



燕山大学

《数据结构与算法》实验报告

Data structure and Algorithm Experiment Report

学生所在学院：软件学院

学生所在班级：18 软件六班

学生姓名：乔翱

学 号：201811040809

指导教师：韩卫 窦燕 尤殿龙 郭丁丁

李季辉 郝晓冰 李可 余扬

教 务 处

2019 年 10 月

目录

实验一 线性表应用-学生成绩链表处理.....	4
实验二 队列应用-Deque.....	8
实验三 队列应用-银行排队问题之单队列多窗口应用.....	12
实验四 堆栈应用- 汉诺塔的非递归实现.....	16
实验五 哈弗曼编码.....	21
实验六 判断是否有欧拉回路-哥尼斯堡的“七桥问题”	25
实验七 拓扑排序-Is Topological Order.....	29
实验八 贪心算法-月饼.....	31
实验九 快速排序-找第 k 小的数.....	34
实验十 背包问题-凑零钱.....	37

实验一 线性表应用-学生成绩链表处理

一、 解题思路

分析题目，要求实现两个函数，一个将输入的学生成绩组织成单向链表；另一个将成绩低于某分数线的学生节点从链表删除。函数 createlist 利用 scanf 从输入中获取学生的信息，将其组织成单向链表，并返回表头指针。函数 deletelist 从以 head 为头指针的链表中删除成绩低于 min_score 的学生，并返回链表的头指针。

创建链表时注意用 while 循环获取信息，直到检测到输入值为 0 为止。在循环中，给链表指针申请新的内存空间，并初始化，同时因节点增加，指针不断后移。

删除函数注意先判断链表是否为空，单独判断头结点，创建临时指针保存满足条件的节点地址，进行删除操作。

二、 程序设计

伪代码：

建立链表函数：

```
if (读入的编号为 0)
then 头指针为空，返回头指针
else 头指针等于临时指针
while (读入的编号不为 0)
do 临时指针的 next 指向读入的节点，临时指针指向读入节点 //尾插法插入节点
end
返回 头指针
```

删除函数：

```
if (头指针为空)

then 返回头指针

while (临时指针的 next 不为空)

    do if (临时指针的 next 的 score 小于最小分数)

        then 临时指针的 next 向后移

        else 临时指针向后移;

end

if (头指针的 score 小于最小分数)

then 头指针向后移

返回 头指针
```

三、 程序详解及运行结果

部分核心代码如下：

```
struct stud_node *createlist()
{
    struct stud_node *head, *tail;
    struct stud_node *x, *y;
    x = (struct stud_node *)malloc(sizeof(struct stud_node));
    scanf("%d%s%d", &x->num, x->name, &x->score);
    if (x->num == 0)
    {
        head = NULL;
        return head;
    }
    else
        head = x;
    while (1)
    {
```

```

        y = (struct stud_node*)malloc(sizeof(struct stud_node));
        scanf("%d", &y->num);
        if (y->num == 0)
        {
            x->next = NULL;
            return head;
        }
        scanf("%s%d", y->name, &y->score);
        x->next = y;
        x = y;
    }
}

struct stud_node *deletelist(struct stud_node *head, int min_score) {
    if (head==NULL)
    {
        return head;
    }
    struct stud_node *x;
    x = (struct stud_node *)malloc(sizeof(struct stud_node));
    x = head;
    while (x->next!= NULL)
    {
        if (x->next->score<min_score)
        {
            x->next = x->next->next;
        }
        else
            x = x->next;
    }
    if (head->score<min_score)
    {
        head = head->next;
    }
    return head;
}

```

运行结果:

```
F:\C++\未命名2.exe
1 zhang 78
2 wang 80
3 li 75
4 zhao 85
0
80
2 wang 80
4 zhao 85

-----
Process exited after 8.506 seconds with return value 0
请按任意键继续. . .
```

四、 问题及解决过程

在实现建表函数的时候对于直接第一个输入的是 0 的情况没有考虑全面，没有考虑到可能会出现建立一个空表的情况。

在删除操作时，漏掉了第一个节点不符合条件的情况，直接判断头指针的 next 是否小于最小分数，漏掉了第一个节点，考虑问题出现疏忽。

仔细分析代码，逐句分析，在建表函数中，开始的时候先判断是否第一个输入就是 0；在删除函数最后判断头指针指向的第一个学生的成绩是否符合条件，分析问题要全面。

实验二 队列应用-Deque

一、 解题思路

本题要实现一种叫做“Deque”的数据结构，其实就是实现一个双端队列，就是在队列的头和尾都可以进行插入和删除操作；deque 是由一个带 header 的双链表实现的。front 和 rear 分别指向 deque 的两端。front 总是指向 header。当 front 和 rear 都指向相同的虚拟节点时，deque 是空的。如果操作可以成功完成，则 Push 和 Inject 返回 1;如果操作失败，返回 0。如果 deque 为空，Pop 和 Eject 必须返回 judge 程序定义的错误。

