Summary

In hard knapsack problem, all known polynomial time algorithms fail to break. There are two algorthms called Schroeppel-Shamir and Howgrave-GrahamJoux (HGJ) that can solve hard knapsacks in exponential time. Although, in paper [17], balanced knapsack problem has been fully discussed, it does not cover unbalanced case too much. Hence, in this dissertation, I will focus on unbalanced case by analyzing complexity. To support my analysis, I will implement experiment to demonstrate its correctness. Moreover, some extension and improvement strategies are also discussed.

In this dissertation, several works has been done:

1. Analyze the time and memory complexity of priority queue and modular version Schroeppel-Shamir algorithm. See section 5.1.1 and 6.1

2. Discuss the time complexity if using early abort strategy. See page 30 and 39

2. Point out an error in original priority queue 4-way merge algorithm and come up with an modified version of algorithm. See page 31.

3. Implement experiments both modified and original version to see running time and correctness. See page 33.

4. Analyze and implement experiment to show that the best value of parameter M in modular 4-way merge algorithm is the same as the input list size. See page 40.

5. Present parallel version of modular Schroeppel-Shamir algorithm and simple HGJ algorithm. See page 37 and 47.

6. Fully discuss the complexity of HGJ. See section 7.1. Implement experiments to demonstrate the correctness of analysis. See page 51.

Contents

1 Introduction

0-1 knapsack problem which is also called subset sum problem (SSP) is a special kind of knapsack problem. 0-1 knapsack problem as a famous NP-hard problem interests both theoreticians and practitioners. It also has numerous applications. To cite one example, to allocate task to parallel independent machines with setup times gives rise to a 01 integer program. In this program, its coefficient reduction problem can be seen as a subset sum problem [1]. More importantly, since this problem is easy to compute in one direction but hard to calculate in inverse direction, subset sum problem can be a very useful cryptographic tool. In fact, a number of cryptosystems are constructed on the basis of hardness of subset sum problem, such as Merkle-Hellman public key cryptosystem [4], Impagliazzo and Naor's cryptographic schemes [8] which is as secure as the subset sum problem and more recently, Gentry's fully homomorphic scheme [22] [2] [25]. NTRU cryptosystem can also be reduced to an unblanced, approximate modular vector knapsack.

1.1 Problem Description

In the broad sense, knapsack problem can be regarded as an optimization problem. Namely, under the constraint of capability of the knapsack, to find a subset from given items whose value or objective function is the greatest. However, in cryptography, to find the greatest value or a value close to the target sum is meaningless. Cryptographers care which subset sums up to exact given target sum. In this paper, the definition in the cryptography will be used.

More precisely, the subset sum problem or 0-1 knapsack problem is that

Given 1. a vector A which consist of a set of positive and pairwise different integers, i.e.

$$A = (a_1, a_2, \dots, a_n), a_i \neq a_j, \text{ if } i \neq j$$

In this paper, we call this vector A as **knapsack vector** and any a_i as **elements** in knapsack vector.

And given 2. A positive integer s, called the target or target sum.

Then the goal of the problem is to find a subset $A' = (a'_1, \dots, a'_l)$ of A which satisfies

$$s = \sum_{a'_1,\dots,a'_l \in A'} a'_i \tag{1}$$

We define the subset we are looking for (i.e. satisfying Equation (1)) as **answer subset**.

Or equivalently, if we define another n-dimensional vector $X = (x_1, x_2, \ldots, x_n)$ with values of x_i in $\{0, 1\}$, subset sum problem then converts into finding a vector X satisfying

$$s = AX = \sum_{i=1}^{n} x_i a_i, \quad x_i \in \{0, 1\} \quad \text{for all} \quad i \tag{2}$$

In other words, x_i means whether a_i is in the answer subset which sum up to target s. If x_i equal to 0, a_i is in the subset. If 1, a_i is not in the subset.

We call vector X as **indices vector**. Similar with the difinition of answer subset, we call indices vector represents the answer subset which sum up to target s as **answer indices vector** or **solution**.

We call a given problem with a knapsack vector and a corresponding target sum as a **knapsack**. Please note that, even though the elements in the knapsack vector and target sum of two knapsacks are same, if the elements arrange in different order, they are different knapsacks.

If we consider the target sum s as the capability of a knapsack and a_i as volume of an items, then subset sum problem can be considered as given a list of items how to fill a knapsack exactly with some collection of those items. That is why we call it knapsack problem.

1.2 Classification of Subset Sum Problem

The subset sum problem is in the complexity class NP. There are two forms of knapsack problems. One is decision knapsack problem, where one should determine whether there exists an answer subset for a given knapsack. The second form is the computational knapsack problem. One should try to find out the solution. The decision problem and the computational problem are equivalent within polynomial factors. If decision knapsack problem can be solved efficiently by an algorithm or oracle, one can also compute computational knapsack problem by calling n round of the oracle- i.e. if we want to know the answer subset of a n-dimensional knapsack $A = (a_1, a_2, \ldots, a_n)$, we can ask the oracle whether there exists a solution of the (n-1)-dimensional subknapsack whose knapsack vector is $A' = (a_1, a_2, \ldots, a_{n-1})$. If there exists, x_n equals to 1, else x_n equals to 0. If one repeats asking existence from $n-1$ to 0, one can find all the value of x_i. The decision problem is NP-complete [3] and the corresponding computational problem is NP-hard. Since, general speaking, there is no polynomial time algorithm for the problem in the complexity class NP, repeating oracle n-times will not effect the time comlexity.

According to Impagliazzo and Naor [8], knapsack problems can also be classified in terms of their density, which is defined as:

$$d = \frac{n}{log_2(max_i\, a_i)}$$

When $d < 1$, usually, there is only a unique solution for a given target s, therefore, this set of integers can be used for encryption. When $d > 1$, most target sum s will have many preimages which makes these sets can be used for hashing scheme. It is proved by Impagliazzo citeImpagliazzo, when the density is close to 1, it is the hardest case of subset sum problem. So this case is informally called hard knapsack problem.

1.3 Subset sum based cryptosystem

Subset sum based cryptosystems are one of the earlist public key scheme, which was believed to be very hard to break, since its security relies on the hardness of a classical NP-complete problem. But in fact, it turns out that almost all subset sum based system is shown to be insecure.

The basic idea of subset sum based cryptosystems is that there are two sets of integers. In one set of integers, subset sum problem can be solved easily, whereas, in the other set, it can be solved hardly. There is a trap door behind these two sets, making these two sets are equivalent to each other when solving subset sum problem. Certainly, no one except decryptor owning the secret key can find out the link between these two sets. So when encrypting, one can use the hard set as public key and get the ciphertext in the form of sum s. When the decryptor gets the ciphertext, he can use easy set as secret key to recover the message from target sum s.

The first and most famous subset sum based scheme was introduced by Merkle and Hellman [4] in 1978. Merkle-Hellman's public key cryptosystem tries to hide superincreasing knapsack problem as an easy one into general knapsack problem. But unfortunately, a polynomial time attack method come up from Shamir [5] in the spring of 1982 announced the fall of singly-iterated Merkle-Hellman cryptosystem using lattice reduction. After that, a number of other knapsack based cryptosystems were broken due to lattice reduction, too. In addition to the attacks on specific knapsack systems, a low-density attack introduced by Lagarias and Odlyzko [6] and later improved by Coster [7] provide a very ideal choice to break subset sum based system. In fact, even Lagarias-Odlyzko low-density attack is solving shortest vector problem (SVP) in lattices which is NP-hard for randomized reduction shown by Ajtai [9]. But due to lattices reduction algorithm such as LLL [10] or the BKZ algorithm of Schnorr [11], in practice, Lagarias-Odlyzko low-density attack works very well when the density $d < 0.64$. After that, the knapsack cryptosystems with density $d < 0.94$ are proved to be insecure after the attack from [7] [12] [13]. On the other hand, a similar attack proposed by Joux and Granboulan [14] which tries to find collision is efficient (mildly exponential time $O(2^{n/1000})$) to break knapsack based hash-function with $d > 1$.

However, there is no a quite efficient way to break the system whose density close to 1 which is so-called hard knapsack problem. In fact, Impagliazzo and Naor [8] proved that the density 1 is the hardest knapsack problem. Before 2009, the known best algorithm for hard knapsack problem was an brute force algorithm called Schroeppel and Shamir algorithm [15] [16]. Schroeppel and Shamir algorithm is improved on the base of birthday algorithm. Its running time is $O(n * 2^{n/2})$ and memory in bits is $O(n * 2^{n/4})$. If using $\widetilde{O}$ representation (i.e. $O(g(n) * log(g(n))^i)$ is represented as $\widetilde{O}(g(n))$, since $g(n)$ is much larger than $log(g(n))^i$ thus dominates the running time), the time and memory complexity of Schroeppel and Shamir algorithm is $\widetilde{O}(2^{n/2})$ and $\widetilde{O}(2^{n/4})$ respectively. In this dissertation, we will always use this notation.

At Eurocrypt 2010, Howgrave-Graham and Joux introduced a more efficient algorithm [17] for hard knapsacks. While in Schroeppel-Shamir Algorithm, two lists are constructed implicitly so as to searching collisions in two lists without overlap, Howgrave-GrahamJoux algorithm (HGJ algorithm) divides original knapsack problem into several subknapsacks with overlap, which allows more freedom. Thus, it lowers the running time down to $\widetilde{O}(2^{0.3113n})$ and space requirement down to $\widetilde{O}(2^{0.256n})$ for almost all knapsacks of density 1.

1.4 Security of subset sum based cryptosystem

Since almost all subset sum based cryptosystems have been shown to be insecure, the major question about subset sum based cryptosystems has always concerned their security. From a high level to analyze subset sum based cryptosystems, there is a doubt asked frequently which also applies to the RSA as well, "What if $P = NP$, somebody comes up with a wonderful new algorithm that solves all problems in NP efficiently?" Even if we put this cliche aside and admit that $P \neq NP$, why most instances of the knapsack problem can be solved easily? In fact, the so-called NP-completeness problem deals with the worst-case situation, which only means that there is no a efficient way to be bound to find the answer. In other words, NP-completeness is independent to the truth that many NP-complete problems are easy to solve on average [19]. Moreover, most these cryptosystems are not pure subset sum problem, since there is a trap door behind the public hard knapsack problem and the private easy knapsack problem.

Besides the general doubt that worries public key cryptosystem metioned above, another cause for suspicion is caused by a result of Brassard [20]. On the very abstract level, it says that if the problem of breaking a cryptosystem is NP-hard then $NP = CoNP$, where $CoNP$ is the complement of NP. That means when facing a cryptosystem like Merkle-Hellman scheme, the only way to break is to come up with an algorithm which will solve any subset sum problem. However, if $NP \neq CoNP$, there should be an attack on these cryptosystems which takes much less time than the algorithms that break subset sum problem.

There is a more specific suspicion about the security of subset sum based cryptosystems — the linearity of such schemes, namely, $\sum_{i=1}^{n} x_i a_i + \sum_{i=1}^{n} y_i a_i = \sum_{i=1}^{n}(x_i + y_i)a_i$. For instance, supposing that not all the a_i are even, by observing the last bit of ciphertext (i.e. the target sum s) we can know whether there is at least an even integer in the plaintext subset. Although, this is such a bit of information about plaintext that it does not even reavel a single bit of plaintext and there is no attack found is based on this linearity, it may lead to defect in its security since linearity in cryptosystems is known to be dangerous.

Anyway,though the security of the subset sum based system are often suspected, subset sum problem is a NP-hard problem after all. We have reason to believe that subset sum based cryptosystems have the potential to be secure if it is designed perfectly.

1.5 Advantages of subset sum based cryptosystem

The main attractive feature of knapsack cryptosystems including Merkle-Hellman knapsack cryptosystems is the speed, which is often the drawback of most public key scyptosystems.. Symmetric encryption system, such as the Data Encryption Standard (DES), When implemented on the dedicated chip, its encryption speed can be up to tens of milions of bits per second. Even if it is only used in the appropriate size of the software, it can still encrypts at the rate of 100000 bits per second. The RSA system is often criticized for its low speed. In the case moduling at a size about 500 bits, even the speed of the fastest dedicated chip can only be 10^4 or $2 * 10^4$ bits per second. If implemented by software, the speed is as low as 100 bits per second. In other words, the RSA encryption is 100 or 1000 times slower than that of traditional symmetric encryption.

The subset sum based cryptosystems seem to provide a possibility with a higher speed compared with other public key systems. Taking Merkle-Hellman subset sum cryptosystem as an example, with parameter $n \approx 100$ (which is an appropriate value, Merkle-Hellman knapsack cryptosystem seemed to be 100 times faster than RSA with modulus of about 500 bits [21] at the time it was introduced. Either used in software or chips, although the speed of subset sum based scheme still does not exceed the speed of symmetric encryption system, it is much faster than general public key scheme. This is because the number (200 bits) in the Merkle-Hellman knapsack cryptosystem is smaller than that in RSA (500 bits) and moreover, there is only one modular multiplication in Merkle-Hellman knapsack cryptosystem. Though, there is a slight disadvantage in that twice the communication capacity is needed which means the size of plaintext will be expanded twice in ciphertext and the size of the public key is larger (when $n \approx 100$, the key size for Merkle-Hellman knapsack cryptosystem is 20,000, compared with 1000 bits for RSA with modulus of about 500 bits).

1.6 Variant of Knapsack Problem

1.6.1 Modular knapsack problem

Similar with the original definition of knapsack problem, modular knapsack problem is given a modulus M, knapsack vector $A = (a_1, a_2, \ldots, a_n)$ where $a_i \in \{0, M-1\}$ and a target sum s, looking for answer subset. Namely, find a indices vector $X = (x_1, x_2, \ldots, x_n)$ that can be written as

$$s \bmod M = AX \bmod M = \sum_{i=1}^{n} x_i a_i \bmod M$$

Actually, the knapsack problem defined over integers is equivalent to modular knapsack problem, in exponential time algorithm. Any algorithm that can solve integer knapsack problem can be used to solve the modular knapsack problem, and vice versa.

Obviously ,in one direction, given an algorithm that can solve modular knapsack problem, when modulus M is large enough, modular knapsack problem will be exactly the same as integer knapsack problem. More precisely, to solve integer knapsack problem, one input modular knapsack algorithm with modulus M, original knapsack vector and original target sum, where M satisfies $M \geq \max(s, \sum_{i=1}^{n} a_i) + 1$. Thus, integer knapsack problem transforms into a modular knapsack problem with a huge M which leads to whether modular M will not effect the numbers in the problem.

In the other direction, assuming that knapsack vector A and the target sum M is represented in the range $[0, M-1]$, then all the possible sums are in the range of $[0, nM-1]$. Hence, in modular knapsack problem, if all the possible target sum is representd in integer, it will be $s, s+M, \ldots, s+(n-1)M$. If given an agorithm that can solve integer knapsack problem, by repeating calling integer knapsack algorithm n times with target sum $s, s+M, \ldots, s+(n-1)M$, modular knapsack problem will be solved. As before, n times iteration will not effect the time complexity of a exponential time algorithm.

1.6.2 Random knapsacks

Given the length of knapsack vector and prescribed density D, random knapsack problem is that all n elements in knapsack vector A is generated in random and the density of knapsack basically meet the prescribed density D. Random knapsack problem is constructed as follow:

Algorithm 1:Random kanpsacks problem generator

```
Input:
n: the number of elements in knapsacks
D: prescribed density

Create vector a and vector x
for i from 0 to n-1
   pick a[i] uniformly random in the range of $[1, \lfloor 2^{n/D} \rfloor]$
   pick x[i] uniformly random in the range of $[0, 1]$
   if x[i]==1
      s=s+a[i]
   end if
end for

output:
a: knapsack vector
x: answer indices vector
s: target sum
```

Note that there is some probability that the actual density d of knapsack is bigger than prescribed density D. However, as n increases, the probability that d and D are different will be gradually reduced. Finally, when n tends to infinity, the two values become arbitrarily close with overwhelming probability. When the problem is generated by the algorithm described above, Coster et al. solved all random knapsack problem with prescribed desity $D < 0.94$ [7] using lattice based algorithm.

