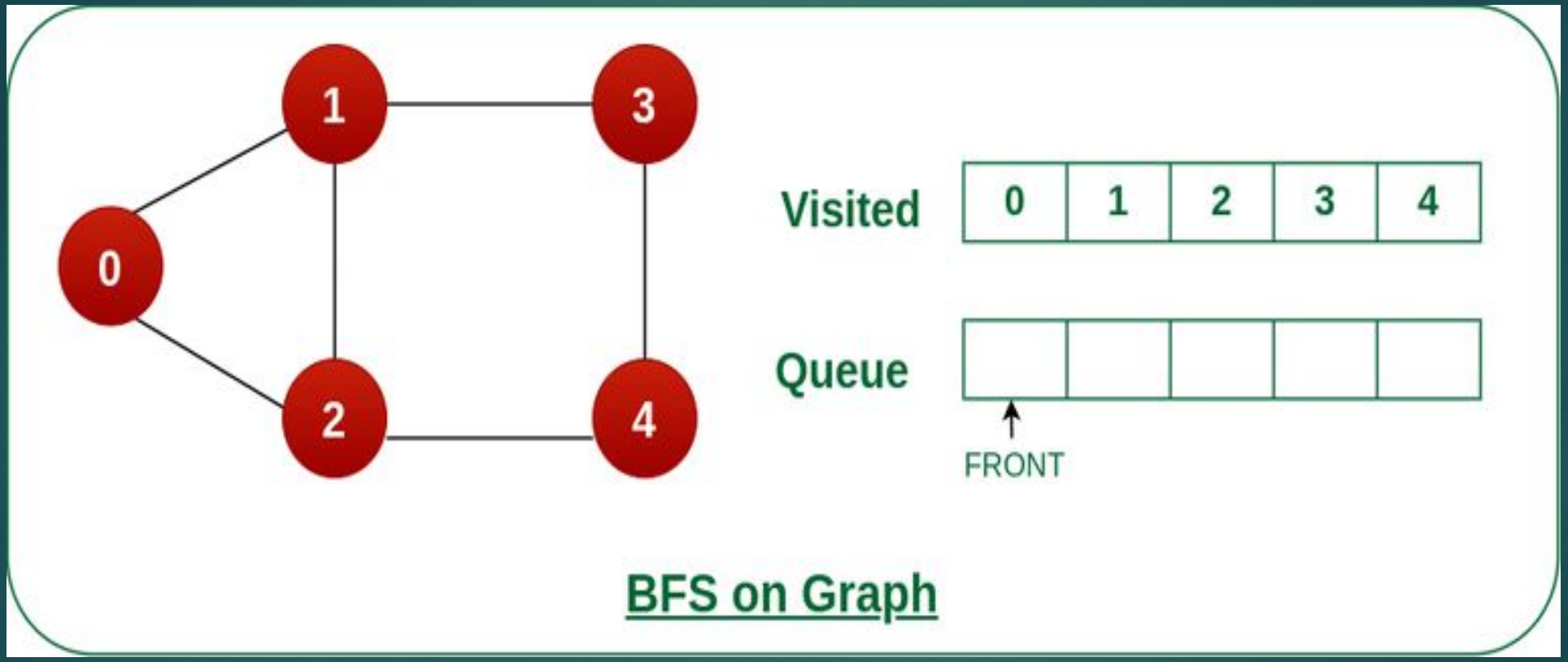

1
3
0
2
4
Visited
0
Queue
0
FRONT
BFS on Graph


1
3
0
2
4
Visited
0
1
2
Queue
1
2
FRONT
BFS on Graph


1
3
0
2
4
Visited
0
1
2
3
Queue
2
3
FRONT
BFS on Graph


1
3
0
2
4
Visited
0
1
2
3
4
Queue
3
4
FRONT
BFS on Graph

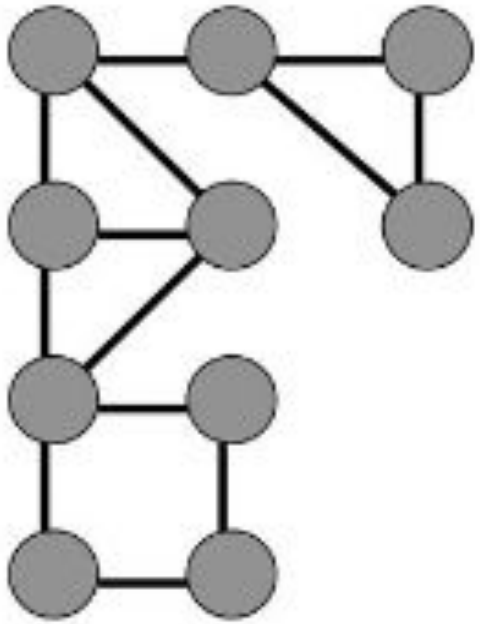

1
3
0
2
4
Visited
0
1
2
3
4
Queue
4
FRONT
BFS on Graph


1
3
0
2
4
Visited
0
1
2
3
4
Queue
FRONT
BFS on Graph

Articulation Point

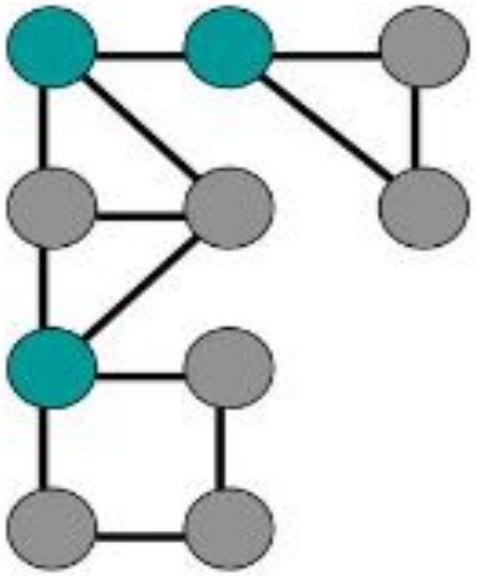
- Let G = (V,E) be a connected undirected graph.

