2018 年华东师范大学机试题目解析（计算机系）

作者: malic　QQ:602451067　Email:m6024m@163.com

前言

计算机系的考研机试在复试中所占比例较大，占复试成绩的 40%，在准备复试时必须将之重视起来。相比于专业课和英语的面试来说，机试是客观型的问题，可以凭实力拉开一些差距，甚至能够绝境反超或可能因准备不充分而被反超淘汰。

2018 年计算机系的机试题目总体难度一般，算法方面主要涉及递归和排序，历年来会出现的动态规划算法今年没有题目涉及到，也没出到数据结构上的树、图的算法。题目把重难点放在了数据量上。当然，对大数的处理也是一项重要技能，如果不会对超过 long long 类型的数字进行处理，那么这些题目只能够拿到小部分样例的分数。这一点也许正是 2018 年题目看似难度不大，实则普遍得分并不高的原因。下面我将在本篇中对每个题用 C, C++, Python3 语言分别做出详细的解析。其中 C 语言程序是基本实现，而 C++的程序将重点放在使用 C++语言的特性如 STL 模板库、运算符重载等技术上来实现，而 Python3 将运用到 Python 语言的一些特性，代码量更小，程序逻辑更清晰。希望本文能对你提高程序设计的实战技能有一点帮助。

我力图将本篇做到完美，但限于水平不足，错误在所难免，恳请各位朋友们批评指正，并多提宝贵意见。本文全部程序代码在 https://github.com/lyw6024/ECNU_2018CS_Contest 上发布，同时也非常欢迎大家与我共同探讨交流知识和经验。

Contents

前言、目录 0
程序设计小技巧 1
　节省反复输入测试数据的时间——输入重定向 1
　搭建代码脚手架 2
　打表 4
　数学优化 4
　其它 5
问题与解 6
　Problem A: 1+1 6
　Problem B: LOGO 8
　Problem C: Easy Sort 10
　Problem D: Boss’s Boss 15
　Problem E: Snake Matrix 19
后记 24

程序设计小技巧

节省反复输入测试数据的时间——输入重定向

许多程序的输入很长，可能是一段文字、一个数列，如果在你程序编写的初期要测试它们，可能要频繁的复制粘贴（Windows 命令终端有时还不能用 ctrl+V 粘贴，需要去菜单中在编辑中选择粘贴选项）。或者你要做下一个测试点，需要更改一下剪贴板中的测试数据，那样每次运行程序都要全部重新输入一遍。为了节省反复输入的时间，你可能想到，改写成文件读入的方式，一运行程序就让数据从文件中读入就好了。C 语言读文件需要你定义文件指针，然后输入输出的 scanf(), printf()函数要改成相应的 fscanf(), fprintf()，而你调试完成后还要再改回去才能顺利通过 OJ (Online Judge,在线测评系统)的测试。这样省去手动输入的步骤，确实可以让你的注意力放在对程序逻辑的控制上，但是还不够方便，如果有多处进行了输入输出，最终交到 OJ 之前遗漏了任何一处对文件 I/O 的改动都不能通过评测。

用文件替代手动输入确实是我们的解决办法，但我们并不需要定义文件指针。解决方法是在终端使用文件输入输出重定向，'<'表示重定向输入，用'>'表示重定向输出。我们一般来说只需要重定向输入。输入重定向不需要对程序做任何改动，原本 scanf()接收键盘输入的数据，用输入重定向之后它将自动接收重定向的输入数据。所以这样测试完成后并不需要做任何改动，可以直接提交到 OJ 上。

假设我们有可执行程序为 p1 ，要想把_in1 当中的内容当作对程序的输入，先将命令行的路径调整到 p1 下（当然_in1 也在同一路径），然后命令行中输入

```
p1<_in1
```

如果你的程序是正确的，控制台上就会显示根据_in1 中的内容作为输出的程序运行结果。

比如，编写有程序 sayHello，输入一个数字 N，输出 N 行"Hello,world!"，在文件_in 中有数字 5，那么就可以使用 `sayHello<_in` 将 5 自动输入到程序中：

```
E:\codes\201804\poj>sayHello<_in
Hello,world!
Hello,world!
Hello,world!
Hello,world!
Hello,world!
```

觉得这还太麻烦？好吧，还可以有更省事的，那就是用重定向 freopen()函数，这会省去在命令终端的操作，程序编译好了会直接进行重定向。这在 C 和 C++中适用。

C 语言的基本输入输出函数 scanf(), printf()定义在头文件<stdio.h>当中，stdio 即 standard input and output（标准输入输出），默认的标准输入 stdin 是键盘，标准输出 stdout 是显示器。我们只需要在程度代码的 main 函数的开始部分用 freopen()函数将 stdin 定义到我们的输入文件中，就能让标准输入改为文件输入，这样在程序运行时也不再需要从键盘读入数据了。

freopen()函数在<stdio.h>当中，其声明为

```
FILE *freopen( const char *filename, const char *mode, FILE *stream )
```

freopen()接受三个参数，分别是：一个 char*型变量 filename 表示文件名，一个 char*型变量 mode 表示重定向类型，一个文件指针 stream 表示文件流。如果成功则返回输入流的文件指针，失败返回 NULL。一般来说我们要用它只做输入重定向，如果文件名叫_in，就直接使用函数 `freopen(“_in”,”r”,stdin);`，需要更改输入文件，只改第 1 个函数参数即可。注意，如果重定向输入的是文本文件.txt 格式，不要忘记在重定向时把扩展名带上，否则是找不到相应文件的。例如用 Windows 记事本新建的 case0.txt 当中存放输入数据，应当使用命令 `p1<case0.txt` 或使用 C 语言的语句 `freopen(“case0.txt”,”r”,stdin);`进行重定向。