1.6.3 Unbalanced knapsack problem

The hamming weight (abbreviated as weight) stands for how many values in a vector is non-zero. The hamming weight of indices vector is often denoted as l. Formally, hamming weight of indices vector l is defined as $l = \sum_{i=1}^{n} x_i$. Thus, the hamming weight of answer indices vector in random knapsack problem may be the arbitrary value in the range of $[0, n]$. But when n is big enough, we generally believe that l is $n/2$. For the reason that the hamming weight l greatly effects the density that can be applied by lattice reduction attack [7] [6], it is natrual to come up with the idea of knapsack problem with different hamming weight which is defined as α-unbalanced knapsack problem.The value of α is defined as $\alpha = l/n$. α-unbalanced knapsack problem can be constructed as follow:

Algorithm 2:α-unbalanced kanpsacks problem generator

Input:
n: the number of elements in knapsacks
D: prescribed density
α:proportion of the value that equals to 1, in answer indices vector

```
Create vector a and vector x
let l be $\lfloor \alpha n \rfloor$
while the weight of x doen't equal to l
  generate a random a
for i from 0 to n-1
  pick a[i] uniformly random in the range of $[1, \lfloor 2^{n/D} \rfloor]$
  if x[i]==1
    s=s+a[i]
  end if
end for
```

output:
a: knapsack vector
x: answer indices vector
s: target sum

Even though a well balanced knapsack with exactly half zeros and ones in knapsack vector, we can use the notation of α-unbalanced knapsack as well and call it 1/2–unbalanced knapsack.

1.6.4 Complementary knapsacks

Given a knapsack problem with knapsack vector $A = (a_1, a_2, \ldots, a_n)$ and target sum s, trying to find answer indices vector $X = (x_1, x_2, \ldots, x_n)$ is equivalent to solve its complementary knapsack problem which is

given:
  original knapsack vector $A = (a_1, a_2, \ldots, a_n)$
  new target sum $s' = \sum_{i=1}^{n} a_i - s$
to find:
  answer indices vector $X' = (x'_1, x'_2, \ldots, x'_n)$

Since $s = AX$ and $s' = \sum_{i=1}^{n} a_i - s = A'X$, $s + s' = \sum_{i=1}^{n} a_i$. Thus, these two answer indices vectors suffices

$$\text{For all } i : x_i + x'_i = 1$$

In other words, to solve original knapsack problem, one can first solve its complementary knapsack problem. Then using $1 - x'_i$ to recover the answer indices vector of the original problem.

Moreover, if we denote l' as the hamming weight of complementary kanpsack problem, l and l' satisfy the relation $l + l' = n$. In other words, if original knapsack is a α-unbalanced knapsack problem, then the complementary knapsack would be a $(1-\alpha)$-unbalanced knapsack problem. Therefore, when we solve any knapsack problem, we can always assume the hamming weight of answer indices vector $l \leq \lfloor n/2 \rfloor$.

1.7 Aims of dissertation

As we described above, for hard knapsack problem, there are 2 generic algorithms. One is Schroeppel and Shamir algorithm and the other one is HGJ algorithm. These two algorithms are both exponential time algorithm. In original paper of HGJ algorithm [17], the balanced case of knapsack problem has been discusses in detail, but it only outlined the unbalanced case briefly. The aim of this dissertation is to fully analyze time and memory complexity of 2 algorithms in unbalanced case. If possible, try to extend the application and come up with some strategy to improve algorithms.

1.8 Dissertation structure

First, in section 3 we will introduce Schroeppel-Shamir algorithm and HGJ algorithm by outlining their processes and giving a short analysis in balanced case. Next, in section 4, we will show how to adapt Schroeppel-Shamir algorithm and HGJ algorithm into unbalanced case by rearranging knapsack vector. This rearrangement is important for our analysis, which helps to reduce the size of list in algorthms and thus meet bound of complexity. Since HGJ algorithm is constructed on the base of Schroeppel-Shamir algorithm, our analysis starts from original Schroeppel-Shamir algorithm. In section 5, complexity of priority queue Schroeppel-Shamir algorithm is discussed in brief. Whereas, we find and correct an error in original algorithm, which will not effect the result in general case. Then in section 6, we will analyze the complexity of modular Schroeppel-Shamir algorithm and best choice of modulus M. A parallel version of modular Schroeppel-Shamir algorithm is also presented in this section. Then we will discuss simple version of HGJ algorithm in detail including its complexity, a parallel version and experiment result. Finally, this dissertation will end with evaluation and possible improvements.

2 Binomial coefficient

In this dissertation, binomial coefficient will be used frequently. Binomial coefficient or combination is a possibility of picking several items out of a larger group, where (unlike permutations) order does not matter. More formally, binomial coefficient $\begin{pmatrix} n \\ l \end{pmatrix}$ is the number of subset of l elements out of a n elements set. Presented in mathematical formula is

$$\begin{pmatrix} n \\ l \end{pmatrix} = \frac{n!}{l! \cdot (n-l)!} = \frac{n(n-1)\dots(n-k+1)}{k(k-1)\dots 1}$$

In convenience of the computation of complexity, we will often use asymptotic approximation for binomials. First, we should be familiar with Stirling formula, which is

$$\ln n! = n \ln n - n + O(\log(n))$$

After simple computation

$$n! = (1+o(1))\sqrt{2\pi n}(\frac{n}{e})^n$$

Following the idea when we compute complexity, we will focus on the part which dominates the size of number. Hence, given a value α in the range of $[0,1]$, $\begin{pmatrix} n \\ \alpha n \end{pmatrix}$ (or $\begin{pmatrix} n \\ \lfloor \alpha n \rfloor \end{pmatrix}$) can be represented as

$$\begin{pmatrix} n \\ \alpha n \end{pmatrix} = \widetilde{O}((\frac{1}{\alpha^\alpha \cdot (1-\alpha)^{1-\alpha}})^n)$$

If we want to present in the form of $\widetilde{O}(2^{cn})$, we should compute c by using the logarithm in basis 2 of numbers coming from asymptotic estimates of binomials. Then

$$\begin{pmatrix} n \\ \alpha n \end{pmatrix} \approx 2^{h(\alpha)n} \tag{3}$$

where

$$h(x) = -x \cdot log_2 x - (1-x) \cdot log_2(1-x)$$

3 Generic algorithms

3.1 Schroeppel and Shamir algorithm

Schroeppel and Shamir introduced their algorithm to break a generic integer knapsack problem in paper [15] and [16]. The Schroeppel and Shamir algorithm can solve n-elements subset sum problem in exponential time $\tilde{O}(2^{n/2})$ with space complexity $\tilde{O}(2^{n/4})$.

This algorithm is evolved from birthday algorithm of Horowitz and Sahni [26], which can be used to attack such a subset sum problem as well. Recalling the subset sum problem formula (2) in section 1.1, if we rewrite formula (2) by moving half of knapsack vector (i.e. $\{a_1, \ldots, a_{\lfloor n/2 \rfloor}\}$) into left hand side of equation, we will have basic birthday algorithm of subset sum problem

$$\sum_{i=1}^{\lfloor n/2 \rfloor} x_i a_i = s - \sum_{i=\lfloor n/2 \rfloor + 1}^{n} x_i a_i \tag{4}$$

where all the x_i should be either 0 or 1. Thus, to implement basic birthday algorithm, we should construct two sets $List^{(1)}$ and $List^{(2)}$ and then search for collisions between these two sets. $List^{(1)}$ contains all possible sums of first $\lfloor n/2 \rfloor$ elements, whereas $List^{(2)}$ contains the possible value that s subtracts all possible sums of other elments. If collision occurrs (i.e. there are a $\{x_1, \ldots, x_{\lfloor n/2 \rfloor}\}$ and a $\{x_{\lfloor n/2 \rfloor}, \ldots, x_n\}$ satisfies equation (4)), it must be a answer for this subset sum problem. If we go through all possible sum, all the solution in this subset sum problem will be found. The time complexity and space complexity of this algorithm are both $\tilde{O}(2^{n/2})$ including computing all possible values of two sets, sorting two sets and searching for collisions.

Instead of generating the full sets $List^{(1)}$ and $List^{(2)}$, Schroeppel and Shamir algorithm uses priority queues (based either on heaps or a Adelson-Velsky and Landis trees) to store possible values in two sets, which reduces memory requirement to $\tilde{O}(2^{n/4})$.

More specifically, denoting $q_1 = \lfloor n/4 \rfloor$, $q_2 = \lfloor n/2 \rfloor$, $q_3 = \lfloor 3n/4 \rfloor$. Then Schroeppel and Shamir algorithm constructs 4 sets $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$, $List_R^{(2)}$ whose sizes are $O(2^{n/4})$. These 4 sets are defined as follows:

$List_L^{(1)}$ stores all possible pairs of $(\sum_{i=1}^{q_1} x_i a_i)$ and its corresponding $(x_1, \ldots, x_{q_1})$ where $x_i \in \{0, 1\}$.

$List_R^{(1)}$ stores all possible pairs of $(\sum_{i=q_1+1}^{q_2} x_i a_i)$ and its corresponding $(x_{q_1+1}, \ldots, x_{q_2})$ where $x_i \in \{0, 1\}$.

$List_L^{(2)}$ stores all possible pairs of $(\sum_{i=q_2+1}^{q_3} x_i a_i)$ and its corresponding $(x_{q_2+1}, \ldots, x_{q_3})$

where $x_i \in \{0, 1\}$.

$List_R^{(2)}$ stores all possible pairs of $(\sum_{i=q_3+1}^{n} x_i a_i)$ and its corresponding $(x_{q_3+1}, \dots, x_n)$ where $x_i \in \{0, 1\}$.

Hence, subset sum problem becomes searching elements in $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$, $List_R^{(2)}$, which satifies $s = List_L^{(1)}[i] + List_R^{(1)}[j] + List_L^{(2)}[k] + List_R^{(2)}[l]$, which is called 4-way merge problem.

Then original 4-way merge routine first sorts $List_R^{(1)}$ and $List_R^{(2)}$ in increasing order. There are two priority queues, initialized by all $List_L^{(1)}[i] + List_R^{(1)}[0]$ and $s - List_L^{(2)}[i] - List_R^{(2)}[n - q_3]$, respectively. Compare the first value in two queue. If these two value are not equal, the next value in $List_L^{(1)} + List_R^{(1)}$ or $s - List_L^{(2)} - List_R^{(2)}$ is inserted into corresponding queue. In other words, by sorting two times, a priority queue outputs the next smallest in set $List^{(1)}$ or $List^{(2)}$. If the output of two priority queues are same, these two outputs must be a solution to this subset sum problem. The complete description can be found in oringinal paper [15].

Here, we use the outline version described in [17] and show the algorithm procedure as follow:

Algorithm 3: Schroeppel-Samir algorithm

```
input:
$(a_1, a_2, \dots, a_n)$: knapsack vector
$s$: target sum

Create lists $List^{(1)}(\kappa)$, $List^{(2)}(\kappa)$, $List^{(3)}(\kappa)$, $List^{(4)}(\kappa)$, $List^{(1)}(x)$, $List^{(2)}(x)$, $List^{(3)}(x)$
and $List^{(4)}(x)$
for i from 1 to 4 do
   for j=1 to $2^{n/4}$ do
      if the weight of j equals $l/4$
         Let vector $(x_1, \dots, x_{n/4})$=$(bit(j_1), \dots, bit(j_{n/4}))$
         insert $\sum_{k=1}^{n/4} x_k a_k$ into $List^{(i)}(\kappa)$
         insert $(x_1, \dots, x_n)$ into $List^{(i)}(x)$
      end if
   end for
end for
Call modular 4-way merge with input $(List^{(1)}(\kappa), List^{(2)}(\kappa), List^{(3)}(\kappa), List^{(4)}(\kappa))$, $n$
and $s$
Get set *solution*.
Create list *indices*
for o=1 to size of *solution*
```

```
    Get (i,j,k,l) from *solution*
    Compose $List^{(1)}(x)[i]$, $List^{(2)}(x)[j]$, $List^{(3)}(x)[k]$, $List^{(4)}(x)[l]$ into *indices*
end for
```

output:
indices: a list of answer indices vector

Algorithm 4: Priority queue 4-way merge

input:
$(List_L^{(1)}, List_R^{(1)}, List_L^{(2)}, List_R^{(2)})$: 4 lists to be merged
n: the number of elements in knapsacks
s: target sum

```
Letting $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$ denote the size of correspongding lists.
Construct empty priority queue $Queue_1$ and $Queue_2$
Sort $List_R^{(1)}$ and $List_R^{(2)}$ and build two arrays $unsort_1$ and $unsort_2$ to store the mapping relationship of sorted arrays and original arrays.
for i from 0 to $List_L^{(1)}$ do
    add tuple (i,0) and priority $List_L^{(1)}[i] + List_R^{(1)}[0]$ in $Queue_1$
end for
for i from 0 to $List_L^{(2)}$ do
    add tuple (i,$List_R^{(2)} - 1$) and priority $s - List_L^{(2)}[i] - List_R^{(2)}[List_R^{(2)} - 1]$ in $Queue_2$
end for
Build list *Solution*
while $Queue_1$ and $Queue_2$ are not empty do
    Empty $List_{q1}$ and $List_{q2}$
    Assign the top element's priority in $Queue_1$ to $q_1$
    Assign the top element's priority in $Queue_2$ to $q_2$
    if $q_2 \geq q_1$ then
        Pop (i,j) from $Queue_1$
        if $j \neq List_R^{(1)} - 1$ then
            add tuple (i,j+1) and priority $List_L^{(1)}[i] + List_R^{(1)}[j + 1]$ in $Queue_1$
        end if
    end if
    if $q_2 \leq q_1$ then
        Pop (k,l) from $Queue_2$
        if $l \neq 0$ then
            add tuple (k,l-1) and priority $s - List_L^{(2)}[k] - List_R^{(2)}[l - 1]$ in $Queue_2$
        end if
    end if
    if $q_2 = q_1$ then
```

```
        Add $(i, unsort_1[j], k, unsort_2[l])$ to $Solution$
    end if
end while
```

output:
Solution: a list of solutions

In Schroeppel-Shamir algorithm, to compute all possible sum and insert them into 4 lists will take $\tilde{O}(2^{n/4})$ time. In original 4-way merge algorithm, if the worst case happen, it will take $\tilde{O}(2^{n/2})$ to traverse in $List^{(1)}$ and $List^{(2)}$. So the overall running time is $\tilde{O}(2^{n/4}+2^{n/2}) = \tilde{O}(2^{n/2})$. When memory is concerned, storing all possible sum in 4 quarters require $4 * \tilde{O}(2^{n/4}) = \tilde{O}(2^{n/4})$. Since every time an element is added into $Queue_1$ or $Queue_2$, algorithm will pop an element in corresponding queue. So the size of queues is the same compared with the size when the two queues were initialized. Since the initial size of two queues are $\tilde{O}(2^{n/4})$, the overall memory requirement of original 4-way merge algorithm would be $2 * \tilde{O}(2^{n/4}) = \tilde{O}(2^{n/4})$. Combining the space complexity of Schroeppel-Shamir algorithm and its subroutine algorithm, it would be $\tilde{O}(2^{n/4})$.

3.2 Modular 4-way merge algorithm

However, in the original 4-way merge algorithm, it requires sophisticated data structure such as priority queue, which is difficult to implement and hard to optimize. In fact, as size of the knapsack n increase, the list size will also increase exponentially, which takes risk of crash in cache and results in extra memory overhead.

In order to overcome these drawback, A simpler but heuristic variant of 4-way merge algorithm is introduced in [17]. Since the list List(1) and List(2) constructed implictly is sorted by the value of modulus, we call it modular 4-way merge algorithm. It should be emphasized that, from the perspective of theoretical analysis, modular 4-way merge algorith is not better than original one. The time complexity of modular 4-way merge algorithm is the same as original one. Whereas, modular 4-way merge algorithm may require more memory.