Articulation Point: is any vertex of G whose removal results in a disconnected graph.



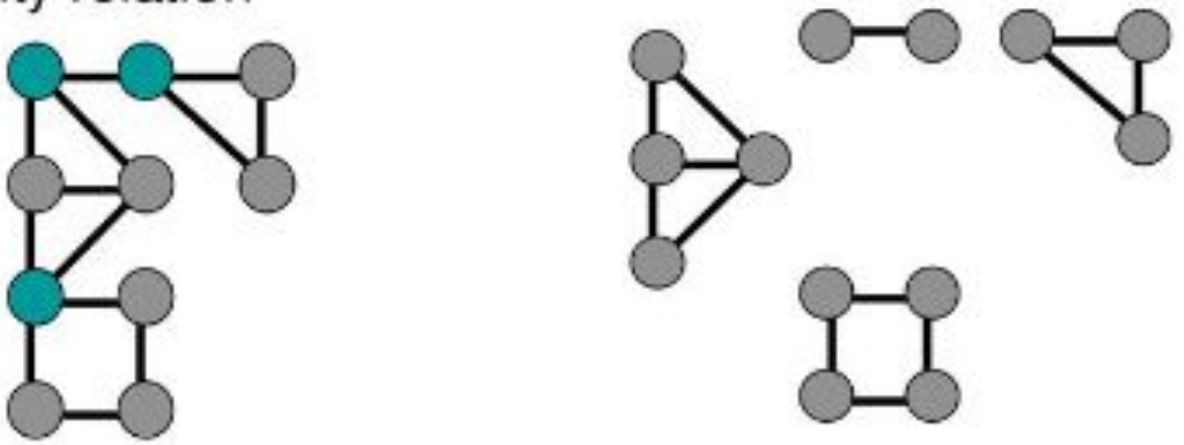
Articulation Point

Articulation Point: is any vertex of G whose removal results in a disconnected graph.



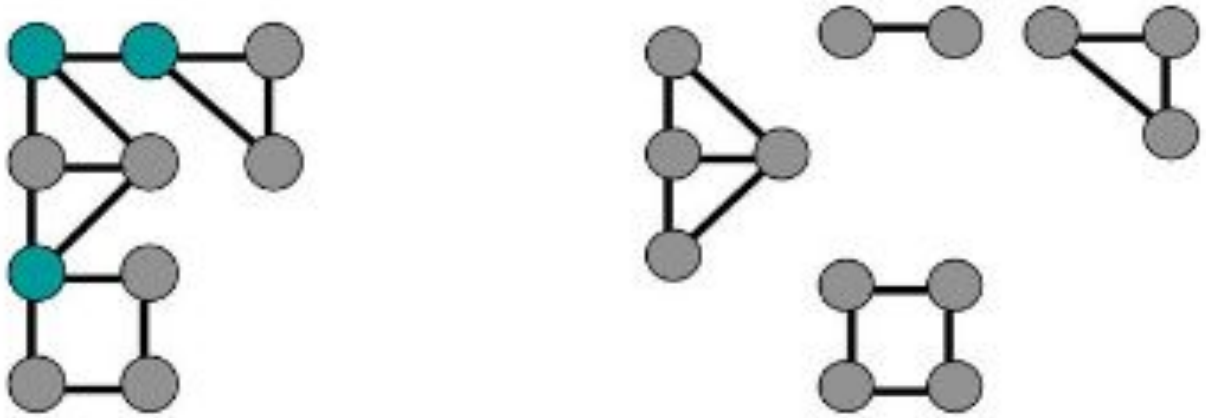
Biconnected components

- A graph is biconnected if it contains no articulation points.
- Two edges are cocyclic if they are equal or if there is a simple cycle that contains both edges. (Two different ways of getting from one edge to the other)
  - This defines an equivalence relation on the edges of the graph
- Biconnected components of a graph are the equivalence classes of cocyclicity relation



Biconnected components

- A graph is biconnected if and only if it consists of a single biconnected component
  - No articulation points



