第五届（2020 年）
全国高校密码数学挑战赛
参赛论文

子集和问题

战队名称 小绿草

学　　校 南京邮电大学

学　　院 计软网安院

专　　业 信息安全

参赛选手 殷祥、曹议、汤慧敏

指导教师 张平

二零二零年六月

摘　要

在本次第五届全国高校密码数学挑战赛之赛题三中，一共有 9 级维度逐级增加的稀疏子集和挑战，我们完成了前 8 级挑战，但因时间和算力的限制，第 9 级挑战只计算出一个欧几里得范数较小的解。为解出这些挑战，我们首先对现有的与子集和问题相关的理论和算法进行了一定的调研。经过调研，我们发现这 9 级挑战中的子集和问题都可以转化为求解格中最短向量的问题。在 Schnorr 等人求解子集和问题实验的基础上，我们设计了自己的求解算法，选择 BKZ 算法作为求解格中最短向量的工具，并使用 SageMath 软件对求解算法进行实现，成功地解出了前 6 级挑战。但由于找到格中最短向量的难度会随着维度的升高而指数级增加，我们未能利用该求解算法解出后 3 级挑战。后来，受 NTRU 密码分析中 Zero-Forced Lattice 的启发，我们将 Zero-Forced 技巧也运用到子集和问题中。利用 Zero-Forced 技巧，我们可以对高维度的格子进行降维，从而在一定程度上降低了求解算法的时间复杂度。最后，我们成功地解出了前 8 级挑战，但由于时间和算力的原因，对于最后第 9 级挑战，我们只能求出近似解。此外，我们还在求解过程中对 BKZ 算法在实际中的性能表现进行了相应的分析。

关键词：子集和问题，格中最短向量问题，BKZ 算法，Zero-Forced Lattices

目 录

第一章　引言

子集和问题指的是，给定某个由n个正整数组成的集合$A = \{a_0, a_1, \dots, a_{n-1}\}$和某个目标值$s$，求解$n$个取值为 0 或 1 的未知向量$X = \{x_0, x_1, \dots, x_{n-1}\}$，使得

$$x_0 a_0 + x_1 a_1 + \cdots + x_{n-1} a_{n-1} = s \qquad (1-1)$$

换句话说就是，从集合A中选取$k\ (0 \le k \le n)$个元素，组成的子集$I \in A$中所有元素的和为s，其中k被称为汉明重量（hamming weight）。

子集和问题被证明是 NP 完全问题[9]，有一些公钥密码算法[12][14][23][4]，都是基于子集和问题。对这些公钥密码系统的安全性分析，实际上就是对其底层难题——子集和问题的分析。在 $P \neq NP$ 的假设下，不存在多项式时间算法可以有效地对一般性子集和问题进行求解。

在本次挑战赛中，一共给出了 9 级子集和问题的挑战，每一级挑战都有 4 组数据，分别对应了不同的背包密度（背包密度$d = \frac{n}{\log_2 \max(A)}$）。第 1 级的维度为 40，每增加一级对应的维度会增加 20，一直到第 9 级维度增加到 200，汉明重量k也会相应地增加以保持$\frac{k}{n} \approx \frac{1}{8}$不变。每一级挑战 4 组数据的背包密度分别为固定的 0.8、0.9、1.0 和 1.1，因此随着维度的增加，集合A中元素的大小也在不断地增加。注意到汉明重量k与维度n的比值$q = \frac{k}{n}$实际上很小，并不是通常子集和问题的$\frac{1}{2}$，这为我们后续使用的 Zero-Forced 技巧提供了条件。此外，虽然每一级挑战都有 4 组数据，但只要解出其中的一组即可得分，由于较低背包密度的子集和问题更容易求解[13][18]，因此我们实际求解过程中使用的数据都是背包密度等于 0.8 的那组。

1985 年，Lagarias 和 Odlyzko 证明了背包密度$d < 0.645$的几乎所有子集和问题的解是某个格子的最短向量，从而将子集和问题的求解规约到格中最短向量的求解[13]。于 1982 年提出的 LLL 算法[15]则可以在多项式时间内找到一个近似最短向量，因此对于较低维度的子集和问题，可以尝试使用 LLL 算法对其进行求解，但对于较高维度的子集和问题使用 LLL 算法则很难找到解。1992 年，Coster 等人对该方法进一步修改优化[5]——把 Lagarias-Odlyzko 格子的最后一行更改为$\left(\frac{1}{2}, \frac{1}{2}, \dots, \frac{1}{2}, s\right)$，使得基于同样的假设下，能够将背包密度$d$的上限提高到 0.9408。2018 年，Yuan 等人通过对子集和问题中某些“无碰撞”特性的研究，将低汉明重量、高密度的子集和问题确定性地规约到格上的最短向量问题[17]。

低汉明重量、高密度的子集和问题可以确定性地转化为格中最短向量问题，但是如何对格中最短向量进行求解？这是一个 NP 困难问题[2]——最短向量问题（Shortest vector problem, SVP）。不可能找到一个能够解决一般性 SVP 的多项式时间算法，但是可以通过格基规约算法（Lattice reduction algorithm）对格子进行规约来找到近似最短向量。虽然格基规约算法并

不能保证找到的短向量中包含最短向量，但是在维度较低的情况下，格基规约算法是求解 SVP 的最好方法。当前主流的格基规约算法主要有 LLL 算法、BKZ 算法等等。前者是多项式时间算法，而后者的时间复杂度则取决于其块大小β。

2012 年，Schnorr 和 Shevchenko 等人使用 BKZ 算法对密度接近 1 的子集和问题进行了实验[21]，他们在 2.67GHz 的个人电脑上对随机的 50 个维度$n = 80$的子集和问题进行了测试，在经过总共 1 个多小时的随机 BKZ 规约后，成功地将这 50 个子集和问题解了出来。在本文中，我们将该方法称为 Randomized-BKZ 算法。基于该实验，我们相应地设计了自己的针对本次挑战赛的求解算法，并将实现好的代码分别放在个人 PC 和 CoCalc 在线计算平台上进行执行，成功地解出了维度为 40、60、80、100、120 和 140 的挑战。我们发现，每往后一级，程序运行所需的时间就呈指数级增长，在单核 2.6GHz 的个人电脑上，求解 140 维的挑战所需的运行时间已经长达数小时。考虑到 140 维以上的挑战可能需要耗费更多时间，我们开始寻找可以用来优化算法的地方。在 NTRU 格密码算法的密码分析报告[24]中，我们发现了一个可以用来降低维度的技巧——Zero-Forced Lattices，通过强行指定解向量中某些位置的值为 0，来将格中对应的那一行移除掉，从而达到降低维度的效果。随后，我们将 Zero-Forced 技巧运用于 Randomized-BKZ 算法中，并在国内某云服务器提供商处租用多核服务器，使用多进程并行技术运行我们的程序，成功地解出了 160 维和 180 维的挑战。至于 200 维的挑战，其所需的计算资源已经超过了我们所能承担的范围，我们只能求解出一个欧几里得范数为$\sqrt{247}$的解。

本文的布局如下：在第二章中，我们将对格理论进行简单的介绍，主要涉及到格的定义、格中的难题和格基规约算法；在第三章中，我们会对前人在子集和问题上的一些理论研究和实验进行回顾；第四章中将会阐述我们的工作，即本次挑战的整个求解过程；最后在第五章里我们将对本次挑战进行总结。

第二章　格理论

2.1 格的定义

令$v_1, \dots, v_n \in \mathbb{R}^m$为一组线性无关的向量，那么我们将$v_1, \dots, v_n$的所有整系数线性组合称作为由$v_1, \dots, v_n$所生成的格子（lattice）$L$。即，

$$L = \{ a_1 v_1 + \cdots + a_n v_n,\ a_1, \dots, a_n \in \mathbb{Z}\} \tag{2-1}$$

格子L可以用一组基（basis）来表示，$v_1, \dots, v_n$就是格子L的一组基。事实上，格子L有无数组基，只要是能够生成格子L的所有n个线性无关向量的集合都是格子L的基。

我们将n称为格子L的维度（dimension）。

我们可以将向量$v_1, \dots, v_n$写成坐标形式，

$$v_1 = (r_{11}, \dots, r_{1m}) \tag{2-2}$$

...

$$v_n = (r_{n1}, , \dots, r_{nm}) \tag{2-3}$$

那么格子L还可以用矩阵来表示，

$$L = (v_1, \dots, v_n) = \begin{pmatrix} r_{11} & \cdots & r_{1m} \\ \vdots & \ddots & \vdots \\ r_{n1} & \cdots & r_{nm} \end{pmatrix} \tag{2-4}$$

如果$n = m$，那么我们称格子L是满秩的。

2.2 格中的难题

在格中有几个基本计算性难题，其中最有名的是最短向量问题。

定义 2.1 最短向量问题 在格子L中找到一个最短的非零向量。即，找到一个非零向量$v_{shortest} \in L$，使得其欧几里得范数$\|v_{shortest}\|$最小。换言之，对于格子L中的所有非零向量v，有$\|v_{shortest}\| \leq \|v\|$.

