23级程序设计基础练习题解

1. 判断可逆素数

思路分析

本题在实现思路上没有任何难点，我们只需要实现这两个功能：

1. 给定一个整数，判断其是否为素数；
2. 给定一个整数，返回其颠倒之后的数。

具体实现过程

根据上面的思路分析，很容易确定程序主题框架：

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

int isPrime(int num);  // 判断是否为素数
int reverseInt(int num);  // 翻转整数

int main()
{
    int n;
    scanf("%d", &n);
    n = n > 0 ? n : -n;  // 正数化

    if (isPrime(n) && isPrime(reverseInt(n))) printf("yes");
    else printf("no");

    return 0;
}
```

其中`isPrime`函数用于判断输入的整数是否为素数，是则返回1，否则返回0；`reverseInt`用于将输入的整数颠倒顺序后输出。

值得注意的是，为了方便两函数的实现，我们在主函数中将输入的整数正数化——在判断素数的问题上正负号无关紧要。（一些简单的`if-else`逻辑可以简化为三目运算符）

接下来实现`isPrime`函数。判断素数有很多种算法，这里就采用最朴素的想法（ds这门课的**常规作业**不考虑效率）：从2开始遍历到$sqrt(n)$，如果都没有整除n，则其就是素数（一个整数的因子不会大于其平方根）：

```
int isPrime(int num) {
    for (int i = 2; i < sqrt(num); i++) {
        if (num % i == 0) return 0;
    }
    return 1;  // 全都没有整除，返回1
}
```

对于`reverse`函数，只要清楚`n % 10`表示n的个位数，`n /= 10`相当于把n的个位数字去除掉，就问题不大。

```
int reverseInt(int n) {
    int res = 0;  // 记录翻转的结果
    while (n) {
        int tmp = n % 10;  // tmp表示当前数的最后一位
        n /= 10;
        res = res * 10 + tmp;
    }
    return res;
}
```

参考代码

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

int isPrime(int num);  // 判断是否为素数
int reverseInt(int num);  // 翻转整数

int main()
{
    int n;
    scanf("%d", &n);
    n = n > 0 ? n : -n;  // 正数化

    if (isPrime(n) && isPrime(reverseInt(n))) printf("yes");
    else printf("no");

    return 0;
}


int isPrime(int num) {
    for (int i = 2; i < sqrt(num); i++) {
        if (num % i == 0) return 0;
    }
    return 1;  // 全都没有整除，返回1
```

```
}


int reverseInt(int n) {
    int res = 0;  // 记录翻转的结果
    while (n) {
        int tmp = n % 10;  // tmp表示当前数的最后一位
        n /= 10;
        res = res * 10 + tmp;
    }
    return res;
}
```

2. 矩形相交

思路分析

本题的题面已经给我们提示了要去计算x轴和y轴上的交集。如果两轴上都有交集，显然计算乘积，否则直接输出0即可。

具体实现过程

首先在主函数中读入数据，并按照题面提示计算两轴上的交集：

```
// 主函数中
    int Ax1, Ay1, Ax2, Ay2, Bx1, By1, Bx2, By2;
    scanf("%d%d%d%d%d%d%d%d", &Ax1, &Ay1, &Ax2, &Ay2, &Bx1, &By1,
&Bx2, &By2);

    int delta_x = min(max(Ax1, Ax2), max(Bx1, Bx2)) - max(min(Ax1,
Ax2), min(Bx1, Bx2));  // x轴上交集
    int delta_y = min(max(Ay1, Ay2), max(By1, By2)) - max(min(Ay1,
Ay2), min(By1, By2));  // y轴上交集
```

其中几个函数可以直接自己写出来（这种非常简单的函数，一行就写完了，不建议用`math.h`中声明好的函数，原因的话，大家可以想想自己是否清楚`asb`和`fabs`函数的区别）：

```
int min(int a, int b) { return a < b ? a : b; }
int max(int a, int b) { return a > b ? a : b; }
int abs(int n) { return n > 0 ? n : 0 - n; }
```

然后进行判断输出：

```
// 主函数中
    if (delta_x < 0 || delta_y < 0) printf("%d", 0);
    else printf("%d", delta_x * delta_y);
```

参考代码

```
#include <stdio.h>

int min(int a, int b) { return a < b ? a : b; }
int max(int a, int b) { return a > b ? a : b; }
int abs(int n) { return n > 0 ? n : 0 - n; }


int main()
{
    int Ax1, Ay1, Ax2, Ay2, Bx1, By1, Bx2, By2;
    scanf("%d%d%d%d%d%d%d%d", &Ax1, &Ay1, &Ax2, &Ay2, &Bx1, &By1,
&Bx2, &By2);

    int delta_x = min(max(Ax1, Ax2), max(Bx1, Bx2)) - max(min(Ax1,
Ax2), min(Bx1, Bx2));  // x轴上交集
    int delta_y = min(max(Ay1, Ay2), max(By1, By2)) - max(min(Ay1,
Ay2), min(By1, By2));  // y轴上交集

    if (delta_x < 0 || delta_y < 0) printf("%d", 0);
    else printf("%d", delta_x * delta_y);

    return 0;
}
```

3. 求差集

思路分析

本题要求最后的输出顺序和A的输入顺序一致，就不用考虑先排序再双指针等想法了，直接暴力求解就可以，我们要做的有这些事情：

1. 读入A的数，并存入数组；
2. 逐个读取B，并在A中寻找是否有相同的数字，如果有，在A中进行标记以区分；
3. 按照顺序打印出A中未被标记的数。

具体实现过程

首先初始化集合，并把A读入集合：

```
// 主函数中
    int set[1005] = {0};
    int tmp;
    int n = 0;

    for (n = 0; ; n++) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        set[n] = tmp;
    }
```

然后读入集合B的值，并判断集合中是否已经有该值，如果有，我们将该位置标记为-1（因为题目说读入的都是自然数，不会有-1）：

```
// 主函数中
    while (1) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        for (int i = 0; i < n; i++) {
            if (set[i] == tmp) set[i] = -1;  // 如果集合中已经存在该数，
标记为-1
        }
    }
```

接下来对`set`输出即可，注意要判断其值是否被标记过（ == -1）：

```
// 主函数中
    for (int i = 0; i < n; i++) {
        if (set[i] != -1) printf("%d ", set[i]);
    }
```

参考代码

```
#include <stdio.h>

int main()
{
    int set[1005] = {0};  // 存放结果的集合
    int tmp;
    int n = 0;

    for (n = 0; ; n++) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        set[n] = tmp;
    }
```

```
    while (1) {
        scanf("%d", &tmp);
        if (tmp == -1) break;
        for (int i = 0; i < n; i++) {
            if (set[i] == tmp) set[i] = -1;  // 如果集合中已经存在该数，
标记为-1
        }
    }

    for (int i = 0; i < n; i++) {
        if (set[i] != -1) printf("%d ", set[i]);
    }

    return 0;
}
```

