

本题揭示的问题是我对dfs返回不够理解深刻

unordered\_map<Node \*,Node \*> recall;//用来记录复制过的点

    Node\* cloneGraph(Node\* node) {

        if(node==nullptr){

            return node;

        }

        if(recall.find(node)!=recall.end()){

            return recall[node];

        }

        recall[node]=new Node (node->val);//深拷贝 相当与访问点的操作

        for(auto &key:node->neighbors){//在访问完全部的点的时候才会递归回来，填充复制节点邻接表

            recall[node]->neighbors.emplace\_back(cloneGraph(key));

        }

        return recall[node];

}

# Bfs 版本

unordered\_map<Node \*,Node \*> recall;

    Node\* cloneGraph(Node\* node) {

      if(node==nullptr){

          return nullptr;

      }

      queue <Node \*> help;

      help.push(node);

      recall[node]=new Node(node->val);//先访问该顶点深拷贝

      while(!help.empty()){

          Node \* tem=help.front();

          help.pop();

          for(auto &key:tem->neighbors){

//没访问过的节点放到队列中

              if(recall.find(key)==recall.end()){

                  recall[key]=new Node(key->val);

                  help.push(key);

              }

             recall[tem]->neighbors.emplace\_back(recall[key]);

          }

      }

      return recall[node];

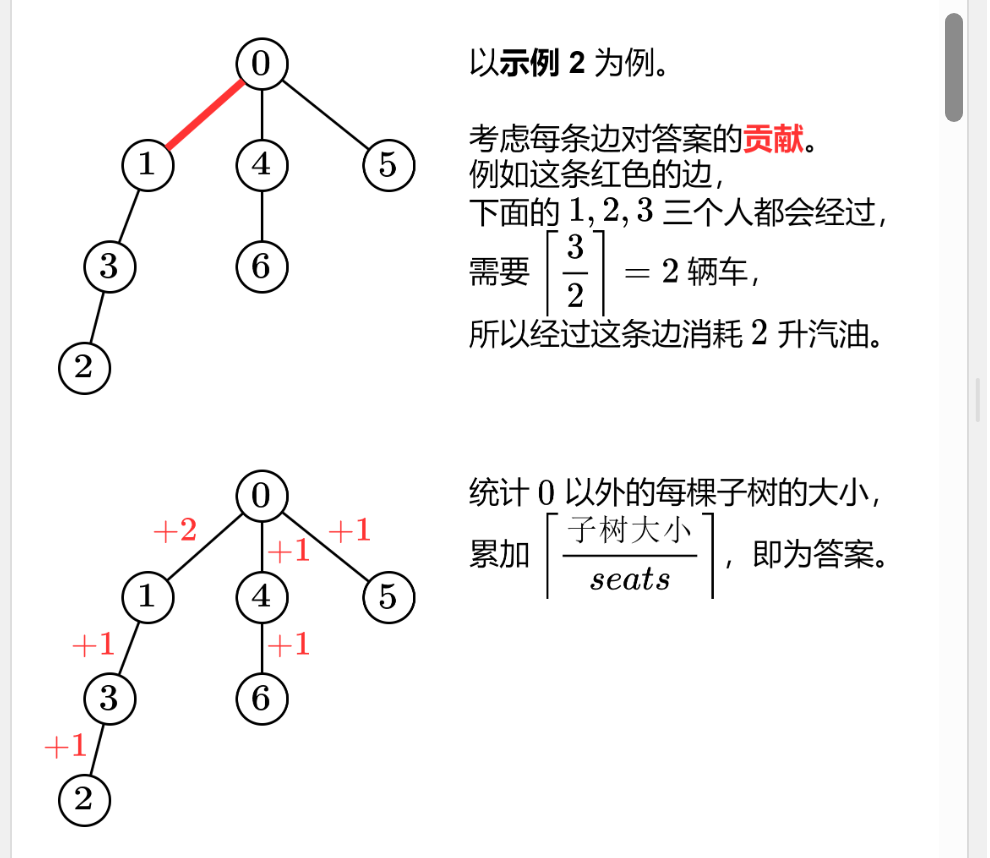
    }



Dfs 加贪心的运用

显然同一条路径上的人一打车划算，因此要计算每个节点的花费，可以计算以该节点为树的树的大小就可以得到从该节点出发到下一个城市需要的车辆---》知道油费。

通过传入父节点可以避免无限递归



long long minimumFuelCost(vector<vector<int>>& roads, int seats) {

        int n = roads.size() ;

        vector<vector<int>> g(n+1);

        for(auto &key : roads){//构建邻接表

            g[key[0]].push\_back(key[1]);

             g[key[1]].push\_back(key[0]);

