15 | 框架思维（下）：用筛法求解其他积性函数

2020-02-18 胡光

讲述：胡光

时长 16:20 大小 13.09M

你好，我是胡光，咱们又见面了。

上一节，我们讲了素数筛这个算法，并且强调了，要按照框架思维去学习算法代码，因为当你学会这么做的时候，它就可以变成解决多个问题的利器了。

本节我将带你具体使用素数筛算法框架，去解决一些其他简单的数论问题。通过解决这 个具体问题的过程，我希望你能找到“框架思维”的感觉。

今日任务

今天这个任务，需要你依靠自己的力量来完成。不过你也不用担心，我会把需要做的准备工作都讲给你。

这个任务和因数和有关，什么叫做因数和呢？就是一个数字所有因数的和。那么什么是一个数字的因数呢？因数就是小于等于这个数字中，能整除当前数字的数。例如，28 这个数字的因数有 1、2、4、7、14、28 ，因数和就是各因数相加，即 56。

所以今天我们要做的，就是求出 10000 以内所有数字的因数和。你明白了要算的结果后，可能已经想出采用如下方法来解决：

复制代码

```
#include <stdio.h>
int sum[10005] = {0};

void init_sum() {
    // 循环遍历 1 到 10000 的所有数字
    for (int i = 1; i <= 10000; i++) {
        // 用 j 循环枚举数字 i 可能的因数
        for (int j = 1; j <= i; j++) {
            // 当 i%j 不等于 0 时，说明 j 不是 i 的因数
            if (i % j) continue;
            sum[i] += j;
        }
    }
    return ;
}

int main() {
    init_sum();
    printf("hello world\n");
    return 0;
```

我们具体来看一下上面这个方法是怎么做的：在代码中，init_sum 函数内部就是初始化 sum 数组信息的方法，sum[i] 存储的就是 i 这个数字所有的因数和。在 init_sum 方法内部，使用了双重循环来进行初始化，外层循环 i 遍历 1 到 10000 所有的数字，内层循环遍历 1 到 i 所有的数字，然后找出其中是数字 i 因数的数字，累加到 sum[i] 里面，以此来计算得到数字 i 所有的因数和。

这个方法呢，诚然是正确的，可如果你真的运行上述代码，你会发现它会运行一段时间，即使你的电脑配置再好，也会感到它好像卡顿一下，然后才在屏幕上输出了 hello world 这一

行信息。什么意思呢？，这表示这种程序方法运行速度较慢。

程序就像一个百米赛跑运动员，衡量一个百米赛跑运动员成绩的指标，除了看他能否到达终点，还有更重要的，就是完成比赛的时间。因此，你不仅要关注程序设计的正确性，还要关注程序的运行效率。

好了，了解完今天的任务以后，下面就让我们来看看，想要设计一个更好更快的程序，都需要准备哪些基础知识吧。

必知必会，查缺补漏

为了解决今天这个问题，你需要一点儿数论基础知识的储备。下面呢，我将分成三部分来给你讲解准备工作：

> 第一部分是掌握数论积性函数基础知识。有道是工欲善其事，必先利其器，数论是完成今日任务的重要利器。
>
> 第二部分，我会举一个具体数论积性函数的例子，就是求一个数字的因数的数量。
>
> 最后，我们会把因数数量的求解问题，套在我们之前所学的素数筛算法框架中，以此来说明**素数筛的算法框架，基本上可以求解所有的数论积性函数**。通过这个过程，彻底让你感受到框架思维的威力。

好了，废话不多说，让我们正式开始今天的学习吧。

1. 数论积性函数

首先我们来看一个知识点，就是关于“数论积性函数”的知识。所谓数论积性函数，首先，是作用在正整数范围的函数，也就是说函数 f(x) = y 中的 x 均是正整数。其次，是数论积性函数的一个最重要的性质，就是如果 n 和 m 互质，那么 f(n*m) = f(n) * f(m) 。

什么是互质呢？就是两个数字的最大公约数为 1，关于最大公约数的相关内容的话，是小学的基本内容，如果你实在是忘记了，就自行上网搜一下吧，我就不再赘述了。总地来说，只要一个函数满足以上两点，我们就可以称这个函数为数论积性函数。

