Description

Special case of a NP-complete problem.

Flag

WMCTF{1508363197285327134921463070467158008637697619610562046}

Writeup

The subset-sum problem is proved to be NP-complete, which means, under the assumption $\mathcal{P} \neq \mathcal{NP}$, no polynomial time algorithm exists to solve it.

However, some special cases of the subset-sum problem have trivial solutions. For instance, the subset-sum problem of a super increasing sequence is quite easy to solve, which the **Merkle-Hellman public key cryptosystem** (invented in 1978) is based on. Unfortunately, Shamir broke the (original) Merkle-Hellman public key cryptosystem in 1982. Three years later, Lagarias and Odlyzko successfully reduced the subset-sum problem with low density to the shortest vector problem in lattice. Although SVP is proved to be NP-hard for randomized reductions by Ajtai, seemingly harder to handle, we have some powerful tools to solve it in low dimension—the lattice reduction algorithms!

In this challenge, we are given an instance of the subset-sum problem with low density (d = 0.8). No doubt that this one belongs to the class that is easy to solve.

We are given a set of 180 elements $A = \{a_0, a_1, \cdots, a_{179}\}$, and a target number s, which is a sum of 160 of them.

To get the flag, we need to find a 0-1 vector

$$\mathbf{m} = \{m_0, m_1, \cdots, m_{179}\}, \quad m_i \in \{0, 1\}$$

such that

$$\sum_{i=0}^{n} m_i a_i = s.$$

For convenience, we reverse the "0"s and "1"s in the solution vector by calculating

$$s' = \sum_{i=0}^{n} m_i - s.$$

Then, the solution vector contains 20 "1"s and 160 "0"s.

From the special construction of this subset-sum instance, we can see that not only the density is small, the hamming weight (number of '1's in the solution vector) is small as well.

One naive solution is to enumerate all the possible combinations and check whether the choice is correct. Ways of choosing 20 elements out of 180 amount to

$$\binom{180}{20} = 1751421058575922480122926555 \approx 2^{88}.$$

This requires enormous computing power and seems to be infeasible in the current (2020).

However, it can be better solved by using lattice reduction. Since the hamming weight is fixed to be 160, we can construct a special lattice $\mathbf{L}$ generated by the following matrix:

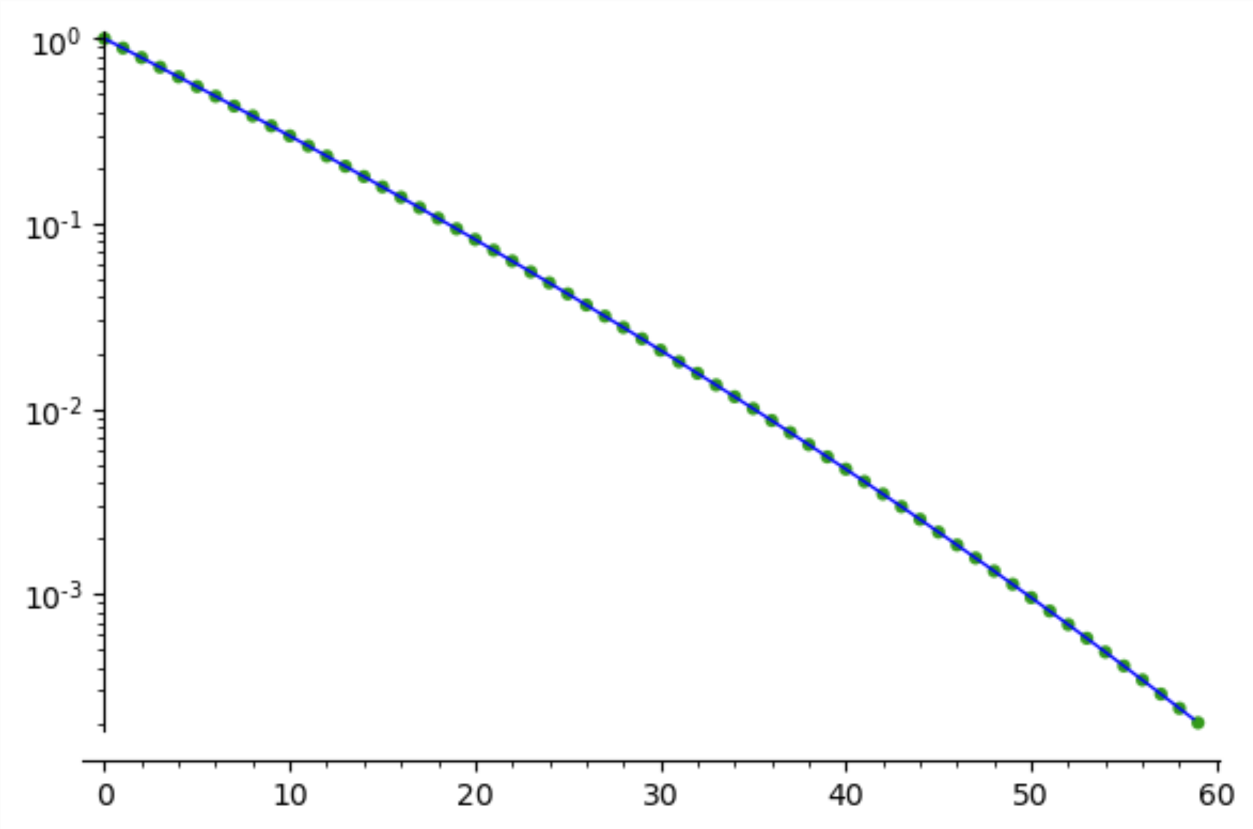
$$\mathbf{B} = \begin{bmatrix} 1 & 0 & \cdots & 0 & Na_0 & N \\ 0 & 1 & \cdots & 0 & Na_1 & N \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & Na_{179} & N \\ 0 & 0 & \cdots & 0 & -Ns & -Nk \end{bmatrix}$$