仔细分析问题，本题没有太大难度，就是实现对队列的增加和删除操作，不同的是，在队列的头和尾都可以进行操作，在实现时应该注意特殊的情形，注意双端队列为空以及双端队列满了时的判断条件。还要注意在实现建立双端队列时初始化时，rear 和 front 指向相同的地址。

二、 程序设计

伪代码：

建立双端队列：

- 1、声明一个双端队列；为双端队列以及其 front 和 rear 声明空间；
- 2、Rear 的 next 以及 front 的 next 都为空，以及 front 和 rear 相等；
- 3、返回声明的双端队列；

Push 函数：

if（双端队列为空）

then 双端队列的 front 的 next 指向插入节点，插入节点的 last 指向双端队列的 front，双端队列的 rear 指向插入节点，插入节点的 next 指向空，返回 1；

else 插入节点的 next 指向双端队列的 front 的 next，双端队列的 front 的 next

的 last 指向插入节点，双端队列的 front 的 next 指向插入节点，插入节点的 last 指向双端队列的 front，返回 1

Pop 函数：

```
if (双端队列为空)

then 返回 error

else if (双端队列只有一个元素)

then 双端队列为空 (双端队列的 rear 等于 front) 返回删除的第一个元素

else 删除双端队列的第一个元素，返回删除的元素
```

Inject 函数：

```
if (双端队列为空)

then 插入元素，双端队列只有一个元素

else 尾插法在双端队列的尾部插入一个元素

返回 1;
```

Eject 函数：

```
if (双端队列为空)

then 返回 error

else 删除最后一个元素，返回删除元素;
```

三、 程序详解及运行结果

核心代码：

```
Deque CreateDeque(){
Deque p;
p=(Deque)malloc(sizeof(struct DequeRecord));
p->Front=(PtrToNode)malloc(sizeof(struct Node));
```

```

p->Rear=(PtrToNode)malloc(sizeof(struct Node));
p->Front->Last=NULL;
p->Rear=p->Front;
p->Rear->Next=NULL;
return p;
}
int Push( ElementType X, Deque D ){
    struct Node *x;
    x = (struct Node *)malloc(sizeof(struct Node));
    if(!x)
return 0;
    x->Element=X;
    if(D->Front==D->Rear){
D->Front->Next=x;
x->Last=D->Front;
D->Rear=x;
x->Next=NULL;
return 1;
    }
    x->Next=D->Front->Next;
D->Front->Next->Last=x;
D->Front->Next=x;
x->Last=D->Front;

    return 1;
}
ElementType Pop( Deque D ){
if(D->Front==D->Rear){
return ERROR;
    }

int temp=D->Front->Next->Element;
struct Node* t=D->Front->Next;
if(D->Front->Next==D->Rear){//一个元素的特殊情况
D->Rear=D->Front;
D->Rear->Next=NULL;
free(t);
return temp;
}
D->Front->Next->Next->Last=D->Front;
D->Front->Next=D->Front->Next->Next;
free(t);
return temp;
}

```

```

}

int Inject( ElementType X, Deque D ){

struct Node *x;
    x = (struct Node *)malloc(sizeof(struct Node));
    if(!x)
return 0;
    x->Element=X;
    if(D->Front==D->Rear){//空双端队列
D->Front->Next=x;
x->Last=D->Front;
D->Rear=x;
return 1;
    }
    D->Rear->Next=x;
    x->Last=D->Rear;
    x->Next=NULL;
D->Rear=x;
return 1;
}
ElementType Eject( Deque D ){
if(D->Front==D->Rear){
return ERROR;
    }
int temp=D->Rear->Element;
struct Node* t=D->Rear;
D->Rear=D->Rear->Last;
D->Rear->Next=NULL;
free(t);
return temp;
}

```

四、 问题及解决过程

每一个节点都有 next 和 last 指针，没有考虑全面，没有考虑 last 指针，导致出现了问题，并且考虑问题不全面，没有注意队列为空特殊情况的处理，仔细分析，全面考虑，对于空队列进行特殊处理。

实验三 队列应用-银行排队问题之单队列多窗口应用

一、解题思路

模拟银行窗口排队问题，要求输出前来等待服务的 N 为顾客的平均等待时间、最长等待时间、最后完成时间，并且统计每个窗口服务了多少名顾客。

先通过输入将队列保存在数组中，之后用队列头元素的到达时间跟窗口的完成时间对比，因为题中说优先考虑近的窗口，所以可以遍历窗口。如果队首的到达时间比这个窗口的完成时间大，就不需要等待，更新这个窗口的等待时间，并且这个窗口人数加一，如果这个窗口无法服务，就求出这个窗口的最快完成时间。如果三个窗口都无法满足，就需要等待，并且求出等待的时间并且用下表记录。最后将需要等待的时间和完成的时间都记录下来。最后将题目要求的数据输出就行。