The basic idea under this algorithm is that first pick a modulus M. Letting $\kappa_L^{(1)}$, $\kappa_R^{(1)}$, $\kappa_L^{(2)}$, $\kappa_R^{(2)}$ denote the elements in 4 lists $List_L^{(1)}, List_R^{(1)}, List_L^{(2)}, List_R^{(2)}$ respectively, then the condition to apply 4-way merge algorithm become $\kappa_L^{(1)} + \kappa_R^{(1)} = s - \kappa_L^{(2)} - \kappa_R^{(2)} (\bmod M)$. Therefore, for any answer subset, there must be a κ_M, Satisfying

$$\kappa_M = \kappa_L^{(1)} + \kappa_R^{(1)} (\bmod M) = S - \kappa_L^{(2)} - \kappa_R^{(2)} (\bmod M) \tag{5}$$

But, there is no useful condition to guess the correct vaule of κ_M. So only we can do is keep trying all possible vaule of κ_M, until a κ_M suffices equation (5). Specifically, the process of modular 4-way merge algorithm is shown in the follow

Algorithm 5: modular 4-way merge

```
input:
$(List_L^{(1)}, List_R^{(1)}, List_L^{(2)}, List_R^{(2)})$: 4 lists to be merged
$n$: the number of elements in knapsacks
$s$: target sum
```

```
Create lists $List_R^{(1)}(M)$ and $List_R^{(2)}(M)$
for i=1 to size of $List_R^{(1)}$ do
  insert $List_R^{(1)}(M)$ with tuple $(List_R^{(1)} \bmod M, i)$
end for
for i=1 to size of $List_R^{(2)}$ do
  insert $List_R^{(2)}(M)$ with tuple $(List_R^{(2)} \bmod M, i)$
end for
Sort $List_R^{(1)}(M)$ and $List_R^{(2)}(M)$ by left value in the tuple
Create list $solution$
for $\kappa_M = 0$ to $M-1$ do
  Clear $List^{(1)}(\kappa_M)$
  for i=0 to size of $List_L^{(1)}$ do
    Let $\kappa_L^{(1)}$=$List_L^{(1)}[i]$
    Let $\kappa_t$=$(\kappa_M - \kappa_L^{(1)}) \bmod M$
    Search $\kappa_t$ in $List_R^{(1)}(M)$
    for each ($\kappa_t$,j) in $List_R^{(1)}(M)$ do
      Insert $(\kappa_L^{(1)} + List_R^{(1)}[j], (i, j))$ to $List^{(1)}(\kappa_M)$
    end for
  end for
  Sort $List^{(1)}(\kappa_M)$ by left value
  for k=1 to size of $List_L^{(2)}$ do
    Let $\kappa_L^{(2)}$=$List_L^{(2)}[k]$
    Let $\kappa_t$=$(s - \kappa_M - \kappa_L^{(2)}) \bmod M$
    Search $\kappa_t$ in $List_R^{(2)}(M)$
    for each ($\kappa_t$,l) in $List_R^{(2)}(M)$ do
      Let $s' = s - \kappa_L^{(2)} - List_R^{(2)}[l]$
      Search $s'$ in $List^{(1)}(\kappa_M)$
      for each (s',(i,j)) in $List^{(1)}(\kappa_M)$ do
        Insert (i,j,k,l) into $solution$
      end for
    end for
    end for
end for
```

---

output:
Solution: a list of solutions

In each time testing κ_M, modular 4-way merge algorithm constructs $List_{(1)}(\kappa_M)$ which contains all possible combination from $List_L^{(1)}$ and $List_R^{(1)}$ such that $\kappa_M = \kappa_L^{(1)} + \kappa_R^{(1)} (\mod M)$. In convenience of searching for each κ_M, $List_R^{(1)}(M)$ is constructed by sorting $List_R^{(1)}$ by values modulo M. Every time picking an $\kappa_L^{(1)}$ in $List_L^{(1)}$, algorithm will binary search corresponding $\kappa_R^{(1)}$ sufficing $\kappa_R^{(1)} = \kappa_M - \kappa_L^{(1)}$ in $List_R^{(1)}(M)$. In fact, by using this method, $List(1)$ constructed in original meet-in-the-middle algorithm is divided into M small lists.

Similarly, when searching the collision for the value $s - \kappa_M$, a $List^{(2)}(M)$ is constructed impictly to divide $List^{(2)}$ into M disjoint sets. Then for each possible value of $\kappa_L^{(2)}$, algorithm searchs $s - \kappa_M - \kappa_L^{(2)}$ in the sorted $List_R^{(2)}(M)$. Once collision occur, final step is to test whether their sum equals target s without modulo.

As explained in section 3.1, the time and space required to build up $List_L^{(1)}, List_R^{(1)}, List_L^{(2)}$ and $List_R^{(2)}$ is $\tilde{O}(2^{n/4})$. Letting M be $2^{n/4}$, the expected size of $List_{(1)}(\kappa_M)$ and $List_{(2)}(\kappa_M)$ would be $\tilde{O}(2^{n/4})$ (although it can not be guaranteed). For the reason that algorithm should guess the correct value of κ_M, algorithm will test all κ_M in worst case. Since M is a $n/4$ bit value, trying every κ_M will contribute $\tilde{O}(2^{n/4})$ to time complexity, which will result in the total running time becoming $\tilde{O}(2^{n/4}) * \tilde{O}(2^{n/4}) = \tilde{O}(2^{n/2})$.

Because algorithm requires to store $List_L^{(1)}, List_R^{(1)}, List_L^{(2)}, List_R^{(2)}, List_{(1)}(\kappa_M)$ and $List_{(2)}(\kappa_M)$ all of which sizes are $\tilde{O}(2^{n/4})$, the memory complexity is $\tilde{O}(2^{n/4})$.

3.3 The Howgrave-GrahamJoux Algorithm

First, let's recall our basic knapsack formula:

$$s = AX = \sum_{i=1}^{n} x_i a_i, \quad x_i \in \{0,1\} \quad \text{for all} \quad i$$

In section 1.6, we have explianed that we can always assume the hamming weight of answer indices vector $l = \sum_{i=1}^{n} x_i \leq \lfloor n/2 \rfloor$, by solving complementary knapsack if needed. In order to facilitate analysis, we further assume that the size of knapsack n is a multiple of four.

The main idea of HGJ algorithm is to transform original big problem into 2 smaller sub-knapsack problems with the same knapsack vector A, but different hamming weight $l/2$, different target sum κ_1 and κ_2. The target sums of these two sub-knapsack problems suffice $s = \kappa_1 + \kappa_2$. Formally, we construct two sub-knapsack problems

$$\kappa_1 = AY = \sum_{i=1}^{n} y_i a_i \text{ and } \kappa_2 = AZ = \sum_{i=1}^{n} z_i a_i \tag{6}$$

where

$$s = \kappa_1 + \kappa_2$$

under the constraint

$$\begin{cases} \sum_{i=1}^{n} y_i = l/2, & y_i \in \{0,1\} \quad \text{for all} \quad i \\ \sum_{i=1}^{n} z_i = l/2, & z_i \in \{0,1\} \quad \text{for all} \quad i \end{cases}$$

As long as 2 answer indices vectors does not overlap, obiviously, if we add two vectors Y and Z, we can thereby obtain our solution X. In other words, if we represent any x_i in the form of (y_i, z_i), then $x_i = 0$ can be represented as $(0,0)$ and $x_i = 1$ can be represented as $(0,1)$ or $(1,0)$, whereas a $(1,1)$ is not a legal status which causes overlaping in x_i. Namely, there is at most one value in tuple equals to one.

Hence, the original answer indices vector X is decompsed into many different represetation according to different value of κ_1 and κ_2. If l is even, the number of all possible decompositions is picking $l/2$ elements from l elements, i.e. $\binom{l}{l/2}$. Whereas, for an odd l, this number would be $2\binom{l}{(l-1)/2}$.

But since κ_2 is determined by $s - \kappa_1$ and the possible value of κ_1 is in the range of $[0, l * 2^{n/D}]$ where D is the prescribed density, the probability of a correct guess of κ_1 is then $\frac{\binom{l}{l/2}}{l * 2^{n/D}}$ assuming l is even. If we consider the hardest knapsack problem whose hamming weight is $n/2$ and D is 1, using equation (3), we obtain the probability of correct guess in this case is $\tilde{O}(2^{-n/2})$. Clearly, this probability is too small. Therefore, if we choose a modulus M, reducing the range of options and increasing redundancy, equation (6) transfers into

$$\kappa_1 = AY = \sum_{i=1}^{n} y_i a_i \equiv R (\text{mod } M) \text{ and } \kappa_2 = AZ = \sum_{i=1}^{n} z_i a_i \equiv s - R(\text{ mod } M)$$

As a result, to find a decomposition of κ_1 and κ_2, two full sets of modular knapsack with target sum R and $s - R$ are constructed. If picking M near the number of decompositions in original problem, i.e. $\binom{l}{l/2}$, the expected number of solutions to each of these two modular subknapsacks is

$$L = \frac{\binom{n}{l/2}}{M} = \frac{\binom{n}{l/2}}{\binom{l}{l/2}}$$

If we still substitute $n/2$ into l and estimate binomial using equation (3), the list size of solution of modular sub-knapsack would be $\tilde{O}(2^{h(1/4)n}/2^{n/2}) = \tilde{O}(2^{0.3113n})$.

HGJ algorithm believes that to solve these two sub-knapsack problem will spend time $\tilde{O}(L)$. Two lists are obtained from these two sub-knapsack problem and we denote them as $List^1_{\lfloor l/2 \rfloor}$ and $List^2_{\lfloor l/2 \rfloor}$. To merge partial sums in $List^1_{\lfloor l/2 \rfloor}$ and $List^2_{\lfloor l/2 \rfloor}$, we search collision between κ_1 and κ_2 and check their legality of overlapping in two lists. By using sort and match technique, it will done in time $\tilde{O}(L) = \tilde{O}(2^{0.3113n})$. (Although May and Meurer [27] proved that there are some mistakes in the analysis of HGJ algorithm in paper [17]. Its actual running time should be $\tilde{O}(2^{(0.337n)}$. For more detail, see [7])

To conclude, the basic idea under HGJ algorithm is to divide original problem into 2 or more modular sub-knapsack problems with target sum $R(\text{mod } M)$ and $s-R(\text{ mod } M)$. By correct guessing of R and $s-R$, using matching method to paste partial solution and get the solution of original problem.

3.4 Simple version of HGJ algorithm

Howgrave Graham and Joux introduced two versions of HGJ algorithm (simple version and recursive version) in [17]. In this paper, we will only focus on its simple version.

We start introducing simple version with assumptions. To facilitate the analysis, we assume that the size of knapsack n is a multiple of 4. Whether n is a multiple of 4 will not effect the result of analysis. The time and space complexity will still hold when n is not a multiple of 4. Another assumption is that the hamming weight of solution satisfies $l = \sum_{i=1}^{n} x_i = \lfloor n/2 \rfloor$. Should this not be the case, algorithm can be run $\lfloor n/2 \rfloor$ times with different target sum from 0 to $\lfloor n/2 \rfloor$. Since $\lfloor n/2 \rfloor$ is within polynomial time, it will not effect the time complexity. In the process of $\lfloor n/2 \rfloor$ times running, $l = \lfloor n/2 \rfloor$ is the hardest case and dominates the total running time.

The process of simple version is shown below

Algorithm 6:Simple version of HGJ algorithm

```
input:
$(a_1, a_2, \dots, a_n)$: knapsack vector
$s$: target sum
$M$: modulus
```

```
Create lists $List^{(1)}(\kappa)$, $List^{(2)}(\kappa)$, $List^{(3)}(\kappa)$, $List^{(4)}(\kappa)$, $List^{(1)}(\kappa)(M)$, $List^{(2)}(\kappa)(M)$
```

```
, $List^{(3)}(\kappa)(M)$ , $List^{(4)}(\kappa)(M)$, $List^{(1)}(x)$, $List^{(2)}(x)$, $List^{(3)}(x)$ and $List^{(4)}(x)$
for i from 1 to 4 do
  for j=1 to $2^{n/4}$ do
    if the weight of j equals $l/4$
      Let vector $(x_1,\ldots,x_{n/4})$=$(bit(j_1),\ldots,bit(j_{n/4}))$
      insert $\sum_{k=1}^{n/4} x_k a_k$ into $List^{(i)}(\kappa)$
      insert $\sum_{k=1}^{n/4} x_k a_k$ mod M into $List^{(i)}(\kappa)(M)$
      insert $(x_1,\ldots,x_n)$ into $List^{(i)}(x)$
    end if
  end for
end for
Create list indices, $List^{(1)}(\kappa)'$, $List^{(2)}(\kappa)'$, $List^{(3)}(\kappa)'$, $List^{(4)}(\kappa)'$, $List^{(1)}(x)'$, $List^{(2)}(x)'$,
$List^{(3)}(x)'$ and $List^{(4)}(x)'$
while indices is empty
  Generate $R_1$, $R_2$ and $R_3$ randomly in the range of $[0, M-1]$
  Let $R_4 = s - R_1 - R_2 - R_3$
  Clear $List^{(1)}(\kappa)'$, $List^{(2)}(\kappa)'$, $List^{(3)}(\kappa)'$, $List^{(4)}(\kappa)'$, $List^{(1)}(x)'$, $List^{(2)}(x)'$, $List^{(3)}(x)'$
and $List^{(4)}(x)'$
  for o from 1 to 4 do
    for p=1 to $l/4$ do
      Call parallel modular 4-way merge with input ($List^{(1)}(\kappa)(M)$, $List^{(2)}(\kappa)(M)$,
$List^{(3)}(\kappa)(M)$, $List^{(4)}(\kappa)(M)$),$n$ and $R_o + pM$
      Get returned set solution
      for i=1 to size of solution
        Get (i,j,k,l) from solution
        Compose $List^{(1)}(x)[i]$, $List^{(2)}(x)[j]$, $List^{(3)}(x)[k]$, $List^{(4)}(x)[l]$ into $List^{(o)}(x)'$
        Insert $List^{(1)}(\kappa)[i]$+$List^{(2)}(\kappa)[j]$+ $List^{(3)}(\kappa)[k]$+ $List^{(4)}(\kappa)[l]$ into $List^{(o)}(\kappa)'$
      end for
    end for
  end for
  Call parallel modular 4-way merge with overlap checking and early abort strategy
and input ($List^{(1)}(\kappa)'$, $List^{(2)}(\kappa)'$, $List^{(3)}(\kappa)'$, $List^{(4)}(\kappa)'$), ($List^{(1)}(x)'$, $List^{(2)}(x)'$,
$List^{(3)}(x)'$, $List^{(4)}(x)'$) ,$n$ and $s$
  for i=1 to size of solution
    Get (i,j,k,l) from solution
    Compose $List^{(1)}(x)'[i]$, $List^{(2)}(x)'[j]$, $List^{(3)}(x)'[k]$, $List^{(4)}(x)'[l]$ into indices
  end for
end while
```

output:
indices: a list of answer indices vector

Different from original idea in section 3.3, instead of solving knapsack problem by dividing into 2 sub-knapsack problem, simple version will decompose original answer indices vector into 4 parts such that:

$$s = \kappa_1 + \kappa_2 + \kappa_3 + \kappa_4$$

where κ_i is a guess of decomposion by picking $\lfloor l/4 \rfloor$ elements from original answer subset. Then each indices vector with target sum κ_i can be obtained sub-knapsack with hamming weight $l' = \lfloor l/4 \rfloor$, i.e.

$$\begin{cases} \kappa_1 = AT = \sum_{i=1}^{n} t_i a_i \\ \kappa_2 = AU = \sum_{i=1}^{n} u_i a_i \\ \kappa_3 = AV = \sum_{i=1}^{n} v_i a_i \\ \kappa_4 = AW = \sum_{i=1}^{n} w_i a_i \end{cases}$$

under the constraint

$$\begin{cases} \sum_{i=1}^{n} t_i = l/4, \quad t_i \in \{0,1\} \quad \text{for all} \quad i \\ \sum_{i=1}^{n} u_i = l/4, \quad u_i \in \{0,1\} \quad \text{for all} \quad i \\ \sum_{i=1}^{n} v_i = l/4, \quad v_i \in \{0,1\} \quad \text{for all} \quad i \\ \sum_{i=1}^{n} w_i = l/4, \quad w_i \in \{0,1\} \quad \text{for all} \quad i \end{cases}$$

Similar with the original idea, answer indices vector $X = (x_1, x_2, \ldots, x_n)$ can be obtained by $X = T + U + V + W$, when there is no overlapping in vector T, U, V and W. We represent x_i in the form (t_i, u_i, v_i, w_i). Then since $x_i = t_i + u_i + v_i + w_i$ and $x_i \in \{0,1\}$, no overlapping means there is at most a 1 in quadruple (t_i, u_i, v_i, w_i), where $t_i, u_i, v_i, w_i \in \{0,1\}$.

