A fast and simple algorithm for the Money Changing Problem

Sebastian Böcker[1] and Zsuzsanna Lipták[2]

[1] Lehrstuhl für Bioinformatik, Friedrich-Schiller-Universität Jena
Ernst-Abbe-Platz 2, 07743 Jena, Germany, boecker@minet.uni-jena.de
[2] AG Genominformatik, Technische Fakultät, Universität Bielefeld
PF 100 131, 33501 Bielefeld, Germany, zsuzsa@CeBiTec.uni-bielefeld.de

This is a reprint of: S. Böcker and Zs. Lipták. A fast and simple algorithm for the Money Changing Problem. Algorithmica, 48(4):413–432, 2007.
https://doi.org/10.1007/s00453-007-0162-8

Abstract. The Money Changing Problem (MCP) can be stated as follows: Given k positive integers $a_1 < \ldots < a_k$ and a query integer M, is there a linear combination $\sum_i c_i a_i = M$ with non-negative integers c_i, a *decomposition* of M? If so, produce one or all such decompositions.
The largest integer without such a decomposition is called the *Frobenius number* $g(a_1, \ldots, a_k)$. A data structure called *residue table* of a_1 words can be used to compute the Frobenius number in time $O(a_1)$. We present an intriguingly simple algorithm for computing the residue table which runs in time $O(ka_1)$, with no additional memory requirements, outperforming the best previously known algorithm. Simulations show that it performs well even on 'hard' instances from the literature. In addition, we can employ the residue table to answer MCP decision instances in constant time, and a slight modification of size $O(a_1)$ to compute one decomposition for a query M. Note that since both, computing the Frobenius number and MCP (decision) are NP-hard, one cannot expect to find an algorithm that is polynomial in the size of the input, i.e., in $k, \log a_k$, and $\log M$.
We then give an algorithm which, using a modification of the residue table, also constructible in $O(ka_1)$ time, computes all decompositions of a query integer M. Its worst-case running time is $O(ka_1)$ for each decomposition, thus the total runtime depends only on the output size and is independent of the size of query M itself.
We apply our latter algorithm to interpreting mass spectrometry (MS) peaks: Due to its high speed and accuracy, MS is now the method of choice in protein identification. Interpreting individual peaks is one of the recurring subproblems in analyzing MS data; the task is to identify sample molecules whose mass the peak possibly represents. This can be stated as an MCP instance, with the masses of the individual amino acids as the k integers $a_1, \ldots, a_k$. Our simulations show that our algorithm is fast on real data and is well suited for generating candidates for peak interpretation.

1 Introduction

Let $a_1 < \ldots < a_k$ be positive integers. Given a query integer M, is M decomposable over $\{a_1, \ldots, a_k\}$, i.e., can M be written as a non-negative linear combination of the form $\sum_i c_i a_i = M$, with $c_i \geq 0$ integers for $i = 1, \ldots, k$? If so, can we find such a decomposition $c = (c_1, \ldots, c_k)$ of M? Can we find all vectors c which are decompositions of M?

The decision form of this problem, namely, does a decomposition for a query integer M exist, is called Money Changing Problem (MCP): One thinks of the $a_1, \ldots, a_k$ as coin denominations, and M is the change which has to be made when an infinite supply of coins is available. The problem is also known as Equality Constrained Integer Knapsack Problem and is NP-complete [17], but can be solved in pseudo-polynomial time (time $O(kM)$) with a dynamic programming algorithm [18]. In principle, one can solve MCP decision problems using generating functions, but the computational cost for coefficient expansion and evaluation is too high in applications [11, Chapter 7.3].

If the masses are relatively prime, then there exists an integer $g(a_1, \ldots, a_k)$, called the *Frobenius number*, which cannot be decomposed but all larger integers can. Computing the Frobenius number is also NP-hard [21]. However, a data structure we refer to as *residue table* of a_1 words can be used to compute the Frobenius number in time $O(a_1)$, as first suggested by Brauer and Shockley [6].

Here, we present an intriguingly simple algorithm, Round Robin, to compute the residue table—and hence, to find $g(a_1, \ldots, a_k)$—in time $O(k\,a_1)$ with constant extra memory. Our algorithm outperforms the best previously known algorithm, due to Nijenhuis [19], both in time and in the additional space requirements, as well as outperforming it in practice, as we show in Section 8. Our algorithm has also been implemented and tested by Beihoffer et al. [3], who made a detailed comparison on many different types of instances and found that Round Robin often compared favourably to other algorithms. In addition, our algorithm remains the only algorithm to compute the residue table with worst case complexity $o(ka_1 \log a_1)$.

Once the residue table has been constructed, we can solve subsequent MCP decision instances in constant time, *for any* M. Moreover, a slight modification of the residue table, with an additional data structure of $2a_1$ words, enables us to produce one decomposition for MCP queries in time $O(k)$. In constrast, the DP tableau of the classical dynamic programming algorithm can be used to produce a decomposition of the query in worst-case time $O(M/a_1)$, thus in time linear in M. Note that the NP-hardness results imply that we cannot hope to find algorithms that are polynomial in $k, \log a_k$, and $\log M$.

We use a modification of the residue table, termed *extended residue table*, to produce all decompositions of a query. Size and construction time of the extended residue table are $O(ka_1)$. Our algorithm for producing all decompositions is based on the idea of backtracking in this data structure in a smart order; the worst-case runtime is $O(ka_1)$ per decomposition, thus the total runtime can be stated as $O(ka_1\gamma(M))$, where $\gamma(M)$ is the number of decompositions of M. In other words, the runtime is independent of the size of the query itself, and only depends on the size of the data structure and on the output size $\gamma(M)$.

The problem of decomposing a number over a given set of numbers occurs frequently when interpreting mass spectrometry data. Mass spectrometry (MS) is a technology which, in essence, allows to determine the molecular mass of input molecules. Because of its speed and accuracy, it has become the prime technology for protein identification, either with the use of a database, or *de novo* [2]. Put in a simplified way, the input of the experiment is a molecular mixture and the output a peak list: a list of masses and their intensities. Ideally, each peak should correspond to the mass of some sample molecule, and its intensity to the frequency of that molecule in the mixture. The situation is, in fact, more blurred, due to noise and other factors; however, the general idea remains. Thus, one problem is to determine for a peak which sample molecules it could possibly represent. For proteomic samples, peptides (strings over the amino acid alphabet) are sought whose mass equals the mass of the peak. Clearly, the order of the amino acids does not influence the total mass of the molecule[3], so we can abstract from the strings themselves and concentrate on the *multiplicity* of each amino acid. In other words, the mass of the peak has to be decomposed over the masses of the amino acids. Thus, after scaling the masses to integers, and using a suitable precision depending on the mass spectrometer, we have an instance of the problem introduced above.

As $\gamma(M)$ grows rapidly with large M (see Section 6), it becomes soon impractical to produce all decompositions of a peak as a possible interpretation. Instead, one will either produce these decompositions in an intermediate step as *candidates* for correct interpretation, and then filter them according to further criteria; or else prune the production algorithm during runtime according to some outside criteria. We have successfully applied both of these techniques to decomposition of metabolites, using isotopic information from the mass spectrometry read [5].

1.1 Related work

There has been considerable work on bounds for Frobenius numbers; see [22] for a survey. Here, we concentrate on exact computation.

In 1884, Sylvester asked for the Frobenius number of $k = 2$ coins a_1, a_2, and Curran Sharp showed that $g(a_1, a_2) = a_1a_2 - a_1 - a_2$ [23]. For three coins a_1, a_2, a_3, Greenberg [12] provides a fast algorithm with runtime $O(\log a_1)$, and Davison [7] independently discovered a simple algorithm with runtime $O(\log a_2)$. Kannan [14] establishes algorithms that for *any fixed* k, compute the Frobenius number in time polynomial in $\log a_k$. For variable k, the runtime of such algorithms has a double exponential dependency on k, and is not competitive for $k \geq 5$. Also, it does not appear that Kannan's algorithms have actually been implemented.

There has been a considerable amount of research on computing the exact Frobenius number if k is variable, see again [22] for a survey. Heap and Lynn [13] suggest an algorithm with runtime $O(a_k^3 \log g)$, and Wilf's "circle-of-light" algorithm [26] runs in $O(kg)$ time, where $g = O(a_1a_k)$ is the Frobenius number of the problem. Until recently, the fastest algorithm to compute $g(a_1, \ldots, a_k)$ was due to Nijenhuis [19]: It is based on Dijkstra's method for computing shortest paths [8] using a priority queue. This algorithm has runtime $O(k\, a_1 \log a_1)$ using binary heaps, and $O(a_1(k + \log a_1))$ using Fibonacci heaps as the priority queue. To find the Frobenius number, the algorithm computes the residue table, which in turn allows simple computation of $g(a_1, \ldots, a_k)$. Nijenhuis' algorithm requires $O(a_1)$ extra memory in addition to the residue table. Recently, Beihoffer et al. [3] developed algorithms that work faster in practice, but none obtains asymptotically better runtime bounds than Nijenhuis's algorithm, and all require extra memory linear in a_1.