4. 矩阵运算

思路分析

本题在思路上并没有什么障碍，我们只需做如下步骤：

1. 定义一个二维数组，表示矩阵，并把第一个矩阵的值读到数组里；
2. 不断地读入操作符与矩阵，直到读到的操作符为#
3. 每次读矩阵的过程中，根据操作符的不同，更新数组里的值。

需要注意的是：本题中从始至终我们只需要定义一个二维数组，而不用每次读一个矩阵就定义一个，因为我们可以在读的过程中去更新数值（大家在下面的过程中体会）：

具体实现过程

首先是初始化数组，并读入第一个矩阵：

```
// 主函数中
    int n;
    char op;  // 操作符
    int matrix[10][10] = {{0}};  // 矩阵

    scanf("%d", &n);

    for (int i = 0; i < n; i++) {  // 读矩阵
        for (int j = 0; j < n; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
```

然后我们要循环读入一个操作符，直到读入的操作符为#。

在循环体中，我们继续读取矩阵的每一个元素——在读入之后可以立即更新数组的值（如果有矩阵乘法，那这样就行不通了，那样我们只能把该矩阵整体读到另一个数组后再处理）：

```
// 主函数中
    while (scanf(" %c", &op) != EOF && op != '#') {  // 注意读字符前吞掉空格
        int tmp;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                scanf("%d", &tmp);
                // 直接根据操作符更新数组
                if (op == '+') matrix[i][j] += tmp;
                else matrix[i][j] -= tmp;
            }
        }
    }
```

最后对数组进行输出即可：

```
// 主函数中
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%5d", matrix[i][j]);
        }
        printf("\n");
    }
```

参考代码

```
#include <stdio.h>

int main()
{
    int n;
    char op;  // 操作符
    int matrix[10][10] = {{0}};  // 矩阵

    scanf("%d", &n);

    for (int i = 0; i < n; i++) {  // 读矩阵
        for (int j = 0; j < n; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
```

```
    while (scanf(" %c", &op) != EOF && op != '#') {  // 注意读字符前吞掉空格
        int tmp;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                scanf("%d", &tmp);
                // 直接根据操作符更新数组
                if (op == '+') matrix[i][j] += tmp;
                else matrix[i][j] -= tmp;
            }
        }
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%5d", matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

5. 文件拷贝

思路分析——双指针法

本题披上了文件操作的外套，如果我们去掉这层外套，可以把题面改写如下：给定一个字符串，将其中连续的多个空白字符替换为一个空格，并输出新的字符串。

这道题的想法很简单，却十分考验大家对代码的流程控制，以及算法功底，相信很多同学写着写着就发现代码写得“又长又烂”，逻辑十分混乱。

我这里介绍一种双指针的处理思路，它的时间复杂度是O(n)，比大家的暴力解法要快很多，逻辑也更清晰。

如果开头的字符不是空白字符（先考虑这种情况，到时候一行代码特判一下空白字符开头的情况就可以），我们可以把输入的字符串看成这样（“若干”的意思是大于等于0）：

第0个非空白字符 若干空白字符 第一个非空白字符 若干空白字符 。。。。。。

可以发现，**除了最开始（第0个）非空白字符外，其他的空白字符一定出现在非空白字符之前**。这里假设我们要处理的字符串变量为`buf`，我们的目的是去除掉`buf`中连续的空白字符。

我们可以定义两个整数`slow`和`fast`，代表遍历的字符数组的下标（也成为快慢“指针”——指向数组位置的针，不是真的指针变量），初始化时都为0.

接下来让`fast`指针遍历`buf`，当该位置上的字符是空白字符时，我们不去管它，只有当遇到了非空白字符时，我们才进行处理——给`slow`所指向的位置添加一个空格，然后`slow`向后移动一位，接下来`slow`和`fast`一起向前移动，并给`slow`所指向的位置赋值，直到遇到了下一个空白字符。最后重置一下`buf`的长度即可。

同时，由于字符串开头还有可能是空白字符，所以需要特判一下。

大家看这份文字版的题解可能觉得难以理解，这里我先给出完整代码：

参考代码

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

int main()
{
    char buf[1024] = {0};
    FILE *in = fopen("fcopy.in", "r");
    FILE *out = fopen("fcopy.out", "w");

    while (fgets(buf, sizeof(buf), in) != NULL) {  // 一行一行读文件
        int fast = 0, slow = 0;  // 用双指针法去除空格
        int len = strlen(buf);
        for (fast = 0; fast < len; fast++) {  // 快指针遍历数组
            if (buf[fast] != ' ' && buf[fast] != '\t') {  // 遇到非空
白字符再处理
                if (!(slow == 0 && fast == 0)) buf[slow++] = ' ';  //
特判文件开头就是空格的情况
                while (fast < len && buf[fast] != ' ' && buf[fast] !=
'\t') {  // 注意此处的约束条件
                    buf[slow++] = buf[fast++];
                }
            }
            if (fast == len - 1 && buf[fast] == ' ' && buf[fast] !=
'\t') buf[slow++] = ' ';  // 特判文件尾是空格
        }
        buf[slow] = '\0';  // 重置字符串长度
        fprintf(out, "%s", buf);
    }

    return 0;
}
```

其实这个想法最难以理解的地方是**只有当我们遇到非空白字符时才处理**，也就是每次都是读到了连续空白字符的最后一个才处理——可能大家常规的想法是读到连续空白字符的第一个时就处理。

其次，这个算法的所有操作都是在`buf`这一个字符串上操作的，我们并没有额外开辟新的存储空间。

有的同学可能有疑问：这不是两层循环吗？但其实数组的每个元素只会被遍历到1次或2次，所以时间复杂阿德是O(n)。

无论如何，文字版的讲解总会显得不那么直观，如果大家有问题，不妨自己模拟一下操作流程，或者和同学或助教一起讨论一下。同时，建议大家理解思路之后自己再手动敲一遍这道题——这是很考验大家流程控制的一个算法。

双指针举例

ds课程中的常规作业不对代码效率做要求，所以大家如果觉得双指针太难理解，也可以适当缓一缓。

双指针是在线性表（数组，还有后面学到的链表等）上非常实用的一种处理思路，大家上网上搜算法题时也能搜到很多双指针的妙用，这里我举一个非常简单也很经典的例子：

给一单链表（链表可以理解成数组，但是其中的每一个元素只能通过它的前一个元素去找到，比如要找到下标为3的结点，只能从下标为0的头结点开始，先找到下标为1的结点，再找到下标为2的结点，再找到3的结点），在不求出链表长度的情况下（显式或隐式都不可以），返回其倒数第n个结点，如没有，返回NULL。

