2012 浙江大学计算机考研机试题解

我要解决什么问题？
我要如何去解决？
编码！
特别注意：为了排序方便使用了 C++函数库 algorithm 中的 sort 函数。

第一题：Hello World for U

Given any string of N (>=5) characters, you are asked to form the characters into the shape of U. For example, "helloworld" can be printed as:

```
h  d
e  l
l  r
lowo
```

That is, the characters must be printed in the original order, starting top-down from the left vertical line with $n_1$characters, then left to right along the bottom line with n_2 characters, and finally bottom-up along the vertical line with n_3 characters. And more, we would like U to be as squared as possible -- that is, it must be satisfied that $n_1 = n_3 = \max \{ k| k <= n_2$ for all $3 <= n_2 <= N \}$ with $n_1 + n_2 + n_3 - 2 = N$.

Input Specification:

Each input file contains one test case. Each case contains one string with no less than 5 and no more than 80 characters in a line. The string contains no white space.

Output Specification:

For each test case, print the input string in the shape of U as specified in the description.

Sample Input:

```
helloworld!
```

Sample Output:

```
h   !
e   d
l   l
lowor
```

解题思路：

这一题需要解决的问题是将一个字符串写成 U 字形。拿到这一题的第一映像是 U 字的写法（可没有茴香豆的“茴”写法多），先是写第一排第一个字符，然后写第二排第一个字符……然后是最后一排，然后是倒数第二排……但在 C 语言中如果我们要这样写 U 字形的字符串就需要在数组中操作了。如果是直接输出的话，那只能自上至下一行一行输出。首先是第一行，写出第一个字符和最后一个字符，第二行写出第二个字符和倒数第二个字符……

最后是最后一行。需要注意的是除了最后一行输出所有字符，前面每一行只输出两个字符。中间还有空格来隔开每行的两个字符（具体有多少空格，待会计算）。

思路有了，看看具体的要求。字符串的长度是 N，n1，n3 代表两边每列字符的数目。n2 代表最后一行的字符数。题目中给了一个算式：

$$n_1 = n_3 = \max \{ k | k <= n_2 \text{ for all } 3 <= n_2 <= N \} \text{ with } n_1 + n_2 + n_3 - 2 = N.$$

仔细研究这个算式，这里的 k 是不大于 n2 的，也就是说 n1 和 n3 是不大于 n2 且满足 n1+n2+n3=N+2 的最大值。那么自然有 n1=n3=(N+2)/3，n2=N+2-(n1+n3)。也就是说设 side 为两边的字符数（包括最后一行的两端），则 side=n1=n3=(N+2)/3。设 mid 为最后一行除去两端的两个字符后剩下的字符数，mid=N-side*2（总长度减去两边的字符数）。同时 mid 也是我们输出除最后一行外前面所有行需要空出的空格数。

最后如何在第一行输出第一个字符和最后一个字符呢？那自然是 str[0]和 str[len-1-i]（len 为字符串的长度，也就是 N）。

于是问题完美解决，步骤如下：

1）计算字符串长度 len；

2）计算两边的字符数 side=(len+2)/3；

3）计算最后一行中间的字符数（前面每行中间的空格数）；

4）输出每行相应的字符。

由于该题目不难，也没有什么需要特别注意的，我也就不写注意点了。具体细节详见参考代码。

AC 代码：

```
#include <stdio.h>
#include <string.h>

int main(){
    char str[100];          // 定义存储字符串的变量
    int len, mid, side;     // 定义存储字符串长度、最后一排的中间的字符数目以及两边的字符数目
    int i, j;               // 迭代变量

    gets(str);              // 读取字符串
    len = strlen(str);      // 获取字符长度
    side = (len+2) / 3;         // 计算两端的字符数目
    mid = len - side * 2;       // 计算最后一排中间的字符数

    for(i=0; i<side-1; i++){
        // 这个循环是打印出除最后一行的前面所有行
        putchar(str[i]);    // 打印每一行的第一个字符
        for(j=0; j<mid; j++){
            // 这个循环是打印每行第一个字符与最后一个字符之间的空格
            putchar(' ');
        }
        putchar(str[len-1-i]);  // 打印每一行的最后一个字符
```

```
        putchar('\n');              // 记得换行
    }

    for(i=0; i<mid+2; i++){
        // 这个循环是用来打印最后一行
        putchar(str[side-1+i]);
    }

    return 0;
}
```

第二题：Sharing

To store English words, one method is to use linked lists and store a word letter by letter. To save some space, we may let the words share the same sublist if they share the same suffix. For example, "loading" and "being" are stored as showed in Figure 1.