[3] For ease of exposition, we ignore mass modifications of the total mass of sample molecules due to the ionization process, such as adding the mass of a single proton.

Different variants of MCP are known as Change Making Problem or Coin Change Problem. The original dynamic programming algorithm is due to Gilmore and Gomory [10], while Wright [27] shows how to find a decomposition with a minimal number of coins in time $O(kM)$. In most papers on the Coin Change Problem, it is assumed that one has a coin of denomination 1, thus the decision problem is trivial.

From the application side, the problem of finding molecules with a given mass was addressed frequently from the biochemical and mass spectrometry viewpoint, see for instance [4, 9, 20, 24].

1.2 Overview

The paper is organized as follows. In Section 2, we give problem definitions, an overview of our solutions' complexities for the four different problems considered (Frobenius, MCP decision, MCP one decomposition, MCP all decompositions), and sketch previous solutions and related problems. We introduce residue tables in Section 3. In Section 4, we present the Round Robin algorithm for constructing the residue table, and introduce a modification of the residue table for finding one decomposition of a query. We then present our algorithm for producing all decompositions of a query (Section 5). This is followed by a closer look at the connection between strings and compomers (equivalance classes of strings with identical character multiplicities) and by some graphs showing the number of decompositions of different types of biomolecules in mass ranges relevant to mass spectrometry in Section 6; and by details of optimization of the Round Robin algorithm in Section 7. Finally, we present our simulation results in Section 8.

2 Problem statements

Given k positive integers $a_1 < \ldots < a_k$, and a positive integer M (the query), a *decomposition* of M is a vector $c = (c_1, \ldots, c_k)$ with non-negative integer entries such that $c_1 a_1 + \ldots + c_k a_k = M$. We call M *decomposable* (over $\{a_1, \ldots, a_k\}$) if such a decomposition exists. We refer to $\{a_1, \ldots, a_k\}$ as an MCP instance, and to any vector c of length k with non-negative integer entries as a *compomer* (over $\{a_1, \ldots, a_k\}$). Compomers[4] can be viewed as equivalence classes of strings, where for $i = 1, \ldots, k$, a_i is a mass assigned to the ith character of the alphabet, and the ith entry c_i denotes the number of occurrences of this character in the string. See Section 6 for more on the connection between strings and compomers. We want to mention that even though we always assume that the masses are ordered, this is only for ease of exposition and not actually necessary for our algorithms. Moreover, our results can be easily generalized to the case where not all masses are distinct.

It is known that if the a_i's are relatively prime, i.e., if $\gcd(a_1, \ldots, a_k) = 1$, then there exists an integer $g = g(a_1, \ldots, a_k)$, the *Frobenius number*, such that g is not decomposable but every $n > g$ is. Conversely, if $\gcd(a_1, \ldots, a_k) = d > 1$ then only multiples of d can be decomposed. In this case, we set $g(a_1, \ldots, a_k) = \infty$. Note that it is not necessary that the a_i's be *pairwise* coprime for the Frobenius number to be finite.

In this paper, we look at the following problems. Fix an MCP instance $\{a_1, \ldots, a_k\}$.

- Frobenius Problem: Compute $g(a_1, \ldots, a_k)$.
- MCP Decision: Given a query integer M, is M decomposable over $\{a_1, \ldots, a_k\}$?
- MCP One Decomposition: Given a query integer M, return a decomposition of M over $\{a_1, \ldots, a_k\}$, if one exists.
- MCP All Decompositions: Given a query integer M, return all decompositions of M over $\{a_1, \ldots, a_k\}$.

Let $\gamma(M)$ denote the number of decompositions of M over $\{a_1, \ldots, a_k\}$. Obviously, any algorithm solving the MCP All Decompositions will have running time $\Omega(\gamma(M))$. The function $\gamma(M)$ grows quite rapidly, namely asymptotically with a polynomial in M of degree $k - 1$, see Section 6.

In Table 1, we give an overview of our algorithms' properties; residue tables and extended residue tables will be introduced in Section 3. Construction space in each case is $O(1)$ in addition to the space required by the data structure computed.

[4] also referred to as Parikh-vectors, compositions, composions, multiplicity vectors

problem	data structure	space	constr. time	query time
Frobenius	residue table	$O(a_1)$	$O(ka_1)$	$O(a_1)$
MCP decision	residue table	$O(a_1)$	$O(ka_1)$	$O(1)$
MCP one decomp.	residue table with witness vector	$O(a_1)$	$O(ka_1)$	$O(k)$
MCP all decomp.	extended residue t.	$O(ka_1)$	$O(ka_1)$	$O(ka_1\gamma(M))$

Table 1. Complexities of algorithms in the paper

2.1 Previous solutions

The three MCP problems can be solved with simple dynamic programming algorithms. These are variants of the original DP algorithm for the Coin Change Problem, due to Gilmore and Gomory [10, 18], where the number of coins has to be minimized. For the Decision and One Decomposition problems, construct a one-dimensional Boolean table A of size $M+1$, where $\mathrm{A}[m]=1$ if and only if m is decomposable over $\{a_1,\dots,a_k\}$. Table A can be computed using the recursion $\mathrm{A}[0]=1$, and for $m\geq 1$, $\mathrm{A}[m]=1$ if exists $1\leq i\leq k$ such that $\mathrm{A}[m-a_i]=1$, and 0 otherwise. Construction time is $O(kM)$ and one decomposition c can be produced by backtracking in time proportional to $|c|$, which is in the worst case $\frac{1}{a_1}M$.

To produce all decompositions, define a two-dimensional Boolean table B of size $k\cdot(M+1)$, where $\mathrm{B}[i,m]=1$ if and only if m is decomposable over $\{a_1,\dots,a_i\}$. The table can be computed with the following recursion: $\mathrm{B}[1,m]=1$ if and only if $m \bmod a_1=0$; for $i>1$, $\mathrm{B}[i,m]=\mathrm{B}[i-1,m]$ if $m<a_i$, and $\mathrm{B}[i,m]=\mathrm{B}[i-1,m]\vee\mathrm{B}[i,m-a_i]$ otherwise. A straightforward backtracking algorithm computes all decompositions of M. Running time is $O(kM)$ for the table construction, and $O(\sum_{c\in\mathcal{C}(M)}|c|)=O(\gamma(M)\cdot\frac{1}{a_1}M)$ for computation of the decompositions, while storage space is $O(kM)$.

Finally, a variant C of the full table computes $\gamma(M)$, the number of decompositions of M, in the last row, where the entries are integers instead of Booleans, using the recursion $\mathrm{C}[i,m]=\mathrm{C}[i-1,m]+\mathrm{C}[i,m-a_i]$. A similar recursion can be used to solve the Change Making Problem, where a minimum number of coins/masses is sought. For an arbitrary linear objective function, finding an *optimal* solution is known as Integer Knapsack Problem, and some approaches exist to find this optimal solution of the knapsack [15]. In this paper, we concentrate on generating all solutions, because objective functions for such problems are often far from linear [5].

3 Residue tables

Fix an MCP instance $\{a_1,\dots,a_k\}$. Now, for any $M\geq 0$, it can be immediately seen that

$$M \text{ is decomposable} \quad\Rightarrow\quad M+a_1 \text{ is decomposable.} \tag{1}$$

In fact, it even holds that $M+a_i$ is decomposable for any $i=1,\dots,k$, but we only need the statement for $i=1$. For $r=0,\dots,a_1-1$, let n_r be the smallest integer such that $n_r \bmod a_1=r$ and n_r is decomposable; set $n_r=\infty$ if no such integer exists. The n_r's are well-defined: If n has a decomposition, so has $n+a_1$, and $n\equiv n+a_1 \pmod{a_1}$. Clearly, $\sum_i c_ia_i=n_r$ implies $c_1=0$ because otherwise, n_r-a_1 has a decomposition, too. If the n_r's are known, then we can test in constant time whether some number M can be decomposed: Set $r=M \bmod a_1$, then

$$M \text{ is decomposable} \quad\Leftrightarrow\quad M\geq n_r. \tag{2}$$

The *residue table* RT is a one-dimensional table of size a_1 where $\mathrm{RT}(r)=n_r$. Given the values n_r for $r=0,\dots,a_1-1$, we can compute the Frobenius number $g(a_1,\dots,a_k)$ and the number ω of *omitted* values that cannot be decomposed over $\{a_1,\dots,a_k\}$ [6]:

$$g(a_1,\dots,a_k)=\max_r\{n_r\}-a_1 \quad\text{and} \tag{3}$$

$$\omega=\sum_r\left\lfloor\frac{n_r}{a_1}\right\rfloor=\frac{1}{a_1}\sum_r n_r-\frac{a_1-1}{2}. \tag{4}$$

Many algorithms for computing the Frobenius number rely on the above result. For example, Davison' algorithm [7] makes implicit use of this table. To explicitly compute the values n_r for $r=0,\dots,a_1-1$, Nijenhuis

[19] gave an algorithm with runtime $O(k\, a_1 \log a_1)$, where the $\log a_1$ factor is due to a binary heap structure that must be updated in every step. One can easily modify Nijenhuis' algorithm by using a Fibonacci heap instead of a binary heap, thereby reaching an $O(a_1(k + \log a_1))$ runtime bound, but the constant factor overhead (runtime and memory) is much higher for a Fibonacci heap.