正常的想法肯定是从头开始遍历到尾，记录出链表长度，然后再从头开始遍历相应的次数就可以，但是题目规定了不可以求出长度。

双指针解法如下：

1. 定义快慢指针`fast = 头结点; slow = 头结点`；
2. 先让`fast`向后移动n次，如果在这过程中超过了链表的最后一个结点（最后一个结点的下一个是NULL），就返回NULL；
3. 然后两指针一起向后移动，直到`fast`到达链表末尾，此时`slow`所指向的结点就是倒数第n个。

这个例子大家可以在学完链表后再来看一看，如果大家对双指针有兴趣，也可以自行去网上搜索相应题目。

6. 求两组整数的异或集

思路1——双指针应用

本题求集合的异或其实和第三题：求差集并没有太多的区别，思路上也是想通的，大家不妨按着第三题的思路先自己做一下本题。（在本文中也不会再给出这种朴素思路的详细分析）

但是，本题的输出要求和第三题不同：本题要求输出的整数按照从大到小排列，结合我在第五题讲的双指针思路，大家先思考一下自己能否想出双指针的解法。

如果采用双指针，我们可以把本题拆成这几个业务逻辑：

1. 读入两个集合，分别存储在两数组`a`，`b`内；
2. **分别**对两数组按照从小到大排序；
3. 设置两个“指针”（就是下标的意思）`i`和`j`，从两数组头部分别开始移动：如果当前位置`a[i] < b[j]`，则输出`a[i]`并让`i++`；若`b[j] < a[i]`则输出`b[j]`并令`j++`；若此时两数组相应位置上的数相等，则表明这个元素在两数组中均出现过，我们什么也不输出，直接令`i++; j++`即可。直到`i`或`j`有一个到达了数组末尾；
4. 对没有到达末尾的那个数组，从当前位置开始输出到末尾。

大家可以自己手动模拟一下这个过程，其精髓在于我们先对a和b排序，使之已经有序了。

具体实现过程

首先声明数组，并读入数据。这里要注意：两组数以换行符分隔，要特判一下：

```
// 主函数中
    int a[25] = {0};
    int b[25] = {0};

    int n1 = 0, n2 = 0;
    while(scanf("%d", &a[n1++]) != EOF) {
        // 不断读到第一个数组，直到遇到换行符终止
        if (getchar() == '\n') {
            break;
        }
    }
```

然后对两数组排序（`cmp`为自己设计的排序比较函数）：

```
    // 先对两数组排序
    qsort(a, n1, sizeof(int), cmp);
    qsort(b, n2, sizeof(int), cmp);
```

其中`cmp`函数实现如下：

```
int cmp(const void *a, const void *b) { return (*(int *)b) - (*(int
*)a); }
```

对`qsort`使用不熟练的同学一定要好好复习一下，包括最简单的，对基本数据类型的排序，以及对结构体的二级排序等，这个函数在本课程中**非常重要**！一般而言期末考试的第一题就是用`qsort`进行排序的。

然后，根据我们上面说的双指针法进行遍历，输出：

```
    // 双指针法输出
    int i = 0, j = 0;
    while (i < n1 && j < n2) {
        if (a[i] > b[j]) printf("%d ", a[i++]);
        else if (a[i] < b[j]) printf("%d ", b[j++]);
        else {
            i++; j++;
        }
    }
```

最后判断一下`i`和`j`哪个还没有遍历到相应数组的末尾，对其进行输出：

```
    if (i <= n1 - 1) {  // i 没有遍历到末尾
        while (i < n1) printf("%d ", a[i++]);
    } else {  // j 没有遍历到末尾
        while (j < n2) printf("%d ", b[j++]);
    }
```

参考代码

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

int cmp(const void *a, const void *b) { return (*(int *)b) - (*(int
*)a); }

int main()
{
    int a[25] = {0};
    int b[25] = {0};

    int n1 = 0, n2 = 0;
    while(scanf("%d", &a[n1++]) != EOF) {
        // 不断读到第一个数组，直到遇到换行符终止
        if (getchar() == '\n') {
            break;
        }
    }

    while (scanf("%d", &b[n2]) != EOF) n2++;  // 读第二个数组

    // 先对两数组排序
    qsort(a, n1, sizeof(int), cmp);
    qsort(b, n2, sizeof(int), cmp);

    // 双指针法输出
```

```
    int i = 0, j = 0;
    while (i < n1 && j < n2) {
        if (a[i] > b[j]) printf("%d ", a[i++]);
        else if (a[i] < b[j]) printf("%d ", b[j++]);
        else {
            i++; j++;
        }
    }
    if (i <= n1 - 1) {  // i 没有遍历到末尾
        while (i < n1) printf("%d ", a[i++]);
    } else {  // j 没有遍历到末尾
        while (j < n2) printf("%d ", b[j++]);
    }

    return 0;
}
```

思路2——结构体排序

本题限制了每组整数中的元素不重复，也就是说对于两组中的所有数，在整体中出现的次数要么是1（在一组中出现），要么是2（在两组中均出现），而我们的目的其实就是输出只出现了一次的数而已。

所以，我们根本没有必要把题目描述的两组数真的看成是两组数！我们只要一直读数，然后记录出每个数字出现的次数就可以。

为此，我们可以设计一个结构体，设置两个属性：其中`num`域记录其真实的数据，`cnt`记录其出现的次数。

我们可以把思路归结如下：

1. 定义结构体类型`element`，并创建结构体数组；
2. 不断读取整数，并判断该数是否出现过，若是则给相应的结点`cnt`域加一，否则创建节点并放到数组里；
3. 对数组排序；
4. 遍历数组，输出`cnt == 1`的数。

具体实现过程

首先定义元素结构体，并创建数组：

```
typedef struct {
    int num;
    int cnt;
} element;

element set[50];  // 集合
int setNum = 0;  // 集合元素个数
```

然后在主函数中不断读取数据，并根据上面的逻辑进行存储：

```
// 主函数中
    while (scanf("%d", &tmp) != EOF) {
        int flag = 0;  // 用于记录该数是否已经在set中出现过
        int index;  // 如果出现过，记录出现的下标
        for (int i = 0; i < setNum; i++) {
            if (set[i].num == tmp) {
                // 找到了
                flag = 1;
                index = i;
                break;
            }
        }


        if (flag) {
            // 集合中已经有该数
            set[index].cnt++;
        } else {
            // 集合中尚没有该数
            set[setNum].num = tmp;
            set[setNum].cnt = 1;
            setNum++;
        }
    }
```

注意我们这里寻找集合中是否已经出现过该数的方法是从头到尾去暴力遍历，时间复杂度是`O(n^2)`，本题最多只有40个数，还可以忍受。如果想要优化，可以等大家学到哈希表之后再回来思考。

接下来要对结构体排序，为此设计排序函数：