Letting Ω denotes the number of all possible decompositions of answer subset, then when l is a multiple of 4, we can get

$$\Omega = \binom{l}{l/4} * \binom{3l/4}{l/4} * \binom{l/2}{l/4} \text{ when } l = 0 \bmod 4$$

However, no matter whether l is a mutiple of four, according to Stirling's formula, Ω would be the size of $\tilde{O}(2^{(n)})$.

Then by picking a prime M near $2^{\tau n}$ where τ is in the range of $(1/4, 1/3)$, κ_1, κ_2, κ_3 and κ_4 can be represented in $\mathbb{Z}_M$, i.e.

$$\begin{cases} \kappa_1 = R_1(\bmod M) \\ \kappa_2 = R_2(\bmod M) \\ \kappa_3 = R_3(\bmod M) \\ \kappa_4 = s - R_1 - R_2 - R_3(\bmod M) \end{cases} \tag{7}$$

Since R_4 is determined by other 3 values, the expected number of solutions out of 4 sub-knapsack problem that satisfies equation (7) would be $\frac{\Omega}{M^3}$.

Then the overall running time of the algorithm is $\tilde{O}(max(2^{0.3113n}, 2^{0.0872n+\tau n})$ and the memory requirement would be $\tilde{O}(2^{0.5436n-\tau n})$. If a more detailed process and analysis is needed, see setion 7.1 for unbalanced case and analogy balanced case from unbalanced case.

4 Rearrange knapsack vector and good arrangement

In last section, we have talked about random knapsack problem in balanced case. In the following sections, we will focus on unbalanced case, particularly when a good arrangement is provided.

Recall unbalanced knapsack problem in section 1.6.3. If $l = \lfloor \alpha \rfloor$ is unknown, we can keep trying every possible values from 0 to $\lfloor n/2 \rfloor$. Since it is in the polynomial time, it will not effect the complexity of time. So we can always assume that l is known.

Basic meet-in-the-middle algorithm constructs two list $List^{(1)}$ and $List^{(2)}$ containing all possible sums in knapsack vector from first half and second half. Then search collision $\kappa_1 = s - \kappa_2$ in these two lists. If collision happen, add indices vector to solution list. If we know $l/2$ elements in the answer subset come from first half of knapsack vector, and $l/2$ from the second half, then basic meet-in-the-middle algorithm can be applied easily to unbalanced case. When computing $List^{(1)}$ and $List^{(2)}$, we just ignore the sums whose hamming weight of indices vector is not $l/2$. This change is important, since the time complexity of basic meet-in-the-middle algorithm as well as Schroeppel-Samir algorithm is $\tilde{O}(L)$, where L is the size of the list. So reducing the size of list will directly reduce the running time. Even though the running time of HGJ algorithm is not $\tilde{O}(L)$, the size of list also effects its running time dramatically, since in HGJ algorithm, Schroeppel-Samir algorithm or other methods would be called to solve its sub-knapsack problem. $List^{(1)}$ and $List^{(2)}$ contains all possible sum whose indices vector's hamming weight is $l/2$ in each half respectively and there are $n/2$ elements in each half, so the size of list would be the number that picking $l/2$ elements from a n set, which is $\binom{n/2}{l/2}$.

Thanks to existence of complementary knapsack, we can always assume that $l \leq n/2$. If $l \neq n/2$, from the view of theoretical analysis, the running time of basic meet-in-the-middle algorithm and Schroeppel-Samir algorithm are not $\tilde{O}(2^{n/2})$ anymore. Since when $l \neq n/2$, $\binom{n/2}{l/2}$ and $2^{n/2}$ are not whithin polynomial factors of each other. Even if $l = n/2$, in practice, $\binom{n/2}{l/2}$ is still smaller than $2^{n/2}$, which will also be helpful to reduce the running time.

Thanks to Coppersmith algorithm which can be found in paper [23], the assumption that half of answer subset come from the first half in knapsack vector and the remaining half come from the second half is reasonable. The basic idea of Coppersmith algorithm is to rearrange the knapsack vector to meet the assumption described above. More specifically, to apply Coppersmith algorithm to basic birthday algorithm, it will distinguish two case whether n is even or not. When n is even, Coppersmith algorithm can be appled directly, it will construct knapsacks with original target sum by rearrange the elements in knapsack vector in different order. One method to rearrange elements is to rotate knapsack vector in the step of 1, i.e. the i-th element in next knapsack is the (i-1)-th element in last knapsack ($a'_i = a_{(i-1) \text{ mod n}}$). Considering that, if knapsack vector in original problem is $A = (a_1, a_2, \ldots, a_n)$, then once applying Coppersmith algorithm will generate a knapsack $(a_2, a_3, \ldots, a_n, a_1)$. Twice applying will generate a knapsack $(a_3, a_4, \ldots a_n, a_1, a_2)$. After n times applying, n knapsack will be generated. Among these n knapsack, there is at least a knapsack whose half of elements in answer subset come from the first half of knapsack vector. To prove this is useful, considering a sliding window with $n/2$ elements intersects fixed $l/2$ elements at least once, for more information see [23]. In worst case, it will take n times iterations of Coppersmith algorithm.

When n is odd, we solve original knapsack problem by solving two knapsack problems. If the knapsack vector is $(a_1, a_2, \ldots, a_n)$, hamming weight is l and target sum is s in original knapsack problem, then when n is odd, solving original problem is equivalent to solving two knapsacks with $n-1$ elements $(a_1, a_2, \ldots, a_{n-1})$ and two target sum s and $(s - a_n)$. In other words, it can distinguish two cases. If a_n is not in the answer subset, solving $n-1$ elements in which l elements is picked knapsack with target sum s can obtain solution. On the other hand, if a_n is in the answer subset, solving $n-1$ elements in which $l-1$ elements is picked knapsack with target sum $s - a_n$ can obtain solution. Then in these two knapsack problems, since $n-1$ is an even number, problem return to even case.

As an alternative, we can randomize the order of elements in knapsack vector A. Then test whether new knapsack satisfies our assumption. Dividing n elements into 2 halves, the number of all possible probabilities are $\binom{n}{l}$. Picking $l/2$ elements from l elements in answer subset, we obtain $\binom{l}{l/2}$. Then by picking rest $n/2 - l/2$ elements in first half from non-answer subset, we obtain $\binom{n-l}{n/2-l/2}$. Then we find the probability of our assumption is $\frac{\binom{l}{l/2} * \binom{n-l}{n/2-l/2}}{\binom{n}{l}}$. Thus the expected number of trials is $\frac{\binom{n}{l}}{\binom{l}{l/2} * \binom{n-l}{n/2-l/2}} = O(\sqrt{n})$, if using Stirling's fomular.

When applying Schroeppel-Samir algorithm to unbalanced case, similarly, we will hope that the elements in answer subset is distributed evenly in four quaters, i.e. the hamming weight of answer indices vector in each quater is $\lfloor l/4 \rfloor$. We call the knapsacks satisfy the assumption described above **good arrangement**.

Similar with basic birthday algorithm, if good arrangement is provided, Schroeppel-Samir algorithm and HGJ algorithm can be modified into unbalanced case easily. When constructing 4 lists, $\binom{n/4}{l/4}$ possible sums would be added to each list, rather than $2^{n/4}$. Let us assume that n mod $4 = 0$. Good arrangement can be achieved by first applying the whole knapsack Coppersmith algorithm and then in each half of knapsack, applying Coppersmith algorithm again. Since the only way we can examine whether the knapsack is in good arrangement is trying to find the solution. So in deterministic way, we will try at most $n^3/4$ times. Similarly, if arrange elements in knapsack vector in random order, the expected number of trials is $O(n^{3/2})$.

From this section onwards, we will discuss the unbalanced knapsack problem, when good arrangement is provided.

5 Priority queue version of Schroeppel-Samir algorithm

When discussing Schroeppel-Samir algorithm, we will pay more attetion to 4 way merge algorithm. The reason is, on one hand, it dominates the running time of Schroeppel-Samir algorithm in most case. On the other hand, it will also be called in HGJ algorithm and it also effects the running time in HGJ algorithm dramatically.

Since priority queue 4-way merge algorithm exists for a long time, in this section, we will only analyze its time and space complexity briefly. Then we will talk about a small error in original algorithm and present a modified version. Finally, we will discuss experiment and its result.

5.1 Theoretical analysis

5.1.1 Time and memory complexity

In order to facilitate analysis, in the following sections, we always assume that n and l are divisible by 4. Recalling Schroeppel-Samir algorithm and its priority queue 4-way merge, first, knapsack vector is divided into 4 quarters. In each quarter, compute all possible sums and add them into a list. Then we obtain 4 lists, whose size would be $\binom{\lceil n/4 \rceil}{l/4} = \binom{\lceil n/4 \rceil}{(\alpha n)/4}$, if good arrangement is provided. Using binomial asymptotic approximation, the size of lists would be $2^{h(\alpha)n/4}$. To merge 4 lists using priority queue, it will take square of the list size time to find all solution, i.e. $\binom{\lceil n/4 \rceil}{l/4}^2 = \tilde{O}(2^{h(\alpha)n/2})$. Then the total running time is $\tilde{O}(2^{h(\alpha)n/4}) + \tilde{O}(2^{h(\alpha)n/2}) = \tilde{O}(2^{h(\alpha)n/2})$. To store priority queue and lists containing possible sums in each quarter, it requires $\binom{\lceil n/4 \rceil}{l/4} = \tilde{O}(2^{h(\alpha)n/4})$ bis memory.

5.1.2 Early abort strategy

However, often, we do not care all the solutions. When only a single solution is concerned, we can quit the algorithm after first solution is found, which we call it early abort strategy. We denote the number of solution as ζ. If all indices vectors have equal probability to be the answer indices vector, heuristically, we expect 4-way merge algorithm will find a solution at time $\tilde{O}(2^{h(\alpha)n/2})/\zeta$. Although, when we solve α-unbalanced knapsack problem generated by algorithm 2 with density close to 1, there is often only one solution. Then algorithm will be expected to stop at half of the running time, i.e. $\tilde{O}(2^{h(\alpha)n/2})/2$, i.e. it does not change the time complexity. But when applying Schroeppel-Samir algorithm to solve sub-knapsack problem in HGJ algorithm, ζ may be large. It may effect the time complexity greatly.

In a word, the overall running time would be the time to compute all possible sum in each quarter add the time to merge 4 lists, i.e.

$$\tilde{O}(2^{h(\alpha)n/4}) + \tilde{O}(2^{h(\alpha)n/2}/\zeta) = \tilde{O}(max(2^{h(\alpha)n/4}), 2^{h(\alpha)n/2}/\zeta))$$

5.1.3 Parallel

Priority queue 4-way merge algorithm can not be modified parallelly. This algorithm needs to construct two list $List^{(1)}$ and $List^{(2)}$ implicitly, and then searches for collision in two lists. The basic idea of priority queue 4-way merge algorithm is to maintain partial of $List^{(1)}$ and $List^{(2)}$ by priority queue, so as to reduce the memory requirement. Since the next state of queues depend on their last state, algorithm can not be divided into several subtasks. Thereby, algorithm can only be executed serially.

5.2 Modified version

There is a small error in priority queue 4-way merge algorithm described in [17]. Original algorithm searches collisions by poping and comparing the lowest value in two priority queues. However, since in priority queue elements can be obtained from the end of queue, when collision happen with several consecutive q_1 and q_2, algorithm will lose part of combination of q_1 and q_2. To make this clear, considering this case, if the state of two priority queues are

$$\text{Queue 1:} \begin{pmatrix} \text{Priority:} & q \\ \text{Tuple:} & (i,j) \end{pmatrix}, \begin{pmatrix} \text{Priority:} & q \\ \text{Tuple:} & (i',j') \end{pmatrix}, \begin{pmatrix} \text{Priority:} & q' \\ \text{Tuple:} & (i'',j'') \end{pmatrix}, \ldots$$

$$\text{Queue 2:} \begin{pmatrix} \text{Priority:} & q \\ \text{Tuple:} & (k,l) \end{pmatrix}, \begin{pmatrix} \text{Priority:} & q \\ \text{Tuple:} & (k',l') \end{pmatrix}, \begin{pmatrix} \text{Priority:} & q'' \\ \text{Tuple:} & (k'',l'') \end{pmatrix}, \ldots$$

here, collision should happen 4 times ([(i,j) and (k,l)], [(i,j) and (k',l')], [(i',j') and (k,l)], [(i',j') and (k',j')]). But in original algorithm priority queue can only compare the value at the end of queue, i.e. [(i,j) and (k,l)], [(i',j') and (k',l')]. Although, when applying Schroeppel-Samir algorithm to solve the problem directly, ζ is small. Hence, collision rarely happen, and moreover consecutive collision with the same priority in queues happen much more rarely. In most case, such an error will not effect the result. However, when applying Schroeppel-Samir algorithm in HGJ algorithm to solve the sub-knapsack problem, ζ would be much larger. Such an error will lose some solution of the sub-knapsack problem.

To fix this error, every time collision at value q happens, algorithm should test whether the next element's priority in queue is q. If it does, add this element into a list. By combining all elements in list, algorithm finds all solutions.

Algorithm 7:modified priority queue 4-way merge

input:
($List_L^{(1)}, List_R^{(1)}, List_L^{(2)}, List_R^{(2)}$): 4 lists to be merged
n: the number of elements in knapsacks
s: target sum

```
Letting $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$, $List_R^{(2)}$, $List_{q1}$ and $List_{q2}$ denote the size of corre-
spongding lists.
Construct empty priority queue $Queue_1$ and $Queue_2$
Sort $List_R^{(1)}$ and $List_R^{(2)}$ and build two arrays $unsort_1$ and $unsort_2$ to store the map-
ping relationship of sorted arrays and original arrays.
for i from 0 to $List_L^{(1)}$ do
  add tuple (i,0) and priority $List_L^{(1)}[i] + List_R^{(1)}[0]$ in $Queue_1$
end for
for i from 0 to $List_L^{(2)}$ do
  add tuple (i,$List_R^{(2)} - 1$) and priority $s - List_L^{(2)}[i] - List_R^{(2)}[List_R^{(2)} - 1]$ in $Queue_2$
end for
Build list $Solution$
Build list $List_{q1}$
Build list $List_{q2}$
while $Queue_1$ and $Queue_2$ are not empty do
  Empty $List_{q1}$ and $List_{q2}$
  Assign the top element's priority in $Queue_1$ to $q_1$
  Assign the top element's priority in $Queue_2$ to $q_2$
  if $q_2 > q_1$ then
    Pop (i,j) from $Queue_1$
    if $j \neq List_R^{(1)} - 1$ then
      add tuple (i,j+1) and priority $List_L^{(1)}[i] + List_R^{(1)}[j+1]$ in $Queue_1$
    end if
  end if
  if $q_2 < q_1$ then
    Pop (k,l) from $Queue_2$
    if $l \neq 0$ then
      add tuple (k,l-1) and priority $s - List_L^{(2)}[k] - List_R^{(2)}[l-1]$ in $Queue_2$
    end if
  end if
  if $q_2 = q_1$ then
    while the top element's priority in $Queue_1$ equals $q_1$
      Pop (i,j) from $Queue_1$
      Add (i,j) to $List_{q1}$
      if $j \neq List_R^{(1)} - 1$ then
        add tuple (i,j+1) and priority $List_L^{(1)}[i] + List_R^{(1)}[j+1]$ in $Queue_1$
```

```
            end if
        end while
        while the top element's priority in $Queue_2$ equals $q_1$
            Pop (k,l) from $Queue_2$
            Add (k,l) to $List_{q2}$
            if $l \neq 0$ then
                add tuple (k,l-1) and priority $s - List_L^{(2)}[k] - List_R^{(2)}[l-1]$ in $Queue_2$
            end if
        end while
        for a from 0 to $List_{q1}$
            for b from 0 to $List_{q2}$
                Get (i,j) from $List_{q1}[a]$
                Get (k,l) from $List_{q2}[b]$
                Add $(i, unsort_1[j], k, unsort_2[l])$ to $Solution$
            end for
        end for
    end if
end while
```

output:
Solution: a list of solutions

5.3 Experiment

When implementing experiment, as n increases, for a given density, the size of number can be larger than the bound of int type in C. So we need a large number library. NTL is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. It can combine with GMP (the GNU Multi-Precision library) for a powerful performance on Unix. WinNTL is a windows version library and can be used in versions after windows 95.

