基于求和的整数分解算法

Justin Friedlander

2025 年 3 月

1 介绍

众多方法已被考虑用于创建快速整数分解算法。尽管看起来简单，找到这样的算法的难度在现代密码学中扮演着关键角色，特别是在 RSA 加密的安全性方面。一些快速因式分解的方法包括试除法、Pollard's Rho 和 p-1 方法以及各种筛法 [1]。

本文介绍了一种新方法，将整数转换为以 2 为底的和。通过结合该整数的十进制和二进制表示，一个时间复杂度为 $\sqrt{n}$ 的算法可以将该和转化为两个整数的乘积，从而对原始数字进行因式分解。

2 方法

第一步：遍历 j 和 i

令 $n = pq$ 对于整数 n,p 和 q。注意，p 和 q 可以用二进制表示。然而，考虑 p 和 q 的最高次幂。即，$\lfloor log_2(p) \rfloor$ 和 $\lfloor log_2(q) \rfloor$。不失一般性，设 $p \geq q$。令 $j = \lfloor log_2(p) \rfloor$ 和 $i = \lfloor log_2(q) \rfloor$。注意，对于某些整数 $c_i < 2^j$ 和 $c_j < 2^i$，有 $p = 2^j + c_i$ 和 $q = 2^i + c_j$。

请注意现在 $n = pq = (2^j + c_i)(2^i + c_j) = 2^{j+i} + c_j 2^j + c_i 2^i + c_j c_i$。

我们也可以用二进制表示 n，然而，它可能与我们对 pq 的表示相同或不同。

定理 *1*:

令 $n = 2^k + c_k$ 为 $k = \lfloor log_2(n) \rfloor$ 和 $c_k < 2^k$。
声明: 对于所有 n,p 和 q, 都有 $k = j+i$ 或 $k = j+i+1$。

证明:
下界 -$n = 2^k + c_k = (2^j + c_i)(2^i + c_j)$。设 $c_j = c_i = 0$。现在，$n = 2^k + c_k = 2^{j+i}$. 因为 k 是在增加到超过 n 之前 2 的最大幂次，$j + i = k$。因此对于任意的 c_j 和 c_i，$j + i \leq k$ 成立。

上界 -$c_j < 2^i$ 和 $c_i < 2^j$。因此，

$$\begin{aligned} n &= 2^k + c_k \\ &= (2^j + c_i)(2^i + c_j) \\ &= 2^{j+i} + c_j 2^j + c_i 2^i + c_j c_i \\ &< 2^{j+i} + 2^i 2^j + 2^j 2^i + 2^i 2^j \\ &= 4 * 2^{j+i} \end{aligned}$$

由于 $2^k + c_k < 4 * 2^{j+i}$，则 $2^k + c_k < 2^{j+i+2}$。因此，为了让等式左边和右边相等，我们必须至少将右边减一。这剩下 $2^k + c_k = 2^{j+i+1} + c_{decrement}$。再次，由于 2^k 是在增加到超过 n 之前 2 的最大幂次，$k = j + i + 1$。

因此，$k \leq j + i + 1$，所以对于所有的 n，p 和 q 均有 $j + i \leq k \leq j + i + 1$。

定理 1 的含义是，算法需要运行一次来检查 $k = j + i$ 的情况，并且在最坏的情况下，还需要第二次运行来检查 $k = j + i + 1$。

此外，当给出一个幂 k 时，数字 j 和 i 是未知的。因此，算法必须搜索所有 j 和 i 的组合，使得 $k = j + i$ 或 $k = j + i + 1$。

步骤二：遍历通过 c_J

由于我们正在遍历所有 j 和 i 的组合，对于算法的下一阶段，我们可以假设我们的选择 j 和 i 是与 p 和 q 对应的正确选择。也就是说，$j = \lfloor log_2(p) \rfloor$ 和 $i = \lfloor log_2(q) \rfloor$。由于以下论证对于 $k = j + i$ 和 $k = j + i + 1$ 几乎相同，我们将为了简单起见假设 $k = j + i$。

我们已知 $n = 2^k + c_k = 2^{j+i} + c_j 2^j + c_i 2^i + c_j c_i$ 和 $2^k = 2^{j+i}$。因此，$c_k = c_j 2^j + c_i 2^i + c_j c_i$。我们可以通过将它转换为二进制形式来表示 c_k。这是一个过程的示例：

