做好闭环（三）：编码能力训练篇的思考题答案都在这里啦！

2020-02-25 胡光

人人都能学会的编程入门课 进入课程 >

讲述：胡光

时长 12:09　大小 9.75M

你好，我是胡光。

不知不觉，我们已经学完了编码能力训练篇的全部内容。其实还有很多东西想给你讲，可限于篇幅，所以咱们整个编码能力训练篇中的内容，都是与接下来的算法数据结构篇有很大的联系，并且它们对于理解程序设计，也是非常基础且重要的内容。

有道是，授之以鱼，不如授之以渔，我也相信只要你跟着课程学习，一定会感觉到自己收获到了“钓鱼工具”。如果能引发你的主动思考，进而触类旁通，举一反三，那这场学习就更加有意义啦。

我也非常高兴，看到很多同学都在紧跟着专栏更新节奏，坚持学习。经常在专栏上线的第一时间，这些同学就给我留言，提出自己的疑惑。大部分留言，我都在相对应的文章中回复过了，而对于文章中的思考题呢，由于要给你充足的思考时间，所以我选择在今天这样一篇文章中，给你进行一一的解答。

看一看我的参考答案，和你的思考结果之间，有什么不同吧。也欢迎你在留言区中，给出一些你感兴趣的题目的思考结果，我希望我们能在这个过程中，碰撞出更多智慧的火花。

数学归纳法：搞定循环与递归的钥匙

在这一章里呢，我们介绍了保证程序正确性的最重要的数学思维：**数学归纳法**。并且，从数学归纳法出发，我们学习了递归程序设计。递归程序设计的几点要素，就是数学归纳法中的几个重要步骤。递归中的边界条件，就是数学归纳法中的 k_0，递归中的递归过程，就是数学归纳法中的假设 k_i 成立并证明 k_{i+1} 也成立那一步，最后两步结论放到一起，就能证明我们的递归程序整体是正确的。

思考题中呢，给你留了两个问题，第一个是将菲波那契数列的递归程序，改写成循环程序，关于这个问题，你可以参考留言区中 @奔跑的八戒、@徐洲更、@一步、@Geek_Andy_Lee00、@我思故我在 等用户的答案以及我在他们当中给出的回复内容。

第二个思考题呢，是做数学归纳法与菲波那契数列递归程序步骤的一一对应，关于这个问题，请看下面我给出的参考答案，看看和你想的有什么差别吧：

复制代码

```c
#include <stdio.h>

int fib(int n) {
    if (n == 1 || n == 2) return 1;
    return fib(n - 1) + fib(n - 2);
}

int main() {
    int n;
    scanf("%d", &n);
    printf("%d\n", fib(n));
    return 0;
}
```

其中代码的第 4 行，n == 1 和 n == 2 的条件判断，就是数学归纳法中所谓的 k_0 成立，这一步保证了，fib 函数计算的第 1 项 和 第 2 项的斐波那契函数值一定是正确的。代码的第 5 行中呢，就是假设 fib(n - 1) 和 fib(n - 2)的值是正确的，那么 fib(n) 就的值就等于 fib(n - 1) + fib(n - 2)，这就是数学归纳法中的第二步，假设 k_i 成立，证明 k_{i+1} 也成立。显然如果可以保证前两项的正确性，那么 fib(n) 的值一定正确。最后我们得出结论，这个fib 递归函数设计是正确的。

程序设计原则：把计算过程交给计算机

这一节中，我们强调了程序设计的基本原则，就是将计算过程交给计算机。我们负责逻辑组织，计算机负责具体计算过程，这就是所谓的专业的事情交给专业的人来做。

本节中的思考题是计算 100 以内自然数的 “和的平方” 与 “平方和” 的差值。在这里呢，我要给用户 @胖胖胖、@不便明言、@Geek_And_Lee00 点赞。具体的答案，你也可以参考这三个用户在留言区中的内容。

关于这道思考题的第一问，我就不给你做演示了，实现起来比较简单，你应该有能力自我完成的。下面，我主要给出 “平方和” 公式的推导过程，而对于 “和的平方” 你可以基于等差数列求和公式来求解。

