

Prim's Algorithm for Minimum Spanning Tree (MST) - GeeksforGeeks

Source: <https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Prim's Algorithm for Minimum Spanning Tree (MST) Last Updated : 25 Dec, 2025 Prim's algorithm is a Greedy algorithm like Kruskal's algorithm . This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way. The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. Working of the Prim's Algorithm Determine an arbitrary vertex as the starting vertex of the MST. We pick 0 in the above diagram. Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex). Find edges connecting any tree vertex with the fringe vertices. Find the minimum among these edges. Add the chosen edge to the MST. Since we consider only the edges that connect fringe vertices with the rest, we never get a cycle. Return the MST and exit Simple Implementation for Adjacency Matrix Representation Follow the given steps to utilize the Prim's Algorithm mentioned above for finding MST of a graph: Create a set mstSet that keeps track of vertices already included in MST. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first. While mstSet doesn't include all vertices => Pick a vertex u that is not there in mstSet and has a minimum key value. => Include u in the mstSet . => Update the key value of all adjacent vertices of u . To update the key values, iterate through all adjacent vertices. For every adjacent vertex v , if the weight of edge u-v is less than the previous key value of v , update the key value as the weight of u-v . The idea of using key values is to pick the minimum weight edge from the cut. The key values are used only for vertices that are not yet included in MST, the key value for these vertices indicates the minimum weight edges connecting them to the set of vertices included in MST. Try it on GfG Practice C++ #include <iostream> #include <vector> using namespace std ; // A utility function to find the vertex with // minimum key value, from the set of vertices // not yet included in MST int minKey (vector < int > & key , vector < bool > & mstSet) { // Initialize min value int min = INT_MAX , min_index ; for (int v = 0 ; v < mstSet . size () ; v ++) if (mstSet [v] == false && key [v] < min) min = key [v] , min_index = v ; return min_index ; } // A utility function to print the // constructed MST stored in parent[] void printMST (vector < int > & parent , vector < vector < int >> & graph) { cout << "Edge \t Weight \n " ; for (int i = 1 ; i < graph . size () ; i ++) cout << parent [i] << " - " << i << " \t " << graph [parent [i]][i] << " \n " ; } // Function to construct and print MST for // a graph represented using adjacency // matrix representation void primMST (vector < vector < int >> & graph) { int V = graph . size () ; // Array to store constructed MST vector < int > parent (V) ; // Key values used to pick minimum weight edge in cut vector < int > key (V) ; // To represent set of vertices included in MST vector < bool > mstSet (V) ; // Initialize all keys as INFINITE for (int i = 0 ; i < V ; i ++) key [i] = INT_MAX , mstSet [i] = false ; // Always include first 1st vertex in MST. // Make key 0 so that this vertex is picked as first // vertex. key [0] = 0 ; // First node is always root of MST parent [0] = -1 ; // The MST will have V vertices for (int count = 0 ; count < V - 1 ; count ++) { // Pick the minimum key vertex from the // set of vertices not yet included in MST int u = minKey (key , mstSet) ; // Add the picked vertex to the MST Set mstSet [u] = true ; // Update key value and parent index of // the

```