另外，若使用 freopen()函数，在提交到 OJ 的时候要把这句删去，不然程序就不从默认的输入开始读取数据了，这样会得到 Wrong Answer 的结果。

搭建代码脚手架

物理学家爱因斯坦得出一个非常著名的方程$E = mc^2$,其含义为 Error=more code^2（大误）。这提示我们要将程序写的短小一些，至少每个函数可以独立地写短一些。当然，在做题的时候，为了监测程序运行是否符合预期，你可能需要在适当位置加入一些输出语句来监测相关的值（当然你可以用更灵活的 debugger tools，比如 gdb，或是 IDE 的各种 debugger）。在初学排序时，也许会在每趟排序后都会把整个数组输出到控制台，以此确定排序函数的流程是否正确。或者在做递归函数时把传入函数的参数值先输出一下，看看递归函数是否按预期的进行。这些输出语句都是类似脚手架一样的在帮助你随时监视程序走向，一旦在某处出现意外值，可以很容易的定位到出错点。

在 OJ 上提交代码之前，一定要把这些脚手架拆去。逐行的删可不是个好办法，删除后还要再编译运行一下以检验是否进行了正确的删除，这种麻烦的后续工作也使很多程序设计者在初学时期并不愿意搭建这种脚手架代码。我们可以利用 C 语言的编译预处理 #if 和 #endif 语句，这能够一键拆除脚手架。#if 与 #endif 的用法非常简单

```
#if PARAMATER
<statement>
#endif
```

如果 PATAMETER 为非零常量，那么<statement>语句就原样保留，如果 PARAMETER 是 0，那么从#if 到#endif 当中的语句将在编译预处理阶段被删去。我们可以预先定义好一个常量比如#define DEBUG 1，这样在程序中，将脚手架监控代码都编为形如

```
#if DEBUG
printf(“DEBUG:index=%d\n”,index);
#endif
```

的形式。在本地测试程序时就保留 DEBUG 为 1，上交到 OJ 上的时候，只需要把 DEBUG 的值改为 0，或者删去 DEBUG 的 define 定义，就能直接通过。另外，把 freopen()语句也放在#if DEBUG 到#endif 当中，这时你在本地测试的程序和 OJ 上能正常测试的代码只是由 1 改为 0 的一个变动而已。

一个代码示例：

```
#define DEBUG 1
int main(void){
    #if DEBUG
    freopen(“_in”,”r”,stdin);//重定向至本地名为“_in”的文件作输入.DEBUG 改为 0 就不再重定向
    #endif
    ...<some statements>...
    #if DEBUG
    for(i=0;i<N;i++)  //可以输出数组 a[i]作测试.DEBUG 改为 0 再编译就不再输出
        printf(“[%d]:\t%d\n”,i,a[i]);
    #endif
    ...<some statements>...
    printf("%d\n",res);//原本的输出语句不放在#if DEBUG 与#endif 间中就能原样输出
    return 0;
}
```

记住不要添加太多输出语句，控制台是有行数限制的，如果输出 1000 行数值，翻到控制台最上边会发现好多数字已经到控制台的界外看不到了。这种情况当然可以重定向输出，但往往这些数值参考价值不大，要记得将脚手架的输出代码放在合适的位置，让它真正发挥其作用，而不是为了能输出而输出。最好是把程序写成不同小段，对每段分别检测，检测无误就应当将脚手架删去，留给后边真正需要再搭建脚手架的地方。

打表

小学学习乘法的时候，老师要我们背“九九乘法表”，做一位数与一位数的乘法运算时，背乘法表直接出结果。计算两位数乘法就需要再列式算，那么如果老师要求把两位数和两位数相乘的结果也背下来，那么两位数和两位数的乘法运算也立即得结果了，不过这种记忆量对于小学生来说太过分了。但是对于计算机没有问题，存储数据对于计算机来说轻而易举。在程序里反复用的关键运算就该减少每次运算的工作量，以提高总体运算速度。如果算一次得出了结果，将结果保存下来，以后再做运算时直接访问数据就最好了，也就是所谓“空间换时间”。例如题目需要反复判断某个数字是否为数列中的某项，又不易用通项公式直接判断时，这时可以预先用一个数组 a[i]，来表示数字 i 是否为存在于数列之中，C 语言可以用 char 型数组节省空间，C++可以用 bool 型。例如质数、fibonacci 数列等相关的问题。例如，有这样两个数列：

isPrime[N]={0,0,1,1,0,1,0,1,0,0,0,1,...}

primer[N]={2,3,5,7,11,13,17,19,23,...}

isPrime[]将根据数组下标是否为质数，构成了逻辑值一维数组，如果下标 i 是质数，则 isPrime[i]=1，否则为 0。而 prime[]数列是货真价实的质数数列。对于判断数字 k 是否为质数的问题，若用 prime [N]的数列，最快使用二分搜索需要 O(lgN)的时间，而在 isPrime[N]当中直接判断 isPrime[k]是 0 还是 1 即知 k 是否为质数，仅需 O(1)的时间。

另外，如果遇到问题你实在不知道怎么入手，只能笔算出几个数据的数值，那也可以把你已经得到数据的结果当常量数组直接写在程序里，并直接输出。一般这办法只有在输入和输出数据非常少的时候才能奏效。或者，你想不出比较快的算法，直接交到 OJ 上会 Time Limit Exceed，那就在可以在本地将数值算出，在上交的程序中把算出的结果作为常量数组写在程序里，只要内存不超限，时间必然是符合要求的。