二、程序设计

1、输入顾客的到达时间以及需要服务的时间，并且存到一个队列里面，注意对于服务时间超过六十分钟的进行特殊处理。

2、遍历每个服务窗口。

if (队首元素的到达时间大于窗口的完成时间)

then 更新窗口的等待时间，并且窗口服务人数加一

if (三个窗口都无法满足)

then 当前顾客需要等待，求出等待时间，记录等待时间和完成时间

3、按要求输出，注意保留小数。

三、程序详解及运行结果

核心代码：

```

#include<iostream>
#include <bits/stdc++.h>
using namespace std;
struct node{
int start,haoshi;
};
queue <node>q;
int main(){
int x;
cin>>x;
for(int i=1;i<=x;i++){
node temp;
cin>>temp.start>>temp.haoshi;
if(temp.haoshi>60)
temp.haoshi=60;
q.push(temp);
}
int n;
cin>>n;
int win[11]={0},num[11]={0};
int wait=0,maxn=0,sum=0;
while(!q.empty())
{
bool flag = true;
int minn = 999999999, imin = 0;
for(int i = 0;i < n;i++)
{
if(win[i] <= q.front().start)
{
win[i] = q.front().start + q.front().haoshi;
num[i]++;
flag = false;
q.pop();
break;
}
if(minn > win[i])
{
minn = win[i];
imin = i;
}
}
if(flag)
{

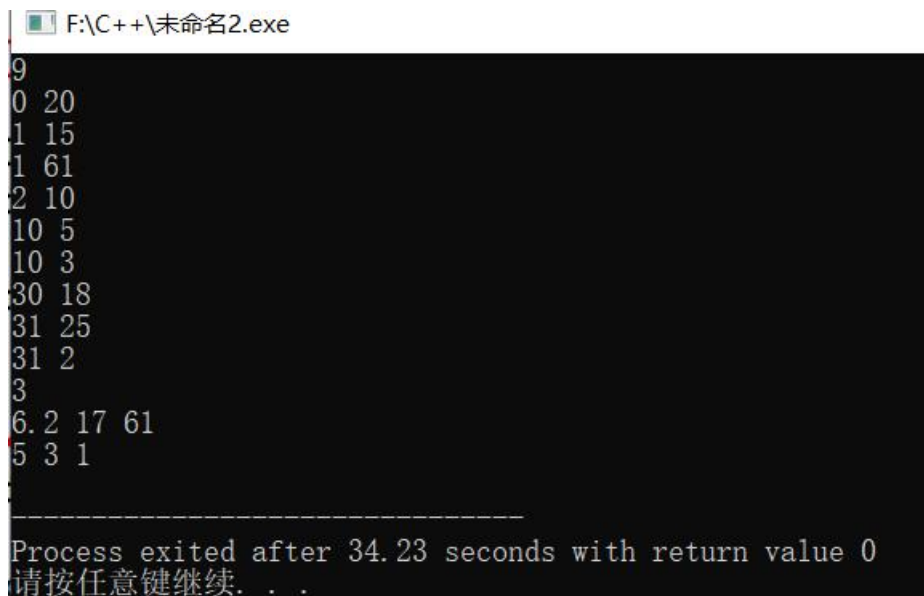
```

```

wait = win[imin] - q.front().start;
if(maxn < wait)
maxn = wait;
win[imin] += q.front().haoshi;
sum += wait;
num[imin]++;
q.pop();
}
}
int last = win[0];
for(int i = 0; i < n; i++)
{
if(last < win[i])
last = win[i];
}
cout<<setprecision(1)<<fixed<<sum*1.0/x<<" "<<maxn<<" "<<last<<endl;
for(int i = 0; i < n; i++)
{
cout<<num[i];
if(i != n-1)
cout<<" ";
}
cout<<endl;
}

```

运行结果:



```

F:\C++\未命名2.exe
9
0 20
1 15
1 61
2 10
10 5
10 3
30 18
31 25
31 2
3
6.2 17 61
5 3 1
-----
Process exited after 34.23 seconds with return value 0
请按任意键继续. . .