For producing all decompositions of a query, more information is needed: Namely, for each $r = 0, \ldots, a_1 - 1$ and each $i = 1, \ldots, k$, the smallest number $n_{r,i}$ congruent r modulo a_1 such that $n_{r,i}$ is decomposable over $\{a_1, \ldots, a_i\}$. More formally, the *extended residue table* ERT is a two-dimensional table of size ka_1, such that

$$\text{ERT}(r,i) = \min\{n \mid n \equiv r \pmod{a_1} \text{ and } n \text{ is decomposable over } \{a_1, \ldots, a_i\}\}, \tag{5}$$

where $\text{ERT}(r,i) = \infty$ if no such integer exists. In Figure 1, we give a small example with $k = 4$. The last column of the extended residue table ERT is in fact the residue table RT of the instance.

Residue Table RT

r	n_r
0	0
1	16
2	12
3	8
4	9

Extended Residue Table ERT

r	$a_1 = 5$	$a_2 = 8$	$a_3 = 9$	$a_4 = 12$
0	0	0	0	0
1	∞	16	16	16
2	∞	32	17	12
3	∞	8	8	8
4	∞	24	9	9

Fig. 1. Residue table and extended residue table for the MCP instance $\{5, 8, 9, 12\}$

We will introduce fast algorithms for constructing both tables, as well as for producing one decomposition of a query using the residue table and an additional data structure, and for producing all decompositions of a query from the extended residue table.

4 The Round Robin algorithm

We compute the values n_r, for $r = 0, \ldots, a_1 - 1$, iteratively for the sub-problems "Find n_r for the instance $a_1, \ldots, a_i$" for $i = 1, \ldots, k$. For $i = 1$ we start with $n_0 = 0$ and $n_r = \infty$, $r = 1, \ldots, a_1 - 1$. When constructing the residue table, in each iteration, the current values for the $n_r, r = 0, \ldots, a_1 - 1$, are updated, while for the extended residue table, they are stored as separate columns. We describe only the construction algorithm for the residue table (Round Robin) in detail.

Suppose we know the correct values n'_r for the sub-problem $a_1, \ldots, a_{k-1}$ and want to calculate those of the original problem $a_1, \ldots, a_k$. We first concentrate on the simple case that $\gcd(a_1, a_k) = 1$. We initialize $n_r \leftarrow n'_r$ for all $r = 0, \ldots, a_1 - 1$ and $n \leftarrow n_0 = 0$. In every step of the algorithm, set $n \leftarrow n + a_k$ and $r \leftarrow n \bmod a_1$. Let $n \leftarrow \min\{n, n_r\}$ and $n_r \leftarrow n$. We repeat this loop until n equals 0.

In case all $a_2, \ldots, a_k$ are coprime to a_1, this short algorithm is already sufficient to find the correct values n_r. In Figure 1, the right side (the extended residue table) can be viewed as each column representing one iteration of the Round Robin algorithm. For example, focus on the column $a_3 = 9$. We start with $n = 0$. In the first step, we have $n \leftarrow 9$ and $r = 4$. Since $n < n_4 = 24$ we update $n_4 \leftarrow 9$. Second, we have $n \leftarrow 9 + 9 = 18$ and $p = 3$. In view of $n > n_3 = 8$ we set $n \leftarrow 8$. Third, we have $n \leftarrow 8 + 9 = 17$ and $r = 2$. Since $n < n_2 = 32$ we update $n_2 \leftarrow 17$. Fourth, we have $n \leftarrow 17 + 9 = 26$ and $r = 1$. In view of $n > n_1 = 16$ we set $n \leftarrow 16$. Finally, we return to $r = 0$ via $n \leftarrow 16 + 9 = 25$.

From the residue table, or from the last column of the extended residue table, we can see that the Frobenius number for this example is $g(5, 8, 9, 12) = 16 - 5 = 11$.

It is straightforward how to generalize the algorithm for $d := \gcd(a_1, a_i) > 1$: In this case, we do the updating independently for every residue $p = 0, \ldots, d - 1$: Only those n_r for $r \in \{0, \ldots, a_1 - 1\}$ are updated that satisfy $r \equiv p \pmod d$. To guarantee that the round robin loop completes updating after a_1/d steps, we have to start the

```
    Algorithm ROUND ROBIN
 1  initialize $n_0 = 0$ and $n_r = \infty$ for $r = 1, \dots, a_1 - 1$;
 2  for $i = 2, \dots, k$ do
 3      $d = \gcd(a_1, a_i)$;
 4      for $p = 0, \dots, d - 1$ do
 5          Find $n = \min\{n_q \mid q \bmod d = p, 0 \le q \le a_1 - 1\}$;
 6          If $n < \infty$ then repeat $a_1/d - 1$ times
 7              $n \leftarrow n + a_i$; $r = n \bmod a_1$;
 8              $n \leftarrow \min\{n, n_r\}$; $n_r \leftarrow n$;
 9          done;
10      done;
11  done.
```

Fig. 2. Construction algorithm of the residue table

loop from a minimal n_r with $r \equiv p \pmod{d}$. For $p = 0$ we know that $n_0 = 0$ is the unique minimum, while for $p \neq 0$ we search for the minimum first. See Figure 2 for the pseudo-code of the algorithm.

The inner loop (lines 6–9) will be executed only if the minimum $\min\{n_q\}$ is finite; otherwise, the elements of the residue class cannot be decomposed over $a_1, \dots, a_i$ because of $\gcd(a_1, \dots, a_i) > 1$.

Lemma 1. *Suppose that n'_r for $r = 0, \dots, a_1 - 1$ are the correct residue table entries for the MCP instance $a_1, \dots, a_{k-1}$. Initialize $n_r \leftarrow n'_r$ for $r = 0, \dots, a_1 - 1$. Then, after one iteration of the outer loop (lines 3–10) of the Round Robin Algorithm, the residue table entries equal the values n_r for $r = 0, \dots, a_1 - 1$ for the MCP instance $a_1, \dots, a_k$.*

Since for $k = 1$, $n_0 = 0$ and $n_r = \infty$ for $r \neq 0$ are the correct values for the MCP with one coin, we can use induction to show the correctness of the algorithm. To prove the lemma, we first note that for all $r = 0, \dots, a_1 - 1$,

$$n_r \le n'_r \quad \text{and} \quad n_r \le n_q + a_k \text{ for } q = (r - a_k) \bmod a_1 \tag{6}$$

after termination. Assume that for some n, there exists a decomposition $n = \sum_{i=1}^{k} c_i a_i$. We have to show $n \ge n_r$ for $r = n \bmod a_1$. Now, $\sum_{i=1}^{k-1} c_i a_i = n - c_k a_k$ is a decomposition of the MCP instance $a_1, \dots, a_{k-1}$ and for $q = (n - c_k a_k) \bmod a_1$ we have $n - c_k a_k \ge n'_q$. We conclude

$$n_r \le n_q + c_k a_k \le n'_q + c_k a_k \le n. \tag{7}$$

By an analogous argument, we infer $n_r = n$ for *minimal* such n. One can easily show that $n_r = \infty$ if and only if no n with $n \equiv r \pmod{a_1}$ has a decomposition with respect to the MCP instance $a_1, \dots, a_k$.

Under the standard model of computation, time and space complexity of the algorithm are immediate and we reach:

Theorem 1. *The Round Robin Algorithm computes the residue table of an instance $\{a_1, \dots, a_k\}$ of the Money Changing Problem in runtime $\Theta(k\, a_1)$ and extra memory $O(1)$.*

By equations (3) and (1), we thus have

Corollary 1. *Using the Round Robin Algorithm for computing the residue table, we can solve the Frobenius problem in time $O(ka_1)$, and a single MCP decision instance in time $O(ka_1)$. Alternatively, by computing the residue table in a preprocessing step in time $O(ka_1)$, we can then compute the Frobenius number in time $O(a_1)$ and anwer subsequent MCP decision instances in time $O(1)$.*

4.1 Finding one decomposition

For solving MCP one decomposition instances, we can compute an additional vector which we refer to as *witness vector*. This idea was first suggested but not detailed in [19]. We slightly modify the Round Robin algorithm such that, along with the residue table, it also constructs a vector w of length a_1 of pairs (index, multiplicity). See Figure 3 for the algorithm. The only change to the original Round Robin algorithm is an additional variable counter, which counts the number of times the current value a_i has been added. The rth entry of w is updated to $(i, \text{counter})$ if the current value of n using the current value a_i is smaller than in the previous iteration. Note that n still assumes the minimum of the value of n_r in the previous iteration and in the current one.

```
Algorithm ROUND ROBIN WITH WITNESS VECTOR
1  initialize $n_0 = 0$ and $n_r = \infty$ for $r = 1, \dots, a_1 - 1$;
2  for $i = 2, \dots, k$ do
3      $d = \gcd(a_1, a_i)$; $w(0) \leftarrow (1, 0)$;
4      for $p = 0, \dots, d - 1$ do
5          Find $n = \min\{n_q \mid q \bmod d = p, 0 \le q \le a_1 - 1\}$;
6          counter $\leftarrow 0$;
7          If $n < \infty$ then repeat $a_1/d - 1$ times
8              $n \leftarrow n + a_i$; $r = n \bmod a_1$; counter $\leftarrow$ counter $+1$;
9              if $n < n_r$ then
10                 $n_r \leftarrow n$; $w(r) \leftarrow (i, \text{counter})$;
11             else counter $\leftarrow 0$;
12             end if;
13             $n \leftarrow n_r$;
14         end if;
15     done;
16 done.
```

Fig. 3. Construction algorithm of the residue table with witness vector

Now the vector w can be used to construct a decomposition with a very simple algorithm: Initially, we set $c_i \leftarrow 0$ for all $i > 1$, and $c_1 = (M - m)/a_1$, where $m \leftarrow \text{RT}[M \bmod a_1]$. While $m > 0$, we repeat $(i, j) \leftarrow w(m \bmod a_1), c_i \leftarrow j, m \leftarrow m - ja_i$. We thus obtain a lexicographically maximal decomposition. Note that unlike the Change Making Problem, we do not try to minimize the number of coins used.