数学优化

利用一些数学处理，主要是优化时间。根据加减乘除和乘幂运算的定义，在程序中遇到的迭代的减法也许可以改用除法完成，这样会降低一些时间。有的题目可能还要试着去探索数学规律，尤其数列和矩阵类型的题目（但这种规律不一定存在，比如质数数列就找不到通项）。

有了数学公式，运算速度不仅非常快，并且接受的数据量还会更大。许多这类“输入一个数，输出一个数（或数列）”的题目都是可能有递推公式的，如果数据量小，也可以本地算出结果然后打表直接交，这样不优化时间也完全没问题了。

其它

如果程序交到 OJ 上报告结果有个别结果没通过，这情况有时比答案不正确还难处理。首先确定思路是正确，但要改的部分很难查，可能涉及数组范围，或是变量溢出等问题。如果查了好久还查不出来，我个人常用的方法是换种语言重写一遍，不仅能重新梳理思路，并且能改正可能存在的原先代码中因手误而导致的个别点的输出错误。比如改写成如 Python 这种可以自动类型的语言，可以避免数值溢出的错误。实在不行，就 C 语言的程序做适当调整，改用 C++语言的 g++编译器提交（比如 printf()改作 cout）。我在 OJ 上做的时候有时用 C++做的提交 Wrong Answer，若能改写成 Python 程序有时就会 Accepted，并且 Python 代码更短，改写难度不大，如果考试还有时间，又实在想不出怎么能改好，不妨换个语言重写试试。这也是没办法的办法了。

Prob	Result All	Time (ms)	Mem (MB)	Length	Lang All
3936 (D)	Accepted	17	3.1	153	Python
3936 (D)	Wrong Answer			324	C
3936 (D)	Wrong Answer			322	C
3936 (D)	Wrong Answer			322	C++
3936 (D)	Wrong Answer			256	C++

问题与解

Problem A: 1+1

给一个小学生都会算的 1 位数与 1 位数运算的代数式，请你求出这个表达式的值。
表达式仅含+-*三种运算。

```
Sample Input 1:
1+1
Sample Output 1:
2
Sample Input 2:
3*4
Sample OutPut2:
12
```

本题难度极小，简单到连解析都写不出来。

C 语言示例

```c
#include <stdio.h>
int main(void)
{
    char op,n1,n2;
    int res;
    n1=getchar();
    op=getchar();
    n2=getchar();
    switch(op)
    {
        case '+':
            res=(n1-'0')+(n2-'0');
            break;
        case '-':
            res=(n1-'0')-(n2-'0');
            break;
        case '*':
            res=(n1-'0')*(n2-'0');
            break;
        default:
            break;
    }
    printf("%d\n",res);
    return 0;
}
```

C++示例代码

```cpp
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
int main(void)
{
    int x,y,res;
    char op;
    cin>>x>>op>>y;
    switch(op)
    {
        case '+':
            res=x+y;
            break;
        case '-':
            res=x-y;
            break;
        case '*':
            res=x*y;
            break;
        default:
            break;
    }
    cout<<res<<endl;
    return 0;
}
```

Python3 当中有个 eval()函数，它接收一个 str 类型的值，这个字符串是数学表达式的话就会正确返回这个数学表达式的结果。所以这个题用 Python3 就是个第一行输出第二行输出的名副其实的签到题。

Python3 示例代码

```python
expr=input()
print(eval(expr))
```

Problem B: LOGO

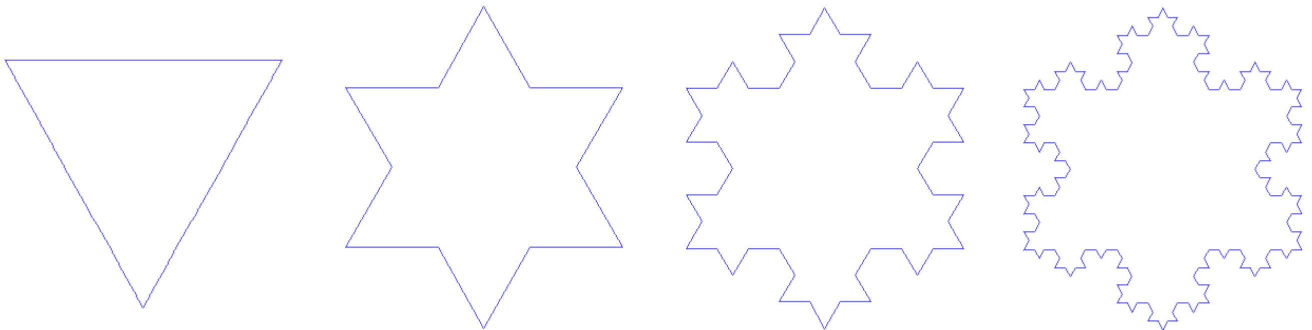
现在小学生也在学习基本的编程，课程目标是让小学生能够有基本的算法思想，并不涉及复杂的数据和实现细节与原理。LOGO 语言就非常适合小学生学习，它通过绘图的方式来直观的表现出如何用程序代码控制事物。例如控制台上初始给出一个点，使用语句 `FD 1/1` 表示将控制台上的点 Forward 1/1 的距离，即，向当前方向移动 1 的距离，这样就画出一条线段。语句 `LT 60` 则表示当前朝向向左转 60 度，接着再使用语句 `FD 1/1` 就画出一条与之前的直线夹角为 120 度的一条线段，这时控制台上就有绘制出了一条折线段。

