Algorithmic complexity of protein identification: combinatorics of weighted strings[☆]

Mark Cieliebak[a], Thomas Erlebach[b], Zsuzsanna Lipták[a], Jens Stoye[c], Emo Welzl[a]

[a] *ETH Zurich, Institute for Theoretical Computer Science, Zurich, Switzerland*
[b] *ETH Zurich, Computer Engineering and Networks Laboratory, Zurich, Switzerland*
[c] *Universität Bielefeld, Technische Fakultät, Bielefeld, Germany*

Received 21 December 2001; received in revised form 7 June 2002; accepted 15 October 2002

Abstract

We investigate a problem which arises in computational biology: Given a constant-size alphabet $\mathscr{A}$ with a weight function $\mu : \mathscr{A} \to \mathbb{N}$, find an efficient data structure and query algorithm solving the following problem: For a string σ over $\mathscr{A}$ and a weight $M \in \mathbb{N}$, decide whether σ contains a substring with weight M, where the weight of a string is the sum of the weights of its letters (One-String Mass Finding Problem). If the answer is **yes**, then we may in addition require a witness, i.e., indices $i \leqslant j$ such that the substring beginning at position i and ending at position j has weight M. We allow preprocessing of the string and measure efficiency in two parameters: storage space required for the preprocessed data and running time of the query algorithm for given M. We are interested in data structures and algorithms requiring subquadratic storage space and sublinear query time, where we measure the input size as the length n of the input string σ. Among others, we present two non-trivial efficient algorithms: Lookup solves the problem with $O(n)$ storage space and $O(n/\log n)$ time; Interval solves the problem for binary alphabets with $O(n)$ storage space in $O(\log n)$ query time. We introduce other variants of the problem and sketch how our algorithms may be extended for these variants. Finally, we discuss combinatorial properties of weighted strings.

Keywords: Computational Biology; Protein Identification; Weighted Strings

[☆] A preliminary version of some of the results contained in this paper were published in “Proceedings of the 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002): Foundations of Information Technology in the Era of Network and Mobile Computing”, Montreal, Canada, Sept. 2002. Ricardo Baeza–Yates, Ugo Montanari, and Nicola Santoro (editors). Kluwer Academic Publishers, pp. 143–156.

E-mail addresses: cielieba@inf.ethz.ch (M. Cieliebak), erlebach@tik.ee.ethz.ch (T. Erlebach), zsuzsa@inf.ethz.ch (Zs. Lipták), stoye@techfak.uni-bielefeld.de (J. Stoye), emo@inf.ethz.ch (E. Welzl).

1. Introduction

In the present paper, we introduce a combinatorial problem which originates from computational biology: Given a string σ over a weighted alphabet $\mathscr{A}$, find a data structure and a query algorithm which, for a given weight $M \in \mathbb{N}$, decides whether σ has a (contiguous) substring of weight M, where the weight of a string is the sum of the weights of its letters. If the answer is **yes**, we may in addition ask for a *witness*, i.e., two positions within σ where a substring with weight M begins and ends. The actual problem in computational biology is to find several masses $M_1, \ldots, M_m$ in a database of strings. We concentrate on the one-string problem because algorithms can be easily extended to the multiple-string problem. We formally define the other problem variants in Section 5 and sketch how extensions may be designed. There are two simple algorithms which solve the one-string problem: One uses linear time for a query and no additional storage space; the other has logarithmic query time, but requires a preprocessing step and additional storage space for the resulting data structure which may be quadratic. Hereby, space and time complexities are measured in the length of the input string. We are interested in algorithms that are better than these two simple algorithms: i.e., we allow preprocessing and look for an algorithm where the data structure needs subquadratic space *and* the running time for a query is sublinear.

Formulated in this way, the problem becomes a purely combinatorial and algorithmic problem: Are there algorithms which allow searching in weighted strings of length n with $o(n^2)$ additional storage space and $o(n)$ query time? If so, can we find a tradeoff between space and time?

The problem differs from traditional string searching problems in one important aspect: While those look for substructures of strings (substrings, non-contiguous subsequences, particular types of substrings such as repeats, palindromes, etc.), we are interested only in *weights* of substrings. This means that, on the one hand, we lose a lot of the structure of strings: e.g., the weight of a string is invariant under permutation of letters; on the other, we gain the additional structure of the weight function, such as its additivity. For instance, the problem of searching in $X + Y$, where X and Y are two sets of numbers, turns out to be closely related to our problem (see [9,14]). However, we have been able to extend negative results which have been reached for that problem [5]: We can show that this approach (using the naïve solution without preprocessing) cannot lead to an efficient algorithm for our problem. Likewise, using suffix trees, which can be applied to efficiently solve a large number of complex string problems, does not seem to help. Note, for instance, that the longest common substring problem [13], although at first sight related, has very different characteristics. A problem that may also appear to be close to the present one is maximum segment sum [4]; however, it appears that it does not lead to good solutions, either. Encoding biological strings on a binary alphabet is not feasible here, because that would only allow very restricted mass functions. The only result related to our problem that we have found in the vast amount of literature on strings (e.g. [1,13,17,24,6]) is one that does not deal with combinatorics, but rather with language classes (see Section 6 for more details).

Our problem is positioned between the areas of string algorithms, search algorithms, and algebra. We believe that it is not only relevant for computational biology, but that it is also of theoretical interest to the field of combinatorial searching. As far as we are aware, no efficient algorithms have so far been presented for this problem.

We would like to stress at this point that, even though the source of our problem is a biological question, the results we present here are primarily of theoretical interest. The reason is twofold: First, none of the algorithms we present are efficiently applicable in their current form. LOOKUP requires sublinear query time, but this is a mainly asymptotic result, since the query time only improves for very long input strings. Algorithm INTERVAL, on the other hand, is very efficient both in query time and storage space, but it only works for alphabets of size 2, a case which does not occur in the usual biological setting. The second reason is that all biological data are prone to errors; in fact, there is no such thing as error-free data. Thus, all applications in computational biology need to be highly fault tolerant. Our algorithms can be straightforwardly adapted to become tolerant to measurement errors. However, this aspect is not included in this paper.

Thus, the present paper demonstrates that efficient algorithms for the problem presented are possible; due to side constraints from the experimental side and difficult problems regarding the definition of a realistic similarity measure on mass spectra, it remains a great challenge to actually find algorithms that are also of more practical value, though.

1.1. Biological motivation

Proteomics is the field that investigates the proteins expressed at a certain time in a specific cell type. Due to the development of novel techniques for protein identification that are compatible with high throughput approaches, large amounts of data are being accumulated at present, which presents particular computational and mathematical challenges.

Proteins are large molecules that play a fundamental role in all living organisms. They are made up of smaller molecules (amino acids) that are linked together in a certain order by peptide bonds. The sequence of amino acids constitutes the so-called *primary structure* of a protein. Protein size ranges from below 100 to several thousand amino acids, where a typical protein has length 300–600. Most proteins are made up of the 20 most common amino acids. For the purposes of this paper, we will view a protein as a finite string over an alphabet of size 20.

The information about the primary structure of known proteins is stored in large databases, such as SWISS-PROT (approximately 100 000 proteins) or PIR (more than 200 000 proteins). When a protein is isolated, one would like to know whether it is already known and if so, to identify it. An obvious way is to establish its primary structure: This is called *de novo protein sequencing*. However, protein sequencing, unlike DNA sequencing, is very expensive (both in time and money!). E.g. identifying *one* amino acid with Edman degradation, one standard method for protein sequencing, takes about 45 minutes, which makes this approach unfeasible in a high-throughput context.

Therefore, methods are required that test the protein against a database without having to sequence it first. One such method—which we will investigate here—makes use of the differences in molecular weights of amino acids: The protein is broken up into smaller pieces and the molecular mass of these pieces is then determined by a process called *mass spectrometry*. This yields a “fingerprint” of the protein that can then be tested against the database: The goal is to find a protein in the database that has substrings matching each of the input masses.

The method used for breaking up the protein into smaller pieces is referred to as *digestion*: a site-specific cleavage agent such as an enzyme, e.g. trypsin, is used that literally cuts the protein in certain well-defined places. Using digestion is algorithmically rather simple, at least with error-free data, since the breakup points are known in advance; it is thus possible to preprocess the database in an appropriate way. The complications arise due to measurement errors and post-translational modifications that alter the molecular mass of the amino acids. There is a large amount of literature on mass spectrometry [8,15,16,18,20,29]; some papers dealing with different aspects and modifications of the problem, e.g. the minimum number of masses needed to identify a protein [20], combinatorial [22] or probabilistic [3] models for scoring the difference of two mass spectra, or approaches for a correct identification even in the presence of post-translational modifications of the protein [19,23,28]. The review [27] as well as Chapter 11 of the book [21] contain more detailed introductions to this topic. For an introduction to computational biology in general, see [25]; for more on molecular biology [26]; while [12] is an easy-going introduction to genetics for non-biologists.