where k is the hamming weight, and N is a balance constant satisfying $N > \sqrt{n}$.

It's easy to show that the row vector $\mathbf{m}^* = \{m_0, m_1, \cdots, m_{179}, 0, 0\}$ is a point on the lattice and the Euclidean norm of $\mathbf{m}^*$ is quite small ($\|\mathbf{m}^*\| = \sqrt{20}$). The shortest vector in the lattice $\mathbf{L}$ is likely to be $\mathbf{m}^*$. Thus, we can try to find it by running lattice reduction algorithms. Note that if we randomly shuffle the row vectors of the lattice, lattice reduction algorithms will output different reduced results. So, we can keep trying to reduce different input row vectors until we find the shortest vector, i.e., the target vector m^*.

Although this instance is an easy-to-solve one, the dimension is not that low—n is 180. Simply continuously running the lattice reduction algorithms such as LLL, or BKZ could rarely success. We did algorithm performance test on lower dimensions. It took about several hours for a 140 dimension one to succeed, and as to 160 dimension, it cost 50+ hours. So, the running time of 180 dimension can be predicted as hundreds of hours.

Cleverly, we can reduce the dimension by forcing somewhere in $\mathbf{m}^*$ to be "0" and removing those from the lattice. In fact, this is a technique, called as "Zero-Forced Lattices", originating from the NTRU technical report. The probability of (randomly) forcing a right coordination is $\binom{160}{1}/\binom{180}{1} = 8/9 \approx 0.8889$. And if want to force r right coordinates, the probability is $\binom{160}{r}/\binom{180}{r}$. As can be seen from the following graph, the probability decreases exponentially as r increases linearly.



This indicates that, by using the "Zero-Forced Lattices" technique, we can gain much in search efficiency due to the fact that the lattices have smaller dimensions, there is also great significance loss of efficiency due for that we need try many times to get a dimension-reduced lattice that contains the target vector. Therefore, using this technique does not optimize much, and sometimes may even need more computing power than not using this technique. Anyway, there must exist an optimal choice of r to achieve the most optimization. However, it is a little bit difficult (or just because of laziness) to find such an optimal choice. Instead of doing some experiments to find the optimal choice, we choose r to be 40.

On the other hand, since we need to keep running lattice reduction algorithms in order to search for the target vector, this procedure can be easily speeded up linearly by multicore computing. That is, assuming that it requires 100 CPU hours on average to find the target vector, we can run this algorithm on 100 CPUs, and just need about 1 hour.