教给你一种比较通用的推导平方和公式的方法，也是我用着最顺手的方法，就是依靠立方和，推导平方和。首先，我们先列出来相邻两项的立方差：

$$
\begin{aligned}
n^3-(n-1)^3 &= 3\times n^2-3\times n+1\\
(n-1)^3-(n-2)^3 &= 3\times (n-1)^2-3\times (n-1)+1\\
(n-2)^3-(n-3)^3 &= 3\times (n-2)^2-3\times (n-2)+1\\
&\cdots\\
2^3-1^3 &= 3\times 2^2-3\times 2^2+1\\
1^3-0^3 &= 3\times 1^2-3\times 1^2+1
\end{aligned}
$$

如上公式所示，我们将上面罗列的 n 个等式的左右两侧分别相加，就得到了如下式子：

$$左侧：n^3 = n^3 - (n-1)^3 + (n-1)^3 - (n-2)^3 + \ldots - 1^3 + 1^3 - 0^3$$

$$右侧：3 \times \sum_{i=1}^{n} i^2 - 3 \times \sum_{i=1}^{n} i + n$$

我们看到左侧就剩下一项 n 的立方了，这一项是可算的，右侧有一个 3 倍的平方和项，和一个 3 倍的等差数列求和项，以及一个常数项 n。接下来，左侧等于右侧，我们将平方和项与其他几项分别置于等式的两侧，就得到了如下平方和公式：

$$左侧 = 右侧：$$

$$n^3 = 3 \times \sum_{i=1}^{n} i^2 - (3 \times \sum_{i=1}^{n} i) + n$$

$$移项：$$

$$3 \times \sum_{i=1}^{n} i^2 = n^3 + (3 \times \sum_{i=1}^{n} i) - n$$

$$\sum_{i=1}^{n} i^2 = \frac{n^3 + (3 \times \sum_{i=1}^{n} i) - n}{3}$$

$$\sum_{i=1}^{n} i^2 = \frac{2 \times n^3 + 3 \times (1+n) \times n - 2 \times n}{6}$$

至此，我们就得到了平方和公式。其实，你还可以尝试使用这种方法，求解立方和公式，整体步骤差不多，就是先表示出相邻两项的四次方差，然后用如上步骤，继续推导即可。

框架思维（上）：将素数筛算法学成框架算法

这一节课，我们学习了素数筛算法，素数筛每一轮找到一个素数，然后在一个标记数组中，标记掉这个素数所有的倍数，剩下没有被标记掉的数字，就是我们要的素数了。最后，我留了一个程序性质证明题，具体看如下代码：

复制代码

```
#include <stdio.h>

// 打印一个素因子，并且在中间输出 * 乘号
```

```c
void print_num(int num, int *flag) {
    if (*flag == 1) printf(" * ");
    printf("%d", num);
    *flag = 1;
    return ;
}

int main() {
    int n, i = 2, flag = 0, raw_n;
    scanf("%d", &n);
    raw_n = n;
    // 循环终止条件，循环到 n 的平方根结束
    while (i * i <= n) {
        //①：只要 n 可以被 i 整除，就认为 i 是 n 的一个素因子
        while (n % i == 0) {
            print_num(i, &flag);
            n /= i;
        }
        i += 1;
    }
    //②：如果最后 n 不等于 1，就说明 n 是最后一个素数
    if (n != 1) print_num(n, &flag);
    printf(" = %d\n", raw_n);
    return 0;
}
```

第一个，是要证明第 18 行代码中，只要 n 可以被 i 整除，i 就一定是素数。关于这个证明，我们可以使用反证法。

假设 i 可以被 n 整除，但 i 不是素数，由算术基本定理可知，一个非素数的数字 N，一定可以分解为几个小于 N 的素数乘积的形式。我们不妨假设 $i = p_1 \times p_2$，这里 p_1 和 p_2 均为素数，如果变量 n 可以被 i 整除，那么 n 也一定可以被小于 i 的素数 p_1 整除。而根据程序的运行流程，n 中已经不可能存在小于 i 的因子了，所以 p_1 不具备存在的条件，故原假设不成立，i 是素数。

第二个，是要证明第 25 行代码中，为什么只要 n 不等于 1，n 就一定是素数呢？其实也可以参考第一问的证明流程。在 while 循环处理过程中，数字 n 中已经不可能存在小于等于 i 的所有的因子了，又因为此时 i 是大于根号 n 的一个值，也就是说，在小于等于根号 n 范围内，找不到数字 n 的非 1 因子，而能够满足这种性质的数字，一定是素数。