In this paper, we deal with algorithmic questions that arise if nothing is known about the breaking points, i.e., we assume random fragmentation. Testing for random weights is algorithmically far more complex than the digestion method, because the cutting places are not known in advance, and hence, it is impossible to compute the expected mass spectrum from the sequence. This approach allows combination of several cleavage agents and it eliminates problems caused by incomplete digestion. In addition, since we never make any assumptions about the probability distribution of breaking points, any algorithm for the random fragmentation method can be used for digested inputs, too. In the long run, however, for the biological application, algorithms are needed that are not only efficient, but also fault tolerant: They need to be tolerant both to measurement errors ($M \pm \varepsilon$; missing or additional masses in the spectrum) and to sequencing errors of the database entries.

1.2. Overview

The paper is organized as follows. We first introduce the problem and all necessary definitions in Section 2, where we also present some simple ideas that motivate our efficiency requirements. In Section 3, we design an algorithm (Lookup) that is asymptotically efficient, with linear storage space and sublinear query running time. Lookup thus serves to demonstrate that the requirements we defined earlier can be met. Section 4 contains an algorithm (Interval) that solves the problem for alphabets of size 2 and has a very good performance. However, we do not think that it can be generalized to larger alphabets. In Section 5, we present two other problem variants and discuss how algorithms for the original problem can be extended to these. In addition, we sketch improvements of our algorithms for special cases. Section 6 investigates combinatorial properties of weighted strings.

2. Problem and simple solutions

Fix an *alphabet* $\mathscr{A}$ of size $|\mathscr{A}| = s$ and a *mass function* $\mu : \mathscr{A} \to \mathbb{N}$. Let $\sigma = \sigma(1)\ldots\sigma(n)$ be a string over $\mathscr{A}$ of length $|\sigma| = n \geqslant 1$. We denote by $\sigma(i,j)$, where $1 \leqslant i \leqslant j \leqslant n$, the substring of σ starting at position i and ending at position j, i.e., $\sigma(i,j) = \sigma(i)\ldots\sigma(j)$. Thus, a non-empty string τ is a substring of σ, denoted $\tau \sqsubseteq \sigma$, if there are $1 \leqslant i \leqslant j \leqslant n$ s.t. $\tau = \sigma(i,j)$. Note that we do not consider the empty string to be a substring. The *mass* (or the *weight*) of σ is defined as the sum of the individual masses $\mu(\sigma) := \sum_{i=1}^{n} \mu(\sigma(i))$. For a mass $M \in \mathbb{N}$, we say that M is a *submass* of σ if σ has a substring of mass M. Finally, for $a \in \mathscr{A}$, let us denote the multiplicity of a in σ by $|\sigma|_a := |\{i \mid \sigma(i) = a\}|$. If the alphabet is $\mathscr{A} = \{a_1, \ldots, a_s\}$, then the *multiplicity vector* of a string σ over $\mathscr{A}$ is $\mathrm{mult}(\sigma) := (|\sigma|_{a_1}, \ldots, |\sigma|_{a_s})$.

The One-String Mass Finding Problem is defined as follows:

Given a string σ of length $|\sigma| = n$ and a mass M, is M a submass of σ?

A simple algorithm to solve the problem is Linsearch, which performs a linear search through the string: For given σ, start at position $\sigma(1)$ and add up masses until reaching the first position j s.t. $\mu(\sigma(1,j)) \geqslant M$. If the mass of the substring $\sigma(1,j)$ equals M, then output **yes** and stop; else start subtracting masses from the beginning of the string until the smallest index i s.t. $\mu(\sigma(i,j)) \leqslant M$ is reached. Repeat until finding a pair of indices (i,j) s.t. $\mu(\sigma(i,j)) = M$, or until reaching the end of the string (i.e., until the current substring is $\sigma(i,n)$ for some i and $\mu(\sigma(i,n)) < M$). The algorithm can be visualized as shifting two pointers ℓ and r through the string, where ℓ points to the beginning of the current substring and r to its end. Linsearch takes $O(n)$ time, since it looks at each position of σ at most twice. If we do not allow any preprocessing, this is asymptotically optimal, since it may be necessary to look at each position of σ at least once.

On the other hand, if preprocessing of σ is allowed, then there is another simple algorithm for the One-String Mass Finding Problem which uses binary search: in a preprocessing step, it calculates the set of all possible submasses of σ (i.e., $\mu(\sigma(i,j))$ for all $1 \leqslant i \leqslant j \leqslant n$) and stores them in a sorted array. Given a query mass M, it performs binary search for M in this array. We will refer to this algorithm as Binsearch. The space required to store the sorted array is proportional to the number of different submasses in σ, which is bounded by $O(n^2)$. The time for answering a query is thus $O(\log n)$.

Since submasses are integers, we can use a hash table instead of a sorted array to store all submasses of σ. In [10], hashing schemes are presented which require storage space linear in the number of elements to be stored, and which allow membership queries in constant time. For the One-String Mass Finding Problem, this yields an algorithm with space proportional to the number of different submasses in σ, and constant query time.

We assume integer weights because in the biological setting, only rational measurements are returned, which can be scaled canonically to integers, and because no irrational numbers can be processed computationally. However, from a mathematical point of view, assuming real weights would be more natural, and we will discuss special cases with real weights where applicable.

From now on, an algorithm for the One-String Mass Finding Problem will consist of three components: a preprocessing phase, a data structure in which the result of the preprocessing is stored, and a query method. For a string σ, the preprocessing will be done only once, while the query step will typically be repeated many times. For this reason, we are interested in algorithms with fast query methods, whereas we ignore time and space required for the preprocessing step (as long as they are within reasonable bounds). Space efficiency is measured in storage space required by the data structure.

We are looking for algorithms that are better than Linsearch and Binsearch, i.e., require storage space $o(n^2)$ for the data structure, and query time $o(n)$. We will call an algorithm *skinny* if the associated data structure requires $o(n^2)$ space, and *speedy* if the query method runs in time $o(n)$.

Another simple algorithm for the One-String Mass Finding Problem, which we will call BooleanArray, works as follows: In the preprocessing phase, define $W := \max\{\mu(a) \mid a \in \mathscr{A}\}$, and let B be a Boolean array of length $\mu(\sigma)$. Set $B[k]$ to `true` if and only if k is a submass of σ. Given a query mass M, we output $B[M]$. This algorithm has query time $O(1)$, while the data structure B requires $\mu(\sigma) \leqslant nW$ bits. Thus, the algorithm is speedy and, if $W = o(n)$, it is skinny, too. However, this does not solve the One-String Mass Finding Problem in general, since we do not want to restrict the size of W.

In the following, we assume that the alphabet $\mathscr{A}$ is of constant size and we do not restrict the maximum weight W of a letter. We assume a machine model with word size $L := \Omega(\log n + \log W)$ in which arithmetic operations on numbers with L bits can be executed in constant time; storage space is measured in terms of the number of machine words used. Without this assumption, we would get an extra factor L in the query time and in the storage space. Since the alphabet is of constant size, an input string σ of length n could be stored in $O(n/L)$ machine words. However, we will assume that the input string occupies n machine words.

3. An algorithm that is both skinny and speedy

In this section, we present algorithm Lookup that solves the One-String Mass Finding Problem with storage space $O(n)$ and query time $O(n/\log n)$. The idea is as follows. Similar to the simple linear search algorithm Linsearch introduced in Section 2, Lookup shifts two pointers along the sequence which point to the potential beginning and end of a substring with mass M. However, $c(n)$ steps of the simple algorithm are bundled into one step here. If $c(n)$ is chosen appropriately, i.e., approximately $\log n$, then this will reduce the number of steps from $O(n)$ to $O(n/\log n)$, while each step will still require only constant time. The storage space required will be $O(n)$. We will hereby heavily exploit the fact that the alphabet has constant size.

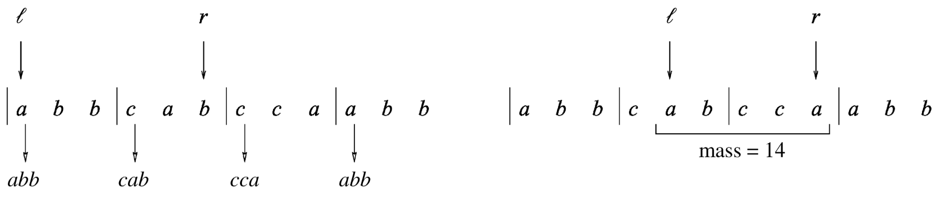


Fig. 1. Example 1—LOOKUP searching for $M = 14$.