现在的任务是输出一段能绘制分形的 LOGO 语言的程序代码。

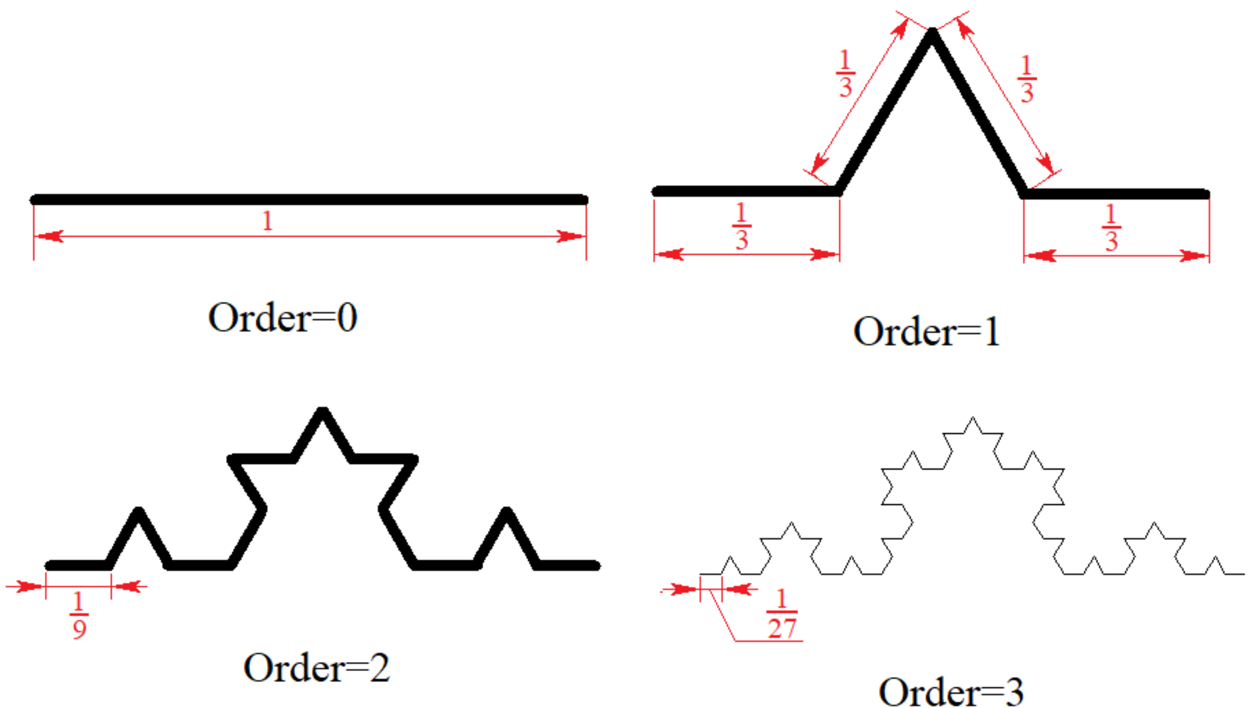
如果你还对分形不了解，下面就先介绍一下分形：

分形(Fractal) 是一个几何形状可以分成数个部分，且每一部分都（至少近似地）是整体缩小后的形状，即具有自相似的性质。自然界中一定程度上具有分形的性质的事物有云朵、闪电、植物根系、雪花等等。著名的科赫曲线就是一种分形，它绘制的是形态类似雪花的图案。

以下是 0 阶到 3 阶的科赫曲线：



本题的任务只要求画出科赫曲线的一部分即可，如 0 至 3 阶是如下的画法：



要求你的程序输出 LOGO 语句，使它画出 N 阶图示的分形曲线。

输入：

1 行，1 个数字 n，表示图形的阶数(0<n<10)

输出：

能绘制上述图形的 LOGO 程序代码

如果你有递归的思想，那么应该不难看出，分形的图案就是一个递归的形式，我们应该先把题目描述的图形画法转化成算法步骤：

画一个长为 x 的图形：

如果 x 已经不能再分成 x/3（此题是用阶数 n 控制是否需要分成 x/3），就画出长为 x 的直线。

否则，画出长为 x/3 的图形，左转 60 度，画出长为 x/3 的图形，左转 240 度，画出长为 x/3 的图形，左转 60 度，画出长为 x/3 的图形，画完。

这样就有了递归函数的基本框架，再把相应的 LOGO 语言的指令填入，就是此题的答案了。本题的 C++和 Python 程序代码不再赘述。

```
#include <stdio.h>
#include <math.h>
void Fractal(int n,int level)
{
	if(level==1){
	    int p=pow(3,n);
	    printf("FD 1/%d\n",p);
	    printf("LD 60\n");
	    printf("FD 1/%d\n",p);
	    printf("LD 240\n");
	    printf("FD 1/%d\n",p);
	    printf("LD 60\n");
	    printf("FD 1/%d\n",p);
	}
	else{
	    Fractal(n,level-1);
	    printf("LD 60\n");
	    Fractal(n,level-1);
	    printf("LD 240\n");
	    Fractal(n,level-1);
	    printf("LD 60\n");
	    Fractal(n,level-1);
	}
}
void output(int n)
{
	Fractal(n,n);
}
```

```c
int main(void)
{
        int n;
        scanf("%d",&n);
        output(n);
        return 0;
}
```

Problem C: Easy Sort

给出一个含有 N (0<N<200000)个数字的数列，请你对它排序，每个数的范围均处于$[-10^{50}, 10^{50}]$。负数前有负号‘-’，正数前没有+号，所有数字均不含前导 0，零用一个 0 表示。