```

四、问题及解决过程

没有注意到题目中，“假设每位顾客事务被处理的最长时间为 60 分钟”，没有对超过 60 分钟的顾客进行提前处理，导致几个测试点答案错误。后来在分析题目后，发现了这个错误，对于超过六十分钟的顾客特殊处理，最后成功通过所有测试点。

实验四 堆栈应用-汉诺塔的非递归实现

一、解题思路

看到题目，先用递归做了一次，结果超时了，后来就想这个汉诺塔咋用递归实现，想了很久也没有思路，就画了几组例子发现了如下规律：

当盘子的个数为 n 时，移动的次数应等于 $2^n - 1$ ，只要轮流进行两步操作就可以了。首先把三根柱子按顺序排成品字型，把所有的圆盘按从大到小的顺序放在柱子 A 上，根据圆盘的数量确定柱子的排放顺序：若 n 为偶数，按顺时针方向依次摆放 A B C；

若 n 为奇数，按顺时针方向依次摆放 A C B。

(1)按顺时针方向把圆盘 1 从现在的柱子移动到下一根柱子，即当 n 为偶数时，若圆盘 1 在柱子 A，则把它移动到 B；若圆盘 1 在柱子 B，则把它移动到 C；若圆盘 1 在柱子 C，则把它移动到 A。

(2)接着，把另外两根柱子上可以移动的圆盘移动到新的柱子上。即把非空柱子上的圆盘移动到空柱子上，当两根柱子都非空时，移动较小的圆盘。这一步没有明确规定移动哪个圆盘，你可能以为会有多种可能性，其实不然，可实施的行动是唯一的。

(3)反复进行(1)(2)操作，最后就能按规定完成汉诺塔的移动。

二、程序设计

- 1、读入数据，初始化盘子数量；
- 2、判断 n 是奇数还是偶数，确定柱子摆放顺序
- 3、把圆盘 1 从现在的柱子移动到下一根柱子
- 4、把另外两根柱子上可以移动的圆盘移动到新的柱子上。
- 5、重复 3、4 步骤，并且每次输出移动的操作

三、程序详解及运行结果

```
#include<string>
#include<iostream>
#include<algorithm>
#include<stack>
using namespace std;
int main()
{
    stack < int > a,b,c;
    int i,j,n;
    cin>>n;
    char s[4]={'p','a','b','c'};
    a.push(99999); b.push(999999); c.push(999999);
    for(i=n;i>=1;i--)
    {
        a.push(i);
    }
    if(n%2==1)
    {
        s[2]='c';
        s[3]='b';
    }
    for(i=1;;i++)
    {
        if(i%3==1)
        {
            printf("%c -> %c\n",s[1],s[2]);
            b.push(a.top());
```

```

        a.pop();
        if(b.size()==n+1 || c.size()==n+1)
break;
        if(c.top()>a.top())
        {
            printf("%c -> %c\n",s[1],s[3]);
            c.push(a.top());
            a.pop();
        }
        else
        {
            printf("%c -> %c\n",s[3],s[1]);
            a.push(c.top());
            c.pop();
        }
    }
else if(i%3==2)
{
    printf("%c -> %c\n",s[2],s[3]);
    c.push(b.top());
    b.pop();
    if(b.size()==n+1 || c.size()==n+1)
break;
    if(b.top()>a.top())
    {
        printf("%c -> %c\n",s[1],s[2]);
        b.push(a.top());
        a.pop();
    }
}

```

```

    }
    else
    {
        printf("%c -> %c\n",s[2],s[1]);
        a.push(b.top());
        b.pop();
    }
}
else
{
    printf("%c -> %c\n",s[3],s[1]);
    a.push(c.top());
    c.pop();
    if(b.size()==n+1||c.size()==n+1)
break;
    if(c.top()>b.top())
    {
        printf("%c -> %c\n",s[2],s[3]);
        c.push(b.top());
        b.pop();
    }
    else
    {
        printf("%c -> %c\n",s[3],s[2]);
        b.push(c.top());
        c.pop();
    }
}
i=0;

```

```
    }  
  }  
}
```

四、问题及解决过程

由于之前没有做过非递归的汉诺塔，做这个题还是费了很长的时间，最后通过找规律发现了非递归的规律，但是开始的时候还没有发现偶数和奇数的规律是有一定区别的，最后反复通过多组例子的实际操作，偶数和奇数分开情形写，才最后通过了所有的测试点。

实验五 哈弗曼编码

一、解题思路

给定一段文字，统计出字母的出现的频率，我们可以根据哈夫曼算法给出一套编码，使得用此编码压缩的原文可以得到最短的编码总长，不过哈夫曼编码不是唯一的，此题就是判断任意一套编码是否是哈夫曼编码。

分析问题可以知道，判断是否是哈夫曼编码，可以通过判断编码后的文本长度是否是和哈夫曼编码的文本长度相等，但是只判断这个是不够的，还有可能长度一样，但是某个字母的编码是另一个字母编码的前缀，这样的编码也不符合要求的，不是哈夫曼编码。

二、程序设计

求哈夫曼树的所有叶结点的带权路径长度之和（求树的 WPL）：

利用优先队列 `priority_queue<int,vector<int>,greater<int> > q`;最小的在队首，构造哈夫曼树，

每次提取队列中最小的两个，计算编码的总长度：

```
int x=q.top();q.pop();
```

```
int y=q.top();q.pop();
```

```
q.push(x+y);
```

```
wt+=x+y;
```

