Proceedings of the 3rd International Conference on Computing Innovation and Applied Physics
DOI: 10.54254/2753-8818/30/20241106

The recovery time complexity of a coin weighing algorithm

Hanxuan Wang

Shanghai Shibei Senior High School, shanghai, China

wanghanxuan007@163.com

Abstract. Given *n* identical-looking coins each with possible weight in {0, 1}, and a scale that can measure the weight of any arbitrary set of coins, the coin weighing problem studies how to find out the weight of every coin with as few weighing as possible. The algorithm in Lindström takes *O*(*n*/log*n*) non-adaptive weighings to determine the coins, which gives an *O*(log*n*) factor improvement compared with the naïve algorithm that measures each coin on its own. However, it is unclear that with the *O*(*n*/log*n*) queries, how long it takes to retrieve *x*. This paper is about establishing and further optimizing the naïve recovery time complexity of Lindström 's algorithm. The recovery time complexity here is defined as the time complexity to recover *x* given *Dx* under the RAM model, where $D \in \{0,1\}^{m\times n}$,, each row being a weighing query, is the Lindström query matrix. The brute force recovery algorithm has running time $O(m^2n)$, whereas our algorithm only takes O(*mn*). Finally, we run experiments to verify our results with the actual running time of the algorithm on various size of inputs.

Keywords: Group testing, Coin Weighing Problem, Quantitative Group Testing

1. Introduction

There has been a long line of work on the “coin weighing problem”. A stack of coins, each has weight either 0 or 1. There is a scale which tells you the total weight of an arbitrary subset of coins. A natural question: how to get the weight of each coin with as few weighing as possible? n weighing are definitely enough, but can we do better? The answer is positive. One can play with the problem a bit and find that 3 weighing is enough to figure out the weights of 4 coins. Due to the simplicity of the problem and its connection to group testing, this problem was almost resolved by mathematicians in 1960s. An important piece of work is by Lindström, in which they found a clever way to construct the queries and recovery the weights [1].

The query complexity of an algorithm with query access is defined as the number of queries the algorithm makes. Cantor and Mills designed the first $O(n/\log n)$ query complexity algorithm for the coin weighing problem [2]. The algorithm is recursive and elegant. Lindström argued the coin weighing problem has query complexity $\Theta(n/\log n)$ [1]. The lower bound uses a combinatorial argument involving determinant of a family of matrices, although one can see the asymptotic lower bound $\Omega(n/\log n)$ nearly immediately by using some ideas from information theory: each query givens at most $\log n$ bits of information, yet there are n bits of information hiding in x.

The study of coin weighing with arbitrary x was almost closed at this point. Since then, more refined groups of coins are conjured and studied. A notable one is the group of coins with total weights d. The coin family now has size $\binom{n}{d}$, while on the other hand each weighing gives at most $\log d$ bits of information. The leads to a $\Omega \frac{d\log(n/d)}{\log d}$ lower bound. The same upper bound is achieved by Bshouty with an adaptive, deterministic algorithm [3]. There are ongoing works on getting better non-adaptive algorithms, with the current best non-adaptive deterministic algorithm of query complexity $O(d\log n)$, achieved by the BCH code, discovered separately by Raj Chandra Bose, Dwijendra Kumar RayChaudhuri and Alexis Hocquenghem [4]. For randomized algorithms, there is an $O\left(d\log\frac{n}{d}\right)$ queries non-adaptive algorithm by [5] [6].

Although the complexity of a query algorithm is defined as the number of queries it makes, we think it is crucial to optimize the remaining operations to make the recovery process faster as well. In fact, several work related to coin weighing provide non-constructive algorithms or randomized algorithm that is unclear if the recovery can be done in polynomial time. Creating better recovery algorithms make the query algorithms themselves to be more practicable. Formally, we define the recovery time complexity to be the following.