输入:

2 行，第 1 行有 1 个数字 N，表示数列中数据的个数

第 2 行有 N 个数字，表示待排序的数列，数字间用空格分隔，题目保证每个数字在$[-10^{50}, 10^{50}]$范围内。

输出:

1 行，将 N 个数字从小到大排序后的结果，数字间用空格分隔。

Sample Input :

```
5
991 -31 -1 5 -100000000000000000000000000000000000000000000000000
```

Sample Output:

```
-100000000000000000000000000000000000000000000000000 -31 -1 5 991
```

这题当然就是排序算法，排序可以自己写，也可以调用 stdlib.h 库中 qsort()或使用 C++ STL 的 sort()。问题的关键在于这种数据量在10^{50}的数字不可能保存成数值类型，必须先设法保存这些数据，不难想到，这种 50 位的数字的大整数可以保存成 char [] 类型，大整数的四则运算就是如此。那么排序怎么办呢。

quickSort()是基于比较的排序算法，所以我们需要实现的关键问题是大整数比较（基于比较的算法 quickSort()表现最好。与基于比较的排序对应的是基于散列的排序，如桶排序，时间复杂度仅为 O(n)，但它需要更多空间）。

关于排序算法的实现规则，核心就是每一步的比较函数的返回值，如果你要让待排序序列按从小到大的顺序，那么就应当让比较函数在两个参数 a,b 比较时，a<b 时返回值为 True。如果想让待排序序列从大到小，那么 a>b 时返回值为 True。回忆一下所有的排序算法，如果你想

更改排序规则，就在排序函数中改一下比较函数即可。一般比较的是基本类型的数据，这样一个小于号就充当了比较函数的角色。如果是复杂数据的排序，就需要自己定义这个比较函数。

解决本题可以定义一个结构体类型用于保存数据。结构成员有：数字的正负标记 bool 型、数字位数 int 型、完整的数字 char[]型。对数据进行比较的函数应当是：若两个结构体变量比较时正负性不同，则直接可以确定数据大小关系。若正负性相同，根据数字位数返回大小关系。否则，比较数字完整的值，即比较字符串（此时已经保证了两个数的正负性和位数相同，字符串比较函数可以与比较数字的值等价）。将这个比较函数传入排序函数，对结构体数组做排序即可顺利完成。

虽然 Python 的整型能够支持到非常大的数值，但如果真的使用 Python 整数保存数据的话，即使排序是 quickSort()也可能会超时，这类追求速度的题就尽量不要用 Python 或 Java。

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

#define MAXDIGIT 50
#define MAXN 200007
typedef struct numNode ElemType;

struct numNode{
    int isPositive;
    int digit;
    char value[MAXDIGIT+1];
};
ElemType a[MAXN];


int myCompare(ElemType a,ElemType b);
void sort(ElemType *a,int N);

int main(void)
{
    int i,N;

    scanf("%d",&N);
    for(i=0;i<N;i++){
        scanf("%s",a[i].value);
        a[i].digit=strlen(a[i].value);
        if(a[i].value[0]=='-')
            a[i].isPositive=0;
```

```
        else
            a[i].isPositive=1;
    }
    sort(a,N);
    for(i=0;i<N;i++)
        printf("%s ",a[i].value);
    printf("\n");
    return 0;

}
int myCompare(ElemType a,ElemType b)
{
    if(a.isPositive && b.isPositive == 1){
        //both of a and b are positive
        if(a.digit!=b.digit)
            return a.digit<b.digit;
        else
            return strcmp(a.value,b.value)<0;
    }
    else if(a.isPositive==0 && b.isPositive==0){
        if(a.digit!=b.digit)
            return a.digit>b.digit;
        else
            return strcmp(a.value+1,b.value+1)>0;
    }
    else
        return a.isPositive==0;
}


void QuickSort(ElemType *a,int lwbd,int upbd)
{
    int i=lwbd,j=upbd;
    if(i<j){
            ElemType tmp=a[i];
        while(i<j)
        {
            while(i<j && myCompare(tmp,a[j]))
                j--;
            if(i<j)
                a[i++]=a[j];
            while(i<j && myCompare(a[i],tmp))
                i++;
            if(i<j)
```

```
                a[j--]=a[i];
        }
        a[i]=tmp;
        QuickSort(a,lwbd,i-1);
        QuickSort(a,i+1,upbd);
    }
}
void sort(ElemType *a,int N)
{
    //封装排序接口
    QuickSort(a,0,N-1);
}
```

C++实现时，可以利用重载'<'运算符的特性，这与上述 C 语言解法中自己写 myCompare() 函数是类似的。重载小于运算后可以直接用 STL 中的 sort(a,a+N)排序。

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using std::sort;
const int MAXDIGIT=50;
struct numNode{
    bool isPositive;
    int digit;
    char value[MAXDIGIT+2];
    bool operator<(const numNode &b) const
    {
        if(isPositive==true &&b.isPositive==true){
            if(digit!=b.digit)
                return digit<b.digit;
            else
                return strcmp(value,b.value)<0;
        }
        else if( isPositive== false && b.isPositive==false){
            if(digit!=b.digit)
                return digit>b.digit;
            else
                return strcmp(value,b.value)>0;
        }
        else
            return b.isPositive;
    }
};
```

```
int main(void)
{
    int i,N;
    scanf("%d",&N);
    numNode *s=new numNode [N];
    for(i=0;i<N;i++)
    {
        scanf("%s",s[i].value);
        s[i].digit=strlen(s[i].value);
        if(s[i].value[0]=='-')
            s[i].isPositive=false;
        else
            s[i].isPositive=true;
    }

    sort(s,s+N);

    for (i=0;i<N;i++)
        printf("%s ",s[i].value);
    printf("\n");
    delete [] s;
    return 0;

}
```

Problem D: Boss’s Boss

有一个研究团队，团队分成许多研究小组，每个小组的一部分成员可能再分成小组。小组成员只知道自己的组长是谁。现在这个团队希望有一个程序能统计一下各组长带领小组的规模，即对每一个成员想知道自己及自己带领下的小组有多少人。

输入：

2 行，第 1 行有 1 个数字 N($0<N<2\times10^5$)，代表小组的人数.

