# 猪脚说第四期



# 提问规范（请大家务必仔细阅读）

开学至今已经一个月的时间了，大家也已经对这门课程完全熟悉了。

大家前两周问我的问题里，我大多仔细耐心地引导大家一步一步去发现自己程序的问题，因为我们认为大家仅仅经过半学期的编程训练，可能对调试方面的技巧掌握得并没有那么熟练。所以在前几期猪脚说里，我还详细地介绍了如何debug——包括何时调试，如何定位错点，自己构造测试数据等，如果大家**仔细阅读并在实践中应用**，那么现在的你的调试水平一定有了质的飞跃。

现在，我们会默认大家现在已经具备了基本的调试能力——自己已经能解决大多数问题。因此，在此对大家的提问方式做一个**明确的规范**——请明确：**助教不是你的编程助手**，发给他一个程序就能返回你的程序哪里出错。对**明显自己偷懒**的提问，我们将拒绝回答。

## 具体的知识点问题

如果有这方面基础知识的问题，比如理论课讲的链表基础知识等，欢迎大家尽情提问。

# 算法思路问题

> 这类问题比如：某道题不知道该怎么编程实现。

ds这门课几乎不要求大家的算法能力，但是如果大家有这方面的问题，也欢迎来提问，但请向助教**讲清楚你自己的思考，以及你的思路里哪里遇到困难不知道怎么解决。**

# debug

这类问题最多，主要有如下几类：

## 算命式问法

> 这类问题比如：我的输出莫名其妙多了一些内容，是怎么回事？

我们甚至不知道你问的是哪道题，你的代码长什么样，**助教不会算卦**，这种问题无法解决。

## 明确知道自己某个样例过不了，并且不知道错点

> 这类问题比如：题里给的样例没有过，能帮我看看是哪里的问题吗？

对这种问题，我们默认提问者是因为自己的懒惰导致不想debug，直接来问助教了——除非你能有合理的理由解释这一点。

原因也很简单：首先，代码已经写完了，说明大家算法上没有障碍。而且有明确的样例过不了了，按照我教给大家的调试方式，**至少是一定可以定位到错点的！**——这和任何因素都没有关系，只是体力活而已。就算助教给你de，那也只是助教帮你干了你自己应该干的体力活。

## 明确知道自己某个样例过不了，但知道错误的地方

> 这类问题比如：我的数据处理步骤结束后的数组里每次都多了一个0，不知道什么原因。

我完全不能理解的一点是：**都知道自己的程序那一块出了问题了，居然不知道怎么解决？？**

当然，在这几周的答疑过程中，确实有一些同学的bug比较隐晦，导致虽然他知道自己哪一步错了，但是具体的细节不清楚的情况。所以如果你有这种问题，请先确保自己**认真地检查过自己程序的相应位置**，并且**详细地告诉助教你进行了怎样的测试**却没有解决问题。

想提醒这类同学一下：提问前请慎重——因为根据经验来看，大多数人的问题都简单到令人发指，**甚至有的人连题面都没读就开始写**导致的这种问题。

## 题里的样例能过，直接来问数据点

> 这类问题比如：我的某题的第三个数据点没过，能把数据点给我吗？

在这里明确的告诉大家：从本次作业开始，不会有数据点放出，我在前几期猪脚说里也强调过自己构造测试数据的重要性。**我们不会给出任何测试点的提示。如果再有来问数据点的，一律不回复。**

## 题里的样例能过，来找debug

> 这类问题比如：我的第二个数据点没过，实在不知道什么问题了，能帮我看看吗？（并发一个代码文件）

问这类问题的同学请注意：如果像上面这样来问，一律视为偷懒——我说过要自己去构造样例。

所以如果有这种问题，请详细地告诉助教你自己尝试着构造了哪些数据点进行了测试——**请不要忽悠助教！**因为几乎所有这么来提问的同学都随便说了说自己怎么构造测试过，**但是我随便输入一个最基本的样例都过不了！**

当然，这些都只是为了规范大家的提问方式，避免同学因依赖助教而偷懒，我们仍然**欢迎大家来找我们一起探讨问题。**

# 链表基本操作

大家理论课应该都学到链表章节了，对应的第三层作业也发布了，这里我以一道最基础的链表操作的题目为例，向大家讲讲链表题目中编程的细节：

## 一道非常基础的链表题目

题目链接

本题实现过程如下：

首先声明结点类型：

```
typedef struct node {
    int val;  // 结点的值
    struct node *next;
} ndoe;
```

然后题目中说最开始有一个元素1，所以在主函数中对这个链表进行初始化：

```
// 主函数中
    node *list = (node *) malloc(sizeof(node));
    // 初始化
    list->val = 1;
    list->next = NULL;
```