adjacent vertices of the picked vertex. // Consider only those vertices which are not // yet included in MST for ( int v = 0 ; v < V ; v ++ ) // graph[u][v] is non zero only for adjacent // vertices of m mstSet[v] is false for vertices // not yet included in MST Update the key only // if graph[u][v] is smaller than key[v] if ( graph [ u ][ v ] && mstSet [ v ] == false && graph [ u ][ v ] < key [ v ] ) parent [ v ] = u , key [ v ] = graph [ u ][ v ]; } // Print the constructed MST printMST ( parent , graph ); } int main () { vector < vector < int >> graph = { { 0 , 2 , 0 , 6 , 0 } , { 2 , 0 , 3 , 8 , 5 } , { 0 , 3 , 0 , 0 , 7 } , { 6 , 8 , 0 , 0 , 9 } , { 0 , 5 , 7 , 9 , 0 } }; // Print the solution primMST ( graph ); return 0 ; } C #include <limits.h> #include <stdbool.h> #include <stdio.h> // Number of vertices in the graph #define V 5 // A utility function to find the vertex with // minimum key value, from the set of vertices // not yet included in MST int minKey ( int key [] , bool mstSet [] ) { // Initialize min value int min = INT_MAX , min_index ; for ( int v = 0 ; v < V ; v ++ ) if ( mstSet [ v ] == false && key [ v ] < min ) min = key [ v ] , min_index = v ; return min_index ; } // A utility function to print the // constructed MST stored in parent[] int printMST ( int parent [] , int graph [ V ][ V ] ) { printf ( "Edge \t Weight \n " ); for ( int i = 1 ; i < V ; i ++ ) printf ( "%d - %d \t %d \n " , parent [ i ] , i , graph [ parent [ i ]][ i ]); } // Function to construct and print MST for // a graph represented using adjacency // matrix representation void primMST ( int graph [ V ][ V ] ) { // Array to store constructed MST int parent [ V ]; // Key values used to pick minimum weight edge in cut int key [ V ]; // To represent set of vertices included in MST bool mstSet [ V ]; // Initialize all keys as INFINITE for ( int i = 0 ; i < V ; i ++ ) key [ i ] = INT_MAX , mstSet [ i ] = false ; // Always include first 1st vertex in MST. // Make key 0 so that this vertex is picked as first // vertex. key [ 0 ] = 0 ; // First node is always root of MST parent [ 0 ] = -1 ; // The MST will have V vertices for ( int count = 0 ; count < V - 1 ; count ++ ) { // Pick the minimum key vertex from the // set of vertices not yet included in MST int u = minKey ( key , mstSet ); // Add the picked vertex to the MST Set mstSet [ u ] = true ; // Update key value and parent index of // the adjacent vertices of the picked vertex. // Consider only those vertices which are not // yet included in MST for ( int v = 0 ; v < V ; v ++ ) // graph[u][v] is non zero only for adjacent // vertices of m mstSet[v] is false for vertices // not yet included in MST Update the key only // if graph[u][v] is smaller than key[v] if ( graph [ u ][ v ] && mstSet [ v ] == false && graph [ u ][ v ] < key [ v ] ) parent [ v ] = u , key [ v ] = graph [ u ][ v ]; } // Print the constructed MST printMST ( parent , graph ); } int main () { int graph [ V ][ V ] = { { 0 , 2 , 0 , 6 , 0 } , { 2 , 0 , 3 , 8 , 5 } , { 0 , 3 , 0 , 0 , 7 } , { 6 , 8 , 0 , 0 , 9 } , { 0 , 5 , 7 , 9 , 0 } }; // Print the solution primMST ( graph ); return 0 ; } Java import java.io.* ; import java.lang.* ; import java.util.* ; class MST { // A utility function to find the vertex with minimum // key value, from the set of vertices not yet included // in MST int minKey ( int key [] , Boolean mstSet [] ) { // Initialize min value int min = Integer . MAX_VALUE , min_index = -1 ; for ( int v = 0 ; v < mstSet . length ; v ++ ) if ( mstSet [ v ] == false && key [ v ] < min ) { min = key [ v ] ; min_index = v ; } return min_index ; } // A utility function to print the constructed MST // stored in parent[] void printMST ( int parent [] , int graph [][] ) { System . out . println ( "Edge \tWeight" ); for ( int i = 1 ; i < graph . length ; i ++ ) System . out . println ( parent [ i ] + " - " + i + "\t" + graph [ parent [ i ]][ i ]); } // Function to construct and print MST for a graph // represented using adjacency matrix representation void primMST ( int graph [][] ) { int V = graph . length ; // Array to store constructed MST int parent [] = new int [ V ] ; // Key values used to pick minimum weight edge in // cut int key [] = new int [ V ] ; // To represent set of vertices included in MST Boolean mstSet [] = new Boolean [ V ] ; // Initialize all keys as INFINITE for ( int i = 0 ; i < V ; i ++ ) { key [ i ] = Integer . MAX_VALUE ; mstSet [ i ] = false ; } // Always include first 1st vertex in MST. // Make key 0 so that this