Articulation points and DFS

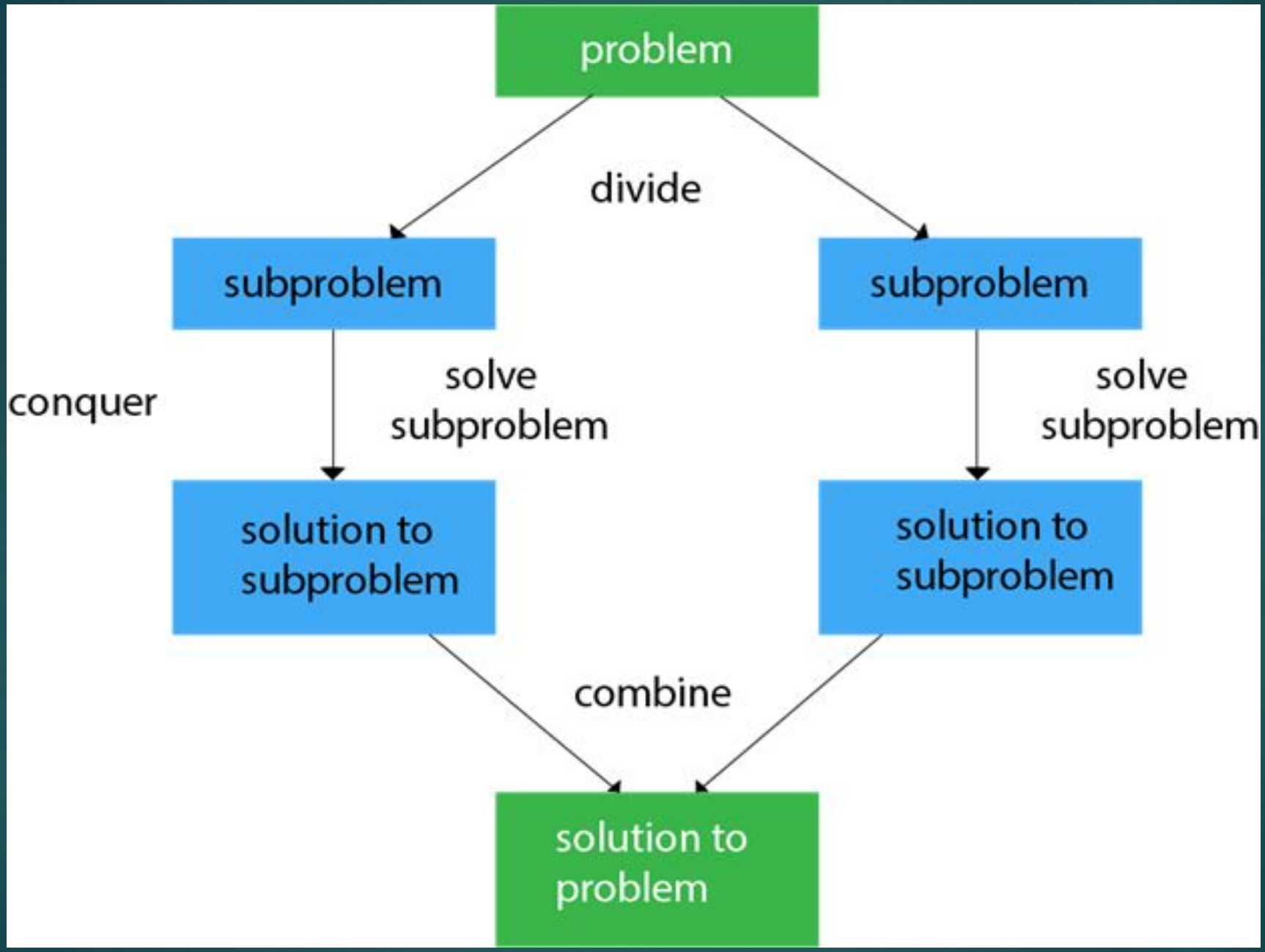
- How to find articulation points?
  - Use the tree structure provided by DFS
  - G is undirected: tree edges and back edges (no difference between forward and back edges, no cross edges)

- Assume G is connected

Divide and Conquer Introduction

- Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

- Divide and Conquer algorithm consists of a dispute using the following three steps.
- **Divide** the original problem into a set of subproblems.
- **Conquer:** Solve every subproblem individually, recursively.
- **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.


problem
divide
subproblem
subproblem
conquer
solve
subproblem
solve
subproblem
solution to
subproblem
solution to
subproblem
combine
solution to
problem

- Generally, we can follow the **divide-and-conquer** approach in a three-step process.
- **Examples:** The specific computer algorithms are based on the Divide & Conquer approach:
- Maximum and Minimum Problem
- Binary Search
- Sorting (merge sort, quick sort)
- Tower of Hanoi.

- **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.

- **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.

- **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.

- Strassen's Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is $\mathbf{O(n^3)}$, since it takes two loops to multiply. Strassen's method was introduced to reduce the time complexity from $\mathbf{O(n^3)}$ to $\mathbf{O(n^{\log 7})}$.

Strassen's Matrix Multiplication

- Belongs to Divide and Conquer Paradigm
- How Matrix Multiplication done Normally
- How divide and Conquer strategy applied for matrix multiplication
- What Strassen had done to improve matrix multiplication using divide and conquer strategy

Matrix Multiplication

$$A=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times B \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

2×2 $\qquad\qquad$ 2×2

$$= C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

2×2

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{ik}$$

Matrix Multiplication (contd.)

```
for(i=0;i<n;i++)
    {
     for(j=0;j<n;j++)
       {
        C[i,j]=0;
         C[i,j] = A[i,k] ×B[k,j];
         }
      }
```

Time Complexity of Matrix multiplication is $O(n^3)$

Divide and Conquer strategy for matrix multiplication

$$A=\underset{2\times 2}{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}} \times B \underset{2\times 2}{\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}}$$

$$= C = \underset{2\times 2}{\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}}$$

- Multiplying a 2×2 matrix is a small problem
- $c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21}$
- $c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$
- $c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$
- $c_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$

Divide and Conquer strategy for matrix multiplication (contd.)

- If each addition term takes one unit of time, then total time is 4 unit, it's a constant
- If there are 8 multiplications and take 8 unit of time, it is also a constant.
- These 4 formulas take constant time.
- If we want to reduce the problem into 1x1 matrix

$$A = [a_{11}], B = [b_{11}]$$

$$A \times B = C$$

$$C = [a_{11} * b_{11}]$$

- If the row/column size is grater than 2, we need to divide the problem and solve each single problem
- We take the problem or assume the problem as power of 2.
- If the matrix is not power of 2 then we make it as power of 2 by adding 0's

Divide and Conquer strategy for matrix multiplication (contd.)

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times B \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

A_{11} A_{12} A_{21} A_{22} $4/2 \times 4/2$ B_{11} B_{12} B_{21} B_{22} $4/2 \times 4/2$

$$= C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

A_{11} A_{12} A_{11} A_{12} $4/2 \times 4/2$

Here A and B are divided into four 2x2 matrices

- Algorithm of divide and conquer matrix multiplication

```
Algorithm MM(A,B,n)
    {
     if(n≤2)
      {
      c11 = a11 × b11+a12 × b21
      c12 = a11 × b12+a12 × b22
      c21 = a21 × b11+a22 × b21
       c22 = a21 × b12+a22 × b22
      }
     else
      {
       MM(A11,B11,n/2)+ MM(A12,B21,n/2)
       MM(A11,B12,n/2)+ MM(A12,B22,n/2)
       MM(A21,B11,n/2)+ MM(A22,B21,n/2)
       MM(A21,B12,n/2)+ MM(A22,B22,n/2)
       }
     }
```

meghav@kanpuruniv.ac.in

Divide and Conquer strategy for matrix multiplication (contd.)