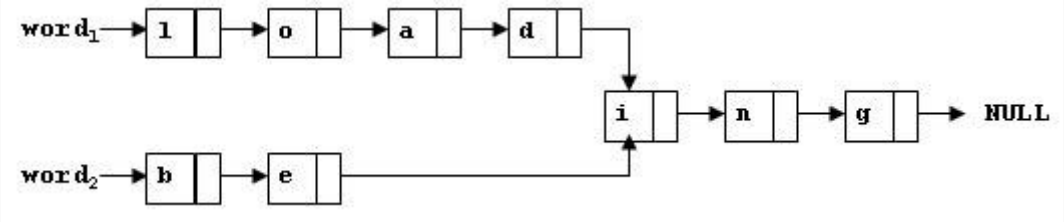

Figure 1

You are supposed to find the starting position of the common suffix (e.g. the position of "i" in Figure 1).

Input Specification:

Each input file contains one test case. For each case, the first line contains two addresses of nodes and a positive N (<= 10^5), where the two addresses are the addresses of the first nodes of the two words, and N is the total number of nodes. The address of a node is a 5-digit positive integer, and NULL is represented by -1.

Then N lines follow, each describes a node in the format:

Address Data Next

where *Address* is the position of the node, *Data* is the letter contained by this node which is an English letter chosen from {a-z, A-Z}, and*Next* is the position of the next node.

Output Specification:

For each case, simply output the 5-digit starting position of the common suffix. If the two words have no common suffix, output "-1" instead.

Sample Input 1:

```
11111 22222 9
67890 i 00002
00010 a 12345
00003 g -1
12345 D 67890
```

```
00002 n 00003
22222 B 23456
11111 L 00001
23456 e 67890
00001 o 00010
```

Sample Output 1:

```
67890
```

Sample Input 2:

```
00001 00002 4
00001 a 10001
10001 s -1
00002 a 10002
10002 t -1
```

Sample Output 2:

```
-1
```

解题思路：

这一题需要解决的问题是求出两个列表共享空间时的起始地址。

要解决这个问题，可以先看看“Y”是怎么写的。“Y”的形状正如同两个列表汇聚在一起。那么我们就可以这样解决这个问题。先将第一个列表的路径标志出来。（“Y”从左上角开始一直到底）然后从右上角遍历第二个列表的路径。当在遍历第二个列表的路径时如果遇到第一个列表的标记，意味着它们在这里汇合。当前标志的地址自然是它们首次汇合的地址。既然思路有了，那数据结构如何设计呢？题目中说明地址是具有 5 个数字的正整数。那么我们就可以使用数组下标表示结点当前地址，结点中包含是否属于第一个列表的标记，以及下一个结点的地址。当然，你也可以使用一个数组表示所有结点，但结点的元素包含的是当前地址、一个是否属于第一个列表的标记和下一个结点的地址这三个内容。