Example 1. Let's look at the MCP instance $\{5, 8, 9, 12\}$ again. Vector w will equal $((1, 0), (2, 2), (2, 4), (2, 1), (2, 3))$ after the first iteration ($i = 2$). In the next iteration ($i = 3$), $w(2)$ and $w(4)$ are updated to $(3, 1)$, and in the last, $w(2)$ again to $(4, 1)$. We thus end up with $w = ((1, 0), (2, 2), (4, 1), (2, 1), (3, 1))$. For query 451, we compute $(87, 2, 0, 0)$.

The construction algorithm uses $O(a_1)$ additional space and time compared to Round Robin, thus running time is $O(ka_1)$ and storage space $O(a_1)$.

The witness vector w has the following property: For $r = 0, \dots, a_1 - 1$, if $w(r) = (i, j)$, then

$$j = \max\{c_i \mid c = (c_1, \dots, c_k), \mu(c) = \text{RT}(r)\},$$

from which it directly follows that the number of iterations of the loop is at most $k - 1$. Thus, the running time of Algorithm FIND-ONE is $O(k)$. We summarize:

Theorem 2. *The MCP one decomposition problem can be solved in time $O(ka_1)$ and space $O(a_1)$, using the residue table with witness vector. Alternatively, we can compute the residue table with witness vector in a preprocessing step in time $O(ka_1)$, using $O(a_1)$ space, and answer subsequent MCP one decomposition questions in time $O(k)$.*

5 Finding all decompositions

For producing all decompositions of query integer M, we use backtracking through the extended residue table, which we construct with the algorithm Extended Round Robin, where the column in each iteration is saved instead of overwritten. Its running time is $O(ka_1)$, space for storing the extended residue table $O(ka_1)$.

For the backtracking, we choose an order of paths to follow such that we can make maximal use of information from the extended residue table. More formally, our algorithm is a recursive algorithm that maintains a current compomer c, an index i, and a current query m. It recursively decomposes values $m' = m - na_i$, grouping together values of n with the same residue modulo $\gcd(a_1, a_i)$. Since $x - n \operatorname{lcm}(a_1, a_i) \equiv x \pmod{a_1}$ for all $n, x \geq 0$, all these values can be handled together (for-loop, lines 5-11). Thus, we only have one table lookup for each value r modulo $\gcd(a_1, a_i)$ (line 7); the table entry $\textsc{ert}(i-1, r)$ serves as a lower bound.

At step i, the entries $c_k, c_{k-1}, \ldots, c_{i+1}$ of compomer c have already been filled in, and the remaining value $m = M - \sum_{j=i+1}^{k} c_j a_j$ will be decomposed over $\{a_1, \ldots, a_i\}$. The invariant at the call of $\textsc{find-all}(m, i, c)$ is that mass m is decomposable over $\{a_1, \ldots, a_i\}$ and $c_j = 0$ for $j = i, i-1, \ldots, 1$. We give the pseudo-code in Figure 4.

Algorithm FIND-ALL **(mass M, index i, compomer c)**
```
 1  if $i = 1$ then
 2      $c_1 \leftarrow M/a_1$; output $c$; return;
 3  end;
 4  $\mathrm{lcm} \leftarrow \mathrm{lcm}(a_1, a_i)$; $\ell \leftarrow \mathrm{lcm}/a_i$;      // least common multiple
 5  for $j = 0, \ldots, \ell - 1$ do
 6      $c_i \leftarrow j$; $m \leftarrow M - ja_i$;                // start with $j$ pieces of $a_i$
 7      $r \leftarrow m \bmod a_1$; lbound $\leftarrow$ ERT$(r, i-1)$;
 8      while $m \geq$ lbound do                    // $m$ is decomposable
 9          FIND-ALL$(m, i-1, c)$;                  //    over $\{a_1, \ldots, a_{i-1}\}$
10          $m \leftarrow m - \mathrm{lcm}$; $c_i \leftarrow c_i + \ell$;
11      done;
12  done.
```

Fig. 4. Algorithm for finding all decompositions using the Extended Residue Table. For a integer M which is decomposable over $\{a_1, \ldots, a_k\}$, $\textsc{find-all}(M, k, 0)$ will recursively produce all decompositions of M.

5.1 Correctness of the algorithm

By construction, any compomer computed by $\textsc{find-all}(M, k, 0)$ will have mass M. Conversely, fix m decomposable over $\{a_1, \ldots, a_i\}$, $i > 1$, and let

$$N(m, i) := \{m' = m - na_i \mid m' \geq 0 \text{ and } m' \text{ is decomposable over } \{a_1, \ldots, a_{i-1}\}\}. \tag{8}$$

We will show that for a call of $\textsc{find-all}(m, i, c)$, the set of values m' for which a recursive call is made (line 9 in Figure 4) is exactly $N(m, i)$. Then it follows by induction over $i = k, k-1, \ldots, 2$ that, given $c = (c_1, \ldots, c_k)$ with mass M, the algorithm will arrive at call $\textsc{find-all}(c_1 a_1, 1, (0, c_2, \ldots, c_k))$ and thus output c (line 2): In the induction step, set $m = M - \sum_{j=i+1}^{k} c_j a_j$ and $m' = m - c_i a_i$.

In order to prove the claim, let $\ell := a_1 / \gcd(a_1, a_i) = \operatorname{lcm}(a_1, a_i)/a_i$, and $r_{q,m} := m - q a_i \bmod a_1$, for $q = 0, \ldots, \ell - 1$. Now consider the sets

$$N(m, i, q) := \{m' \geq \textsc{ert}(r_{q,m}, i-1) \mid m' = m - na_i, n \equiv q \bmod \ell\}, \quad \text{for } q = 0, \ldots, \ell - 1. \tag{9}$$

Observe that $N(m, i, q)$ is exactly the set of values for which a recursive call is made within the while-loop (line 9) for $j = q$ (line 5). Clearly, $\cup_{q=0}^{\ell-1} N(m, i, q) \subseteq N(m, i)$. On the other hand, let $m' = m - na_i \in N(m, i)$. Further, let $r = m' \bmod a_1$ and $q = n \bmod \ell$. Since $r \equiv m' \bmod a_1$ and $m' = m - na_i \equiv m - qa_i \bmod a_1$, we have $m - qa_i \bmod a_1 = r$. Since m' is decomposable over $\{a_1, \ldots, a_{i-1}\}$, it must hold that $m' \geq \textsc{ert}(r, i-1)$ by property (5) of the Extended Residue Table. Thus, we have

$$N(m, i) = \bigcup_{q=0}^{\ell-1} N(m, i, q), \tag{10}$$

as claimed.

5.2 Complexity of the algorithm

As we have seen, step i of Algorithm FIND-ALL makes one recursive call for each $m' \in N(m,i,q)$, $q = 0,\dots,\ell-1$, where $\ell = \mathrm{lcm}(a_1,a_i)/a_i$ (line 9). By (10), each of these calls will produce at least one decomposition. In order to check which m' are in $N(m,i,q)$, the algorithm enters the while-loop at line 8, and will thus make one unsuccessful comparison before exiting the loop. In the worst case, the current call FIND-ALL(m,i,c) will produce only one decomposition; in this case, we will have $\ell-1$ additional comparisons. Since for all $i = 2,\dots,k$, $\ell = \mathrm{lcm}(a_1,a_i)/a_i \leq a_1$, we have

$$\text{number of comparisons for FIND-ALL}(M,k,0) \quad \leq \quad k a_1 \gamma(M). \tag{11}$$

The previous discussion yields the following theorem:

Theorem 3. *Given the Extended Residue Table of an MCP instance $\{a_1,\dots,a_k\}$ with smallest mass a_1, the* FIND-ALL *algorithm computes all decompositions of a query M in time $O(ka_1\gamma(M))$, where $\gamma(M)$ is the number of decompositions of M.*

6 Strings and compomers

We now take a closer look at the connection between strings and compomers, and then plot $\gamma(M)$, the number of compomers with mass M, for the most common biomolecule alphabets.