Definition 1. (Recovery time complexity) Let $\mathrm{TC}(\mathcal{A}(x))$ be the time complexity of running algorithm $\mathcal{A}$ on input x. Given a query algorithm $\mathcal{D}$, the corresponding recovery algorithm $\mathcal{R}_{\mathcal{D}}$, the recovery time complexity of $\mathcal{R}_{\mathcal{D}}$ is defined as $\max_x \mathrm{TC}\,(\mathcal{R}_{\mathcal{D}}(\mathcal{D}(x)))$.

In English, the recovery time complexity of the recovery algorithm $\mathcal{R}_{\mathcal{D}}$, for query algorithm $\mathcal{D}$, is the worst case time complexity of reconstructing $\mathcal{D}(x)$.

1.1. Our Results

Theorem 2. Let $\mathcal{D}$ be the Lindström algorithm that given x outputs Dx where D is the Lindström matrix in $\{0,1\}^{m\times n}$ and x is an arbitrary vector in $\{0,1\}^n$. There is an algorithm $\mathcal{R}_{\mathcal{D}}$ with recovery time complexity $O(mn)$.

Remark: Note the trivial recovery time complexity of Lindström algorithm is $O(m^2 n)$. This is because we iterate i through m to 1 (in a decreasing order), in each iteration we try to learn $\alpha(i)$ new entries of x. In each iteration, the most expensive operation is computing $\lambda_i D$. λ_i is a vector of carefully chosen coefficients with dimension $1\times m$, and D is a matrix with dimension $m\times n$. computing Dx takes $O(mn)$ time. Hence the overall complexity is $O(m^2 n)$.

To see why computing $\lambda_i D$ is needed, see a brief discussion of how Lindström algorithm works in Section 2, and remarks in Section 4.

2. Preliminaries

Let's abstract the problem into math language. Given a hidden vector $v \in \{0,1\}^n$, we want to find a set of queries $q_1, q_2, \cdots, q_m \in \{0,1\}^n$ such that $q_1^\top \cdot v, q_2^\top \cdot v, \cdots, q_m^\top \cdot v$ uniquely determines v while minimizing m. Note if $q_1, q_2, \cdots, q_m$ are independent, then we say the algorithm (which is the set of queries) non-adaptive. If $q_1, q_2, \cdots, q_m$ are generated deterministically (without using random bits), then the algorithm is deterministic. In this work, we focus on Lindström's work, which is both non-adaptive and deterministic [1]. For non-adaptive algorithm, since the queries are independent, we can treat each query as a row, and stack them together to form the query matrix

$$D = \begin{bmatrix} -q_1^\top - \\ -q_2^\top - \\ \vdots \\ -q_m^\top - \end{bmatrix} \tag{1}$$

Before we dive deeper into the construction of Lindström matrix, we introduce some **notations and definitions**.

For any ordered object s with length, we use $s[i]$ to represent the value of s at index i (0-based).

Definition 3. (Set definition of natural numbers) Let $a \in \mathbb{N}$, binary(a) be its binary representation.

Define $s(a)$ be the set that contains 2^i if and only if binary $(a)[i] = 1$. For instance, $s(4) = \{4\}, s(7) = \{1,2,4\}$. When the context is clear, we might omit the function $s()$ for brevity. With this definition, we can use $\subseteq, \nsubseteq$ between natural numbers.

Define $\alpha(a)$ to be $|s(a)|$.

It's easy to see a bijection between $\mathbb{N}$ and $s(a)$.

Definition 4. (Distinguishing vector) Let $x, y \in \{0,1\}^n$ and $v \in \{0,1\}^n \cdot v$ is a distinguishing matrix if $v \cdot x \neq v \cdot y$ for $x \neq y$.

One common choice of v is $d_n = [2^0, 2^1, \cdots, 2^{n-1}]$.

Definition 5. (Distinguishing matrix) Let $x, y \in \{0,1\}^n$ and $D \in \{0,1\}^{m\times n}. D$ is a distinguishing matrix if $Dx \neq Dy$ for $x \neq y$.

