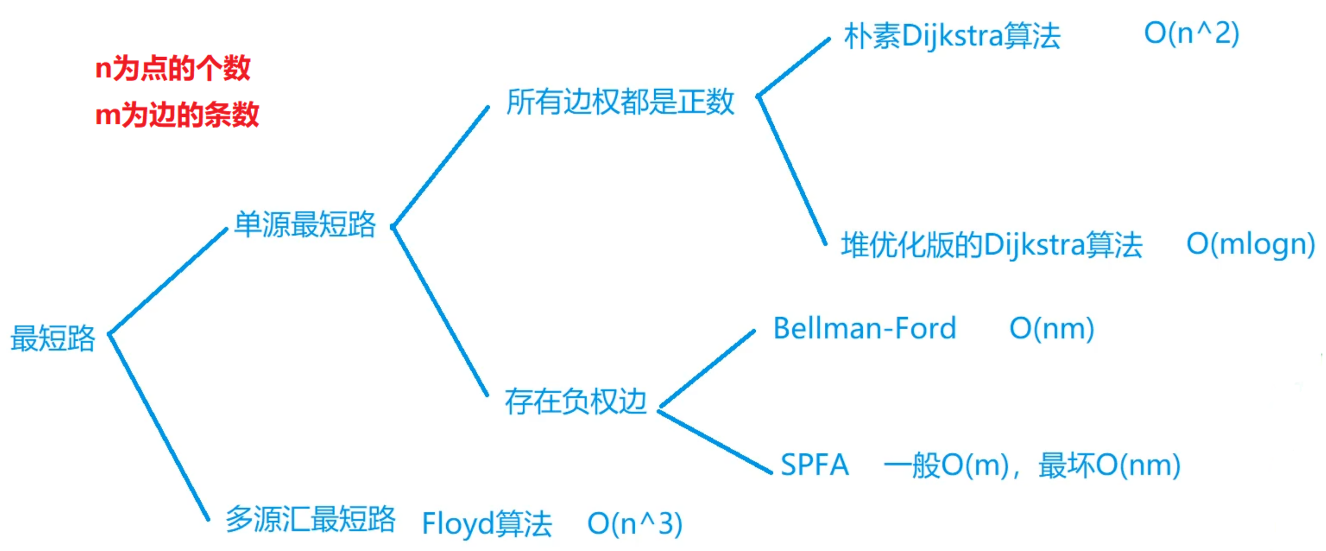

1. 朴素Dijkstra

```
int n,m;    //n是点的个数，m是边的条数
int g[N][N]; //邻接矩阵记录图
int dist[N]; //记录每个点的最短距离
bool vis[N]; //已经确定最短距离的点、

实现思路:

1.初始化
dis[1] = 0, dis[i] = INF;
第一个点到起点的距离为0，其余点的距离到起点的距离为正无穷

2.迭代n遍
for(int i = 0; i < n; i++){
    在所有未确定最短距离的点中，找到距离起点最短距离的点
    vis[t] = true;
    用t更新其他点的距离：从t出去的可以走到的边是不是最小
}
```

```
#include<iostream>
#include<cstring>
using namespace std;

const int N = 510;
```

```
int n,m;   //n是点的个数，m是边的条数
int g[N][N]; //邻接矩阵记录图
int dis[N]; //记录每个点的最短距离
bool vis[N]; //已经确定最短距离的点、

int dijkstra()
{
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

    for(int i = 0; i < n; i++){
        int t = -1;
        //在所有未确定最短距离的点中，找到距离起点最短距离的点
        for(int j = 1; j <= n; j++){
            if(!vis[j] && (t == -1 || dis[t] > dis[j]))
                t = j;
        }

        vis[t] = true;

        for(int j = 1; j <= n; j++){
            dis[j] = min(dis[j], dis[t] + g[t][j]);
        }
    }

    if(dis[n] == 0x3f3f3f3f) return -1;
    return dis[n];
}


int main()
{
    cin >> n >> m;
    memset(g, 0x3f, sizeof g);

    while(m--){
        int from,to,wei;
        cin >> from >> to >> wei;
        g[from][to] = min(wei, g[from][to]);
    }

```

```
    cout << dijkstra();

    return 0;
}
```

2. 堆优化版Dijkstra

```
优化思路:

//在所有未确定最短距离的点中，找到距离起点最短距离的点

主要是对上面这一步进行优化：因为这一步是要在所有未确定的点中，找到距离最小的一个点
所以就可以使用堆来优化，直接找到距离最小的点（不需要遍历n次，在堆顶就是最小的点）
```

```
typedef pair<int,int> PII;

struct node
{
    int to,w;
};

const int N = 15 * 1e4 + 10;
int n,m;
vector<node> g[N];
bool vis[N];
int dis[N];

int dijkstra()
{
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({1, 0});

    while(!heap.empty()){
        auto t = heap.top();
        heap.pop();
```

```cpp

        int id = t.first, distance = t.second;
        if(vis[id]) continue;

        for(int i = 0; i < g[id].size(); i++){
            int index = g[id][i].to;
            int weight = g[id][i].w;

            if(dis[index] > weight + distance){
                dis[index] = weight + distance;
                heap.push({index, dis[index]});
            }
        }
    }

    if(dis[n] == 0x3f3f3f3f) return -1;
    return dis[n];
}

```

3. Bellman-Ford算法（基本上不用！）

注意：bellman - ford算法擅长解决有边数限制的最短路问题

(1)什么是bellman - ford算法?

