Bellman Bellman Ford Algorithm

```
class Graph:
  def __init__(self, vertices):
    self.V = vertices #nbr. of vert.
    self.edges = []
  def add_edge(self, u, v, w):
    self.edges.append((u, v, w))
  def bellman_ford(self, src):
    dist = [float("inf")]*self.V
    dist[src] = 0
    for _ in range(self.V -1):
      for u, v, w in self.edges:
        if dist[u] != float("inf")
           and dist[u] + w< dist[v]:
           dist[v] = w + dist[u]
    #check for negative weight cycles
    for u, v, w in self.edges:
      if dist[u] != float("Inf") and
            dist[u] +w < dist[v]:
            print("G has -ve cycles")
            return
```

Floyd Warshall (all pairs)

```
def floyd_warshall(graph):
  # copy input matrix
  dist = [[graph[i][j] for j in range(V)] for i range(V)]
  for k in range(V):
    for i in range(V):
      for j in range(V):
        dist[i][j] = min(dist[i][j],
                 dist[i][k] + dist[k][j])
```

Minimum Spanning Tree (PRIM)

```
class Graph:
  def __init__(self, graph vertices):
    self.V = vertices #count of vert.
    self.graph = graph
  #find vertex with min. key
  def min_key(self, key, mst_set):
    min_val = float("inf")
    min_idx = -1
    for v in range(self.V):
      if key[v] <min_val and
              not mst_set[v]:
        min_val = key[v]
        min_idx = v
    return min_idx
  def prim_mst(self):
    key = [float('inf')] *self.V
    parent = [None]* self.V
    key[0] = 0
    mst_set = [False]* self.V
    parent[0] = -1
    for _ in range(~~V~~self.V):
      if self.graph[u][v] >0 and
          (not (mst_set[v])) and
            key[v] > self.graph[u][v]:
            key[v] = self.graph[u][v]
            parent[v] = u
```

Union Find

```
class UnionFind:
  def __init__(self, n):
    self.parent = [i for i in range(n)]
    self.rank = [0] *n
  def find(self, x):
    if self.parent[x] != x:
      self.parent[x] = self.find(
                 self.parent[x])
    return self.parent[x]
  def union(self, x, y):
    root_x = self.find(x)
    root_y = self.find(y)
    if root_x != root_y:
      if self.rank[root_x] > self.rank[root_y]:
        self.parent[root_y] = root_x
      elif self.rank[root_x] < self.rank[root_y]:
        self.parent[root_x] = root_y
      else:
        self.parent[root_y] = root_x
        self.rank[root_x] += 1
```

Djikstra (single source)

```
def djikstra(graph, start):
  heap = []
  dist = {node: float('inf') for node
                 in graph}
  dist[start] = 0
  heapq.heappush(heap, (0, start))
  while heap:
    cur_dist, cur_vert = heapq.heappop(heap)
    if cur_dist > dist[cur_vert]:
      continue
    for neigh, weight in graph[cur_vert]:
      distance = cur_dist + weight
      if distance < dist[neigh]:
        dist[neigh] = distance
        heapq.heappush(heap,
          (distance, neigh))
  return dist
```