1996 年，Ajtai 证明了，在随机规约下，SVP 被证明是 NP 困难的[2]，所以在一般性情形下，很难解决 SVP。在实际应用中，有一些公钥密码系统是基于 SVP 的[1]，但这些密码系统由于要考虑到设计上的一些原因，所以并不能完全使用一般性情形的困难问题来构建，其底层基于的数学难题实际上只是这些 NP 困难问题的一个特殊子类，而这种子类的某些特殊属性则令其可能比一般性情形更容易解决。本次挑战中的子集和问题就是一般性情形下的一种

特殊情况，因此将其规约到 SVP 问题后，就可以使用一些现有能有效解决 SVP 的方法去进行求解。

一般性 SVP 很难解决，在实际应用中，对于维度较高的情况，我们只能解决近似 SVP。下面我们给出 SVP 变形形式的定义。

定义 2.2 近似最短向量问题（apprSVP）令$\varphi(n)$是一个关于格子L维度n的函数，那么近似最短向量问题就是要找到一个向量$v_{appr} \in L$，使得

$$\|v_{appr}\| \leq \varphi(n)\|v_{shortest}\|. \qquad (2\text{-}5)$$

现在一些比较实用的格基规约算法都只能找到指数级倍的 apprSVP，目前还没有任何能够找到常数倍的 apprSVP 的多项式时间算法。在下一小节中，我们将对格基规约算法进行介绍。

事实上，格子L中的最短向量的长度取决于格子的维度n和$\det(L)$。有关于格中最短向量的长度，下面这个定理给出了一个上限：

定理 2.1 Hermite 定理 每一个格子L中都有一个非零向量$v \in L$满足

$$\|v\| \leq \sqrt{n}\det(L)^{\frac{1}{n}}. \qquad (2\text{-}6)$$

该定理可以很好地帮助我们来估计格中最短向量的长度。

2.3 格基规约算法

给定某个格子L的一组基，如果这组基是“优质基”，那么相关问题就较容易求解，但如果是“劣质基”则很难求解，需要对其进行格基规约才可求解。格基规约的目的就是将给定的一组劣质基转化为一组优质基——由长度较短且近似正交的向量所组成的一组基。在本小节中，我们将介绍两个格基规约算法：LLL 算法和 BKZ 算法。

2.3.1 LLL 算法

LLL 格基规约算法是由 Arjen Lenstra, Hendrik Lenstra 和 László Lovás 于 1982 年提出的一个多项式时间的格基规约算法[15]。LLL 算法提出的最初目的是为了给出一个能够在多项式时间内分解有理系数多项式的算法，后来，该算法在其他多个领域有广泛的应用。

简单来说，LLL 算法就是在格上找到一组 LLL-规约基。

定义 2.3 令格子L的一组基为$B = \{v_1, v_2, \dots, v_n\}$，并设$B^* = \{v_1^*, v_2^*, \dots, v_n^*\}$为基$B$所对应的 Gram-Schmidt 正交基，我们将基B称为 LLL-规约基当且仅当基B中所有的向量v_i均满足：

1) （Size 条件）对所有的$1 \leq j < i \leq n$，有

$$\left|\mu_{i,j}\right| = \frac{|v_i \cdot v_j^*|}{\left\|v_j^*\right\|^2} \leq \frac{1}{2}. \quad (2\text{-}7)$$

2) （Lovász 条件）对所有的$1 < i \leq n$，有

$$\|v_i^*\|^2 \geq \left(\frac{3}{4} - \mu_{i,i-1}^2\right)\|v_{i-1}^*\|^2. \quad (2\text{-}8)$$

LLL 算法的基本思想是，当给定一组基$\{v_1, v_2, \dots, v_n\}$时，可以很容易地得到另一组满足 Size 条件的基，然后需要根据 Lovász 条件来对向量进行排序，从而达到格基规约的目的。简略地说，可以利用从向量v_k中减去向量$v_1, v_2, \dots, v_{k-1}$整数倍的方法使得向量v_k逐渐减小。LLL 算法通过分步来实现此目的，而非一次性完成所有的步骤。执行 Size 条件规约后再检查向量是否满足 Lovász 条件，若条件满足，则可以得到一组顺序（近似）最优的向量；否则，就要对向量进行重新排序，然后进一步做 Size 条件规约。

Lenstra 等人给出的基本性结论就是，LLL-规约基是优质基且可以在多项式时间内被计算出。LLL 算法有如下性质：

性质 2.1 假设 LLL 算法在n维格子$L = \{v_1, \dots, v_n\}$上运行，令$B = \max\|v_i\|$，则在最多$O(n^2 \log n + n^2 \log B)$步运算后算法将会停止，且能够保证找到的最短向量$b_1$满足

$$\|b_1\| \leq 2^{\frac{n-1}{4}} \det(L)^{\frac{1}{n}}. \quad (2\text{-}9)$$

结合 Hermite 定理可知，LLL 算法可以将 apprSVP 解到大约$2^{n/2}$倍左右。

在构建格子时通常需要根据该性质来对格子进行调整，以使得 LLL 算法能够找到想要的那个向量。有关于 LLL 算法更深入的分析，请参考文后文献[8]。

2.3.2 BKZ 算法

1994 年，Schnorr 等人提出的 BKZ 格基规约算法[20]是目前最实用的格基规约算法之一，被广泛应用于格困难问题的求解和格密码体制的安全性评估。

BKZ 算法是 LLL 算法的一个变种算法，其基于 KZ 规约基，并借助 LLL 算法的思想对多个 KZ 规约基再进行规约，以达到找到短向量的目的。下面给出 KZ 规约基的定义：

定义 2.4 令格子L的一组基为$B = \{v_1, v_2, \dots, v_n\}$，并设$B^* = \{v_1^*, v_2^*, \dots, v_n^*\}$为基$B$所对应的 Gram-Schmidt 正交基，再令$\pi_i(v) = v - \sum_{j=1}^{i} \frac{v \cdot v_j^*}{\left\|v_j^*\right\|^2} v_j^*$ 为从L到$\mathbb{R}^n$的一个映射，我们将基B称为 KZ-规约基（Korkin-Zolotarev reduced basis）当且仅当基B中满足下面这三个条件：

1) v_1是格子L的最短向量。

2) 对所有的$2 \leq i \leq n$，都有$\pi_{i-1}(v_i)$是$\pi_{i-1}(L)$中的最短向量。

3) 对所有的$1 \le i \le j \le n$，都有

$$\left|\pi_{i-1}(v_i) \cdot \pi_{i-1}(v_j)\right| \le \frac{1}{2}\|\pi_{i-1}(v_i)\|^2. \qquad (2\text{-}10)$$

可见，KZ 规约基要比 LLL 规约基好很多，KZ 规约基的第一个向量总是格子L的最短向量。当然，KZ 规约基也要比 LLL 规约基难求得多，目前已知最好的求 KZ 规约基的算法的时间复杂度是指数级的。

BKZ 算法的基本思想是，将 LLL 算法中交换两个向量位置的那一步更换为对某个块进行 KZ 规约。在 LLL 算法中，需要不断地根据 Lovász 条件来交换v_{k-1}和v_k；而 BKZ 算法则转而对一块长度为β的子格$L_{[k,i+\beta-1]} = \{v_k, v_{k+1}, \dots, v_{k+\beta-1}\}$进行 KZ 规约。显然，当$\beta$变大，这一步所需要的耗时将会指数级增加，但带来的好处就是找到的短向量质量会变好。