判断是否有前缀码：

1、利用 `sort` 按每个字母的编码的长度递增排序；

2、利用 C++ 中的 `substr` 函数截取字符串判断某个字符的编码是不是另一个字符的编码的前缀。

```
int isPrefix() {///判断是否有前缀码

sort(cd+1,cd+1+n,cmp);

for(int i=1;i<=n;i++) {

for(int j=i+1;j<=n;j++) {

if( cd[j].substr(0,cd[i].size()) == cd[i] ) return 1;

}

}
```

三、程序详解及运行结果

核心代码：

```
#include<bits/stdc++.h>
using namespace std;
int n,m,w,wt,res;
char ch;
string cd[100];
map<char,int> ma;
priority_queue<int,vector<int>,greater<int> > q;
bool cmp(string x,string y) {///根据长度排序
return x.size()<y.size();
}
int isPrefix() {///判断是否有前缀码
sort(cd+1,cd+1+n,cmp);
for(int i=1;i<=n;i++) {
for(int j=i+1;j<=n;j++) {
if( cd[j].substr(0,cd[i].size()) == cd[i] ) return 1;
}
}
return 0;
}
int main(){
```

```

    cin>>n;
    for(int i=1;i<=n;i++) {
cin>>ch>>w;
ma[ch]=w;
q.push(w);
    }
    ///计算 WPL
    while(q.size()>=2) {
int x=q.top();q.pop();
int y=q.top();q.pop();
q.push(x+y);
wt+=x+y;
    }
    q.pop();
    cin>>m;
    for(int i=1;i<=m;i++) {
res=0;
for(int j=1;j<=n;j++) {
cin>>ch>>cd[j];
res+=ma[ch]*cd[j].size();
}
if(res!=wt) cout<<"No"<<endl;
else if(isPrefix()) cout<<"No"<<endl;
else cout<<"Yes"<<endl;
    }
    return 0;
}

```

运行结果:

```

F:\C++\未命名2.exe
7
A 1 B 1 C 1 D 3 E 3 F 6 G 6
4
A 00000
B 00001
C 0001
D 001
E 01
F 10
G 11
Yes
A 01010
B 01011
C 0100
D 011
E 10
F 11
G 00
Yes
A 000
B 001
C 010
D 011
E 100
F 101
G 110
No

```

```

A 00000
B 00001
C 0001
D 001
E 00
F 10
G 11
No

```

四、问题及解决过程

思考问题开始的时候欠妥，只是想到，哈夫曼编码后的字符串长度应该相等，没有注意到可能会有前缀码的问题，没有注意到一个符号的编码可能是另一个符号的编码的前缀，这样的编码是不符合要求的，不是哈夫曼编码，这样的编码是错误的。后来我发现了这个问题后，通过截取字符串判断一个字符的编码是否是另一个字符的前缀编码解决了这个问题，排除了前缀码的错误情况，最后通过了所有测试点。

实验六 判断是否有欧拉回路-哥尼斯堡的“七桥问题”

一、解题思路

本体要求判断是否是欧拉回路，欧拉回路是指不令笔离开纸面，可画过图中每条边仅一次，且可以回到起点的一条回路。现给定一个无向图，判断是否具有欧拉回路？若欧拉回路存在则输出 1，欧拉回路不存在则输出 0。

通过图中所有边一次且仅一次行遍所有顶点是回路，称为欧拉回路。具有欧拉回路的图称为欧拉图。如果一个无向图是欧拉图，则当且仅当图是连通图且没有奇度顶点，那么，我们就仅仅需要判断这个图的连通性和顶点的度数就好啦。

1、判断顶点的度数：

我采用 `vector<int> G[maxn]`，把和一个顶点所有相邻的点都插入进去这个数组中，最后判断如果 `G[i]` 的大小对 2 取余等于 0，则这个顶点的度为偶数。

2、判断图的连通性

方法一 DFS 搜索判断：从任一结点开始，进行一次深度优先遍历。深度优先遍历的结果是一个图的连通分量。当一次遍历没有访问到所有结点，那么就说明图是不连通。

方法二 BFS 搜索判断：从一个结点开始，访问与其关联的所有结点，将这些结点入队，重复此过程，直至队列为空。访问的结点构成一个连通分量，若该连通分量未包含所有结点，则无向图不连通。

方法三利用并查集判断：根据读入的边对进行集合的划分，读入的边对证明是相连的，那么我们就划分到一个集合中，然后读入新的边对就划分到新的集合中，一旦读入的边对（两个顶点）都在同一个集合中出现过，那么证明是相互连通的，如果读入的边对，两个顶点是出现在两个集合中，那么说明这两个集合至少通过该边是可以相连的，那么集合进行合并。重复以上过程，即可得到求解结果。最后只

