Kosaraju’s Algorithm

```
//kosaraju's algorithm
std::vector<int> graph[N],rev_graph[N];
std::vector<bool> used;
std::vector<int> order,component;

int n;


void dfs1(int node){
  used[node]=true;
  for(auto nbr:graph[node]){
    if(!used[nbr])
      dfs1(nbr);
  }
  order.pb(node);
}

void dfs2(int node){
  used[node]=true;
  for(auto nbr:rev_graph[node]){
    if(!used[nbr])
      dfs2(nbr);
  }
  component.pb(node);
}


void get_component(){
  //code for that particular component

}

void kosaraju(){

  used.assign(n+1,false);
```

```
  order.clear();
  component.clear();

  for(int i=0;i<n;++i){
    if(!used[i])
      dfs1(i);
  }

  //order [3 5 4 2 1]

  reverse(all(order));

  used.assign(n+1,false);

  for(auto node:order){
    if(!used[node]){
      dfs2(node);
      //component vector will store the largest scc that this node is a
part of
      get_component();
      //get the max size of all components size
      component.clear();
    }
  }

}
```

Dijkstra Algorithm:

```
void dijkstra(){
  ll n;
  cin>>n;

  ll m;
  cin>>m;

  ll src;
  cin>>src;

  vector<pair<ll,ll>>adj[n+1];

  for(ll i=0;i<m;i++){
      ll u,v,w;
      cin>>u>>v>>w;
      adj[u].push_back({v,w});
      adj[v].push_back({u,w});
  }

  set<pair<ll,ll>>st; // {dis[node],node}

  ll dis[n+1];

  for(ll i=1;i<=n;i++)
  dis[i]=INF;

  dis[src]=0;

  st.insert({dis[src],src});

  while(!st.empty()){
      auto fst=*st.begin();
      st.erase(fst);

      ll node=fst.second;

      for(auto e:adj[node]){
          if(dis[e.first]>dis[node]+e.second){
```

```
                if(dis[e.first]!=INF)
                st.erase({dis[e.first],e.first});
                dis[e.first]=dis[node]+e.second;
                st.insert({dis[e.first],e.first});
            }
        }
    }
}

set :: erase -> O(logN)
Time Complexity: O(NLOGN) O(VLOGV+ELOGV)
Space Complexity: O(V+E)
```

```
Limitation of Dijsktra:
Doesn't work with negative edges !
```

BellmanFord Algorithm :

Can be used for finding shortest path from a single source in a graph with negative edges. Can also detect negative edge cycle if any.

The simplest implementation

```
struct edge
{
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve()
{
    vector<int> d (n, INF);
```

```
    d[v] = 0;
    vector<int> p (n - 1);
    int x;
    for (int i=0; i<n; ++i)
    {
        x = -1;
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);
                    p[e[j].b] = e[j].a;
                    x = e[j].b;
                }
    }

    if (x == -1)
        cout << "No negative cycle from " << v;
    else
    {
        int y = x;
        for (int i=0; i<n; ++i)
            y = p[y];

        vector<int> path;
        for (int cur=y; ; cur=p[cur])
        {
            path.push_back (cur);
            if (cur == y && path.size() > 1)
                break;
        }
        reverse (path.begin(), path.end());

        cout << "Negative cycle: ";
        for (size_t i=0; i<path.size(); ++i)
            cout << path[i] << ' ';
    }
}
```

Flloyd Warshal Agorithm:

Used for finding all pair shortest path. Works in O(n^3) time complexity. And O(n^2) space complexity. It's a kind of dp.

```
for(int k=1;k<=n;k++){
    for(int i=1;i<=n;i++){
        for(int j=1;j<=n;j++){
            if(dis[i][k]+dis[k][j]<dis[i][j])
            dis[i][j]=dis[i][k]+dis[k][j];
        }
    }
}
```

Link to Rough