一、问题要求

【题目大意】
在n个人中，某些人的银行账号之间可以互相转账。这些人之间转账的手续费各不相同。给定这些人之间转账时需要从转账金额里扣除百分之几的手续费，请问A最少需要多少钱使得转账后B收到100元。
【输入格式】
第一行输入两个正整数n,m，分别表示总人数和可以互相转账的人的对数。
以下m行每行输入三个正整数x,y,z，表示标号为x的人和标号为y的人之间互相转账需要扣除z%的手续费(z<100)。
最后一行输入两个正整数A,B。数据保证A与B之间可以直接或间接地转账。
【输出格式】
注意： 2<=N<=20，1<=M<=20，1<=Q<=100
输出A使得B到账100元最少需要的总费用。精确到小数点后8位。

二、问题分析

- 处理的数据对象： double 型数据来存储图的数据
- 实现的功能： 利用图结构来存储数据，然后利用 Dijkstra 算法来求最小值， 然后输出使得B收到100元A要出多少钱
- 处理后结果显示：显示由A到B需要至少需要多少钱就可

三、数据结构和算法设计

抽象数据类型的设计

```
/*
 *图边的定义
 */
class Edge
{
    int vert, wt;
public:
    Edge()
    {
        vert = -1;
        wt = -1;
    }
    Edge(int v, int w)
    {
        vert = v;
        wt = w;
    }
    int vertex()
    {
        return vert;
    }
    int weight()
    {
        return wt;
    }
};
```

```
/*
 *用邻接表实现图
 *Link 的 ADT
 *
 */
template <typename E> class Link {
public:
  E element;       // 结点值
  Link *next;         // 结点指针：在链表中指向下一结点
  // 构造函数
  Link(const E& elemval, Link* nextval =NULL)
    { element = elemval;  next = nextval; }
  Link(Link* nextval =NULL) { next = nextval; }
};
```

```
/*
 *链表的ADT
 */
template <typename E> class List { // List ADT
private:
  void operator =(const List&) {}      // Protect assignment
  List(const List&) {}           // Protect copy constructor
public:
  List() {}           // 默认构造函数
  virtual ~List() {} // 基本的析构函数

  // 从列表中清除内容,让它空着
  virtual void clear() = 0;

  // 在当前位置插入一个元素
  // item: 要插入的元素
  virtual void insert(const E& item) = 0;

  // 在列表的最后添加一个元素
  // item: 要添加的元素
  virtual void append(const E& item) = 0;

  // 删除和返回当前元素
  // Return: 要删除的元素
  virtual E remove() = 0;

  // 将当前位置设置为列表的开始
  virtual void moveToStart() = 0;

  // 将当前位置设置为列表的末尾
  virtual void moveToEnd() = 0;

  // 将当前位置左移一步，如果当前位置在首位就不变
  virtual void prev() = 0;

  // 将当前位置右移一步，如果当前位置在末尾就不变
  virtual void next() = 0;

  // 返回列表当前元素个数
  virtual int length() const = 0;

```

```
  // 返回当前位置
  virtual int currPos() const = 0;

  // 设置当前位置
  // pos: 要设置的当前位置
  virtual void moveToPos(int pos) = 0;

  // Return: 当前位置的元素
  virtual const E& getValue() const = 0;
};
```

```
/*
 *图的ADT 用来存储数据
 *
 */
class Graph {
private:
  void operator =(const Graph&) {}
  Graph(const Graph&) {}

public:
  Graph() {}
  virtual ~Graph() {}

  virtual void Init(int n) =0;


  virtual int n() =0;
  virtual int e() =0;


  virtual int first(int v) =0;


  virtual int next(int v, int w) =0;


  virtual void setEdge(int v1, int v2, int wght) =0;


  virtual void delEdge(int v1, int v2) =0;

  virtual bool isEdge(int i, int j) =0;


  virtual int weight(int v1, int v2) =0;


  virtual int getMark(int v) =0;
  virtual void setMark(int v, int val) =0;

  virtual int getInDegree(int v) = 0;
  virtual int getOutDegree(int v) = 0;
};
```