这里我给出一个具体示例，帮助你理解：

数论积数函数：　1：定义在正整数的范围

2：如果 n 和 m 互质，则 f(n * m) = f(n) * f(m)

示例：　已知，f 是数论积性函数

并且，f(2) = 3，f(3) = 4

则，f(6) = f(2) * f(3) = 12

其实我给你讲述这个数论积性函数这个定义的时候呢，并不希望你对它是死记硬背，而是希望你在理解这个定义的时候，可以凭借敏锐的嗅觉，或者说培养自己这方面的意识，能在这里面想到更多。

什么意思呢？当你看到数论积性函数中的 f(n * m) = f(n) * f(m) 的公式的时候，这就应该引起警觉：这个公式中，n*m 是一个要比 n 和 m 都大的值，而 f(n * m) 的函数值却是由 f(n) 和 f(m) 决定的。

这说明什么？说明我们可以利用较小数据 f(n) 和 f(m) 的函数值，计算得到较大数据 f(n * m) 的函数值。再往深的想，这其实就是一个由前向后的递推公式（可以看到递推公式的应用范围其实很广），也就是说，只要函数 f 是数论积性函数，就可以做递推！

这么说的话，你可能还是一脸懵，可以做递推有啥好的？那你就想错了，简单来说，做递推公式可以计算的更快！下面呢，我们就来看一个具体数论积性函数的例子。

2. 因数个数函数

在前面我们介绍了因数和的概念，那么因数个数的概念，就不难理解了，它指的是一个数字因数的数量。例如，数字 6，有 1、2、3、6 这 4 个因数，因数个数就是 4。

通常情况下，我们如何计算因数个数呢？这个其实比较简单，我们利用反向思维，考虑如何构造一个数字的因数。就拿 12 个数字来说吧，12 的因数需要满足什么条件呢？

第一，就是 12 的所有因数中只能包含 2 和 3 两种素因子；第二，就是 12 的所有因数中，2 和 3 素因子的幂次，不能超过 12 本身的 2 和 3 素因子的幂次。也就是说，12 的因数中最终可以含有 2 的 2 次方，不能含有 2 的 3 次方，因为 12 中最多就只有 2 个素因子 2，一个素因子中含有 3 个 2 的数字，不可能是 12 的因数。

综合以上两点，我们其实只要组合 2 和 3 可能取到的所有幂次，就能得到所有 12 的因数。

$$
\begin{aligned}
12 &= 2^2 \times 3^1 \\
1 &= 2^0 \times 3^0 \\
2 &= 2^1 \times 3^0 \\
4 &= 2^2 \times 3^0 \\
3 &= 2^0 \times 3^1 \\
6 &= 2^1 \times 3^1 \\
12 &= 2^2 \times 3^1
\end{aligned}
$$

正如你所看到的，在构造 12 的因数的时候，2 的幂次从 0～2 有 3 种取值，3 的幂次从 0～1 有 2 种取值，总共的组合数就是 3 * 2 = 6 个，也就是说，12 一共有 6 个因数。

最后，就让我们来总结一下，如何计算一个数字的因数数量。对于一个数字 N，假设数字 N 的素因子分解式可以表示为：

$$
N = {p_1}^{a_1} \times {p_2}^{a_2} \times {p_3}^{a_3} \times \ldots \times {p_m}^{a_m}
$$

其中，p_i，就是数字 N 中的第 i 种素因子，a_i 就是第 i 种素因子的幂次。根据上面我们对于 12 这个数字因数数量的分析，就可以得到数字 N 的因数数量函数 g(N) 的公式表示：

$$
g(N) = (a_1 + 1) \times (a_2 + 1) \times (a_3 + 1) \times \ldots \times (a_m + 1)
$$

正如你所见，g 函数计算的就是数字 N 中各种素因子幂次数的一个组合数，就是数字 N 的因数数量。而这个 g 函数呢，就是我们之前所说的数论积性函数。对于数论积性函数来

说，关键就是证明第二点，即当 n 和 m 互素，g(n * m) = g(n) * g(m)。关于这个证明，首先我们先把 n 和 m 的素因子分解式和因数数量表示出来：

n 和 m 的素因子分解式

$$n = {p_1}^{a_1} \times {p_2}^{a_2} \times \ldots \times {p_i}^{a_i}$$
$$m = {q_1}^{b_1} \times {q_2}^{b_2} \times \ldots \times {q_j}^{q_j}$$

n 和 m 的因数数量表示

$$g(n) = (a_1 + 1) \times (a_2 + 1) \times \ldots \times (a_i + 1)$$
$$g(m) = (b_1 + 1) \times (b_2 + 1) \times \ldots \times (q_j + 1)$$