All experiments in this paper is implemented by NTL library. One reason to use NTL is its speed. NTL's polynomial arithmetic is the fastest available anywhere. Its speed for polynomial factorization even holds the world record. Another reason is NTL is a open source software distributed under the terms of the GNU General Public License, which we can use it free.

Experiments run in a dual-core Intel core i3 cpu at 2.13*2 Ghz with memory 2 GB.

Table 1 shows as n increases, the running time of Schroeppel-Shamir algorithm grows exponentially. When early abort strategy is applied, experment result conforms our analysis. The first row indicates the size of knapsack n.

n	32	36	40	44	48
Early abort	1.322	7.41	32.894	125.626	433.180
Total time	2.496	13.9	61.651	249.849	850.169

Table 1.the experimental running time in seconds of different n, when $l = 16$.

Table 2 shows the number of solution ζ found by different algorithm when solving modular knapsack problem. From table, we can see original priority queue 4-way merge will lose some solutions when ζ is large. The theoretical expectations of ζ is computed by $\frac{\binom{n/4}{l/16}^4}{M}$. The first row indicates the value of M. We can see that the less M is, the more ζ is and thus the more solutions lose.

M	35	50	69
expectation	418	292.8	212
original priority queue	94	78	65
modified version	418	295	209

Table 2.the number of solution ζ found by different algorithm when solving modular knapsack problem.

6 Modular version

6.1 Theoretical analysis

6.1.1 Time complexity

Recall the modular 4-way merge algorith in section 3.2. As input, to construct 4 lists $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$ in Schroeppel and Shamir algorithm is inevitable process. When good arrangement is provided, the number of all possible sums in each quarter is $\binom{n/4}{l/4}$. Namely, there would be $\binom{n/4}{l/4}$ values adding into each list. Using Stirling's fomular, $\binom{n/4}{l/4} = 2^{h(\alpha)n/4}$, where $\alpha = l/n$. Thereby, the time complexity of modular version used by Schroeppel-Shamir algorithm is at least $\tilde{O}(2^{h(\alpha)n/4})$.

There are 4 loops in the mainbody of algorithm. Obiviously, outermost loop will be executed M times. The number of times of the second loop is the size of $\max(List_L^{(1)}, List_L^{(2)}$, i.e. $\binom{n/4}{l/4} = 2^{h(\alpha)n/4}$. The third loop is trying to match a sum in $List_L^{(1)}$ to a modular sum in $List_R^{(1)}(M)$. To estimate the number that matching happen for each element in $List_L^{(1)}$ and for each κ_M, it is equivalent to address such a probability problem.

Imagine that we have a balls and b boxes. We want to put these a balls into b boxes. Our goal is caculating the expected number of balls in each box, assuming that each ball has equal probability to put into each box and every ball should be located into a box. Then for each box the probability to get a ball is $1/b$. We have a balls, so the expected number of balls is a/b. In this simple problem, we can also believe that the expected number of balls equals the number that assign a balls equally into b boxes, i.e. a/b.

If we take sums in $List_R^{(1)}(M)$ as a balls, modulus M as b boxes, then by substituting size of $List_R^{(1)}(M) = \binom{n/4}{l/4} = (2^{h(\alpha)n/4})$ into a and M into b respectively, we can get for each elements and for each κ_M in $List_L^{(1)}$, the expected number of elements in $List_R^{(1)}(M)$ satisfying $\kappa_M = \kappa_L^{(1)} + \kappa_R^{(1)} \bmod M$ is $\frac{2^{h(\alpha)n/4}}{M}$.

Therefore, the total time complexity is the product of iterations of 4 loops, i.e. $M * 2^{h(\alpha)n/4} * \frac{2^{h(\alpha)n/4}}{M} * \zeta = \tilde{O}(2^{h(\alpha)n/2}) * \zeta$. In most case, the number of solution ζ is small and can be considered as constant, so the expected running time is $\tilde{O}(2^{h(\alpha)n/2})$. But when applying modular 4-way merge to solve the subknapsack in HGJ algorithm, ζ can be large. In this case, the time complexity can be written as $max(\tilde{O}(2^{h(\alpha)n/2}), \zeta)$. Since $\tilde{O}(2^{h(\alpha)n/2})$ and $\tilde{O}(2^{h(\alpha)n/4})$ are not within the polynomial factor with each other, the time to construct 4 input lists are ignored.

From final result, we can see M has nothing to do with time complexity. But in fact, M does effect the running time and the best M should equal to the size of 4 input lists, i.e. $\binom{n/4}{l/4}$. As decribed above, a smaller M increases the size of $List^{(1)}(\kappa_M)$, which will cause more time in sorting and searching. Consider extreme case, when

$M = 1$, thus Schroeppel and Shamir algorithm degrade into basic meet-in-the-middle algorithm. However, a larger M also causes growth in running time. Since equation of our running time is $M * 2^{h(\alpha)n/4} * \frac{2^{h(\alpha)n/4}}{M} * \zeta$ and, in each loop, code would be executeed at least once, i.e. $\frac{2^{h(\alpha)n/4}}{M} \geq 1$. So when $M \geq 2^{h(\alpha)n/4}$, running time becomes $M * 2^{h(\alpha)n/4} * 1 * \zeta$. Thus, when $M \geq 2^{h(\alpha)n/4}$, as M increases, the running time increases. For an experimental result that different M effects the running time, see section 6.2.

6.1.2 Memory complexity

There are 8 lists to be stored in algorithm. Lists $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$ store all the possible sum in each quarter. Thus their size should be $\binom{n/4}{l/4} = 2^{h(\alpha)n/4}$, if good arrangement is provided. $List_R^{(1)}(M)$ and $List_R^{(2)}(M)$ store modular of the elements in $List_R^{(1)}$ and $List_R^{(2)}$ respectively, so their size are $2^{h(\alpha)n/4}$ as well. As a variant version of meet-in-the-middle algorithm, instead of constructing the full list of $List^{(1)}$ and $List^{(2)}$ to search for collisions in these two lists, modular version of Schroeppel and Shamir algorithm constructs $List^{(1)}(\kappa_M)$ dividing $List^{(1)}$ into disjoint union of smaller sets. By searching collisions in two smaller sets, modular version of Schroeppel and Shamir algorithm saves the memory requirement. Then to estimate the size of $List^{(1)}(\kappa_M)$, our problem becomes caculating the expected number for each value of M when from a $(\binom{n/4}{l/4})^2$ set $List^{(1)}$ mapping to a M space. Using the equation we come up from balls and boxes, we obtain the expected size of $List^{(1)}(\kappa_M)$ is $\frac{(\binom{n/4}{l/4})^2}{M} = 2^{h(\alpha)n/2}$.

But we can not always guarantee that the size of $List_R^{(1)}(M)$ is $\frac{2^{h(\alpha)n/2}}{M}$. Considering this case, we choose a M where every elements a_i in knapsack vector is divisible by M. In this case, all sums in $List^{(1)}$ is distributed in $List^{(1)}(\kappa_M = 0)$, i.e. the biggest size of is $List^{(1)}(\kappa_M) = List^{(1)}(\kappa_M = 0) = List^{(1)} = 2^{h(\alpha)n/2}$. However, according to Corollary 1 in paper [17], this probability is exponentially small.

Adding these 8 lists together, the memory complexity is $max(\binom{n/4}{l/4}, \frac{(\binom{n/4}{l/4})^2}{M}, \zeta)$. Hence, it suggests that the memory complexity is $\binom{n/4}{l/4}$ at least. To minimize the memory complexity, we want $\frac{\binom{n/4}{l/4}^2}{M} \leq \binom{n/4}{l/4}$. If we pick $M = \binom{n/4}{l/4}$, then the memory complexity becomes $max(\binom{n/4}{l/4}, \zeta) = max(2^{h(\alpha)n/4}, \zeta)$.

6.1.3 Parallel

Modular version of Schroeppel-Samir algorithm can be modified into a parallel algorithm. When creating lists $List^{(1)}(\kappa)$, $List^{(2)}(\kappa)$, $List^{(3)}(\kappa)$ and $List^{(4)}(\kappa)$, it can be divided into 4 sub-task, i.e. computing all possible sums in each quarter. In each sub-tasks, the algorithm can parallel computing each sum. When the algorithm finishes this part, synchronization is needed to ensure that 4 lists has been computed completely. In modular 4-way merge, each trial of κ_M can be computed parallelly. In each trial of κ_M, each trial of $List_L^{(1)}[i]$ can be completed parallelly. After that, a synchronization is needed to ensure each $List^{(1)}(\kappa_M)$ completes their computing.

Note that in Schroeppel-Samir algorithm, when computing the lists, all lists are mutual exclusion fields. In modular 4-way merge, every $List^{(1)}[\kappa_M]$ are mutual exclusion fields as well. Lock or other methods should be applied to ensure consistency. In the pseudo-code, we ignore the method to ensure consistency.

A formal version of parallel Schroeppel-Samir algorithm with modular 4-way merge is shown as follow:

```
Algorithm 8: Parallel Schroeppel-Samir algorithm

input:
$(a_1, a_2, \dots, a_n)$: knapsack vector
$s$: target sum
$l$: hamming weight of answer indices vector

Create lists $List^{(1)}(\kappa)$, $List^{(2)}(\kappa)$, $List^{(3)}(\kappa)$, $List^{(4)}(\kappa)$, $List^{(1)}(x)$, $List^{(2)}(x)$, $List^{(3)}(x)$
and $List^{(4)}(x)$
for processor i from 1 to 4 par-do
   for j=1 to $2^{n/4}$ par-do
      if the weight of j equals $l/4$
         Let vector $(x_1, \dots, x_{n/4})=(bit(j_1), \dots, bit(j_{n/4}))$
         insert $\sum_{k=1}^{n/4} x_k a_k$ into $List^{(i)}(\kappa)$
         insert $(x_1, \dots, x_n)$ into $List^{(i)}(x)$
      end if
   end for
end for
synchronization barrier
Call parallel modular 4-way merge with input $(List^{(1)}(\kappa), List^{(2)}(\kappa), List^{(3)}(\kappa), List^{(4)}(\kappa))$,
$n$ and $s$
Get set *solution*.
Create list *indices*
for o=1 to size of *solution*
   Get (i,j,k,l) from *solution*
```

```
    Compose $List^{(1)}(x)[i]$, $List^{(2)}(x)[j]$, $List^{(3)}(x)[k]$, $List^{(4)}(x)[l]$ into $indices$
end for
```

```
output:
$indices$: a list of answer indices vector
```

Algorithm 9: Parallel modular 4-way merge

```
input:
$(List_L^{(1)}, List_R^{(1)}, List_L^{(2)}, List_R^{(2)})$: 4 lists to be merged
$n$: the number of elements in knapsacks
$s$: target sum
```

```
Create lists $List_R^{(1)}(M)$ and $List_R^{(2)}(M)$
for i=1 to size of $List_R^{(1)}$ par-do
    insert $List_R^{(1)}(M)$ with tuple $(List_R^{(1)} \bmod M, i)$
end for
for i=1 to size of $List_R^{(2)}$ par-do
    insert $List_R^{(2)}(M)$ with tuple $(List_R^{(2)} \bmod M, i)$
end for
Sort $List_R^{(1)}(M)$ and $List_R^{(2)}(M)$ by left value in the tuple
Create list array $List^{(1)}[\kappa_M]$
Create list $solution$
for processor $\kappa_M = 0$ to $M - 1$ par-do
    for i=0 to size of $List_L^{(1)}$ par-do
        Let $\kappa_L^{(1)}$=$List_L^{(1)}[i]$
        Let $\kappa_t$=$(\kappa_M - \kappa_L^{(1)}) \bmod M$
        Search $\kappa_t$ in $List_R^{(1)}(M)$
        for each ($\kappa_t$,j) in $List_R^{(1)}(M)$ do
            Insert $(\kappa_L^{(1)} + List_R^{(1)}[j], (i, j))$ to $List^{(1)}[\kappa_M]$
        end for
    end for
    synchronization barrier for each processor
    Sort $List^{(1)}[\kappa_M]$ by left value
    for k=1 to size of $List_L^{(2)}$ par-do
        Let $\kappa_L^{(2)}$=$List_L^{(2)}[k]$
        Let $\kappa_t$=$(s - \kappa_M - \kappa_L^{(2)}) \bmod M$
        Search $\kappa_t$ in $List_R^{(2)}(M)$
        for each ($\kappa_t$,l) in $List_R^{(2)}(M)$ do
            Let $s' = s - \kappa_L^{(2)} - List_R^{(2)}[l]$
            Search $s'$ in $List^{(1)}[\kappa_M]$
            for each (s',(i,j)) in $List^{(1)}[\kappa_M]$ do
                Insert (i,j,k,l) into solution
            end for
        end for
    end for
end for
```

---

output:
Solution: a list of solutions

For time complexity, $2^{h(\alpha)n/2}$ still holds in parallel version. However, memory requirement will increase to $2^{h(\alpha)n/2}$. Because this parallel algorithm will construct all $List^{(1)}(\kappa_M)$ as an array at the same time, which is equivalent to construct full list of $List^{(1)} = \binom{n/4}{l/4}^2 = 2^{h(\alpha)n/2}$.

6.1.4 Other issues

High bit version

In addition to dividing $List^{(1)}$ by modulus M, a high bit 4-way merge can also works. Recall equation of 4-way merge

$$\kappa^{(1)} = \kappa_L^{(1)} + \kappa_R^{(1)} = s - \kappa_L^{(2)} - \kappa_R^{(2)}$$

The main idea is that, this algorithm will try to guess the higher $n/4$ bits of $\kappa^{(1)}$. For each trial of high bit of $\kappa^{(1)}$, high bit version searches in $List_L^{(1)}$ and $List_R^{(1)}$ satisfying the higher $n/4$ bits of $\kappa_L^{(1)} + \kappa_R^{(1)}$ equal to high bit of $\kappa^{(1)}$. Similarly, in $List_L^{(2)}$ and $List_R^{(2)}$, the algorithm search for these high bits too. Finally, by testing $s = \kappa_L^{(1)} + \kappa_R^{(1)} + \kappa_L^{(2)} + \kappa_R^{(2)}$, the algorithm output the solutions. The time and memory complexity would be the same as modular version, which is $2^{h(\alpha)n/2}$ and $2^{h(\alpha)n/4}$ respectively.

Early abort strategy

Early abort strategy can also applied to modular version of Schroeppel-Samir algorithm, if we only concern a single solution. As we analyze in section 5.1.2, the heuristic expected running time is $\tilde{O}(max(2^{h(\alpha)n/4}, 2^{h(\alpha)n/4}/\zeta))$.