```
int cmp(const void *a, const void *b) {
    element *e1 = (element *)a;
    element *e2 = (element *)b;
    return e2->num - e1->num;
}
```

然后在主函数中进行排序：

```
// 主函数中
    qsort(set, setNum, sizeof(element), cmp);
```

最后再遍历数组，对`cnt == 1`的数据进行输出即可：

```
// 主函数中
    for (int i = 0; i < setNum; i++) {
        if (set[i].cnt == 1) printf("%d ", set[i].num);
    }
```

参考代码

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

typedef struct {
    int num;
    int cnt;
} element;

element set[50];  // 集合元素个数
int setNum = 0;

int cmp(const void *a, const void *b);

int main()
{
    int tmp;  // 临时变量，用于暂存读取的数据
    while (scanf("%d", &tmp) != EOF) {
        int flag = 0;  // 用于记录该数是否已经在set中出现过
        int index;  // 如果出现过，记录出现的下标
        for (int i = 0; i < setNum; i++) {
            if (set[i].num == tmp) {
                // 找到了
                flag = 1;
                index = i;
                break;
            }
        }


        if (flag) {
            // 集合中已经有该数
            set[index].cnt++;
        } else {
```

```
            // 集合中尚没有该数
            set[setNum].num = tmp;
            set[setNum].cnt = 1;
            setNum++;
        }
    }

    qsort(set, setNum, sizeof(element), cmp);

    for (int i = 0; i < setNum; i++) {
        if (set[i].cnt == 1) printf("%d ", set[i].num);
    }

    return 0;
}

int cmp(const void *a, const void *b) {
    element *e1 = (element *)a;
    element *e2 = (element *)b;
    return e2->num - e1->num;
}
```

7. 凸多边形面积

思路分析

本题提示中提示了计算三角形的面积，题面也明确了是凸四边形，大家很容易想到我们要把多边形分解成多个三角形去计算。具体的方法大家可以自己画一个凸四边形，然后选定一个顶点，把这个顶点和其他顶点分别相连，就得到了多个三角形，我们分别计算这些三角形的面积即可。

我们可以把思路归结成如下步骤：

1. 先读入两个点的坐标，其中一个点作为定点不动；
2. 每次再读入一个点的坐标，计算当前三角形的面积，然后把第二个点的坐标置成当前点的坐标，不断循环。

具体实现过程

本题的主函数框架非常简单，可以根据上面的思路分析得到：

```
int main()
{
    int n;
    double x1, y1, x2, y2, x3, y3;
    double res = 0;  // 结果
    scanf("%d", &n);
```

```
    // 先读入两个点的坐标，其中第一个点为定点
    scanf("%lf%lf%lf%lf", &x1, &y1, &x2, &y2);

    for (int i = 2; i < n; i++) {
        scanf("%lf%lf", &x3, &y3);
        // 计算当前三角形面积，并加到
        res += triangle_area(x1, y1, x2, y2, x3, y3);
        // 更新第二个点的左坐标
        x2 = x3;
        y2 = y3;
    }

    printf("%.2lf", res);

    return 0;
}
```

其中`triangle_area`函数接受三个点的坐标，返回三角形面积：

```
double triangle_area(double x1, double y1, double x2, double y2,
double x3, double y3) {
    double a = distance(x1, y1, x2, y2);
    double b = distance(x2, y2, x3, y3);
    double c = distance(x3, y3, x1, y1);
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
```

其中`distance`函数计算两点间距离：

```
double distance(double x1, double y1, double x2, double y2) {
    return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}
```

参考代码

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

double distance(double x1, double y1, double x2, double y2);  // 计算
两点之间的距离
double triangle_area(double x1, double y1, double x2, double y2,
double x3, double y3);  // 使用海伦公式计算三角形面积


int main()
{
```

```
    int n;
    double x1, y1, x2, y2, x3, y3;
    double res = 0;  // 结果
    scanf("%d", &n);
    // 先读入两个点的坐标，其中第一个点为定点
    scanf("%lf%lf%lf%lf", &x1, &y1, &x2, &y2);

    for (int i = 2; i < n; i++) {
        scanf("%lf%lf", &x3, &y3);
        // 计算当前三角形面积，并加到
        res += triangle_area(x1, y1, x2, y2, x3, y3);
        // 更新第二个点的左坐标
        x2 = x3;
        y2 = y3;
    }

    printf("%.2lf", res);

    return 0;
}


double distance(double x1, double y1, double x2, double y2) {
    return sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}


double triangle_area(double x1, double y1, double x2, double y2,
double x3, double y3) {
    double a = distance(x1, y1, x2, y2);
    double b = distance(x2, y2, x3, y3);
    double c = distance(x3, y3, x1, y1);
    double s = (a + b + c) / 2;
    return sqrt(s * (s - a) * (s - b) * (s - c));
}
```

8. 整数的N进制字符串表示

思路分析

其实本题就是进制转化，如果是把十进制转化成二进制大家肯定都会——不断地除以二，把余数存储起来就可以。

关键的问题是进制大于10的情况，比如11是用字母'b'表示的，要是在程序中加上一堆`if - else`当然可行，但是可读性大打折扣，为此我们可以设计一个映射表来映射出对应的值。

具体实现过程

主函数非常简单，直接就可以写出来：

```
int main()
{
    int n, b;
    char s[105] = {0};
    scanf("%d%d", &n, &b);
    itob(n, s, b);
    printf("%s", s);

    return 0;
}
```

关键在于`itob`的设计，我们先顶一个一个映射表：

```
char map[40] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
                'v', 'w', 'x', 'y', 'z'};
```

这样，假如我们想找出11对应的字符，只要用下标索引`map[11]`即可。

在`itob`函数中，我们可以先把正负号去掉，将输入的数正数化，然后逐位取模存储，最后翻转一下记录的字符串，再拼接到正负号前面就可以（说的比较啰嗦，但是道理很简单）：

```
void itob(int n, char *s, int b) {
    // 处理负号
    if (n < 0) s[0] = '-';
    // 正数化
    n = n > 0 ? n : -n;

    char tmp[105] = {0};  // 临时变量,记录序列
    int len = 0;  // 序列长度

    while (n) {
        tmp[len++] = map[n % b];
        n /= b;
    }
    // 翻转字符串
    reverseStr(tmp, len);
    // 拼接正负号
    strcat(s, tmp);
}
```

其中`reverseStr`函数的设计非常简单，不做讲解：