在本文中，我们将块长度为β的 BKZ 算法称为 BKZ-β算法。BKZ-β算法有如下性质：

性质 2 假设 BKZ-β算法在n维格子L上运行，令c和d为两个小常数，则在最多$O\left(\beta^{c\beta} n^d\right)$步运算后算法将会停止，且能够保证找到的最短向量$v_1$满足

$$\|v_1\| \le \left(\frac{\beta}{\pi e}\right)^{\frac{n-1}{\beta-1}} \|v_{shortest}\|. \qquad (2\text{-}11)$$

上述性质说明 BKZ-β算法可以将 apprSVP 解到大约$\beta^{n/\beta}$倍之内，而 LLL 算法则将 apprSVP 解到约$2^{n/2}$倍；当$\beta = 2$时，BKZ-β算法即为 LLL 算法。可以看到，BKZ-β的运行时间关于n呈指数增长，因此 BKZ-β算法实际上并不是一个多项式时间算法，但在β较小的时候，其在较低维度上的运行时间还是可以接受的。

第三章　子集和问题

3.1 基于子集和问题的公钥密码算法

子集和问题在密码学中的应用，可以一直追溯到一开始最有名的两个公钥密码算法提议之一——Merkle-Hellman 公钥密码算法[12]，另外一个是 RSA 算法[19]。除 Merkle-Hellman 算法之外，基于子集和问题的公钥算法还有很多[14][23][4]。相较于 RSA 算法，基于子集和问题的公钥密码算法具有运算速度的特点，因此在公钥密码学早期的发展中得到了很多研究。

这些基于子集和问题的公钥密码算法大多都将一个权重集合$A = \{a_i : 1 \le i \le n\}$作为公共信息，并将明文信息用一个由 0-1 组成的向量$X = \{x_0, x_1, \dots, x_n\}$来表示，通过计算$s = \sum_{i=1}^{n} a_i x_i$来对明文信息的加密得到密文$s$。从$s$和$A$中解密出明文信息$X$，实际上就是一个子集和问题。为了能够解密，解密者必须掌握有关A的陷门信息，因此A的选择并不是完全随机的，而是为了满足一些条件而被特别构造出的。

但正是由于这些基于子集和问题的公钥密码体系内部存在一些特殊的构造，所以会有一些其他方法能够有效地求解这些特殊的子集和问题。

在 1982 年，Shamir 针对 Merkle-Hellman 公钥密码算法中的特殊结构，提出了一个能够在多项式时间内破解该密码算法的攻击算法[22]。自此之后，一些针对其他基于子集和问题的公钥密码算法的攻击方法也相继被提出，大部分基于子集和问题的公钥密码算法基本都被攻破，或者被证明是不实际的。

3.2 一般性子集和问题

这些公钥密码体系都是基于子集和问题的，因此对其底层难题——子集和问题的根本性分析将有助于我们对这些密码体系进行密码分析。

对于一般性的子集和问题，最简单的方法就是暴力穷举法，维度为n的子集和问题，该方法的时间复杂度为$O(2^n)$。近些年来，研究人员不断地提出新的针对子集和问题的求解算法[10][3]，子集和问题的求解算法的时间复杂度和空间复杂度在不断地降低。目前已知最好的通用算法是在 2020 年欧洲密码年会上由 Andre Esser 和 Alexander May 提出的启发式多项式空间碰撞查找算法[6]，该算法能够在时间复杂度为$O(2^{0.645n})$、空间复杂度为常数的条件下找到子集和问题的解，但其复杂度依旧是指数级别的。

在本次挑战赛中，如果我们使用这些通用算法，即使是最小维度$n = 40$，其所需要的巨大运算量也是我们无法接受的。

3.3 背包密度相关研究

在本次挑战赛中，我们观察到所有子集和挑战的背包密度有 4 种情况，因此，我们对与背包密度相关的研究进行了一定的调研。在本小节中，我们将回顾一些前人的研究结果。

3.3.1 低密度子集和问题

1985 年，Lagarias 和 Odlyzko 利用格规约算法提出了一个能够针对背包密度d小于 0.645 的攻击方法[13]。

Lagarias 和 Odlyzko 利用已知的$A = \{a_i: 1 \le i \le n\}$和$s$构造了一个矩阵

$$B = (b_1, b_2, \dots, b_n, b_{n+1}) = \begin{pmatrix} 1 & 0 & \cdots & 0 & -a_1 \\ 0 & 1 & \cdots & 0 & a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & -a_n \\ 0 & 0 & \cdots & 0 & s \end{pmatrix}. \tag{3-1}$$

那么向量$X' = (x_1, x_2, \dots, x_n, 0)$就是由矩阵$B$所表示的格子中的一个向量，

$$\sum_{i=1}^{n} x_i b_i + b_{n+1} = (x_1, x_2, \dots, x_n, 0) = X' \tag{3-2}$$

注意到该向量里只含有 0 或 1，因此其欧几里得范数实际上很小，

$$\|X'\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \le \sqrt{n}. \tag{3-3}$$

Lagarias 和 Odlyzko 证明了，在背包密度小于 0.645 的情况下，向量X'极大概率上就是由B所表示的格子中的最短向量。只要使用 LLL 算法对其进行格基规约，就有可能找到这个最短向量。这是第一个将低密度子集和问题转化为格中最短向量的方法。此外，Lagarias 和 Odlyzko 还使用该方法进行了一些实验分析，他们主要对较低维度（$n = 14, 20, 26, 30, 40, 50$）的子集和问题进行了测试，测试结果验证了他们的理论。此次实验的数据还表现出了另外一个特征：背包密度越低的子集和问题，LLL 算法就越容易找到最短向量。

1992 年，Coster 等人在 Lagarias 和 Odlyzko 工作的基础上，进行了优化——对 Lagarias 和 Odlyzko 的算法提出了两种修改方式，使得在相同的假设下，可以将背包密度的上限从 0.645 提高到 0.9408[5]。其中的一种修改方式是，将矩阵B的最后一行修改为$\left(\frac{1}{2}, \frac{1}{2}, \dots, \frac{1}{2}, s\right)$，并为最后一列乘上一个常数$N$，得到

$$B' = (b_1', b_2', \dots, b_n', b_{n+1}') = \begin{pmatrix} 1 & 0 & \cdots & 0 & Na_1 \\ 0 & 1 & \cdots & 0 & Na_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & Na_n \\ \frac{1}{2} & \frac{1}{2} & \cdots & \frac{1}{2} & Ns \end{pmatrix}. \tag{3-4}$$

向量$X'' = \left(x_1 - \frac{1}{2}, x_2 - \frac{1}{2}, \dots, x_n - \frac{1}{2}, 0\right)$也在该矩阵所表示的格子中，

$$\sum_{i=1}^{n} x_i b_i' - b_{n+1}' = \left(x_1 - \frac{1}{2}, x_2 - \frac{1}{2}, \ldots, x_n - \frac{1}{2}, 0\right) = X''. \quad (3\text{-}5)$$

该向量中每一项只能是$\frac{1}{2}$或$-\frac{1}{2}$，其欧几里得范数也很小，

$$\|X''\| = \sqrt{\left(x_1 - \frac{1}{2}\right)^2 + \left(x_2 - \frac{1}{2}\right)^2 + \cdots + \left(x_n - \frac{1}{2}\right)^2} \leq \frac{1}{2}\sqrt{n}. \quad (3\text{-}6)$$

且 Coster 等人证明了，在背包密度小于 0.9408 的情况下，向量X''也极大概率上是由B'所表示的格子中的最短向量。我们只需要利用某个理想化的 SVP 预言机去求出最短向量，就能得到原子集和问题的解。

值得一提的是，低背包密度的子集和问题不单单可以转化为 SVP，还可以转化为 CVP（Closest vector problem，最近向量问题）。在 2001 年，Nguyen 等人利用了一个与 Coster 等人差不多的格子，将低背包密度的子集和问题转化为了求解格中最近向量问题[11]。

在本次挑战赛中，每一级挑战的前 2 组数据的背包密度分别为 0.8 和 0.9。因此，按照 Coster 等人的方法，只要能够对 SVP 进行求解，我们就能解出所有的挑战。但是，SVP 是一个 NP 困难问题，在现实中，并不存在一个能够求解任意格子 SVP 的预言机。对于低维度的格子，可以通过格基规约算法来求出最短向量；但对于维度较高的格子，格基规约算法则无能为力。