至此，我们就证明完了程序中两处代码的性质。

数据结构（上）：突破基本类型的限制，存储更大的整数

在这一节中，我们学习了大整数表示法，说明了如果是数据表示的导致的程序设计过程不可行，那么我们就需要在数据结构中寻找解决方案了。

在大整数表示法中，我们是将一个数字，从右到左倒着存储在数组中，并且用数组的 0 位存储数字的位数。数组中存储的数字大小，应该等于其每一位的数字乘上相关存储位置的位权，数组的 1 位位权为 1，也就是 10 的 0 次方，2 位位权为 10，也就是 10 的 1 次方，以此类推。

那么接下来，我们理解大整数的乘法，也是通过这种数学公式上面的等价关系，来理解大整数乘法过程。最后给你留了一个编程题，是关于实现读入两个大整数，并且计算两个大整数加法结果的程序，以下是我的参考代码：

复制代码

```
#include <stdio.h>
#include <string.h>
#define MAX_N 1000
char str_a[MAX_N + 5], str_b[MAX_N + 5];
int num1[MAX_N + 5], num2[MAX_N + 5], num3[MAX_N + 5];

void convert_to(char *str, int *num) {
    num[0] = strlen(str);
    for (int i = num[0] - 1; i >= 0; i--) {
        num[num[0] - i] = str[i] - '0';
    }
    return ;
}

void output_big_integer(int *num) {
    for (int i = num3[0]; i >= 1; i--) {
        printf("%d", num3[i]);
    }
    return ;
}

int main() {
    scanf("%s%s", str_a, str_b);
    convert_to(str_a, num1);
    convert_to(str_b, num2);
    plus_big_integer(num1, num2, num3);
    output_big_integer(num3);
    return 0;
}
```

可以看到，首先读入两个字符串 str_a 和 str_b，分别代表第一个和第二个大整数。然后调用 convert_to 方法，将第一个字符串与第二个字符串，转换成大整数表示法，分别存储在 num1 和 num2 数组中；然后再调用 plus_big_integer 方法，将两个大整数的加法结果，存储在 num3 数组中；最后，输出 num3 数组中所存储的大整数。其中，提到的 plus_big_integer 方法，在原文中有给出，你可以回到原文中进行查看。

这段程序设计中，最应该值得你注意的是，我们将大整数操作的相关过程，均封装成了函数方法。字符串转大整数表示法，封装成了函数 convert_to；大整数加法过程，封装成了 plus_big_integer；输出大整数，封装成了 output_big_integer。

封装成函数方法的好处，就在于只要保证每一个小方法是正确的，就能保证整个程序的正确性。更重要的是，如果你单独看主函数的话，即使不看每一个方法的具体实现过程，你也能够清晰的知道，这个程序流程究竟在干什么，增强了代码的可读性。最后一点好处，就是出现 Bug 的时候，便于改错。

关于第 17 篇文章中，所说的改进 Shift-And 算法中的数据结构，我这里给你个提示，你可以参考大整数表示法，再参照这道题目中的程序设计原则，将操作封装成函数。

对于改进 Shift-And 算法中的数据结构，你需要做的就是用大整数表示法，表示一个二进制数字，然后根据 Shift-And 算法的需求，做好需要封装的操作有：**左移**、**或 1 操作**、**与运算**以及**判断这个数字的第 m 位是否为 1** 这些需要封装的操作。最终你会发现，算法流程没有改变，改变的只有程序样式。更多内容呢，你可以参考文章中，我与 @陈洲更 的留言讨论内容。

好了今天的思考题答疑就结束了，如果你还有什么不清楚的，或者有更好的想法的，欢迎告诉我，我们留言区见！

上一篇　17 | 数据结构（下）：大整数实战，提升 Shift-And 算法能力

精选留言 (2)

写留言

胖胖胖
2020-02-25

老师，函数指针后面会讲嘛，还挺想学这部分内容的。。。

作者回复: 后面就主要给你们讲算法和数据结构相关的东西了。关于 C 语言的函数指针相关的内容，其实掌握的关键点就在于函数指针的定义语法，其他的部分，和正常指针无异。

一步
2020-02-25

老师，关于第15讲【框架思维下】有点疑问，已在第15讲下留言，希望老师能解惑

作者回复: 好的，疫情期间，被憋在家办公，需要处理自己的生活起居，衣食住行，连外卖都点不了，所以回复不及时。-_-|||