```c
void reverseStr(char *s, int len) {
    int i = 0, j = len - 1;
    int tmp;
    while (i <= j) {
        tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
        i++; j--;
    }
}
```

参考代码

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

char map[40] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
                'v', 'w', 'x', 'y', 'z'};

void reverseStr(char *s, int len);
void itob(int n, char *s, int b);


int main()
{
    int n, b;
    char s[105] = {0};
    scanf("%d%d", &n, &b);
    itob(n, s, b);
    printf("%s", s);

    return 0;
}


void reverseStr(char *s, int len) {
    int i = 0, j = len - 1;
    int tmp;
    while (i <= j) {
        tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
        i++; j--;
    }
}
```

```
void itob(int n, char *s, int b) {
    // 处理负号
    if (n < 0) s[0] = '-';
    // 正数化
    n = n > 0 ? n : -n;

    char tmp[105] = {0};  // 临时变量,记录序列
    int len = 0;  // 序列长度

    while (n) {
        tmp[len++] = map[n % b];
        n /= b;
    }
    // 翻转字符串
    reverseStr(tmp, len);
    // 拼接正负号
    strcat(s, tmp);
}
```

9. 最长升序子串

动态规划原理

本题是一道比较经典的题目，这里介绍一种叫做“动态规划”的算法。但正如题面所说，本题作为选做题其实并不需要大家掌握这种算法的设计技巧，这也不是本门课程的重点，这里写出题解供学有余力的同学参考。

其实动态规划的思想在大家小学学的一些稍有难度的数学题里就有体现，比如下题：

有一个4X3的网格，一个人从左下的位置出发，问到达其他块分别最短需要多少步（假设只能横竖移动）：

0		

大家在高中阶段解决这道题会用排列组合相关知识，但是小学时，我们学过一种更加“无脑”的操作：现在假设从左下的起点开始，标记为0（意为最短需要0步）。下一步可以到达的位置显然最短需要的步数就是1：

1		
0	1	

而接下来，我们可以根据已经“到达”的点继续迭代标记：

2		
1	2	
0	1	2

接下来同理：

3		
2	3	
1	2	3
0	1	2

以此类推，接下来的两步我就不展示了。

这种方法的好处在于：即使所给的网格并不规整（比如本题只有横竖交叉），我们依然可以根据“已经到达的地方”去不断扩展“未知的领域”。大家也能很容易想到：加入当前的位置已经被标记过（也就是到达过），那么当再有一个点到达此处时，我们要比较出已标记和新的路径的长度，取其小的值——这也是“规划”一词的含义。

大家可以在本题中继续领略这一算法的思想。

思路分析

经过上面的铺垫，可以依照相同的思想将本题的思路归纳如下：

1. 开一个`int``型数组，用于记录以此位置为重点的子串中最长的长度；
2. 遍历输入的字符串，并判断在之前走过的路径中可以到达此点的路径最短值（如果之前的都到不了，则记为1——这一点也可以放在第一步数组的初始化时——我后面的代码就会这么去写）——可以想象的到，这就是动态地去更新的过程。
3. 为了记录数组中最大值，应设置一个变量也动态地去记录。

本题的代码十分简单，这里直接给出代码，如果大家对此不是很理解，可以按照上面的步骤自己手动模拟一下。

参考代码

```
#include <stdio.h>

int max(int a, int b) { return a > b ? a : b; }

int main()
{
    char s[10005] = {0};
    int dp[10005] = {1};
    int res = 0;  // 记录结果

    scanf("%s", s);
    for (int i = 1; s[i]; i++) {
        for (int j = 0; j < i; j++) {
            if (s[i] >= s[j]) dp[i] = max(dp[i], dp[j] + 1);
        }
        if (dp[i] > res) res= dp[i]; // 取长的子序列
    }

    printf("%d\n", res);
    return 0;
}
```

扩展思考

大家可以思考一下，如果本题要求找出最长的**连续**的子串，又有什么思路呢？大家可以去搜索一下“**滑动窗口**”算法，看看会不会有启发，欢迎大家将自己的想法分享到群里和大家一起探讨。（第11题就是考察滑动窗口算法）

10. 合并字符串

本题和第六题的双指针思路简直没有任何区别——本题更是直接告诉了大家两串都是有序的，建议大家自己写一遍，然后对照着参考代码看看代码的简洁性与规范性：

参考代码

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

void str_bin(char str1[], char str2[]);
```

```c
int main()
{
    char str1[1024] = {0};
    char str2[1024] = {0};
    scanf("%s%s", str1, str2);
    str_bin(str1, str2);
    printf("%s", str1);

    return 0;
}


void str_bin(char str1[], char str2[]) {
    char tmp[1024] = {0};  // 记录结果的数组
    int i = 0, j = 0;  // 双指针
    while (str1[i] && str2[j])  // 双指针遍历移动
        tmp[i + j] = str1[i] < str2[j] ? str1[i++] : str2[j++];

    // 将没有到头的那一个串的剩余部分输出
    if (str1[i])  while (str1[i]) tmp[i + j] = str1[i++];
    else while (str2[j]) tmp[i + j] = str2[j++];
    strcpy(str1, tmp);
}
```

11. 连续正整数的和

本题就用到了我在第10题留下的悬念——滑动窗口算法。

思路分析

我们可以想象正整数依次排列开来：1 2 3 4 5 ...

我们的目的是寻找连续的正整数，所以其实只要确定两个数——即开始的数和结尾的数——就可以确定这样一个连续的区间。

所以不难想象，可以用两层`for`循环，第一层遍历代表连续区间的末尾，第二层代表区间的起始位置，判断当前区间内的数值是否符合要求，如果有符合要求的数，我们就记录下来，并比较出最长的那一种情况，最后输出即可。

但是这样两层`for`循环显然时间复杂度是`O(n^2)`——这种算法不难实现，建议大家自己敲一遍练练手。

其实利用滑动窗口，我们可以将此算法的时间复杂度优化掉一层，变成`O(n)`。

具体如何实现呢，大家可以想一下：假如当前遍历到的区间是[4, 7]，目标是22，显然得到了目标，而当我们将区间的末尾移动至8时，其实最理想的情况下，如果有满足条件的区间的话，区间开头也要从5开始——这一点是不是有点像一个窗口一直在滑动？

所以，每当我们更新了区间尾时，没有必要从头开始遍历区间开头，只要从上一次区间的头开始继续往后遍历即可。

同时，区间尾的移动虽然是一个一个数移动的，但是头部却不然，因为尾部的数一定比头部的数大很多——我们只要一直维持着区间内部的和小于等于目标值即可（遇到目标值就记录）。

所以，我们只需遍历一层——这一层是代表窗口区间的末尾。

同时，为了记录结果（起始位置，终止位置，窗口长度），我们不妨设置一个结构体（其实没必要，就是想让大家练习一下结构体）。

具体实现过程

声明结果类型结构体：

```
typedef struct {
    int i;
    int j;
    int len;
} result;
```

其中`i`，`j`，`len`分别代表窗口的起始位置，终止位置与长度。

然后在主函数中读入`n`，并定义结果变量，并进行初始化：