第 2 行有 N 个数a_1,a_2，...，a_i，...，a_N，表示第 i 个人的领导是a_i。团队的领导用 0 表示，说明没有人做他的组长。数据保证没有环路。单独的一个成员视为 1 个人的小组。

输出：

1 行，N 个数字，表示第 i 名成员的团队的规模

Sample Input:

0 1 2 1 2 2

Sample Output:

6 4 1 1 1 1

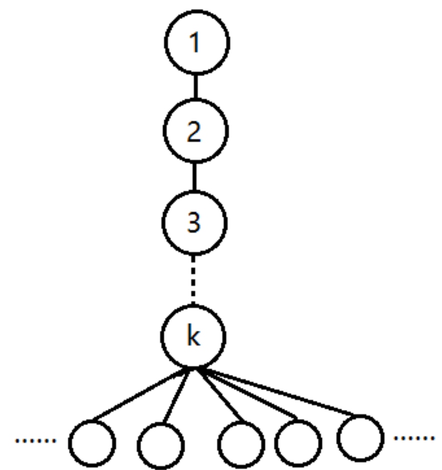
直接搜索倒是不难实现，读入数据是领导，对每个成员的结构体变量定义一个 leader 当作指针（广义的）指向领导，再在结构体里定义这个成员所带的团队规模。遍历每个人的时候使他规模增加 1，然后找到他的领导再增加规模，直到遍历完所有人员为止。

C 语言程序代码示例:

```
#include <stdio.h>
#define MAXN 200007
struct sNode
{
    int scale;
    int leader;
}a[MAXN];

int main(void)
{
    int i,N,curr_index;
    scanf("%d",&N);
    for(i=1;i<=N;i++)
    {
        scanf("%d",&(a[i].leader));
        a[i].scale=1;
    }
    for(i=1;i<=N;i++)
    {
        curr_index=i;
        while(a[curr_index].leader>0){
```

```
        // if current index also have boss,
        // go to his node and add his scale.
            curr_index=a[curr_index].leader;
            a[curr_index].scale++;
        }
    }
    for(i=1;i<=N;i++)
        printf("%d ",a[i].scale);
    return 0;
}
```



程序正确性显然，但这段程序不够快。试想一个团队的模式是类似于右图的形状，即上层的各个领导只带领一人，直到下边某个深度之后，领导了一大批人，这时遍历到最底层的这批人就要把整个自底向上的路径都遍历一趟。时间复杂度 $O(N^2)$。

不难看出，时间浪费在了最后这一批人身上。下层团队每人都要遍布所有上层访问一次。但在实际统计的时候并不是每个人都层层上报，而是在下层的团队统计完成后交给上层，这个上层全统计完成后再交到它的上一层，直到汇总到最大领导。这就是 O(N)的方法。程序中把输入数据按它所处的层分开，按成员所处的层按自下层向上的顺序依次遍历。只需要给结构体加一个索引 index，把团队规模的数据分离出来，只对结构体按层数进行排序，就可以实现一趟遍历就统计出规模的 O(N)算法。

C++语言程序代码示例:

```
#include <cstdio>
#include <algorithm>
using std::sort;

struct sNode
{
    int index;
    int leader;
    int level;
    bool operator<(sNode const &t){return level<t.level;}
   //重载了<运算符，只对 level 值做排序
}*mem;

int main(void)
{
```

```
    int *scale;
    int i,N;
    scanf("%d",&N);

    scale=new int [N+1]; //position [0] is invalidity, drop it.
    mem=new sNode [N+1]; // store the value from [1] to [N]

    for(i=1;i<=N;i++){
        scale[i]=0;
        mem[i].index=i;
        scanf("%d",&mem[i].leader);
        mem[i].level=mem[mem[i].leader].level+1;
    }
    sort(mem+1,mem+N+1);

    for(i=N;i>0;i--){
        scale[mem[i].index]+=1;
        scale[mem[i].leader]+=scale[mem[i].index];
    }

    for(i=1;i<=N;i++){
        if(i>1)
            printf(" ");
        printf("%d",scale[i]);
    }
    return 0;
}
```

Python 语言实现时就用 tuple 保存(index,leader,level)数据,所有数据放于一个 tuple 的 list 当中，排序时对 level 进行，遍历整个 list，就可以完成任务。只是用 python 内置的排序不能完成任务，所以排序函数需要自己设计，程序代码如下

```
def mySort(array):
    less = []
    equal = []
    greater = []
    if len(array)>1:
        pivot = array[0][2]       # array[0] is a tuple,
        for x in array:           # x is a tuple, tuple[2] is ‘level’ data
            if x[2] < pivot:
                less.append(x)
            if x[2] == pivot:
```

```
                equal.append(x)
            if x[2] > pivot:
                greater.append(x)
        return mySort(less)+equal+mySort(greater)
    else:
        return array

N=int(input())
memberList=list()                #make a new empty list
memberList.append((0,0,0))       #position [0] is an invalid value
inputList=list(map(int,input().split(' ')))     #read and store input value
for i in range(1,N+1):
    leader=inputList[i-1]
    level=memberList[leader][2]+1
    memberList.append((i,leader,level))
mySort(memberList)

scale=[0]
for i in range(N):
    scale.append(0)       # initialise the array, set all the value is 0
for i in range(N,0,-1):
    scale[memberList[i][0]]+=1
    scale[memberList[i][1]]+=scale[memberList[i][0]]

for i in range(1,N+1):
    if(i>1):
        print(' ',end='')
    print(scale[i],end='')
print()                    #output a new empty line (not necessary)
```