3.1. An example

Example 1. Let $\mathscr{A} = \{a,b,c\}$, $\mu(a) = 1$, $\mu(b) = 2, \mu(c) = 5$. Let us assume that we are looking for $M = 14$ in $\sigma = abbcabccaabb$. LINSEARCH would shift two pointers ℓ and r through the sequence until reaching positions 5 and 9, respectively. Here, it would stop because the substring $\sigma(5,9) = abcca$ has weight 14. Let us assume that $c(n) = 3$. We divide the sequence σ into blocks of size $c(n)$. Now, rather than shifting the two pointers letter by letter, we will shift them by a complete block at a time. In order to do this, for each block we store a pointer to an index I corresponding to the substring which starts with the first letter of the block and ends with the last. Let us assume now that ℓ is at the beginning of the first block, and r is at the end of the second block, as indicated in Fig. 1. We are interested in the possible *changes* to the current submass if we shift the two pointers at most $c(n)$ positions to the right. Given a list of these, we could search for $M - \mu(\sigma(\ell,r))$. For example, the current submass in Fig. 1 is $\mu(\sigma(1,6)) = 13$, and we want to know whether, by moving ℓ and r at most 3 positions to the right, we can achieve a change of $14 - 13 = 1$.

We can calculate these possible changes and store them in a $(c(n)+1) \times (c(n)+1)$ matrix whose (i,j)-entry holds the submass change when ℓ is moved $i-1$ positions to the right, and r is moved $j-1$ positions to the right:

$$T[abb,cca]: \begin{pmatrix} 0 & 5 & 10 & 11 \\ -1 & 4 & 9 & 10 \\ -3 & 2 & 7 & 8 \\ -5 & 0 & 5 & 6 \end{pmatrix}.$$

Using this matrix, we can find out whether the difference we are looking for is there. In addition, we will store the entries of the matrix in a hash table that will allow us to make this decision in constant time. In the present case, 1 is not in the matrix, which tells us that we have to move one of the two pointers to the next block.

To determine which pointer to move, we consider what the linear search algorithm LINSEARCH would do when searching for M and starting in the current positions of the left resp. right pointer. Since M is not present within these two blocks, at least one of the two pointers would reach the end of its current block. Here, we want to move the pointer which would first reach the end of its block. We can determine which pointer this is if we compare the difference $M - \mu(\sigma(\ell,r))$ with the matrix entry corresponding to $c(n)-1$ moves of both the left and the right pointer (in this case 7). If the difference is smaller, we move the left pointer to the next block, otherwise we move the right one. In our example, we have a difference of 1, thus we move the left pointer to the next block.

This will change the current submass by -5 (the minimum of the array), yielding $\mu(\sigma(4,6)) = 13-5 = 8$. Thus, we now look for $M - \mu(\sigma(4,6)) = 14-8 = 6$. The matrix for this pair of positions is as follows:

$$T[cab, cca]: \begin{pmatrix} 0 & 5 & 10 & 11 \\ -5 & 0 & 5 & 6 \\ -6 & -1 & 4 & 5 \\ -8 & -3 & 2 & 3 \end{pmatrix}.$$

Value 6 is in the matrix: By looking in the matrix, we can see that a difference of 6 can be achieved by moving the left pointer by 1 position and the right pointer by three positions. The algorithm outputs positions 5 and 9 and then terminates.

3.2. *Algorithm* Lookup

We postpone the exact choice of the function $c(n)$ to the analysis, but assume for now that it is approximately $\log n$. For simplicity, we assume that $c(n)$ is a divisor of n.

Preprocessing: Given σ of length n, first compute $c(n)$. Next, build a table T of size $|\mathscr{A}|^{c(n)} \times |\mathscr{A}|^{c(n)}$. Each row resp. column of T will be indexed by a string from $\mathscr{A}^{c(n)}$. For $I, J \in \mathscr{A}^{c(n)}$, the table entry $T[I,J]$ contains the matrix of all differences $\mu(\text{prefix}(J)) - \mu(\text{prefix}(I))$ as described above. Furthermore, we store a hash table which contains the set of all entries of the matrix. Note that the table T depends only on n and $\mathscr{A}$, and not on the sequence σ itself. Next, divide σ into blocks of length $c(n)$. For each block, store a pointer to an index I that will be used to look up table T. Each such index I represents one string from $\mathscr{A}^{c(n)}$.

Query algorithm: Given M, set $\ell := 1$ and $r := 0$. Repeat the following steps until M has been found or $r > n$:

1. Say ℓ is set to the beginning of the ith block and r to the end of the $(j-1)$th block. The pointer of block i resp. j points to index I resp. J. Use the hash table stored in $T[I,J]$ to find whether difference $M - \mu(\sigma(\ell,r))$ is in the corresponding matrix, i.e., whether the difference can be achieved by moving ℓ resp. r at most $c(n)$ positions to the right.
2. If a difference of $M - \mu(\sigma(\ell,r))$ can be found, search for an entry (k,l) in the matrix stored in $T(I,J)$ which equals $M - \mu(\sigma(\ell,r))$ by exhaustive search,[1] and return **yes**, along with the witness $i' := (i-1)c(n)+k, j' := (j-1)c(n)+(l-1)$, since $\mu(\sigma(i',j'))$ has mass M.

[1] Alternatively, we could have stored (k,l) during the preprocessing, too.

3. Otherwise, $M - \mu(\sigma(\ell, r))$ cannot be found. If $M - \mu(\sigma(\ell, r))$ is less than the matrix entry at position $(c(n), c(n))$, then increment ℓ by $c(n)$ and set $\mu(\sigma(\ell, r)) := \mu(\sigma(\ell, r)) + \min(\text{array})$; otherwise, increment r by $c(n)$ and set $\mu(\sigma(\ell, r)) := \mu(\sigma(\ell, r)) + \max(\text{array})$.

Analysis: First we derive formulas for space and time, and then we show how to choose $c(n)$. To store one entry of table T, we have to store a matrix with $(c(n)+1)^2$ differences, and the corresponding hash table. We use a hashing scheme which requires space $\mathrm{O}(c(n)^2)$ and which allows membership queries in constant time (such hashing schemes exist for a finite universe U of integers, see e.g. [10]).

The space needed for storing the entire table T is

$$\begin{aligned}&(\text{number of entries in } T) \cdot \mathrm{O}(c(n)^2)\\ &= |\mathscr{A}|^{2c(n)} \cdot \mathrm{O}(c(n)^2)\\ &= \mathrm{O}(|\mathscr{A}|^{2c(n)} \cdot c(n)^2).\end{aligned}$$

The number of bits needed for storing the pointer at each block is

$$\begin{aligned}&\text{number of blocks} \cdot \log(\text{number of elements in } \mathscr{A}^{c(n)})\\ &= \frac{n}{c(n)} \cdot \log(|\mathscr{A}|^{c(n)}) = \mathrm{O}(n).\end{aligned}$$

For the last equality, recall that $\mathscr{A}$ is of constant size. For the query time, observe that after each iteration (consisting of Steps 1–3), either ℓ or r is advanced to the next block. As each of the pointers can advance at most $n/c(n)$ times, there can be at most $2n/c(n)$ iterations. Each iteration except the last one takes constant time. The last iteration may take time $\mathrm{O}(c(n)^2)$.

In total, the algorithm requires storage space $\mathrm{O}(n+|\mathscr{A}|^{2c(n)}c(n)^2)$ and time $\mathrm{O}(n/c(n)+c(n)^2)$. Now, if we choose $c(n)=(\log_{|\mathscr{A}|} m)/4$, then we obtain $|\mathscr{A}|^{c(n)}=n^{1/4}$. This yields a storage space of $\mathrm{O}(n + n^{1/2}\log^2 n) = \mathrm{O}(n)$ and query time $\mathrm{O}(n/\log n)$, which is both skinny and speedy. Other choices of $c(n)$ do not asymptotically improve time and space at the same time.

Theorem 1. *Algorithm* Lookup *solves the* One-String Mass Finding Problem *with storage space* $\mathrm{O}(n)$ *and query time* $\mathrm{O}(n/\log n)$.

Algorithm Lookup can be modified to work on real weights rather than on integers. Here, instead of storing the distances in hash tables, we can use sorted arrays. Each membership query to a hash table is replaced by binary search in the corresponding array. Since each array has size $\mathrm{O}(c(n)^2)$, this results in an additional factor $\mathrm{O}(\log c(n))$ in the query time. Again, with $c(n)$ chosen as above, this yields storage space $\mathrm{O}(n)$ and query time $\mathrm{O}((n/\log n)\log\log n)$.

Lookup beats both the query time of Linsearch and the storage space of Binsearch. However, its practical use is limited to very long sequences: In order to obtain a block size of, say, $c(n) = 10$, the input string would have to have length $n = |\mathscr{A}|^{40}$. In the next section we present a practical algorithm for binary alphabets.

4. A speedier algorithm for binary alphabets

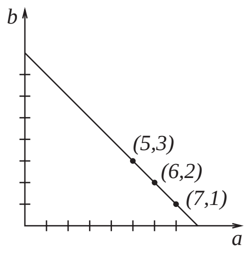
In this section, we present algorithm INTERVAL which solves the ONE-STRING MASS FINDING PROBLEM for an alphabet of size 2. It uses storage space $O(n)$ and has query time $O(\log n)$. The algorithm *decides* whether a given mass is a submass of σ, but does not return a witness.

