# 一、问题分析

- ## 分析并确定要处理的对象（数据）是什么

  n 个路口，m 条道路，分为小道和大道，每条道路的长度 c

- ## 分析并确定要实现的功能是什么

  从键盘上输入路口及道路信息，存储每条道路的信息
  计算走每条道路需要的疲劳度
  找到路口 1 与路口 m 之间所需要的最小疲劳度
  输出该最小的疲劳度

- ## 分析并确定处理后的结果如何显示

  直接在屏幕上输出疲劳度

# 二、数据结构和算法设计

- ## 抽象数据类型设计

**选用何种 ADT：**

该题每个路口有多条路每条路指向其它路口，符合图的数据关系。每条路径有长度，且长度对应了不同的疲劳度，因此有权，对于每条路从一个路口走向另一路口的疲劳度等于从另一路口走向当前路口的疲劳度，因此是无向的。因此选用无向有权图。

- ## 物理数据对象设计

**物理数据结构的选择**

基于邻接矩阵的图实现起来较快，且该问题数据量较小，用邻接矩阵也可快速实现，因此选用基于邻接矩阵的无相有权图。

**物理数据结构的设计**

```
template<typename T>
class map_matirx:public Graph
{
public:
    map_matirx(int num);
    ~map_matirx();
    int n();                    //返回点数
    int e();                    //返回边数
    int first(int v);           //返回该点第一个邻点的下标
    int next(int v, int w);     //返回该点下一个邻点的下标
    bool setPoint(int v, T val);
    bool setEdge(int v1, int v2, int wght);
    int weight(int v1, int v2);
    bool setMark(int v, bool val);
```

```
        bool getMark(int v);
        T getPoint(int v);
    private:
        int numPoint, numEdge;
        T* point;
        bool* mark;
        int** matirx;
};
```

- **算法思想的设计**

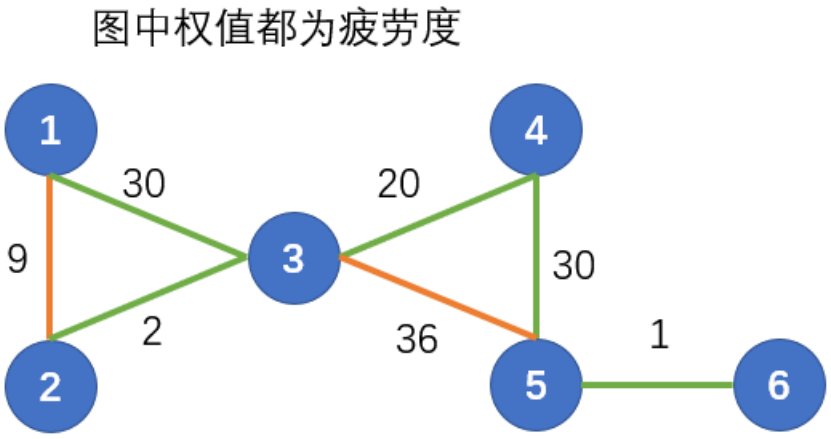
**输入模块：**

输入点数和边数，建立相应的大小的邻接矩阵，将边权赋值为两点间的疲劳度。

**计算模块：**

使用 Dijkstra 算法，求出点 1 与点 N 间的最小疲劳度，返回该疲劳度

- **详细给出样例求解过程**




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 初始化 | 0 | ∞ | ∞ | ∞ | ∞ | ∞ |
| 第一次 | 0 | 9 | 30 | ∞ | ∞ | ∞ |
| 第二次 | 0 | 9 | 11 | ∞ | ∞ | ∞ |
| 第三次 | 0 | 9 | 11 | 33 | 47 | ∞ |
| 第四次 | 0 | 9 | 11 | 33 | 47 | ∞ |
| 第五次 | 0 | 9 | 11 | 33 | 47 | 48 |

由此过程得出，从点 1 到点 6 的最小疲劳度为 48。(已确定最小值的点为黄色)

- **关键功能的算法步骤**

**输入模块**

1、给每个点赋上编号

2、将每条路的长度转换为疲劳度

3、将给每条路赋权值为疲劳度

```
    void creat(map_matirx<int>& G, int n, int e)
```

```
    {
        for (int i = 0; i < n; i++)
        {
            G.setPoint(i, i);                //给每个点赋编号
            G.setMark(i, false);             //将每个点标记为未访问
        }
        for (int i = 0; i < e; i++)
        {
            bool f;
            int s1, s2，w;
            cin >> f >> s1 >> s2 >> w;  //输入每条路的信息
            if (f)
                w = w * w;                   //将长度转化未疲劳度
            G.setEdge(s1-1, s2-1, w);  //将疲劳度赋值给每条路
            G.setEdge(s1-1, s2-1, w);
        }
}
```

**计算模块**

1、创建一个整型数组存储到该路口的最小疲劳度，创建一个布尔型数组，存储每个点是否求出到它的最小疲劳度

2、找出当前未求出最小疲劳度的点中疲劳度最小的那个点

3、该点当前的疲劳度即为到该点最小疲劳度，将该点设为已找出最小疲劳度

4、以该点为起点，遍历与它相连的点，判断从该点到那个点的疲劳度加上该点目前的疲劳度是否小于到那个点的已知的疲劳度，若小于，则对到另一点最小疲劳度赋值为从该点到那个点的疲劳度加上该点目前的疲劳度

5、重复 2-4 步，直至 N 号点找到了最小疲劳度

```
Int Dijkstra(map_matirx<int>& G)
{
        //初始化存储疲劳度与是否求出最小疲劳度的点的数组
        int* len = new int[n];
        bool* flag = new bool[n];
        for (int i = 0; i < n; i++)
        {
            len[i] =  ∞;
            flag[i] = 0;
        }

        //将点 1 设为已求出最小疲劳度，且该疲劳度为 0
        len[0] = 0;
        flag[0] = 1;

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

```
            //找出目前未找出最小疲劳度的点的下标
            int v = cmin(len,flag, n);

            //如果有点无法到达或已找到目标点的最小疲劳度则退出
            if (len[v] == 100000 || v == n - 1)
                break;

            flag[v] = 1;          //将这个最小的点标记为已找到最小疲劳度

        //以该点为路径上经过的点，若从该点到其邻点的疲劳度更小，则对其邻点的疲劳度重新赋值
            for (int j = G.first(v); j < G.n(); j = G.next(v, j))
            {
                len[j] = min(len[j], len[v] + G.weight(v, j));
            }
        }
    Return   len[n - 1];
}
```

# 三、算法性能分析

**对于输入模块：**存储了每个点和边，因此时间复杂度为 O(N+E);
**对于计算模块：**Djikstra 算法的时间复杂度为 O(N^2),对数组初始化时间复杂度为 O(N),因此时间复杂度为 O(N^2);
**总的时间复杂度**：O(N^2+E)