于是结点类型定义为：

```
struct NodeType{
    int flag, next;
};
```

解题步骤如下：

1）定义数据结构；

2）读取结点信息；

3）标记第一个列表的路径；

4）遍历第二个列表，一旦遇到第一个节点元素则停止遍历；

5）输出相应的结果。

有一些情况需要注意：

两个列表可能都为空，也可能一个为空。可能一开始就在汇聚在一起也可能根本就不汇聚在一起。

AC 代码：

```
#include <stdio.h>
```

```
// 定义存储元素的结构体，
// 其中 flag 是用来标记该元素是不是第一个列表中的结点
// next 为下一个节点地址
struct NodeType{
    int flag, next;
};

int main(){

    NodeType nodes[110000] = {0};  // 定义存储所有结点的变量
    int a1, a2, a, next;           // 分别定义存储第一、第二列表的首地址，结点地址和后一个结点地址的变量
    int num;                       // 定义存储节点总数的变量
    int i;                         // 迭代变量
    scanf("%d%d%d",&a1, &a2, &num);
    while(num--){                  // 读取所有结点必要的信息
        scanf("%d%*s%d", &a, &next);// 这里内容字符是没有起到实际作用的，所以就读取不存储了
        nodes[a].next = next;
    }

    // 遍历第一个列表的所有结点并标记
    i = a1;
    while(i!=-1){
        nodes[i].flag = true;
        i = nodes[i].next;
    }

    // 遍历第二个列表的结点，如果某个结点同时也是第一个列表的结点，说明它们开始公用结点了
    i = a2;
    while(i!=-1 && !nodes[i].flag){
        i = nodes[i].next;
    }

    // 输出结果
    if(i == -1){
        puts("-1");
    }else{
        printf("%05d\n", i);
    }

    return 0;
}
```

第三题： To Fill or Not to Fill

With highways available, driving a car from Hangzhou to any other city is easy. But since the tank capacity of a car is limited, we have to find gas stations on the way from time to time. Different gas station may give different price. You are asked to carefully design the cheapest route to go.

Input Specification:

Each input file contains one test case. For each case, the first line contains 4 positive numbers: C_{max} (<= 100), the maximum capacity of the tank; D (<=30000), the distance between Hangzhou and the destination city; D_{avg} (<=20), the average distance per unit gas that the car can run; and N (<= 500), the total number of gas stations. Then N lines follow, each contains a pair of non-negative numbers: P_i, the unit gas price, and D_i (<=D), the distance between this station and Hangzhou, for i=1,...N. All the numbers in a line are separated by a space.

Output Specification:

For each test case, print the cheapest price in a line, accurate up to 2 decimal places. It is assumed that the tank is empty at the beginning. If it is impossible to reach the destination, print "The maximum travel distance = X" where X is the maximum possible distance the car can run, accurate up to 2 decimal places.

Sample Input 1:

```
50 1300 12 8
6.00 1250
7.00 600
7.00 150
7.10 0
7.20 200
7.50 400
7.30 1000
6.85 300
```

Sample Output 1:

```
749.17
```

Sample Input 2:

```
50 1300 12 2
7.10 0
7.00 600
```

Sample Output 2:

```
The maximum travel distance = 1200.00
```

解题思路：

该题目所要解决的问题是：给定若干加油站信息，问能否驾驶汽车行驶一定的距离。如果能够行驶完全程，则计算最小花费。若不能行驶完全程，则最远能够行驶多长距离。

拿到这一题，首先判断汽车是否能够行驶到终点。什么情况下汽车无法行驶到终点呢？

两种情况：起点根本就没有加油站，汽车无法启动；或者中途两个加油站之间的距离大于加满油后汽车能够行驶的最大距离。前者汽车行驶的最大距离为 0.00，而后者最大距离为当前加油站的距离加上在这个加油站加满油后能够行驶的最大距离。在这里，需要将加油站按到杭州的距离从小到大排序。

接下来在能够行驶到终点的情况下计算最小花费。我们首先从路程来考虑，如果在路上，我们能够使用最便宜的汽油，当然就在那个加油站加油了。所以从起点开始遍历每个加油站。假设遍历到了第 i 个加油站，我们现在来判断在加油站 i 应该加多少油。设当前汽车离杭州的距离为 curLen，当前加油站离杭州的距离为 nodes[i].dis，加满油以后汽车能够行驶的最大距离为（dis=cmax*len）。这样就有 node[i].dis <= curLen <= nodes[i].dis+dis，否则的话第 i 个加油站的油是不起作用的。于是在第 i 个加油站的作用范围内寻找有没有更为便宜的加油站，如果有，则下次使用这个加油站的油（j），这次汽车应该行驶到这个加油站，即 touch=nodes[j].dis。如果没有找到更为便宜的加油站则可以在第 i 个加油站加满油，即 touch=nodes[i].dis+dis。然后判断下次应该行驶到的距离与当前距离的关系，如果下次应当行驶到的距离大于当前距离，则汽车行驶，否则不动（也就是说上个加油站的油更便宜，后一个加油站也便宜，根本就不需要在当前加油站加油）。