Let $\Sigma = \{\sigma_1,\dots,\sigma_k\}$ be a *weighted alphabet*, i.e., each character σ_i is assigned a positive integer $\mu(\sigma_i) = a_i$ (a *weight* or *mass*) by a mass function $\mu : \Sigma \to \mathbb{Z}^+$. For a string $s = s_1 \dots s_n \in \Sigma^*$, we define $\mu(s) := \sum_{i=1}^{n} \mu(s_i)$, the *mass* of s. We further define $\mathrm{comp}(s) = (c_1,\dots,c_k)$ by $c_i = \#\{j \mid 1 \leq j \leq n, s_j = \sigma_i\}$, the compomer associated with s. For a compomer $c = (c_1,\dots,c_k) \in (\mathbb{Z}_0^+)^k$, we set $\mu(c) := \sum_{i=1}^{k} c_i \cdot \mu(\sigma_i) = \sum_{i=1}^{k} c_i a_i$, the *mass* of c, and $|c| := \sum_{i=1}^{k} c_i$, the *length* of c. Obviously, if $c = \mathrm{comp}(s)$, then $\mu(c) = \mu(s)$ and $|c| = |s|$.

The following simple lemmas establish some connections between strings and compomers.

Lemma 2. *Given a compomer $c = (c_1,\dots,c_k)$, the number of strings s with* $\mathrm{comp}(s) = c$ *is* $\binom{n}{c_1,\dots,c_k} = \frac{n!}{c_1!\cdots c_k!}$, *where $n = |c|$.*

Proof. Clearly, any string s with $\mathrm{comp}(s) = c$ has length n. There are $\binom{n}{c_1,\dots,c_k}$ many ways of partitioning an n-set into k subsets of sizes $c_1,\dots,c_k$, which is exactly the number of ways of choosing how to position the c_i many σ_i's, for $0 \leq i \leq k$.

Lemma 3. *Given an integer $n \geq 0$, the number of compomers c with $|c| = n$ is $\binom{n+k-1}{k-1}$, and the number of compomers c with $|c| \leq n$ is $\binom{n+k}{k}$.*

Proof. Consider the following graphical representation of a compomer $c = (c_1,\dots,c_k)$ of length n: On a line of $n+k-1$ dots, place $k-1$ many crosses in the following way: Place a cross on dot number c_1+1, one on dot number c_1+c_2+2, etc. There are obviously $\binom{n+k-1}{k-1}$ many ways to do this, see Figure 5.

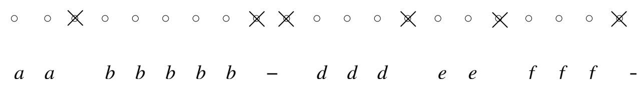

Fig. 5. Graphical representation of compomer $(2,5,0,3,2,3,0)$ over the alphabet $\Sigma = \{a,b,c,d,e,f,g\}$.

The second claim follows using the identity $\sum_{i=0}^{m} \binom{i+r}{i} = \binom{m+r+1}{r+1}$ for binomial coefficients.

Thus, the search space is greatly reduced by computing compomers instead of strings with a given mass. However, the number of compomers of a given mass M also increases very fast.

As before, we denote by $\gamma(M)$ the number of compomers with mass M. There is no "good" closed form known for $\gamma(M)$. The generating function method, by expanding the coefficients of an appropriately defined generating function (see e.g. [25]), yields the best closed form, but it is only of theoretical interest because the cost of the coefficient expansion, as well as the cost of the evaluation, are in applications usually too high. However, as we saw in Section 2, $\gamma(M)$ can be computed with a dynamic programming algorithm in time $O(kM)$, thus in time linear in M.

The following asymptotic result is due to I. Schur and can be shown using generating functions [25]: Given $0 < a_1 < \ldots < a_k \in \mathbb{Z}^+$ with $\gcd(a_1, \ldots, a_k) = 1$. Then,

$$\gamma(M) \sim \frac{M^{k-1}}{(k-1)! a_1 a_2 \cdots a_k} \qquad (M \to \infty). \tag{12}$$

Thus, the number of decompositions grows very rapidly with increasing M. In particular, for proteins, $\gamma(M)$ grows with a polynomial of degree 18. Note that the condition of coprimality can be easily achieved by choosing an appropriate computation precision.

In Figure 6, we plot $\gamma(M)$ for the three most common types of biomolecules: Peptides over the amino acid (AA) alphabet of 19 masses (having equated the two amino acids with identical mass, leucine (L) and isoleucine (I)), DNA compomers over the nucleotide alphabet of 4, and, the most general case, molecules over the alphabet of the 6 elements that are occur most commonly in biomolecules: carbon (C), hydrogen (H), nitrogen (N), oxygen (O), phosphor (P), and sulfur (S). In real life applications, one never seeks the exact explanation of a peak because some measurement inaccuracies always have to be taken into account. Thus, we merge the output within a certain range.

Computation precision for the AA and DNA alphabets is 10^{-3} and output is merged within 0.1 Da: This means that for the AA and DNA alphabets, the number of decompositions of 100 points are summed up. We also plot the actual number of decompositions for AA compomers up to 800 Da, as well as the maximum over this range. The minimum is always 0, since in this range, there are always values which are not decomposable. For DNA, the number of decompositions with precision 10^{-3} always yields either 1 or 0 decompositions (data not shown).

For the bioatom alphabet CHNOPS, we use computation precision 10^{-5} Da and merging within 0.001 Da, because with the parameters used for AA and DNA, combinatorial effects appear which in fact do not occur in reality. Thus, here 1 000 points are merged. The actual number of decompositions in the range up to 100 with the given precision is below 250 without merging; here, the variability is quite low because of the presence of the character H with mass approximately 1 Da [5].

7 Optimizing Round Robin

In this section, we give several optimization techiques of the Round Robin algorithm.

We can improve the Round Robin Algorithm in the following ways: First, we do not have to explicitly compute the greatest common divisor $\gcd(a_1, a_i)$. Instead, we do the first round robin loop (lines 6–9) for $r = 0$ with $n = n_0 = p = 0$ until we reach $n = p = 0$ again. We count the number of steps t to this point. Then, $d = \gcd(a_1, a_i) = \frac{a_1}{t}$ and for $d > 1$, we do the remaining round robin loops $r = 1, \ldots, d-1$.

Second, for $r > 0$ we do not have to explicitly search for the minimum in $N_r := \{n_q : q = r, r+d, r+2d, \ldots, r+(a_1-d)\}$. Instead, we start with $n = n_r$ and do exactly $t-1$ steps of the round robin loop. Here, $n_r = \infty$ may hold, so we initialize $p = r$ (line 5) and update $p \leftarrow (p + a_i) \bmod a_1$ separately in line 7. Afterwards, we continue with this loop until we first encounter some $n_p \leq n$ in line 8, and stop there. The second loop takes at most $t-1$ steps, because at some stage we reach the minimal $n_p = \min N_r$ and then, $n_p < n$ must hold because of the minimality of n_p. This compares to the t steps for finding the minimum.

Third, A. Nijenhuis suggested the following improvement (personal communication): Suppose that k is large compared to a_1, for example $k = O(a_1)$. Then, many round robin loops are superfluous because chances are high that some a_i is representable using $a_1, \ldots, a_{i-1}$. To exclude such superfluous loops, we can check after line 2 whether $n_p \leq a_i$ holds for $p = a_i \bmod a_1$. If so, we can skip this a_i and continue with the next index $i+1$, since this implies that a_i has a decomposition over $a_1, \ldots, a_{i-1}$. In addition, this allows us to find a minimal subset of $\{a_1, \ldots, a_k\}$ sufficient to decompose any number that can be decomposed over the original MCP instance $a_1, \ldots, a_k$.

Fourth, if $k \geq 3$ then we can skip the round robin loop for $i = 2$: The Extended Euclid's Algorithm [16] computes integers d, u_1, u_2 such that $a_1 u_1 + a_2 u_2 = d = \gcd(a_1, a_2)$. Hence, for the MCP instance a_1, a_2 we have $n_p = \frac{1}{d}((p\, a_2 u_2) \bmod (a_1 a_2))$ for all $p \equiv 0 \pmod{d}$, and $n_p = \infty$ otherwise. Thus, we can start with the round robin loop for $i = 3$ and compute the values n'_p of the previous instance a_1, a_2 on the fly using the above formula.