Let σ be a string over $\mathscr{A} = \{a, b\}$ of length n and fix $k \leqslant n$. Observe that, when sliding a window of size k over σ, then in one step, the multiplicities of a and b within the window change at most by one. We represent substrings of σ by points in the $\mathbb{Z} \times \mathbb{Z}$ lattice, where the two coordinates signify the multiplicities of a and b:

$$S_k := \{(i,j) \in \mathbb{Z} \times \mathbb{Z} \mid i + j = k, \text{ there is a substring } \tau \text{ of } \sigma\text{: } |\tau|_a = i, |\tau|_b = j\}.$$

All points in S_k will lie on a line (a diagonal), and moreover, they will be neighbours. We will refer to such a set of neighbours on a line as an *interval*. Each such interval has two extremal points.

Example 2. $\sigma = aaaaabaabb$. The figure shows the representation of all substrings of length $k = 8$. Extremal points of this interval are $(5, 3)$ and $(7, 1)$.



Assume for a moment that we know the multiplicities of a and b in M, e.g., $M = i\mu(a) + j\mu(b)$. Then we can easily find out whether M is a submass of σ: We store the S_k's, for $1 \leqslant k \leqslant n$ by their extremal points during the preprocessing phase. Now we only have to check whether $(i,j) \in S_{i+j}$, which takes $O(1)$ time. This requires storage space linear in n. If, in addition, i and j were known to be the only feasible multiplicities of a and b (i.e., the unique solution of the equation $x\mu(a) + y\mu(b) = M$), then this algorithm would even decide whether M is a submass of σ, and we would be done.

Unfortunately, we do not know the multiplicities of a and b in M. We define $d := \mu(b) - \mu(a)$ (w.l.o.g., assume $\mu(a) < \mu(b)$) and use the residue of $M \bmod d$ to look up a table. The table, generated during the preprocessing phase, contains representations of all submasses of σ.

Let $M_k := \{\mu(\tau) \mid \tau \text{ is a } k\text{-length substring of } \sigma\}$. Observe that consecutive elements of M_k (when sorted) differ by exactly d. Therefore, we can write $M_k = \{c_k + \ell d \mid \ell = 0, \ldots, n_k - 1\}$, where $c_k = \min M_k$ and $n_k = |M_k|$. Furthermore, $M_k = \{r_k + \ell d \mid \ell = a_k, \ldots, b_k\}$, where $r_k = (c_k \bmod d)$, $a_k = \lfloor c_k/d \rfloor$ and $b_k = a_k + n_k - 1$. This says that all submasses of the same length have the same residue modulo d.

Example 2 cont'd. Let $\sigma = aaaaabaabb$ and $\mu(a) = 2$ and $\mu(b) = 7$. Then $d = 5$, and

$S_{10} = \{(7,3)\}$,	$M_{10} = \{35\}$,	$r_{10} = 0, a_{10} = 7, b_{10} = 7$
$S_9 = \{(7,2),(6,3)\}$,	$M_9 = \{28,33\}$,	$r_9 = 3, a_9 = 5, b_9 = 6$,
$S_8 = \{(7,1),(6,2),(5,3)\}$,	$M_8 = \{21,26,31\}$	$r_8 = 1, a_8 = 4, b_8 = 6$
$S_7 = \{(6,1),(5,2),(4,3)\}$	$M_7 = \{19,24,29\}$	$r_7 = 4, a_7 = 3, b_7 = 5$
$S_6 = \{(5,1),(4,2),(3,3)\}$	$M_6 = \{17,22,27\}$	$r_6 = 2, a_6 = 3, b_6 = 5$
$S_5 = \{(5,0),(4,1),(3,2),(2,3)\}$	$M_5 = \{10,15,20,25\}$	$r_5 = 0, a_5 = 2, b_5 = 5$
$S_4 = \{(4,0),(3,1),(2,2)\}$	$M_4 = \{8,13,18\}$	$r_4 = 3, a_4 = 1, b_4 = 3$
$S_3 = \{(3,0),(2,1),(1,2)\}$	$M_3 = \{6,11,16\}$	$r_3 = 1, a_3 = 1, b_3 = 3$
$S_2 = \{(2,0),(1,1),(0,2)\}$	$M_2 = \{4,9,14\}$	$r_2 = 4, a_2 = 0, b_2 = 2$
$S_1 = \{(1,0),(0,1)\}$	$M_1 = \{2,7\}$	$r_1 = 2, a_1 = 0, b_1 = 1$

Observe that $r_k = (k\mu(a) \bmod d)$. Thus, we may have the same residue modulo d for different values of k. Instead of storing a_k and b_k for each r_k individually (which could result in linear query time), we will store the union of all intervals which belong to the same residue r, sorted by their endpoints.

Example 2 cont'd. In the example, this yields the following preprocessed data. For residues 1 and 4, the intervals have been merged.

Residue modulo d	Union of intervals
0	$[2,5],[7,7]$
1	$[1,6]$
2	$[0,1],[3,5]$
3	$[1,3],[5,6]$
4	$[0,5]$

4.1. *Algorithm* Interval

In the preprocessing phase, we calculate the r_k's, a_k's, and b_k's as above. We then sort the r_k's, thus obtaining a sorted array $q_1, \ldots, q_m$, where $m \leqslant n$ (since different S_k's may have the same residue). For each q_l, we compute a list of interval endpoints which represents the union of all intervals $[a_k, b_k]$ with $r_k = q_l$. This list consists of one or more disjoint intervals, which we store in sorted order in an array A_l.

Now, when querying whether a given mass M is contained in σ:

1. decompose $M = gd + r$, where $r = (M \bmod d)$ and $g \in \mathbb{N}$;
2. find index $l \in \{1, \ldots, m\}$ such that $r = q_l$, using binary search; if no such index can be found, then M is not a submass of σ, and the algorithm outputs **no**;
3. otherwise, find whether there is an interval $[a,b]$ in array A_l such that $g \in [a,b]$, using binary search on (the left endpoints of) the intervals; M is a submass of σ if and only if such an interval exists.

Since the total number of intervals to be stored is n, the storage space needed is $O(n)$. The first step of the query algorithm takes time $O(1)$. The second step takes time $O(\log n)$, since the number of different residues is at most n. The third step takes time $O(\log n)$, since the maximum number of intervals stored in one array A_l is n. We obtain a total query time $O(\log n)$.

Theorem 2. *Algorithm* INTERVAL *solves the* ONE-STRING MASS FINDING PROBLEM *for binary alphabets with storage space* $O(n)$ *and query time* $O(\log n)$.

The problem in generalizing this approach to larger alphabets is that the algorithm relies on the crucial fact that points representing substrings of the same length lie on a line and form an interval. This does not generalize to higher dimensions, since there we only know that the points representing substrings of the same length are connected.

5. Problem variants

The MULTIPLE-STRING MASS FINDING PROBLEM is defined as follows:

Given k strings $\sigma_1, \ldots, \sigma_k$ and a mass $M \in \mathbb{N}$, return a list $i_1, \ldots, i_r$ of those strings σ_{i_j} which have M as a submass.

An algorithm Ψ for the ONE-STRING MASS FINDING PROBLEM can be extended to an algorithm for the MULTIPLE-STRING MASS FINDING PROBLEM by running Ψ on each string σ_i one by one. Required storage space and query time simply sum up.

Alternatively, we can adapt an approach from Group Testing (cf. [7]): We define a new string $\sigma := \sigma_1 \omega \sigma_2 \omega \ldots \omega \sigma_k$, where ω is a new letter with mass $\mu(\omega) := \max\{\mu(\sigma_i) \mid 1 \leqslant i \leqslant k\} + 1$. Before applying Ψ to σ, we check whether $M \geqslant \mu(\omega)$. If so, then M cannot be a submass of any of the strings, and we are done. Otherwise, we know that whenever Ψ finds mass M in σ, then it is a submass of σ_i for some index i. If algorithm Ψ can output *all* positions of M in σ, this solves the MULTIPLE-STRING MASS FINDING PROBLEM. If Ψ only *decides* whether M is a submass of σ (i.e., it outputs only **yes** or **no**), we use a kind of "binary tree search" BINTREESEARCH to find all σ_i with submass M as follows. First, we run Ψ on σ as described above. If it outputs **no**, then no string σ_i has submass M, and we are done. Otherwise, we divide σ into two new strings $\sigma_l := \sigma_1 \omega \ldots \omega \sigma_{\lfloor k/2 \rfloor}$ and $\sigma_r := \sigma_{\lfloor k/2 \rfloor + 1} \omega \ldots \omega \sigma_k$ and run Ψ on both strings separately. We repeat the division step until the new strings cover exactly one σ_i, in which case the answer of Ψ determines whether σ_i has a submass M. Analysis of BINTREESEARCH depends heavily on storage space and query time required by Ψ. For instance, if algorithm Ψ requires storage space linear in the length of the string, then the storage space of BINTREESEARCH is $O((\log k) \sum_{i=1}^{k} |\sigma_i|)$. Query time of BINTREESEARCH depends on the number of strings with submass M, in contrast to the simple idea of applying Ψ to each string separately.