> $c_k = 61$, $j = 4$, $i = 2$.
> $c_k = 2^5 + 2^4 + 2^3 + 2^2 + 2^0 = 2 * 2^4 + 2^4 + 2 * 2^2 + 2^2 + 2^0 = 3 * 2^4 + 3 * 2^2 + 1 = 3 * 2^j + 3 * 2^i + 1$

我们可以定义 c_J 和 c_I 分别等于 2^j 和 2^i 的相应系数，并且在将 c_k 化简为这种形式后，定义 B 等于 2^0 的系数。注意 $c_J 2^j + c_I 2^i + B = (c_J - e)2^j + (c_I + e2^{j-i})2^i + B = c_J 2^j + (c_I - d)2^i + (B + d2^i)$ 对于某些整数 e 和 d。从上面的示例中，我们可以写出：

> $3 * 2^4 + 3 * 2^2 + 1 = (3 - 2) * 2^4 + (3 + 2 * 2^{4-2})2^2 + 1 = 2^4 + (11 - 2)2^2 + (1 + 2 * 2^2) = 2^4 + 9 * 2^2 + 9$

现在，如果我们重新引入 2^k 项，我们得到

> $2^k + c_k = 2^{4+2} + 2^4 + 9 * 2^2 + 9 = (2^4 + 9)(2^2 + 1) = 25 * 5 = pq$

请注意，当我们得到正确的 c_j 和 c_i 系数时，$c_j c_i = b$ 就会成立，其中 b 是我们的 2^0 系数。

我找到的将 c_J、c_I 和 B 转换为 c_j、c_i 和 b 的算法必须考虑，在最坏的情况下，从 c_J 到 1 的所有 2^j 系数的迭代。由于我们正在遍历该系数的所有值，我们可以假设该系数是 c_j。

第三步：寻找 c_i

令 $e = c_J - c_j$ 和 $c'_I = c_I + e * 2^{j-i}$。我们可以使用下面的方程来找出 d 之间的差异 c'_I 和 c_i：

方程 1：$\frac{(c_J - e)c'_I + B}{c_J - e + 2^i} = d$

由此，我们可以从 $c'_I - d$ 计算出 c_i，并从 $c_J - e$ 计算出 c_j。由于我们知道我们的 c_j 和 c_i，我们也知道 j 和 i，所以我们知道了项 $(2^j + c_i)(2^i + c_j) = n$，因此我们可以推导出我们的 p 和 q。

3 时间复杂度

在算法的第一部分，我们正在遍历所有 j 和 i 的组合，使得 $k = j + i$ 或 $k = j + i + 1$。由于 k 近似于 $log(n)$，这一步需要近似 $log(n)$ 次迭代。在算法的第二步中，我们必须遍历 2^j 项的所有系数。由于 $c_j < 2^i$ 和 $j \geq i$，在最坏情况下我们有 $c_J \leq \sqrt{n}$。这意味着这一步可以进行 $\sqrt{n}$ 次迭代。在第三步，我们从 c_j 计算 c_i，这是一个常数时间计算。

因此，整个算法似乎需要 $O(\sqrt{n}log(n))$ 的时间来运行。然而，更仔细的检查揭示了这一数字的一个小改进。当 $j \approx k$ 时，则 $i \approx 0$ 因为 $k - j = i$。在这种情况下，c_J 比 $\sqrt{n}$ 更接近于 0。更一般地说，每次迭代 j 和 i 都会使 c_J 的可能值增加大约两倍。因此，在这个算法中执行的总操作数更接近于 $2 * \sum_{k=0}^{log_2(n)} \frac{\sqrt{n}}{2^k} \approx 4\sqrt{n}$。因此，该算法的运行时间是 $\sqrt{n}$ 级别的。

4 讨论

该算法未能改进一般数域筛法 [2] 的时间复杂度，但它确实引入了一种新的整数分解方法，据我所知，这种方法之前尚未被考虑过。经过超出本文范围的分析，我认为在不完全改变算法的情况下，显著降低此算法的时间复杂度是不可行的。因此，我现在正在探索可能为进一步优化打开大门的量子计算选项，并鼓励对这一方法感兴趣的其他人也这样做。
经典算法的 Python 实现可以在这里找到。

参考文献

[1] Samuel S. Wagstaff Jr., *The Joy of Factoring, Chapter 3: Classical Factorization Methods*, 2002, Online, https://www.cs.purdue.edu/homes/ssw/chapter3.pdf.

[2] Hendrik W. Lenstra Jr., *Algorithms in Number Theory*, Proceedings of the International Congress of Mathematicians, 1993, pp. 897-908, https://pub.math.leidenuniv.nl/~lenstrahw/PUBLICATIONS/1993e/art.pdf.