› Time Complexity

8 times recursive call is required.

$$T(n)=\begin{cases}1 & n\leq 2\\ 8T\left(\frac{n}{2}\right)+n^2 & n>2\end{cases}$$

Time complexity is $\theta(n^3)$

› If we reduce the number of multiplication we can make the algorithm faster

Strassen's Matrix Multiplication

- Reduce 8 to 7 multiplication
- Different equations are applied on the four formulas
- No. of multiplication is reduced but addition and subtraction increased.

$P = (A_{11}+A_{22})(B_{11}+B_{22})$

$Q = (A_{21}+A_{22})$

$R = A_{11}(B_{12}-B_{22})$

$S = A_{22}(B_{21}-B_{11})$

$T = (A_{11}+A_{12})B_{22}$

$U = (A_{21}-A_{11})(B_{11}+B_{12})$

$V = (A_{12}-A_{22})(B_{21}+B_{22})$

$C_{11} = P+S-T+V$

$C_{12} = R+T$

$C_{21} = Q+S$

$C_{22} = P+R-Q+U$

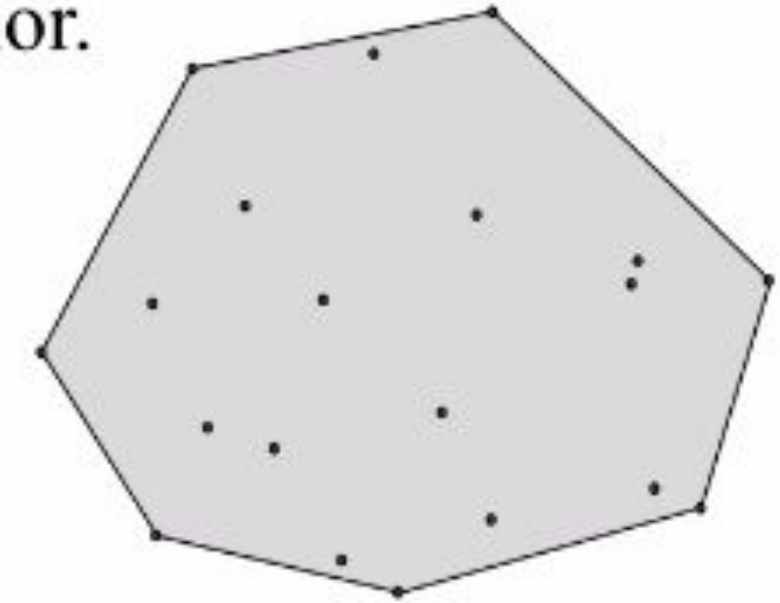
Recurrence relation

$$T(n)=\begin{cases}1 & n\leq 2\\ 8T\left(\frac{n}{2}\right)+n^2 & n>2\end{cases}$$

Time complexity is $O(n^{2.81})$

Convex hull

- Convex hull is defined as a set of given object
- Convex hull of a set Q of points, denoted by CH(Q)is the smallest convex polygon P for which each points in Q is either on the boundary of P or in its interior.



There are many algorithms for computing the convex hull

a) Brute Force Algorithm
b) Quick Hull
c) Divide and Conquer
d) Grahams scan
e) Jarvis march(Gift wrapping)

Convex hull using Brute Force Algorithm

- Given a set of points P, test each line segment to see if it makes up an edge of convex hull.

- If the rest of the points are on one side of the segment ,the segment is on the convex hull.
- Otherwise the segment is not on the hull.

Time complexity of convex hull using Brute force technique

Computation time of convex hull using brute force algoritm is $O(n^3)$:-

- $O(n)$ complexity tests, for each of $O(n^2)$ edges.
- In a d-dimensional space, the complexity is $O(n^{d+1})$

Quick Hull

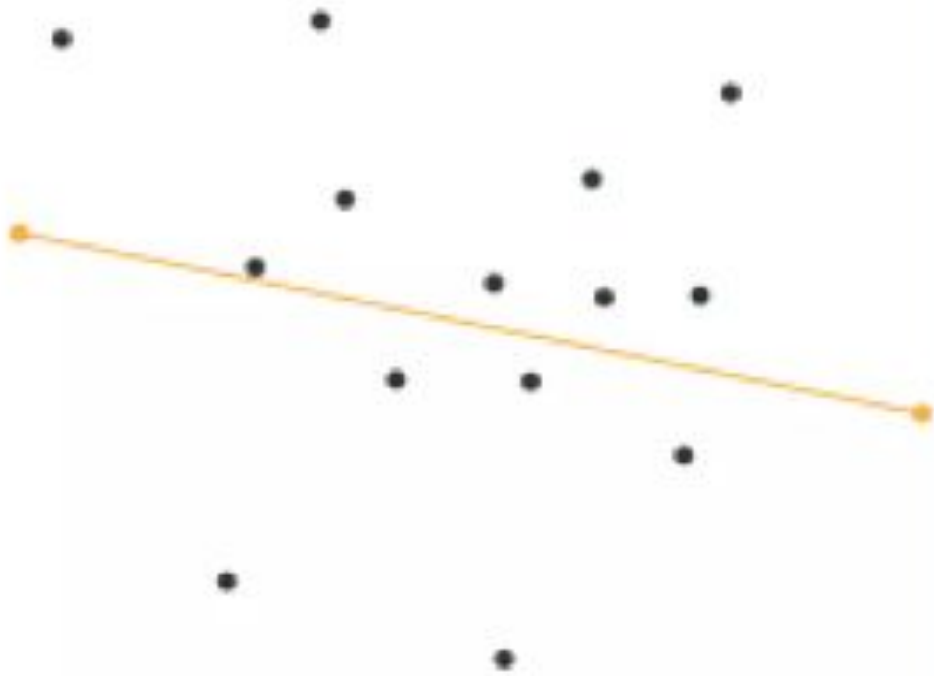
- Quick Hull uses a divide and conquer approach.
- For all a,b,c ϵ P, the points contained in $\Delta abc \cap P$ cannot be on the convex hull.