Bellman - ford 算法是求含负权图的单源最短路径的一种算法，效率较低，代码难度较小。其原理为连续进行松弛，在每次松弛时把每条边都更新一下，若在 n-1 次松弛后还能更新，则说明图中有负环，因此无法得出结果，否则就完成。

(通俗的来讲就是：假设 1 号点到 n 号点是可达的，每一个点同时向指向的方向出发，更新相邻的点的最短距离，通过循环 n-1 次操作，若图中不存在负环，则 1 号点一定会到达 n 号点，若图中存在负环，则在 n-1 次松弛后一定还会更新)

(2)bellman - ford算法的具体步骤

for n次

    备份dist数组

    for 所有边 a,b,w (松弛操作)

        dist[b] = min(dist[b],back[a] + w)

注意：back[] 数组是上一次迭代后 dist[] 数组的备份，由于是每个点同时向外出发，因此需要对 dist[] 数组进行备份，若不进行备份会因此发生串联效应，影响到下一个点

题目描述：

给定一个 n 个点 m 条边的有向图，图中可能存在重边和自环， 边权可能为负数。 请你求出从 1 号点到 n 号点的最多经过 k 条边的最短距离，如果无法从 1 号点走到 n 号点，输出 impossible。 注意：图中可能 存在负权回路 。

输入格式：

第一行包含三个整数 n,m,k。 接下来 m 行，每行包含三个整数 x,y,z，表示存在一条从点 x 到点 y 的有向边，边长为 z。

输出格式：

输出一个整数，表示从 1 号点到 n 号点的最多经过 k 条边的最短距离。 如果不存在满足条件的路径，则输出 impossible。

数据范围

$1 \le n, k \le 500$,
$1 \le m \le 10000$,
任意边长的绝对值不超过 10000。

输入样例：

```
3 3 1
1 2 1
2 3 1
1 3 3
```

输出样例：

```
3
```

```
#include<iostream>
#include<cstring>
using namespace std;

const int N = 510, M = 10010;

struct Edge
{
    int from,to,w;
}edges[M];

int n,m,k;
int dis[N],backup[N];

void Bellman_ford()
```

```
{
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

    for(int i = 0; i < k; i++){
        memcpy(backup, dis, sizeof dis);

        for(int j = 0; j < m; j++){
            int from = edges[j].from, to = edges[j].to, w = edges[j].w;

            dis[to] = min(dis[to], backup[from] + w);
        }
    }
}


int main()
{
    cin >> n >> m >> k;

    for(int i = 0; i < m; i++){
        int from,to,w;
        scanf("%d%d%d", &from, &to, &w);
        edges[i] = {from,to,w};
    }

    Bellman_ford();

    //在下面代码中，是否能到达n号点的判断中需要进行if(dist[n] > INF/2)判断，
    //而并非是if(dist[n] == INF)判断，原因是INF是一个确定的值，并非真正的无穷大，
    //会随着其他数值而受到影响，dist[n]大于某个与INF相同数量级的数即可
    if (dis[n] > 0x3f3f3f3f / 2) puts("impossible");
    else printf("%d\n", dis[n]);


    return 0;
}
```

4. SPFA算法（推荐使用）

（1）SPFA算法是对bellman_ford算法的优化

Bellman_ford算法会遍历所有的边，但是有很多的边遍历了其实没有什么意义，我们只用遍历那些到源点距离变小的点所连接的边即可，只有当一个点的前驱结点更新了，该节点才会得到更新；因此考虑到这一点，我们将创建一个队列每一次加入距离被更新的结点。

```
for(int j = 0; j < m; j++){
    int from = edges[j].from, to = edges[j].to, w = edges[j].w;

    dis[to] = min(dis[to], backup[from] + w);
}
```