Proof. First, both D_x and D_y are two matrices and $\mathrm{x}, \mathrm{y} \in \{0,1\}^{A(m)}$. We purpose $D_x = D_y$ but $x \neq y$. By selecting the last element from two vectors separately, we can obtain $x_m \neq y_m$. Because $D_x = D_y, \lambda_x \cdot D_x = \lambda_y \cdot D_y$. Then $d_x = d_y$ and $x_m = y_m$. And this conflicts with what we purposed $x_m \neq y_m$. So if $x \neq y, D_x \neq D_y$

Definition 6. (Lindström matrix) Lindström matrix is a $m \times \sum_{r=1}^{m} \alpha(a)$ size matrix. We describe each $m \times \alpha(r)$ submatrix D_r.

$$D_r[i][j] = \begin{cases} \text{in a way such that} \sum_j D_r[i] = \left[2^{0,2^{1,2^2}} \ldots, 2^{\alpha(r)-1}\right] & i \subseteq r, \alpha(i) \bmod 2 = 1 \\ 0 & i \subseteq r, \alpha(i) \bmod 2 = 0 \\ D_r[r \cap i][j] & i \nsubseteq r \end{cases} \tag{2}$$

Some explanations:

(a) When $i \subseteq r$, If $\alpha(r)$ is an odd number, the summation of the rows with satisfying index r should be $d_{\alpha(r)} = \left[2^0, 2^1, 2^2 \ldots, 2^{\alpha(i)-1}\right]$. Conversely, if $\alpha(r)$ is even, the pertinent rows will be replete with 0 s.

For example, if $m = 6, i = 6, D_6$ is a 6×2 matrix because $\alpha(6) = 2. \{2\}, \{4\}, \{2,4\} \subseteq s(6)$, $\alpha(2)$ and $\alpha(4)$ are odd, but $\alpha(6)$ is even. So the row 6 is filled with 0 s, $D_6[6] = [0,0]$. For rows 2 and $4, D_6[2] + D_6[4] = [1,2]$, so we can let $D_6[2] = [1,1], D_6[4] = [0,1]$. (The precise answer of D_2 and D_4 is flexible. $D_6[2] = [0,1], D_6[4] = [1,1]$ is also fine. We only need to make sure the sum of these rows is [1,2].

(b) When $i \nsubseteq r, D_r[i][j] = D_r[r \cap i][j]$. Following with the previous example. 1,3,5 are $\nsubseteq$ 6. $5 \cap 6 = 4$, So $D_6[5][1] = D_6[5 \cap 6][1] = D_6[4][1] = 0, D_6[5][2] = D_6[5 \cap 6][2] = D_6[4][2] = 1$. If $i \cap r = p$, then the i th rows is exactly the p th row. Thus, we can just use corresponding rows instead of computing numbers one by one.

(c) Finally, 0th row is the zero row vector by default. For example, $1 \cap 6 = 0$, so $D_6[1] = [0,0]$. **Since the all zero query doesn't give any information, we don't include the 0th row in our query matrix.** As a result,

$$D_6 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \tag{3}$$

And for completeness,

$$D = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{4}$$

Definition 7. λ_r is a coefficient vector designed with the purpose that $\lambda_r^\top D_r$ is a distinguishing vector.

$$\lambda_i[j] = \begin{cases} (-1)^{\alpha(j)+1} & j \subseteq i \\ 0 & j \not\subseteq i \end{cases} \tag{5}$$

Lemma 8. $\lambda_r^\top D_r = d_{\alpha(r)}$.

Eg:

$$[\lambda_6^\top \cdot D_6 = [0 \quad 1 \quad 0 \quad 1 \quad 0 \quad -1] \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} = [1 \quad 2]] \tag{6}$$

Proof. Based on the definition of λ_i, we only need to focus on those numbers that are a subset of $s(i)$. Since we raise -1 to the power of $\alpha(j) + 1$, rows with odd α will be added. When $\alpha(j)$ is an even, its corresponding number should be -1 in λ. However, because of the definition, this rows must be filled with 0 s in the D_r, which will not affect our calculation. As a result, the sum of those rows is $d_{\alpha(r)}$.