Initial input

- The initial input to the algorithm is an arbitrary set of points.

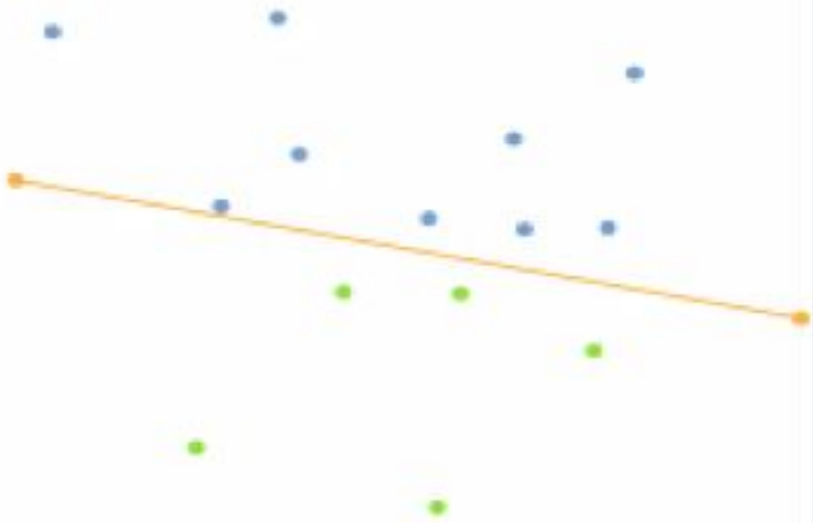
First two points on the convex hull

- Starting with the given set of points the first operation done is the calculation of the two maximal points on the horizontal axis.



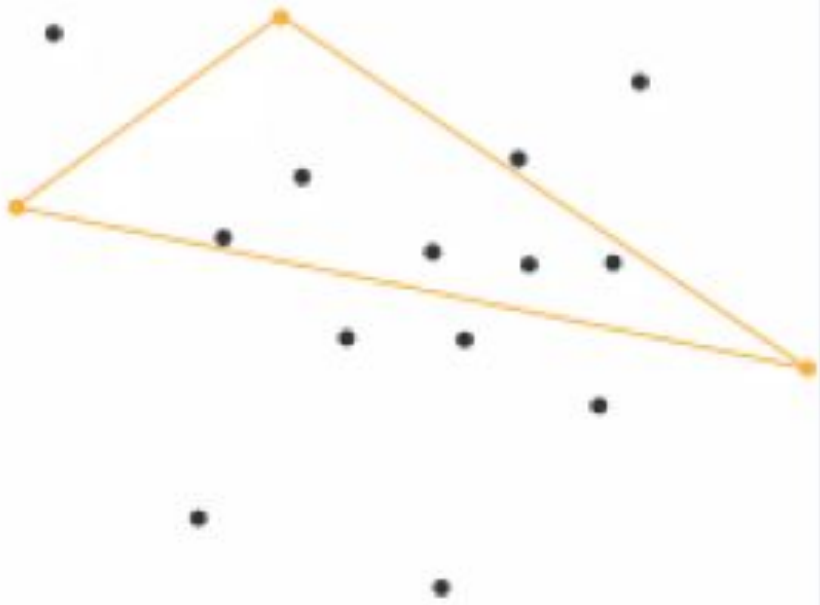
Recursively divide

- Next the line formed by these two points is used to divide the set into two different parts.
- Everything left from this line is considered one part, everything right of it is considered another one.
- Both of these parts are processed recursively.



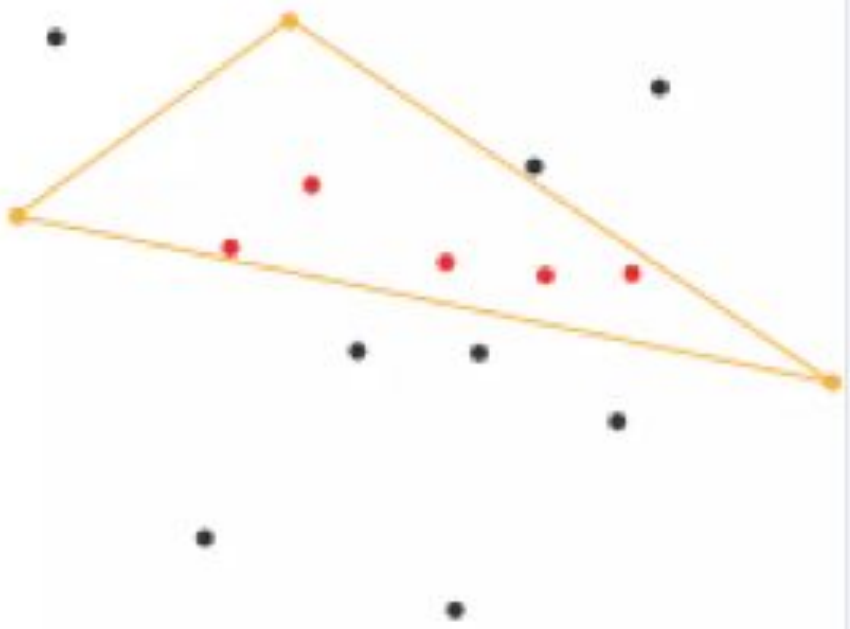
Max distance search

- To determine the next point on the convex hull a search for the point with the greatest distance from the dividing line is done.
- This point, together with the line start and end point forms a triangle.