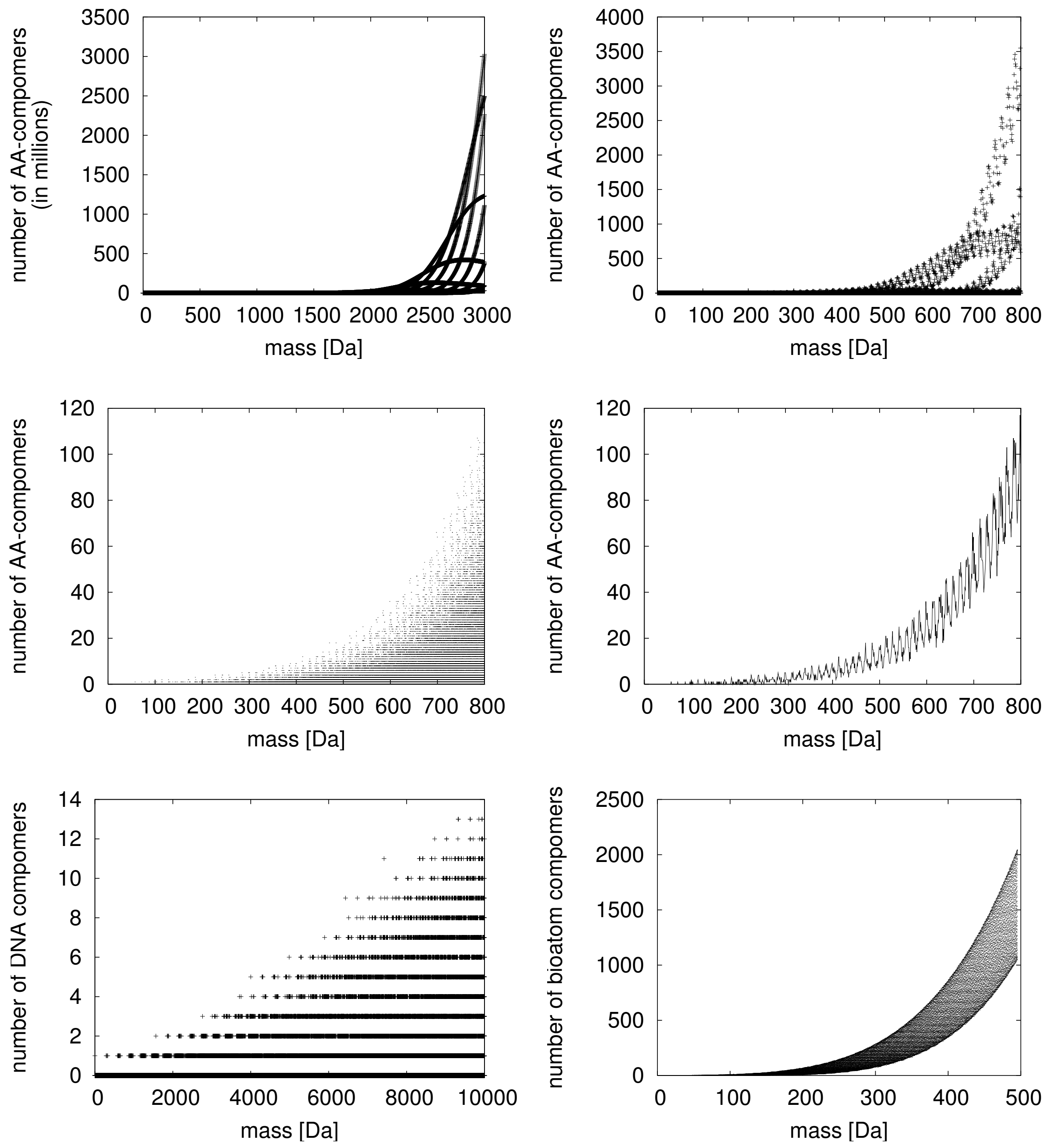

Fig. 6. Top: Number of amino acid compomers up to 3000 Da (left) and up to 800 Da (right). Computation precision is 0.001, output is merged within 0.1 Da. Center: AA compomers without merging, precision 0.001 (left), maximum of same (right). Bottom: Number of DNA compomers up to 10 000 Da, with precision 0.001, output merged within 0.1 Da (left); and biomolecules up to 500 Da (right), precision 0.00001, output is merged within 0.01 Da.

Our last improvement is based on the following observation: The residue table $(n_p)_{p=0,\dots,a_1-1}$ is very memory consuming, and every value n_p is read (and eventually written) exactly once during any round robin loop. Nowadays processors usually have a layered memory access model, where data is temporarily stored in a cache that has much faster access times than the usual memory. The following modification of the Round Robin Algorithm allows for cache-optimized access of the residue table: We exchange the $r = 0, \dots, \gcd(a_1, a_i) - 1$ loop (lines 4–10) with the inner round robin loop (lines 6–9). In addition, we make use of the second improvement introduced above and stop the second loop as soon as *none* of the n_{p+r} was updated. Now, we may assume that the consecutive memory access of the loop over r runs in cache memory.

We want to roughly estimate how much runtime this improvement saves us. For two random numbers u, v drawn uniformly from $\{1, \dots, n\}$, the expected value of the greatest common divisor is approximately $\mathbb{E}\big(\gcd(u, v)\big) \approx 6/\pi^2 H_n$, where H_n is the n'th harmonic number [16]. This leads us to the approximation $\mathbb{E}\big(\gcd(u, v)\big) \approx 1.39 \cdot \log_{10} n + 0.35$, so the expected greatest common divisor grows logarithmically with the coin values, for random input.[5] Even so, the improvement has relatively small impact on average: Let t_{mem} denote the runtime of our algorithm in main memory, and t_{cache} the runtime for the same instance of the problem if run in cache memory. For an instance $a_1, \ldots, a_k$ of MCP, the runtime t_{mod} of our modified algorithm roughly depends on the values $1/\gcd(a_1, a_i)$:

$$t_{\text{mod}} \approx t_{\text{cache}} + (t_{\text{mem}} - t_{\text{cache}}) \frac{1}{k-1} \sum_{i=2}^{k} \frac{1}{\gcd(a_1, a_i)}.$$

For random integers u, v uniformly drawn from $\{1, \ldots, n\}$ we estimate (analogously to [16], Section 4.5.2)

$$\mathbb{E}\left(\frac{1}{\gcd(u,v)}\right) \approx \sum_{d=1}^{n} \frac{p}{d^2} \frac{1}{d} = p \sum_{d=1}^{n} \frac{1}{d^3} = pH_n^{(3)} \quad \text{with } p = 6/\pi^2,$$

where the $H_n^{(3)}$ are the harmonic numbers of third order. The $H_n^{(3)}$ form a monotonically increasing series with $1.2020 < H_n^{(3)} < H_\infty^{(3)} < 1.2021$ for $n \geq 100$, so $\mathbb{E}\big(1/\gcd(u,v)\big) \approx 0.731$. If we assume that accessing main memory is the main contributor to the overall runtime of the algorithm, then we reduce the overall runtime by roughly one fourth. This agrees with our runtime measurements for random input as reported in the next section.

```
    Round Robin Algorithm (optimized version for $k \geq 3$)
1   Initialize $n_0, \ldots, n_{a_1-1}$ for the instance $a_1, a_2, a_3$; // fourth improvement
2   For $i = 4, \ldots, k$ do
3       If $n_p \leq a_i$ for $p = a_i \bmod a_1$ then continue with next $i$; // third improvement
4       $d = \gcd(a_1, a_i)$;
5       $p \leftarrow 0$; $q \leftarrow a_i \bmod a_1$; // $p$ is source residue, $q$ is destination residue
6       Repeat $a_1/d - 1$ times
7           For $r = 0, \ldots, d-1$ do
8               $n_{q+r} \leftarrow \min\{n_{q+r}, n_{p+r} + a_i\}$;
9           done;
10          $p \leftarrow q$; $q \leftarrow (q + a_i) \bmod a_1$;
11      done;
12      // update remaining entries, second improvement
13      Repeat
14          For $r = 0, \ldots, d-1$ do
15              $n_{q+r} \leftarrow \min\{n_{q+r}, n_{p+r} + a_i\}$;
16          done;
17          $p \leftarrow q$; $q \leftarrow (q + a_i) \bmod a_1$;
18      until no entry $n_{q+r}$ was updated;
19  done.
```

Fig. 7. Optimized version of the Round Robin Algorithm.

In Figure 7 we have incorporated all but the first improvements into the Round Robin Algorithm; note that the first and last improvement cannot be incorporated simultaneously. All improvements presented are runtime heuristics, so the resulting algorithm still has runtime $O(k\, a_1)$.

8 Computational results

In this section, we report simulation results for our two main algorithms, Round Robin and FindAll.

8.1 Simulations for Round Robin

We generated 12 000 random instances of MCP, with $k = 5, 10, 20$ and $10^3 \leq a_i \leq 10^7$. We have plotted the runtime of the optimized Round Robin Algorithm against a_1 in Figures 8 and 9. As expected, the runtime of the

[5] For simplicity, we ignore the fact that due to the sorting of the input, $a_1 = \min\{a_1, \ldots, a_k\}$ is *not* drawn uniformly, and we also ignore the dependence between the drawings.

algorithm is mostly independent of the structure of the underlying instance. The processor cache, on the contrary, is responsible for major runtime differences. The left plot contains only those instances with $a_1 \leq 10^6$; here, the residue table of size a_1 appears to fit into the processor cache. The right plot contains all random instances; for $a_1 > 10^6$, the residue table has to be stored in main memory.