因为 n 和 m 互素，所以 n * m 的素因子分解式和因数数量表示出来，就如下式所示：

n * m 的素因子分解式

$$\begin{aligned} n * m = {} & {p_1}^{a_1} \times {p_2}^{a_2} \times \ldots \times {p_i}^{a_i} \\ & \times {q_1}^{b_1} \times {q_2}^{b_2} \times \ldots \times {q_j}^{q_j} \end{aligned}$$

n * m 的因数数量表示

$$\begin{aligned} g(n * m) & = (a_1 + 1) \times (a_2 + 1) \times \ldots \times (a_i + 1) \\ & \times (b_1 + 1) \times (b_2 + 1) \times \ldots \times (q_j + 1) \\ & = g(n) * g(m) \end{aligned}$$

这样，我们就证明了，在 n 和 m 互素的情况下，g(n * m) = g(n) * g(m)，所以 g 函数是数论积性函数。至此，我们完成了所有基础数学知识的准备。

下面呢，我们将从理论向实践迈进，也就是朝代码实现的方向迈进，实现一个求解 10000 以内所有正整数因子个数的程序。

3. 素数筛框架登场

如果想利用 g 函数的数论积性特点，我们就必须能够将一个数字 n，快速的分解成互素的两部分。如果我们能快速的拆解出一个数字 n 中的某种素数的话，那么这种素数，与剩余的部分，不就是互素的两部分么？

例如，如果我们能从数字 12 中，快速的拆解出只包含素数 2 的部分，就是因子 4，那么 4 与剩余的部分，数字 3 之间一定是互素的。想要完成这个子任务，我们可以求助素数筛框架，我对素数筛的代码做了一个小小的改动：

复制代码

```
#define MAX_N 10000
int prime[MAX_N + 5] = {0};
void init_prime() {
    for (int i = 2; i * i <= MAX_N; i++) {
        if (prime[i]) continue;
        // 素数中最小的素因子是其本身
        prime[i] = i;
        for (int j = 2 * i; j <= MAX_N; j += i) {
            if (prime[j]) continue;
            // 如果 j 没有被标记过，就标记成 i
            prime[j] = i;
        }
    }
    return ;
}
```

正如代码所示，init_prime 函数是初始化 prime 数组信息的方法，只不过是 prime 数组中记录的信息与之前的素数筛程序不同了。这个程序中，prime[i] 中记录的是数字 i 中最小的素因子，例如 prime[8]中记录的是 2，prime[25] 中记录的是 5。当初始化完 prime 数组以后，我们利用 prime 数组中的信息，就可以快速地完成将一个数字拆解成互素的两部分。

下面这份代码，展示的就是我们如何利用 prime 数组，计算因数数量：

复制代码

```
int g_cnt[MAX_N + 5];
void init_g_cnt() {
    // 1 的因数数量就是 1 个
    g_cnt[1] = 1;
    for (int i = 2; i <= MAX_N; i++) {
        int n = i, cnt = 0, p = prime[i];
        // 得到数字 n 中，包含 cnt 个最小素因子 p
        while (n % p == 0) {
            cnt += 1;
            n /= p;
        }
        // 此时数字 n 和最小素数 p 部分，就是互素的
        g_cnt[i] = g_cnt[n] * (cnt + 1);
    }
    return ;
}
```

这份代码中，g_cnt 数组记录的就是因数数量信息。在 init_g_cnt 函数中，一开始将 g_cnt[1] 置为 1，由于数字 1 的因数数量只有它自己本身，所以也就是 1 个。然后从 2 到 10000 循环，依次求解每个数字的因数数量。

循环内部，将数字 i 中，除去最小素因子的剩余部分存储到 n 中，将最小素因子的次数存储在 cnt 变量中。由于因数数量函数是积性函数，最终用 g_cnt[n] 乘上最小素因子 p 部分的 g_cnt 的值，也就是 cnt + 1 的值，即可。