```
// 主函数中
    // i代表窗口头，j代表窗口尾
    int i = 1, j = 1;
    int sum = 0;  // 当前窗口内的和
    result res; res.len = 0;
```

然后遍历窗口尾，并且每次都移动窗口头，始终位置窗口内的和小于等于目标值：

```
// 主函数中
    for (j = 1; j < n; j++) {
        sum += j;
        while (sum > n) sum -= (i++);
        if (sum == n && j - i + 1 > res.len) {  // 更新res
            res.i = i;
            res.j = j;
            res.len = j - i + 1;
        }
    }
```

大家可能会想这不是也是两层循环吗，但其实如果单独看每一个数的话，最多被遍历到了两次，所以其实复杂度还是`O(n)`。

最后判断有没有结果(`res.len == 0 ?`)，并进行输出：

```
// 主函数中
    if (res.len == 0) printf("No Answer");
    else {
        printf("%d=", n);
        for (i = res.i; i <= res.j; i++) {
            printf("%d", i);
            if (i != res.j) printf("+");
        }
    }
```

完整代码

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

typedef struct {
    int i;
    int j;
    int len;
} result;

int main()
{
    int n;
    scanf("%d", &n);

    // i代表窗口头，j代表窗口尾
    int i = 1, j = 1;
    int sum = 0;  // 当前窗口内的和
    result res; res.len = 0;

    for (j = 1; j < n; j++) {
        sum += j;
        while (sum > n) sum -= (i++);
        if (sum == n && j - i + 1 > res.len) {  // 更新res
            res.i = i;
            res.j = j;
            res.len = j - i + 1;
        }
    }

    if (res.len == 0) printf("No Answer");
    else {
        printf("%d=", n);
        for (i = res.i; i <= res.j; i++) {
```

```
            printf("%d", i);
            if (i != res.j) printf("+");
        }
    }

    return 0;
}
```

扩展——如何迅速判断有没有解

大家可能发现了，所有的奇数都可以分解为连续的整数和（至少2个，除以二和除以二加一就行），但是偶数呢？

如果大家注意力惊人，会发现2,4,8,16...这些无法分解，其他都可以——但这是数学而不是编程，所以我不做证明，只是强调一点：如果用位运算去判断2,4,8,16...这些2的整数次幂呢？

其实只要判断`n & (n - 1)  == 0 ?`即可，所以本题也可以写成如下版本：

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

typedef struct {
    int i;
    int j;
    int len;
} result;

int main()
{
    int n;
    scanf("%d", &n);

    if ((n & (n - 1)) == 0) printf("No Answer");
    else {
        // i代表窗口头，j代表窗口尾
        int i = 1, j = 1;
        int sum = 0;  // 当前窗口内的和
        result res; res.len = 0;
        for (j = 1; j < n; j++) {
            sum += j;
            while (sum > n) sum -= (i++);
            if (sum == n && j - i + 1 > res.len) {  // 更新res
                res.i = i;
                res.j = j;
                res.len = j - i + 1;
            }
        }
```

```
        printf("%d=", n);
        for (i = res.i; i <= res.j; i++) {
            printf("%d", i);
            if (i != res.j) printf("+");
        }
    }

    return 0;
}
```

12. 合数分解

思路分析

合数分解算法有很多，这里就作一个最基本的分法讲解。

我们都知道，一个数除了1和其自身外，其它所有因子都不会超过气自己的平方根（数学知识），基于这一点，我们很容易设计出主意的算法：

1. 读入`n`，令`i`从2开始遍历到`sqrt(n)`，如果整除了该数，就输出这个因子`i`，并让`n /= i`，同时`i`自己要减一（因为一个数可能有多个相同的因子，比如45 = 3 * 3 * 5）；
2. 最后输出更新后的`n`——这一步不能忘，因为我们之前的步骤找到的都是除了`n`自身以外的因子，而还没有考虑到它自己。

参考代码

由于本题算法太简单，这里直接放出代码：

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


int main()
{
    int n;
    scanf("%d", &n);

    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) {
            printf("%d ", i);
            n /= i;
            i--;
        }
    }
    printf("%d", n);
```

```
    return 0;
}
```

需要注意的是`i <= sqrt(n)`的判断条件决定了每次遍历都要重新算一下`sqrt(n)`，会造成一定程度的性能浪费，但是由于我们的设计中，`n`是不断变化的，所以还只能这么去写（其它写法大家可以自己思考一下）。

13. 字母频率统计

思路分析

本题非常简单，我们要做的只有两件事：

1. 用一个数组去存每个字符出现的次数——此时可以用`ch - 'a'`来做下标的映射；
2. 设计一个`print`函数去可视化地输出。

具体实现过程

主函数中的逻辑应该非常清晰：

```
int main()
{

    int chart[26] = {0};  // 字母表
    char ch;

    while (scanf("%c", &ch) != EOF) {
        if (ch >= 'a' && ch <= 'z') chart[ch - 'a']++;
    }
    print(chart);

    return 0;
}
```

`print`函数是我们要着重设计的。

这个设计也非常简单：由于我们是一行一行输出的，所以就判断每一行中各个字母要不要输出就行，而一共有多少行显然取决于最大的字母频率：

```
void print(int *chart) {
    int cnt = 0;  // 最高的频率
    for(int i = 0;i < 26;i++)    cnt = max(cnt, chart[i]);
    while(cnt != 0) {
        for(int i = 0;i < 26;i++) {
            if(chart[i] == cnt) {
                printf("*");
                chart[i]--;  // 该字母的频率减一
```

```
            }
            else printf(" ");
        }
        cnt--;  // 行数减一
        printf("\n");
    }
    for(int i = 0;i < 26;i++)
        printf("%c",'a' + i);
}
```

参考代码

```
#include <stdio.h>

int max(int a, int b) { return a > b ? a : b; }
void print(int *chart);

int main()
{

    int chart[26] = {0};  // 字母表
    char ch;

    while (scanf("%c", &ch) != EOF) {
        if (ch >= 'a' && ch <= 'z') chart[ch - 'a']++;
    }
    print(chart);

    return 0;
}


void print(int *chart) {
    int cnt = 0;  // 最高的频率
    for(int i = 0;i < 26;i++)   cnt = max(cnt, chart[i]);
    while(cnt != 0) {
        for(int i = 0;i < 26;i++) {
            if(chart[i] == cnt) {
                printf("*");
                chart[i]--;  // 该字母的频率减一
            }
            else printf(" ");
        }
        cnt--;  // 行数减一
        printf("\n");
    }
    for(int i = 0;i < 26;i++)
```

```
        printf("%c",'a' + i);
}
```

