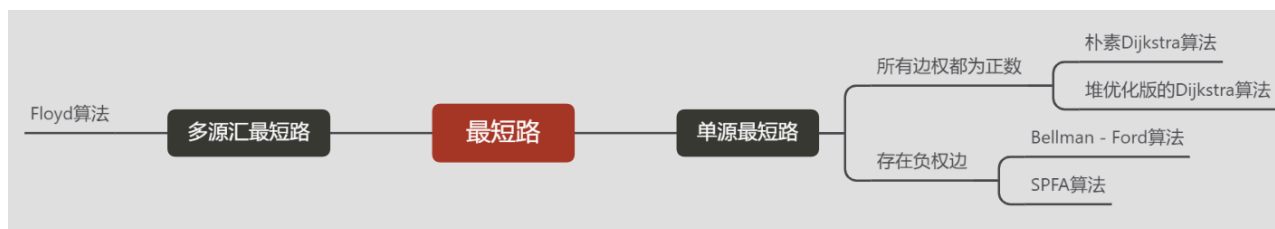


1: 知识汇总:



2: 朴素 Dijkstra算法: (用于稠密图, $O(n^2)$)

具体思路:

首先用一个集合 s 存储当前已确定最短路径的点

```
1 1: dist[1] = 0, dist[i] = 正无穷大
2 2: 迭代过程, i 从 1 - n 开始
3 t <--- 不在s中的距离最近的点
4 s <---- t;
5 用 t 更新其它点的距离
```

例题:

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, 所有边权均为正值。

请你求出 1 号点到 n 号点的最短距离, 如果无法从 1 号点走到 n 号点, 则输出 -1 。

输入格式

第一行包含整数 n 和 m 。

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 1 号点到 n 号点的最短距离。

如果路径不存在, 则输出 -1 。

输入样例:

```
3 3
1 2 2
2 3 1
1 3 4
```

输出样例:

```
3
```

```
1 import java.util.*;
2 public class Main{
```

```
3     public static int N = 510;
4     public static int n, m;
5     public static int max = 0x3f3f3f3f;
6     public static int[][] g = new int[N][N];
7     public static int[] dist = new int[N];
8     public static boolean[] st = new boolean[N];
9     public static int dijkstra(){
10         for(int i = 1; i <= n; i++){
11             dist[i] = max;
12         }
13         dist[1] = 0;
14         for(int i = 0; i < n; i++){
15             int t = -1;
16             for(int j = 1; j <= n; j++){
17                 if(!st[j] && (t == -1 || dist[j] < dist[t])){
18                     t = j;
19                 }
20             }
21             st[t] = true;
22             for(int j = 1; j <= n; j++){
23                 dist[j] = Math.min(dist[j], dist[t] + g[t][j]);
24             }
25         }
26         if(dist[n] == max){
27             return -1;
28         }else{
29             return dist[n];
30         }
31     }
32     public static void main(String[] args){
33         Scanner sc = new Scanner(System.in);
34         n = sc.nextInt();
35         m = sc.nextInt();
36         for(int i = 1; i <= n; i++){
37             for(int j = 1; j <= n; j++){
38                 g[i][j] = max;
39             }
40         }
41         while(m -- > 0){
42             int x = sc.nextInt();
```

```

43         int y = sc.nextInt();
44         int z = sc.nextInt();
45         g[x][y] = Math.min(g[x][y], z);
46     }
47     int res = dijkstra();
48     System.out.println(res);
49 }
50 }

```

3: 堆优化版的 Dijkstra算法: (用于稀疏图, $O(m \log n)$)

算法分析:

例题:

```

1  import java.io.*;
2  import java.util.*;
3  public class Main{
4      public static int N = 150010;
5      public static int n, m, idx;
6      public static int max = 0x3f3f3f3f;
7      public static int[] h = new int[N];
8      public static int[] e = new int[N];
9      public static int[] ne = new int[N];
10     public static int[] dist = new int[N];
11     public static boolean[] st = new boolean[N];
12     public static int[] w = new int[N];
13     public static void add(int x, int y, int z){
14         e[idx] = y;
15         w[idx] = z;
16         ne[idx] = h[x];
17         h[x] = idx;
18         idx++;
19     }
20     public static int dijkstra(){
21         PriorityQueue<PIIs> queue = new PriorityQueue<PIIs>();
22         for(int i = 1; i <= n; i++){

