Representation

```
///Adjacency Matrix//////////////////
int V, E, A, B, W, g[1005][1005];
cin >> V >> E; memset(g, -1, sizeof(g));
for (int i = 0; i < E; i++) {
    cin >> A >> B >> W;
    //Weight, can set for both or single direction
    g[A][B] = W;
    g[B][A] = W;
}
///Adjacency List////////////////////
vector<pair<int, int> > g[1005];
int V, E, A, B, W;
cin >> V >> E;
for (int i = 0; i < E; i++) {
    cin >> A >> B >> W;
    g[A].push_back(make_pair(B, W));
    g[B].push_back(make_pair(A, W));
}
```

Floyd-Warshall

```
//initialise dist[i][j] to infinity at the start
for (int k=0;k<n;k++)
    for (int i=0;i<n;i++)
        for (int j=0;j<n;j++)
        // if there is a shorter path through node k, take it!
            dist[i][j] = min(dist[i][j], dist[i][k]+dist[k][j]);w
```

Floyd-Warshall algorithm uses the idea of triangle inequality, and is very
easy to code (just 4 lines!)

If there are negative cycles, dist[i][i] will be negative. Note the order!!!

Prim's Algorithm

```
//Lol just copied from hackerearth website
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <utility>
using namespace std;
const int MAX = 1e4 + 5;
typedef pair<long long, int> PII;
bool marked[MAX];
vector <PII> adj[MAX];
long long prim(int x)
{
    priority_queue<PII, vector<PII>, greater<PII> > Q;
    int y;
    long long minimumCost = 0;
    PII p;
    Q.push(make_pair(0, x));
    while(!Q.empty())
    {
        // Select the edge with minimum weight
        p = Q.top();
        Q.pop();
        x = p.second;
        // Checking for cycle
        if(marked[x] == true)
            continue;
        minimumCost += p.first;
        marked[x] = true;
```

Prim's Algorithm (cont)

```
        for(int i = 0;i < adj[x].size();++i)
        {
            y = adj[x][i].second;
            if(marked[y] == false)
                Q.push(adj[x][i]);
        }
    }
    return minimumCost;
}
int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0;i < edges;++i)
    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
    }
    // Selecting 1 as the starting node
    minimumCost = prim(1);
    cout << minimumCost << endl;
    return 0;
}
```

Used to Construct MST from Graph

Lowest Common Ancestor of Tree

```
ll lca(ll N,ll a,ll b){
    if(depth[a]<depth[b])
swap(a,b);
    //Equalise depth
    for(ll k=log2(N);k>=0;k--){
        ll parent = find_pare-
nt(a,k);//p[a][k]
        if(parent!=-1 && depth[-
parent]>=depth[b]){
            a=parent;
        }
    }
    if (a==b)return a;
    //Jump parent by parent
    for(ll k=log2(N);k>=0;k--){
        ll parent = find_pare-
nt(a,k);//p[a][k]
        ll parentb = find_pare-
nt(b,k);//p[b][k]
        if(parent!=parentb)a=-
parent,b=parentb;
    }
    return p[a][0];
}
```

Requires 2k Decomposition of Parents

Breadth First Search

```
vector<int> g[100005];
queue<pair<int, int> > q;
int dist[1000005];
fill(dist, dist+1000005, -1);
while (!q.empty()) {
    int v = q.front().first;
    int d = q.front().second;
    q.pop();
    if (dist[v] != -1) continue;
//Visited
    dist[v] = d;
    for (int i = 0; i < g[v].s-
ize(); i++) {
        q.push(make_pair(g[v]
[i], d+1));
    }
}
```

Time Complexity: O(|V| + |E|)
Space Complexity: O(b^d)
where d is the depth of the graph and b is the branching factor.

BFS is more suitable when the goal is close to the source, BFS is still faster in such cases.

We can use this algorithm to find the shortest path in a grid/unweighted graph

Bellman-Ford

```
dist[s]=0; //dist all others =
INF
for (int i=0; i<N-1; i++){
    for (int j=0; j<E; j++){
        // if path is shorter
through node u, take it!
        dist[v] = min(dist[v],
dist[u]+cost);
    }
}
```

Solves the Single Source Shortest Path (SSSP) problem. (shortest path from one node (source) to all other nodes)
Can be used with negative edges, Run the algorithm twice to detect for negative cycles

Time Complexity: O(VE)
Space Complexity: O(V)