Problem E: Snake Matrix

所谓“蛇形矩阵”，是指从左上角第 1 个格子开始首先向右，到达边界时按顺时针的方向由外圈向内逐一填充的方形矩阵。给出一个数字 N，要将 1 至N^2填入一个 N 行 N 列的蛇形矩阵。

例如当 N=4 时，蛇形矩阵为

```
 1  2  3  4
12 13 14  5
11 16 15  6
10  9  8  7
```

当 N=5 时，蛇形矩阵为

```
 1  2  3  4  5
16 17 18 19  6
15 24 25 20  7
14 23 22 21  8
13 12 11 10  9
```

现在我想知道每一行的蛇形矩阵的和，希望你能通过编写程序求解。

输入：

1 行，1 个数字 N （1<N<2×10^5）

输出：

N 行，第 i 行表示蛇形矩阵第 i 行的总和。

数据要求 N 会达到2×10^5，可能开不出那么大的数组，不妨先打个小数据的矩阵看一看：

```
#include <stdio.h>
#define DEBUG 1
#define MAXN 100
// if MAXN reach the upperbound 2e+5, it must beyond the memory limit

int N; //the order of Matrix

int valid(int x,int y) //if (x,y) in the range of matrix, return true
{
    if(x>=0 && x<N & y>=0 && y<N)
        return 1;
    else
        return 0;
}

typedef long long resType;
resType accumulate(resType *a)// sum a certain line of matrix
{
```

```
    int i;
    resType s=0;
    for (i=0;i<N;i++)
        s+=a[i];
    return s;
}

int main(void)
{

    int i,k,x,y;

    resType a[MAXN][MAXN];

    for(x=0;x<MAXN;x++)
        for(y=0;y<MAXN;y++)
            a[x][y]=0;

    scanf("%d",&N);

    int dir[4][2]={{0,1},{0,-1},{1,0},{-1,0}};
    a[0][0]=1;

    k=0;x=0;y=0;
    for(i=2;i<=N*N;i++)
    {
    //if (x+dx,y+dy)out of range or a[x+dx][y+dy] is not 0, try next direction
        while(!(valid(x+dir[k][0],y+dir[k][1]) && a[x+dir[k][0]][y+dir[k][1]]==0))
            k=(k+1)%4;
// (x+dx,y+dy)in range of matrix,and a[x+dx][y+dy]==0, then assign value i to it.
        x=x+dir[k][0];
        y=y+dir[k][1];
        a[x][y]=i;
    }

#if DEBUG
// output the matrix
    for(x=0;x<N;x++){
        for(y=0;y<N;y++)
            printf("%4d",a[x][y]);
        printf("\n");
    }
#endif
```

```
    for(i=0;i<N;i++)
        printf("%lld\n",accumulate(a[i]));

    return 0;
}
```

结果的正确性显然，因为这个程序真的把蛇形矩阵生成了，并逐一去计算每一行的和。不过在 N 比较大的时候，程序运行的时间和内存均会超出系统可接受的范围，既不可能开的出那么大的二维数组，也不可能快速对那么大的数组逐行求和而不超时。解到这里最多能拿到本题 50%的分数。如开篇所说，应当在这种时候去试着从小规模上寻找数学规律。若能找出数学规律，此问题的时间复杂度最好可以降到 O(N)，空间复杂度仅为 O(1)。

从第一行算起，第 1 行一定是从 1 开始到 N 的和。先将这行的 accumulate()累加函数的时间 O(N)优化成 O(1)：用一步运算 N*(N+1)/2 代替 accumulate()。观察第 2 行，除了最末一个数是位于 N 之下的 N+1 之外，第 2 行另外的 N-1 个数与第 1 行对应位置的差恰好是一个矩阵最外圈（从 1 到 4(N-1)，即第 2 行的每个数都恰好比第 1 行对应位置多 4(N-1)-1 ）这样，第 2 行与第 1 行相比，有 N-1 个元素多 4*(N-1)-1，有 1 个元素多 1，即第 2 行的总和就是第一行的和再加上(N-1)*(4*(N-1)-1)+1*1。到第 3 行，与第 2 行相比，右侧有 2 个元素是与上一行相差 1，左侧 N-2 个元素与上行相差一个次外圈，同样可以推导出它与第 2 行的差。不难看出，相邻两行之和的差值是呈某个规律递减的，收集这些差值构造一个数列，会发现(不是那么容易发现)它是一个二阶等差数列，也就是数列相邻项的差是等差数列，而等差数列的公差是常数 8，这对所有的 N 都成立。这是非常激动人心的性质，这就不需要开出 N^2 大小的矩阵，只需要根据 N 计算出二阶等差数列的首项和每个邻项的差，就能在 O(N)的时间内输出二阶等差数列，也就是得到了答案。