3.3.2 低重量、高密度子集和问题

低密度的子集和问题可以转化为格中最短向量问题，只要能求解 SVP，就能求解这些子集和问题。为了防止这种攻击，有一些基于子集和问题的公钥密码算法转而通过利用降低汉明重量的方式来提高背包密度，Chor-Rivest 公钥密码算法[4]就是其中的一个。但时间证明，即使提高了背包密度，也依旧无法逃过这种攻击。

1998 年，Vaudenay 攻破了 Chor-Rivest 公钥密码算法[25]。

2018 年，Yuan 等人通过观察到这些基于子集和问题的公钥密码体系中的一些“抗碰撞”特性，成功地在不基于任何条件假设下，将子集和问题确定性地转化为格中最短向量问题[17]。

Yuan 等人证明了，如下格子的最短向量中即含有解向量（其中的k是子集和的汉明重量）：

$$B = (b_0, b_1, \ldots, b_{n-1}, b_n) = \begin{pmatrix} 1 & 0 & \cdots & 0 & Na_1 & N \\ 0 & 1 & \cdots & 0 & Na_2 & N \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & Na_n & N \\ 0 & 0 & \cdots & 0 & -Ns & -Nk \end{pmatrix}. \quad (3\text{-}7)$$

具体的证明请参考[17]。

可以发现，Yuan 等人所证明的是在子集和的汉明重量已知的情况下，这与本次挑战中给出了子集和问题的汉明重量一样；因此，我们在后面求解的过程中所使用的格子跟 Yuan 等人证明所用的基本相同。

3.4 相关实验

基于这些理论，许多研究人员也开始在计算机上实践这些理论的求解思路，实现了求解算法，并进行了相应的实验，得出了一些结论。在本小节中，我们讲回顾部分相关的实验。

1993 年，Schnorr 和 Euchner 提出了一个实际可行的浮点数版本的 LLL 算法、一个 LLL 算法的变种——“deep insertions”的 LLL 算法和一个实际可行的 BKZ 算法，分别利用这三种算法，对维度分别为 42、50、58 和 66 的子集和问题进行了实验测试[20]。实验结果表明，“deep insertions”的 LLL 算法表现要明显优于 LLL 算法，且略优于 BKZ-10 算法，但不及 BKZ-20 算法；BKZ 算法的成功率会随着块大小的增加而升高，但运行时间也会相应地增加。

2012 年，Schnorr 和 Shevchenko 继续利用随机 BKZ 规约来对背包密度接近于 1（该情形下的子集和问题被认为是较难的）的子集和问题进行求解实验[21]。Schnorr 和 Shevchenko 使用的是如下格子（该格子是采用的列向量表示法，而之前的都是行向量表示法）

$$B = (b_1, \dots, b_{n+1}) = \begin{pmatrix} 2 & \cdots & 0 & 1 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 2 & 1 \\ Na_1 & \cdots & 0 & -Ns \\ 0 & \cdots & 0 & 1 \\ N & \cdots & N & \frac{n}{2}N \end{pmatrix} \in \mathbb{Z}^{(n+3)(n+1)}. \qquad (3\text{-}8)$$

Schnorr 和 Shevchenko 设计了两种求解算法。第一种求解算法利用不带剪枝的 BKZ 算法对B迭代进行块大小为 2、4、8、16 和 32 的格基规约；第二种算法对B分别用逐一增加块大小（30, 31,…,80）的 BKZ 算法进行格基规约，并且根据对应的块大小选取最优的剪枝参数。实验结果表明，第一种求解算法十分高效，能够大幅度降低 BKZ 算法的运行时间，同时还能够提高成功率；BKZ 算法的剪枝参数对运行时间的影响很小；虽然第一种算法很高效，对于维度超过 32 的 BKZ 算法，第一种求解算法并没有多少优化。值得一提的是，虽然 Schnorr 和 Shevchenko 在这次实验中所使用的子集和问题维度只有 80，但却为我们的工作提供了很多参考价值。

总的来说，这些实验表明，通过将子集和问题转化为格中最短向量问题，然后再利用格基规约算法进行求解，是可行的。但是这种方法也并不是万能的，因为 SVP 也是一个 NP 困难问题，格基规约算法只能在维度较低的时候，可以作为一个近似的 SVP 预言机，但是当维度升高后，就很难解出 SVP 了。

第四章　求解算法

在本章中，我们将阐述此次挑战赛中我们设计的求解算法，并给出我们利用该求解算法对每一级挑战进行求解的具体过程。

4.1 Randomized-BKZ 算法

Yuan 等人证明了，某些情况下的子集和问题可以确定性地转化为下面这个格子的最短向量[17]：

$$B = (b_0, b_1, \dots, b_{n-1}, b_n) = \begin{pmatrix} 1 & 0 & \cdots & 0 & Na_1 & N \\ 0 & 1 & \cdots & 0 & Na_2 & N \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & Na_n & N \\ 0 & 0 & \cdots & 0 & -Ns & -Nk \end{pmatrix}. \quad (4\text{-}1)$$

观察到本次挑战赛中，每一个子集和问题挑战都给定了汉明重量k，因此，我们在求解过程中使用的格子即为该格子[1]。

受 2012 年 Schnorr 等人实验[21]的启发，我们设计了自己的求解算法——Randomized-BKZ 算法。

随后，我们利用开源数学软件 SageMath[28]对该算法进行了实现。由于 SageMath 是基于 Python 语言的，因此实现起来较为轻松，整个求解算法的实现仅用了百余行代码，代码看起来也很简洁。虽然 Python 的性能远不及 C/C++，但好在整个求解算法中最为耗时的 BKZ 算法，调用的是底层 C++实现的 fplll 库[27]中的 BKZ 算法[2]，因此用 SageMath 实现的代码在实际运行过程中仍然是十分高效的，后续的求解过程也验证了这一点。

在具体的实现过程中，我们 BKZ 算法使用的块大小为 22，即 BKZ-22。由于在较低维度下 BKZ 算法的运行时间还是很快的，因此我们没有使用 BKZ 算法的剪枝技术。关于格子中参数N的选取，只要能够满足$N > \sqrt{n}$即可，我们在代码实现中选择的是$\lceil \sqrt{n} \rceil$。在程序的执行过程中，我们发现上述格子中有很多也能满足条件且长度比较小的解，不过这些解中的每一项并不都是 0 或 1，这些解称之为寄生解（parasitic solutions）[8]。对于较低维度的格子，我们的求解算法能够较为成功地求出理想解（每一项都为 0 或 1，且 1 的个数等于汉明重量k），但对于中维度和高维度的格子，我们求出来的大多都是寄生解。因此，我们选择不断地随机调整格子的行向量排列，以使得 BKZ 算法每次都能对不同的输入进行格基规约，直到求出理想解。

[1] 在具体实现过程中，我们将最后一行中的后两项都修改为正的。
[2] SageMath 还提供了 NTL 库[29]的选项，但其默认用的是 fplll 库。

由于该求解算法很大程度上依赖于 BKZ 算法是否能够找到理想解，因此该求解算法实际上是一个概率算法，对其时间复杂度进行分析是一件比较困难的事情，不过我们对该算法在较低维度中的表现进行了较为全面的测试，具体的分析结果将在下一小节中给出。

表 1 Randomized-BKZ 算法

Algorithm 1： Randomized-BKZ 算法

Input 权重集合 $\mathrm{A}=\{a_1,a_2,\dots,a_n\}$

目标值 s

汉明重量 k

Process

1. 根据输入值构造矩阵B
2. **repeat**
3. 随机置换矩阵B中的每一行得到新的矩阵B′
4. 对矩阵B′进行 BKZ 规约得到$\mathrm{B}'_{\mathrm{BKZ}}$
5. **for** 矩阵$\mathrm{B}'_{\mathrm{BKZ}}$中的每一行$v=(v_1,v_2,\dots,v_n,v_{n+1},v_{n+2})$ **do**
6. **if** v满足$\sum_{i=1}^{n} v_i\,a_i=s$ **then**
7. 记录v
8. **if** $\|v\|^2=k$ **then**
9. **return** v
10. **end for**
11. **until** 达到停止条件