解题时注意全程、邮箱容量等是数字并没说是整数。具体请参阅参考代码。

AC 代码：

```
#include <stdio.h>
#include <algorithm>          // 需要用到其中的排序算法
using namespace std;

struct NodeType{
    double price;  // 油价
    double dis;          // 该加油站到杭州的距离
};   // 定义加油站的结点类型

bool compare(NodeType a, NodeType b){
    return a.dis < b.dis;
}    // 定义按到杭州的距离从小到大排序的比较函数

/*
* 求最大能够行驶多少距离
* 如果能够到达目的地，则返回值就是全程的距离
* nodes: 所有的加油站结点
* n: 加油站数目
* terminal: 终点离杭州的距离
* dis: 加满油以后能够行驶的最大距离
* 算法如下：
* 如果起点没有加油站，则最大行驶距离为 0
* 如果某两个加油站之间的距离大于加满油后可以行驶的最大距离或者最后一个加油站距离终点大于加满油后汽车行驶的最大距离，
* 则汽车是不能够行驶到终点的，此时行驶的最大距离就是该加油站加满油以后行驶的最远
```

```
距离
* 汽车能够行驶到终点。
*/
double MaxDis(NodeType *nodes, int n, double terminal, double dis){
    int i = 0;
    if(nodes[0].dis != 0){    // 判断第一个加油站是否在起点
        return 0.0;
    }
    for(i=1; i<n ;i++){       // 遍历加油站，看能否在该加油站加满油后行驶到下一个加油站
        if(nodes[i-1].dis + dis < nodes[i].dis){
            return nodes[i-1].dis+dis;
        }
    }
    if(nodes[n-1].dis + dis < terminal){      // 判断在最后一个加油站加满油后能否行驶到终点
        return nodes[n-1].dis + dis;
    }
    return terminal;
}


/*
* 求全程的最小花费。本身油价是按升算的，这里按距离计算油价了。所以最后调用结果时除以每升油行驶的距离。
* nodes: 所有的加油站结点
* n: 加油站数目
* terminal: 终点离杭州的距离
* dis: 加满油以后能够行驶的最大距离
* 算法如下：
* 遍历所有的加油站，判断在该加油站起作用的范围内是否需要在该加油站加油，如果需要的话应该加多少。
*/
double MinPrice(NodeType *nodes, int n, double terminal, double dis){
    int i, j;
    double touch;       // 汽车应该使用当前加油站的油行驶到的位置
    double curLen = 0; // 汽车行驶的当前距离
    double total = 0;    // 总共的花费
    for(i=0; i<n; i++){  // 遍历每一个加油站

        // 从这个加油站开始，遍历后面的加油站，判断在该加油站的作用范围内有没有加油站的油价会更低。
        // 如果有的话，此时就不需要使用这个加油站的油了。
        for(j=i+1; j<n && nodes[i].dis+dis>nodes[j].dis; j++){
```

```
            if(nodes[j].price < nodes[i].price){
                break;
            }
        }

        // 如果在当前加油站的作用范围内没有找到油价更低的加油站，则汽车应该行驶到距离该加油站一个 dis 距离
        // 或者行驶到终点
        if(j>=n || nodes[i].dis+dis<nodes[j].dis){
            if(nodes[i].dis + dis >= terminal){
                touch = terminal;
            }else{
                touch = nodes[i].dis + dis;
            }
        }else{    // 如果在当前加油站的作用范围内找到了油价更低的加油站，则应当行驶到油价更低的加油站。
                // 这样接下来就可以使用油价更低的油了。
            touch = nodes[j].dis;
        }

        if(touch > curLen){// 如果接下来要行驶的位置大于当前距离，则汽车行驶。否则不行驶。
            total += (touch-curLen)*nodes[i].price;
            curLen = touch;
        }
    }

    return total;
}

int main(){

    NodeType nodes[550];            // 定义存储加油站信息的数组
    double cmax, terminal, len;     // 分别是存储油箱容积、终点以及单位汽油可以行驶的最大距离
    double maxDis;                  // 定义存储汽车能够行驶的最大距离
    int n;
    int i;
    scanf("%lf%lf%lf%d", &cmax, &terminal, &len, &n);
    for(i=0; i<n; i++){
        scanf("%lf%lf", &nodes[i].price, &nodes[i].dis);
    }
    sort(nodes, nodes+n, compare);  // 按到杭州的距离进行从小到大排序，以便后面计算汽车能够行使的最大距离。
```

```
    maxDis = MaxDis(nodes, n, terminal, cmax*len);// 计算汽车能够行驶的最大距离
    if(maxDis < terminal){  // 如果汽车行驶的最大距离小于终点，则输出结果
        printf("The maximum travel distance = %.2lf\n", maxDis);
    }else{    // 如果能够行驶完全程，则结算最小花费
        printf("%.2lf\n", MinPrice(nodes, n, terminal, cmax*len)/len);
    }

    return 0;
}
```