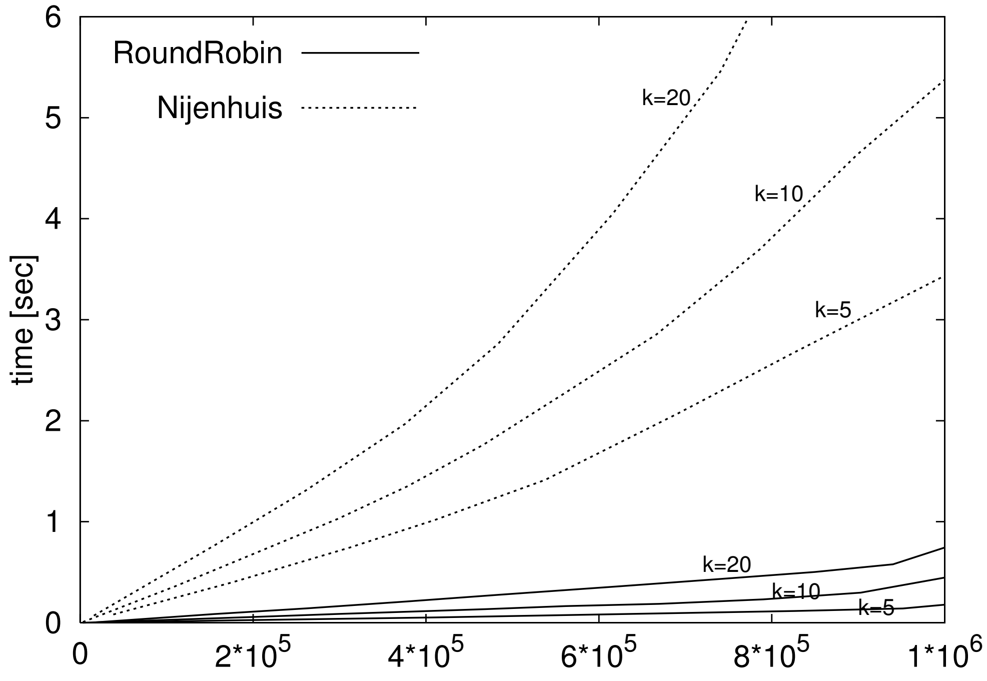

Fig. 8. Runtime of Round Robin vs. a_1 for $k = 5, 10, 20$ where $a_1 \leq 10^6$

To compare runtimes of the Round Robin Algorithm with those of Nijenhuis' algorithm [19], we re-implemented the latter in C++ using a binary heap as the priority queue. As one can see in Figures 8 and 9, the speedup of our optimized algorithm is about 10-fold for $a_1 \leq 10^6$, and more than threefold otherwise. Regarding Kannan's algorithm [14], the runtime factor $k^{k^k} > 10^{2184}$ for $k = 5$ makes it impossible to use this approach.

Comparing the original Round Robin Algorithm with the optimized version, the achieved speedup was 1.67-fold on average (data not shown).

We also tested our algorithm on some instances of the Money Changing Problem that are known to be "hard": We used all twenty-five examples from [1], along with runtimes given there for standard linear programming based branch-and-bound search. These instances satisfy $5 \leq k \leq 10$ and $3719 \leq a_1 \leq 48709$. The runtime of the optimized Round Robin Algorithm (900 MHz UltraSparc III processor, C++) for every instance is below 10 ms, see Table 2. Note that in [1], Aardal and Lenstra do not compute the Frobenius number g but only verify that g cannot be decomposed. In contrast, the Round Robin Algorithm computes the residue table of the instance, which in turn allows to answer *all* subsequent questions whether some n is decomposable, in constant time. Still and all, runtimes usually compare well to those of [1] and clearly outperform LP-based branch-and-bound search (all runtimes above 9000 ms), taking into account the threefold processor power running the Round Robin Algorithm.

8.2 Simulations for FindAll

We implemented both algorithms for the MCP All Decompositions Problem (the classical dynamic programming and our algorithm). Runtimes on a SUN Fire 880, 900 MHz, for the amino acid alphabet are shown in Figure 10. In the total runtime of the preprocessing together with the algorithm producing all decompositions, our algorithm is greatly superior to the dynamic programming algorithm. This is mainly due to our very fast preprocessing Extended Round Robin which has constant runtime; we also plot the times for the preprocessing separately from the algorithms for producing all decompositions. The algorithms producing all decompositions, without the preprocessing stage, are very close, with our algorithm slighty slower. Since the runtime depends on the output

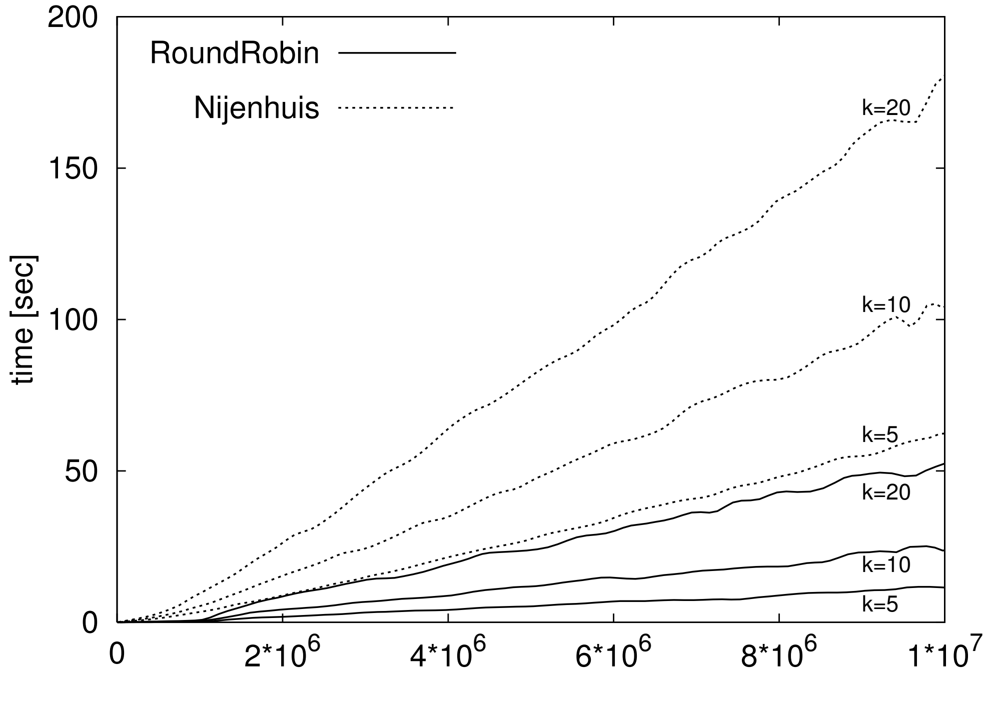

Fig. 9. Runtime of Round Robin vs. a_1 for $k = 5, 10, 20$ where $a_1 \leq 10^7$

Instance	c1	c2	c3	c4	c5	p1	p2	p3	p4	p5	p6	p7	p8
Aardal & Lenstra	1	1	1	1	1	1	1	2	1	2	1	2	1
Round Robin	0.8	0.9	0.9	1.1	1.6	3.1	1.2	4.9	9.2	4.0	3.4	3.1	2.8

Instance	p9	p10	p11	p12	p13	p14	p15	p16	p17	p18	p19	p20
Aardal & Lenstra	3	2	5	12	6	12	80	80	150	120	100	5
Round Robin	0.4	6.5	1.8	2.1	2.4	1.8	2.1	6.4	2.5	3.7	3.2	9.0

Table 2. Runtimes on instances from [1] in milliseconds, measured on a 359 MHz UltraSparc II (Aardal & Lenstra) and on a 900 MHz UltraSparc III (Round Robin).

size, we also plot the time needed per decomposition computed. We ran the algorithms with precision 0.01 and 0.001 Da. We plot the results for 0.01 Da; those for 0.001 are very similar.

For the DNA and CHNOPS alphabets, the runtime comparisons are similar (data not shown).

From our simulations, it appears that the dynamic programming algorithm also has constant runtime per decomposition, even though we have no formal proof for this. However, our algorithm is still greatly superior in its preprocessing time and its space requirements.

9 Conclusion

We presented new and efficient algorithms for the Money Changing Problem in its different flavours: for computing the Frobenius number of an MCP instance $\{a_1, \ldots, a_k\}$, for deciding whether a query is decomposable over $\{a_1, \ldots, a_k\}$, for computing one decomposition of a query, and for computing all decompositions of a query.

Our first main algorithm, Round Robin, constructs the residue table of an MCP instance of size $O(a_1)$, which in turn allows to efficiently solve the first three variants of the problem (the third with a minor modification). With a running time of $O(ka_1)$ and space requirements of $O(1)$ in addition to the residue table, it outperforms the best previously known algorithm, both theoretically and on known 'hard' instances from the literature. Moreover, it is very simple to understand and to implement.

Our second main algorithm, FindAll, computes all decompositions of a query integer using backtracking in the extended residue table of an MCP instance, which has size $O(ka_1)$. Its running time is $O(ka_1)$ per decomposition, thus the total runtime is linear in the size of the output.