Output 子集和问题的解

我们将实现好的代码放在了 CoCal 在线计算平台[26]上执行，很轻松地就解出了$n=40,60,80,100,120$这前 5 级的挑战。对于 140 维的挑战，我们在等待了大概 2~3 小时后，也得到了答案。

有关前 6 级挑战的程序源代码和计算结果，请详见附录 8。

4.2 算法性能测试

我们在使用 Randomized-BKZ 算法计算出前 6 级挑战的结果后，意识到参数的选择十分重要。根据我们对 BKZ 算法理论的了解，以及实际操作得到的经验，我们发现，BKZ 算法的块长度β这一参数对规约运算需要花费的时间以及规约基的质量有着很大关系：对于一组数据，β越大，速度越慢、规约基的质量越高（找到最短向量的概率越高）。观察到，β与找到最短向量的概率正相关，与速度负相关。由此我们猜想，对于每一个维度，都会有一个最优β使得每次找到最短向量的平均效率最高。

我们对 40 维到 120 维的挑战进行了测试，选取不同的参数β，并分别计算对应的平均每次成功所需要的运行时间。表 2 给出了维度为 120 时的情况，其中，“平均每次时间”为执行一次 BKZ-β算法所需要花费的平均时间；“每次成功时间”表示找到理想解所需要花费的平均时间。更多相关测试数据，请详见附录 8。

表 2 120 维挑战不同块大小β所花费的时间[3]

n	Block Size	总次数	成功次数	成功率	平均每次时间	每次成功时间
120	8	100	1	0.010	6.393	639.250
	10	100	1	0.010	6.432	643.200
	12	100	2	0.020	6.420	321.000
	14	100	2	0.020	6.767	338.350
	16	100	3	0.028	7.436	265.579
	18	100	4	0.040	8.549	213.725
	20	100	6	0.058	9.948	173.000
	22	100	5	0.055	14.363	261.136
	24	100	12	0.120	51.682	430.683
	26	100	8	0.08	147.887	1848.588

根据表 2，我们以β为横坐标，平均每次时间为纵坐标作图（图 4-1）。

不难发现，BKZ-β算法的运行时间随β的增大而增大，并呈现出前期较缓，后期较陡的趋势。特别是当β超过 22 以后，就会呈现出极其陡峭的增长趋势，这与 Gama 等人的实验结果是相符的，BKZ-β算法在β为 20 和 25 之间，其运行时间会出现一个突然的暴涨[7]。出现这种情况的主要原因是，β的增加导致 BKZ-β算法内部一个子算法（枚举算法）的调用次数变多。因此，若想使得 BKZ-β算法高效运行，最好选取低于 24 的β。

[3] 用于测试的 PC 配置为：Intel Core i5-8250U@1.60GHz CPU、8GB RAM、Windows 10 操作系统。

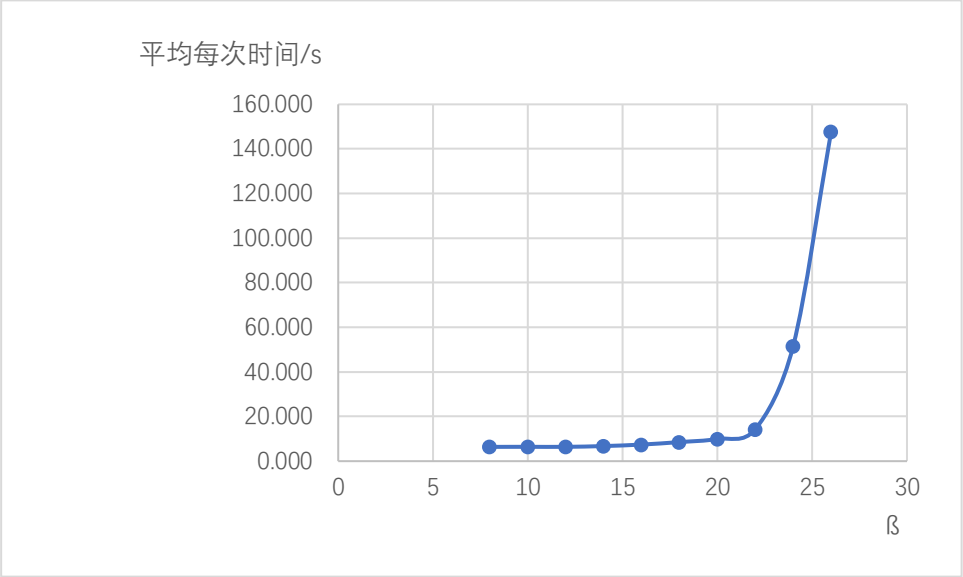

图 4-1 不同β对应的平均每次时间

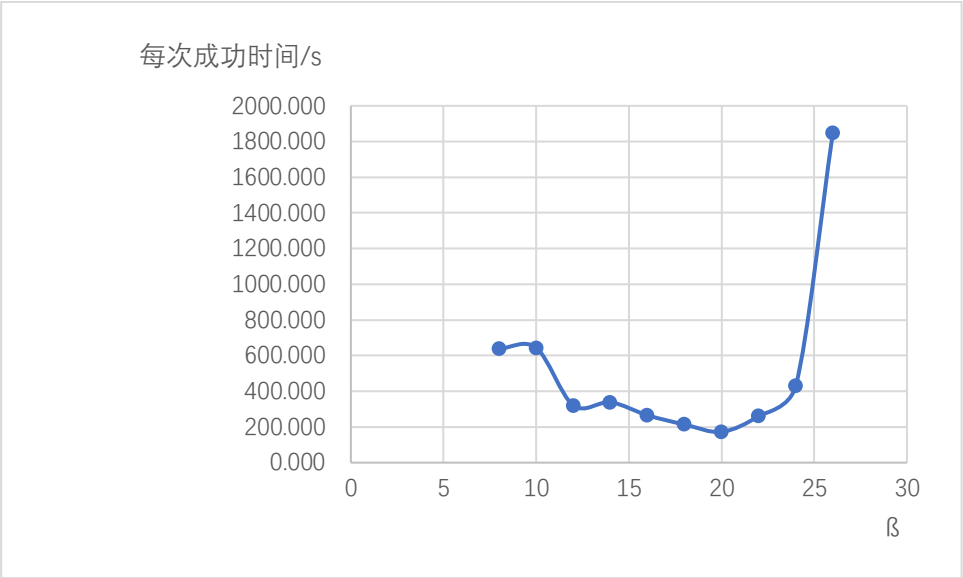

图 4-2 不同β对应的每次成功时间

对于维度为 120 的挑战，我们以β为横坐标，每次成功时间为纵坐标作图，如图 4-2 所示。

从图 4-2 中可以看出，对于 120 维来说，最优β选择为 20。此外，虽然参数β的增加能使得规约基的质量变好，但其带来的运行时间的增长是无法被弥补的。

考虑到要继续后 3 级挑战，以及每个维度对应的最优β选择在不断增加，因此我们对β为 22 的各个维度的每次成功时间进行了作图，以期望能为较高维度的情况进行预估。表 3 给出了相应的数据，按表中的数据作图 4-3。

表 3 $\beta = 22$时不同维度所对应的时间

n	Block Size	总次数	成功次数	成功率	平均每次时间	每次成功时间
40	22	1000	1000	1.000	0.563	0.563
60	22	1000	1000	1.000	2.503	2.503
80	22	1000	614	0.614	3.531	5.751
100	22	100	23	0.230	7.200	31.304
120	22	100	5	0.050	14.363	261.136