Lemma 9. If $r < i, \lambda_i^\top D_r = [0, \cdots, 0]$.

Eg:

$$[\lambda_5^\top \cdot \mathrm{D}_4 = [1 \quad 0 \quad 0 \quad 1 \quad -1 \quad 0] \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} = 0] \tag{7}$$

Proof. Since $i \geq r$, there must be an element e that only exists in $s(i)$ and not in $s(r)$. This implies that we can find a bijection between the subsets of $s(i)$, where the bijection maps every set x to $y = x \cup \{e\}$.

$\alpha(y) = \alpha(x) + 1$. Moreover, $D_r[x] = D_r[x \cap r]$ and $D_r[y] = D_r[r \cap y] = D_r[r \cap (x \cup \{e\})] = D_r[r \cap x]$. We conclude that $(-1)^{\alpha(x)} D_r[x] + (-1)^{\alpha(y)} D_r[r \cap y] = D_r[x] - D_r[x] = 0$.

$$\begin{aligned} \lambda_i^\top D_r &= \sum_{j \subseteq i} (-1)^{\alpha(j)+1} D_r[j] \\ &= \sum_{j \subseteq i, e \in j} (-1)^{\alpha(j)+1} D_r[j] + \sum_{j \subseteq i, e \notin j} (-1)^{\alpha(j)+1} D_r[j] \\ &= \sum_{j \subseteq i, e \in j} (-1)^{\alpha(j)+1} D_r[j \cap i] + \sum_{j \subseteq i, e \notin j} (-1)^{\alpha(j)+1} D_r[j \cap i] \\ &= 0 \text{ (bijection argument)} \end{aligned} \tag{8}$$

Lemma 10. Lindström matrix is a distinguishing matrix.

Proof. Following easily from the two lemmas above.

3. Recover Coin Weights by Brute Force

Let x be a length n vector representing the hidden coin weights, and let D represent the $m \times n$ Lindström matrix. We first describe a brute force way to compute x. The high level idea:

1. In every loop r indexed from m to 1 we try to recover $\alpha(r)$ elements in x
2. In loop r, we compute λ_i, multiply it with Dx and D, then use property of $\lambda_i D$ and the equation $\lambda_i D \cdot x = \lambda_i Dx$ to recover $x\left[\sum_{j=1}^{i-1} \alpha(j) : \sum_{j=1}^{i} \alpha(j)\right]$.

Note the bottleneck is computing $\lambda_i D$, with time complexity $O(mn)$.

3.1. Numbering

Algorithm 1 getLambdaD

```
Input: D: an $m \times n$ Lindstr¨om matrix; $\lambda$: a length $m$ vector.
Output: the product of $\lambda$ and D.
1: $n \leftarrow D.shape[0]$
2: $m \leftarrow D.shape[1]$
3: $result \leftarrow np.zeros(n)$
4: for $i = 0$ to $m$ do
5:     for $j = 0$ to $n$ do
6:         $result[j] \leftarrow result[j] + \lambda[i] * D[i][j]$
7:     end for
8: end for
9: return $result$
```

Algorithm 2 solveX

```
Input: D: an $m \times n$ Lindstr¨om matrix; Dx: the dot product of D and x.
Output: $x \in \{0, 1\}^{1\times m}$ such that $D \cdot x = Dx$.
1: $m \leftarrow len(D)$, $n \leftarrow len(D[0])$
2: $end \leftarrow n$
3: $sol \leftarrow [0] \times n$
4: for $r = m$ to 0 do
5:     $\lambda \leftarrow$ getLambdar(m, r)        ▷ See Defnition 7.
6:     $num \leftarrow$ getLambdaDx($\lambda$, Dx)        ▷ getLambdaDx($\lambda$, Dx) returns the dot product of $\lambda$ and Dx.
7:     $v \leftarrow$ getLambdaD($\lambda$, D)        ▷getLambdaD($\lambda$, D) returns dot product of $\lambda$ and D.
8:     $start \leftarrow end - \alpha(r)$
9:     if end < $n$ then
10:        for $i = end$ to $n$ do
11:            $num \leftarrow num - v[i] * sol[i]$
12:            $x \leftarrow$ solvePartialx(v, num, start, end) ▷ solvePartialx returns the weight of each coin in the form of vector.
13:            $sol \leftarrow sol + x$
14:            $end \leftarrow start$
15:        end for
16:    end if
17: end for
18: return $sol$
```

3.2. Analysis of the time complexity in the original algorithm

One can easily check the correctness of Algorithm 2.