Given a specific algorithm for the ONE-STRING MASS FINDING PROBLEM, there might be even better ways to extend it to the MULTIPLE-STRING MASS FINDING PROBLEM: E.g.

for Binsearch, we can use *one* sorted array to store all submasses of all strings. For each submass x we store the set of indices I_x of all those strings which have a submass x. Given mass M, we perform binary search in the array and output all indices stored in I_M. Required storage space remains unchanged, but the running time becomes $\mathrm{O}(\log(\sum_{i=1}^{k}|\sigma_i|)+|I_M|)$, where $|I_M|\leqslant k$ is the size of the output. A similar idea applies to Lookup, where we could store only one table T of size $|\mathscr{A}|^{c_{\max}}\times|\mathscr{A}|^{c_{\max}}$ where $c_{\max}=\max_{i=1}^{k} c(|\sigma_i|)$, and use it for all runs of the algorithm. However, this does not decrease the asymptotic space required, which still remains linear.

We define a third problem variant, the Multiple-String Multiple-Mass Finding Problem:

> Given k strings $\sigma_1,\ldots,\sigma_k$, m masses $M_1,\ldots M_m\in\mathbb{N}$, and a threshold $1\leqslant t\leqslant m$, return a list $i_1,\ldots,i_r$ of those strings σ_{i_j} which have at least t of the masses as submasses.

In the setting of our application in computational biology, this will be a more realistic formulation, since typically, one breaks a given protein in several pieces and wants to find the protein in the database which contains all (or at least many) of these pieces. Obviously, the Multiple-String Multiple-Mass Finding Problem can be solved by applying algorithms for the Multiple-String Mass Finding Problem m times. We are investigating the question whether concurrently searching for m masses can be performed more efficiently.

Finally, we present an improvement of all our algorithms for “short masses”: Let the *length* of a mass M be defined as $\lambda(M):=\max(\{|\tau|\mid\tau\in\mathscr{A}^*,\mu(\tau)=M\}\cup\{-1\})$. Here, $\lambda(M)=-1$ means that there is no string with mass M. Suppose that we know in advance that all query masses are short in comparison to n, i.e., that there is a function $f(n)$ such that $\lambda(M)\leqslant f(n)=\mathrm{o}(n)$ for all queries M. Then there is a simple algorithm to solve the One-String Mass Finding Problem, which is a variant of Binsearch: In the preprocessing, we store all submasses of σ of length $\ell\leqslant f(n)$ in a sorted array. This requires storage space $\mathrm{O}(nf(n))$, since for each position i in σ, at most $f(n)$ substrings of length $\ell\leqslant f(n)$ start in i. For a query, we do binary search in this array. This takes time $\mathrm{O}(\log n)$, which is speedy. Since $f(n)=\mathrm{o}(n)$, the algorithm is skinny, too. We can use this approach to improve our algorithms in the sense that they will run faster on short masses.

6. Weighted strings

A question closely related to the analysis of algorithms for the One-String Mass Finding Problem is the following: Given a string σ of length n, how many different submasses does σ have? For example, the storage space required by Binsearch is proportional to this number. Let $\mathscr{A}=\{a_1,\ldots,a_s\}$. Given a string σ, let us define three combinatorial functions on σ. Recall that we denote the multiplicity vector of a string σ by $\mathrm{mult}(\sigma)$ (also referred to as *Parikh-vector*, see [2]).

1. $\mathbf{S}(\sigma):=|\{\tau\mid\tau\sqsubseteq\sigma\}|$, the number of different substrings of σ,

2. $\mathbf{P}(\sigma) := |\{\text{mult}(\tau) \mid \tau \sqsubseteq \sigma\}|$, the number of different multiplicity vectors of substrings of σ, and
3. $\mathbf{M}(\sigma) := |\{\mu(\tau) \mid \tau \sqsubseteq \sigma\}|$, the number of different submasses of σ.

Note again that these definitions exclude the empty string. An obvious relation between these three functions is

$$n \leqslant \mathbf{M}(\sigma) \leqslant \mathbf{P}(\sigma) \leqslant \mathbf{S}(\sigma) \leqslant \frac{n(n+1)}{2}.$$

For $n \in \mathbb{N}$, define strings σ_n and τ_n:

$$\sigma_n := a_1^{m_1} a_2^{m_2} \dots a_s^{m_s},$$

$$\tau_n := (a_1 \dots a_s)^k a_1 \dots a_r,$$

where $\sum_{i=1}^{s} m_i = n$ s.t. for all $i = 1, \dots, s$, $m_i = \lfloor n/s \rfloor$ or $m_i = \lfloor n/s \rfloor + 1$, i.e., all m_i are approximately equal, and $k = \lfloor n/s \rfloor, r = n \bmod s$. In particular, if n is a multiple of s, then $m_i = k$ for all $i = 1, \dots, s$, and $n = ks$. Then

$$\sigma_n = a_1^k a_2^k \dots a_s^k,$$

$$\tau_n = (a_1 \dots a_s)^k.$$

The following theorems state that the strings σ_n and τ_n maximize resp. minimize these three functions up to a factor of 2.

Theorem 3 (Asymptotically maximal strings w.r.t. $\mathbf{S}, \mathbf{P}, \mathbf{M}$). *Let $n \in \mathbb{N}$ and $n_i \in \mathbb{N}$ for $i = 1, \dots, s$ such that $\sum_{i=1}^{s} n_i = n$.*

1. *σ_n has quadratic values $\mathbf{S}, \mathbf{P}, \mathbf{M}$: $\mathbf{S}(\sigma_n), \mathbf{P}(\sigma_n) = \Theta(n^2)$, and for certain classes of weight functions, $\mathbf{M}(\sigma_n) = \Theta(n^2)$.*
2. *σ_n maximizes $\mathbf{S}$ up to a factor of 2 for $n = ks$ and equal multiplicities:* $\mathbf{S}(\sigma_n) \geqslant \frac{1}{2}\max\{\mathbf{S}(\sigma) \mid \forall i = 1, \dots, s : |\sigma|_{a_i} = k\}$.
3. *σ_n maximizes $\mathbf{P}$: $\mathbf{P}(\sigma_n) = \max\{\mathbf{P}(\sigma) \mid |\sigma| = n\}$. Moreover, the string $a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}$ maximizes $\mathbf{P}$ for fixed multiplicities n_i: $\mathbf{P}(a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}) = \max\{\mathbf{P}(\sigma) \mid \forall i = 1, \dots, s : |\sigma|_{a_i} = n_i\}$.*
4. *There are classes of weight functions s.t. σ_n maximizes $\mathbf{M}$: $\mathbf{M}(\sigma_n) = \max\{\mathbf{M}(\sigma) \mid |\sigma| = n\}$. Moreover, the string $a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}$ maximizes $\mathbf{M}$ for fixed multiplicities n_i: $\mathbf{M}(a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}) = \max\{\mathbf{M}(\sigma) \mid \forall i = 1, \dots, s : |\sigma|_{a_i} = n_i\}$.*

Theorem 4 (Asymptotically minimal strings w.r.t. $\mathbf{S}, \mathbf{P}, \mathbf{M}$). *Let $n \in \mathbb{N}$.*

1. *τ_n has linear values $\mathbf{S}, \mathbf{P}, \mathbf{M} : \mathbf{S}(\tau_n), \mathbf{P}(\tau_n), \mathbf{M}(\tau_n) = \Theta(n)$.*
2. *τ_n minimizes $\mathbf{S}$ up to a factor of 2 for $n = ks$ and equal multiplicities: $\mathbf{S}(\tau_n) \leqslant 2\min\{\mathbf{S}(\sigma) \mid \forall i = 1, \dots, s : |\sigma|_{a_i} = k\}$.*
3. *τ_n minimizes $\mathbf{P}$ up to a factor of 2 for $n = ks$ and equal multiplicities: $\mathbf{P}(\tau_n) \leqslant 2\min\{\mathbf{P}(\sigma) \mid \forall i = 1, \dots, s : |\sigma|_{a_i} = k\}$.*

4. *There are classes of weight functions s.t.* τ_n *minimizes* $\mathbf{M}$ *up to a factor of* 2 *for* $n = ks$ *and equal multiplicities*: $\mathbf{M}(\tau_n) \leqslant 2\min\{\mathbf{M}(\sigma) \mid \forall i = 1,\dots,s : |\sigma|_{a_i} = k\}$.

We devote the rest of this section to the proof of these theorems.