要判断所以结点是否在一个集合就可得到图是否连通，若在一个集合，则图连通，反之不连通。

由于并查集不是很熟悉，这里我的具体实现代码采用并查集判断。

二、程序设计

判断度：

1、声明 vector: `vector<int> G[maxn];`

2、读入边 a-b, `G[a].push_back(b);`

`G[b].push_back(a);`

3、如果 `G[a]` 的大小为偶数，则顶点 a 的度为偶数。

利用并查集判断图连通性：

并查集模板：

```
void init_set() {                                //初始化

    for(int i = 1; i <= maxn; i++)

        s[i] = i;

}

void union_set(int x, int y) {    //合并

    x = find_set(x);

    y = find_set(y);

    if(x != y) s[x] = s[y];

}

int find_set(int x) {                //查找    （递归）
```

```
return x==s[x]? x:find_set(s[x]);
```

```
}
```

三、程序详解及运行结果

核心代码：

```
#include<iostream>
#include<string>
#include<vector>
#include<cstdio>
using namespace std;
#define Inf 0x3f3f3f3f
const int maxn=1005;
vector<int> G[maxn];
int n,m;
int pre[maxn];
void init(){
    for(int i=1;i<=n;i++)
        pre[i]=i;
}
int find(int x){
    if(pre[x]==x)
        return x;
    else
        return pre[x]=find(pre[x]);
}
void merge(int x,int y){
    int fx=find(x);
    int fy=find(y);
    if(fx!=fy)
        pre[fx]=fy;
}
int main()
{
    cin>>n>>m;
    init();
    int a,b;
    for(int i=1;i<=m;i++){
        scanf("%d%d",&a,&b);
```

```

        merge(a,b);
        G[a].push_back(b);
        G[b].push_back(a);
    }
    int cnt=0,num=0;
    for(int i=1;i<=n;i++){
        if(G[i].size()%2)
            cnt++;
        if(pre[i]==i)
            num++;
    }
    if(cnt==0&&num==1)
        cout<<"1"<<endl;
    else
        cout<<"0"<<endl;
    return 0;
}

```

运行结果：

```

F:\C++\未命名2.exe
6 10
1 2
2 3
3 1
4 5
5 6
6 4
1 4
1 6
3 4
3 6
1
-----
Process exited after 2.16 seconds with return value 0
请按任意键继续. . .

```

```

F:\C++\未命名2.exe
5 8
1 2
1 3
2 3
2 4
2 5
5 3
5 4
3 4
0
-----
Process exited after 1.728 seconds with return value 0
请按任意键继续. . .

```

四、问题及解决过程

在判断顶点的度的时候开始的时候又用一个数组记录每个顶点的度，发现这样很麻烦，后来使用 `vector`，判断 `vector` 的大小，这样就很方便的可以判断顶点的度。

判断图是否连通的时候，由于以前没怎么使用过并查集，先用 DFS 搜索写了一次过了，但是并查集写的时候出了些问题，后来先仔细查了查并查集的使用方法，使用并查集判断了图的连通性。

实验七 拓扑排序-Is Topological Order

一、解题思路

题目本身的意思是传进来一个数组判断这个数组里面的数字能否构成一个拓扑序列，对一个有向无环图(Directed Acyclic Graph 简称 DAG)G 进行拓扑排序，是将 G 中所有顶点排成一个线性序列，使得图中任意一对顶点 u 和 v，若边 $\langle u, v \rangle \in E(G)$ ，则 u 在线性序列中出现在 v 之前。

思路：将传进来的数组用一个队列保存，然后依次出队，出队的顶点必定是入度为 0 的顶点，若入度不为 0，直接返回 false。队列空了之后还没有返回 false 就说明能够构成一个拓扑序列。

二、程序设计

1、初始化读入并用邻接表存储有向图，并储存每个节点的入度；

2、对每个查询序列先读入，并复制每个节点的入度作测试；

3、对序列从零开始验证节点入度是否为 0，如是则将该节点对应的邻接表内的元素入度减 1；如以上循环在过程中失败，则表示非拓扑序列，输出组号，否则表示此序列为拓扑序列；返回零值。