```

```

23         dist[i] = max;
24     }
25     dist[1] = 0;
26     queue.add(new PIIs(0, 1));
27     while(!queue.isEmpty()){
28         PIIs p = queue.poll();
29         int t = p.getSecond();
30         int distance = p.getFirst();
31         if(st[t]){
32             continue;
33         }
34         st[t] = true;
35         for(int i = h[t]; i != -1; i = ne[i]){
36             int j = e[i];
37             if(dist[j] > distance + w[i]){
38                 dist[j] = distance + w[i];
39                 queue.add(new PIIs(dist[j], j));
40             }
41         }
42     }
43     if(dist[n] == max){
44         return -1;
45     }else{
46         return dist[n];
47     }
48 }
49 public static void main(String[] args) throws IOException{
50     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
51     PrintWriter pw = new PrintWriter(new OutputStreamWriter(System.out));
52     String[] st = br.readLine().split(" ");
53     n = Integer.parseInt(st[0]);
54     m = Integer.parseInt(st[1]);
55     for(int i = 1; i <= n; i++){
56         h[i] = -1;
57     }
58     while(m -- > 0){
59         String[] str = br.readLine().split(" ");
60         int x = Integer.parseInt(str[0]);
61         int y = Integer.parseInt(str[1]);
62         int z = Integer.parseInt(str[2]);

```

```

63         add(x, y, z);
64     }
65     pw.println(dijkstra());
66     pw.flush();
67     br.close();
68     pw.close();
69 }
70 }
71 class PIIs implements Comparable<PIIs>{
72     private int first;
73     private int second;
74     public int getFirst(){
75         return this.first;
76     }
77     public int getSecond(){
78         return this.second;
79     }
80     public PIIs(int first, int second){
81         this.first = first;
82         this.second = second;
83     }
84     @Override
85     public int compareTo(PIIs o){
86         return Integer.compare(first, o.first);
87     }
88 }

```

4: Bellman - Ford 算法：（擅长解决有边数限制的最短路问题，可以用结构体存储边， $O(nm)$ ）

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)

具体步骤：

```

1 for n次 // 不超过 n 条边
2 for 所有边 a,b,w (松弛操作)
3 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
例题：

```

1 import java.util.*;
2 import java.io.*;
3 public class Main{
4     public static int N = 510;
5     public static int M = 100010;
6     public static int n, m , k;
7     public static int[] dist = new int[N];
8     public static Node[] list = new Node[M];
9     public static int max = 0x3f3f3f3f;
10    public static int[] back = new int[N];
11    public static void bellman_ford(){
12        for(int i = 1; i <= n; i++){
13            dist[i] = max;
14        }
15        dist[1] = 0;
16        for(int i = 0; i < k; i++){
17            back = Arrays.copyOf(dist, N);
18            for(int j = 0; j < m; j++){
19                Node node = list[j];
20                int x = node.x;
21                int y = node.y;
22                int z = node.z;
23                dist[y] = Math.min(dist[y], back[x] + z);
24            }
25        }
26        if(dist[n] > max / 2){ //是否能到达n号点的判断中需要进行if(dist[n] > INF/2)判断，
27                                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

```
28
29     System.out.println("impossible");
30 }else{
31     System.out.println(dist[n]);
32 }
33 }
34 public static void main(String[] args) throws IOException{
35     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
36     String[] str = br.readLine().split(" ");
37     n = Integer.parseInt(str[0]);
38     m = Integer.parseInt(str[1]);
39     k = Integer.parseInt(str[2]);
40     for(int i = 0; i < m; i++){
41         String[] st = br.readLine().split(" ");
42         int x = Integer.parseInt(st[0]);
43         int y = Integer.parseInt(st[1]);
44         int z = Integer.parseInt(st[2]);
45         list[i] = new Node(x, y, z);
46     }
47     bellman_ford();
48 }
49 }
50 class Node{
51     int x, y, z;
52     public Node(int x, int y, int z){
53         this.x = x;
54         this.y = y;
55         this.z = z;
56     }
57 }
```

5: SPFA 算法: (最坏情况下为 $O(nm)$, 一般为 $O(m)$)