6.1. Number of different substrings

Obviously, $n \leqslant \mathbf{S}(\sigma) \leqslant n(n+1)/2$. The lower bound is attained exactly by the strings a^n for arbitrary $a \in \mathscr{A}$. The upper bound can only be attained if all letters are different, i.e., $\mathbf{S}(\sigma) = n(n+1)/2$ implies $n \leqslant s$. The number $\mathbf{S}(\sigma)$ can be computed using a suffix tree: In a suffix tree, each substring is represented by a unique path from the root. Thus, adding up the label lengths of the edges of the suffix tree of σ will yield just $\mathbf{S}(\sigma)$. The suffix tree of σ can be computed in time $O(n)$, see e.g. [13]. The number of edges is linear in n, thus $\mathbf{S}(\sigma)$ can also be computed in linear time. Moreover, we can enumerate all substrings of σ in time $O(\mathbf{S}(\sigma))$, if we only output tuple (i,j) for substring $\tau = \sigma(i,j)$. If we output the sequence of letters of τ instead, we obtain $O(\sum_{\tau \sqsubseteq \sigma} |\tau|)$.

Lemma 5 (Linear and quadratic examples for $\mathbf{S}$). *For all* $n \in \mathbb{N}$, *there exist strings* $\sigma, \tau \in \mathscr{A}^n$ *s.t.* $\mathbf{S}(\sigma) = \Theta(n)$ *and* $\mathbf{S}(\tau) = \Theta(n^2)$ *and* $|\sigma|_a, |\tau|_a \geqslant 1$ *for all* $a \in \mathscr{A}$. *In particular, for* $k, r, n_1, \dots, n_s \in \mathbb{N}$ *s.t.* $r < s$ *and* $\sum_{i=1}^{s} n_i = n$,

1. (a) $\mathbf{S}((a_1 \dots a_s)^k) = (k-1)s^2 + \frac{1}{2}(s^2+s)$,
   (b) $\mathbf{S}((a_1 \dots a_s)^k a_1 \dots a_r) = (k-1)s^2 + \frac{1}{2}(s^2+s) + rs$,
2. $\mathbf{S}(a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}) = n + \sum_{1 \leqslant i < j \leqslant s} n_i n_j$.

Proof. 1. (a) For fixed length $m \leqslant (k-1)s$, there are s different substrings of length m, namely $\sigma(i, i+m)$ for $i = 1,\dots,s$. There are s substrings of length $(k-1)s+1$, $s-1$ substrings of length $(k-1)s+2$, and so on, and finally, exactly one substring of length $ks = n$. Thus, $\mathbf{S}((a_1 \dots a_s)^k) = (k-1)s \cdot s + \sum_{i=1}^{s} i$.

1. (b) In addition to substrings of $(a_1 \dots a_s)^k$, each of the final r positions of σ contributes s different new substrings, namely those beginning within the first block $a_1 \dots a_s$ and ending in this position.

2. First consider substrings that start and end with the same letter a_i. For fixed $1 \leqslant i \leqslant s$, there are n_i different substrings of this type, yielding $\sum_{i=1}^{s} n_i = n$ different substrings. All other substrings start with some letter a_i and end with a different letter a_j, where $i < j$. For each pair i, j, there are $n_i n_j$ different choices of the first and final positions, which all generate different substrings. Thus, $\mathbf{S}(a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}) = n + \sum_{1 \leqslant i < j \leqslant s} n_i n_j$.

Since the alphabet size s is constant and $k = \lfloor n/s \rfloor$, we have $\mathbf{S}((a_1 \dots a_s)^k a_1 \dots a_r) = \Theta(n)$. On the other hand, if for all $i = 1,\dots,s$, $n_i = \lfloor n/s \rfloor$ or $n_i = \lfloor n/s \rfloor + 1$ s.t. $\sum_{i=1}^{s} n_i = n$, i.e., all n_i are roughly equal, this yields $\mathbf{S}(a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}) = \sum_{i=1}^{s} n_i + \sum_{1 \leqslant i < j \leqslant s} n_i n_j \approx n + \binom{s}{2} (\frac{n}{s})^2 \approx n + \frac{1}{2} n^2 = \Theta(n^2)$. □

For $n = ks$, the string $\sigma_n = a_1^k a_2^k \dots a_s^k$ is thus maximal up to a factor of 2 w.r.t. $\mathbf{S}$, since $\mathbf{S}(\sigma_n) = n + ((s-1)/(2s))n^2 \geqslant \frac{1}{2} n(n+1)/2$. In the next section, we will prove a lower bound on $\mathbf{P}(\sigma)$ (Lemma 6), which depends on the multiplicities $|\sigma|_{a_i}$ of the different letters. For equal multiplicities $|\sigma|_{a_i} = k = n/s$ for all i, this lower bound is $(k/2)(s^2 + s)$, implying $\mathbf{S}(\sigma) \geqslant \mathbf{P}(\sigma) \geqslant (k/2)(s^2 + s)$. Since $\mathbf{S}((a_1 \ldots a_s)^k) = (k-1)s^2 + \frac{1}{2}(s^2 + s) \leqslant 2(k/2)(s^2 + s)$, the string $\tau_n = (a_1 \ldots a_s)^k$ is minimal up to a factor of 2 w.r.t. $\mathbf{S}$.

6.2. Number of different multiplicities

The question of the value of $\mathbf{P}(\sigma)$ for a given string σ is equivalent to the question of how many different multiplicity vectors mult(τ) the set $L_\sigma := \{\tau \mid \tau \sqsubseteq \sigma\}$ has. If we denote by $\mathscr{A}^\oplus$ the free commutative monoid over $\mathscr{A}$, then any language $L \subseteq \mathscr{A}^*$ induces a subset $L^\oplus$ of $\mathscr{A}^\oplus$, namely $L^\oplus := \{\prod_{a\in\mathscr{A}} a^{|\tau|_a} \mid \tau \in L\}$ (see [2]). Now, we have $\mathbf{P}(\sigma) = |L_\sigma^\oplus|$. We are not aware that $|L_\sigma^\oplus|$ has been characterized in the literature.

We can compute $\mathbf{P}(\sigma)$ trivially by enumerating all substrings of σ, computing their multiplicity vectors, and ordering them. This can be done in time $O(\mathbf{S}(\sigma)\log(\mathbf{S}(\sigma)))$.

For a lower bound on $\mathbf{P}$, we define the index of the first occurrence of a letter $a \in \mathscr{A}$ in a string σ as $\mathrm{First}_a(\sigma) := \min(\{i \mid \sigma(i) = a\} \cup \{|\sigma| + 1\})$.

Lemma 6 (Lower bound on $\mathbf{P}$). *Let $n \in \mathbb{N}$ and $\sigma \in \mathscr{A}^n$. Then $\mathbf{P}(\sigma) \geqslant \sum_{a\in\mathscr{A}} |\sigma|_a \cdot \mathrm{First}_a(\sigma)$. In particular, if $|\sigma|_a = k = n/s$ for all $a \in \mathscr{A}$, then $\mathbf{P}(\sigma) \geqslant (k/2)(s^2 + s)$.*

Proof. Let $x = \sigma(n)$. If x does not occur in $\sigma(1, n-1)$, then appending x to $\sigma(1, n-1)$ generates n new multiplicities, i.e., $\mathbf{P}(\sigma) = \mathbf{P}(\sigma(1, n-1)) + n = \mathbf{P}(\sigma(1, n-1)) + \mathrm{First}_x(\sigma(1, n-1))$. On the other hand, if x does occur in $\sigma(1, n-1)$, then it generates at least $\mathrm{First}_x(\sigma(1, n-1))$ new multiplicities, since those substrings starting in positions $i = 1, \ldots, \mathrm{First}_x(\sigma(1, n-1))$ and ending in $\sigma(n) = x$ will have $|\sigma(i, n)|_x = |\sigma(1, n-1)|_x + 1$. Thus, in both cases we obtain $\mathbf{P}(\sigma) \geqslant \mathbf{P}(\sigma(1, n-1)) + \mathrm{First}_x(\sigma(1, n-1))$. Applying this $n-1$ times, we obtain

$$\mathbf{P}(\sigma) \geqslant 1 + \sum_{i=2}^{n} \mathrm{First}_{\sigma(i)}(\sigma(1, i-1)).$$

Let $i \in \{2, \ldots, n\}$. If $\sigma(i)$ occurs in $\sigma(1, i-1)$, then $\mathrm{First}_{\sigma(i)}(\sigma(1, i-1)) = \mathrm{First}_{\sigma(i)}(\sigma)$. In the sum above, this happens $|\sigma|_{\sigma(i)} - 1$ times. For the first occurrence of letter $\sigma(i)$ in σ, $\mathrm{First}_{\sigma(i)}(\sigma(1, \mathrm{First}_{\sigma(i)}(\sigma) - 1)) = \mathrm{First}_{\sigma(i)}(\sigma)$ by definition. Since $\mathrm{First}_{\sigma(1)}(\sigma) = 1$, we can write $1 + \sum_{i=2}^{n} \mathrm{First}_{\sigma(i)}(\sigma(1, i-1)) = \sum_{a\in\mathscr{A}} \mathrm{First}_a(\sigma)|\sigma|_a$.