第四题：Head of a Gang

One way that the police finds the head of a gang is to check people's phone calls. If there is a phone call between A and B, we say that A and B is related. The weight of a relation is defined to be the total time length of all the phone calls made between the two persons. A "Gang" is a cluster of more than 2 persons who are related to each other with total relation weight being greater than a given threthold K. In each gang, the one with maximum total weight is the head. Now given a list of phone calls, you are supposed to find the gangs and the heads.

Input Specification:

Each input file contains one test case. For each case, the first line contains two positive numbers N and K (both less than or equal to 1000), the number of phone calls and the weight threthold, respectively. Then N lines follow, each in the following format:

Name1 Name2 Time

where Name1 and Name2 are the names of people at the two ends of the call, and Time is the length of the call. A name is a string of three capital letters chosen from A-Z. A time length is a positive integer which is no more than 1000 minutes.

Output Specification:

For each test case, first print in a line the total number of gangs. Then for each gang, print in a line the name of the head and the total number of the members. It is guaranteed that the head is unique for each gang. The output must be sorted according to the alphabetical order of the names of the heads.

Sample Input 1:

```
8 59
AAA BBB 10
BBB AAA 20
AAA CCC 40
DDD EEE 5
EEE DDD 70
FFF GGG 30
GGG HHH 20
HHH FFF 10
```

Sample Output 1:

```
2
AAA 3
GGG 3
```

Sample Input 2:

```
8 70
AAA BBB 10
BBB AAA 20
AAA CCC 40
DDD EEE 5
EEE DDD 70
FFF GGG 30
GGG HHH 20
HHH FFF 10
```

Sample Output 2:

```
0
```

解题思路：

这一题需要解决的问题是给定通话记录，找出各党派。输出首领以及党派里的成员数目。

首先想到的是并查集，但考研数据结构并没有涉及到并查集。那就使用图论里的搜索。对，就 dfs 了。但无论是深搜还是广搜，在一个二维数组里是比较方便的。当然题目中每个人之间的联系是能够构成一个二维数组，但人名是用字符串标志的，而非整数。这样对使用 dfs 造成了一定的难度。于是我们可以将姓名与整数一一对应。首先想到的解决方案是使用 STL 中的 map，但似乎又是不必要的。于是就直接使用一个字符串数组来记录姓名了，这样姓名就与下标一一对应了。