三、程序详解及运行结果

核心代码：

```
bool IsTopSeq(LGraph Graph, Vertex Seq[]) //用一个队列来处理
{
    // 题目测试数据最多 只有 1000 个，所以数组直接开 1000，不然会段错误。
    PtrToAdjVNode p;
    int i, qu[1000], rear = -1, w, front = -1;
    int count[1000];
    for (i = 0; i < Graph->Nv + 1; i++)
        count[i] = 0;
    for (i = 0; i < Graph->Nv; i++)
    {
        p = Graph->G[i].FirstEdge;
```

```

        while (p)
        {
            count[p->AdjV]++;
            p = p->Next;
        }
    }
    // 顶点是从 0 开始存放的 比如; 1 被放在 0, 2 被放在 1, i 被放在 i-1 个位置上面
    // 所以 传进来的数组 Seq[i] = Seq[i] - 1; 大坑一个
    for (i = 0; i < Graph->Nv; i++)
        qu[++rear] = Seq[i] - 1; //按照传进来的拓扑序列依次进队

    while (front != rear)
    {
        w = qu[++front]; //出队
        if (count[w] != 0) return false; //出队的一定是入度为 0 的顶点, 否则返回 false;
        p = Graph->G[w].FirstEdge;
        while (p)
        {
            count[p->AdjV]--;
            if (count[p->AdjV] < 0) return false;
            p = p->Next;
        }
    }
    return true;
}

```

四、问题及解决过程

开始的时候没有注意到题目中的一句很关键的提示:

Note: Although the vertices are numbered from 1 to MaxVertexNum, they are indexed from 0 in the LGraph structure.

意思就是 输入的顶点是从 0 开始存放的。顶点是从 0 开始存放的 比如; 1 被放在 0, 2 被放在 1, i 被放在 i-1 个位置上面, 所以在开始的时候要注意处理,

传进来的数组 $Seq[i] = Seq[i] - 1$ 。

实验八 贪心算法-月饼

一、解题思路

看到题目判断这是典型的贪心算法例题，也是较简单的贪心算法例题；

可通过每种月饼的 总售价/库存量 求出每种月饼的单位售价（单位价值），均用 float 或 double 表示。

然后根据单位售价递减排序，并按此顺序出售以获得最大收益。注意出售第 i 种月饼时，若剩余市场需求量 $<$ 这种月饼的库存量时，出售这种月饼可获得的收益 = 剩余市场需求量 * 这种月饼的单位售价。

本题需要注意的是，除了种类和最大需求量是整型数据，其他数据皆是小数，这意味着库存量也是小数而不是正整数，如果不注意数据类型，很容易被卡住。

二、程序设计

1、读入数据，把月饼的总售价和库存量存在一个个结构体里

2、按照单位售价递减排序，这里可以采用 C++ 中的 sort 对结构体排序

3、遍历月饼

if（月饼的库存量小于需求）

then 收益等于月饼的总售价，需求量减去月饼的库存量

else 出售这种月饼可获得的收益 = 剩余市场需求量 * 这种月饼的单位售价

三、程序详解及运行结果

核心代码：

```
#include<iostream>
#include<algorithm>
#include <iomanip>
using namespace std;
```

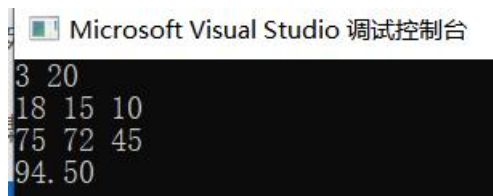
```

struct mooncake
{
    double price;
    double liang;
    double averge;
};
bool cmp(struct mooncake a, struct mooncake b)
{
    return a.averge>b.averge;
}
mooncake m[1005];
int main() {
    int x, y;
    cin >> x >> y;

    for(int i = 0; i<x; i++)
    {
        cin >> m[i].liang;
    }
    for (int i = 0; i < x; i++)
    {
        cin >> m[i].price;
        m[i].averge = m[i].price / m[i].liang;
    }
    sort(m, m + x,cmp);
    double sum_price=0;
    int sum=y;
    for (int i = 0; i < x; i++)
    {
        if (m[i].liang<sum)
        {
            sum_price += m[i].price;
            sum -= m[i].liang;
        }
        else
        {
            sum_price += m[i].averge*sum;
            break;
        }
    }
    cout << setiosflags(ios::fixed) << setprecision(2) << sum_price << endl;
}

```


运行结果：



```
Microsoft Visual Studio 调试控制台
3 20
18 15 10
75 72 45
94.50
```

四、问题及解决过程

这道题是很简单，很基础的一个贪心问题，每次先买收益最大的月饼，最后达到总收益最大，典型的基本贪心问题，开始的时候没有注意数据类型，导致输出出现问题，后来改用 `double` 成功通过所有测试点。

实验九 快速排序-找第 k 小的数

一、解题思路

题目要求设计一个平均时间为 $O(n)$ 的算法，在 n 个无序的整数中找出第 k 小的数。快速排序就是利用分治的思想对数组进行快速排序，具体思路如下：

1、利用快排的 `Partition()` 函数将数组分成两部分，返回基准值 `value`，小于 `value` 的都在左边，大于的在右边

2、如果 `index` 刚好等于 k ，则说明 `index` 位置的数就是我们要找的数，如果值小于它，就肯定在左边，大于就在右边

3、递归在 `index` 的左边或者右边进行查找

二、程序设计

快速排序伪代码：

```
partition(A, lo, hi)

    pivot = A[hi]

    i = lo //place for swapping

    for j = lo to hi - 1

        if A[j] <= pivot

            swap A[i] with A[j]

            i = i + 1

    swap A[i] with A[hi]

return i

quicksort(A, lo, hi)
```

```
if lo < hi

    p = partition(A, lo, hi)

    quicksort(A, lo, p - 1)

    quicksort(A, p + 1, hi)
```