物理数据对象的设计

```
/*
 *链表的实现
 */
template <typename E> class LList: public List<E> {
private:
  Link<E>* head;        // 指向链表头结点
  Link<E>* tail;        // 指向链表最后一个结点
  Link<E>* curr;        // 指向当前元素
  int cnt;              // 当前列表大小

  void init() {         // 初始化
    curr = tail = head = new Link<E>;
    cnt = 0;
  }

  void removeall() {   // Return link nodes to free store
    while(head != NULL) {
      curr = head;
      head = head->next;
      delete curr;
    }
  }

public:
  LList(int size=100) { init(); }    // 构造函数
  ~LList() { removeall(); }                    // 析构函数
  void print() const;                // 打印列表内容
  void clear() { removeall(); init(); }        // 清空列表

  // 在当前位置插入“it”
  void insert(const E& it) {
    curr->next = new Link<E>(it, curr->next);
    if (tail == curr) tail = curr->next;  //新的尾指针
    cnt++;
  }

  void append(const E& it) { // 追加“it”到列表中
    tail = tail->next = new Link<E>(it, NULL);
    cnt++;
  }

  // 删除并返回当前元素
  E remove() {
    assert(curr->next != NULL);//"No element"
    E it = curr->next->element;       // 保存元素值
    Link<E>* ltemp = curr->next;      // 保存指针域信息
    if (tail == curr->next) tail = curr; // 重置尾指针
    curr->next = curr->next->next;    // 从列表中删除
    delete ltemp;                     //回收空间
    cnt--;                // 当前列表大小减一
    return it;
  }

  void moveToStart() // 将curr设置在列表头部
    { curr = head; }

  void moveToEnd()    // 将curr设置在列表尾部
```

```
    { curr = tail; }

  // 将curr指针往前（左）移一步；如果已经指向头部了就不需要改变
  void prev() {
    if (curr == head) return;        // 之前没有元素
    Link<E>* temp = head;
    // March down list until we find the previous element
    while (temp->next!=curr) temp=temp->next;
    curr = temp;
  }

  // 将curr指针往后（右）移一步；如果已经指向尾部了就不需要改变
  void next()
    { if (curr != tail) curr = curr->next; }

  int length() const  { return cnt; } // 返回当前列表大小

  // 返回当前元素的位置
  int currPos() const {
    Link<E>* temp = head;
    int i;
    for (i=0; curr != temp; i++)
      temp = temp->next;
    return i;
  }

  // 向下移动到列表“pos”位置
  void moveToPos(int pos) {
    assert ((pos>=0)&&(pos<=cnt));//"Position out of range"
    curr = head;
    for(int i=0; i<pos; i++) curr = curr->next;
  }

  const E& getValue() const { // 返回当前元素
    assert(curr->next != NULL);//"No value"
    return curr->next->element;
  }
};
```

```
/*
 *图的邻接表实现
 */
class Graphl : public Graph
{
private:
    List<Edge>** vertex;        // List headers
    int numVertex, numEdge;     // Number of vertices, edges
    int *mark;                  // Pointer to mark array
public:
    Graphl(int numVert)
    {
        Init(numVert);
    }

    ~Graphl()          // Destructor
    {
        delete [] mark; // Return dynamically allocated memory
```

```
        for (int i = 0; i < numVertex; i++) delete [] vertex[i];
        delete [] vertex;
    }

    void Init(int n)
    {
        int i;
        numVertex = n;
        numEdge = 0;
        mark = new int[n];  // Initialize mark array
        for (i = 0; i < numVertex; i++) mark[i] = UNVISITED;
        // Create and initialize adjacency lists
        vertex = (List<Edge>**) new List<Edge>*[numVertex];
        for (i = 0; i < numVertex; i++)
            vertex[i] = new LList<Edge>();
    }

    int n()
    {
        return numVertex;    // Number of vertices
    }
    int e()
    {
        return numEdge;    // Number of edges
    }

    int first(int v)   // Return first neighbor of "v"
    {
        if (vertex[v]->length() == 0)
            return numVertex;      // No neighbor
        vertex[v]->moveToStart();
        Edge it = vertex[v]->getValue();
        return it.vertex();
    }

    // Get v's next neighbor after w
    int next(int v, int w)
    {
        Edge it;
        if (isEdge(v, w))
        {
            if ((vertex[v]->currPos() + 1) < vertex[v]->length())
            {
                vertex[v]->next();
                it = vertex[v]->getValue();
                return it.vertex();
            }
        }
        return n(); // No neighbor
    }
    // Set edge (i, j) to "weight"
    void setEdge(int i, int j, int weight)
    {
//   Assert(weight>0, "May not set weight to 0");
        assert(weight > 0);
        Edge currEdge(j, weight);
        if (isEdge(i, j))   // Edge already exists in graph
        {
```

```
            vertex[i]->remove();
            vertex[i]->insert(currEdge);
        }
        else   // Keep neighbors sorted by vertex index
        {
            numEdge++;
            for (vertex[i]->moveToStart();
                    vertex[i]->currPos() < vertex[i]->length();
                    vertex[i]->next())
            {
                Edge temp = vertex[i]->getValue();
                if (temp.vertex() > j) break;
            }
            vertex[i]->insert(currEdge);
        }
    }

    void delEdge(int i, int j)    // Delete edge (i, j)
    {
        if (isEdge(i, j))
        {
            vertex[i]->remove();
            numEdge--;
        }
    }

    bool isEdge(int i, int j)   // Is (i,j) an edge?
    {
        Edge it;
        for (vertex[i]->moveToStart();
                vertex[i]->currPos() < vertex[i]->length();
                vertex[i]->next())              // Check whole list
        {
            Edge temp = vertex[i]->getValue();
            if (temp.vertex() == j) return true;
        }
        return false;
    }

    int weight(int i, int j)   // Return weight of (i, j)
    {
        Edge curr;
        if (isEdge(i, j))
        {
            curr = vertex[i]->getValue();
            return curr.weight();
        }
        else return 0;
    }

    int getMark(int v)
    {
        return mark[v];
    }
    void setMark(int v, int val)
    {
        mark[v] = val;
    }
```

```
135
136     int getInDegree(int v)   // 求顶点v的入度
137     {
138         int result = 0;
139
140         //.............. 在此行以下插入补充代码
141         for(int i = 0 ; i < numVertex ; i++)
142         {
143             for(vertex[i]->moveToStart() ; vertex[i]->currPos() <
    vertex[i]->length();
144                 vertex[i] -> next() )
145             {
146                 Edge tmp = vertex[i]->getValue();
147                 if(tmp.vertex() == v )result++;
148             }
149         }
150
151
152
153         //.............. 在此行以上插入补充代码
154         return result;
155     }
156
157     int getOutDegree(int v)    // 求顶点v的出度
158     {
159         int result=0;
160          //.............. 在此行以下插入补充代码
161         for( vertex[v] -> moveToStart() ; vertex[v] -> currPos() <
    vertex[v] ->length();
162             vertex[v]->next())
163         {
164             result++;
165         }
166
167         //.............. 在此行以上插入补充代码
168         return result;
169     }
170 };
```