We applied our algorithm FindAll to interpreting MS peaks and showed that it far outperforms the classical DP algorithm. The fact that FindAll has runtime linear in the size of the output is crucial for it to be useful

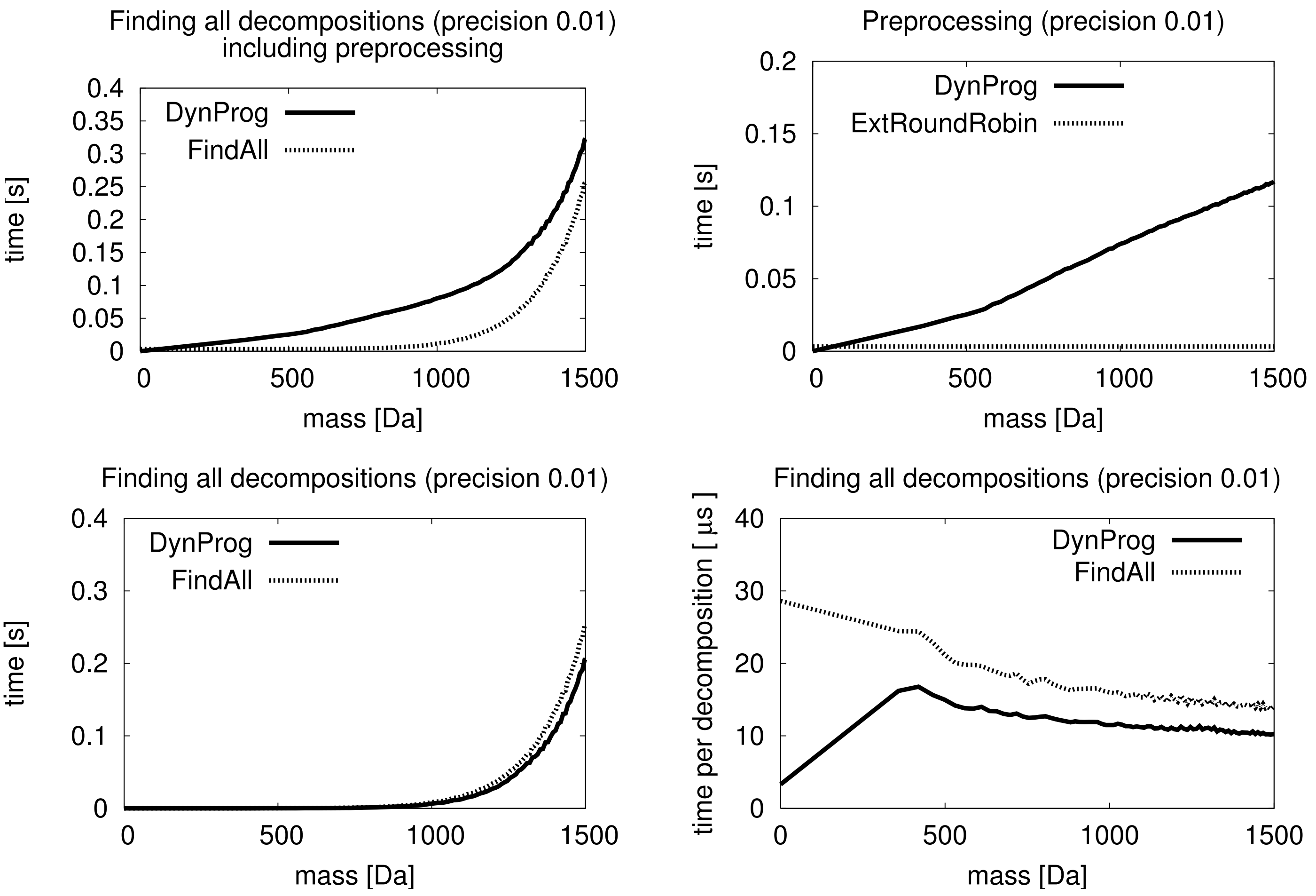

Fig. 10. Runtime comparisons at precision 0.01 Da. Top: Runtime for finding all decompositions, including preprocessing (left) and only for preprocessing (right). Bottom: Runtime for finding all decompositions excluding preprocessing: absolute time (left), and time per decomposition (right).

in interpreting mass spectrometry peaks, because of the sheer size of the search space. We have shown that our algorithm performs well in simulations with different biomolecules. We have also been successful in applying it to interpreting metabolomic mass spectrometry data (published elsewhere [5]). For this latter application, our approach results in a 100-fold reduction of memory and much better runtimes when compared to classical dynamic programming.

Acknowledgements

Both authors were supported by the Deutsche Forschungsgemeinschaft (BO 1910/1) within the Emmy Noether Computer Science Action Program at the time of submission. ZsL is currently supported by the Alexander von Humboldt Foundation and the Bundesministerium für Bildung und Forschung, within the research group "Combinatorial Search Algorithms in Bioinformatics." Implementation and simulations by Marcel Martin and Henner Sudek. Graphs by Marco Kortkamp, Marcel Martin, and Henner Sudek. We thank Stan Wagon and Albert Nijenhuis for helpful discussions, and an anonymous referee for his or her spot-on comments.

Bibliography

[1] K. Aardal and A. K. Lenstra. Hard equality constrained integer knapsacks. In *9th International Integer Programming and Combinatorial Optimization Conference (IPCO 2002)*, pages 350–366, 2002.

[2] R. Aebersold and M. Mann. Mass spectrometry-based proteomics. *Nature*, 422:198–207, March 2003.

[3] D. E. Beihoffer, J. Hendry, A. Nijenhuis, and S. Wagon. Faster algorithms for Frobenius numbers. *Electronic Journal of Combinatorics*, 12(#R27), 2005.

[4] M. Bertrand, P. Thibault, M. Evans, and D. Zidarov. Determination of the empirical formula of peptides by fast atom bombardment mass spectrometry. *Biomed. Environ. Mass Spectrom.*, 14(6):249–256, 1987.

[5] S. Böcker, M. C. Letzel, Zs. Lipták, and A. Pervukhin. Decomposing metabolomic isotope patterns. In *Proc. of the 6th Workshop on Algorithms in Bioinformatics (WABI 2006), LNBI/LNCS 4175*, pages 12–23, 2006.

[6] A. Brauer and J. E. Shockley. On a problem of Frobenius. *J. Reine Angew. Math.*, 211:215–220, 1962.

[7] J. L. Davison. On the linear diophantine problem of Frobenius. *J. Number Theory*, 48(3):353–363, 1994.

[8] E. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.

[9] A. Fürst, J.-T. Clerc, and E. Pretsch. A computer program for the computation of the molecular formula. *Chemometrics and Intelligent Laboratory Systems*, 5:329–334, 1989.

[10] P. Gilmore and R. Gomory. Multi-stage cutting stock problems of two and more dimensions. *Oper. Res.*, 13:94–120, 1965.

[11] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, second edition, 1994.

[12] H. Greenberg. Solution to a linear diophantine equation for nonnegative integers. *J. Algorithms*, 9(3):343–353, 1988.

[13] B. R. Heap and M. S. Lynn. A graph-theoretic algorithm for the solution of a linear diophantine problem of Frobenius. *Numer. Math.*, 6:346–354, 1964.

[14] R. Kannan. Lattice translates of a polytope and the Frobenius problem. *Combinatorica*, 12:161–177, 1991.

[15] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer, 2004.

[16] D. E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms.* Addison-Wesley, 3 edition, 1997.

[17] G. S. Lueker. Two NP-complete problems in nonnegative integer programming. Technical Report TR-178, Department of Electrical Engineering, Princeton University, March 1975.

[18] S. Martello and P. Toth. *Knapsack Problems.* John Wiley and Sons, Chichester, 1990.

[19] A. Nijenhuis. A minimal-path algorithm for the "money changing problem". *Amer. Math. Monthly*, 86:832–835, 1979. Correction in *Amer. Math. Monthly* **87** (1980) 377.

[20] S. C. Pomerantz, J. A. Kowalak, and J. A. McCloskey. Determination of oligonucleotide composition from mass spectromectrically measured molecular weight. *J. Am. Soc. Mass Spectrom.*, 4:204–209, 1993.

[21] J. L. Ramírez-Alfonsín. Complexity of the Frobenius problem. *Combinatorica*, 16(1):143–147, 1996.

[22] J. L. Ramírez-Alfonsín. *The Diophantine Frobenius Problem.* Oxford University Press, 2005.

[23] J. J. Sylvester and W. J. Curran Sharp. Problem 7382. *Educational Times*, 37:26, 1884.

[24] M. Wehofsky, R. Hoffmann, M. Hubert, and B. Spengler. Isotopic deconvolution of matrix-assisted laser desorption/ionization mass spectra for substance-class specific analysis of complex samples. *J. Mass Spectrom.*, 7:39–46, 2001.

[25] H. Wilf. *generatingfunctionology.* Academic Press, 1990.

[26] H. S. Wilf. A circle-of-lights algorithm for the "money-changing problem". *Amer. Math. Monthly*, 85:562–565, 1978.

[27] J. Wright. The change-making problem. *J. Assoc. Comput. Mach.*, 22(1):125–128, 1975.