6.2 Experiment results

To examine the correctness of analysis, two experiments are implemented.

The aim of first experiment is to test how modulus M effects running time. The first row of table indicates different values of M.

	240	280	310	330	350	380	420
Early abort	978.879	1021.748	1056.24	1034.43	1133.647	1294.25	1221.68
Total time	1957.487	2042.648	2117.35	2058.875	2271.36	2387.31	2451.84

Table 3.the experimental running time in seconds of different M, when $n = 44$, $l = 16$, using modular version of Schroeppel-Samir algorithm.

Experiment results reveal that when M closes to the size of 4 input lists (in this case is $\binom{11}{4} = 330$), algorithm obtains best running time.

Table 4 shows the running time of modular version of Schroeppel-Samir algorithm under different n and M is set to the size of 4 input lists. It can be seen, the running time increases exponentially.

	32	36	40	44	48
Early abort	6.133	36.872	188.66	1034.43	5664.90
Total time	12.262	73.991	374.341	2058.875	11323.812

Table 4.the experimental running time in seconds of different n, when $M = \binom{11}{4} = 330$, $l = 16$, using modular version of Schroeppel-Samir algorithm.

Note that we should not compare the running time of priority queue version and modular version of Schroeppel-Samir algorithm, because it depends on the detail when implemented. Bad coding may effect two same level algorithm greatly.

7 simple HGJ algorithm in good arrangement

7.1 Theoretical analysis

7.1.1 Time complexity

First, let us recall the main idea of simple HGJ algorithm. Instead of solving original knapsack problem directly, simple HGJ algorithm divide original problem into 4 sub-knapsack problem with original knapsack vector and target sum κ_1, κ_2, κ_3, κ_4. Where κ suffices

$$s = \kappa_1 + \kappa_2 + \kappa_3 + \kappa_4$$

To help understanding this equation, suppose that we have already known the answer subset of problem. Then if we further decompose the answer subset into 4 sets, the sum of every elements in each set always satisfies $s = \kappa_1 + \kappa_2 + \kappa_3 + \kappa_4$. In other words, every correct guess of $(\kappa_1, \kappa_2, \kappa_3, \kappa_4)$ stands for a decomposion of answer subset. Let Ω denotes all decomposition divided into 4 sets, then if good arrangement is provided

$$\Omega = \binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4$$

Using Stirling's fomular, and substitute $\alpha = l/n$

$$\Omega = 2^{2\alpha n}$$

To increase the correct rate of guessing, a modulus M is used. Namely, algorithm tries to search for a decomposition for the answer subset such that $\kappa_1 = R_1$ (mod M), $\kappa_2 = R_2$ (mod M), $\kappa_3 = R_3$ (mod M) and $\kappa_4 = R_4 = s - R_1 - R_2 - R_3$ (mod M).

Now, to estimate the number of decomposion found for a given M, let's come to our geme of balls and boxes. Obiviously, the number of balls is the number of decomposisions Ω. When the value of R_1, R_2, R_3 is determined, R_4 is fixed. And because the number of possible values for R_1, R_2 and R_3 are M. For all possible combination of R_1, R_2 and R_3, there are M^3 possilbe values, i.e. there are M^3 boxes. Now, we want to allocate Ω balls into M^3 boxes, the expected number of balls in each box is Ω/M^3. Namely, we expect there are Ω/M^3 decomposions of answer subset, for every trial R_1, R_2, R_3..

To make this heuristic expectation formally, Corollary 2 and Corollary 5 in paper [17] are needed. When applying Corollary 5 to our unbalanced case when good arrangement is given, the only difference with Corollary 5 is that the establishment of conditions $log_2(M) > (3log_2(3)/16)n$ should be changed into $\Omega^{1/3} > M > \Omega^{(3log_2(3)/16)} \approx \Omega^{0.2972}$. In other words, when $M > \Omega^{(3log_2(3)/16)}$ we believe there will be Ω/M^3 decomposions found for every trial R_1, R_2, R_3. Following the idea of [17], we extend the condition and belive that when $\Omega^{1/3} > M > \Omega^{0.25}$, our expected number will still hold.

To generate 4 lists containing decompositions of solution, 4 modular knapsack problems are constructed, whose modulus is M and target sums equal to R_1, R_2, R_3 and

R_4 respectively. When we get these 4 lists, then knapsack problem becomes a searching and matching problem which can be solved by 4 way merge algorithm.

When solving the modular sub-knapsack problems, we recall section 1.6.1 that modular knapsack problems and integer knapsack problems are equivalent in exponential time algorithm. Notice that there is a little difference in the strategy as we described in section 1.6.1, when applying to unbalanced case. Since there are l elements will add to our modular target sum, the range of target sum in integer is $[0, lM-1]$. Hence, if s is a value less than M, then modular unbalanced knapsack problem becomes l integer knapsack problem with target sum s, $s+M$, $\ldots$, $S+(l-1)M$.

Then by calling integer knapsack problem algorithm, we can obtain 4 lists containing decomposions of solution. As original problem is divided into 4 sub-knapsack problems, the hamming weight of 4 sub-knapsack should be $l/4$. In convenience of analysis, here, we use modular version of Schroeppel-Samir algorithm to solve sub-knapsack problem. Recall our time analysis in section 6.1.1. In the sub-knapsack problem, knapsack vector is divided into 4 quarters. Then in each quarter, the possible sum would be added by $l/16$ elements. Hence, the size of $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$ would be $\binom{n/4}{l/16}$, if good arrangement is provided. As the time complexity of Schroeppel-Samir algorithm is square of the size of inputed lists, then it would be $(\binom{n/4}{l/16})^2$.

However, depending on the choice of M, the number of solutions may be large, which may dominate the running time. There are $\binom{n/4}{l/16}^4$ possible sums in each sub-knapsack problem. We map these $\binom{n/4}{l/16}^4$ values into a size M space. Then there will be $\frac{\binom{n/4}{l/16}^4}{M}$ sums for value in M. In other words, we expect the number of solutions of sub-knapsack problem is $\frac{\binom{n/4}{l/16}^4}{M}$.

In a word, the time complexity to slove the sub-knapsack problem is $\tilde{O}(max(\binom{n/4}{l/16}^2, \frac{\binom{n/4}{l/16}^4}{M}))$. If using Stirling's formular, it will be

$$\tilde{O}(max(\binom{n/4}{l/16}^2, \frac{\binom{n/4}{l/16}^4}{M})) = \tilde{O}(max(2^{h(0.25\alpha)n/2}, 2^{h(0.25\alpha)n}/M))$$

In next step, we use 4-way merge algorithm to search collisions in the solution of 4 sub-knapsacks. Recall overlapping in section 3.4, if a indices vector overlaps, it can not be a valid solution. To check the overlapping, modular 4-way merge should be modified.

Algorithm 10: modular 4-way merge with overlap checking

input:

$(List_L^{(1)}, List_R^{(1)}, List_L^{(2)}, List_R^{(2)})$: 4 lists to be merged
$(List_L^{(1)}(x), List_R^{(1)}(x), List_L^{(2)}(x), List_R^{(2)})(x)$: 4 lists of indicis vector corresponding to lists to be merged
n: the number of elements in knapsacks
s: target sum

```
Create lists $List_R^{(1)}(M')$ and $List_R^{(2)}(M')$
for i=1 to size of $List_R^{(1)}$ do
  insert $List_R^{(1)}(M')$ with tuple $(List_R^{(1)} \bmod M', i)$
end for
for i=1 to size of $List_R^{(2)}$ do
  insert $List_R^{(2)}(M')$ with tuple $(List_R^{(2)} \bmod M', i)$
end for
Sort $List_R^{(1)}(M')$ and $List_R^{(2)}(M')$ by left value in the tuple
Create list $solution$
for $\kappa_{M'} = 0$ to $M - 1$ do
  Clear $List^{(1)}(\kappa_{M'})$
  for i=0 to size of $List_L^{(1)}$ do
    Let $\kappa_L^{(1)}$=$List_L^{(1)}[i]$
    Let $\kappa_t$=$(\kappa_M - \kappa_L^{(1)}) \bmod M'$
    Search $\kappa_t$ in $List_R^{(1)}(M')$
    for each ($\kappa_t$,j) in $List_R^{(1)}(M')$ do
      if ($List_L^{(1)}(x)[i]$ and $List_R^{(1)}(x)[j]$ do not overlap
        Insert $(\kappa_L^{(1)} + List_R^{(1)}[j], (i, j))$ to $List^{(1)}(\kappa_{M'})$
      end if
    end for
  end for
  Sort $List^{(1)}(\kappa_{M'})$ by left value
  for k=1 to size of $List_L^{(2)}$ do
    Let $\kappa_L^{(2)}$=$List_L^{(2)}[k]$
    Let $\kappa_t$=$(s - \kappa_{M'} - \kappa_L^{(2)}) \bmod M'$
    Search $\kappa_t$ in $List_R^{(2)}(M')$
    for each ($\kappa_t$,l) in $List_R^{(2)}(M')$ do
      if ($List_L^{(2)}(x)[k]$ and $List_R^{(2)}(x)[l]$ do not overlap
        Let $s' = s - \kappa_L^{(2)} - List_R^{(2)}[l]$
        Search $s'$ in $List^{(1)}(\kappa_{M'})$
        for each (s',(i,j)) in $List^{(1)}(\kappa_{M'})$ do
          if ($List_L^{(1)}(x)[i]$, $List_R^{(1)}(x)[j]$,($List_L^{(2)}(x)[k]$ and $List_R^{(2)}(x)[l]$ do not overlap
            Insert (i,j,k,l) into $solution$
            return $solution$
          end if
        end for
```

```
            end if
        end for
    end for
end for
```

---

output:
Solution: a list of solutions

In order to distinguish the modulus when used in 4-way merge and solving sub-knapsack problem, we denote the modulus in modular 4-way merge as M'. Actually, there are only several difference between original modular 4-way merge and the overlap checking version. First, to check overlapping, 4-way merge needs to know indices vector corresponding to the sums. Second, when every time two lists merge, check whether elements in list overlap. Rather than check overlapping when inserting *solution* once only, checking overlaps when inserting $List^{(1)}(\kappa_{M'})$, implict list $List^{(2)}(\kappa_{M'})$ and *solution* three times per literation is prefered. Since prevent inserting the list in early stage, will be helpful to reduce expansion in list size. Third, early abort strategy should be applied. Depending on the choice of M in last stage, the number of decompositions containing in the $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$ may be in exponential size which will effect the running time. Note that, the bound of our analysis will hold as long as early abort strategy is applied.

In original paper [17], it suggests algorithm to pick a random $\kappa_{M'}$, instead of try every $\kappa_{M'}$ in order. But we do not think that is a better choice. Though, the expected running time are the same, to pick $\kappa_{M'}$ in order will let algorithm be bound to stop after M' times iterations. If, in last stage, algorithm fail to guess correct values of κ_1, κ_2, κ_3 and κ_4, randomized approach will never end.

Let us come back to our complexity analysis. Recall that the size of solutions output from sub-knapsack problem is $\frac{\binom{n/4}{l/16}^4}{M}$), i.e. the list size input into modular 4-way merge is $\frac{\binom{n/4}{l/16}^4}{M}$). For a full round of modular 4-way merge algorithm, the time complexity is the square of input list size, i.e. $(\frac{\binom{n/4}{l/16}^4}{M})^2$. In order to distinguish the ζ used in sub-knapsack problem, here we use ζ' denotes number of solutions when merging the solution, i.e. the number of decompositions of solution for a given trial of κ_1, κ_2, κ_3 and κ_4. From our previous analysis, this expected number of decompositions is $\zeta' = \Omega/M^3 = \frac{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4}{M^3}$.

If $\zeta' < 1$, this expected number of decompositions means the probability, for a given M, , which pick 4 random values and these 4 value can be represented as a decomposion of solution. In this case, we heuristically believe that running simple HGJ algorithm once will not guess the values of κ_1, κ_2, κ_3 and κ_4 correctly. Actually, we expect by running the algorithm the reciprocal of probability times (i.e. M^3/Ω) will find a correct combination of κ_1, κ_2, κ_3 and κ_4. If $\zeta' > 1$, we expect there are more than one decompositions of answer subset can be found, when we apply 4-way merge. Thus, we can use early abort strategy and expect 4-way merge will stop at $1/\zeta$ of the time of whole progress.

We summarize that if using early abort strategy or running algorithm repeatedly, a solution will be found at time $(\frac{\binom{n/4}{l/16}^4}{M})^2/\zeta' = (\frac{\binom{n/4}{l/16}^4}{M})^2/\frac{\binom{l/4}{l/16}^4*\binom{3l/16}{l/16}^4*\binom{l/8}{l/16}^4}{M^3} = \frac{\binom{n/4}{l/16}^8*M}{\binom{l/4}{l/16}^4*\binom{3l/16}{l/16}^4*\binom{l/8}{l/16}^4}$.
Using Stirling's fomular, it will be

$$\tilde{O}(\frac{\binom{n/4}{l/16}^8 * M}{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4}) = \tilde{O}(2^{2h(0.25\alpha)n-2\alpha n} * M)$$

To composite the time complexity of two stage, we have $\tilde{O}(max(\binom{n/4}{l/16}^2, \frac{\binom{n/4}{l/16}^4}{M}, \frac{\binom{n/4}{l/16}^8*M}{\binom{l/4}{l/16}^4*\binom{3l/16}{l/16}^4*\binom{l/8}{l/16}^4}))$.
Next, we prove that as long as $\Omega^{1/3} \geq M \geq \Omega^{0.25}$ (i.e. in the range of $[2^{(2/3)\alpha n}, 2^{0.5\alpha n}]$), $\frac{\binom{n/4}{l/16}^8*M}{\binom{l/4}{l/16}^4*\binom{3l/16}{l/16}^4*\binom{l/8}{l/16}^4} \geq \frac{\binom{n/4}{l/16}^4}{M}$ is always true.
To prove:

$$\begin{aligned} & \frac{\binom{n/4}{l/16}^8*M}{\binom{l/4}{l/16}^4*\binom{3l/16}{l/16}^4*\binom{l/8}{l/16}^4} \geq \frac{\binom{n/4}{l/16}^4}{M} \\ \iff & 2^{2h(0.25\alpha)n-2\alpha n} * M \geq 2^{h(0.25\alpha)n}/M \\ \iff & M \geq 2^{\alpha n - h(0.25\alpha)n/2} \end{aligned}$$

Because

$$M \geq 2^{0.5\alpha n}$$

To prove

$$M \geq 2^{\alpha n - h(0.25\alpha)n/2}$$

is equivalent to prove

$$\begin{aligned} & 2^{0.5\alpha n} \geq 2^{\alpha n - h(0.25\alpha)n/2} \\ \iff & 2^{h(0.25\alpha)n/2-0.5\alpha n} \geq 1 \\ \iff & 2^{-0.25\alpha log_2(0.25\alpha)-(1-\alpha/4)log_2(1-\alpha/4)-\alpha n} \geq 1 \\ \iff & \alpha \leq 0.6257 \end{aligned}$$

Thanks to complementary knapsack, we can imply that $\alpha \leq 0.6257$ is always true. Thus, $\frac{\binom{n/4}{l/16}^8*M}{\binom{l/4}{l/16}^4*\binom{3l/16}{l/16}^4*\binom{l/8}{l/16}^4} \geq \frac{\binom{n/4}{l/16}^4}{M}$ is always true.

