活动安排

设有 n 个活动的集合 $E = \{1, 2, \ldots, n\}$，其中每个活动都要求使用同一资源，如演讲会场等，而在同一时间内只有一个活动能使用这一资源。

每个活动 i 都有一个要求使用该资源的起始时间 s_i 和一个结束时间 f_i，且 $s_i < f_i$。

如果选择了活动 i，则它在时间区间 $[s_i, f_i)$ 内占用资源。

若区间 $[s_i, f_i)$ 与区间 $[s_j, f_j)$ 不相交，则称活动 i 与活动 j 是相容的。

也就是说，当 $f_i \leq s_j$ 或 $f_j \leq s_i$ 时，活动 i 与活动 j 相容。

选择出由互相兼容的活动组成的最大集合。

输入格式

第一行一个整数 n；

接下来的 n 行，每行两个整数 s_i 和 f_i。

输出格式

输出互相兼容的最大活动个数。

思路是贪心。为了尽可能多的参与活动，我们直接将所有活动按照结束时间的早晚升序排序。然后设置当前时间节点，从第一个活动开始向后遍历，当时间节点允许我们参与这次活动时，则参加，并将时间节点更新至这次活动的结束时间；反之不参加这一次，等下一个我们可以参加的活动。

算法复杂度主要来自排序。用快排之后算法的复杂度为：
$O(nlogn + n) = O(nlogn)$
空间复杂度$O(n)$

一份代码：

```
# include<bits/stdc++.h>
using namespace std;
typedef long long ll;
const int N=1e7;
int s[N];
int f[N];
int n;
vector<pair<int,int>>q;
bool cmp(pair<int,int>a,pair<int,int>b)
{
        if(a.second!=b.second)
        return a.second<b.second;
```

```
        else if(a.second==b.second)
        {
                return a.first>=b.first;
        }
}
int main (void)
{
        cin>>n;
        for(int i=0;i<n;i++)
        {
                int a,b;
                cin>>a>>b;
                q.push_back({a,b});

        }
        sort(q.begin(),q.end(),cmp);

        int start=0;
        int res=0;
        for(int i=0;i<n;i++)
        {
                if(start<=q[i].first)
                {
                        res++;
                        start=q[i].second;
                }

        }
        cout<<res<<endl;

        }
```

最优装载

> 有一批集装箱要装上一艘载重量为c的轮船。其中集装箱i的重量为Wi。最优装载问题要求确定在装载体积不受限制的情况下，将尽可能多的集装箱装上轮船。

与0-1背包不一样，本题不考虑体积，每个集装箱的价值可以认为都等于为1。

我们直接将集装箱按照重量升序排序然后依次装上船即可，当载重超重时退出。

复杂度主要来自于排序，为$O(nlogn)$。

最小生成树

最小生成树的概念就是在一个无向联通图中找出包含所有节点，且所有边权值之和最小的无向联通无环子图。

主要考虑prim和kruskal算法。

#prim算法

prim算法的原理是，构造最小生成树的点集合，当点集合为满时结束过程。每一次选择到点集合距离最短的点进入，并将对应的边加上，生成新的树。每选择一次节点，需要更新一次每个点到集合的距离。

模板：

```
# include<bits/stdc++.h>
# include<algorithm>
typedef long long ll;
int n,m;
int p[5100][5100];
int dis[5100];
bool istrue[5100];
using namespace std;
int prim()
{
        int res=0;
        memset(dis,0x3f,sizeof dis);      //在此注意要将dis距离初始化为无穷大
        for(int i=0;i<n;i++)
        {
                int t=-1;
                for(int j=1;j<=n;j++)
                {
                        if(!istrue[j]&&(t==-1||dis[j]<dis[t]))
                        t=j;
                }
                if(i&&dis[t]==0x3f3f3f3f)     //第一次的时候的距离必然为无限大，所以需要特
判
                return 114514;
                if(i)
                res+=dis[t];
                istrue[t]=true;
                for(int j=1;j<=n;j++)
```

```
                dis[j]=min(dis[j],p[t][j]);

        }
        return res;
}

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

        for(int i=0;i<m;i++)
        {
                int a,b,c;
                cin>>a>>b>>c;
                p[a][b]=p[b][a]=min(p[a][b],c);
        }
        int t=prim();
        if(t==114514)
        {
                cout<<"orz"<<endl;
                return 0;
        }
        cout<<t<<endl;
        return 0;

}
```

设图共有n个节点，那么复杂度为$O(n^2)$

#kruskal算法

在实际中我们用的更多的也是K算法。我们主要关注K算法。

生成一个最小生成树，本质是构造一个连通块，其中有树的所有节点并且权值之和最小。
K算法原理是将所有边按照边权重从小到大排序，如果这条边的两个端点不在同一个连通块中，则把他们合并并且把边加入生成树中。