三、程序详解及运行结果

核心代码：

```
#include<iostream>
#include<algorithm>
using namespace std;
int a[10005];
int partition (int left,int right){
    int temp = a[right];
    while (left<right)
    {
        while (left < right && a[left] <=temp)
            left++;
        a[right] = a[left];
        while (left < right &&a[right] >=temp)
            right--;
        a[left] = a[right];
    }
    a[right] = temp;
    return right;
}

int find(int left, int right, int k) {
    int temp = partition(left, right);
    if (temp == k)
    {
        return a[k];
    }
    else if (temp < k)
        find(temp + 1, right, k);
    else
        find(left, temp - 1, k);
}
```

```
int main() {  
    int n, k;  
    cin >> n >> k;  
    for (int i = 1; i <=n; i++)  
        cin >> a[i];  
    cout << find(1, n, k) << endl;  
}
```

运行结果：

Microsoft Visual Studio 调试控制台

```
10 4  
2 8 9 0 1 3 6 7 8 2  
2
```

F:\C++\Project3\Debug\Project3.exe (进程 8420) 已退出，返回代码为: 0。
若要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...

四、问题及解决过程

开始的时候做这个题，直接套用模板对整个数组进行快排，对中间的左右两部分进行递归，后来发现查找第 k 小的数，不需要递归左右两部分，只需要递归 k 所在的部分就够了，这样可以节省很多时间。

实验十 背包问题-凑零钱

一、解题思路

分析题目，这是一个典型的背包问题。要求输出满足条件的最小序列，对于背包问题，通常采用的是动态规划的思想，而动态规划的核心思想就是要找到状态转移方程，这里就是要用最小面额的硬币构成需要的钱。具体思路是：首先先对硬币从大到小进行排序，因为题目要求最小序列，所以逆序遍历时需要从小的硬币开始遍历，同时要用一个 path 数组进行路径的记录，记录下选择的硬币。

二、程序设计

- 1、先对硬币进行从大到小排序
- 2、利用滚动数组进行动态规划填表，同时记录路径：

```
if (dp[j] <= dp[j - v[i]] + v[i])  
  
    {  
  
        path[i][j] = 1;//记录路径  
  
        dp[j] = dp[j - v[i]] + v[i];  
  
    }
```

- 3、输出满足条件的最小序列。

三、程序详解及运行结果

核心代码：

```
#include<iostream>  
#include<algorithm>  
#include<vector>  
using namespace std;  
int dp[105];  
int path[10001][101];  
int cmp(int a, int b)  
{
```

```

    return a > b;
}
int main() {
    vector<int> v;
    int n, m;
    cin >> n >> m;
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        int temp;
        cin >> temp;
        v.push_back(temp);
        sum += temp;
    }
    if (sum < n)
    {
        cout << "No Solution" << endl;
        return 0;
    }
    sort(v.begin(), v.end(), cmp);
    for (int i = 0; i < n; i++)
    {
        for (int j = m; j >= v[i]; j--)
        {
            if (dp[j] <= dp[j - v[i]] + v[i])
            {
                path[i][j] = 1;
                dp[j] = dp[j - v[i]] + v[i];
            }
        }
    }
    if (dp[m] != m) {
        cout << "No Solution" << endl;
    }
    else {
        int j = m;
        int i = n - 1;
        while (j > 0) {
            if (path[i][j] == 1) {
                if (j != m) {
                    cout << " ";
                }
                cout << v[i];
            }
        }
    }
}

```

```

        j -= v[i];
    }
    i--;
}
}
return 0;
}

```

运行结果：

```

F:\C++\未命名2.exe
4 8
7 2 4 3
No Solution
-----
Process exited after 1.783 seconds with return value 0
请按任意键继续. . .

F:\C++\未命名2.exe
8 9
5 9 8 7 2 3 4 1
1 3 5
-----
Process exited after 9.031 seconds with return value 0
请按任意键继续. . .

```

四、问题及解决过程

在记录路径的时候，记录路径出现了一些问题，对背包问题掌握不熟练，动态规划方面的知识还需要加强，这部分动态规划算法掌握欠缺。

刚开始做这个题采用的是二维数组记录动态规划的表格，但是后来发现当前状态只和上一行数据有关，所以后来就改成了一维数组，利用滚动数组来记录动态规划的表格，这样节省了空间，并且更方便。