DSA Prim's Algorithm

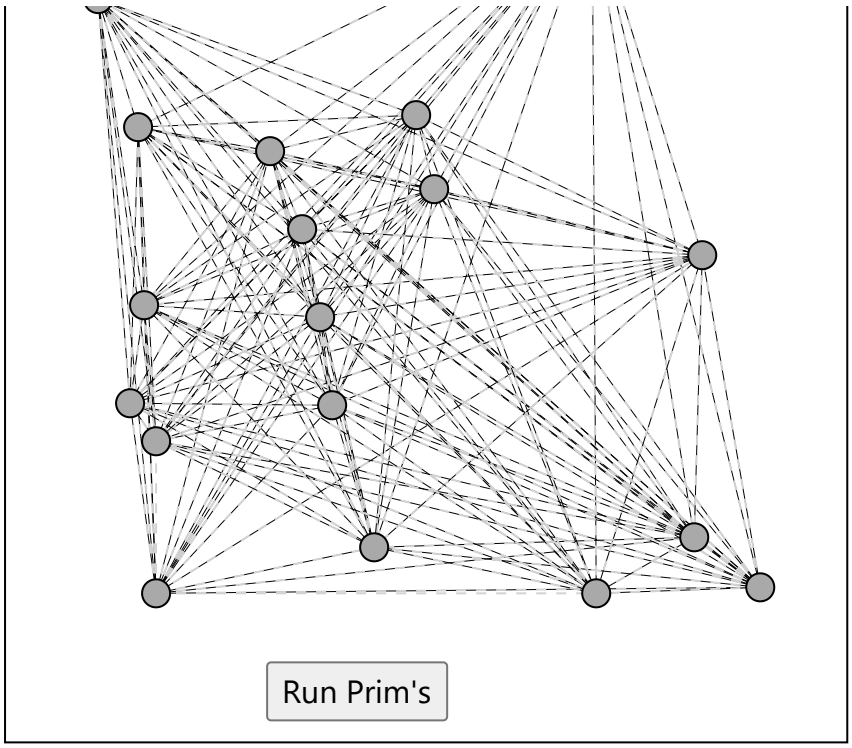
‹ Previous Next ›

Prim's algorithm was invented in 1930 by the Czech mathematician Vojtěch Jarník.

The algorithm was then rediscovered by Robert C. Prim in 1957, and also rediscovered by Edsger W. Dijkstra in 1959. Therefore, the algorithm is also sometimes called "Jarník's algorithm", or the "Prim-Jarník algorithm".

Prim's Algorithm

Prim's algorithm finds the Minimum Spanning Tree (MST) in a connected and undirected graph.



The MST found by Prim's algorithm is the collection of edges in a graph, that connects all vertices, with a minimum sum of edge weights.

Prim's algorithm finds the MST by first including a random vertex to the MST. The algorithm then finds the vertex with the lowest edge weight from the current MST, and includes that to the MST. Prim's algorithm keeps doing this until all nodes are included in the MST.

Prim's algorithm is greedy, and has a straightforward way to create a minimum spanning tree.

For Prim's algorithm to work, all the nodes must be connected. To find the MST's in an unconnected graph, Kruskal's algorithm can be used instead. You can read about Kruskal's algorithm on the next page.

How it works:

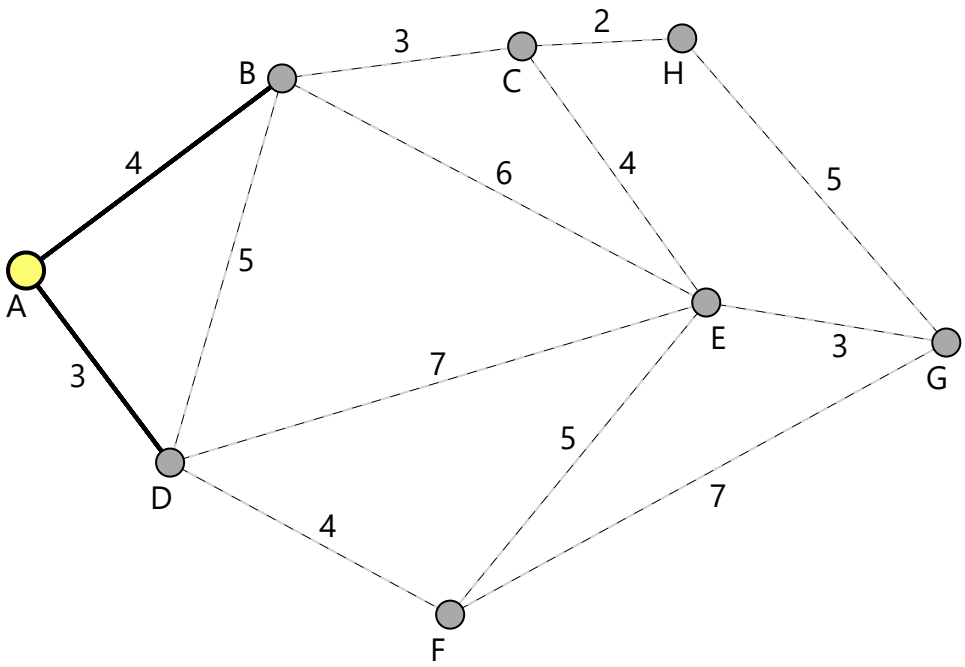
1. Choose a random vertex as the starting point, and include it as the first vertex in the MST.
2. Compare the edges going out from the MST. Choose the edge with the lowest weight that connects a vertex among the MST vertices to a vertex outside the MST.
3. Add that edge and vertex to the MST.

NOTE: Since the starting vertex is chosen at random, it is possible to have different edges included in the MST for the same graph, but the total edge weight of the MST will still have the same minimum value.

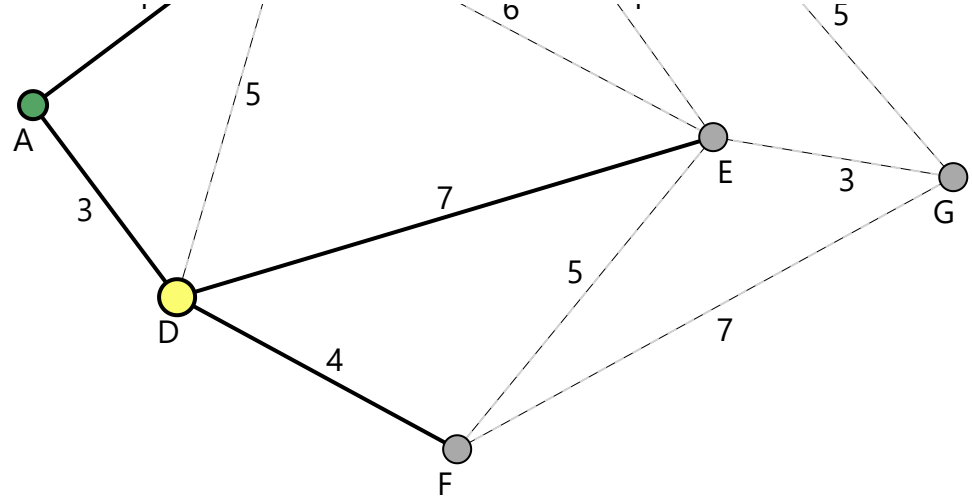
Manual Run Through

Let's run through Prim's algorithm manually on the graph below, so that we understand the detailed step-by-step operations before we try to program it.

Prim's algorithm starts growing the Minimum Spanning Tree (MST) from a random vertex, but for this demonstration vertex A is chosen as the starting vertex.



From vertex A, the MST grows along the edge with the lowest weight. So vertices A and D now belong to the group of vertices that belong to the Minimum Spanning Tree.



A `parents` array is central to how Prim's algorithm grows the edges in the MST.

At this point, the `parents` array looks like this:

```
parents = [-1,  0, -1,  0,  3,  3, -1, -1]
#vertices [ A,  B,  C,  D,  E,  F,  G,  H]
```

Vertex A, the starting vertex, has no parent, and has therefore value `-1`. Vertex D's parent is A, that is why D's parent value is `0` (vertex A is located at index 0). B's parent is also A, and D is the parent of E and F.