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，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，将创建一个队列每一次加入距离被更新的结点

st数组的作用：判断当前的点是否已经加入到队列当中了；已经加入队列的结点就不需要反复的把该点加入到队列中了，就算此次还是会更新到源点的距离，那只用更新一下数值而不用加入到队列当中

SPFA算法看上去和Dijkstra算法长得有一些像但是其中的意义还是相差甚远的：

Dijkstra算法中的st数组保存的是当前确定了到源点距离最小的点，且一旦确定了最小那么就不可逆了(不可标记为true后改变为false)；SPFA算法中的st数组仅仅只是表示的当前发生过更新的点，且spfa中的st数组可逆(可以在标记为true之后又标记为false)

Dijkstra算法里使用的是优先队列保存的是当前未确定最小距离的点，目的是快速的取出当前到源点距离最小的点；SPFA算法中使用的是队列，目的只是记录一下当前发生过更新的点

Bellman_ford算法可以存在负权回路，是因为其循环的次数是有限制的因此最终不会发生死循环；但是SPFA算法不可以，由于用了队列来存储，只要发生了更新就会不断的入队，因此假如有负权回路请你不要用SPFA否则会死循环

SPFA求解最短路问题的思路：

bellman_ford算法对于所有的边都要遍历一遍，但是对于式子 $dist[b] = dist[a] + w$ ， $dist[b]$ 能更新的条件是 $dist[a]$ 被更新过，所以spfa算法就使用一个队列来保存所有被更新过的节点，每一次都用更新过的节点去更新其它节点

例题：

```
1 import java.io.*;
2 import java.util.*;
3 public class Main{
4     public static int N = 100010;
5     public static int n, m, idx;
6     public static int max = 0x3f3f3f3f;
7     public static int[] h = new int[N];
8     public static int[] e = new int[N];
9     public static int[] ne = new int[N];
10    public static int[] dist = new int[N];
11    public static boolean[] st = new boolean[N];
12    public static int[] w = new int[N];
13    public static void add(int x, int y, int z){
14        e[idx] = y;
15        w[idx] = z;
16        ne[idx] = h[x];
17        h[x] = idx;
18        idx ++;
19    }
```



```

20 public static void spfa(){
21     Arrays.fill(dist, max);
22     dist[1] = 0;
23     st[1] = true;
24     Queue<Integer> q = new LinkedList<Integer>();
25     q.add(1);
26     while(!q.isEmpty()){
27         int u = q.poll();
28         st[u] = false;
29         for(int i = h[u]; i != -1; i = ne[i]){
30             int j = e[i];
31             if(dist[j] > dist[u] + w[i]){
32                 dist[j] = dist[u] + w[i];
33                 if(!st[j]){
34                     q.add(j);
35                     st[j] = true;
36                 }
37             }
38         }
39     }
40     if(dist[n] == max){
41         System.out.println("impossible");
42     }else{
43         System.out.println(dist[n]);
44     }
45 }
46 public static void main(String[] args) throws IOException{
47     BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
48     String[] str = br.readLine().split(" ");
49     n = Integer.parseInt(str[0]);
50     m = Integer.parseInt(str[1]);
51     Arrays.fill(h, -1);
52     while(m -- > 0){
53         String[] st = br.readLine().split(" ");
54         int x = Integer.parseInt(st[0]);
55         int y = Integer.parseInt(st[1]);
56         int z = Integer.parseInt(st[2]);
57         add(x, y, z);
58     }
59     spfa();

```

```
60     }  
61 }
```

SPFA 算法求负环问题：

求负环的常用方法，基于SPFA

方法 1：统计每个点入队的次数，如果某个点入队n次，则说明存在负环

方法 2：统计当前每个点的最短路中所包含的边数，如果某点的最短路所包含的边数大于等于n，则也说明存在环

1: **dist [x]** 记录虚拟源点到x的最短距离

2: **cnt [x]** 记录当前x点到虚拟源点最短路的边数，初始每个点到虚拟源点的距离为0，只要他能再走n步，即 $\text{cnt} [x] \geq n$ ，则表示该图中一定存在负环，由于从虚拟源点到x至少经过n条边时，则说明图中至少有n + 1个点，表示一定有点是重复使用

3: 若 $\text{dist} [j] > \text{dist} [t] + w [i]$ ，则表示从t点走到j点能够让权值变少，因此进行对该点进行更新，并且对应 $\text{cnt} [j] = \text{cnt} [t] + 1$ ，往前走一步