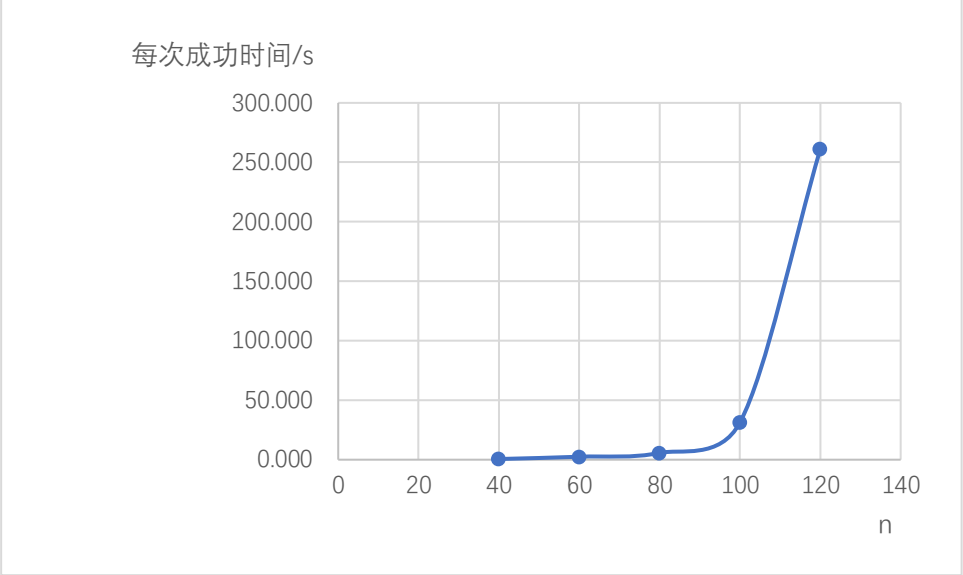

图 4-3 $\beta = 22$时不同维度对应的每次成功时间

从图-3 中可以出看出，每次成功时间与维度近乎是呈指数级相关的，且维度越高，每次成功时间增长得也越快。

4.3 Zero-Forced 技巧

前 6 级挑战，我们都能够使用第 1 小节中的 Randomized-BKZ 算法在较短时间内求解出来。但是当维度升高后，BKZ-β算法的成功率急剧下降，且每一轮的 BKZ-β算法的运行时间也呈指数级增长。根据上一小节中得出的结论，我们预估，维度为 160、180 和 200 的挑战所需要的算力资源将会很大。因此，我们尝试着去寻找了一些能够优化求解算法的方法。

我们观察到解向量中 0 的个数非常多，大概占据了整个解向量的$7/8$，由此我们联想到了 NTRU 密码分析中的 Zero-Forced Lattices[24]。NTRU 格的目标向量中也有很多地方为 0，Alexander May 利用该特征提出了 Zero-Run Lattices[16]，其主要思想是对 NTRU 格中的随机r列乘上一些比较大的数，以强制让诸如 LLL 算法这样的格基规约算法找到的向量中，这些

列所在的位置均为 0。后来，Silverman 干脆直接强制假设这r列的规约结果为 0，从而将这r列从格子中移除，以达到降维的目的，即 Zero-Forced Lattices[24]。

这种 Zero-Forced 技巧能够在一定程度上缩短格基规约算法的运行时间，因为维度降低了。但是这种技巧只有在降维后的格子中包含解向量才会成功，即随机选取强制为 0 的地方均命中。所以这种技巧实际上就是，通过多次运行的代价，来换取每次运行时间的减少。

在本次挑战赛的后 3 级挑战的求解过程中，我们运用到了这个 Zero-Force 技巧。主要是由于如下几个原因：解向量中的 0 的占比很多，每级挑战都有大概$7/8$的位置是 0；降维后，背包密度$d = \frac{n}{\log_2 \max(A)}$也会变小，能够提高 BKZ 算法的成功率；我们还未能使用 BKZ-β算法对较高维度（160 维及以上）的格子进行成功的求解，但对于 140 维及以下的情况我们有一定的经验和分析结果。

下面我们将对 Zero-Forced 技巧命中的概率进行分析。假设格子的维度为n，我们随机指定了r个位置为 0，那么此次指定命中的概率为：

$$\mathrm{P} = \frac{C_{n-k}^{r}}{C_{n}^{r}}$$

表 2 给出了 160、180 和 200 维的格子分别在$r = 10, 20, 30, 40, 50, 60$时命中的概率。

表 4 Zero-Forced 技巧命中的概率

n	k	r=10	r=20	r=30	r=40	r=50	r=60
160	20	25.226%	5.746%	1.163%	0.205%	0.031%	0.004%
180	22	26.190%	6.287%	1.367%	0.265%	0.045%	0.007%
200	25	25.447%	5.981%	1.286%	0.250%	0.043%	0.007%

可见使用 Zero-Forced 技巧并不会总能带来优化，当降维的维度过多时，反而会因为运行次数的增加而导致了整体的运行时间增多，导致“负优化”。

在实际求解过程中，我们对 160 维的挑战进行了降 20 维的 Zero-Forced 处理，并在 CoCalc 在线平台上新建了 5 个项目来跑这个程序，在大约 CPU 总耗时 30 小时后，成功地得到了理想解。

对于 180 维的挑战，我们仍然选择降维到 140 维的 Zero-Forced 处理，并在程序中加入了多进程功能，租用了某云服务器提供商的 160 核服务器[4]，以及某位学长无偿提供的 64 核服务器[5]，在大约 CPU 总耗时 1000 小时后，也成功地得到了理想解。

对于 200 维的挑战，我们发现如果继续降维到 140 维的话，其命中所需的运行次数高达数万次，反而增加了总的运行时间。因此，我们先对 200 维的数据在 150 维的格子上进行了 BKZ-β算法的测试，测试是在一台 32 核的服务器上运行的，32 个 CPU 核心同时对 150 维的格子执行 BKZ-β算法。一共测试了 6 次，每次测试成功所耗费的时间分别为：2123 秒、883 秒、897 秒、3491 秒、3521 秒。按照单核来计算的话，平均耗时为 58208 秒，大约 16 小时。如果我们降 50 维，我们大约需要运行 2000 多次才会有较大概率能够使得 Zero-Forced 命中，若想要解出 200 维的挑战，我们需要大概 4 万小时的 CPU 算力。4 万小时，即使是在 160 核的服务器上运行，也需要大约 250 小时的时间，而当前 160 核服务器的零售价为 20 元人民币/小时，因此想要求出这 200 维的挑战，我们预估将要花费 5000 元人民币。可能降 40 维能在一定程度上减少运行时间，但由于时间和经费的原因，我们没有对降 40 维的情况进行测试。尽管如此，我们还是利用现有的计算资源对 200 维的挑战进行了尝试，并解得了一个欧几里得范数为$\sqrt{247}$的近似解。

有关后 3 级挑战的源代码和计算结果，请详见附录 8。

[4] 160 核服务器的配置为：160 核 AMD EPYC™ Rome(2.6 GHz)、320GB 内存、Ubuntu 18.04 操作系统。
[5] 64 核服务器的配置为：64 核 Intel® Xeon® CPU E5-2686 v4@2.30GHz、256GB 内存、Ubuntu 18.04 操作系统。

第五章　结论

在本次挑战赛中，我们根据前人的理论，将子集和问题转化为格中最短向量问题，并在前人实验的基础上，利用 SageMath 软件实现了我们自己的求解算法，成功地解出了前 8 级挑战，以及给出了最后一级挑战的一个近似解。事实上，我们的算法还存在很多可以优化改善的地方，但由于时间和算力的原因，我们未能得到更好的结果。

通过这次挑战赛，我们可以感受到，子集和问题虽然在一般情形下是 NP 完全问题，但在特殊情况下还是比较容易求解的，很多基于子集和问题的公钥密码体系都被研究人员攻破了，或者被证明是不实际的，因此想要利用子集和问题来构建公钥密码算法还是比较困难的。但不可否认的是，子集和问题作为前期公钥密码体系的底层难题主要来源之一，其理论研究价值还是很大的。

第六章　参考文献

[1] Ajtai, Miklós, and Cynthia Dwork. "A public-key cryptosystem with worst-case/average-case equivalence." *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997.

[2] Ajtai, Miklós. "The shortest vector problem in L2 is NP-hard for randomized reductions." *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. 1998.

[3] Becker, Anja, Jean-Sébastien Coron, and Antoine Joux. "Improved generic algorithms for hard knapsacks." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2011.

[4] Chor, Benny, and Ronald L. Rivest. "A knapsack-type public key cryptosystem based on arithmetic in finite fields." *IEEE Transactions on Information Theory* 34.5 (1988): 901-909.

[5] Coster, Matthijs J., et al. "Improved low-density subset sum algorithms." *Computational complexity* 2.2 (1992): 111-128.

[6] Esser, Andre, and Alexander May. "Low Weight Discrete Logarithm and Subset Sum in $2^{0.65n}$ with Polynomial Memory." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Cham, 2020.

[7] Gama, Nicolas, and Phong Q. Nguyen. "Predicting lattice reduction." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2008.

[8] Galbraith, Steven D. *Mathematics of public key cryptography*. Cambridge University Press, 2012.