        }

        long long  ans = 0;

        function<int(int,int)> dfs=[&](int cur ,int fa) ->int{

           int tree\_size = 1;

           for(auto &idx : g[cur]){

               if(idx!=fa){

                   tree\_size+=dfs(idx,cur);//计算树的大小

               }

           }

             if(cur ){

                  ans += (tree\_size-1)/seats +1;//计算花费

              }

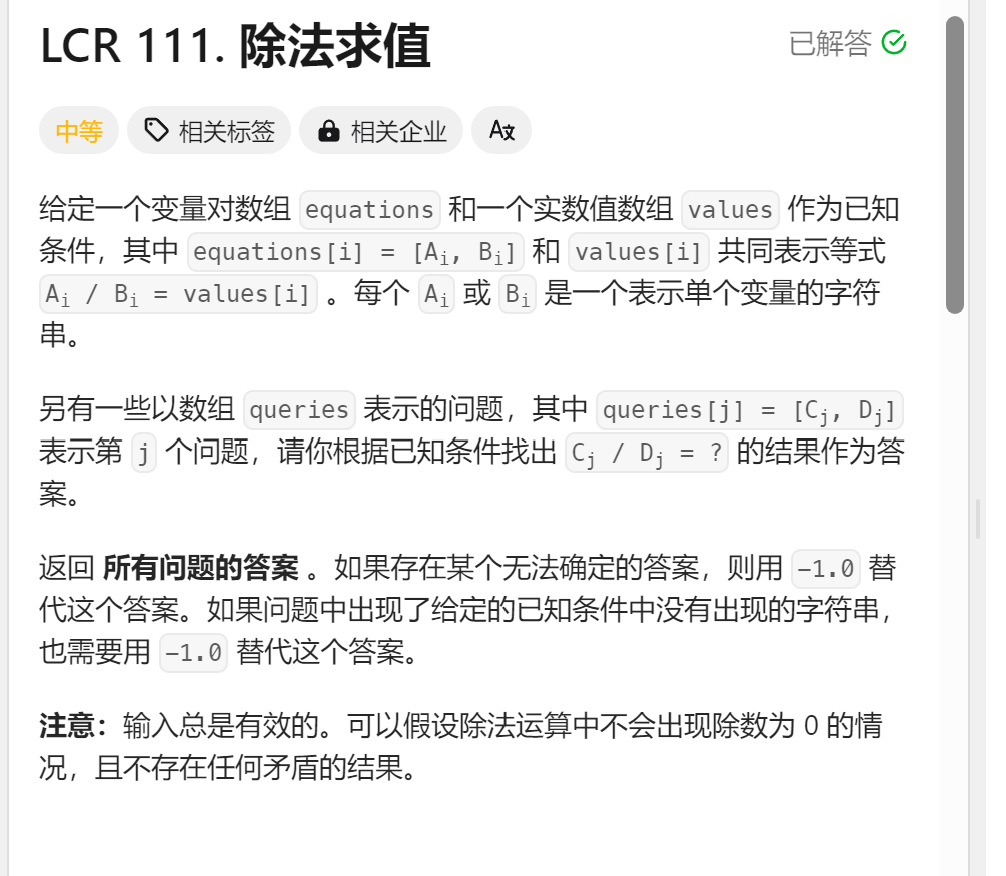
           return tree\_size;