For fixed multiplicities $n_1, \ldots, n_s$, the sum $\sum_{a\in\mathscr{A}} \mathrm{First}_a(\sigma)|\sigma|_a$ is minimized over all strings with these multiplicities if all different letters occurring in σ are positioned in the first positions of σ, ordered ascending according to their multiplicities. In particular, if each letter occurs exactly k times, we obtain $\mathbf{P}(\sigma) \geqslant k \sum_{i=1}^{s} i = (k/2)(s^2 + s)$.

Lemma 7 (Linear and quadratic examples for $\mathbf{P}$). *For all $n \in \mathbb{N}$, there exist strings $\sigma, \tau \in \mathscr{A}^n$ s.t. $\mathbf{P}(\sigma) = \Theta(n)$ and $\mathbf{P}(\tau) = \Theta(n^2)$ and $|\sigma|_a, |\tau|_a \geqslant 1$ for all $a \in \mathscr{A}$. In particular, for $k, r, n_1, \ldots, n_s \in \mathbb{N}$ s.t. $r < s$ and $\sum_{i=1}^{s} n_i = n$,*

1. (a) $\mathbf{P}((a_1 \ldots a_s)^k) = (k-1)(s^2 + 1 - s) + \frac{1}{2}(s^2 + s)$,
   (b) $\mathbf{P}((a_1 \ldots a_s)^k a_1 \ldots a_r) = (k-1)(s^2 + 1 - s) + \frac{1}{2}(s^2 + s) + r(s-1)$,
2. $\mathbf{P}(a_1^{n_1} a_2^{n_2} \ldots a_s^{n_s}) = n + \sum_{1\leqslant i<j\leqslant s} n_i n_j$.

Proof. 1. (a) Let $\sigma := (a_1 \dots a_s)^k$. First consider only substrings with length $m \leqslant (k-1)s$ and observe that for $m = \ell s$, there is exactly one multiplicity vector $(\ell, \dots, \ell)$ for all substrings of length m. Otherwise, if $m = \ell s + p$ where $0 < p < s$, then for each $1 \leqslant j \leqslant s$, there is a substring τ with multiplicities

$$|\tau|_{a_i} = \begin{cases} \ell + 1 & \text{if there is } q \in \{0, \dots, p-1\} \text{ s.t. } i = (j+q) \bmod s, \\ \ell & \text{otherwise.} \end{cases}$$

Putting this together yields $(k-1)(1+(s-1)s)$. Finally, for lengths $m > (k-1)s$, the numbers of different multiplicity vectors decrease one by one: There are s different multiplicity vectors of substrings with length $(k-1)s+1$, $s-1$ with length $(k-1)s+2$ and so on, yielding $\sum_{i=s}^{1} i = \frac{1}{2}(s^2+s)$ different multiplicity vectors. Thus, $\mathbf{P}((a_1 \dots a_s)^k) = (k-1)(s^2+1-s) + \frac{1}{2}(s^2+s)$.

1. (b) This is a simple extension of 1(a), noting that each of the last r positions of σ will contribute $s-1$ new multiplicity vectors for substrings ending in this position: For a_j, $1 \leqslant j \leqslant r$, the multiplicity vector of the substring $\sigma(i, ks+j)$ will be new for all $i \in \{1, \dots, j, j+2, \dots, s\}$.

2. Observe that for string $\tau = a_1^{n_1} \dots a_s^{n_s}$, $\mathbf{P}(\tau) = \mathbf{S}(\tau)$, and thus, $\mathbf{P}(\tau) = n + \sum_{1 \leqslant i < j \leqslant s} n_i n_j$ by Lemma 5.

Similar to the proof of Lemma 5, we have $\mathbf{P}((a_1 \dots a_s)^k) = \Theta(n)$ for constant s and $\mathbf{P}(a_1^{n_1} a_2^{n_2} \dots a_s^{n_s}) = \Theta(n^2)$ for roughly equal multiplicities n_i. $\square$

For equal multiplicities $|\sigma|_{a_i} = k = n/s$ for all i, the lower bound on $\mathbf{P}$ is $(k/2)(s^2+s)$. Since $\mathbf{P}((a_1 \dots a_s)^k) = (k-1)(s^2+1-s) + \frac{1}{2}(s^2+s) \leqslant 2\frac{k}{2}(s^2+s)$, the string $\tau_n = (a_1 \dots a_s)^k$ is minimal up to a factor of 2 w.r.t. $\mathbf{P}$.

The next two lemmas are used to prove a tight upper bound on $\mathbf{P}$ (Lemma 10). Hereby, we denote by $[\Lambda]$ the characteristic value of a proposition Λ, i.e., $[\Lambda] = 1$ if Λ is true, and $[\Lambda] = 0$ otherwise.

Lemma 8 (Maximal growth of $\mathbf{P}$). *Let $n \in \mathbb{N}, x \in \mathscr{A}$ and $\sigma \in \mathscr{A}^n$.*

1. *If σ does not contain letter x, then $\mathbf{P}(\sigma x) = \mathbf{P}(\sigma) + (n+1)$.*
2. *If σ contains letter x, then $\mathbf{P}(\sigma x) \leqslant \mathbf{P}(\sigma) + n - |\sigma|_x + [\sigma(n) = x]$.*

Proof. 1. Obvious.

2. There are $\mathbf{P}(\sigma)$ different multiplicity vectors of substrings starting and ending within σ. Furthermore, n substrings of σx start within σ and end in x. For each index $1 \leqslant i \leqslant n-1$ s.t. $\sigma(i) = x$, we have $\text{mult}(\sigma(i,n)) = \text{mult}(\sigma(i+1,n)x)$. Thus, none of these substrings has a new multiplicity vector. There are $|\sigma|_x$ such substrings if $\sigma(n) \neq x$, and $|\sigma|_x - 1$ otherwise. $\square$

The next lemma shows that concentrating each letter in blocks maximizes the number of multiplicity vectors.

Lemma 9 (**P**-maximal strings). *Let $n \in \mathbb{N}$ and fix $0 \leqslant n_1, \dots, n_s \in \mathbb{N}$ s.t. $\sum_{i=1}^{s} n_i = n$. Then,*

$$\mathbf{P}(a_1^{n_1} \dots a_s^{n_s}) = \max\{\mathbf{P}(\sigma) \mid \forall i = 1, \dots, s : |\sigma|_{a_i} = n_i\}.$$

Proof. By induction on n: For $n = 1$, the claim is obvious. Choose $\sigma \in \mathscr{A}^{n+1}$ and denote by $n_i := |\sigma|_{a_i}$ for $i = 1, \dots, s$. Up to relabeling (which leaves **P** invariant), we may assume that the last letter of σ is a_s, thus we can write $\sigma = \sigma' a_s$. If $n_s = 1$, then

$$\begin{aligned} \mathbf{P}(\sigma) &= \mathbf{P}(\sigma') + (n+1) && \text{by Lemma 8,} \\ &\leqslant \mathbf{P}(a_1^{n_1} \dots a_{s-1}^{n_{s-1}}) + (n+1) && \text{by the induction hypothesis,} \\ &= \mathbf{P}(a_1^{n_1} \dots a_s^{n_s}) && \text{by Lemma 7,} \end{aligned}$$

otherwise, $n_s > 1$, and

$$\begin{aligned} \mathbf{P}(\sigma) &\leqslant \mathbf{P}(\sigma') + n - |\sigma'|_{a_s} + 1 && \text{by Lemma 8,} \\ &\leqslant \mathbf{P}(a_1^{n_1} \dots a_s^{n_s - 1}) + n - (n_s - 1) + 1 && \text{by the induction hypothesis} \\ &= n + \sum_{1 \leqslant i < j < s} n_i n_j + \sum_{i=1}^{s-1} n_i(n_s - 1) + n - (n_s - 1) + 1 && \text{by Lemma 7,} \\ &= n + \sum_{1 \leqslant i < j < s} n_i n_j + \sum_{i=1}^{s-1} n_i(n_s - 1) + \sum_{i=1}^{s-1} n_i + 1 \\ &= (n+1) + \sum_{1 \leqslant i < j \leqslant s} n_i n_j = \mathbf{P}(a_1^{n_1} \dots a_s^{n_s}). \end{aligned}$$ □

Lemma 10 (Tight upper bound on **P**). *Let $\sigma \in \mathscr{A}^n$. Then $\mathbf{P}(\sigma) \leqslant n + \sum_{1 \leqslant i < j \leqslant s} m_i m_j$, where $m_i = \lfloor n/s \rfloor$ or $m_i = \lfloor n/s \rfloor + 1$ for $i = 1, \dots, s$ and $\sum_{i=1}^{s} m_i = n$. In particular, if n is a multiple of s, then $\mathbf{P}(\sigma) \leqslant ((s-1)/(2s))n^2 + n$. This bound is tight.*