这样能够将姓名与整数一一对应，随后使用 dfs 便极为简单了。具体详见参考代码。

AC 代码：

```
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;

char name[2000][4];          // 储存名字的数组
int values[2000];          // 记录每个人的通话总时间
int visited[2000];         // 记录每个人是否被访问过了
int phone[2000][2000];  // 记录通话联系，谁与谁通过电话
int numOfName;               // 名字数
int numOfHead;               // 首领人数
int totalValue;            // 在遍历某个党派时记录的总通话时间
int maxValueIndex = 0;  // 最大值下标，这个变量将指示谁是首领
struct HeadType{
    char name[4];
```

```
    int num;
};                                  // 首领的数据结构：首领的姓名以及该党派中的人数
HeadType heads[2000]; // 记录首领的数组

/*
* 设置首领，传递进党派的人数。
* 全局变量 maxValueIndex 指示了谁是首领。
*/
void setHead(int num){
    strcpy(heads[numOfHead].name, name[maxValueIndex]);
    heads[numOfHead].num = num;
    numOfHead++;
}


/*
* 由于要求将结果按首领的姓名非递减排序，所以需要写比较函数
*/
bool cmp(HeadType a, HeadType b){
    return strcmp(a.name, b.name) < 0;
}

/*
* 获得某个字符串在姓名数组中的下标，如果没有该姓名则在最后插入之，并返回插入后的下标
*/
int getStrIndex(char *str){
    int i = 1;
    while(i <= numOfName){
        if(!strcmp(str, name[i])){
            break;
        }
        i++;
    }
    if(i > numOfName){
        numOfName++;
    }
    strcpy(name[i], str);
    return i;
}

/*
* 通过某一个人进行深度优先搜索，遍历和这个人有关系的所有人。
* 在这场遍历中，需要记录的是这群有关系的人中总共的通话时间以及人数。
```

```
*/
int dfs(int index){
    visited[index] = 1;                    // 遍历过的人注意标记
    totalValue += values[index];  // 记录总通话时间
    if(values[index] > values[maxValueIndex]){
        maxValueIndex = index;        // 记录通话时间最多的人
    }
    int num = 1;
    int i;
    int next;
    for(i=1; i<=phone[index][0]; i++){       // 对和 index 个人有关系的所有的人进行判断
        next = phone[index][i];
        if(!visited[next]){          // 只对没有标记过的人遍历
            num += dfs(next);
        }
    }
    return num;
}

int main(){
    int n, k;
    char str1[5], str2[5];       // 记录一次通话中的两个人
    int value; // 记录通话时间
    int i;
    int str1Index, str2Index; // 记录通话两人对应的数组下标
    int numOfMember; // 记录总共有多少人，注意通话的人是有重复的
    scanf("%d%d", &n, &k);
    for(i=0; i<n; i++){
        scanf("%s%s%d", str1, str2, &value);

        // 首先获得名字对应的数字
        str1Index = getStrIndex(str1);
        str2Index = getStrIndex(str2);

        // 相应的通话记录上增加通话时间
        values[str1Index] += value;
        values[str2Index] += value;

        // 记录他们之间的联系
        phone[str1Index][0]++;
        phone[str1Index][phone[str1Index][0]] = str2Index;
        phone[str2Index][0]++;
        phone[str2Index][phone[str2Index][0]] = str1Index;
    }
```

```
    // 对所有的人进行遍历
    for(i=1; i<=numOfName; i++){
        if(!visited[i]){ // 如果某个人没有被访问过，则他/她可能是新党派中的一员
            totalValue = 0;
            maxValueIndex = i;
            numOfMember = dfs(i); // 深度遍历后得到有联系的总人数
            if(numOfMember>2 && totalValue>2*k){  // 如果人数大于 2，总通话时间大于
限制的 2 倍（注意通话是双向的）
                setHead(numOfMember);
            }
        }
    }

    // 输出结果，注意要按首领的姓名排序
    if(numOfHead > 0){
        sort(heads, heads+numOfHead, cmp);
        printf("%d\n", numOfHead);
        for(i=0; i<numOfHead; i++){
            printf("%s %d\n", heads[i].name, heads[i].num);
        }
    }else{
        puts("0");
    }
    return 0;
}
```