Similarly, we can imply that $\binom{n/4}{l/16}^2 \leq \frac{\binom{n/4}{l/16}^8*M}{\binom{l/4}{l/16}^4*\binom{3l/16}{l/16}^4*\binom{l/8}{l/16}^4}$ is always true. Thus, our time complexity of simple HGJ algorithm becomes

$$\tilde{O}(max(\binom{n/4}{l/16}^2, \frac{\binom{n/4}{l/16}^4}{M}, \frac{\binom{n/4}{l/16}^8 * M}{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4})) = \tilde{O}(\frac{\binom{n/4}{l/16}^8 * M}{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4}) \tag{8}$$

Depending on the choice of M where $2^{(2/3)\alpha n} \leq M \leq 2^{0.5\alpha n}$, our time complexity is in the range of

$$[\tilde{O}(2^{h(0.25\alpha)n-1.5\alpha n}), \tilde{O}(2^{h(0.25\alpha)n-(4/3)\alpha n})]$$

Next, we prove that the time complexity of simple HGJ algorithm is always better than that of Schroeppel-Samir algorithm, i.e. to prove

$$\begin{aligned} &\tilde{O}(2^{h(0.25\alpha)n-(4/3)\alpha n}) < \tilde{O}(2^{h(\alpha)n/2}) \\ \iff &(0.25\alpha)log_2(0.25\alpha) + (1-0.25\alpha)(log_2(1-0.25\alpha) \\ &+(4/3)\alpha - \alpha log_2\alpha - (1-\alpha)log_2(1-\alpha)) > 0 \end{aligned}$$

Computing derivation we find that this is a monotonically increasing function when $0 < \alpha \leq n/2$. So when we let α close to 0, it will get the minimal value which is still larger than 0. As a consequence, we conclude that simple HGJ algorithm is always faster than Schroeppel-Samir algorithm.

7.1.2 Memory complexity

When memory is concerned, according to the analysis in section 3.2, it will occupy $\tilde{O}(max(\binom{n/4}{l/16}, \zeta))$ memory, when solving sub-knapsacks. The memory complexity to merge the solutions of these sub-knapsack problem is the same size as input lists, i.e. ζ. Hence the total memory requirement is $\tilde{O}(max(\binom{n/4}{l/16}, \zeta))$. Using Stirling's fomular, $\binom{n/4}{l/16}$ equals to $2^{h(\alpha/4)n/4}$ and the number of solutions $\zeta = \frac{\binom{n/4}{l/16}^4}{M} = \frac{2^{h(\alpha/4)n}}{M}$. We prove that $\zeta = \frac{2^{h(\alpha/4)n}}{M}$ is always larger than $2^{h(\alpha/4)n/4}$.

To prove:

$$\frac{2^{h(\alpha/4)n}}{M} > 2^{h(\alpha/4)n/4}$$

is equivalent to prove

$$M < 2^{(3/4)h(\alpha/4)} \text{ is always true}$$

Because M is in the range of $[\Omega^{0.25}, \Omega^{1/3}] = [2^{0.5\alpha n}, 2^{\alpha n*(2/3)}]$. Hence, to prove original inequality is equivalent to prove

$$\begin{aligned} &2^{\alpha n*(2/3)} < 2^{(3/4)h(\alpha/4)} \text{ always true} \\ \iff &h(\alpha/4) - (8/9)\alpha > 0 \end{aligned}$$

By computation, we know when α closes to 0, $h(\alpha/4) - (8/9)\alpha$ will close to its minimal value which is larger than 0. Hence, that the time complexity when solving sub-knapsack problem is

$$\tilde{O}(max(\binom{n/4}{l/16}, \zeta)) = \tilde{O}(\zeta) = \tilde{O}(\frac{\binom{n/4}{l/16}^4}{M}) \tag{9}$$

Depending on the choice of M where $2^{(2/3)\alpha n} \leq M \leq 2^{0.5\alpha n}$, its memory complexity is in the range of

$$[\tilde{O}(2^{h(0.25\alpha)n-0.5\alpha n}), \tilde{O}(2^{h(0.25\alpha)n-(2/3)\alpha n})]$$

From equation (8) and equation (9), we can see there is a trade-off between time and memory complexity. For a experiment result, see section 7.2.

7.1.3 Parallel

Simple HGJ algorithm can be modified into parallel version. When computing all possible sums for sub-knapsacks, it is similar with parallel Schroeppel-Samir algorithm. Differece is that modular sums should be inserted into another 4 lists. When sloving sub-knapsack problems, according to section 1.6.1, every modular knapsack problem can be divided into $l/4$ integer knapsack problems. We have 4 modular sub-knapsack problems. Then we solve problem by first compute 4 modular knapsack problem parallelly. In each modular knapsack problem, we solve $l/4$ integer knapsack problems parallelly. Note that, $List^{(i)}(\kappa)$, $List^{(i)}(\kappa)(M)$, $List^{(i)}(x)$, $List^{(o)}(x)'$ and $List^{(o)}(\kappa)'$ are mutual exclusion fields and moreover the sequence of $List^{(i)}(\kappa)$, $List^{(i)}(\kappa)(M)$ and $List^{(i)}(x)$ should be the same. Similarly, the sequence of $List^{(o)}(x)'$ and $List^{(o)}(\kappa)'$ should be guaranteed as well. So synchronous and lock technique should be used, while inserting these lists.

When algorithm completes computing sub-knapsacks, a barrier should be applied to stop all processes and ensure the integrity of $List^{(o)}(x)'$ and $List^{(o)}(\kappa)'$. When merging the solutions of sub-knapsacks, we use parallel modular 4-way merge with overlap checking and early abort strategy. Although, in this paper, we do not include this algorithm. But by combining the idea of Algorithm 10 and Algorithm 8, it can be come up easily.

Algorithm 11: Parallel simple HGJ algorithm

```
input:
$(a_1, a_2, \ldots, a_n)$: knapsack vector
$s$: target sum
$l$: hamming weight of answer indices vector
$M$: modulus

Create lists $List^{(1)}(\kappa)$, $List^{(2)}(\kappa)$, $List^{(3)}(\kappa)$, $List^{(4)}(\kappa)$, $List^{(1)}(\kappa)(M)$, $List^{(2)}(\kappa)(M)$
, $List^{(3)}(\kappa)(M)$ , $List^{(4)}(\kappa)(M)$, $List^{(1)}(x)$, $List^{(2)}(x)$, $List^{(3)}(x)$ and $List^{(4)}(x)$
for processor i from 1 to 4 par-do
  for j=1 to $2^{n/4}$ par-do
    if the weight of j equals $l/4$
      Let vector $(x_1, \ldots, x_{n/4})$=$(bit(j_1), \ldots, bit(j_{n/4}))$
```

```
            insert $\sum_{k=1}^{n/4} x_k a_k$ into $List^{(i)}(\kappa)$
            insert $\sum_{k=1}^{n/4} x_k a_k \bmod M$ into $List^{(i)}(\kappa)(M)$
            insert $(x_1, \ldots, x_n)$ into $List^{(i)}(x)$
        end if
    end for
end for
synchronization barrier
Generate $R_1$,$R_2$ and $R_3$ randomly in the range of $[0, M-1]$
Let $R_4 = s - R_1 - R_2 - R_3$
Create list $indices$, $List^{(1)}(\kappa)'$, $List^{(2)}(\kappa)'$, $List^{(3)}(\kappa)'$, $List^{(4)}(\kappa)'$, $List^{(1)}(x)'$, $List^{(2)}(x)'$, $List^{(3)}(x)'$ and $List^{(4)}(x)'$
while $indices$ is empty
    Clear $List^{(1)}(\kappa)'$, $List^{(2)}(\kappa)'$, $List^{(3)}(\kappa)'$, $List^{(4)}(\kappa)'$, $List^{(1)}(x)'$, $List^{(2)}(x)'$, $List^{(3)}(x)'$ and $List^{(4)}(x)'$
    for processor o from 1 to 4 par-do
        for p=1 to $l/4$ par-do
            Call parallel modular 4-way merge with input $(List^{(1)}(\kappa)(M)$, $List^{(2)}(\kappa)(M)$, $List^{(3)}(\kappa)(M)$, $List^{(4)}(\kappa)(M))$,$n$ and $R_o + pM$
            Get returned set $solution(o * 4 + p)$
            for i=1 to size of $solution(o * 4 + p)$
                Get (i,j,k,l) from $solution(o * 4 + p)$
                Compose $List^{(1)}(x)[i]$, $List^{(2)}(x)[j]$, $List^{(3)}(x)[k]$, $List^{(4)}(x)[l]$ into $List^{(o)}(x)'$
                Insert $List^{(1)}(\kappa)[i]$+$List^{(2)}(\kappa)[j]$+ $List^{(3)}(\kappa)[k]$+ $List^{(4)}(\kappa)[l]$ into $List^{(o)}(\kappa)'$
            end for
        end for
    end for
    synchronization barrier
    Call parallel modular 4-way merge with overlap checking and early abort strategy and input $(List^{(1)}(\kappa)'$, $List^{(2)}(\kappa)'$, $List^{(3)}(\kappa)'$, $List^{(4)}(\kappa)')$, $(List^{(1)}(x)'$, $List^{(2)}(x)'$, $List^{(3)}(x)'$, $List^{(4)}(x)')$ ,$n$ and $s$
    for i=1 to size of $solution$
        Get (i,j,k,l) from $solution$
        Compose $List^{(1)}(x)'[i]$, $List^{(2)}(x)'[j]$, $List^{(3)}(x)'[k]$, $List^{(4)}(x)'[l]$ into $indices$
    end for
end while
```

output:
indices: a list of answer indices vector

7.2 Experiment

7.2.1 Simulation of decomposition distribution

To examine our analysis, we start experiment by simulating the distribution of decompositions of solution. Namely, for given values of R_1, R_2, R_3 and $R_4 = s - R_1 - R_2 - R_3 \bmod M$, if dividing answer subset into 4 sets, we caculate the number of such decompositions where the sum of first set equals to R_1, the sum of second set equals to R_2, the sum of third set equals to R_3 and the sum of forth set equals to R_4. Compared with simulating and caculating the size of lists by using HGJ algorithm, using algorithm 12 would be much faster and thus can be used to run many times to see statistical probability.

Simulating algorithm is outline as follow:

Algorithm 12: Simulation of decomposition distribution

input:
$(a'_1, a'_2, \ldots, a'_l)$: answer subset
l: hamming weight of answer indices vector
s: target sum
R_1, R_2, R_3: the conditions of decompositions should be satisfied
M: modulus

```
Create list $solution$
for i=1 to $2^l$ do     if hamming weight of $(bit(i_1), \ldots, bit(i_{l/4}))$
      if hamming weight of $(bit(i_{l/4+1}), \ldots, bit(i_{l/2}))$
         if hamming weight of $(bit(i_{l/2+1}), \ldots, bit(i_{3l/4}))$
            if hamming weight of $(bit(i_{3l/4+1}), \ldots, bit(i_l))$
               Insert $(\sum_{j=1}^{n} x_j a'_j \bmod M, (bit(i_1), \ldots, bit(i_l)))$ into $solution$
            end if
         end if
      end if
   end is
end for
Create list $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$
$R_4 = s - R_1 - R_2 - R_3$
for i=1 to size of $solution$ do
   if left member of $solution[i] = R1$
      Insert $solution[i]$ into $List_L^{(1)}$
   end if
   if left member of $solution[i] = R2$
      Insert $solution[i]$ into $List_R^{(1)}$
   end if
```

```
    if left member of $solution[i] = R3$
        Insert $solution[i]$ into $List_L^{(2)}$
    end if
    if left member of $solution[i] = R4$
        Insert $solution[i]$ into $List_R^{(2)}$
    end if
end for
Creat list $List^{(1)}$ and $List^{(2)}$
for i=1 to size of $List_L^{(1)}$
    for j=1 to size of $List_R^{(1)}$
        if $List_L^{(1)}$ and $List_R^{(1)}$ does not overlap
            Insert ($List_L^{(1)}[i]$, $List_R^{(1)}[j]$ into $List^{(1)}$
        end if
    end for
end for
for i=1 to size of $List_L^{(2)}$
    for j=1 to size of $List_R^{(2)}$
        if $List_L^{(2)}$ and $List_R^{(2)}$ does not overlap
            Insert ($List_L^{(2)}[i]$, $List_R^{(2)}[j]$ into $List^{(2)}$
        end if
    end for
end for
Create list $indices$
for i=1 to size of $List^{(1)}$
    for j=1 to size of $List^{(2)}$
        if $List^{(1)}$ and $List^{(2)}$ does not overlap
            Insert ($List^{(1)}[i]$, $List^{(2)}[j]$ into $indices$
        end if
    end for
end for
```

```
output:
the size of $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$
the size of $List^{(1)}$ and $List^{(2)}$
the size of $indices$
```

Using this algorithm, we can check the analysis of average execution times of simple HGJ algorithm. According to our analysis, it is $\frac{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4}{M^3}$. Table 5 indicates when choosing different M, experiment results conform our theoretical analysis.

M	35	50	69	80
Theoretical analysis	7.73	2.654	1.01	0.648
Experiment result	7.52	2.503	0.99	0.693

Table 5.the number of decompositions for every combinations of R_1, R_2, R_3 and $R_4 = s - R_1 - R_2 - R_3$ in different M, when $l = 16$.

Recall in Algorithm 10: modular 4-way merge with overlap checking. We remove overlapping elements everytime elements are inserted to $List^{(1)}(\kappa_{M'})$, $List^{(2)}(\kappa_{M'})$ (constructed implictly) and *solution*, rather than remove overlapping elements once only when inserting to *solution*. Algorithm 12 examine the effect of 3 times removing. The size of *solution* when removing overlapping only once, is computed by the product of the size of $List_L^{(1)}$, $List_R^{(1)}$, $List_L^{(2)}$ and $List_R^{(2)}$. The size of list when using 3 times removing, is computed by maximal size of $List^{(1)}$, $List^{(2)}$ and *indices*. First row of table indicates values of M.

M	40	50	69	80
$List_L^{(1)}$	7.3	4.82	3.71	2.51
$List_R^{(1)}$	7.23	4.6	3.72	2.47
$List_L^{(2)}$	7.54	5.35	3.73	2.96
$List_R^{(2)}$	7.08	4.9	3.8	2.34
$List^{(1)}$	16.72	6.7	4.06	2.11
$List^{(2)}$	17.44	8.25	4.32	2.32
One time removing	2817.51	581.239	195.618	42.94
Three times removing	17.44	8.25	4.32	2.96

Table 6.the size of lists when using different strategies, implemented in different M, when $l = 16$.

From Table 6, we can see three time removing help to reduce the size of lists greatly when implemented in Algorithm 10.

7.2.2 Implementing simple HGJ algorithm

To exam our analysis of simple HGJ algorithm, we implement experiment.

Table 7 reveals that the size of solutions when solving 4 sub-knapsack problems, when input by different M. According to our analysis, we expect their size is $\frac{\binom{n/4}{l/16}^4}{M}$.

M	40	50	69	80
Theoretical analysis	366	293	212	183
Experiment result	370	305	209	172

Table 7.the size of solutions of 4 sub-knapsacks, implemented in different M, when $l = 16$ and $n = 44$.

Table 8 shows that time and memory trade-off among different M.

In our analysis, we recall that our time complexity is $\tilde{O}(\frac{\binom{n/4}{l/16}^8 * M}{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4})$. In our experiment, we let $n = 44$, $l = 16$, then the time complexity is that $\tilde{O}(646M)$. Memory complexity of simple HGJ algorithm is $\tilde{O}(\frac{\binom{n/4}{l/16}^4}{M})$. Then substituting experimental parameter, it becomes $\tilde{O}(\frac{14641}{M})$.

The first option is to minimize the time complexity by picking M as small as possilbe which should be $\sqrt[4]{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4} = 24$. To compare with Schroeppel-Samir algorithm, we let simple HGJ run in the same memory with Schroeppel-Samir algorithm, i.e. $M = 44$. Third option is picking $M = \binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^{4^{0.2936}} = 41$, which can be used to prove the algorithm. The final option is to minimize the memory requirement by maximize M close to (1/3), when $\sqrt[3]{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4} = 69$.

M	24	41	44	69
Running time	155.81	189.12	189.87	213.45
Memory	2.6m	2.5m	2.5m	2.5m

Table 8.Running time in seconds and memory in bytes, implemented in different M, when $l = 16$ and $n = 44$.