这个程序之所以运行效率快的原因呢，我今天不做具体讨论，你只需要知道，这个程序比我们开始说的那个双层循环程序，运行速度快了一个数量级。

实际上，如果你掌握了“欧拉筛”相关内容，这个程序你会实现得更加漂亮，也更加能够体现我们所说的“框架思维”。“欧拉筛”实际上也是一种筛选出素数的方法，比我们之前学的素数筛更高效，同时，我也认为它体现的思想也更优美，你要是有兴趣，可以自行网上搜索了解。

一起动手，搞事情

前面，我给出了完整的求解因数数量的代码，以及相关数学公式的推导过程。其实，在最开始我们所说的因数和的求解任务，和因数数量的求解类似，都是基于对数字 N 的素因子分解式的观察和思考，得到相关的推导公式。并且，我这里可以预先给你一个确定性的结论，那就是因数和公式，本身也是数论积性函数。

说到这里，你可能就明白了，今天这堂课的作业，其实就是让你参照本节求解“因数数量”的过程，完成求解“因数和”的任务。你需要自行搜索的内容就是约数和公式，或者可以搜索任意一篇相关数论积性函数的文章，里面大概率也都会讲到这部分知识，然后找到解题方法。

课程小结

最后，我们来做一下今天的课程总结。我就希望你记住一点：所谓代码框架，就是要活学活用。

因为在真正的工作中，你所做的事情，大多是在多种代码框架之间做选择及组合拼装，每个算法代码只会解决遇到的一部分问题。而你在使用这些算法代码的时候，往往不能照搬照用，反而要做一些适应性的改变，这些都是“框架思维”中所重视的。

好了，今天就到这里了，我是胡光，我们下期见。

上一篇 14 | 框架思维（上）：将素数筛算法写成框架算法

下一篇 16 | 数据结构（上）：突破基本类型的限制，存储更大的整数

精选留言 (4)

写留言

一步

2020-02-23

12 的因数需要满足什么条件呢?
第一，就是 12 的所有因数中只能包含 2 和 3 两种素因子；第二，就是 12 的所有因数中，2 和 3 素因子的幂次，不能超过 12 本身的 2 和 3 素因子的幂次

对于上面这句话没有理解。 ...

展开

作者回复: 你把 12 的因子都写出来，然后用素数的形式表示一下，你看看 12 的因子，都有什么特点。

我们之所以要观察 12 因子的特点，是因为后续，我们需要通过某种方法，反向构造出 12 的因子。

关于第2点的解释，你想想8也含有素数2，12也含有素数2，为什么8不是12的因子？因为8里面，含有3个素数2，也就是2的三次方，12只含有2个2，所以如果一个数字是12的因子，那么这个数字中素因子 2 的幂次，肯定不超过 2 次。

胖胖胖

2020-02-22

```
//嵌入欧拉筛版
#include <stdio.h>
#include <math.h>
#define MAX_N 100
int p[MAX_N + 1] = {0};...
```

展开

作者回复: 不错，不过p_sum[i]在i与p[j]不互素中的赋值操作是多余的。

胖胖胖
2020-02-22

因数和：
#include <stdio.h>
#include <math.h>
#define MAX_N 100
int p[MAX_N + 1] = {0};...
展开

作者回复: 程序的思路没有错，其实可以把因数和的过程，直接镶嵌到欧拉筛的算法中，而不是像素数筛一样。欧拉筛可以做到，求完素数以后，因数和就求完了。
你主要观察你代码欧拉筛中的标记过程，标记过程，使用 i 标记掉 i * p[j]，而此时 p[j] 是素数，而且明显有两种情况：1、p[j] 与 i 互素以及 p[j] 与 i 不互素。互素的情况用积性函数的性质，直接计算即可，不互素的情况，稍微处理一下即可。^_^

😁HappyJoo
2020-02-19

老师你好，我看了一下《Expert C Programming》，hmmm，好像还是看不太懂呢，我是不是应该先去把啊哈算法看完？又或者应该先去打多一些代码？虽然我连框架思维这两章还没怎么学明白，是不是应该多写一写别的代码呢？
展开

作者回复: 对，C专家编程放到后面看。先看一些基础的书籍。多打多练。要不然后面更困难。(｡ì _ í｡)

1