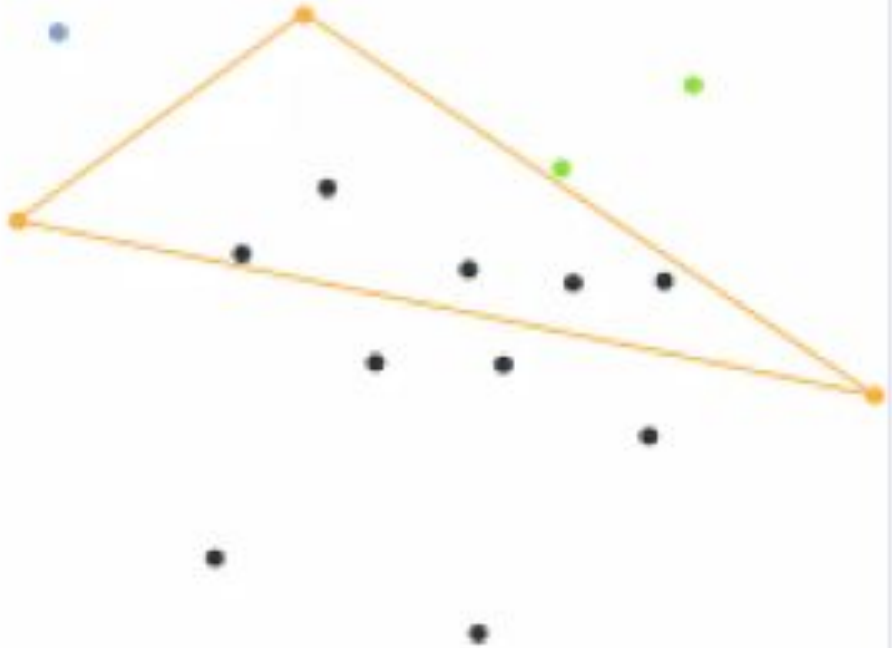
Point exclusion

- All points inside this triangle can not be part of the convex hull polygon, as they are obviously lying in the convex hull of the three selected points.
- Therefore these points can be ignored for every further processing step.



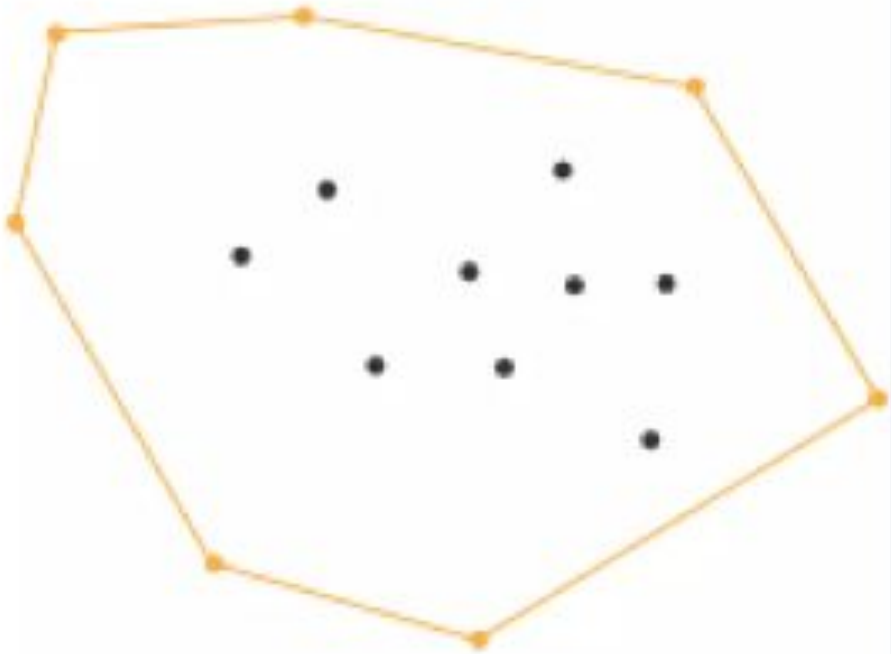
Recursively divide

- Having this in mind the recursive processing can take place again.
- Everything right of the triangle is used as one subset, everything left of it as another one.



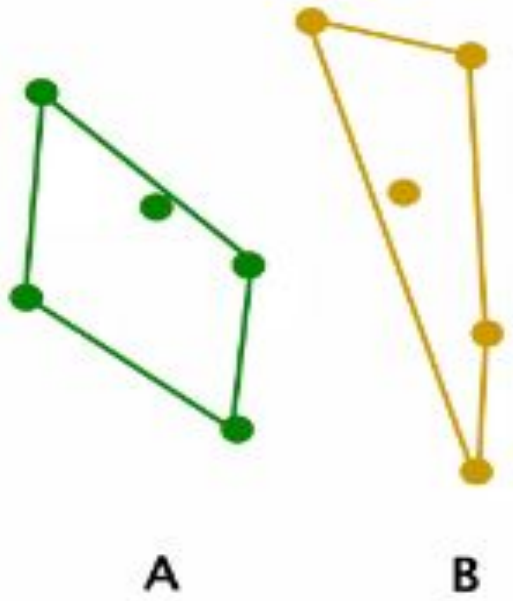
Abort condition

- At some point the recursively processed point subset does only contain the start and end point of the dividing line.
- If this is case this line has to be a segment of the searched hull polygon and the recursion can come to an end.