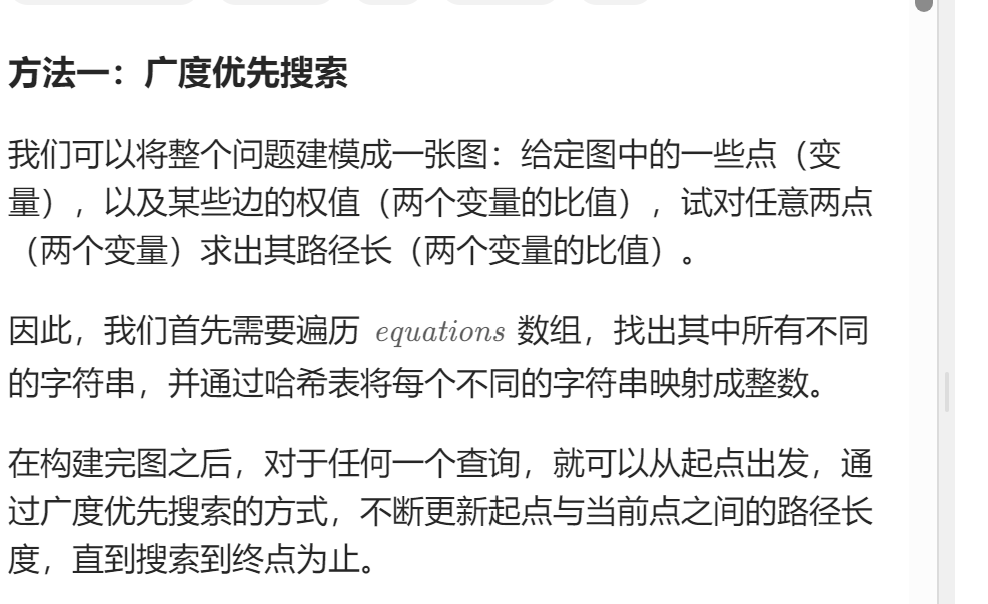
        };

        dfs(0,-1);

        return ans ;

    }





范例：用字符表示的图，应该学会将字符映射成熟悉的数字图。-----学会利用map来转换

Bfs的综合利用

 vector<double> calcEquation(vector<vector<string>>& equations, vector<double>& values, vector<vector<string>>& queries) {

     //将字符串表示的图映射成数字表示的图

     int nvars = 0 ; //映射的点的个数

     map<string,int> recall ;//用来记录字符串和 数字的映射情况

     int n = equations.size();

     for(int i = 0 ; i< n ; i++){

         if(recall.find(equations[i][0]) == recall.end()){

             recall[equations[i][0]] = nvars ++;

         }

         if(recall.find(equations[i][1]) == recall.end()){

             recall[equations[i][1]] = nvars ++;

         }

     }

     //将题目给的相除的信息转换为 每一个点到其直接相连的点的距离，并且存储每一个点与其所连接点的情况

    vector<vector<pair<int ,double>>> near\_edges(nvars);

    for(int i = 0 ; i < n ; i++ ){

        int va = recall[equations[i][0]],vb =recall[equations[i][1]];//获得映射的标号

        near\_edges[va].push\_back(make\_pair(vb,values[i]));

        near\_edges[vb].push\_back(make\_pair(va,1.0/values[i]));

    }

    vector<double> ans ; //需要返回的答案

    for(const auto& q :queries){

        double result  = -1.0; //搜索的结果

        if(recall.find(q[0])!=recall.end() && recall.find(q[1])!=recall.end()){//搜索的双方节点都已经存在

          if(q[0] == q[1]){//如果是自己到自己默认结果为1.0

            result = 1.0;

          }

          else {

              queue<int> help;//用来记录从起点开始的bfs的节点

              vector<double> road(nvars, -1.0);//road[i]用来记录从查询起点开始到 节点i 的距离, - 1.0 表明还没有遍历到

              help.push(recall[q[0]]);

              road[recall[q[0]]] = 1.0;//自己到自己距离为1.0；

              while(!help.empty()){

                  int x = help.front();

                  help.pop();

                  for(auto &[y,val] : near\_edges[x]){//bfs

                     if(road[y] < 0){

                         road[y] = road[x] \* val;

                         help.push(y);

                     }

                  }

              }

              result = road[recall[q[1]]];