接下来不断读入操作即可：

```
// 主函数中
    scanf("%d", &q);
    for (int i = 0; i < q; i++) {
        scanf("%d", &op);
        if (op == 1) {
            // 插入
            scanf("%d%d", &x, &y);
            insert(list, x, y);
        } else if (op == 2) {
            scanf("%d", &x);
            enquire(list, x);
        } else {
            scanf("%d", &x);
            delete(list, x);
        }
    }
```

下面重点就是设计这三个函数：

## insert

函数原型如下：

```
node* insert(node *list, int x, int y);
```

最开始要新建一个值为y的结点：

```
    // 创建新结点，其值为y
    node *target = (node *) malloc(sizeof(node));
    target->next = NULL;
    target->val = y;
```

然后要找到元素x的位置：

```
    // 从头开始，找到x元素的地址
    node *p = list;
    while (p->val != x) p++;
```

将新结点插入到`p`之后的本质其实就是改变`next`域的指向，建议大家在脑中或纸上模拟一下这个过程：

```
    // 实现插入
    target->next = p->next;
    p->next = target;
```

## enquire

函数原型如下：

```
void enquire(node *list, int x)
```

同样，也要先找到值为x的结点的位置：

```
    // 找到值为x的结点的位置
    node *p = list;
    while (p->val != x) p++;
```

然后要判断该节点后面是否为`NULL`来判断这是否为尾结点，并输出查询到的值：

```
    //特判尾结点，并输出
    if (p->next == NULL) printf("0\n");
    else printf("%d\n", p->next->val);
```

## delete

这个函数和`enquire`几乎相同，大家还是自己模拟一下这个删除的操作：

```
void delete(node *list, int x) {
    // 找到值为x的结点的位置
    node *p = list;
    while (p->val != x) p++;

    if (p->next != NULL) {
        // 不是尾结点，进行删除
        node *tmp = p->next;  // 要删除的结点
        p->next = tmp->next;
        free(tmp);
    }
}
```

但是洛谷网站上没有说明白输入的都是合法的，也即输入的`x`都一定对应着已经存在的结点，所以三个函数中对这个地方特判一下就行：

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

typedef struct node{
    int val;  // 结点的值
    struct node *next;
} node;

void insert(node *list, int x, int y);
void enquire(node *list, int x);
void delete(node *list, int x);

int main()
{

    node *list = (node *) malloc(sizeof(node));
    // 初始化
    list->val = 1;
    list->next = NULL;

    int q;  // 操作次数
    int op;  // 操作码
    int x, y;

    scanf("%d", &q);
    for (int i = 0; i < q; i++) {
        scanf("%d", &op);
        if (op == 1) {
            // 插入
            scanf("%d%d", &x, &y);
            insert(list, x, y);
        } else if (op == 2) {
            scanf("%d", &x);
            enquire(list, x);
        } else {
            scanf("%d", &x);
            delete(list, x);
        }
```

```
    }
    return 0;
}


void insert(node *list, int x, int y) {
    // 创建新结点，其值为y
    node *target = (node *) malloc(sizeof(node));
    target->next = NULL;
    target->val = y;

    // 从头开始，找到x元素的地址
    node *p = list;
    while (p->val != x && p) p++;
    if (!p) return;

    // 实现插入
    target->next = p->next;
    p->next = target;
}


void enquire(node *list, int x) {
    // 找到值为x的结点的位置
    node *p = list;
    while (p->val != x && p) p++;
    if (!p) return;

    //特判尾结点，并输出
    if (p->next == NULL) printf("0\n");
    else printf("%d\n", p->next->val);
}


void delete(node *list, int x) {
    // 找到值为x的结点的位置
    node *p = list;
    while (p->val != x && p) p++;
    if (!p) return;

    if (p->next != NULL) {
        // 不是尾结点，进行删除
        node *tmp = p->next;  // 要删除的结点
        p->next = tmp->next;
        free(tmp);
    }
}
```

## 哑结点

**理论课上老师应该讲过哑结点相关的知识**——有些操作中，使用哑结点会使得代码统一性强很多。

上面那道例题中，对于插入和删除的操作，如果改成插入或删除目标结点前面的结点的值，大家可以想象一下，这下一定要特判目标结点是头结点的情况：

1. 如果在头结点前面插入一个结点，显然链表的头结点改变了，需要更新；

2. 如果删除掉头结点，同理，头结点也会变化，也需要更新。

如此相对比较麻烦，但是如果应用了哑结点，这些操作就都统一起来了——因为头结点永远是这个“没用的”哑结点。

对于上面那道题，如果用哑结点，主函数中初始化应该这么去写：

```
// 主函数中
node *list = (node *) malloc(sizeof(node));  // 哑结点
list->next = NULL;

node *firstNode = (node *) malloc(sizeof(node));  // 第一个结点
firstNode->val = 1;
firstNode->next = NULL;

list->next = firstNode;  // 第一个结点复制
```

**请大家自己写一下**上面说的两种在前面插入和删除的操作，体会一下两种方法的不同之处。

# 23级程序设计基础训练题解

## 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：

```
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：

```
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

```
#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;
}
```