Union Find Data Structure

```
int root (int x ) {
    if (x == parent [x]) return
x ;
    return root (parent[x]) ;
}
bool is_connected (int x,int y)
{
    return root (x) == root(y) ;
}
void connect ( int x , int y ) {
    int root_x = root (x);
    int root_y = root (y);
    if (root_x != root_y)
        parent [root_x] = root_y
;
}
////For Rankin-
g//////////////////////
int rank[N];
void connect (int x , int y) {
    int root_x = root (x) ,
root_y = root (y) ;
    if (root_x == root_y) return
; // same root
    if (rank[root_x] > rank[r-
oot_y]) {
        parent[root_y] = root_x
;
    } else if (rank[root_x] <
rank[root_y]) {
        parent[root_x] = root_y
;
    } else {
        parent[root_y] = root_x
;
        rank[root_x]++;
    }
}
```

Kruskal's Algorithm for MST

```
vector <tuple<int,int,int> >
edges ; // weight,node A,node B
sort (edges.begin(), edges.end
()) ;
int total_weight = 0;
for (auto e : edges) {
    int weight, a, b;
    tie (weight,a,b) = e ;
    if (root(a) == root(b)) //
taking this edge will cause a
cycle
        continue;
    total_weight += weight ; //
take this edge
    connect (a, b) ; // connect
them in the UFDS
}
```

Sort the list of edges by weight
For each edge in ascending order: If both nodes aren't already
connected, take it. Else, skip this edge.
Time complexity: O(E log V) (but faster than Prim's algorithm in
practice)
UFDS is needed to check if the nodes are connected in (2).

Depth First Search

```
bool vis[N];
vector<int> adjList[N];
void dfs(int node) {
    if (vis[node]) return;
    vis[node] = true;
    for (int a = 0; a < (int)a-
djList[node].size(); ++a)
        dfs(adjList[node][a]);
}
```

Depth First Search (cont)

```
///Iterat-
ive//////////////////////////
///
bool vis[N];
vector<int> adjList[N];
stack<int> S;
while (!S.empty()) {
    int node = S.top();
    S.pop();
    if (vis[node]) continue;
    vis[node] = true;
    for (int a = 0; a < (int)a-
djList[node].size(); ++a)
        S.push(adjList[node]
[a]);
}
```

DFS uses O(d) space, where d is the depth of the graph

DFS is not suited for infinite graphs.

Some applications of DFS include:
1. Topological Ordering (covered later)
2. Pre-/In-/Post-order numbering of a tree
3. Graph connectivity
4. Finding articulation points
5. Finding bridges

Dijkstra's Algorithm

```
vector<pair<int,int> >
adjList[10000]; // node, weight
priority_queue<pair<int,int>,
vector<pair<int,int> >, greate-
r<pair<int,int> > > pq; //
distance, node
int dist[10000];
memset(dist, -1, sizeof(dist));
pq.push(make_pair(0, source));
dist[0] = 0;
while(!pq.empty()){
```

Dijkstra's Algorithm (cont)

```
    pair<int,int> c = pq.top();
    pq.pop();
    if(c.first != dist[c.se-
cond]) continue;
    for(auto i : adjList[c.se-
cond]){
        if(dist[i.first] == -1
|| dist[i.first] > c.first +
i.second){
            dist[i.first] =
c.first + i.second;
            pq.push(make_pair(-
dist[i.first], i.first));
        }
    }
}
```

Time Complexity of our implementation: O(E log E)
Space Complexity: O(V+E)

Solves the Single Source Shortest Path (SSSP) problem. Means shortest path from one node to all other nodes. Cannot be used with negative edges as it runs too slow
Especially cannot be used with negative cycles

2k Parent Decomposition

```
typedef long long ll;
ll p[V][K]; //node,kth ancestor
//DFS to compute node parents
for p[i][0], first parent
bool visited[V];
ll depth[V];
void dfs(ll x){
    if (visited[x])return;
    visited[x]=true;
    for (auto i:adjlist[x]){
        if (!visited[i.first]){
```

2k Parent Decomposition (cont)

```
            if (p[i.first][0] == -1){
                //cout<<i.first<<"<-"
<<x<<endl;
                p[i.first][0] = x;
                depth[i.first] =
depth[x]+1;
            }
            dfs(i.first);
        }
    }
}
void calc_k_parents(ll N){
    for (ll k=1;k<K;k++){
        for (ll i=0;i<N;i++){
            if (p[i][k-1] != -1){
                p[i][k]= p[p[i][k--
1]][k-1];
            }else{p[i][k]=-1;}
           // if (k==2)cout<<i<<","
<<k<<":"<<p[i][k-1]<<","<<p[p[i][k-
1]][k-1]<<","<<p[i][k]<<endl;
        }
    }
}
ll find_parent(ll x,ll k){
    for (ll i=K;i>=0;i--){
        if (k>= (1<<i)){
            if (x==-1)return -1;
            x=p[x][i];
            k-=1<<i;
        }
    }
    return x;
```

2k Parent Decomposition (cont)

```
}
```