# Prim’s algorithm

## Introduction

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.

**Time Complexity:** O(V2)
**Auxiliary Space:** O(V)

## Working

**Step 1:** Determine an arbitrary vertex as the starting vertex of the MST. We pick 0 in the below diagram.
**Step 2:** Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
**Step 3:** Find edges connecting any tree vertex with the fringe vertices.
**Step 4:** Find the minimum among these edges.
**Step 5:** 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.
**Step 6:** Return the MST and exit

## How to implement Prim's Algorithm?

- 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.

# Code

## 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 \tWeight")
        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)


# Driver's code
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++

```
#include <bits/stdc++.h>
```

```
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 \tWeight\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);
}

// Driver's code
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;
}
```

## Javascript

```
// Number of vertices in the graph
let V = 5;

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
function minKey(key, mstSet) {
    // Initialize min value
    let min = Number.MAX_VALUE, min_index = -1;

    for (let v = 0; v < V; v++)
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            min_index = v;
        }

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
function printMST(parent, graph) {
    console.log("Edge   Weight");
    for (let i = 1; i < V; i++)
        console.log(parent[i] + " - " + i + "    " +
graph[parent[i]][i]);
}

// Function to construct and print MST for
// a graph represented using adjacency matrix
function primMST(graph) {
    // Array to store constructed MST
    let parent = new Array(V);

    // Key values used to pick minimum weight edge in cut
    let key = new Array(V);

    // To represent set of vertices included in MST
    let mstSet = new Array(V);

    // Initialize all keys as INFINITE
    for (let i = 0; i < V; i++) {
        key[i] = Number.MAX_VALUE;
        mstSet[i] = false;
    }
    // Always include first vertex in MST.
    key[0] = 0;
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (let count = 0; count < V - 1; count++) {
        // Pick the minimum key vertex from the set of vertices
not yet included in MST
        let 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.
        for (let v = 0; v < V; v++) {
            // graph[u][v] is non-zero only for adjacent vertices
of u
            // 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] && graph[u][v] < key[v])
{
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    // Print the constructed MST
    printMST(parent, graph);
}

// Driver code
let 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);
```

## Advantages

- Guarantees the optimal (minimum weight) spanning tree.
- Efficient with appropriate data structures (min-heap).

## Disadvantages

- Only works for connected graphs.
- Less efficient than Kruskal's algorithm in **sparse graphs**.