4: 注意：判断是否存在负环，并非判断是否存在从1开始的负环，因此需要将所有的点都加入队列中，更新周围的点

例题：

```
1  import java.io.*;  
2  import java.util.*;  
3  public class Main{  
4      public static int N = 2010;  
5      public static int M = 100010;  
6      public static int n, m, idx;  
7      public static int[] h = new int[M];  
8      public static int[] e = new int[M];  
9      public static int[] ne = new int[M];  
10     public static int[] cnt = new int[M];  
11     public static int[] w = new int[M];  
12     public static int[] dist = new int[M];  
13     public static boolean[] st = new boolean[M];  
14     public static void add(int x, int y, int z){  
15         e[idx] = y;  
16         w[idx] = z;  
17         ne[idx] = h[x];  
18         h[x] = idx;
```

```
19         idx ++;
20     }
21     public static boolean spfa(){
22         Queue<Integer> q = new LinkedList<Integer>();
23         for(int i = 1; i <= n; i++){
24             q.add(i);
25             st[i] = true;
26         }
27         while(!q.isEmpty()){
28             int t = q.poll();
29             st[t] = false;
30             for(int i = h[t]; i != -1; i = ne[i]){
31                 int j = e[i];
32                 if(dist[j] > dist[t] + w[i]){
33                     dist[j] = dist[t] + w[i];
34                     cnt[j] = cnt[t] + 1;
35                     if(cnt[j] >= n){
36                         return true;
37                     }
38                     if(!st[j]){
39                         q.add(j);
40                         st[j] = true;
41                     }
42                 }
43             }
44         }
45         return false;
46     }
47     public static void main(String[] args) throws IOException{
48         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
49         String[] str = br.readLine().split(" ");
50         n = Integer.parseInt(str[0]);
51         m = Integer.parseInt(str[1]);
52         for(int i = 1; i <= n; i++){
53             h[i] = -1;
54         }
55         while(m -- > 0){
56             String[] st = br.readLine().split(" ");
57             int x = Integer.parseInt(st[0]);
58             int y = Integer.parseInt(st[1]);
```

```

59         int z = Integer.parseInt(st[2]);
60         add(x, y, z);
61     }
62     if(spfa()){
63         System.out.println("Yes");
64     }else{
65         System.out.println("No");
66     }
67 }
68 }

```

6: Floyd 算法: (时间复杂度: $O(n^3)$)

基本思路:

```

1 public static void floyd(){
2     for(int k = 1; k <= n; k++){
3         for(int i = 1; i <= n; i++){
4             for(int j = 1; j <= n; j++){
5                 g[i][j] = Math.min(g[i][j], g[i][k] + g[k][j]);
6             }
7         }
8     }
9 }

```

例题:

```

1 import java.util.*;
2 public class Main{
3     public static int N = 210;
4     public static int max = 0x3f3f3f3f;
5     public static int n, m, k;
6     public static int[][] g = new int[N][N];
7     public static void floyd(){
8         for(int k = 1; k <= n; k++){
9             for(int i = 1; i <= n; i++){
10                 for(int j = 1; j <= n; j++){
11                     g[i][j] = Math.min(g[i][j], g[i][k] + g[k][j]);
12                 }
13             }
14         }
15     }
16 }

```

```
13     }
14 }
15 }
16 public static void main(String[] args){
17     Scanner sc = new Scanner(System.in);
18     n = sc.nextInt();
19     m = sc.nextInt();
20     k = sc.nextInt();
21     for(int i = 1; i <= n; i++){
22         for(int j = 1; j <= n; j++){
23             if(i == j){
24                 g[i][j] = 0; // 可能询问自身到自身的距离，设置为0
25             }else{
26                 g[i][j] = max;
27             }
28         }
29     }
30     while(m -- > 0){
31         int x = sc.nextInt();
32         int y = sc.nextInt();
33         int z = sc.nextInt();
34         g[x][y] = Math.min(g[x][y], z);
35     }
36     floyd();
37     while(k -- > 0){
38         int x = sc.nextInt();
39         int y = sc.nextInt();
40         int t = g[x][y];
41         if(t >= max / 2){
42             System.out.println("impossible");
43         }else{
44             System.out.println(t);
45         }
46     }
47 }
48 }
```