          }

        }

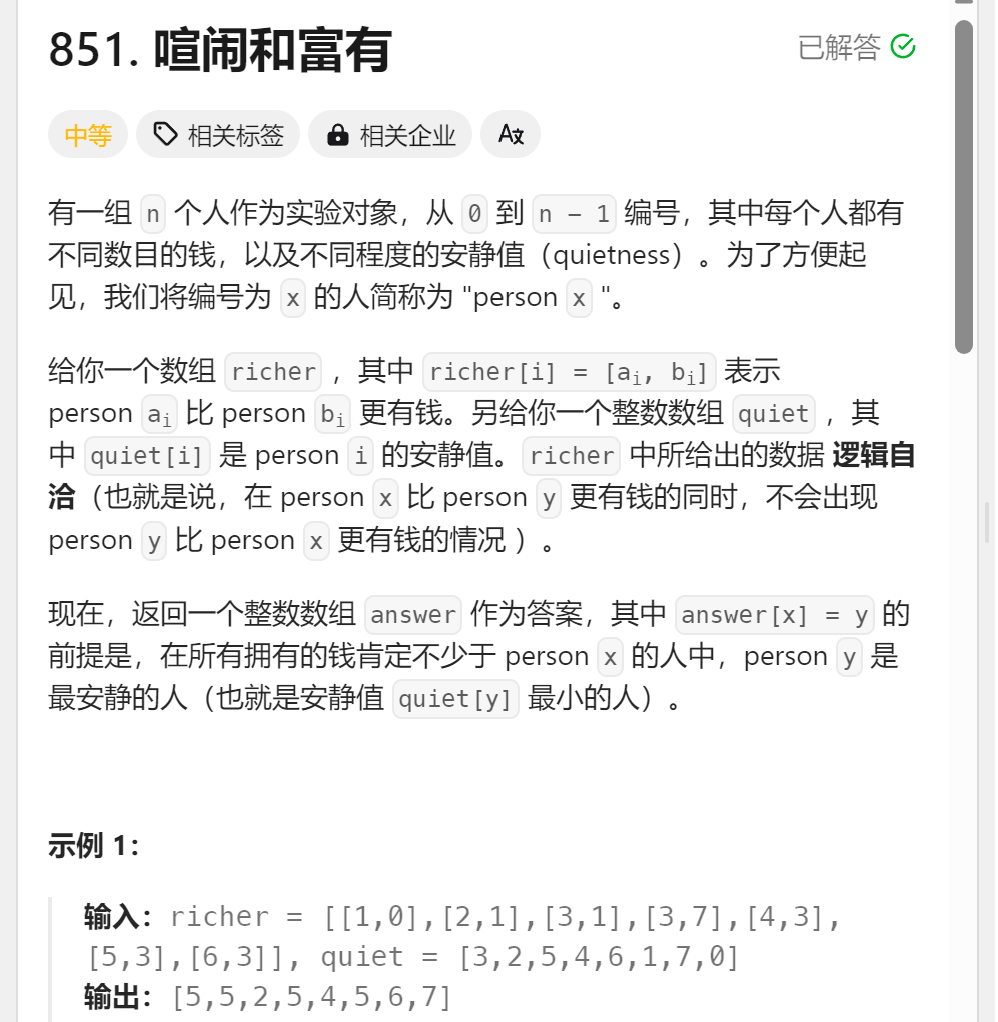
        //当前问题搜寻完将答案放入

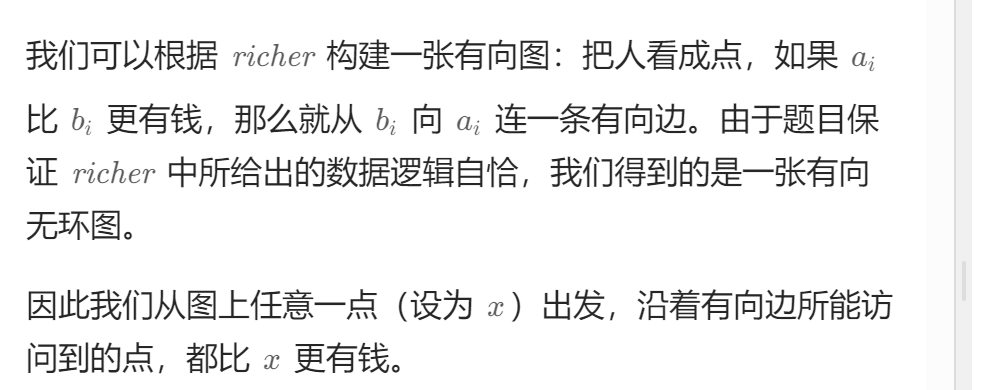
        ans.push\_back(result);

    }

    return ans ;

    }





vector<int> loudAndRich(vector<vector<int>>& richer, vector<int>& quiet){

         int n = quiet.size() ;

        vector<int> ans(n,-1);

        vector<vector<int>> near\_grip(n);

        for(const auto &key : richer){

            near\_grip[key[1]].push\_back(key[0]);

        }

      //巧妙的利用dfs 将最深处的节点的答案返回出来

      //从而避免重复的解决问题

      function<void(int)> dfs = [&](int x ){

             if(ans[x] !=-1){//记忆

                 return;