14. 计算公式： 求π的值b

思路分析

本题的思路也非常简单：每次都加上公式中的下一项，判断是否小于给定的精度。唯一需要注意的是这个公式中我们自己要解出来π。

另外，本题全程都是浮点数运算，为了不给自己找麻烦，建议所有变量直接都声明成`double`。

具体实现过程

主函数的框架非常简单：

```c
int main()
{
    double e;
    scanf("%lf", &e);
    double cur = 2;

    for (int i = 2; ; i++) {
        double minus = 2 * factorial(i - 1) / double_factorial(i);
        cur += minus;
        if (minus < e) {
            printf("%d %.7lf", i, cur);
            break;
        }
    }

    return 0;
}
```

其中`factorial`和`double_factorial`函数分别计算阶乘和双阶乘：

```c
double factorial(int n) {
    double res = 1;
    for (int i = 1; i <= n; i++) res *= i;
    return res;
}

double double_factorial(int n) {
    double res = 1;
    for (int i = 3; i <= 2 * n - 1; i += 2) {
        res *= i;
    }
    return res;
}
```

参考代码

```c
#include <stdio.h>

double factorial(int n) ;
double double_factorial(int n) ;


int main()
{
    double e;
    scanf("%lf", &e);
    double cur = 2;

    for (int i = 2; ; i++) {
        double minus = 2 * factorial(i - 1) / double_factorial(i);
        cur += minus;
        if (minus < e) {
            printf("%d %.7lf", i, cur);
            break;
        }
    }

    return 0;
}


double factorial(int n) {
    double res = 1;
    for (int i = 1; i <= n; i++) res *= i;
    return res;
}
```

```
double double_factorial(int n) {
    double res = 1;
    for (int i = 3; i <= 2 * n - 1; i += 2) {
        res *= i;
    }
    return res;
}
```

15. 文件排版（非文件）

思路分析

首先虽然规则中没有说明，但是通过样例，我们可以看出需要将连续的空格和制表符合并为一个空格；假设我们将处理前读入的字符串存入 `str` 中；要把处理后 `:` 前的串存入 `str2`，处理后 `:` 后的串存入 `str3`，那么我们就可以通过遍历 `str` 的每一位，当遇到空格或者制表符时，如果当前 `str2` 或者 `str3` 的最后一位是空格（你遇到 `:` 前处理的就是 `str2`，遇到 `:` 后处理的就是 `str3`），那么我们就忽略这一位，开始枚举下一位；否则就在 `str2` 或者 `str3` 的末尾存入空格。这样就能合并连续的空格和制表符，变成一个空格。

我们得到 `str2` 和 `str3` 后，可以对 `str2` 按照指定位对齐格式化输出后，再输出 `:` `str3`。有的时候 `:` 前面和后面可能会有空格，有的时候则没有，为了统一处理，我们把最后要求输出时加在 `:` 前面的空格单独当做一个空格输出，而不是 `str2` 的一部分，也就是说处理完后如果 `str2` 最后有空格我们就把它去掉；把 `:` 的空格作为 `str3` 的一部分，也就是说在处理 `str3` 之间赋值 `str3[0]` 为空格。

最后要处理的问题是我们虽然知道应该把 `str2` 按照 `n - 2` 位右对齐输出（-2 是因为 `:` 和空格），但是 `n` 每次是个变量。实际上我们上学期程设也遇到过类似的问题，一种解决方法是用 `sprintf` 动态地改变格式串，即 `sprintf(geshi,"%-%%ds",n - 2);` ，其中负号代表右对齐。

参考代码

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

int read() {
    int tem;
    scanf("%d",&tem);
    return tem;
}

char str[10005]; // 原字符串
char str3[10005];// 处理后的 : 后的字符串
char str2[10005];// 处理后的 : 前的字符串
char geshi[10005];
```

```
int main() {

    int n = read();
    sprintf(geshi,"%-%%ds",n - 2);
    // 吃掉第一行有效语句前的所有无效字符
    while(getchar() != '\n');

    while(gets(str)) {
        memset(str2,0,sizeof(str2));
        memset(str3,0,sizeof(str3));
        int len = strlen(str);
        int pos = 0;
        int pos2 = 0, pos3 = 0; // 记录 str2 和 str3 枚举到的位置

        for(pos = 0;str[pos] != ':'; pos++) {
            // 合并空格和制表符为空格
            if(str[pos] == ' ' || str[pos] == '\t') {
                if(str2[pos2 - 1] != ' ')
                    str2[pos2++] = ' ';
            }
            else
                str2[pos2++] = str[pos];
        }
        if(str2[pos2 - 1] == ' ')
            str2[pos2 - 1] = 0;

        str3[pos3++] = ' ';

        pos++;  // 跳过 :
        for(;str[pos];pos++) {
            // 合并空格和制表符为空格
            if(str[pos] == ' ' || str[pos] == '\t') {
                if(str3[pos3 - 1] != ' ')
                    str3[pos3++] = ' ';
            }
            else
                str3[pos3++] = str[pos];
        }