Complexity: We first observe that the bottleneck in the for loop in Line 4 is getLambdaD which takes $O(mn)$ time. The details of getLambdaD, which is essentially a matrix product between $\lambda \in \{0,1\}^{1\times m}$ and $D \in \{0,1\}^{m\times n}$, takes $O(mn)$ time.

Line 4 is a for loop that iterates m times, implying that getLambdaD (λ, D) must run m times, resulting in a time complexity of $O(m^2n)$. For larger values of m, the program's running time will be considerably slow. However, there is potential for improvement in the method of calculating x owing to the properties of this method to get x. Therefore, we aim to make some alterations to decrease the algorithm's time complexity.

4. Improved Algorithm to Recover

The use of sections to divide the text of the paper is optional and left as a decision for the author. Where the author wishes to divide the paper into sections the formatting shown in table 2 should be used.

4.1. Algorithm

Algorithm 3 getX

Input: Both r and s are a list of numbers that get from definition 3; *res* is a list of coefficient improved from initial numbers satisfy $s(num) \subseteq s(r)$ *and* $\alpha(num)$ *equals an odd* that build on the basic of definition 13

Output: a list partly same as list *res*, but filling with 0s in the position of missing number compared with s.

```
1: for $i = 0$ to $len(s)$ do
2:     if $i < len(r)$ then
3:     if $s[i] = r[i]$ then
4:     continue
5:     else
6:     $r.insert(i, 0)$             ▷ Keep the lists $r$ and $s$ the same length
7:     $res.insert(i, 0)$
8:     end if
9:     else
10:    $r.append(0)$
11:    $res.append(0)$▷ Fill the vacant position with 0$s$ to facilitate subsequent calculations
12:    end if
13: end for
14: return $res$
```

Algorithm 4 getLambdaDr

Input: l is a list of numbers in the $s(num)$; x is the modified coefficient for each row. (Rows don't need to add, their coefficient will be zeroes.)

Output: *result* is a part that $i \leq r$ of $\lambda * D$

```
1: $total \leftarrow []$▷ $total$ keeps track of the number formation of different rows
2: $result = np.zeros(len(l))$
3: for $i$ in $l$ do
4:     index = $bisect.bisect\ right(l, i) - 1$▷ Through binary search to certificate the amount of 0$s$
5:     $vector = [0] * index + [1] * (len(l) - index)$ ▷ See Definition 13, Forming rows in new definition
6:     $total.append(vector)$
7: end for
8: for $j = 0$ to $len(l)$ do
9:     if $x[j] = 0$ then
10:    continue
11:    else
12:    $result = result + int(x[j]) * np.array(total[j])$▷ $result$, a vector of length $\alpha(s)$ that equals to $\lambda_i * D_r$
13:    end if
14: end for
15: return $result$
```

Algorithm 5 getModifiedLambdaD

```
Input: m:is the number of rows in D; r: selects from 1 to m in order;
Output: a vector of length n that equals to $\lambda \cdot D$.
1: result ← []▷ result keeps track of the formation of $\lambda D$
2: for i = 1 to m + 1 do
3:      if i < r then
4:      D_i ← [0] * α(i)▷ See definition 3 α(i) is the length of s(i)
5:      result = np.concatenate((result, D_i), axis = 0)▷ Put numbers in D_i into result in order
6:      end if
7:      if i ≥ r then
8:      if set(IntegerToList(r)) ⊆ set(IntegerToList(i)) then▷See definition 3, convert r to a list similarly, sorted
9:      x ← getX()▷ x returns modified list to directly calculate 10:D_i = lambda_iD_r(l, x)▷ See definition 3,l can get from IntegerToList() 11:result = np.concatenate((result, D_i), axis = 0)
12:     else
13:     D_i = [0] * α(i)
14:     result = np.concatenate((result, D_i), axis = 0)
15:     end if
16:     end if
17: end for
18: return result
```

4.2. Analysis about modification in the algorithm

Lemma 11. When $r > i$ and $i \not\subset r, \lambda_i \cdot D_r = [0] * \alpha(r)$.