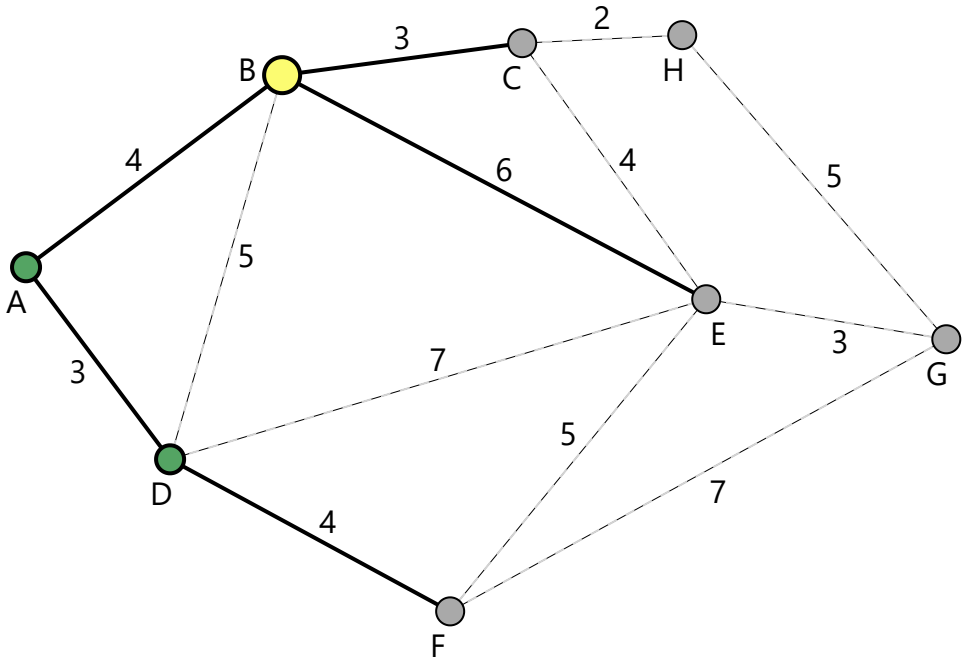
The `parents` array helps us to keep the MST tree structure (a vertex can only have one parent).

Also, to avoid cycles and to keep track of which vertices are currently in the MST, the `in_mst` array is used.

The `in_mst` array currently looks like this:

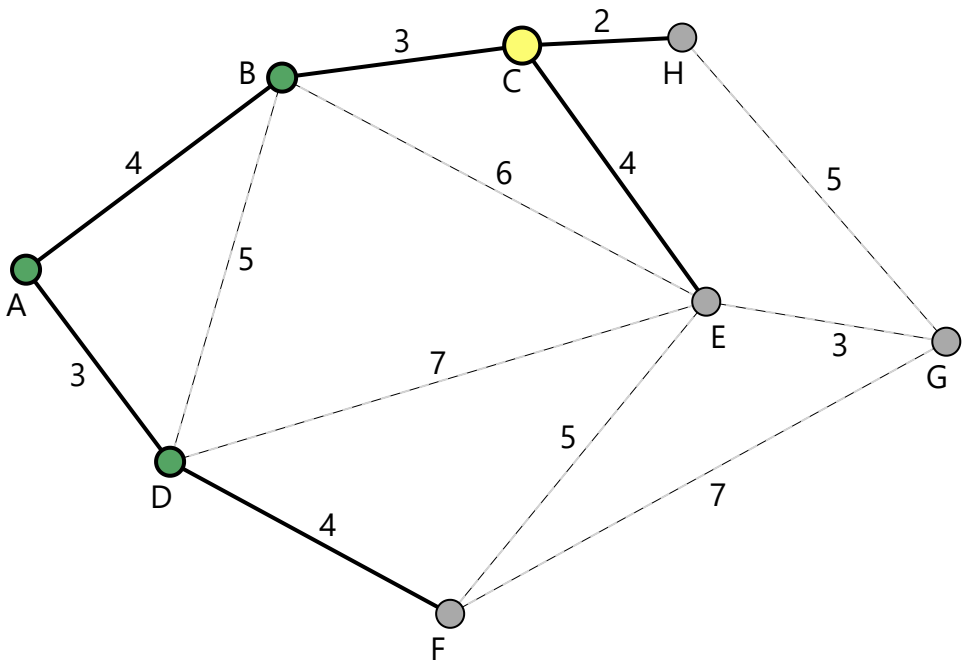
```
 in_mst = [ true, false, false,  true, false, false, false, false]
#vertices [    A,     B,     C,     D,     E,     F,     G,     H]
```