        printf(geshi,str2);
        printf(" :%s\n",str3);
    }

    return 0;
}
```

16. 注释比例

思路分析

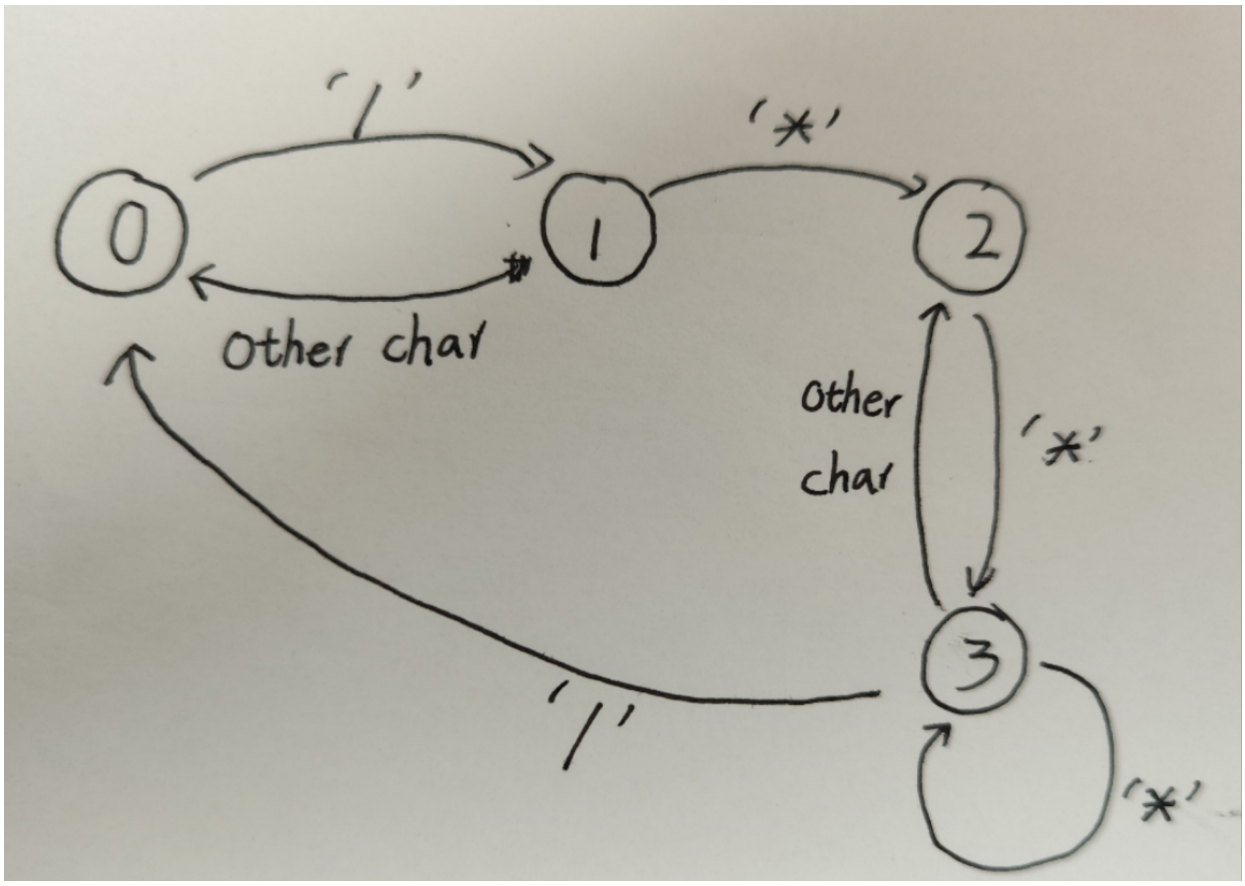
这道题无非是统计出一共有多少字符，多行注释内有多少字符。一共有多少字符很好统计，我们可以一个字符一个字符读入，每读入一个字符就把当前的总字符个数 + 1。那么该如何统计注释内有多少字符呢？其实跟统计一共有多少字符差不多，只不过我们需要知道当前是在多行注释内，再增加当前注释内的总字符个数 + 1。分别标记注释内的总字符个数为 `zhushi`，总的字符个数为 `total`。

假设我们定义一个变量 `sta`，当这个变量的值为 2 时表示我们在多行注释中，其它值为不在多行注释中，那么很显然当 $sta = 2$ 时，我们同时增加 `zhushi` 和 `total` 的值；当 $sta \neq 2$ 时，我们只增加 `total` 的值。

当程序开始时，我们让 `sta` 的值为 0，标记这是最一般的状态。那么什么时候有可能进入到多行注释呢？实际上，当我们在 `sta = 0` 时，遇到一个 `/`，我们就要小心了，因为如果下一个字符是 `*`，那么我们就进入了一个多行注释。所以，当我们在状态 `0` 下，读入了一个 `/`，那么我们就可以让 `sta = 1`，标记这是一个危险的状态。当 `sta = 1` 时，如果我们新读入的字符是 `*`，就证明我们真的进入了一个多行注释，标记 `sta = 2`；否则，如果是其它字符，证明我们刚才不过是虚惊一场，可以把 `sta` 重新赋值为 0，代表我们回到了一个普通状态，等待下一个 `/` 的出现。（因为按照题意，不会有单行注释，所以我们不必担心紧跟着 `/` 的这个字符还是 `/`）。

同样，在多行注释中时，我们还要时刻关注何时多行注释结束了。当我们遇到一个 `*` 时，如果下一个字符是 `/`，那么我们就离开多行注释了。所以当 `sta = 2` 时，当前读入字符为 `*`，我们可以让 `sta = 3`，表示准备离开多行注释的状态。当 `sta = 3`，且当前读入字符为 `/` 时，表示我们真的离开了一个多行注释，回到普通状态，重新让 `sta = 1`；否则证明我们只是虚惊一场，还是在注释内，让 `sta` 回到 2，标记在多行注释内。不过这里要注意，我们在 `sta = 3` 时可能还读入一个 `*`，这个 `*` 仍然可能是一个多行注释结束的表示，所以当 `sta = 3` 且读入字符是 `*` 时，我们仍要保持 `sta = 3`，这是一个非常容易错的地方（第四次作业括号匹配的题目可能就有这种风险）。

这就是状态转移的思想，在第四次作业中我们可以用这种思想去处理单行注释、多行注释、字符常量、字符串常量，用一张图表达这样的思想就是这样的：



最后输出 `%` 别忘了转义

参考代码

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

int main() {
    int total = 0, zhushi = 0;
    int sta = 0;
    char ch;    // 用来读入新的字符
    while(scanf("%c",&ch) != EOF) {
        total++;

        // 因为 * 可能是多行注释结束的标志，不作为多行注释的一部分
        if(sta == 2 && ch != '*')
            zhushi++;

        if(sta == 0 && ch == '/')   sta = 1;
        else if(sta == 1 && ch == '*')  sta = 2;
        else if(sta == 2 && ch == '*')  sta = 3;
        else if(sta == 3 && ch == '/')  sta = 0;
        else if(sta == 3 && ch == '*')  zhushi++;   // 证明我们上一个 *
时注释的一部分
```

```
        else if(sta == 3)    sta = 2;     // 虚惊一场，仍在注释中
    }

    printf("%d%%",zhushi * 100 / total);

    return 0;
}
```

17. 删除子串

思路分析

直接最朴素的思想，枚举原字符串的每一位，表示可能与子串匹配的起始位置。用一个变量 `i` 标记枚举到原串的位置，如果第 `i` 位不等于子串的第一位，那么显然以 `i` 开始的串不会是子串，直接把第 i 位存到答案输出数组中；否则比较一下，看看以 `i` 开始的若干位（子串长度位）是不是子串，如果是的话，那么就跳过这一段，直接让 `i` 的值增加子串长度位，下次从子串长度位之后开始枚举。

也可以用 `strstr` 查找原串中子串的位置，如果原串中没有子串，直接输出原串；否则如果找到子串的位置是 `pos`，那么就将 `pos` 之前的所有位存入答案数组，再标记查找的起始位置为 `pos + 子串长度`，直到找不到子串，再将剩下的位存入答案数组。

给出的代码就是第二种实现方式。

参考代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef long long ll;

char str[10005];
char subStr[10005];
char ans[10005];

int main() {
    gets(str);gets(subStr);
    int pos = 0;
    int find = 0;
    int cnt = 0,len = strlen(subStr),len2 = strlen(str);

    while ((find = strstr(str + pos,subStr) - str) > -1) {
        for(int i = pos;i < find;i++)
            ans[++cnt] = str[i];
        pos = find + len;
    }
```

```
    for(int i = pos;i < len2;i++)
        ans[++cnt] = str[i];

    printf("%s",ans + 1);

    return 0;
}
```