             }

             ans[x] = x ;

             for(auto &k : near\_grip[x]){//将大问题拆成多个子问题,返回的时候将子问题解决了

                 dfs(k);

                 if(quiet[ans[x]] > quiet[ans[k]]){

                     ans[x] = ans[k];//并查集的find函数类似的逻辑

                 }

             }

      };

      for(int i = 0 ; i < n ;i++){

          dfs(i);

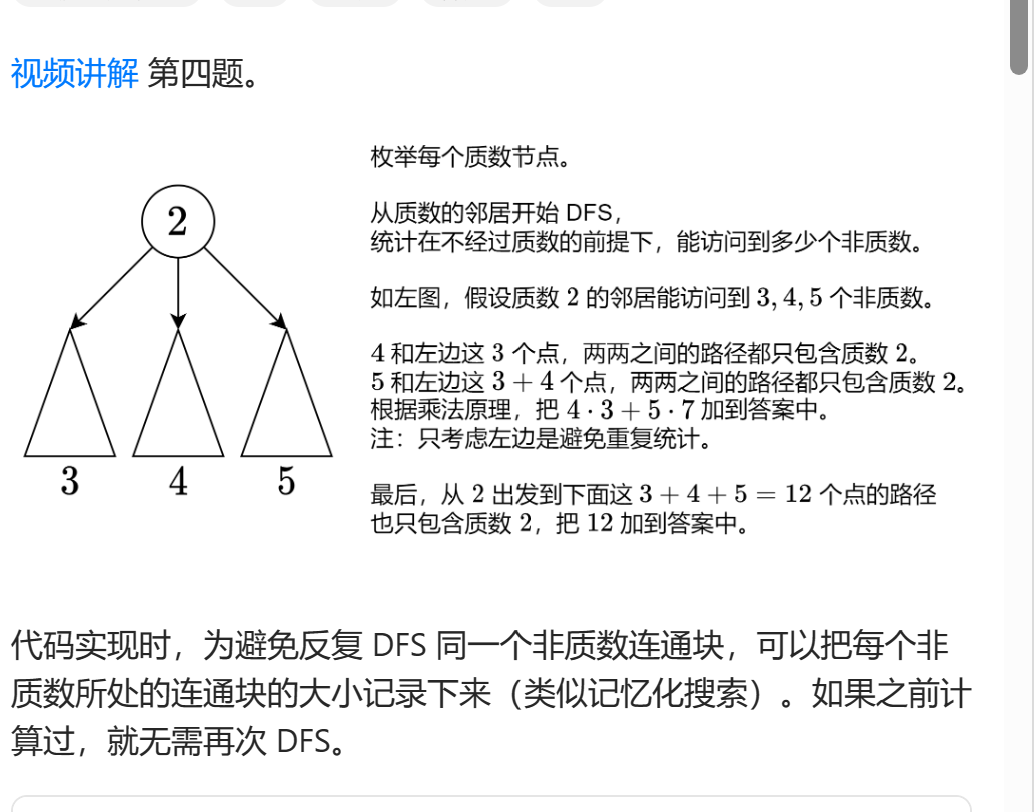
      }

       return ans ;

    }

};





Dfs + 记忆化搜索加并查集思想。

class Solution {

public:

    vector<bool> recall ; //用来记录质数有哪些

    vector<int> nodes ;

    void init(int n ){//埃式筛选法

        recall.resize(n+3,false);//质数flase 非质数true

        recall[1] =true;

        recall[0] =true ;

        for(int i = 2 ; i <= n ; i++){

            if(!recall[i]){

                if((long long) i \* i  <= n){

                    for(int j = i \* i ; j <= n ; j+= i){

                        recall[j] = true ;

                    }

                }

            }

        }

    }

    void get\_dfs(int x , int fa,vector<int> &nodes,vector<vector<int>> &near\_grip){

           nodes.push\_back(x);

           for(int y : near\_grip[x]){

               if(recall[y] && y!=fa){

               get\_dfs(y,x,nodes,near\_grip);

               }

           }

         return ;

    }

    long long countPaths(int n, vector<vector<int>>& edges) {

        //写图的题目的时候如果题目没有给完整的邻接矩阵 或者邻接表则自己需要构建

        //写图论的题目需要学会画图思考

        init(n ) ;

        vector<vector<int>> near\_grip(n+1) ; //邻接表

        for(auto &k :edges ){

            near\_grip[k[0]].push\_back(k[1]);

            near\_grip[k[1]].push\_back(k[0]);

