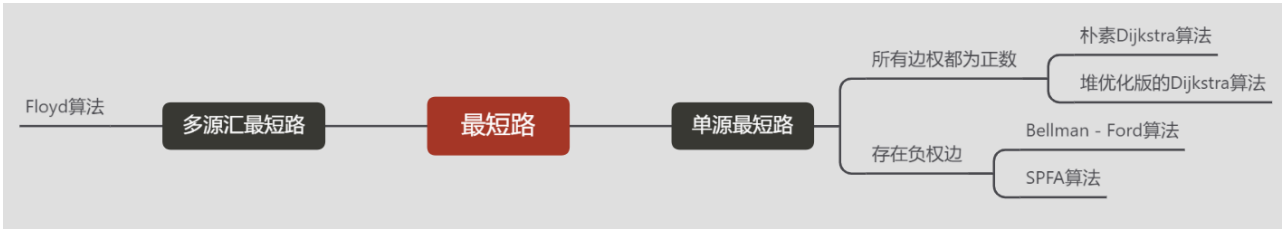
1：知识汇总：



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：dist[1] = 0, dist[i] = 正无穷大
2：迭代过程，i 从 1 - n 开始
t <--- 不在s中的距离最近的点
s <---- t;
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
```

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

```java
    public static int N = 510;
    public static int n, m;
    public static int max = 0x3f3f3f3f;
    public static int[][] g = new int[N][N];
    public static int[] dist = new int[N];
    public static boolean[] st = new boolean[N];
    public static int dijkstra(){
        for(int i = 1; i <= n; i++){
            dist[i] = max;
        }
        dist[1] = 0;
        for(int i = 0; i < n; i++){
            int t = -1;
            for(int j = 1; j <= n; j++){
                if(!st[j] && (t == -1 || dist[j] < dist[t])){
                    t = j;
                }
            }
            st[t] = true;
            for(int j = 1; j <= n; j++){
                dist[j] = Math.min(dist[j], dist[t] + g[t][j]);
            }
        }
        if(dist[n] == max){
            return -1;
        }else{
            return dist[n];
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        m = sc.nextInt();
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                g[i][j] = max;
            }
        }
        while(m -- > 0){
            int x = sc.nextInt();
```

```
            int y = sc.nextInt();
            int z = sc.nextInt();
            g[x][y] = Math.min(g[x][y], z);
        }
        int res = dijkstra();
        System.out.println(res);
    }
}
```

3：堆优化版的 Dijkstra算法：（用于稀疏图，O（mlogn））

算法分析：

例题：

```
import java.io.*;
import java.util.*;
public class Main{
    public static int N = 150010;
    public static int n, m, idx;
    public static int max = 0x3f3f3f3f;
    public static int[] h = new int[N];
    public static int[] e = new int[N];
    public static int[] ne = new int[N];
    public static int[] dist = new int[N];
    public static boolean[] st = new boolean[N];
    public static int[] w = new int[N];
    public static void add(int x, int y, int z){
        e[idx] = y;
        w[idx] = z;
        ne[idx] = h[x];
        h[x] = idx;
        idx ++;
    }
    public static int dijkstra(){
        PriorityQueue<PIIs> queue = new PriorityQueue<PIIs>();
        for(int i = 1; i <= n; i++){
```

```
            dist[i] = max;
        }
        dist[1] = 0;
        queue.add(new PIIs(0, 1));
        while(!queue.isEmpty()){
            PIIs p = queue.poll();
            int t = p.getSecond();
            int distance = p.getFirst();
            if(st[t]){
                continue;
            }
            st[t] = true;
            for(int i = h[t]; i != -1; i = ne[i]){
                int j = e[i];
                if(dist[j] > distance + w[i]){
                    dist[j] = distance + w[i];
                    queue.add(new PIIs(dist[j], j));
                }
            }
        }
        if(dist[n] == max){
            return -1;
        }else{
            return dist[n];
        }
    }
    public static void main(String[] args) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        PrintWriter pw = new PrintWriter(new OutputStreamWriter(System.out));
        String[] st = br.readLine().split(" ");
        n = Integer.parseInt(st[0]);
        m = Integer.parseInt(st[1]);
        for(int i = 1; i <= n; i++){
            h[i] = -1;
        }
        while(m -- > 0){
            String[] str = br.readLine().split(" ");
            int x = Integer.parseInt(str[0]);
            int y = Integer.parseInt(str[1]);
            int z = Integer.parseInt(str[2]);
```