别高兴的太早，在遇到含 N^2 那个数的一行之前还比较顺利，但是蛇形矩阵到了这一行之后，数列变成递减，越往下每行之和会越小，而这后半程每一行的和的规律将因 N 的奇偶性不同而有差异。幸好，这只是另一个新的以 8 为二阶公差的二阶等差数列，有了前边的经验，再推导出后半边在 N 为奇数和偶数的情况就完成任务了。

C 与 C++实现时，应注意输出数据的数据量超出了 int 范围，相关数据要注意使用 long long 类型，注意数据保存与输出格式。数据溢出会导致错误的输出一些负数，而往往是在 N 比较大的时候，输出行数也比较多，这时就可以用输出重定向，把结果输出到本地的文本文件里去检查是否有负数被错误的输出。

```c
#include <stdio.h>
int main(void)
{
    int i,N;
    long long spl,currD; //spl is sum per line, currD is current difference(1 order)
    const int deepD=8; // deeper difference (2 order difference).
    scanf("%d",&N);

    spl=N*(N+1)/2;
    currD=4*(N-1)-1;
    printf("%lld\n",spl);
    for(i=N-1;i>0;i-=2){
        spl+=currD*i+1;
        printf("%lld\n",spl);
        currD-=deepD;
    }
    currD=currD+deepD-10;

    if(N%2==1){
        for(i=1;i<N;i+=2){
            spl+=currD*i;
            printf("%lld\n", spl);
            currD-=deepD;
        }
    }
    else{
        for(i=2;i<N;i+=2){
            spl+=currD*i;
            printf("%lld\n", spl);
            currD-=deepD;
        }
    }

    return 0;
}
```

Python 3 和 C 语言程序实现的思路是一样的，如下所示:

```python
N=int(input())
spl=int(N*(N+1)/2)  #sum per line
currD=4*(N-1)-1     #current difference,1 order difference
deepD=8             #2 order difference
print(spl)          #sum of the first line
```

```
for i in range(N-1,0,-2):
    spl+=currD*i+1
    print(spl)
    currD-=deepD
currD=currD+deepD-10

if(N%2==1):
    for i in range(1,N,2):
        spl+=currD*i
        print(spl)
        currD-=deepD
else:
    for i in range(2,N,2):
        spl+=currD*i
        print(spl)
        currD-=deepD
```

后记

这份文档从 4 月开始计划开篇，到今天写完大概用去 3 个月，这当中做自己的各种事情占去许多时间，断断续续的才把这本报告写完。不过从程序的完善到语言的组织，还是颇费些功夫的。在第一部分中写到一些我个人在编程时的经验和技术，不敢说是指导，只是和大家分享和探讨我的一点经验。对于 18 年机试的 5 道题，因为 OJ 上没有这些题，只得按照考试时我 AC 的思路再编写代码并做些解释，你可以用它做数据的输出测试，用以比较自己的程序是否正确。

华罗庚说“勤能补拙是良剂，一分辛苦一分才”[1]。要做好上机考试，足量的训练是必不可少的。初学之时，PTA (https://pintia.cn/problem-sets)上的“基础编程题目集”与“数据结构与算法题目集（中文）”都是非常好的，它的测试点会有提示，看你对程序测试中出现的各种情况是否能考虑周全。PAT-B 级题目难度也不大，大部分题目的数据要求不高，有时用 $O(N^2)$的算法都能过，并且 PAT 评分标准也是按测试点给分，对于学习编程的初期是非常有用的。而在 Atcoder (https://beta.atcoder.jp/)会一段时间进行一次比赛，分为 Beginner, regular, grand 三个级别，初学者可以做 Beginner 级的题目。Atcoder 的判题要求所有测试点全通过才得分。比赛结束后判题不再计分，并会给出题解，所有人都可以任意的提交，还可以看到他人提交的程序代码，学习他人优秀的编程风格和经验。学到进阶程度之时可以去北大 ACM 训练平台”百练”(http://bailian.openjudge.cn/)和华东师大的 EOJ (https://acm.ecnu.edu.cn/)，它们都可以根据题目类型的标签筛选题目，以针对某一类的题做专项训练。 EOJ 上还会根据过题人数的多少而给不同数额的奖励分，可以依据此初步判断题目的难度。要在 OJ 上训练，必须首先在 OJ 上注册账号，而使用 Virtual Judge (https://cn.vjudge.net/)就不用在每个 OJ 平台都注册一遍，它将许多知名的 OJ 通过爬虫将题目数据抓取出来，还能够随时进行自定义比赛，或者邀请伙伴们一起比赛。

市面上关于机试的参考书目也相当多。王道系列《计算机考研：机试指南》内容非常详细，可惜针对此书的 OJ 已经关闭，王道论坛官方便将此书 pdf 发布出来供大家下载(http://www.cskaoyan.com/thread-647811-1-1.html)。不过，这类基础算法和上机考试书的内容都大同小异，胡凡、曾磊主编的《算法笔记》以 PAT 甲级和乙级的题目为实例，同样详细地讲解了各类基础算法。书上还提到 C++ STL 的简单使用方法，如果你只会用基本的 C 语言，没有接触过 C++，那这本书可能会非常适合你，学着利用 C++的 STL 提高做题效率，许多函数能省去自己编写的麻烦。百练 OJ 也有相应的参考书《算法基础与在线实践》。还有诸如《挑战程序设计竞赛》《算法竞赛入门经典》等，当中有些内容的难度已超过考研机试的水平，凡所种种不再列举。大家可以根据自己情况进行相应书目的参考和阅读。

[1] 摘自《下棋找高手》,华罗庚, 128-136 页《在困境中要发愤求进》。文章原载于 1981 年 6 月 11 日《浙江日报》