Convex Hull: Divide & Conquer

- Preprocessing: sort the points by x-coordinate
- Divide the set of points into two sets **A** and **B**:
  - **A** contains the left $\lfloor n/2 \rfloor$ points,
  - **B** contains the right $\lceil n/2 \rceil$ points
- Recursively compute the convex hull of **A**
- Recursively compute the convex hull of **B**
- Merge the two convex hulls



Convex Hull: Runtime

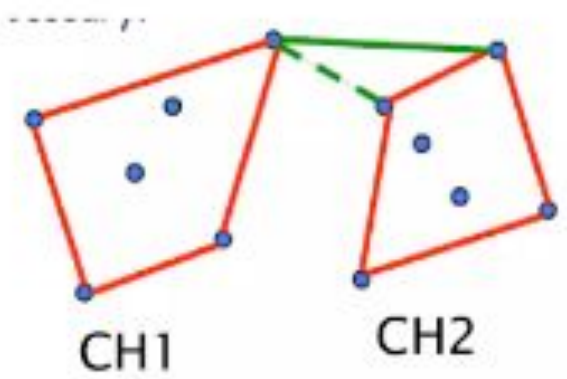
- Preprocessing: sort the points by x-coordinate — O(n log n)
- Divide the set of points into two sets **A** and **B**: — O(1)
  - **A** contains the left $\lfloor n/2 \rfloor$ points,
  - **B** contains the right $\lceil n/2 \rceil$ points
- Recursively compute the convex hull of **A** — T(n/2)
- Recursively compute the convex hull of **B** — T(n/2)
- Merge the two convex hulls — O(n)

Merg hull

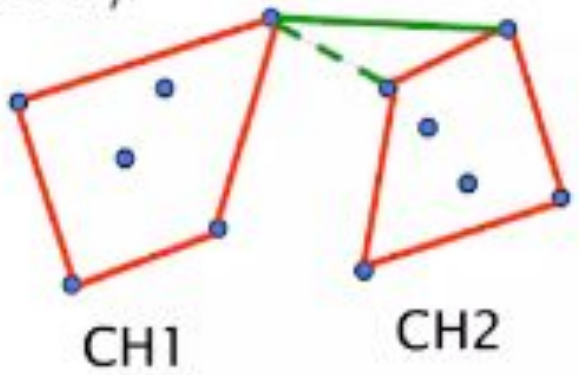
In merging hull we find upper and lower tangent of CH1 and CH2

Find the Upper Tangent:

- Start at the rightmost vertex of CH1 and the leftmost vertex of CH2.
- Upper tangent will be a line segment such that it makes a left turn with next counter-clockwise vertex of CH1 and makes a right turn with next clockwise vertex of CH2.

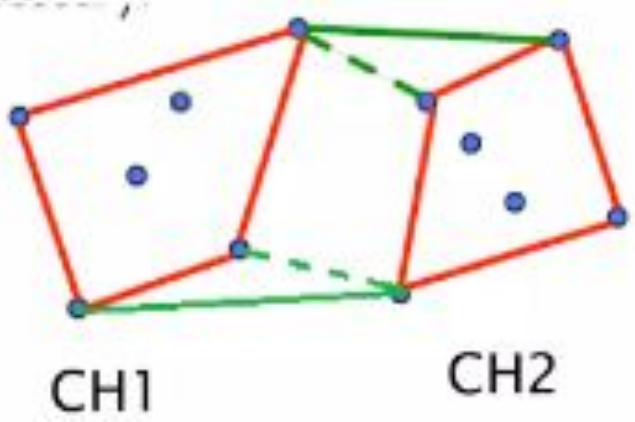


- If the line segment connecting them is not the upper tangent:
  - If the line segment does not make a left turn with next counter clockwise vertex of CH1, rotate to the next counter-clockwise vertex in CH1.
  - Else if the line segment does not make a right turn with the next clockwise vertex of CH2, rotate to the next clockwise vertex in CH2. Repeat as necessary.



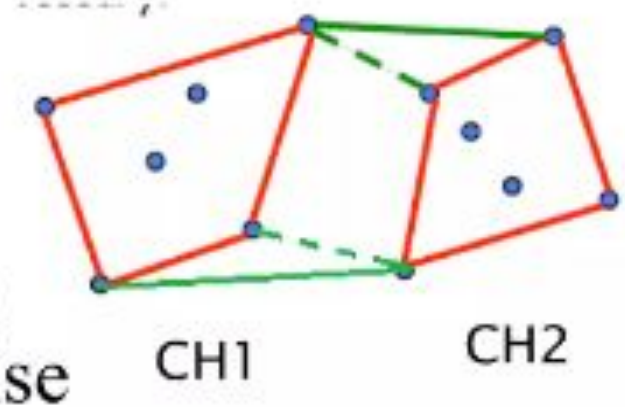
Find the Lower Tangent

- Start at the rightmost vertex of CH1 and the leftmost vertex of CH2.
- Lower tangent will be a line segment such that it makes a right turn with next clockwise vertex of CH1 and makes a left turn with next counter clockwise vertex of CH2.



- If the line segment connecting them is not the lower tangent:
  - If the line segment does not make a right turn with next clockwise vertex of CH1, rotate to the next clockwise vertex in CH1.
  - Else if the line segment does not make a left turn with the next counter-clockwise vertex of CH2, rotate to the next counter clockwise vertex in CH2.

    Repeat as necessary.



Algorithm

QuickHull(s)

{

//Find convex hull from the set S of n points.

Convex hull:={}

1. Find left and right most points,say A and B, and add A and B to convex hull
2. Segment AB divides the remaining (n-2)points into 2 groups S1 and S2.

   where S1 are points in S that are on the right side of the oriented line from A to B

   and S2 are points in S that are on the right side of the oriented line from B to A
3. FindHull (S1,A,B)
4. FindHull (S2,B,A)

Findhull (Sk,P,Q)

{

//find points on convex hull from the set of points that are on the right side of the oriented line from P to Q

If Sk has no points,

then return.