Proof. Let $\sigma \in \mathscr{A}^n$. Denote by $n_i := |\sigma|_{a_i}$ for $i = 1, \dots, s$. Then, by Lemma 9, $\mathbf{P}(\sigma) \leqslant \mathbf{P}(a_1^{n_1} \dots a_s^{n_s}) = n + \sum_{1 \leqslant i < j \leqslant s} n_i n_j$. Let $f(x_1, \dots, x_s) := \sum_{1 \leqslant i < j \leqslant s} x_i x_j$. Function f attains its maximum on the set $B_n := \{(x_1, \dots, x_s) \mid \sum_{i=1}^{s} x_i = n\}$ if all values are approximately equal, i.e., $\max\{f(B_n)\} = f(m_1, \dots, m_s)$ where for all i, $m_i = \lfloor n/s \rfloor$ or $m_i = \lfloor n/s \rfloor + 1$ and $\sum_{i=1}^{s} m_i = n$. Moreover, since $\mathbf{P}(a_1^{m_1} \dots a_s^{m_s}) = n + \sum_{1 \leqslant i < j \leqslant s} m_i m_j$, this bound is tight. If n is a multiple of s, then $m_i = n/s$ for all i, and thus:

$$\max\{\mathbf{P}(\sigma) \mid |\sigma| = n\} = n + \binom{s}{2} \left(\frac{n}{s}\right)^2 = \frac{s-1}{2s} n^2 + n. \qquad \square$$

6.3. Number of different submasses

A trivial tight lower bound for $\mathbf{M}(\sigma)$ is n; strings of the form a^n for any $a \in \mathscr{A}$ have exactly n different submasses, since $\mu(a) > 0$. If we have $\mathbf{M}(\sigma) = \mathbf{P}(\sigma)$, we also have a tight upper bound and can specify strings with a quadratic number of submasses. We therefore define the Unique Decomposition Property:

A mass function μ has the Unique Decomposition Property (UDP) if, for all strings σ and τ:

$$\mu(\sigma) = \mu(\tau) \quad \Leftrightarrow \quad \forall a \in \mathscr{A}\colon\ |\sigma|_a = |\tau|_a.$$

This just means that the masses are linearly independent over the integers. With the Udp, a mass M has at most one decomposition $M = \sum_{a\in\mathscr{A}} v(a)\mu(a)$ where $v(a) \in \mathbb{N}$. In this context, the question naturally arises whether a given mass M can be the weight of a string. If the size of the alphabet is variable, then this question is a variant of the Integer Knapsack Problem, and is NP-complete (cf. [11]). If the alphabet size is constant, the question can be solved with a simple integer linear program.

With the Udp, we have $\mathbf{M}(\sigma) = \mathbf{P}(\sigma)$ for all σ. Note that this condition never holds if the masses are integers or rational numbers. If, however, we allow real numbers as masses, i.e., if $\mu : \mathscr{A} \to \mathbb{R}^+$, then the masses can be chosen to satisfy the Udp. For example, if $\mathscr{A} = \{a, b\}$, then μ with $\mu(a) = 1$ and $\mu(b) = \pi$ has the Udp.

We can even achieve that $\mathbf{M}(\sigma) = \mathbf{P}(\sigma)$ with integers if the masses may depend on the input size. Then, we can set $\mu(a_i) := (n+1)^{i-1}$ for all $i = 1, \ldots, s$. However, this results in exponentially large masses.

If $\mathbf{M}(\sigma) = \mathbf{P}(\sigma)$ holds for all strings σ, then all results from the previous section on $\mathbf{P}$ carry over to $\mathbf{M}$.

7. Conclusion

With Lookup, we presented an algorithm for the One-String Mass Finding Problem that is both skinny and speedy. This proves that it is asymptotically possible to beat both Linsearch and Binsearch at the same time. This raises the question whether there are more practical algorithms that are skinny and speedy. In the long run, we are interested in the tradeoff between query time and storage space for the One-String Mass Finding Problem. Do algorithms exist that can be parametrized to allow for adjustment of this tradeoff?

Acknowledgements

We would like to thank Juraj Hromkovic who read an earlier version of this paper and made many helpful suggestions. We also thank Volker Diekert for the pointer to Parikh-vectors and Sacha Baginsky for advice on molecular biology. Finally, we would like to thank Benno Schwikowski and Jimmy K. Eng for calling our attention to the problems discussed in this paper.

References

[1] A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Springer, Berlin, 1995.

[2] J.-M. Autebert, J. Berstel, L. Boasson, Context-free languages and pushdown automata, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol. 1, Springer, Berlin, 1997, pp. 111–174 (Chapter 3).
[3] V. Bafna, N. Edwards, SCOPE: A probabilistic model for scoring tandem mass spectra against a peptide database, Bioinformatics 17 (Suppl. 1) (2001) S13–S21.
[4] P.J. Bentley, Programming Pearls, Wesley, Reading, MA, 1986.
[5] M. Cosnard, J. Duprat, A.G. Ferreira, The complexity of searching in $X+Y$ and other multisets, Inform. Process. Lett. 34 (1990) 103–109.
[6] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, New York, NY, 1994.
[7] D.-Z. Du, F.K. Hwang, (Eds.), Combinatorial Group Testing and its Applications, 2nd edition, World Scientific, Singapore, 2000.
[8] J. Eng, A. McCormack, J.R. Yates III, An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database, J. Amer. Soc. Mass Spectrom. 5 (1994) 976–989.
[9] M.L. Fredman, Two applications of a probabilistic search technique: sorting $X+Y$ and building balanced search trees, in: Conference Record of Seventh Annual ACM Symposium on Theory of Computing (STOC), 1975, pp. 240–244.
[10] M.L. Fredman, J. Komlós, E. Szemerédi, Storing a sparse table with O(1) worst case access time, J. ACM 31 (3) (1984) 538–544.
[11] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.
[12] L. Gonick, M. Wheelis, The Cartoon Guide to Genetics, HarperPerennial (updated Edition), 1991.
[13] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, Cambridge, UK, 1997.
[14] L.H. Harper, T.H. Payne, J.E. Savage, E. Straus, Sorting $X+Y$, Comm. ACM 18 (6) (1975) 347–349.
[15] W.J. Henzel, T.M. Billeci, J.T. Stults, S.C. Wong, C. Grimley, C. Watanabe, Identifying proteins from two-dimensional gels by molecular mass searching of peptide fragments in protein sequence databases, Proc. Natl. Acad. Sci. USA 90 (11) (1993) 5011–5015.
[16] P. James, M. Quadroni, E. Carafoli, G. Gonnet, Protein identification by mass profile fingerprinting, Biochem. Biophys. Res. Comm. 195 (1) (1993) 58–64.
[17] M. Lothaire, Combinatorics on Words, 2nd Edition, Cambridge University Press, Cambridge, UK, 1997.
[18] M. Mann, P. Højrup, P. Roepstorff, Use of mass spectrometric molecular weight information to identify proteins in sequence databases, Biol. Mass Spectrom. 22 (6) (1993) 338–345.
[19] M. Mann, M. Wilm, Error-tolerant identification of peptides in sequence databases by peptide sequence tags, Anal. Chem. 66 (24) (1994) 4390–4399.
[20] D.J.C. Pappin, P. Højrup, A.J. Bleasby, Rapid identification of proteins by peptide-mass fingerprinting, Curr. Biol. 3 (6) (1993) 327–332.
[21] P.A. Pevzner, Computational Molecular Biology: An Algorithmic Approach, MIT Press, New York, 2000.
[22] P.A. Pevzner, V. Dančík, C.L. Tang, Mutation-tolerant protein identification by mass spectrometry, J. Comp. Biol. 7 (6) (2000) 777–787.
[23] P.A. Pevzner, Z. Mulyukov, V. Dančík, C.L. Tang, Efficiency of database search for identification of mutated and modified proteins via mass spectrometry, Genome Res. 11 (2) (2001) 290–299.
[24] G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol. 1–3, Springer, Berlin, 1997.
[25] J. Setubal, J. Meidanis, Introduction to Computational Molecular Biology, PWS, Boston, 1997.
[26] L. Stryer, Biochemistry, Freeman, New York, 1988.
[27] J.R. Yates III, Database searching using mass spectrometry data, Electrophoresis 19 (6) (1998) 893–900.
[28] J.R. Yates, J.K. Eng, A.L. McCormack, Mining genomes: correlating tandem mass-spectra of modified and unmodified peptides to sequences in nucleotide databases, Anal. Chem. 67 (18) (1995) 3202–3210.
[29] J.R. Yates III, S. Speicher, P.R. Griffin, T. Hunkapillar, Peptide mass maps: a highly informative approach to protein identification, Anal. Biochem. 214 (1993) 397–408.