```
            add(x, y, z);
        }
        pw.println(dijkstra());
        pw.flush();
        br.close();
        pw.close();
    }
}
class PIIs implements Comparable<PIIs>{
    private int first;
    private int second;
    public int getFirst(){
        return this.first;
    }
    public int getSecond(){
        return this.second;
    }
    public PIIs(int first, int second){
        this.first = first;
        this.second = second;
    }
    @Override
    public int compareTo(PIIs o){
        return Integer.compare(first, o.first);
    }
}
```

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：

```
for n次 // 不超过 n 条边
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

例题：

```
import java.util.*;
import java.io.*;
public class Main{
    public static int N = 510;
    public static int M = 100010;
    public static int n, m , k;
    public static int[] dist = new int[N];
    public static Node[] list = new Node[M];
    public static int max = 0x3f3f3f3f;
    public static int[] back = new int[N];
    public static void bellman_ford(){
        for(int i = 1; i <= n; i++){
            dist[i] = max;
        }
        dist[1] = 0;
        for(int i = 0; i < k; i++){
            back = Arrays.copyOf(dist, N);
            for(int j = 0; j < m; j++){
                Node node = list[j];
                int x = node.x;
                int y = node.y;
                int z = node.z;
                dist[y] = Math.min(dist[y], back[x] + z);
            }
        }
        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，
                               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
            System.out.println("impossible");
        }else{
            System.out.println(dist[n]);
        }
    }
    public static void main(String[] args) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] str = br.readLine().split(" ");
        n = Integer.parseInt(str[0]);
        m = Integer.parseInt(str[1]);
        k = Integer.parseInt(str[2]);
        for(int i = 0; i < m; i++){
            String[] st = br.readLine().split(" ");
            int x = Integer.parseInt(st[0]);
            int y = Integer.parseInt(st[1]);
            int z = Integer.parseInt(st[2]);
            list[i] = new Node(x, y, z);
        }
        bellman_ford();
    }
}
class Node{
    int x, y, z;
    public Node(int x, int y, int z){
        this.x = x;
        this.y = y;
        this.z = z;
    }
}
```

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Dijstra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：

```
import java.io.*;
import java.util.*;
public class Main{
    public static int N = 100010;
    public static int n, m, idx;
    public static int max = 0x3f3f3f3f;
    public static int[] h = new int[N];
    public static int[] e = new int[N];
    public static int[] ne = new int[N];
    public static int[] dist = new int[N];
    public static boolean[] st = new boolean[N];
    public static int[] w = new int[N];
    public static void add(int x, int y, int z){
        e[idx] = y;
        w[idx] = z;
        ne[idx] = h[x];
        h[x] = idx;
        idx ++;
    }
```

```
    public static void spfa(){
        Arrays.fill(dist, max);
        dist[1] = 0;
        st[1] = true;
        Queue<Integer> q = new LinkedList<Integer>();
        q.add(1);
        while(!q.isEmpty()){
            int u = q.poll();
            st[u] = false;
            for(int i = h[u]; i != -1; i = ne[i]){
                int j = e[i];
                if(dist[j] > dist[u] + w[i]){
                    dist[j] = dist[u] + w[i];
                    if(!st[j]){
                        q.add(j);
                        st[j] = true;
                    }
                }
            }
        }
        if(dist[n] == max){
            System.out.println("impossible");
        }else{
            System.out.println(dist[n]);
        }
    }
    public static void main(String[] args) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] str = br.readLine().split(" ");
        n = Integer.parseInt(str[0]);
        m = Integer.parseInt(str[1]);
        Arrays.fill(h, -1);
        while(m -- > 0){
            String[] st = br.readLine().split(" ");
            int x = Integer.parseInt(st[0]);
            int y = Integer.parseInt(st[1]);
            int z = Integer.parseInt(st[2]);
            add(x, y, z);
        }
        spfa();
```

```
    }
}
```