vertex is // picked as first vertex key [ 0 ] = 0 ; // First node is always root of MST parent [ 0 ] = -1 ; // The MST will have V vertices for ( int count = 0 ; count < V - 1 ; count ++ ) { // Pick the minimum key vertex from the set of // vertices not yet included in MST int u = minKey ( key , mstSet ); // Add the picked vertex to the MST Set mstSet [ u ] = true ; // Update key value and parent index of the // adjacent vertices of the picked vertex. // Consider only those vertices which are not // yet included in MST for ( int v = 0 ; v < V ; v ++ ) // graph[u][v] is non zero only for adjacent // vertices of m mstSet[v] is false for // vertices not yet included in MST Update // the key only if graph[u][v] is smaller // than key[v] if ( graph [ u ][ v ] != 0 && mstSet [ v ] == false && graph [ u ][ v ] < key [ v ] ) { parent [ v ] = u ; key [ v ] = graph [ u ][ v ]; } } // Print the constructed MST printMST ( parent , graph ); } public static void main ( String [] args ) { MST t = new MST (); int graph [][] = new int [][] { { 0 , 2 , 0 , 6 , 0 } , { 2 , 0 , 3 , 8 , 5 } , { 0 , 3 , 0 , 0 , 7 } , { 6 , 8 , 0 , 0 , 9 } , { 0 , 5 , 7 , 9 , 0 } }; // Print the solution t . primMST ( graph ); } } Python # Library for INT_MAX import sys class Graph (): def __init__ ( self , vertices ): self . V = vertices self . graph = [[ 0 for column in range ( vertices )] for row in range ( vertices )] # A utility function to print # the constructed MST stored in parent[] def printMST ( self , parent ): print ( "Edge \t Weight" ) for i in range ( 1 , self . V ): print ( parent [ i ] , "-" , i , "\t" , self . graph [ parent [ i ]][ i ]); # A utility function to find the vertex with # minimum distance value, from the set of vertices # not yet included in shortest path tree def minKey (
```

```

self , key , mstSet ): # Initialize min value min = sys . maxsize for v in range ( self . V ): if key [ v ] < min and mstSet [ v ] == False : min = key [ v ] min_index = v return min_index # Function to construct and print MST for a graph # represented using adjacency matrix representation def primMST ( self ): # Key values used to pick minimum weight edge in cut key = [ sys . maxsize ] * self . V parent = [ None ] * self . V # Array to store constructed MST # Make key 0 so that this vertex is picked as first vertex key [ 0 ] = 0 mstSet = [ False ] * self . V parent [ 0 ] = - 1 # First node is always the root of for cout in range ( self . V ): # Pick the minimum distance vertex from # the set of vertices not yet processed. # u is always equal to src in first iteration u = self . minKey ( key , mstSet ) # Put the minimum distance vertex in # the shortest path tree mstSet [ u ] = True # Update dist value of the adjacent vertices # of the picked vertex only if the current # distance is greater than new distance and # the vertex in not in the shortest path tree for v in range ( self . V ): # graph[u][v] is non zero only for adjacent vertices of m # mstSet[v] is false for vertices not yet included in MST # Update the key only if graph[u][v] is smaller than key[v] if self . graph [ u ][ v ] > 0 and mstSet [ v ] == False \ and key [ v ] > self . graph [ u ][ v ]: key [ v ] = self . graph [ u ][ v ] parent [ v ] = u self . printMST ( parent ) if __name__ == '__main__': g = Graph ( 5 ) g . graph = [[ 0 , 2 , 0 , 6 , 0 ],[ 2 , 0 , 3 , 8 , 5 ],[ 0 , 3 , 0 , 0 , 7 ],[ 6 , 8 , 0 , 0 , 9 ],[ 0 , 5 , 7 , 9 , 0 ]] g . primMST () C# using System ; using System.Collections.Generic ; class MST { // A utility function to find the vertex with minimum // key value, from the set of vertices not yet included // in MST int MinKey ( int [] key , bool [] mstSet ) { // Initialize min value int min = int . MaxValue , minIndex = - 1 ; for ( int v = 0 ; v < mstSet . Length ; v ++ ) if ( ! mstSet [ v ] && key [ v ] < min ) { min = key [ v ]; minIndex = v ; } return minIndex ; } // A utility function to print the constructed MST // stored in parent[] void PrintMST ( int [] parent , int [,] graph ) { Console . WriteLine ( "Edge \tWeight" ); for ( int i = 1 ; i < graph . GetLength ( 0 ); i ++ ) Console . WriteLine ( parent [ i ] + " - " + i + "\t" + graph [ parent [ i ], i ]); } // Function to construct and print MST for a graph // represented using adjacency matrix representation public void PrimMST ( int [,] graph ) { int V = graph . GetLength ( 0 ); // Array to store constructed MST int [] parent = new int [ V ]; // Key values used to pick minimum weight edge in // cut int [] key = new int [ V ]; // To represent set of vertices included in MST bool [] mstSet = new bool [ V ]; // Initialize all keys as INFINITE for ( int i = 0 ; i < V ; i ++ ) { key [ i ] = int . MaxValue ; mstSet [ i ] = false ; } // Always include first 1st vertex in MST. // Make key 0 so that this vertex is // picked as first vertex key [ 0 ] = 0 ; // First node is always root of MST parent [ 0 ] = - 1 ; // The MST will have V vertices for ( int count = 0 ; count < V - 1 ; count ++ ) { // Pick the minimum key vertex from the set of // vertices not yet included in MST int u = MinKey ( key , mstSet ); // Add the picked vertex to the MST Set mstSet [ u ] = true ; // Update key value and parent index of the // adjacent vertices of the picked vertex. // Consider only those vertices which are not // yet included in MST for ( int v = 0 ; v < V ; v ++ ) // graph[u][v] is non zero only for adjacent // vertices of m mstSet[v] is false for // vertices not yet included in MST Update // the key only if graph[u][v] is smaller // than key[v] if ( graph [ u , v ] != 0 && ! mstSet [ v ] && graph [ u , v ] < key [ v ]) { parent [ v ] = u ; key [ v ] = graph [ u , v ]; } } // Print the constructed MST PrintMST ( parent , graph ); } public static void Main ( string [] args ) { MST t = new MST (); int [,] graph = new int [,]{ { 0 , 2 , 0 , 6 , 0 }, { 2 , 0 , 3 , 8 , 5 }, { 0 , 3 , 0 , 0 , 7 }, { 6 , 8 , 0 , 0 , 9 }, { 0 , 5 , 7 , 9 , 0 } }; // Print the solution t . PrimMST ( graph ); } } JavaScript function primMST ( graph ) { const V = graph . length ; const key = new Array ( V ). fill ( Number . MAX_VALUE ); const parent = new Array ( V ). fill ( null ); const mstSet = new Array ( V ). fill ( false ); // Start from vertex 0 key [ 0 ] = 0 ; parent [ 0 ] = - 1 ; // first node is root // Build MST for ( let i = 0 ; i < V - 1 ; i ++ ) { // Pick the minimum key vertex not yet in MST let u = - 1 , min = Number . MAX_VALUE ; for ( let v = 0 ; v < V ; v ++ ) { if ( ! mstSet [ v ] && key [ v ] < min ) { min = key [ v ]; u = v ; } } mstSet [ u ] = true ; // Update key and parent for neighbors for ( let v = 0 ; v < V ; v ++ ) { if ( graph [ u ][ v ] > 0 && ! mstSet [ v ] && graph [ u ][ v ] < key [ v ]) { key [ v ] = graph [ u ][ v ]; parent [ v ] = u ; } } } // Print the constructed MST console . log ( "Edge \tWeight" ); for ( let i = 1 ; i < V ; i ++ ) { console . log ( parent [ i ] + " - " + i + "\t" + graph [ parent [ i ][ i ]]); } } // Driver Code const graph = [[ 0 , 2 , 0 , 6 , 0 ],[ 2 , 0 , 3 , 8 , 5 ],[ 0 , 3 , 0 , 0 , 7 ],[ 6 , 8 , 0 , 0 , 9 ],[ 0 , 5 , 7 , 9 , 0 ]]; primMST ( graph ); Output Edge Weight 0 - 1 2 1 - 2 3 0 - 3 6 1 - 4 5 Time Complexity: O(V 2 ), As, we are using adjacency matrix, if the input graph is represented using an adjacency list, then the time complexity of Prim's algorithm can be reduced to O((E+V) * logV) with the help of a binary heap. Auxiliary Space: O(V) Efficient Implementation using Priority Queue and Adjacency List For adjacency list representation, we can achieve O((E+V)*log(V)) because we can find all adjacent of every vertex in O(V + E) time and we can get minimum using priority queue in O(Log V) time. We use a priority queue (min-heap) to always select the edge with the smallest weight. Push the first vertex and its weight into the queue. While the queue is not empty, extract the minimum-weight edge. If the vertex is unvisited, add its weight to a variable (res) and mark it as visited. Push all unvisited adjacent vertices of this vertex into the queue. After all vertices are

```

processed, return the total weight stored in res.