Table 9 shows the running time according to different n and same $l = 16$

n	32	36	40	44
Memory	1.5m	1.8m	2.1m	2.5m
Total time	2.1	8.457	52.45	213.45

Table 8.Running time in seconds and memory in bytes, implemented in different n, when $l = 16$ and $M = 41$.

8 Critical evaluation

The main work of this paper is to analyze Schroeppel-Samir algorithm and HGJ algorithm in unbalanced case and support the analysis with experiments. Here are some conclusions:

In priority queue Schroeppel-Samir algorithm:

1. When good arrangement of knapsack vector is provided, the time complexity is

$$\tilde{O}(\binom{n/4}{l/4}^2)$$

and memory complexity is

$$\tilde{O}(\binom{n/4}{l/4})$$

2. When using early abort strategy, the time complexity becomes

$$\tilde{O}(\binom{n/4}{l/4}^2/\zeta)$$

where ζ is the number of solutions.

3. When there are a large number of solutions, original priority queue Schroeppel-Samir algorithm may fail to find out all solutions. We improve it by a modified algorithm.

In modular Schroeppel-Samir algorithm:

4. The time and memory complexity is also $\tilde{O}(\binom{n/4}{l/4}^2)$ and $\tilde{O}(\binom{n/4}{l/4})$ respectively.

5. Best choice of modulus used in 4-way merge should be the same size of input lists, i.e.

$$\tilde{O}(\binom{n/4}{l/4})$$

6. A parallel version of modular Schroeppel-Samir algorithm is presented.

In HGJ algorithm:

7. The time complexity is

$$\tilde{O}(\frac{\binom{n/4}{l/16}^8 * M}{\binom{l/4}{l/16}^4 * \binom{3l/16}{l/16}^4 * \binom{l/8}{l/16}^4})$$

and memory complexity is

$$\tilde{O}(\frac{\binom{n/4}{l/16}^4}{M})$$

8. The time complexity of HGJ algorithm is always better than Schroeppel-Samir algorithm.

9. A parallel version of HGJ algorithm is presented.

Although, I believe that I have achieved the goal I set at the beginning of this dissertation. There are several defects

1. All the theoretical analysis is obtained on the basis of list size. But sometimes list size can not stand for complexity completely. For example, in HGJ algorithm, we believe that sub-knapsack problems can be solved within the time of size of solution. This assumption may not be satisfied in some situations.

2. Dad coding result in great difference in running time, which leads to modular version of Schroeppel-Samir algorithm can not compare to priority version.

3. Due to time constraints, all experiment is implemented in a small n, which makes experimental result not as obvious as thought.

9 Further work

Although, this dissertation fully discusses two version of Schroeppel-Samir algorithm and simple version of HGJ algorithm in unbalanced case. There are still some tasks left when solving unbalanced knapsack problem:

1. Recursive version of HGJ algorihm. Because of time constraints, we do not cover the analysis of recursive version. In recursive version, algorithm recursively divide original problem into 2 sub-knapsacks. Actually, according to [17], recursive version is faster than simple version.

2. In 2011, Becker, Coron and Joux [18] improved a generic algorithm and got running time down to $O(2^{0.291n})$. This algorithm represent coefficients of answer indices vector from (0,1) in HGJ algorithm to (-1,0,1), which introduces additional degree of freedom that decrease the running time. It is worthy of study to apply this algorithm to unbalanced case.

References

[1] B. L. Dietrich, L. F. Escudero, and F. Chance.Efficient reformulation for 01 programs: methods and computational results, Discrete Appl.*Math*, 42 (23). pages 147–175, 1993.

[2] C. Gentry.Fully homomorphic encryption using ideal lattices.*Symposium on the Theory of Computing (STOC)*, pages 169–178.

[3] M. R. Gentry and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP - Completeness.W. H. Freeman and Company, San Francisco, 1979.

[4] R. Merkle and M. Hellman. Hiding information and signatures in trapdoor knapsacks.*IEEE Trans. Information Theory*, 24(5):pages 525–530,1978.

[5] A. Shamir. A Polynomial Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem.*IEEE Trans. Inform. Theory*,vol. IT-30,pages 699–704,1984.

[6] J. C. Lagarias and A. M. Odlyzko. Solving Low Density Subset Sum Problems.*J. Assoc. Comp. Mach*,vol. 32.pages 229–246,1985.Preliminary version in *Proc. 24th IEEE Symposium on Foundations of Computer Science*,pages 1–10,1983.

[7] M. J. Coster, A. Joux, B. A. Lamacchia, A. M. Odlyzko,C. Schnorr and J. Stern. Improved low-density subset sum algorithms.*Computational Complexity*,2:,pages 111–128,1992.

[8] R. Impagliazzo and M. Naor. Efficient Cryptographic Schemes Provably as Secure as Subset Sum.*Proc. 30th IEEE Symposium on Foundations of Computer Science*,IEEE,pages 236–241,1989.

[9] M. Ajtai.The shortest vector problem in L2 is NP-hard for randomized reductions(extended abstract).*In 30th ACM STOC*,pages 10–19,1998.

[10] A. K. Lenstra, H. W. Lenstra,Jr., and L. Lovasz.Factoring polynomialswith rational coecients. *Math.Ann*,261:pages 515–534,1982.

[11] C. P. Schnorr.A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*,53:pages 201–224,1987.

[12] F. Jorissen, J. Vandewalle, and R. Govaerts.Extension of Brickell's algorithm for breaking high density knapsacks. *Proc. of Eurocrypt'87, LNCS* 304 pages 109–115,1988.

[13] A. Joux, and J. Stern. Improving the critical density of the Lagarias-Odlyzko attack against subset sum problems. *Proc. of Fundamentals of Computation Theory'91, LNCS* 529 pages 258–264,1991.

[14] A. Joux, and L. Granboulan. A practical attack against knapsack based hash functions (extended abstract). In Alfredo De Santis, editor, *EUROCRYPT'94, volume 950 of LNCS* 529 pages 58–66,1994.

[15] R. Schroeppel, and A. Shamir. A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems. In *FOCS*, pages 328–336,1979.

[16] R. Schroeppel, and A. Shamir. A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems. SIAM *Journal on Computing*,10(3) pages 456–464,1981.

[17] N. Howgrave-Graham, and A. Joux. New generic algorithms for hard knapsacks. In *EUROCRYPT'2010*, pages 235–256,2010.

[18] A. Becker, J. Coron and A. Joux.Improved Generic Algorithms for Hard Knapsacks. In *EUROCRYPT'2011*, 2011.

[19] H. S. Wilf.Backtrack: An $O(1)$ Expected Time Graph Coloring Algorithm. In *Inform. Proc. Letters*,vol. 18,pages 119–122, 1984.

[20] G. Brassard.A note on the complexity of cryptography. *IEEE Trans. lnformat. Theory*,vol. IT-25,pages 232–233, 1979.

[21] A. M. Odlyzko.The rise and fall of knapsack cryptosystems. In *Cryptology and computational number theory (Boulder, CO, 1989), volume 42 of Proc. Sympos. Appl. Math*, pages 75–88, 1990.

[22] C. Gentry.A fully homomorphic encryption scheme.PhD Thesis stanford university, 2009.

[23] D. R. Stinson. Some baby-step giant-step algorithms for the low hamming weight discrete logarithm problem. *Mathematics of Computation*, vol.71, no.237, pages 379–391, 2001.

[24] D. Coppersmith and G. Seroussi.On the minimum distance of some quadratic residue codes. In *IEEE Trans*, Inform. Theory 30, pages 407–411, 1984.

[25] C. Gentry and S. Halevi. Implementing Gentry's Fully-Homomorphic Encryption Scheme. Extended abstract in *EUROCRYPT 2011*, 2011.

[26] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *J. Assoc. Comp. Mach*, 21(2), pages:277–292, 1974.

[27] May, A., Meurer, A.: Personal communication.

Appendix

Due to a large amount of code, we just paste essential parts of your code, here. For a full version, see source file. The code of priority queue 4-way merge , modular 4-way merge and the main body of simple HGJ algorithm is presented in appendix.

Priority queue 4-way merge

```
vector<vec_ZZ> queue_4_way_merge(vector<ZZ> SL1_tau,vector<ZZ> SR1_tau,vector<ZZ>
SL2_tau,vector<ZZ> SR2_tau,vector<vec_ZZ> SL1_xita,vector<vec_ZZ>
SR1_xita,vector<vec_ZZ> SL2_xita,vector<vec_ZZ> SR2_xita,long N,ZZ
TargetSum,vector<ZZ> Sol,double t){
	vector<vec_ZZ> Sol_xita;
	priority_queue<NODE> Q1,Q2;
	int truth=0;
	int k,l,m,n;
	long aa,bb,cc;
	for(k=0;k<SL1_tau.size();k++){
		Q1.push(NODE(SL1_tau[k]+SR1_tau[0],k,0));
	}
	for(k=0;k<SL2_tau.size();k++){

	Q2.push(NODE(TargetSum-SL2_tau[k]-SR2_tau[SR2_tau.size()-1],k,SR2_tau.size()-1))
;
	}
	cout<<"Part 2 finishes!"<<endl;
	ZZ q1_,q2_;
	while(!(Q1.empty())&&!(Q2.empty())){
		q1_=Q1.top().priority;
		q2_=Q2.top().priority;
		if(q1_<=q2_){
			k=Q1.top().i;
			l=Q1.top().j;
			Q1.pop();
			if(l!=SR1_tau.size()-1){
				Q1.push(NODE(SL1_tau[k]+SR1_tau[l+1],k,l+1));
			}
		}
		if(q1_>=q2_){
			m=Q2.top().i;
			n=Q2.top().j;
			Q2.pop();
```

```
				if(n!=0){
					Q2.push(NODE(TargetSum-SL2_tau[m]-SR2_tau[n-1],m,n-1));
				}
			}
			if(q1_==q2_){
				cout<<"get answer! at "<<GetTime()-t<<endl;
				Sol.push_back(TargetSum);
				Sol_xita.push_back(SL1_xita[k]+SR1_xita[l]+SL2_xita[m]+SR2_xita[n]);
				cout<<Sol_xita[0]<<endl;
			}
		}
		return Sol_xita;
	}
```

Modular 4-way merge

```
for(i=0;i<SR1.size();i++){
			temp=SR1[i]%M;
			SR1_M.push_back(NODE(temp,i));
		}
		for(i=0;i<SR2.size();i++){
			temp=SR2[i]%M;
			SR2_M.push_back(NODE(temp,i));
		}
sort(&SR1_M,0,SR1_M.size());
sort(&SR2_M,0,SR2_M.size());
for(tau_M=0;tau_M<M;tau_M++){
			S1.clear();
			cout<<"tau_M="<<tau_M<<endl;
			for(i=0;i<SL1.size();i++){
				tau_L1=SL1[i];
				tau_t=(tau_M-tau_L1)%M;
				pos=half(SR1_M,0,SR1_M.size()-1,tau_t);
				if(pos>=0){
					S1.push_back(NODE2(tau_L1+SR1[SR1_M[pos].i],i,SR1_M[pos].i));
					for(k=pos+1;k<SR1_M.size()&&SR1_M[k].value==tau_t;k++){
						S1.push_back(NODE2(tau_L1+SR1[SR1_M[k].i],i,SR1_M[k].i));
					}
					for(k=pos-1;k>=0&&SR1_M[k].value==tau_t;k--){
						S1.push_back(NODE2(tau_L1+SR1[SR1_M[k].i],i,SR1_M[k].i));
					}
				}
			}
			sort(&S1,0,S1.size());
			for(k=0;k<SL2.size();k++){
				tau_L2=SL2[k];
```

```
tau_t=(T-tau_M-tau_L2)%M;
pos=half(SR2_M,0,SR2_M.size()-1,tau_t);
if(pos>=0){
        T_=T-tau_L2-SR2[SR2_M[pos].i];
        pos2=half(S1,0,S1.size()-1,T_);
        if(pos2>=0){
                temp2=new int [4];
                temp2[0]=S1[pos2].i;
                temp2[1]=S1[pos2].j;
                temp2[2]=k;
                temp2[3]=SR2_M[pos].i;
                Sol.push_back(temp2);
                for(j=pos2+1;j<S1.size()&&S1[j].value==T_;j++){
                        temp2=new int [4];
                        temp2[0]=S1[j].i;
                        temp2[1]=S1[j].j;
                        temp2[2]=k;
                        temp2[3]=SR2_M[pos].i;
                        Sol.push_back(temp2);
                }
                for(j=pos2-1;j>=0&&S1[j].value==T_;j--){
                        temp2=new int [4];
                        temp2[0]=S1[j].i;
                        temp2[1]=S1[j].j;
                        temp2[2]=k;
                        temp2[3]=SR2_M[pos].i;
                        Sol.push_back(temp2);
                }
        }
        for(l=pos+1;l<SR2_M.size()&&SR2_M[l].value==tau_t;l++){
                T_=T-tau_L2-SR2[SR2_M[l].i];
                pos2=half(S1,0,S1.size()-1,T_);
                if(pos2>=0){
                        temp2=new int [4];
                        temp2[0]=S1[pos2].i;
                        temp2[1]=S1[pos2].j;
                        temp2[2]=k;
                        temp2[3]=SR2_M[l].i;
                        Sol.push_back(temp2);
                        t1=GetTime()-t1;
                        for(j=pos2+1;j<S1.size()&&S1[j].value==T_;j++){
                                temp2=new int [4];
                                temp2[0]=S1[j].i;
                                temp2[1]=S1[j].j;
```

```
                                        temp2[2]=k;
                                        temp2[3]=SR2_M[l].i;
                                        Sol.push_back(temp2);
                                }
                                for(j=pos2-1;j>=0&&S1[j].value==T_;j--){
                                        temp2=new int [4];
                                        temp2[0]=S1[j].i;
                                        temp2[1]=S1[j].j;
                                        temp2[2]=k;
                                        temp2[3]=SR2_M[l].i;
                                        Sol.push_back(temp2);
                                }
                        }
                }
                for(l=pos-1;l>=0&&SR2_M[l].value==tau_t;l--){
                        T_=T-tau_L2-SR2[SR2_M[l].i];
                        pos2=half(S1,0,S1.size()-1,T_);
                        if(pos2>=0){
                                temp2=new int [4];
                                temp2[0]=S1[pos2].i;
                                temp2[1]=S1[pos2].j;
                                temp2[2]=k;
                                temp2[3]=SR2_M[l].i;
                                Sol.push_back(temp2);
                                for(j=pos2+1;j<S1.size()&&S1[j].value==T_;j++){
                                        temp2=new int [4];
                                        temp2[0]=S1[j].i;
                                        temp2[1]=S1[j].j;
                                        temp2[2]=k;
                                        temp2[3]=SR2_M[l].i;
                                        Sol.push_back(temp2);
                                }
                                for(j=pos2-1;j>=0&&S1[j].value==T_;j--){
                                        temp2=new int [4];
                                        temp2[0]=S1[j].i;
                                        temp2[1]=S1[j].j;
                                        temp2[2]=k;
                                        temp2[3]=SR2_M[l].i;
                                        Sol.push_back(temp2);
                                }
                        }
                }
        }
}
```

```
}
```

Simple HGJ algorithm

```
RandomBnd(R1,M);
RandomBnd(R2,M);
RandomBnd(R3,M);
ZZ R4=(target-R1-R2-R3)%M;
for(i=0;i<Hammingweight/4;i++){
        subproblem(R1,a,n,Hammingweight/4,1);
        R1=R1+M;
}
for(i=0;i<Hammingweight/4;i++){
        subproblem(R2,a,n,Hammingweight/4,2);
        R2=R2+M;
}
for(i=0;i<Hammingweight/4;i++){
        subproblem(R3,a,n,Hammingweight/4,3);
        R3=R3+M;
}
for(i=0;i<Hammingweight/4;i++){
        subproblem(R4,a,n,Hammingweight/4,4);
        R4=R4+M;
}
Random_M_odular_4_way_M_erge(Sol1,Sol2,Sol3,Sol4,n,target);
```