算法思想的设计

【关键想法】
此题目可以转换为从A到B的路径省的钱最多的钱是多少，然后可以通过数学的处理将初始值设-100，就可以用Dijkstra算法求其最短的路径，否则求出来就是最大的花费。
【注意点】
此题目一个处理数据巧妙点就是将其乘以1e9，然后取8位在除以1e9，就可以实现不进位的结果。
还有是得出了最短的路径要注意路径是负的，要将其转化为正的。

关键功能的算法步骤

```
1 //Dijkstra 算法
2 //算法相对于书上就改了一个地方
3  void Dijkstra1(long double* D)
4     {
5         for(int i = 0 ; i < G->n() ; i++)
```

```
        {
            int v = 0;
            v = minVertex(D);
            if(D[v] == INFINITY)return ;
            G->setMark(v,VISITED);
            for(int w = G->first(v) ; w < G->n() ; w = G->next(v,w))
            {
            //注意这里巧妙的处理过程，将其改为了乘法，然后乘以权重，求最小的（因为初
        //始值设为负的）
                if(D[w]  > D[v] * ( 100.0 - G->weight(v,w)) * 0.01)
                {
                    D[w] = D[v] * ( 100.0 - G->weight(v,w)) * 0.01;
                }
            }
        }
    }
```

```
//main函数处理数据的过程
option op(g);
    long double* D = new long double[n + 1];
    fill(D, D + n + 1, INFINITY);
    int a,b;
    cin>>a>>b;
    D[a] = -100;
    op.Dijkstra1(D);
    //这里把D[b]转为正的，然后乘以1e9来实现截断
    double result = (10000.0 / - D[b] ) * 1000000000;
    //实现截断数据
    cout << fixed<<setprecision(8)<<result / 1000000000 ;
```

三、算法性能分析

本算法主要时间消耗在Dijkstra算法上

- minVertex 函数 寻找最小值 需要 O（n）
- Dijkstra O（n）
- 时间复杂度为 O（n**2）

四、日志

```
    这道题一开始想的是Floyd算法，后来想太暴力了，复杂度有点高，就改用了Dijkstra算法，其中有一次是处理数据的时候忘记将其转为正的，然后输出错误，改过来就正确了。
```