会用到并查集[并查集]。

859. Kruskal算法求最小生成树 - AcWing题库

```
# include<bits/stdc++.h>
using namespace std;
```

```
typedef long long ll;

int n,m;
struct Edge
{
        int a,b,c;
};
Edge edge[200005];
bool cmp(Edge x,Edge y)
{
        return x.c<y.c;
}
int p[100005];

int find(int x)
{
        if(p[x]!=x)
        p[x]=find(p[x]);
        return p[x];

}
int main (void)
{
        cin>>n>>m;

        for(int i=0;i<m;i++)
        {
                int a,b,c;
                cin>>a>>b>>c;
                edge[i].a=a;
                edge[i].b=b;
                edge[i].c=c;


        }
        for(int i=1;i<=n;i++)
        p[i]=i;
        sort(edge,edge+m,cmp);

        int ans=0;

        int cnt=0;
        for(int i=0;i<m;i++)
        {
```

```
                int a=edge[i].a;
                int b=edge[i].b;

                if(find(a)!=find(b))
                {
                        ans+=edge[i].c;
                        p[find(a)]=find(b);
                        cnt++;
                        if(cnt==n-1)
                        break;
                }
        }

        if(cnt<n-1)
        {
                puts("impossible");
        }
        else
        cout<<ans<<endl;

        return  0;

}
```

设图的边数为e，那么算法的复杂度为$O(eloge)$。

通过对比可以发现，k算法面对稀疏图表现远好于prim。但在稠密图中prim优于K算法。

多机调度

到目前为止这个问题还是无解，考试也不会考察。

单源最短路

dijkstra算法。

注意，优化版的jk算法需用邻接表表示图，模拟链表见单链表

同时jk算法用到邻接表图的遍历方法树与图的深度优先遍历与存储

朴素dijkstra算法的思想是，给定好了起始点和目标点，找出这个点到目标点的最短距离。但事实上在这个过程中起点需要更新一次到所有点的最短距离，所以事实上也可以求单源最短路问题。

先定义一个概念：

- 一个集合（起个名字叫S，而且其中包含若干个点，不妨先称呼他们为b, c, d）到一个点（起个名字叫a）的距离，就是S中的点到这个点距离的最小值(比如说b, c, d到a的距离分别是$1, 2, 3$，那么S到a的距离就是1)

我们从起始点开始，并设置好点集合S（在S中的点表示我们已经求出了起点到这个点的最短距离），将起点纳入集合中，然后找出当前到集合S距离最小的点，并将他纳入集合中。不断重复操作，直到所有点加入集合中。

以下是一种朴素的实现方法。但是只能处理1e5以下的数据。

```
# include<bits/stdc++.h>
# include<algorithm>
typedef long long ll;
using namespace std;
int n,m;
bool istrue[114514];
int p[510][510];
int dis[510];
int dijstra()
{
        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(!istrue[j]&&(t==-1||dis[t]>dis[j]))
                                t=j;
                }
                for(int i=1;i<=n;i++)
                {
                        dis[i]=min(dis[i],dis[t]+p[t][i]);

                }
                istrue[t]=true;

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

```
}

int main (void)
{
        cin.tie(0);
        cout.tie(0);
        cin>>n>>m;
        memset(p,0x3f,sizeof p);
        for(int i=0;i<m;i++)
        {
                int a,b,c;
                cin>>a>>b>>c;
                p[a][b]=min(p[a][b],c);

        }
        int t=dijstra();
        cout<<t<<endl;
        return 0;

}
```

#dijkstra优化

[850. Dijkstra求最短路 II - AcWing题库]()

jk算法中最慢的一步在于找到点集中到集合距离最短的一个点。因此可以用堆来优化查找速度。

原理和朴素版jk算法一样，但是使用邻接表存储稀疏图。

```
# include<bits/stdc++.h>
using namespace std;
int n,m;
const int N=150005;

int e[N],h[N],idx,w[N],ne[N];
typedef pair<int,int>PII;

bool st[N];
int dist[N];

void add(int a,int b,int c)
{
```

```
        e[idx]=b;
        ne[idx]=h[a];
        w[idx]=c;
        h[a]=idx++;

}

int dijkstra()
{

        memset(dist,0x3f,sizeof dist);
        dist[1] = 0;
    priority_queue<PII, vector<PII>, greater<PII>> heap;//将堆设置为小根堆
    heap.push({0, 1});  //将最初的点，编号为1，到起点距离为0 的点压入堆中。

        while(heap.size())
        {
                auto t=heap.top();
                heap.pop();
                int ver=t.second;    //ver是heap堆顶点的编号
                int distance=t.first;//distance是顶点到起点的距离

                if(st[ver])
                continue;            //如果该点已经入集合，直接跳过
                st[ver]=true;
                for(int i=h[ver];i!=-1;i=ne[i])
                {
                        int j=e[i];

                        if(dist[j]>distance+w[i])
                        {
                                dist[j]=distance+w[i];

                                heap.push({dist[j],j});

                        }
                }
        }

        if(dist[n]==0x3f3f3f3f)
        return -1;
        else
        return dist[n];
```

```
}
int main (void)
{
        cin>>n>>m;
        memset(h,-1,sizeof h);

        while(m--)
        {
                int a,b,c;
                cin>>a>>b>>c;
                add(a,b,c);


        }
        int t=dijkstra();
        cout<<t<<endl;
        return 0;

}
```