```

C++ #include <iostream> #include <vector> #include <queue>
using namespace std ; // Returns total weight of the Minimum Spanning Tree
int spanningTree ( int V , vector < vector < int >> adj [] ) { // Min-heap storing {weight, vertex} priority_queue
< pair < int , int > , vector < pair < int , int >> , greater < pair < int , int >>> pq ; // Marks vertices already
taken in MST vector < bool > visited ( V , false ); int res = 0 ; // Start from node 0 pq . push ({ 0 , 0 });
while ( ! pq . empty () ) { auto p = pq . top (); pq . pop (); int wt = p . first ; int u = p . second ; if ( visited [ u ] )
continue ; res += wt ; visited [ u ] = true ; // Push adjacent edges for ( auto & v : adj [ u ] ) { if ( ! visited [ v [ 0 ] ] )
{ pq . push ({ v [ 1 ], v [ 0 ]}); } } } return res ; } int main () { int V = 3 ; vector < vector < int >> adj [ V ];
adj [ 0 ]. push_back ({ 1 , 5 }); adj [ 1 ]. push_back ({ 0 , 5 }); adj [ 1 ]. push_back ({ 2 , 3 }); adj [ 2 ]. push_back ({ 1 , 3 });
adj [ 0 ]. push_back ({ 2 , 1 }); adj [ 2 ]. push_back ({ 0 , 1 }); cout << spanningTree ( V , adj ) << endl ; return 0 ; }

Java import java.util.PriorityQueue ; import java.util.ArrayList ; class GFG {
// Returns total weight of the Minimum Spanning Tree
static int spanningTree ( int V , ArrayList < ArrayList < int []>> adj ) { // Min-heap storing {weight, vertex} PriorityQueue < int []> pq =
new PriorityQueue <> (( a , b ) -> a [ 0 ] - b [ 0 ]); boolean [] visited = new boolean [ V ] ; int res = 0 ; // Start
from node 0 pq . add ( new int [] { 0 , 0 }); while ( ! pq . isEmpty () ) { int [] p = pq . poll (); int wt = p [ 0 ];
int u = p [ 1 ]; if ( visited [ u ] ) continue ; res += wt ; visited [ u ] = true ; // Push adjacent edges for ( int []
v : adj . get ( u ) ) { if ( ! visited [ v [ 0 ] ] ) { pq . add ( new int [] { v [ 1 ], v [ 0 ]}); } } } return res ; }

public static void main ( String [] args ) { int V = 3 ; ArrayList < ArrayList < int []>> adj = new ArrayList <> (); for ( int i = 0 ; i < V ; i ++ ) adj . add ( new ArrayList <> () ); adj . get ( 0 ). add ( new int [] { 1 , 5 });
adj . get ( 1 ). add ( new int [] { 0 , 5 }); adj . get ( 1 ). add ( new int [] { 2 , 3 }); adj . get ( 2 ). add ( new int [] { 1 , 3 });
adj . get ( 0 ). add ( new int [] { 2 , 1 }); adj . get ( 2 ). add ( new int [] { 0 , 1 }); System . out . println (
spanningTree ( V , adj )); } }

Python import heapq # Returns total weight of the Minimum Spanning Tree
def spanningTree ( V , adj ): # Min-heap storing (weight, vertex)
pq = [] visited = [ False ] * V res = 0 # Start from node 0
heapq . heappush ( pq , ( 0 , 0 )) while pq : wt , u = heapq . heappop ( pq ) if visited [ u ]:
continue res += wt visited [ u ] = True # Push adjacent edges for v in adj [ u ]: if not visited [ v [ 0 ]]:
heapq . heappush ( pq , ( v [ 1 ], v [ 0 ])) return res if __name__ == '__main__': V = 3 adj = [ [] for _ in
range ( V )] adj [ 0 ]. append ([ 1 , 5 ]); adj [ 1 ]. append ([ 0 , 5 ]); adj [ 1 ]. append ([ 2 , 3 ]); adj [ 2 ]. append ([ 1 , 3 ]);
adj [ 0 ]. append ([ 2 , 1 ]); adj [ 2 ]. append ([ 0 , 1 ]); print ( spanningTree ( V , adj ))

C# using System ; using System.Collections.Generic ; class GFG { // Returns total weight of the Minimum Spanning Tree
static int spanningTree ( int V , List < int []> adj ) { var pq = new PriorityQueue < int [], int > (); bool [] visited =
new bool [ V ] ; int res = 0 ; // Start from node 0
pq . Enqueue ( new int [] { 0 , 0 }, 0 ); while ( pq . Count > 0 ) { var p = pq . Dequeue (); int wt = p [ 0 ];
int u = p [ 1 ]; if ( visited [ u ] ) continue ; res += wt ; visited [ u ] = true ; // Push adjacent edges foreach ( var v in
adj [ u ] ) { if ( ! visited [ v [ 0 ] ] ) { pq . Enqueue ( new int [] { v [ 1 ], v [ 0 ]}, v [ 1 ]); } } } return res ; }

static void Main () { int V = 3 ; List < int []> adj = new List < int []> [ V ]; for ( int i = 0 ; i < V ; i ++ ) adj [ i ] =
new List < int []> (); adj [ 0 ]. Add ( new int [] { 1 , 5 }); adj [ 1 ]. Add ( new int [] { 0 , 5 }); adj [ 1 ]. Add ( new int [] { 2 , 3 });
adj [ 2 ]. Add ( new int [] { 1 , 3 }); adj [ 0 ]. Add ( new int [] { 2 , 1 }); adj [ 2 ]. Add ( new int [] { 0 , 1 });
Console . WriteLine ( spanningTree ( V , adj )); }

JavaScript class PriorityQueue {
constructor () { this . heap = []; } enqueue ( value ) { this . heap . push ( value ); let i = this . heap . length - 1 ;
while ( i > 0 ) { let j = Math . floor (( i - 1 ) / 2 ); if ( this . heap [ i ][ 0 ] >= this . heap [ j ][ 0 ]) { break ; }
[ this . heap [ i ], this . heap [ j ]] = [ this . heap [ j ], this . heap [ i ]]; i = j ; } } dequeue () { if ( this . heap .