        }

         vector<int> set\_size(n+1 , 0);

       long long ans = 0 ;

       for(int x =1 ; x <= n ; x++){

           if(recall[x]){//跳过非质数

             continue;

           }

           //以x 为断点将这棵树分成若干个部分

           //分别求这几个部分非质数的个数

           long long sum = 0 ;

           for(int y : near\_grip[x]){

               nodes.clear();

              if(!recall[y]) continue ;

              if(set\_size[y] == 0){//该联通变量没有被计算过

               get\_dfs(y,-1,nodes,near\_grip);

               for(int k : nodes){

                   set\_size[k] = nodes.size();//防止重复计算

               }

              }

              ans += (long long) set\_size[y] \*sum ;

              sum +=set\_size[y];

           }

           ans +=sum;

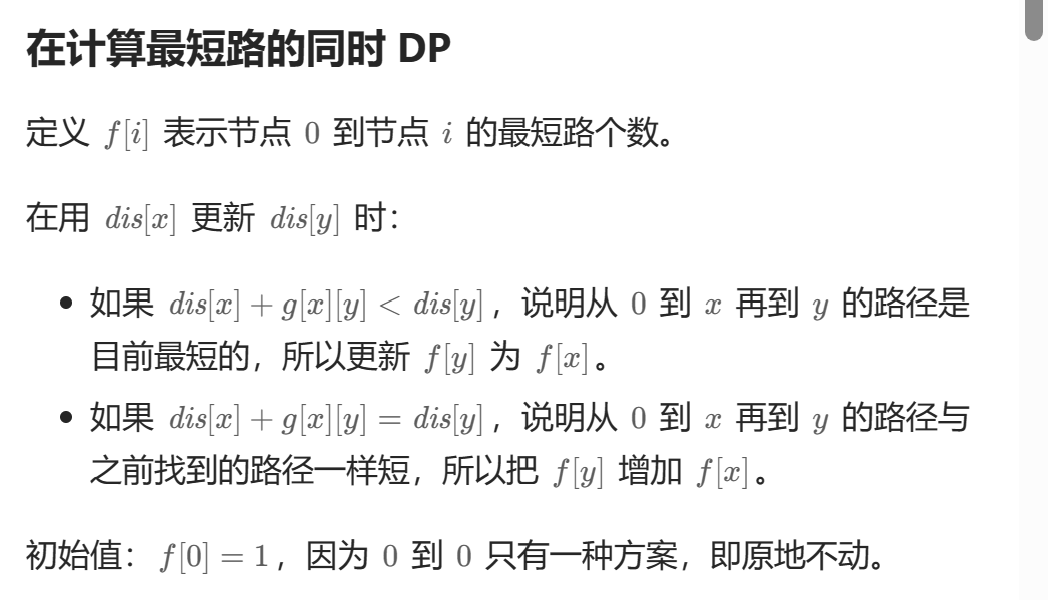
       }

       return ans ;

    }

};





int countPaths(int n, vector<vector<int>>& roads) {

        //dj算法求最短路径

        //得全部用long long 类型存储不然会数据溢出导致正数变成负数

     vector<vector<long long >> near\_grip(n+1,vector<long long >(n+1,LLONG\_MAX/2)) ;

     vector<bool> vis(n+1,false);

     vector<long long> dis(n,LLONG\_MAX/2) ;//用来记录到 i 的最短路径长度

     for(auto v\_e: roads){//创建邻接矩阵

         near\_grip[v\_e[0]][v\_e[1]] = v\_e[2] ;

         near\_grip[v\_e[1]][v\_e[0]] =v\_e[2] ;

     }

     dis[0] = 0;

     vector<int> f(n),done(n);// f[x] 表示以最短路径到达x节点的条数

     f[0] =1;

      while(true){

          int x = - 1;

          for(int i = 0 ; i< n ; i++){//找最短的路线

              if(!vis[i] && (x < 0 || dis[x] >dis[i])){

                  x=i;

              }

          }

          if(x==n-1){

              return f[n-1] ;

          }

          vis[x] = true ;

          for(int y= 0 ; y < n ; y++){

              long long new\_dis = dis[x] + near\_grip[x][y];

              if(new\_dis < dis[y]){

                  dis[y] = new\_dis ;

                  f[y]=f[x] ;

              }

              else if(new\_dis==dis[y]){

                  f[y] = (f[y] + f[x]) % 1000000007;

              }

          }

      }

    }