[9] Gary M R, Johnson D S. Computers and Intractability: A Guide to the Theory of NPCompleteness WH Freeman and Co[J]. New York, 1979: 226.

[10] Howgrave-Graham, Nick, and Antoine Joux. "New generic algorithms for hard knapsacks." *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Berlin, Heidelberg, 2010.

[11] Nguyen, Phong Q., and Jacques Stern. "The two faces of lattices in cryptology." *International Cryptography and Lattices Conference*. Springer, Berlin, Heidelberg, 2001.

[12] Merkle, Ralph, and Martin Hellman. "Hiding information and signatures in trapdoor knapsacks." *IEEE transactions on Information Theory* 24.5 (1978): 525-530.

[13] Lagarias, Jeffrey C., and Andrew M. Odlyzko. "Solving low-density subset sum problems." *Journal of the ACM (JACM)* 32.1 (1985): 229-246.

[14] Lempel, Abraham. "Cryptology in transition." *ACM Computing Surveys (CSUR)* 11.4 (1979): 285-303.

[15] Lenstra, Arjen K., Hendrik Willem Lenstra, and László Lovász. "Factoring polynomials with

rational coefficients." *Mathematische annalen* 261.ARTICLE (1982): 515-534.

[16] May, Alexander. "Cryptanalysis of NTRU." *preprint, February* (1999).

[17] Ping, Yuan, et al. "Deterministic lattice reduction on knapsacks with collision-free properties." *IET Information Security* 12.4 (2017): 375-380.

[18] Radziszowski, Stanislaw, and Donald Kreher. "Solving subset sum problems with the L^ 3 algorithm." *The Charles Babbage Research Centre: The Journal of Combinatorial Mathematics and Combinatorial Computing* 3 (1988).

[19] Rivest, Ronald L., Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems." *Communications of the ACM* 21.2 (1978): 120-126.

[20] Schnorr, Claus-Peter, and Martin Euchner. "Lattice basis reduction: Improved practical algorithms and solving subset sum problems." *Mathematical programming* 66.1-3 (1994): 181-199.

[21] Schnorr, Claus-Peter, and Taras Shevchenko. "Solving Subset Sum Problems of Densioty close to 1 by" randomized" BKZ-reduction." *IACR Cryptol. ePrint Arch.* 2012 (2012): 620.

[22] Shamir, Adi. "A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem." *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. IEEE, 1982.

[23] Shamir, Adi. "Embedding cryptographic trapdoors in arbitrary knapsack systems." *Information processing letters* 17.2 (1983): 77-79.

[24] Silverman, Joseph H. Dimension-reduced lattices, zero-forced lattices, and the NTRU public key cryptosystem. Vol. 13. NTRU Cryptosystems Technical Report, 1999.

[25] Vaudenay, Serge. "Cryptanalysis of the Chor-Rivest cryptosystem." *Annual International Cryptology Conference*. Springer, Berlin, Heidelberg, 1998.

[26] CoCalc – Collaborative Calculation and Data Science [online]. [ref. 2020-06-27]. Available: https://cocalc.com/

[27] The *FPLLL* development team. "*fplll*, a lattice reduction library." 2016. Available: https://github.com/fplll/fplll

[28] SageMath - Open-Source Mathematical Software System [online]. [ref. 2020-06-27]. Available: https://www.sagemath.org/

[29] Shoup, Victor. Number Theory C++ Library (NTL) version 11.4.3. Available: https://www.shoup.net/ntl/

第七章　谢辞

本文是在我们的指导老师张平老师的悉心指导下完成的，在本文的撰写过程中，张平老师给我们提供了很多宝贵的建议。在此，感谢张平老师对我们的指导和关怀。此外，我们还十分感谢为我们免费提供 64 核服务器的傅继晗学长，没有他的无偿资助，我们将无法求解出第 8 级挑战。

第八章　附录

8.1 各级挑战的计算结果

8.1.1 挑战难度一：40 维子集和问题

题目 1.1.　n=40，d=0.8，k=5

计算结果如下：

(0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)

8.1.2 挑战难度二：60 维子集和问题

题目 2.1.　n=60，d=0.8，k=7

计算结果如下：

(0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1）

8.1.3 挑战难度三：80 维子集和问题

题目 3.1.　n=80，d=0.8，k=10

计算结果如下：

(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)

8.1.4 挑战难度四：100 维子集和问题

题目 4.1.　n=100，d=0.8，k=12

计算结果如下：

(0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0)

8.1.5 挑战难度五：120 维子集和问题

题目 5.1. n=120，d=0.8，k=15

计算结果如下：

(0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0)

8.1.6 挑战难度六：140 维子集和问题

题目 6.1. n=140，d=0.8，k=17

计算结果如下：

(0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0)

8.1.7 挑战难度七：160 维子集和问题

题目 7.1. n=160，d=0.8，k=20

计算结果如下：

(0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

8.1.8 挑战难度八：180 维子集和问题

题目 8.1. n=180，d=0.8，k=22

计算结果如下：

(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)

8.1.9 挑战难度九：200 维子集和问题

题目 9.1. n=200，d=0.8，k=25

欧几里得范数为$\sqrt{\mathbf{247}}$的一个解：

(0, 0, 0, 2, 0, 0, 0, 0, 2, 1, 0, 0, 0, -1, 0, 1, 0, 0, 0, 0, -2, 0, 0, 0, 0, 1, 2, 1, 0, 2, 2, 0, -1, 0, 0, 1, 0, 1, -1, 1, 2, 0, -2, 0, 1, -1, 0, 3, 2, 0, 0, 1, 1, 0, 1, 0, 3, -1, 0, 0, 2, 0, 2, -2, -2, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, -1, 0, -1, 0, -1, 1, -3, 1, 0, -1, 0, 0, 0, 0, 1, -1, 0, 0, 4, 0, 0, 0, 0, 0, 3, 2, 0, 0, -2, 0, 0, 0, 1, 0, -3, -1, 0, 0, 0, 0, 0, 0, -1, 0, 3, 2, 2, 1, 0, 1, 0, 2, 0, 0, 0, 0, -1, 0, 1, 0, 0, 0, 0, 0, -3, 0, 0, 0, 0, 2, 1, -1, -1, -2, 1, -1, 0, -1, -2, 0, -1, 0, -1, 1, 0, -1, 0, -1, 0, 0, 3, -1, -1, 0, -3, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 0, -2, 0, 0, 0, 0, 3, 0, 0, 1, 0, -1, 0, 0, 1)

8.2 求解程序的源代码

运行环境：SageMath 9.1

8.2.1 前 6 级挑战

```
import re
import random

from functools import partial


LEVEL = [
    # n     k     s
    (40,    5,    2038015735279982),
    (60,    7,    13358837623210362761435 2),
    (80,    10,   380874853941482352011927516 3007),
    (100,   12,   214326896127070329471158810547025670819),
    (120,   15,   898756692547666853240820529622271845609989 0634),
    (140,   17,   464673946465200003242377774810378300764193429253004028),
    (160,   20,
172965030755792793515672577047987565372937948064781523455096 91),
    (180,   22,
757680399933334153634722410993690867586002859113929552263090603594409),
    (200,   25,
255248804816026878214495272730208647452717682177642673568083235409039 38
724725)
]

def readData(filename):
```

```
        """Read data from the file and parse data to python object.
        """
        A = []
        with open(filename, "r") as f:
                data = f.read().strip()
        for a in data.split("    "):
                A.append(int(a.split("=")[1]))
        return A

def check(sol, A, s):
        """Check whether *sol* is a solution to the subset-sum problem.
        """
        return sum(x*a for x, a in zip(sol, A)) == s

def solve(A, n, k, s, BS=22):
        N = ceil(sqrt(n)) # parameter used in the construction of lattice

        # 1. Construct the lattice
        #    (n+1) * (n+2)
        #    1 0 ... 0 a_0*N      N
        #    0 1 ... 0 a_1*N      N
        #    . . ... .    ...       .
        #    0 0 ... 1 a_n*N      N
        #    0 0 ... 0    s*N     k*N
        lat = []
        for i, a in enumerate(A):
                lat.append([1*(j == i) for j in range(n)] + [N*a] + [N])
        lat.append([0]*n + [N*s] + [k*N])

        # main loop
        itr = 0
        start_time = cputime()
        while True:
                itr += 1

                # 2. Randomly shuffle
                l = lat[::]
                shuffle(l)

                # 3. BKZ!!!
                m = matrix(ZZ, l)
                t_BKZ = cputime()
                m_BKZ = m.BKZ(block_size=BS)
                print(f"n={n} {itr} runs. BKZ running time: {cputime(t_BKZ):.3f}s")
```

```
            # 4. Check the result
            for i, row in enumerate(m_BKZ):
                if check(row, A, s):
                    if row.norm()^2 == k:
                        print(f"n={n} After {itr} runs. FIND SVP!!! {row}\n"
                              f"Single core time used: {cputime(start_time):.3f}s")
                        return True

def main():
    for n, k, s in LEVEL[:6]:
        A = readData(f"{n}.txt")
        solve(A, n, k, s)


if __name__ == "__main__":
    main()
```