只有当前驱节点的from更新，整个dis数组才会更新，否则后面更新的都是无效的

```
#include<iostream>
#include<vector>
#include<queue>
#include<cstring>
using namespace std;

const int N = 1e5 + 10;
const int M = 1e5 + 10;

struct Edge
{
    int to,w;
}edges[M];

vector<Edge> g[N];

int n,m;
int dis[N],cnt[N];
bool vis[N];

void spfa()
{
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

    queue<int> q;
```

```
    q.push(1);
    vis[1] = true;

    while(!q.empty())
    {
        auto t = q.front();
        q.pop();

        vis[t] = false;

        for(int i = 0; i < g[t].size(); i++){
            int to = g[t][i].to, w = g[t][i].w;

            if(dis[to] > dis[t] + w){
                dis[to] = dis[t] + w;
                if(!vis[to]){
                    q.push(to);
                    vis[to] = true;
                }
            }
        }
    }
}

int main()
{
    cin >> n >> m;
    for(int i = 0; i < m ; i++){
        int from, to, w;
        cin >> from >> to >> w;
        g[from].push_back({to, w});
    }

    spfa();

    if(dis[n] == 0x3f3f3f3f)    cout << "impossible";
    else cout << dis[n];

    return 0;
}
```

4. SPFA算法判断负环

（1）判断负环只需对spfa算法进行一些修改

维护一个cnt数组，这个数组的值是最远距离到这个点的所经过的边数。所以当cnt数组的值 >= n 时，说明经过了 n + 1个点，即经过了重复的点，既可说明有负环。

```
for(int i = 0; i < g[t].size(); i++){
    int to = g[t][i].to, w = g[t][i].w;

    if(dis[to] > dis[t] + w){
        dis[to] = dis[t] + w;
        cnt[to] = cnt[t] + 1;
        if(cnt[to] >= n)     return true;
        q.push(to);
        vis[to] = true;
    }
}
```

注意：我们要把所有点都push进队列。因可能从1出发无法到达有负环的边，所以要从所有点都出发一遍。

```
memset(dis, 0x3f, sizeof dis);
dis[1] = 0;

queue<int> q;
for(int i = 1; i <= n; i++) q.push(i);

vis[1] = true;
```

```
#include<iostream>
#include<vector>
#include<queue>
#include<cstring>
using namespace std;

const int N = 2010;
const int M = 10010;

struct Edge
{
    int to,w;
}edges[M];

vector<Edge> g[N];

int n,m;
int dis[N],cnt[N];
```

```
bool vis[N];

bool spfa()
{
    queue<int> q;
    for(int i = 1; i <= n; i++) q.push(i);

    vis[1] = true;

    while(!q.empty())
    {
        auto t = q.front();
        q.pop();

        vis[t] = false;

        for(int i = 0; i < g[t].size(); i++){
            int to = g[t][i].to, w = g[t][i].w;

            if(dis[to] > dis[t] + w){
                dis[to] = dis[t] + w;
                cnt[to] = cnt[t] + 1;
                if(cnt[to] >= n)    return true;
                q.push(to);
                vis[to] = true;
            }
        }
    }

    return false;
}

int main()
{
    cin >> n >> m;
    for(int i = 0; i < m ; i++){
        int from, to, w;
        cin >> from >> to >> w;
        g[from].push_back({to, w});
    }

    if(spfa())  cout << "Yes";
```

```
    else cout << "No";

    return 0;
}
```

4. Floyd算法

```
#include<iostream>
using namespace std;

const int N = 210, INF = 0x3f3f3f3f;
int d[N][N];
int n,m,k;

//就是一个三重循环
void floyd()
{
    for(int k = 1; k <= n; k++)
        for(int i = 1; i <= n; i++)
            for(int j = 1; j <= n; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

int main()
{
    cin >> n >> m >> k;

    //初始化数组
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            if(i == j)  d[i][j] = 0;     //表示从自己到自己的权重值为0，这个情况是自环
            else d[i][j] = INF;
        }
    }

    while(m--){
        int a,b,c;
        cin >> a >> b >> c;
        d[a][b] = min(d[a][b], c);
    }
```

```cpp

    floyd();

    while(k--){
        int a,b;
        cin >> a >> b;

        if(d[a][b] > INF / 2)  cout << "impossible" << endl;
        else cout << d[a][b] << endl;
    }

    return 0;
}
```

5. 最短路径的路径记录：

维护一个pre数组，记录每个节点的前驱节点

```cpp
int dijkstra()
{
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

    for(int i = 0; i < n; i++){
        int t = -1;
        for(int j = 1; j <= n; j++){
            if(!vis[j] && (t == -1 || dis[j] < dis[t]))
                t = j;
        }

        vis[t] = true;

        for(int j = 1; j <= n; j++){
            if(dis[t] + g[t][j] < dis[j]){
                dis[j] = dis[t] + g[t][j];
                pre[j] = t;    //记录下当前节点的前驱节点
            }
        }
    }
```

```

    if(dis[n] == 0x3f3f3f3f) return -1;
    else return dis[n];
}

//输出当前最短路的路径
int f = n; //f是当前最短路的终点
cout << n << ' ';
//从终点向前遍历路径
while(pre[f] != 0){
    cout << pre[f] << ' ';
    f = pre[f];
}
```