



## 1. 朴素Dijkstra

```
1 int n,m;    //n是点的个数，m是边的条数
2 int g[N][N]; //邻接矩阵记录图
3 int dist[N]; //记录每个点的最短距离
4 bool vis[N]; //已经确定最短距离的点、
5
6 实现思路：
7
8 1.初始化
9 dis[1] = 0, dis[i] = INF;
10 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
11
12 2.迭代n遍
13 for(int i = 0; i < n; i++){
14     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
15     vis[t] = true;
16     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
17 }
```

```
1 #include<iostream>
2 #include<cstring>
3 using namespace std;
4
5 const int N = 510;
```

```

6
7 int n,m;    //n是点的个数，m是边的条数
8 int g[N][N]; //邻接矩阵记录图
9 int dis[N]; //记录每个点的最短距离
10 bool vis[N]; //已经确定最短距离的点、
11
12 int dijkstra()
13 {
14     memset(dis, 0x3f, sizeof dis);
15     dis[1] = 0;
16
17     for(int i = 0; i < n; i++){
18         int t = -1;
19         //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
20         for(int j = 1; j <= n; j++){
21             if(!vis[j] && (t == -1 || dis[t] > dis[j]))
22                 t = j;
23         }
24
25         vis[t] = true;
26
27         for(int j = 1; j <= n; j++){
28             dis[j] = min(dis[j], dis[t] + g[t][j]);
29         }
30     }
31
32     if(dis[n] == 0x3f3f3f3f) return -1;
33     return dis[n];
34 }
35
36
37 int main()
38 {
39     cin >> n >> m;
40     memset(g, 0x3f, sizeof g);
41
42     while(m--){
43         int from,to,wei;
44         cin >> from >> to >> wei;
45         g[from][to] = min(wei, g[from][to]);
46     }
47

```

```
48     cout << dijkstra();
49
50     return 0;
51 }
```

## 2. 堆优化版Dijkstra

```
1  优化思路：
2
3  //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
4
5  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
6  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）
```

```
1  typedef pair<int,int> PII;
2
3  struct node
4  {
5      int to,w;
6  };
7
8  const int N = 15 * 1e4 + 10;
9  int n,m;
10 vector<node> g[N];
11 bool vis[N];
12 int dis[N];
13
14 int dijkstra()
15 {
16     memset(dis, 0x3f, sizeof dis);
17     dis[1] = 0;
18
19     priority_queue<PII, vector<PII>, greater<PII>> heap;
20     heap.push({1, 0});
21
22     while(!heap.empty()){
23         auto t = heap.top();
24         heap.pop();
```

```

25
26     int id = t.first, distance = t.second;
27     if(vis[id]) continue;
28
29     for(int i = 0; i < g[id].size(); i++){
30         int index = g[id][i].to;
31         int weight = g[id][i].w;
32
33         if(dis[index] > weight + distance){
34             dis[index] = weight + distance;
35             heap.push({index, dis[index]});
36         }
37     }
38 }
39
40 if(dis[n] == 0x3f3f3f3f) return -1;
41 return dis[n];
42 }
43

```

### 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)

$\text{dist}[b] = \min(\text{dist}[b], \text{back}[a] + w)$

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

$1 \leq n, k \leq 500$ ,  
 $1 \leq m \leq 10000$ ,  
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
```

```
1  #include<iostream>
2  #include<cstring>
3  using namespace std;
4
5  const int N = 510, M = 10010;
6
7  struct Edge
8  {
9      int from,to,w;
10 }edges[M];
11
12 int n,m,k;
13 int dis[N],backup[N];
14
15 void Bellman_ford()
```

```

16 {
17     memset(dis, 0x3f, sizeof dis);
18     dis[1] = 0;
19
20     for(int i = 0; i < k; i++){
21         memcpy(backup, dis, sizeof dis);
22
23         for(int j = 0; j < m; j++){
24             int from = edges[j].from, to = edges[j].to, w = edges[j].w;
25
26             dis[to] = min(dis[to], backup[from] + w);
27         }
28     }
29 }
30
31
32 int main()
33 {
34     cin >> n >> m >> k;
35
36     for(int i = 0; i < m; i++){
37         int from,to,w;
38         scanf("%d%d%d", &from, &to, &w);
39         edges[i] = {from,to,w};
40     }
41
42     Bellman_ford();
43
44     //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，
45     //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，
46     //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
47     if (dis[n] > 0x3f3f3f3f / 2) puts("impossible");
48     else printf("%d\n", dis[n]);
49
50
51     return 0;
52 }

```

## 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

```
1  #include<iostream>
2  #include<vector>
3  #include<queue>
4  #include<cstring>
5  using namespace std;
6
7  const int N = 1e5 + 10;
8  const int M = 1e5 + 10;
9
10 struct Edge
11 {
12     int to,w;
13 }edges[M];
14
15 vector<Edge> g[N];
16
17 int n,m;
18 int dis[N],cnt[N];
19 bool vis[N];
20
21 void spfa()
22 {
23     memset(dis, 0x3f, sizeof dis);
24     dis[1] = 0;
25
26     queue<int> q;
```