8.2.2 后 3 级挑战

```
import re
import random
import logging
import multiprocessing as mp

from functools import partial


logging.basicConfig(level=logging.DEBUG)

LEVEL = [
    # n       k       s
    (40,     5,     2038015735279982),
    (60,     7,     13358837623210362761435 2),
    (80,     10,    380874853941482352011927516 3007),
    (100,    12,    214326896127070329471158810547025670819),
    (120,    15,    898756692547666853240820529622271845609989 0634),
    (140,    17,    464673946465200003242377774810378300764193429253004028),
    (160,    20,
172965030755792793515672577047987565372937948064781523455096 91),
    (180,    22,
757680399933334153634722410993690867586002859113929552263090603594409),
```

```
        (200,    25,
2552488048160268782144952727302086474527176821776426735680832354090393872
4725)
]

ZERO_FORCE = {
        # n      r
        160: 20,
        180: 40,
        200: 50,
}

def readData(filename):
        """Read data from the file and parse data to python object.
        """
        A = []
        with open(filename, "r") as f:
                data = f.read().strip()
        for a in data.split("    "):
                A.append(int(a.split("=")[1]))
        return A

def check(sol, A, s):
        """Check whether *sol* is a solution to the subset-sum problem.
        """
        return sum(x*a for x, a in zip(sol, A)) == s


small_vec = None
def solve(A, n, k, s, r, ID=None, BS=22):
        N = ceil(sqrt(n)) # parameter used in the construction of lattice
        rand = random.Random(x=ID) # seed

        indexes = set(range(n))
        global small_vec # may be process unsafe

        itr = 0
        total_time = 0.0
        while True:
                # 1. initalization
                t0 = cputime()
                itr += 1
                # print(f"[{ID}] n={n} Start... {itr}")
```

```
            # 2. Zero Force
            kick_out = set(sample(range(n), r))
            # (k+1) * (k+2)
            # 1 0 ... 0 a0*N      N
            # 0 1 ... 0 a1*N      N
            # . . ... . ...       .
            # 0 0 ... 1 a_k*N N
            # 0 0 ... 0 s*N       k*N
            new_set = [A[i] for i in indexes - kick_out]
            lat = []
            for i,a in enumerate(new_set):
                lat.append([1*(j==i) for j in range(n-r)] + [N*a] + [N])
            lat.append([0]*(n-r) + [N*s] + [k*N])

            # 3. Randomly shuffle
            shuffle(lat, random=rand.random)

            # 4. BKZ!!!
            m = matrix(ZZ, lat)
            t_BKZ = cputime()
            m_BKZ = m.BKZ(block_size=BS)
            print(f"[{ID}] n={n} {itr} runs. BKZ running time: {cputime(t_BKZ):.3f}s")

            # 5. Check the result
            # print(f"[{ID}] n={n} first vector norm: {m_BKZ[0].norm().n(digits=4)}")
            for i, row in enumerate(m_BKZ):
                if check(row, new_set, s) and row.norm()^2 < 300:
                    if small_vec == None:
                        small_vec = row
                    elif small_vec.norm() > row.norm():
                        small_vec = row
                        print(f"[{ID}] n={n} Good", i, row.norm()^2, row, kick_out)
                        if row.norm()^2 == k:
                            print(f"[{ID}] n={n} After {itr} runs. FIND SVP!!!\n"
                                  f"[{ID}] n={n} Single core time used:
{total_time}s")
                            return True

            # 6. log average time per iteration
            itr_time = cputime(t0)
            total_time += itr_time
            # average_time = float(total_time / itr)
            # print(f"[{ID}] n={n} average time per itr: {average_time:.3f}s")
```

```
def main():
    CPU_CORE_NUM = 160

    for n, k, s in LEVEL[-3:]:
        r = ZERO_FORCE[n]
        A = readData(f"{n}.txt")
        solve_n = partial(solve, A, n, k, s, r)
        with mp.Pool(CPU_CORE_NUM) as pool:
            reslist = pool.imap_unordered(solve_n, range(CPU_CORE_NUM))

            # terminate all processes once one process returns
            for res in reslist:
                if res:
                    pool.terminate()
                    break


if __name__ == "__main__":
    main()
```

8.3 算法性能测试结果

	n	Block Size	总次数	成功次数	成功率	平均每次时间	每次成功时间
Challenge 1	40	8	1000	1000	1.000	0.555	0.555
		10	1000	1000	1.000	0.562	0.562
		12	1000	1000	1.000	0.552	0.552
		14	1000	1000	1.000	0.551	0.551
		16	1000	1000	1.000	0.554	0.554
		18	1000	1000	1.000	0.555	0.555
		20	1000	1000	1.000	0.549	0.549
		22	1000	1000	1.000	0.563	0.563
		24	1000	1000	1.000	0.566	0.566
		26	1000	1000	1.000	0.589	0.589
		28	1000	1000	1.000	0.586	0.586

		30	1000	1000	1.000	0.606	0.606
		32	1000	1000	1.000	0.639	0.639
Challenge 2	60	8	1000	979	0.979	1.854	1.894
		10	1000	993	0.993	1.857	1.870
		12	1000	998	0.998	1.899	1.903
		14	1000	999	0.999	1.958	1.960
		16	1000	1000	1.000	2.036	2.036
		18	1000	1000	1.000	2.088	2.088
		20	1000	1000	1.000	2.226	2.226
		22	1000	1000	1.000	2.503	2.503
		24	1000	1000	1.000	2.799	2.799
		26	1000	1000	1.000	3.050	3.050
		28	1000	1000	1.000	3.518	3.518
		30	1000	1000	1.000	4.078	4.078
		32	1000	1000	1.000	5.420	5.420
Challenge 3	80	8	1000	282	0.282	2.359	8.365
		10	1000	322	0.322	2.334	7.248
		12	1000	409	0.409	2.464	6.024
		14	1000	413	0.413	2.464	5.966
		16	1000	490	0.490	2.566	5.237
		18	1000	535	0.535	2.650	4.953
		20	1000	594	0.594	2.929	4.931
		22	1000	614	0.614	3.531	5.751
		24	1000	743	0.743	5.180	6.972
		26	100	80	0.800	14.653	18.316
		28	100	99	0.990	20.541	20.748
		30	100	100	1.000	32.645	32.645
		32	100	100	1.000	47.236	47.236
Challenge 4	100	8	100	6	0.057	4.087	72.124
		10	100	6	0.060	4.377	72.950
		12	100	8	0.080	4.780	59.750
		14	100	13	0.130	4.681	36.008
		16	100	18	0.185	4.323	23.368

		18	100	21	0.215	5.305	24.674
		20	100	20	0.200	6.129	30.645
		22	100	23	0.230	7.200	31.304
		24	100	35	0.355	12.125	34.155
		26	100	41	0.410	34.950	85.244
		28	100	49	0.490	63.839	130.284
		30	100	51	0.500	287.433	574.866
		32	100	62	0.400	466.151	1165.378
Challenge 5	120	8	100	1	0.010	6.393	639.250
		10	100	1	0.010	6.432	643.200
		12	100	2	0.020	6.420	321.000
		14	100	2	0.020	6.767	338.350
		16	100	3	0.028	7.436	265.579
		18	100	4	0.040	8.549	213.725
		20	100	6	0.058	9.948	173.000
		22	100	5	0.055	14.363	261.136
		24	100	12	0.120	51.682	430.683
		26	100	8	0.08	147.887	1848.588