For the purpose of testing, we rented 4 cloud virtual machine instances from the cloud service provider Tencent, where each instance was equipped with a 32-core CPU and 64GB RAM and only cost 0.8RMB/hour. And then we ran our algorithm to search for the target vector. We were quite lucky, it took just about 300+ CPU hours to find the solution:

```
[240] n=180 32 runs. BKZ running time: 82.175s
[254] n=180 34 runs. BKZ running time: 29.683s
[232] n=180 33 runs. BKZ running time: 45.130s
[261] n=180 38 runs. BKZ running time: 32.356s
[251] n=180 30 runs. BKZ running time: 55.224s
[249] n=180 36 runs. BKZ running time: 32.321s
[248] n=180 30 runs. BKZ running time: 59.355s
[236] n=180 30 runs. BKZ running time: 14.466s
[256] n=180 34 runs. BKZ running time: 59.053s
[242] n=180 33 runs. BKZ running time: 18.093s
[253] n=180 33 runs. BKZ running time: 53.770s
[233] n=180 36 runs. BKZ running time: 17.921s
[257] n=180 36 runs. BKZ running time: 29.799s
[245] n=180 31 runs. BKZ running time: 55.053s
[245] n=180 Good 0 236 (-3, 0, 1, 2, -1, 1, 0, 0, 2, 0, 0, 1, 0, 2, 3, 0, 2, -3, -1, 2, 0, -2, 1, -1, 0, 1, 1, 0, 0, -1,
 0, 0, 2, 1, 1, 0, 0, 2, 1, 0, 0, 2, 2, -1, -2, -1, 0, 1, -1, -2, -3, 2, -1, -1, 0, 0, 0, -2, 0, 2, 0, 0, -2, 0, 0, -1,
-1, 0, -1, -1, 1, 3, 0, -1, 0, -1, 1, 1, -1, -1, 0, -1, 1, 1, 2, -1, 1, -1, -1, 0, 0, 1, 0, -1, 0, 0, 0, -1, -2, 0, -3,
0, 1, -1, 0, 1, 0, 0, 1, 1, 0, 1, 0, -1, 3, 0, 0, 2, 1, 0, 1, 0, 0, 2, 2, 0, 1, 2, -3, 0, 0, 0, 3, 2, 0, -2, 0, 3, 0, 0,
 0, 0) {0, 128, 2, 131, 1, 5, 134, 7, 136, 10, 147, 149, 25, 28, 157, 159, 162, 36, 40, 169, 170, 175, 53, 60, 66, 70, 7
5, 83, 85, 86, 90, 91, 98, 104, 106, 108, 113, 121, 122, 125}
[237] n=180 31 runs. BKZ running time: 38.172s
[246] n=180 34 runs. BKZ running time: 46.138s
[260] n=180 30 runs. BKZ running time: 56.161s
[260] n=180 Good 0 20 (0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0
, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0) {131, 3, 8, 137, 9, 139, 136, 1
6, 144, 19, 21, 22, 151, 154, 156, 30, 32, 171, 43, 47, 50, 52, 53, 60, 68, 69, 70, 72, 74, 75, 92, 98, 99, 102, 104, 10
5, 106, 123, 126, 127}
[260] n=180 After 30 runs. FIND SVP!!!
[260] n=180 Single core time used: 1065.91568500000s
```

> In fact, we had run in total 30000+ times of lattice reduction algorithms before we found this solution.

The source code is shown as below (written in SageMath 9.1):

```python
# !/usr/bin/env sage
import random
import multiprocessing as mp

from json import load
from functools import partial


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, 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))
    small_vec = None

    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 = 32
```

```python

    k, n, d = 160, 180, 0.8
    s, A = load(open("data", "r"))
    r = 40 # ZERO FORCE

    new_k = n - k
    new_s = sum(A) - s
    solve_n = partial(solve, A, n, new_k, new_s, r)
    with mp.Pool(CPU_CORE_NUM) as pool:
        reslist = pool.imap_unordered(solve_n, range(200, 200+CPU_CORE_NUM))

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


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

PS: The `BKZ` algorithm in SageMath is underlying using the implementation of **fplll**. Since most of the time of the program is spent on running the `BKZ` algorithm, there is little performance loss by using SageMath, which is based on Python.