length === 0 ) { throw new Error ( "Queue is empty" ); } let i = this . heap . length - 1 ; const result = this . heap [ 0 ];
this . heap [ 0 ] = this . heap [ i ]; this . heap . pop (); i -- ; let j = 0 ; while ( true ) { const left = j * 2 + 1 ;
if ( left > i ) { break ; } const right = left + 1 ; let k = left ; if ( right <= i && this . heap [ right ][ 0 ] < this . heap [ left ][ 0 ]) { k = right ; } if ( this . heap [ j ][ 0 ] <= this . heap [ k ][ 0 ]) { break ; }
[ this . heap [ j ], this . heap [ k ]] = [ this . heap [ k ], this . heap [ j ]]; j = k ; } return result ; } get count () { return this . heap . length ; } }

// Returns total weight of the Minimum Spanning Tree
function spanningTree ( V , adj ) { const pq = new PriorityQueue (); const visited = new Array ( V ). fill ( false );
let res = 0 ; // Start from node 0
pq . enqueue ([ 0 , 0 ]); while ( pq . count > 0 ) { const [ wt , u ] = pq . dequeue (); if ( visited [ u ]):
continue ; res += wt ; visited [ u ] = true ; // Push adjacent edges for ( const v of adj [ u ] ) { if ( ! visited [ v [ 0 ] ] ):
pq . enqueue ([ v [ 1 ], v [ 0 ]]); } } return res ; }

// Driver Code
const V = 3 ; const adj = Array . from ( { length : V , () => []}); adj [ 0 ]. push ([ 1 , 5 ]); adj [ 1 ]. push ([ 0 , 5 ]);
adj [ 1 ]. push ([ 2 , 3 ]); adj [ 2 ]. push ([ 1 , 3 ]); adj [ 0 ]. push ([ 2 , 1 ]); adj [ 2 ]. push ([ 0 , 1 ]); console . log ( spanningTree ( V , adj ));

Output 4 Time Complexity: O((E+V)*log(V)) where V is the number of vertex and E is the number of edges
Auxiliary Space: O(E+V) where V is the number of vertex and E is the number of edges
How Does it Work? The core is based on a fundamental property of MSTs called the cut property (If we partition the vertices into two groups (a cut), then the lightest edge that crosses that cut must be part of

```

some MST of the graph. Since, we always consider cut vertices, a cycle is never formed. Advantages and Disadvantages of Prim's algorithm Advantages: Prim's algorithm is guaranteed to find the MST in a connected, weighted graph. It has a time complexity of $O((E+V)*\log(V))$ using a binary heap or Fibonacci heap, where E is the number of edges and V is the number of vertices. It is a relatively simple algorithm to understand and implement compared to some other MST algorithms. Disadvantages: Like Kruskal's algorithm, Prim's algorithm can be slow on dense graphs with many edges, as it requires iterating over all edges at least once. Prim's algorithm relies on a priority queue, which can take up extra memory and slow down the algorithm on very large graphs. The choice of starting node can affect the MST output, which may not be desirable in some applications. Also Check: Kruskal's Algorithm for MST Minimum cost to connect all cities Minimum cost to provide water Second Best Minimum Spanning Tree Check if an edge is a part of any MST Minimize count of connections Comment Article Tags: Article Tags: Graph Greedy DSA Amazon Minimum Spanning Tree Samsung Cisco Prim's Algorithm.MST + 4 More