Proof. Same as the proof of lemma 9.

Lemma 12. When $i \subset r$, only need to calculate those $\lambda_i[j] = 1$ (j means the position of number in λ)

Proof. In λ_i, only $j \subseteq i$ will have a value of either 1 or -1 . According to definition 7 , we can determine that when $\alpha(j) = 2k(k \in \mathbb{N})$, in $\lambda_i[j]$ will have a value of -1 . However, in the matrix D_r, these rows will equal to $[0] * \alpha(r)$ which means it will not affect our calculation.

This can directly decrease half of the numbers that need to be calculated. Because each set will get 2^t subsets, t is related to the numbers in the set. Then in those sets which include even numbers in the set occupy half of the all subsets. According to lemma 12, we only need to calculate the another half of the subsets that include odd numbers in the set.

Definition 13. We define the free rows in D_r as below. (j is the position of i in $s(r)$)

$$D_r[i] = \begin{cases} [0] * s(i)[j] + [1] * (\alpha(r) - s(i)[j]) & i \subseteq r \text{ and } \alpha(i) = 1 \\ D_r[\lfloor \log_2 i \rfloor] & i \subseteq r \text{ and } \alpha(i) = 2k + 1\ (k > 1) \end{cases} \tag{9}$$

For example, if $m = 15, D_1 = [1,1,1,1]$ $D_2 = [0,1,1,1]$ $D_4 = [0,0,1,1]$ $D_7 = [0,0,1,1]$ $D_8 = [0,0,0,1]$ $D_{11} = [0,0,0,1]$ $D_{13} = [0,0,0,1]$ $D_{14} = [0,0,0,1]$

Proof. With the property of 2^k, we know that the sum of front numbers must smaller than the next number(These numbers specifically refer to $2^k(k \in \mathbb{N})$). For example, $1 + 2 + 4 + 8 < 16$. Because

$$2^0 + 2^1 + \cdots 2^k = \frac{1-2^{k+1}}{1-2} = 2^{k+1} - 1 < 2^{k+1} \tag{10}$$

The way the algorithm works: To understand how the algorithm works, we first need to turn numbers into sets as definition 3. Once this is done, we can list the subset of $s(i)$ that satisfy $\alpha($ subsets $) = 2k + 1(k \in \mathbb{N})$, which are the ones we will be working with. Next, we convert these numbers into coefficients that can be directly multiplied with their corresponding rows in the D_r. We create a new list called coefficient and store the numbers in satisfiedNumber that were originally $2^k(k \in \mathbb{N})$ as $1s$. Continuing with definition 13 , if any number in the list meets the condition $2^k <$ num $< 2^{k+1}$, we add 1 to its corresponding number in coefficient. We then compare $s(i)$ with the $s(r)$ and append a 0 in the corresponding position of any number that exists in r but not in i. This completes the process of forming the modified coefficient of $\lambda * D_r$ Moving on, we need to describe part of the D_r. We generate lines using the numbers in the $s(r)$ list and arrange them according to definition 13. We then multiply the coefficients with their corresponding rows. If a number in the coefficient list is 0 , it is skipped. Finally, the various steps are consolidated to obtain the answer of $\lambda_i D_r$.

Lemma 14. Computing $\lambda_i D_r$ with $i < r, i \subseteq r$ takes $O(\alpha(r) \cdot \log\ m)$ time.