```

27     q.push(1);
28     vis[1] = true;
29
30     while(!q.empty())
31     {
32         auto t = q.front();
33         q.pop();
34
35         vis[t] = false;
36
37         for(int i = 0; i < g[t].size(); i++){
38             int to = g[t][i].to, w = g[t][i].w;
39
40             if(dis[to] > dis[t] + w){
41                 dis[to] = dis[t] + w;
42                 if(!vis[to]){
43                     q.push(to);
44                     vis[to] = true;
45                 }
46             }
47         }
48     }
49 }
50
51 int main()
52 {
53     cin >> n >> m;
54     for(int i = 0; i < m ; i++){
55         int from, to, w;
56         cin >> from >> to >> w;
57         g[from].push_back({to, w});
58     }
59
60     spfa();
61
62     if(dis[n] == 0x3f3f3f3f)    cout << "impossible";
63     else cout << dis[n];
64
65     return 0;
66 }

```



## 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  $\geq n$  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;
```

```
1 #include<iostream>
2 #include<vector>
3 #include<queue>
4 #include<cstring>
5 using namespace std;
6
7 const int N = 2010;
8 const int M = 10010;
9
10 struct Edge
11 {
12     int to,w;
13 }edges[M];
14
15 vector<Edge> g[N];
16
17 int n,m;
18 int dis[N],cnt[N];
```

```
19 bool vis[N];
20
21 bool spfa()
22 {
23     queue<int> q;
24     for(int i = 1; i <= n; i++) q.push(i);
25
26     vis[1] = true;
27
28     while(!q.empty())
29     {
30         auto t = q.front();
31         q.pop();
32
33         vis[t] = false;
34
35         for(int i = 0; i < g[t].size(); i++){
36             int to = g[t][i].to, w = g[t][i].w;
37
38             if(dis[to] > dis[t] + w){
39                 dis[to] = dis[t] + w;
40                 cnt[to] = cnt[t] + 1;
41                 if(cnt[to] >= n) return true;
42                 q.push(to);
43                 vis[to] = true;
44             }
45         }
46     }
47
48     return false;
49 }
50
51 int main()
52 {
53     cin >> n >> m;
54     for(int i = 0; i < m ; i++){
55         int from, to, w;
56         cin >> from >> to >> w;
57         g[from].push_back({to, w});
58     }
59
60     if(spfa()) cout << "Yes";
```

```
61     else cout << "No";
62
63     return 0;
64 }
```

## 4. Floyd算法

```
1  #include<iostream>
2  using namespace std;
3
4  const int N = 210, INF = 0x3f3f3f3f;
5  int d[N][N];
6  int n,m,k;
7
8  //就是一个三重循环
9  void floyd()
10 {
11     for(int k = 1; k <= n; k++)
12         for(int i = 1; i <= n; i++)
13             for(int j = 1; j <= n; j++)
14                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
15 }
16
17 int main()
18 {
19     cin >> n >> m >> k;
20
21     //初始化数组
22     for(int i = 1; i <= n; i++){
23         for(int j = 1; j <= n; j++){
24             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
25             else d[i][j] = INF;
26         }
27     }
28
29     while(m--){
30         int a,b,c;
31         cin >> a >> b >> c;
32         d[a][b] = min(d[a][b], c);
33     }
```

```

34
35     floyd();
36
37     while(k--){
38         int a,b;
39         cin >> a >> b;
40
41         if(d[a][b] > INF / 2) cout << "impossible" << endl;
42         else cout << d[a][b] << endl;
43     }
44
45     return 0;
46 }

```

## 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

```

1  int dijkstra()
2  {
3      memset(dis, 0x3f, sizeof dis);
4      dis[1] = 0;
5
6      for(int i = 0; i < n; i++){
7          int t = -1;
8          for(int j = 1; j <= n; j++){
9              if(!vis[j] && (t == -1 || dis[j] < dis[t]))
10                 t = j;
11            }
12
13            vis[t] = true;
14
15            for(int j = 1; j <= n; j++){
16                if(dis[t] + g[t][j] < dis[j]){
17                    dis[j] = dis[t] + g[t][j];
18                    pre[j] = t;    //记录下当前节点的前驱节点
19                }
20            }
21        }

```

```
22
23     if(dis[n] == 0x3f3f3f3f) return -1;
24     else return dis[n];
25 }
26
27 //输出当前最短路的路径
28 int f = n; //f是当前最短路的终点
29 cout << n << ' ';
30 //从终点向前遍历路径
31 while(pre[f] != 0){
32     cout << pre[f] << ' ';
33     f = pre[f];
34 }
```