the next MST vertex. We choose B as the next MST vertex for this demonstration.



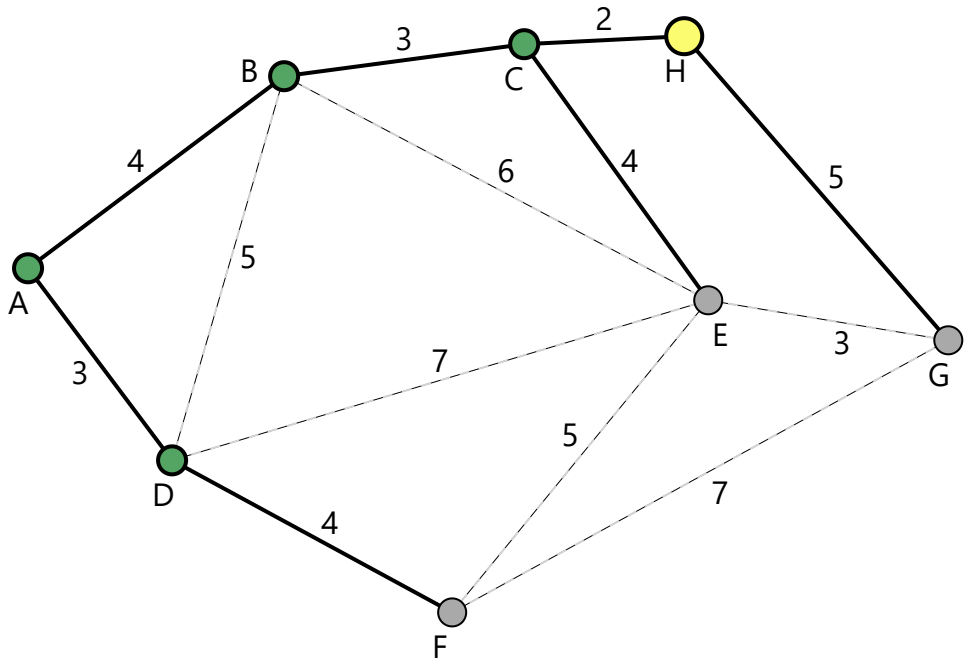
As you can see, the MST edge to E came from vertex D before, now it comes from vertex B, because B-E with weight `6` is lower than D-E with weight `7`. Vertex E can only have one parent in the MST tree structure (and in the `parents` array), so B-E and D-E cannot both be MST edges to E.

The next vertex in the MST is vertex C, because edge B-C with weight `3` is the shortest edge weight from the current MST vertices.