For example, $i = 15, r = 31, s(i) = \{1,2,4,8\}, s(r)[r$ is a number selected from 1 to $m] = \{1,2,4,8,16\}$,

satisfiedNumber $= [1,2,4,7,8,11,13,14]$, coefficient $= [1,1,2,4]$. Due to $s(i)$ without 16 , so modifiedCoefficient $= [1,1,2,4,0]$. The final answer is $1 * D_1 + 1 * D_2 + 2 * D_4 + 4 * D_8 = [1,2,4,8,8]$.

Proof. First, r should be divided into sets such as $r = \{2^{x_1}, 2^{x_2} \ldots, 2^{\alpha(r)-1}\}$. According to lemma 11 and lemma 12, The computation of λD_r can be simplified by focusing solely on the subset of r whose length is an odd number. By summing these rows, the time complexity can be reduced to $O(m * \alpha(r))$, where m is the largest number of subsets whose length equals an odd number. For example, when $m = r = 2^{k-1}, \alpha(r) = k - 1$ and the number of subsets whose length is odd number should be $\frac{2^{k-1}}{2}$. Thus, its time complexity equals $O(m)$. The value of $\alpha(r)$ represents the length of each row that needs to be added. However, there is a more efficient way. We count the numbers satisfied the condition mentioned above and lie between two numbers like 2^k. Then adding it to the coefficient of its related vector. Therefore, the actual rows we need to calculate are those $\alpha(\text{ num }) = 1$ and the final time complexity should be $\log\ m * \alpha(r)$. (num refers to row numbers in matrix D)

Remark: Why do we have to calculate λD first? Why not just calculate λDx which saves a lot of time? Indeed, calculate $\lambda \times Dx$ only takes $O(m)$ time. However, $\lambda \times Dx$ will only get us a number, but our goal is to know the weight of each coin. Thus, we still have to know the coefficient of each variable coin in the equation. Because of the way to know the weight of coin depends on the special property that the specific coefficient brings, it is not clear how to avoid calculating the coefficient of those number to get their weight.

4.3. Analysis of the time complexity in the modified algorithm

We want to analyze the total running time of computing $\lambda_i D$, where i ranges from 1 to m. To compute $\lambda_i D$, we break it down to computing $\lambda_i D_r$ where r ranges from 1 to m.

Now according to Algorithm 5, we break it down into 4 cases:

1. $(i > r) \lambda_i D_r = \mathbf{0}$. The zero vector $\mathbf{0}$ has length $\alpha(r)$, so the number of unit operations is $\alpha(r)$.
2. $(i = r) \lambda_i D_r = d_{\alpha(r)}$. Similarly, the number of unit operations is $\alpha(r)$.
3. $(i < r, i \nsubseteq r) \lambda_i D_r = \mathbf{0}$. The number of unit operations is $\alpha(r)$.
4. $(i < r, i \subseteq r)$ As discussed above, the number of unit operations is $\alpha(r) \cdot \log\ m$.

For the first two cases, the total number of operations is $\sum_{r=1}^{m} \alpha(r) \cdot (m + 1 - r)$.

For the last two cases, the number of operations is $\sum_{r=1}^{m} (r - 2^{\alpha(r)}) \alpha(r)$ and $\sum_{r=1}^{m} 2^{\alpha(r)} \alpha(r) \log\ m$, respectively.

Adding those together, the overall complexity is

$$\sum_{r=1}^{m} \alpha(r) \cdot (m + 1 - r) + (r - 2^{\alpha(r)}) \alpha(r) + 2^{\alpha(r)} \alpha(r) \log m$$

$$= \sum_{r=1}^{m} \alpha(r) \cdot \left(m + 1 + 2^{\alpha(r)} (\log m - 1)\right) \tag{11}$$

To simplify $\sum_{r=1}^{m} \alpha(r) 2^{\alpha(r)} \log\ m$, we assume $m = 2^k$ for some $k \in \mathbb{N}$.

$$\begin{aligned}\sum_{r=1}^{m}\alpha(r)2^{\alpha(r)}\log m &= \sum_{r=1}^{2^k}\alpha(r)2^{\alpha(r)}k \\ &= \sum_{i=1}^{k}\binom{k}{i} i2^i k \\ &= 2k^2 3^{k-1} \\ &= \frac{2}{3(\log m)^2 m^{\log 3}}\end{aligned} \tag{12}$$

Hence the sum $\sum_{r=1}^{m}\alpha(r)\cdot\left(m+1+2^{\alpha(r)}(\log\, m-1)\right)$ is dominated by $\sum_{r=1}^{m}\alpha(r)\cdot m = O(mn) = O(m^2\log\, m)$.