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cnt [x] >= 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dist [j] > 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j进行更新，并且对应cnt [j] = 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：

```java
import java.io.*;
import java.util.*;
public class Main{
    public static int N = 2010;
    public static int M = 100010;
    public static int n, m, idx;
    public static int[] h = new int[M];
    public static int[] e = new int[M];
    public static int[] ne = new int[M];
    public static int[] cnt = new int[M];
    public static int[] w = new int[M];
    public static int[] dist = new int[M];
    public static boolean[] st = new boolean[M];
    public static void add(int x, int y, int z){
        e[idx] = y;
        w[idx] = z;
        ne[idx] = h[x];
        h[x] = idx;
```

```
        idx ++;
    }
    public static boolean spfa(){
        Queue<Integer> q = new LinkedList<Integer>();
        for(int i = 1; i <= n; i++){
            q.add(i);
            st[i] = true;
        }
        while(!q.isEmpty()){
            int t = q.poll();
            st[t] = false;
            for(int i = h[t]; i != -1; i = ne[i]){
                int j = e[i];
                if(dist[j] > dist[t] + w[i]){
                    dist[j] = dist[t] + w[i];
                    cnt[j] = cnt[t] + 1;
                    if(cnt[j] >= n){
                        return true;
                    }
                    if(!st[j]){
                        q.add(j);
                        st[j] = true;
                    }
                }
            }
        }
        return false;
    }
    public static void main(String[] args) throws IOException{
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String[] str = br.readLine().split(" ");
        n = Integer.parseInt(str[0]);
        m = Integer.parseInt(str[1]);
        for(int i = 1; i <= n; i++){
            h[i] = -1;
        }
        while(m -- > 0){
            String[] st = br.readLine().split(" ");
            int x = Integer.parseInt(st[0]);
            int y = Integer.parseInt(st[1]);
```

```
            int z = Integer.parseInt(st[2]);
            add(x, y, z);
        }
        if(spfa()){
            System.out.println("Yes");
        }else{
            System.out.println("No");
        }
    }
}
```

6：Floyd 算法：(时间复杂度：O (n ^ 3))

基本思路：

```
public static void floyd(){
    for(int k = 1; k <= n; k++){
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <=n; j++){
               g[i][j] = Math.min(g[i][j],g[i][k] + g[k][j]);
            }
        }
    }
}
```

例题：

```
import java.util.*;
public class Main{
    public static int N = 210;
    public static int max = 0x3f3f3f3f;
    public static int n, m, k;
    public static int[][] g = new int[N][N];
    public static void floyd(){
        for(int k = 1; k <= n; k++){
            for(int i = 1; i <= n; i++){
                for(int j = 1; j <= n; j++){
                    g[i][j] = Math.min(g[i][j], g[i][k] + g[k][j]);
                }
```

```
            }
        }
    }
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        n = sc.nextInt();
        m = sc.nextInt();
        k = sc.nextInt();
        for(int i = 1; i <= n; i++){
            for(int j = 1; j <= n; j++){
                if(i == j){
                    g[i][j] = 0; // 可能询问自身到自身的距离，设置为0
                }else{
                    g[i][j] = max;
                }
            }
        }
        while(m -- > 0){
            int x = sc.nextInt();
            int y = sc.nextInt();
            int z = sc.nextInt();
            g[x][y] = Math.min(g[x][y], z);
        }
        floyd();
        while(k -- > 0){
            int x = sc.nextInt();
            int y = sc.nextInt();
            int t = g[x][y];
            if(t >= max / 2){
                System.out.println("impossible");
            }else{
                System.out.println(t);
            }
        }
    }
}
```