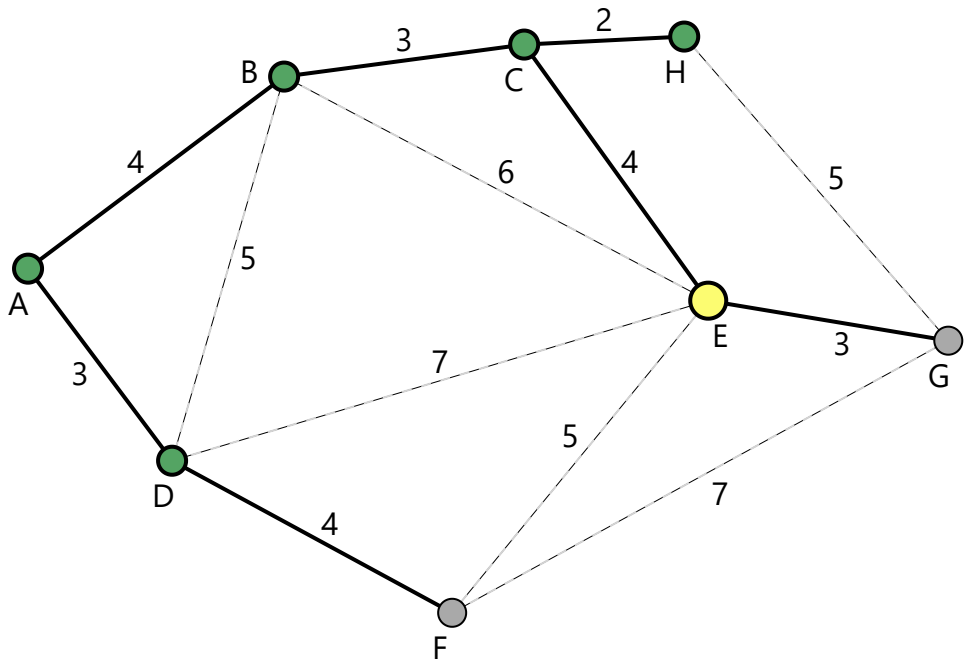
with edge weight `2`.

Vertex H is the next to be included in the MST, as it has the lowest edge weight `6`, and vertex H becomes the parent of vertex G in the `parents` array.

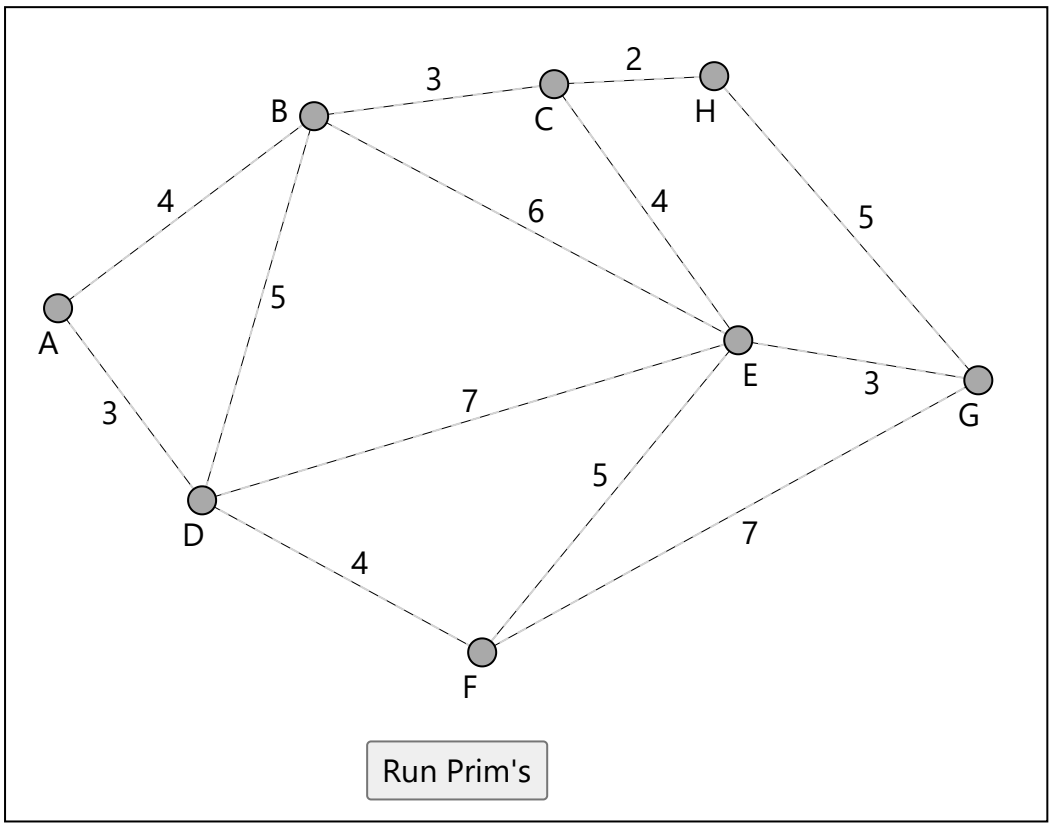


The next vertex to be included in the MST is either E or F because they have both the lowest edge weight to them: `4`.