From the given set of points in Sk, find farthest point,say c, from segment PQ

Add points C to convex hull at the location between P and Q three points P,Q and C partition the remaining points of Sk into 3 subsets : S0,S1,S2

Where S0 are points inside triangle PCQ, S1 are points on the right side of the oriented line from P to C and S2 are points on the right side of the oriented line from from C to Q.

FindHull (S1,P,C)

FindHull (S2,C,Q)

Time complexity

$T(n)=2T(n/2)+O(n)$

$T(n)= \Theta(n\log n)$; Average case

$T(n)=O(n^2)$; Worst case

Graham scan

- Graham scan is a method of computing the convex hull of a finite set of points in the plane with time complexity O(n logn).
- The algorithm finds all vertices of the convex hull ordered along its boundary
- Graham's scan solves the convex-hull problem by maintaining a stack S of candidate points. Each point of the input set Q is pushed once onto the stack.

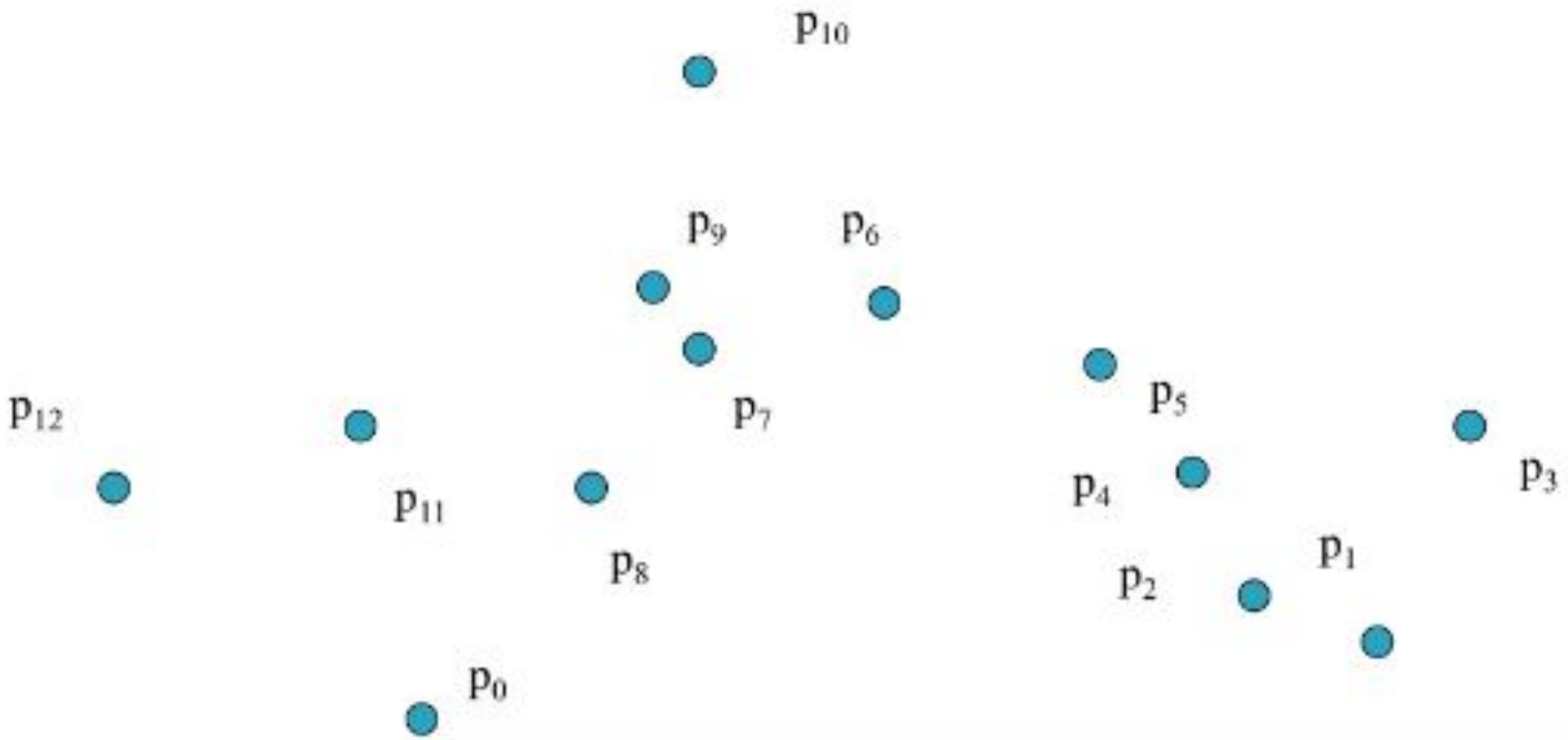
- And the points that are not vertices of CH(Q) are eventually popped from the stack. When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.
- When we traverse the convex hull counter clockwise, we should make a left turn at each vertex.
- Each time the while loop finds a vertex at which we make a non left turn ,the vertex is popped from the vertex.

Algorithm

GRAHAM-SCAN(*Q*)

1. Let $p0$ be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie.
2. Let $(p1, p2, \ldots, pm)$ be the remaining points in Q, sorted by polar angle in counterclockwise order around $p0$(if more than point has the same angle, remove all but the one that is farthest from $p0$)
3. $top[S] \neg 0$
4. PUSH($p0$,S)
5. PUSH($p1$,S)
6. PUSH($p2$,S)
7. for $i \neg 3$ to m
8. do while the angle formed by points PREV-TO-TOP(S), TOP(S), and pi makes a nonleft turn
9. do POP(S)
10. PUSH(S,pi)
11. return S

Graham Scan – Example



Graham Scan – Example