5. Experiments

We run the brute force method and our method on the same set of Q, Qx and plot the time required to recover x.

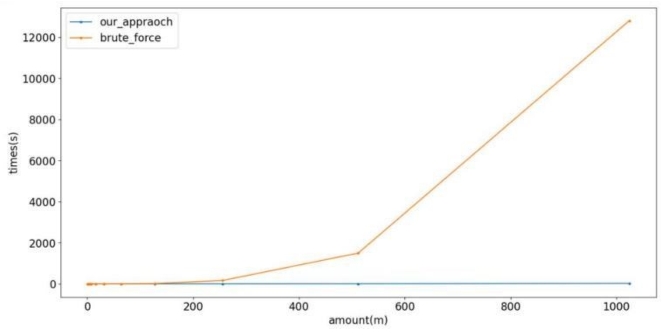

Figure 1. n v.s. time with the brute force and our approach

Note the brute force approach uses for loop instead of built-in @ operator when computing vector matrix multiplication. We did this on purpose since @ operation is optimized and does not reflect the underlying time complexity of matrix multiplication. Of course, we avoid using @ operator in our optimized algorithm for fairness.

We then plot the running time of our method against the recovery time complexity.

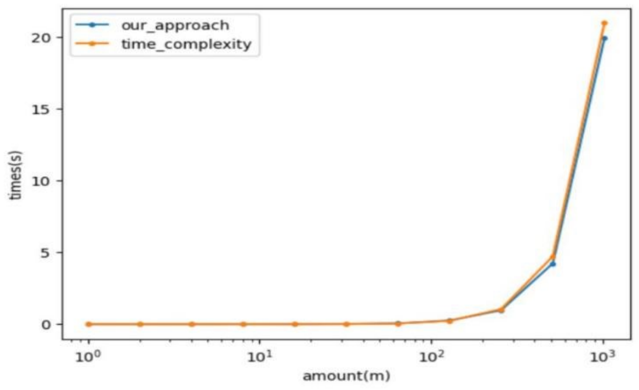

Figure 2. n v.s. time with the time complexity and our approach

6. Conclusion and Future Works

We proposed an efficient recovery algorithm that utilizes the math property of Lindström's construction. The algorithm is much faster than the brute force approach, and its running time with our implementation is on par with what we conclude in theory.

For open problems, we note that we haven't explored any lower bounds yet - it would be an interesting question to see if there is any non-trivial lower bound and how large the gap is with our current upper bound. One concrete question is, is $\Omega(mn)$ the lower bound? Do we have to compute most of $\lambda_r D$ for every r ? Can we possibly do better?

References

[1] Lindström, B. (1971). On Möbius functions and a problem in combinatorial number theory. *Canadian Mathematical Bulletin*, 14(4):513-516.
[2] Cantor, D. G. and Mills, W. H. (1964). Determination of a subset from certain combinatorial properties.
[3] Bshouty, N. H. (2009). Optimal algorithms for the coin weighing problem with a spring scale. In COLT, volume 2009, page 82.
[4] Bose, R. C. and Ray-Chaudhuri, D. K. (1960). On A class of error correcting binary group codes. Inf. Control., 3(1):68-79.
[5] Gebhard, O., Hahn-Klimroth, M., Kaaser, D., and Loick, P. (2019). Quantitative group testing in the sublinear regime. CoRR, abs/1905.01458.
[6] Karimi, E., Kazemi, F., Heidarzadeh, A., Narayanan, K. R., and Sprintson, A. (2019).