We choose vertex E as the next vertex to be included in the MST for this demonstration.



Run the simulation below to see Prim's algorithm doing the manual steps that we have just done.



Implementation of Prim's Algorithm

For Prim's algorithm to find a Minimum Spanning Tree (MST), we create a `Graph` class. We will use the methods inside this `Graph` class later to create the graph from the example above, and to run Prim's algorithm on it.

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
```

```
                self.vertex_data[vertex] = data
```

Line 3-5: At first, the adjacency matrix is empty, meaning there are no edges in the graph. Also, the vertices have no names to start with.

Line 7-10: The `add_edge` method is for adding an edge, with an edge weight value, to the undirected graph.

Line 12-14: The `add_vertex_data` method is used for giving names to the vertices, like for example 'A' or 'B'.

Now that the structure for creating a graph is in place, we can implement Prim's algorithm as a method inside the `Graph` class:

```
16      def prims_algorithm(self):
20
22
23          print("Edge \tWeight")
24          for _ in range(self.size):
26
27              in_mst[u] = True
28
29              if parents[u] != -1:  # Skip printing for the first ver
30                  print(f"{self.vertex_data[parents[u]]}-{self.vertex
31
```

to each vertex outside the MST.

Line 19: The MST edges are stored in the `parents` array. Each MST edge is stored by storing the parent index for each vertex.

Line 21: To keep it simple, and to make this code run like in the "Manual Run Through" animation/example above, the first vertex (vertex A at index `0`) is set as the staring vertex. Changing the index to `4` will run Prim's algorithm from vertex E, and that works just as well.

Line 25: The index is found for the vertex with the lowest key value that is not yet part of the MST. Check out these explanations for `min` and `lambda` to better understand this Python code line.

Line 32-35: After a new vertex is added to the MST (line 27), this part of the code checks to see if there are now edges from this newly added MST vertex that can lower the key values to other vertices outside the MST. If that is the case, the `key_values` and `parents` arrays are updated accordingly. This can be seen clearly in the animation when a new vertex is added to the MST and becomes the active (current) vertex.

Now let's create the graph from the "Manual Run Through" above and run Prim's algorithm on it:

Example

Python:

```
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            self.adj_matrix[v][u] = weight  # For undirected grap

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
```

Line 32: We can actually avoid the last loop in Prim's algorithm by changing this line to `fo _ in range(self.size - 1):`. This is because when there is just one vertex not yet in th MST, the parent vertex for that vertex is already set correctly in the `parents` array, so the MST is actually already found at this point.

Time Complexity for Prim's Algorithm

For a general explanation of what time complexity is, visit this page.

With V as the number of vertices in our graph, the time complexity for Prim's algorithm is

$$O(V^2)$$

The reason why we get this time complexity is because of the nested loops inside the Prim's algorithm (one for-loop with two other for-loops inside it).

The first for-loop (line 24) goes through all the vertices in the graph. This has time complexity $O(V)$.

The second for-loop (line 25) goes through all the adjacent vertices in the graph to find the vertex with the lowest key value that is outside the MST, so that it can be the next vertex included in the MST. This has time complexity $O(V)$.

After a new vertex is included in the MST, a third for-loop (line 32) checks all other vertices to see if there are outgoing edges from the newly added MST vertex to vertices outside the MST that can lead to lower key values and updated parent relations. This also has time complexity $O(V)$.

Putting the time complexities together we get:

```
61
62
63
```

$$= O(V^2)$$

By using a priority queue data structure to manage key values, instead of using an array like we do here, the time complexity for Prim's algorithm can be reduced to:

$$O(E \cdot \log V)$$

Where E is the number of edges in the graph, and V is the number of vertices.

Such an implementation of Prim's algorithm using a priority queue is best for sparse graphs. A graph is sparse when the each vertex is just connected to a few of the other vertices.

‹ Previous Sign in to track progress Next ›

Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate