Faster Deterministic Modular Subset Sum

Krzysztof Potępa ✉
Jagiellonian University, Kraków, Poland

Abstract

We consider the *Modular Subset Sum* problem: given a multiset X of integers from $\mathbb{Z}_m$ and a target integer t, decide if there exists a subset of X with a sum equal to $t \pmod m$. Recent independent works by Cardinal and Iacono (SOSA'21), and Axiotis et al. (SOSA'21) provided simple and near-linear algorithms for this problem. Cardinal and Iacono gave a randomized algorithm that runs in $\mathcal{O}(m \log m)$ time, while Axiotis et al. gave a deterministic algorithm that runs in $\mathcal{O}(m \text{ polylog } m)$ time. Both results work by reduction to a text problem, which is solved using a dynamic strings data structure.

In this work, we develop a simple data structure, designed specifically to handle the text problem that arises in the algorithms for Modular Subset Sum. Our data structure, which we call the *shift-tree*, is a simple variant of a segment tree. We provide both a hashing-based and a deterministic variant of the *shift-trees*.

We then apply our data structure to the Modular Subset Sum problem and obtain two algorithms. The first algorithm is Monte-Carlo randomized and matches the $\mathcal{O}(m \log m)$ runtime of the Las-Vegas algorithm by Cardinal and Iacono. The second algorithm is fully deterministic and runs in $\mathcal{O}(m \log m \cdot \alpha(m))$ time, where α is the inverse Ackermann function.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Algorithm design techniques

Keywords and phrases Modular Subset Sum, String Problem, Segment Tree, Data Structure

Digital Object Identifier 10.4230/LIPIcs.ESA.2021.76

Acknowledgements I would like to thank Lech Duraj, Krzysztof Pióro, and Adam Polak for their help with preparing the paper, and the anonymous reviewers for their useful suggestions.

1 Introduction

The *Subset Sum* is a fundamental problem in computer science. It is defined as follows: given a multiset X of n positive integers and a target integer t, decide if there exists a subset of X, such that the sum of its elements is exactly t. The problem is known to be NP-complete [14], but only in a weak sense: a classic dynamic programming approach of Bellman [5] solves it in pseudo-polynomial $\mathcal{O}(nt)$ time. In recent years, there has been a lot of research towards improving the runtime [16, 17, 7, 12], which culminated in near-linear $\widetilde{\mathcal{O}}(n+t)$ algorithms [7, 12].[1]

In this work, we focus on the *Modular Subset Sum* problem. The *Modular Subset Sum* is a natural variant of the Subset Sum problem, where all sums are taken modulo m, for some given modulus m. We assume that the input multiset X is provided in a compact form: as a list of $\mathcal{O}(m)$ distinct elements along with their multiplicities. This assumption allows us to omit dependence on the number of elements n in algorithm complexities. Moreover, we focus on algorithms that return all possible subset sums, i.e. a set of all attainable values of t.

The dynamic programming of Bellman [5] can be easily adapted to solve the modular case in $\mathcal{O}(nm)$ time. Let S_i be the set of all attainable subset sums using only the first i elements. Bellman's algorithm iteratively computes the sets $S_1, ..., S_n$ using formula

[1] By writing $\widetilde{\mathcal{O}}(f(n))$, we mean $\mathcal{O}(f(n) \text{ polylog } f(n))$.

$S_i = S_{i-1} \cup (S_{i-1} + x_i)$, where x_i is the i-th input element and $S_{i-1} + x_i = \{a + x_i : a \in S_{i-1}\}$. Most of the currently known improved algorithms simply simulate the consecutive iterations of Bellman's algorithm faster. An early notable exception is the $\widetilde{\mathcal{O}}(m^{5/4})$ algorithm of Koiliaris and Xu [16], which uses a divide-and-conquer approach based on results from number theory.

Abboud et al. [1] obtained a SETH-based conditional lower bound for the Subset Sum problem, which in particular implies that the Modular Subset Sum cannot be solved in $\mathcal{O}(m^{1-\varepsilon})$ time for any $\varepsilon > 0$. The first randomized algorithm that matched their lower-bound (up to subpolynomial factors) was introduced by Axiotis et al. in [4]. They achieved a running time of $\mathcal{O}(m \log^7 m)$ by simulating Bellman's dynamic programming faster using ideas from linear sketching.

Recently, simple and practical algorithms were provided independently in [9, 3]. Both results work by reducing the problem of computing Bellman's iteration to a text problem, but use different data structures to solve it efficiently. A Las-Vegas randomized $\mathcal{O}(m \log m)$ algorithm by Cardinal and Iacono [9] uses the dynamic strings data structure of Gawrychowski et al. [10]. The authors also introduced a simpler alternative, called *Data Dependent Trees*, with logarithmic bounds per operation. On the other hand, Axiotis et al. [3] obtained a deterministic $\mathcal{O}(m \text{ polylog } m)$ algorithm by employing a deterministic data structure of Mehlhorn et al. [18] instead. More precisely, their algorithm is output-sensitive and works in $\mathcal{O}(|X^*| \text{ polylog } |X^*|)$ time, where X^* is the set of all attainable subset sums. The authors provided also a very simple, randomized $\mathcal{O}(m \log^2 m)$ algorithm that uses only an elementary prefix sum structure.

A very recent result of Bringmann and Nakos [8] provides near-linear algorithms for computing the sumset $A_1 + ... + A_n$, for $A_1, ..., A_n \subseteq \mathbb{Z}_m$. This problem generalizes the Modular Subset Sum: the set of all attainable subset sums can be expressed as a sumset $\{0, x_1\} + ... + \{0, x_n\}$.

1.1 Our contributions

In this work, we develop a simple tree-based data structure, designed specifically to handle the text problem that arises in the algorithms for Modular Subset Sum. Our data structure, which we call a *shift-tree*, maintains a string s under the following operations:

- **(i)** change a single character of s;
- **(ii)** cyclically shift s by k positions;
- **(iii)** given another string t with its corresponding shift-tree, and an interval $[a; b]$, list all positions in $[a; b]$ where strings s and t differ.

We provide two variants of the data structure: a hashing-based one, and a deterministic one with slightly worse time complexity (by $\alpha(n)$, where α is the inverse Ackermann function). By applying shift-trees to the Modular Subset Sum problem, we obtain the following algorithms:

▶ **Theorem 1.** *There exists an algorithm that returns all attainable modular subset sums of a multiset of integers from $\mathbb{Z}_m$ with high probability, in time $\mathcal{O}(m \log m)$ and space $\mathcal{O}(m)$.*

▶ **Theorem 2.** *There exists a deterministic algorithm that returns all attainable modular subset sums of a multiset of integers from $\mathbb{Z}_m$ in time $\mathcal{O}(m \log m \cdot \alpha(m))$ and space $\mathcal{O}(m)$.*

The first variant is Monte-Carlo randomized and matches the runtime of Las-Vegas algorithm by Cardinal and Iacono [9]. The second variant is fully deterministic and improves upon the result of Axiotis et al. [3]. Our algorithms are offline as they process the input elements in specific order to achieve their running times.

Although we provide a detailed analysis only for Monte-Carlo randomized and deterministic shift-trees, it is also possible to obtain a Las-Vegas implementation of the data structure. Such an implementation automatically leads to a Las-Vegas algorithm for Modular Subset Sum that truly matches the runtime obtained in [9]. We outline this approach in Remark 8.

Sketch of the shift-tree data structure

We now explain the high-level idea behind our data structure. The shift-tree is a perfect binary tree built upon some string s. The leaves of the tree store the consecutive letters of string s. Since the tree is perfect, the length of string s is required to be a power of two. The inner nodes correspond to substrings of s formed from underlying leaves and store their hashes. The hashes can be updated in logarithmic time after changing a single character of s.

Consider two shift-trees T_1 and T_2 built for strings s_1 and s_2 respectively, such that $|s_1| = |s_2| = m$. We can find all positions where s_1 and s_2 differ by descending from the roots of both trees simultaneously. We compare hashes in the roots and proceed recursively with the children if the hashes differ. Assuming there is no hash collision, we end up in leaves corresponding to positions where s_1 and s_2 differ. Such a procedure will take $\mathcal{O}(k \log m)$ time, where k is the number of differences.

The tricky operation is the cyclic shift of the maintained string. A naive approach would be to simply rebuild the whole tree in linear time. We improve this by noticing that some parts of the tree can be reused. Assume that the string has length m and we want to shift the string by 2^j. Such operation is equivalent to shifting subtrees of size 2^j by 1, what can be done by changing links to children on the appropriate level of the shift-tree. After such modification, hashes on higher levels still need to be updated, but not the hashes in the moved subtrees. This yields a total time of $\mathcal{O}(m/2^j)$ for a cyclic shift by 2^j, and it can be easily extended to shifts of form $k2^j$. Even though it seems like a subtle improvement, it is enough to obtain a fast algorithm for Modular Subset Sum.

To make the shift-trees deterministic, we replace hashes with *tags*. Tags are identifiers associated with strings, but unlike hashes, they are not unique: one string can be represented by multiple tags. Each time a node is updated it receives a new tag. We propagate the information about tags that represent the same strings lazily while searching for differences. More specifically, if the tags are not known to be equal, the search procedure always recurs. If the recursion was unnecessary, we know about it upon return and we can memorize that the respective tags were equivalent.

Sketch of the algorithm for Modular Subset Sum

Our algorithm follows the ideas of [3, 9]. We simulate Bellman's algorithm faster. We iteratively compute the sets of new attainable subset sums C_i after adding the i-th element. More precisely, $C_i = S_i \setminus S_{i-1} = (S_{i-1} + x_i) \setminus S_{i-1}$. The key idea is to notice that instead of computing C_i, we can compute the symmetric difference $D_i = (S_{i-1} + x_i) \triangle S_{i-1}$, and then reduce it to C_i, because $|D_i| = 2|C_i|$.

Let $s_i \in \{0,1\}^m$ be the characteristic vector of the set S_i, i.e. $s_i[j] = 1$ iff $j \in S_i$. The problem of finding the set D_{i+1} is then reduced to the problem of finding differences between the string s_i and its cyclic shift. We apply shift-trees to solve this problem efficiently. The shift-tree requires the length of the string to be a power of two, so we assume that $m = 2^k$ for now (we show how to get rid of this assumption in section 5). Consider two shift-trees T_1 and T_2 built for string s_i and its cyclic shift respectively. We simulate the Bellman's algorithm step as follows:

(i) adjust the cyclic shift of T_2;
(ii) find the set D_i by comparing T_1 and T_2;
(iii) update the trees with new attainable subset sums.

The bottleneck of the algorithm are the adjustments of cyclic shift of T_2: if elements are processed in arbitrary order, the total complexity of shift operations can be $\mathcal{O}(m^2)$. In section 4, we show that if elements are processed in a bit-reversal order, then the shift operations amortize to $\mathcal{O}(m \log m)$.

1.2 Preliminaries

We introduce the following notation for strings. We use the same notation for other sequences.

▶ **Definition 3.** *Given a string $s = c_0 ... c_{n-1}$, we refer to c_i as $s[i]$ and to substring $c_i ... c_j$ as $s[i : j]$.*

▶ **Definition 4.** *Given a string s and $k \in \mathbb{Z}$, we denote by s^{+k} and s^{-k} the* cyclic shift *of s by k positions to the right and left respectively. In other words, for every $i \in \{0, ..., |s| - 1\}$:*

$$s[i] = s^{+k}[(i + k) \bmod |s|] = s^{-k}[(i - k) \bmod |s|]$$

2 Shift-trees

2.1 Overview

We introduce *shift-trees*, a variant of the segment tree data structure. A *shift-tree* T maintains a string s of length $m = 2^n$ over an alphabet Σ and supports the following operations:

- T.Init(s): Initialize the data structure with string s.
- T.Set(i, x): Given an index $i \in \{0, ..., |s| - 1\}$ and a letter $x \in \Sigma$, change $s[i]$ to x.
- T.Shift(k): Given an offset $k \in \mathbb{Z}$, replace s with s^{+k}, i.e. cyclically shift the string s by k positions to the right.
- T.Diff(Q, a, b): Given another shift-tree Q representing a string q such that $|s| = |q|$, list all differences between $s[a : b]$ and $q[a : b]$, i.e. return the list L of all integers x such that $a \le x \le b$ and $s[x] \neq q[x]$.

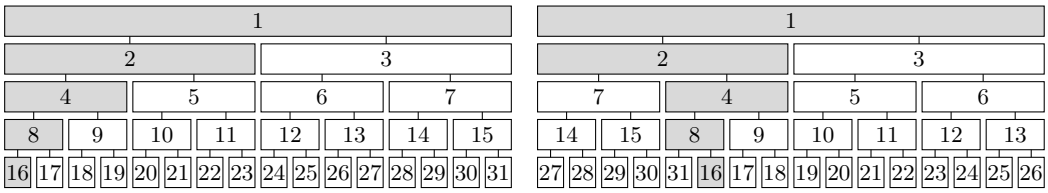
In this section, we describe a hashing-based version of the data structure, which uses $\mathcal{O}(m)$ memory and supports these operations in the following time complexities:

- Init: $\mathcal{O}(m)$;
- Set: $\mathcal{O}(\log m)$;
- Shift(k): $\mathcal{O}(m/2^j)$, where j is the largest integer such that $2^j \mid k$;
- Diff: $\mathcal{O}((d + 1) \log m)$, where $d = |L|$ is the number of differences.[2]

In section 3, we present a variant that is fully deterministic, but achieves a slightly worse time complexity (by $\alpha(m)$, where α is the inverse Ackermann function).

We require an integer alphabet Σ of size $\mathcal{O}(\text{poly}(m))$. We use a standard Rabin-Karp rolling hash function [15]: we choose a sufficiently big prime p and an integer $r \in \mathbb{Z}_p$, where r is chosen uniformly at random. The hash of a string s is defined as $h(s) = \sum_{i=0}^{|s|-1} s[i] \cdot r^i \bmod p$. We assume that $\Sigma \subseteq \mathbb{Z}_p$, so $h(x) = x$ for $x \in \Sigma$. If hashes of two strings of the same length are equal, then the strings are equal with high probability. Moreover, given hashes of some strings s_1 and s_2, one can compute hash of their concatenation using the following identity: $h(s_1 s_2) = h_1 + h_2 \cdot r^{|s_1|}$. To enable constant-time computation of this formula, we precompute powers of r up to r^m. We use these properties extensively in our data structure.

[2] By writing $d + 1$ in the complexity, we mean that the runtime of Diff operation is $\mathcal{O}(\log m)$ if $d = 0$.



(a) $\Delta = 0000_2 = 0$. **(b)** $\Delta = 0101_2 = 5$.

Figure 1 Shift-tree node numbering for different values of Δ.

2.2 Structure

Let s be the string maintained by the data structure and let $|s| = 2^n$. The shift-tree is a perfect binary tree built upon the string s. The leaves of the tree store the consecutive letters of string s, with the leftmost one corresponding to $s[0]$ and the rightmost one corresponding to $s[|s| - 1]$. The inner nodes correspond to substrings of s formed from underlying leaves and store hashes to enable their fast comparison. By level(v) we denote the distance from the node v to the root node. The root node has level 0 and the leaves have level n. There are 2^k nodes on the k-th level.

We now provide a compact $\mathcal{O}(m)$ memory representation of the data structure that enables us to achieve the desired complexity of SHIFT operation. The only data stored in memory is an array of hashes $H[1 : 2^{n+1} - 1]$ and a single integer $\Delta \in \{0, ..., 2^n - 1\}$. The values in $H[2^n : 2^{n+1} - 1]$ correspond to the leaves and are letters of the represented string. The value of Δ defines a cyclic shift of leaf indices, i.e. the j-th leftmost leaf of the tree has index $(j - \Delta) \bmod 2^n + 2^n$.

The tree structure is defined implicitly based on the value of Δ as follows. The nodes of the tree are numbered from 1 to $2^{n+1} - 1$. The nodes on the k-th level are numbered from 2^k to $2^{k+1} - 1$. In particular, the root node has index 1 and the leaves have indices from 2^n to $2^{n+1} - 1$. We first introduce the following auxiliary function:

$$\text{skew}(k) = \left\lfloor \frac{\Delta}{2^{n-k}} \right\rfloor \bmod 2$$

Note that floor division by a power of two is equivalent to right bitwise shift, so value of skew(k) is simply $(n - k)$-th least significant bit of Δ. Let i be an inner node and let $k = \text{level}(i)$. We define the children of node i as follows:

$$\text{left}(i) = (2i - \text{skew}(k + 1)) \bmod 2^{k+1} + 2^{k+1}$$
$$\text{right}(i) = (2i + 1 - \text{skew}(k + 1)) \bmod 2^{k+1} + 2^{k+1}$$

We also define the parent of node $i \neq 1$ at level k.

$$\text{parent}(i) = \left\lfloor \frac{(i + \text{skew}(k)) \bmod 2^k + 2^k}{2} \right\rfloor$$

The *left*, *right* and *parent* functions can be implemented in constant time. The following lemma and corollary summarize the properties of a tree structure defined as above.

▶ **Lemma 5.** *The functions left, right and parent define a perfect binary tree, such that the indices of nodes on the k-th level, when ordered from left to right, form a sequence $(2^k, ..., 2^{k+1} - 1)^{+\lfloor \Delta / 2^{n-k} \rfloor}$.*

Proof. We prove the lemma by induction on the level. The condition is satisfied for the 0-th level, which contains only the root node with index 1. Assume now that the condition is satisfied for the k-th level, i.e. the nodes on the k-th level form a sequence $(2^k, ..., 2^{k+1}-1)^{+\delta}$, where $\delta = \lfloor \Delta/2^{n-k} \rfloor$. By substituting indices of children in this sequence, we obtain the following sequence for the $(k+1)$-st level:

$$(\text{left}(2^k), \text{right}(2^k), ..., \text{left}(2^{k+1}-1), \text{right}(2^{k+1}-1))^{+2\delta}$$

The shift is now 2δ, because each element has been replaced by two elements. We want to prove that this is exactly the sequence $(2^{k+1}, ..., 2^{k+2}-1)^{+\lfloor \Delta/2^{n-k-1} \rfloor}$. The shift $\lfloor \Delta/2^{n-k-1} \rfloor$ can be rewritten as $2\delta + x$, where $x = \text{skew}(k+1) = \lfloor \Delta/2^{n-k-1} \rfloor \bmod 2$. We now simplify the equation:

$$\begin{aligned}
(\text{left}(2^k), \text{right}(2^k), ..., \text{left}(2^{k+1}-1), \text{right}(2^{k+1}-1))^{+2\delta} &\stackrel{?}{=} (2^{k+1}, ..., 2^{k+2}-1)^{+2\delta+x} \\
(\text{left}(2^k), \text{right}(2^k), ..., \text{left}(2^{k+1}-1), \text{right}(2^{k+1}-1)) &\stackrel{?}{=} (2^{k+1}, ..., 2^{k+2}-1)^{+x} \\
(\text{left}(2^k) - 2^{k+1}, \text{right}(2^k) - 2^{k+1}, ..., \text{right}(2^{k+1}-1) - 2^{k+1}) &\stackrel{?}{=} (0, ..., 2^{k+1}-1)^{+x}
\end{aligned}$$

After substituting the values of *left* and *right*, and simplifying, we obtain the following:

$$((0-x) \bmod 2^{k+1}, (1-x) \bmod 2^{k+1}, ..., (2^{k+1}-1-x) \bmod 2^{k+1}) \stackrel{?}{=} (0, ..., 2^{k+1}-1)^{+x}$$

The obtained equation trivially satisfies the definition of cyclic shift by x, and all transformations were equivalent. This completes the induction.

We complete the proof by showing that *parent* function is well-defined. Consider an inner node on the k-th level with index $2^k + i$. It is enough to show that it is parent of its children. After substituting and simplifying the formulas, we get the desired result:

$$\begin{aligned}
\text{parent}(\text{left}(2^k + i)) &= \left\lfloor \frac{2i + 2^{k+1}}{2} \right\rfloor = 2^k + i \\
\text{parent}(\text{right}(2^k + i)) &= \left\lfloor \frac{2i + 1 + 2^{k+1}}{2} \right\rfloor = 2^k + i
\end{aligned}$$

◀

▶ **Corollary 6.** *The functions left, right and parent define a perfect binary tree such that:*
(a) *nodes on the k-th level have indices from 2^k to $2^{k+1}-1$, for each valid k;*
(b) *the indices of leaves, when ordered from left to right, form a sequence $(2^n, ..., 2^{n+1}-1)^{+\Delta}$;*
(c) *the structure of subtrees rooted at the k-th level depends only on $\Delta \bmod 2^{n-k}$.*

Proof. The first two properties follow instantly from the lemma 5. For the property (c), notice that the links on these levels depend only on the values $\text{skew}(k+1), ..., \text{skew}(n)$. These values are exactly the $n-k$ least significant bits of Δ. ◀

2.3 Invariant

Let s be the string maintained by the data structure and let $|s| = 2^n$. We define the string associated with a node i recursively as follows:

$$\text{str}(i) = \begin{cases} s^{-\Delta}[i - 2^n] & \text{for } i \geq 2^n \text{ (i.e. } i \text{ is leaf node)} \\ \text{str}(\text{left}(i))\,\text{str}(\text{right}(i)) & \text{for } i < 2^n \text{ (i.e. } i \text{ is inner node)} \end{cases}$$

By corollary 6b, the k-th letter of string s is associated with the k-th leftmost leaf. It follows that $s = \text{str}(1)$, i.e. string associated with the root node is s. We maintain the following invariant:

▶ **Invariant 7.** *For each node i the following holds:* $H[i] = h(\text{str}(i))$.

The invariant ensures that each node stores a hash of its associated string. We use this property to implement the DIFF operation in required time complexity.

2.4 Operations

Let s be the string maintained by the data structure and let $|s| = m = 2^n$. We first define an UPDATE(i) primitive that is used by all operations that modify the data structure. The UPDATE procedure simply recalculates the hash of an inner node i based on hashes of its children. This can be done in constant time using basic modular arithmetic, if appropriate powers of r are precomputed.

Algorithm 1 The UPDATE procedure.

```
1: function UPDATE(i)
2:     H[i] ← (H[left(i)] + H[right(i)] · r^|str(left(i))|) mod p
```

Init. We initialize Δ with 0 and leaves with the letters of the input string s. Specifically, we set $H[2^n + i] = s[i]$ for each $i \in \{0, ..., 2^n - 1\}$, because letter $s[i]$ is associated with the node $2^n + i$. We then compute all the hashes by calling an UPDATE on the remaining nodes, beginning at the bottom of the tree. Overall, the INIT operation updates $\mathcal{O}(m)$ nodes and runs in $\mathcal{O}(m)$ time.

Set. Assume that we change $s[i]$ to x. Let $j = (i - \Delta) \bmod 2^n + 2^n$. Notice that, $\text{str}(j) = s^{-\Delta}[j - 2^n] = s[(j + \Delta) \bmod 2^n] = s[i]$. In order to fix the invariant, we set $H[j] = x$ and update hashes of all the ancestors of j. The tree has $\mathcal{O}(\log m)$ levels, so the total runtime of SET operation is $\mathcal{O}(\log m)$.

Algorithm 2 INIT operation.

```
1: function INIT(s)
2:     Δ ← 0
3:     for i ← 0, ..., 2^n − 1 do
4:         H[2^n + i] ← s[i]
5:     for i ← 2^n − 1, ..., 1 do
6:         UPDATE(i)
```

Algorithm 3 SET operation.

```
1: function SET(i, x)
2:     j ← (i − Δ) mod 2^n + 2^n
3:     H[j] ← x
4:     while j ≠ 1 do
5:         j ← parent(j)
6:         UPDATE(j)
```

Shift. Assume that we apply a right cyclic shift by k positions to s. Let j be the largest integer such that $2^j \mid k$. In order to fix the invariant, we first set Δ to $(\Delta + k) \bmod 2^n$. Notice that the invariant is now satisfied for leaves. Moreover, by corollary 6c the structure of subtrees rooted at level $n - j$ didn't change, so the invariant is also satisfied for levels $n - j, ..., n$. It remains to update hashes on the remaining $n - j$ levels by calling the UPDATE procedure on their nodes. Overall, $\mathcal{O}(2^{n-j})$ nodes are updated and the SHIFT operation runs in $\mathcal{O}(m/2^j)$ time.[3]

[3] Provided algorithm requires computation of $j = \max\{d \in \mathbb{N} : 2^d \mid k\}$. In practice, it is sufficient to compute 2^j instead of computing j directly. This can be done in constant time using the following bit-hack: `k & ~(k-1)`.

Algorithm 4 SHIFT operation.

```
1: function SHIFT(k)
2:     if k mod 2^n = 0 then
3:         return
4:     j ← max{d ∈ ℕ : 2^d | k}
5:     Δ ← (Δ + k) mod 2^n
6:     for i ← 2^{n−j} − 1, ..., 1 do
7:         UPDATE(i)
```

Diff. Assume we look for differences between the strings maintained by the trees T and Q in interval $[a; b]$. We provide a recursive procedure FINDDIFFERENCES(T, Q, a, b, i, j, x, y) that returns required set of differences between substrings associated with node i of the tree T and node j of the tree Q. The procedure additionally tracks an interval $[x; y]$ that is associated with both nodes, i.e. $T.\mathrm{str}(i) = T.s[x : y]$ and $Q.\mathrm{str}(j) = Q.s[x : y]$.

The FINDDIFFERENCES procedure works as follows. If $[a; b] \cap [x; y] = \emptyset$ then procedure returns empty set instantly, because we only look for differences in the interval $[a; b]$. If hashes of nodes i and j are equal then the substrings are equal w.h.p., so the procedure returns no differences as well. Otherwise, there is at least one difference between the strings associated with nodes i and j. If the nodes are leaves, then we report the difference. If the nodes are inner nodes, the procedure is invoked recursively on left and right children.

The DIFF operation simply calls FINDDIFFERENCES on roots of the trees T and Q. The procedure will return all the required differences as long as there is no hash collision.

Algorithm 5 The DIFF operation.

```
1: function T.DIFF(Q, a, b)
2:     return FINDDIFFERENCES(T, Q, a, b, 1, 1, 0, 2^n − 1)
3:
4: function FINDDIFFERENCES(T, Q, a, b, i, j, x, y)
5:     if [a; b] ∩ [x; y] = ∅ or T.H[i] = Q.H[j] then
6:         return ∅
7:     if x = y then        ▷ The nodes i and j are leaves if they represent an unit interval
8:         return {x}
9:     z ← (x+y+1)/2 ▷ The left nodes represent [x : z − 1] and the right nodes represent [z : y]
10:    A ← FINDDIFFERENCES(T, Q, a, b, T.left(i), Q.left(j), x, z − 1)
11:    B ← FINDDIFFERENCES(T, Q, a, b, T.right(i), Q.right(j), z, y)
12:    return A ∪ B
```

We now argue the complexity of DIFF operation. Let d be the number of differences that were found. Let s_k be the number of FINDDIFFERENCES calls for which the processed nodes were on the k-th level and the procedure recurred. If the procedure recurred, there existed at least one difference between the strings associated with the nodes. We can divide such calls into two categories:

(a) there is a difference in $[x; y] \cap [a; b]$ that should be reported;

(b) there is a difference in $[x; y] \setminus [a; b]$ that should be ignored and $[x; y] \cap [a; b] \neq \emptyset$.

The number of calls that belong to the category (a) is bounded by d, i.e. number of reported differences. There are at most 2 calls that belong to the category (b), because the interval $[x; y]$ must contain a or b. It follows that $s_k \leq d + 2$ and $s_0 + ... + s_{n-1} \leq (d + 2) \cdot n$. We can charge the calls that didn't recur to their parents, so the total running time of DIFF operation is $\mathcal{O}((d + 1) \log m)$.

In order to complete the analysis, we bound the probability that DIFF operation fails to report all differences. Such situation may occur only if nodes processed by FINDDIFFERENCES have different associated strings, but equal hashes. Let $s_1 \neq s_2$ be strings of length k.

$$\Pr\left[h(s_1) = h(s_2)\right] = \Pr\left[\sum_{i=0}^{k-1}(s_1[i] - s_2[i])r^i \equiv 0 \pmod{p}\right] \leq \frac{k}{p}$$

The inequality holds, because the sum on the left side is a non-zero polynomial of degree at most k, evaluated in randomly chosen point r. By application of union bound, we obtain that the probability of failure is at most $m \log m/p$. Assuming operations on hashes of size $\mathcal{O}(\log m)$ are taking constant time, we can choose $p = \Theta(\text{poly}(m))$ and obtain high probability of success.

▶ Remark 8. It is also possible to achieve a Las-Vegas implementation of shift-trees. The key observation is that the hash function doesn't need to be associative, i.e. one can hash "subtrees" instead of substrings. This allows us to replace the hash function with any injective mapping $h(x, y)$ from pairs of hashes into new hashes. Only the UPDATE procedure needs to be adapted to compute the hash $H[i]$ of i-th node as $h(H[\text{left}(i)], H[\text{right}(i)])$.

The missing piece is how to implement the mapping $h(x, y)$. This can be done by simply generating it on demand, and storing the mapping in a hashtable. Some garbage collection mechanism (such as reference counting) is required to maintain linear memory usage. This yields a Las-Vegas implementation of shift-trees, with the same expected runtime bounds.

By replacing hashtable with a BST, one can obtain a deterministic implementation of the data structure. Such modification introduces logarithmic runtime overhead. In the next section, we improve upon this by allowing amortization.

3 Deterministic shift-trees

3.1 Overview

We now provide a deterministic variant of the data structure introduced in previous section. Let $m = 2^n$. Assume that we maintain several shift-trees $T_1, ..., T_r$ associated with strings $s_1, ..., s_r \in \Sigma^m$ of the same length, allowing comparisons between those strings. Let $K = |s_1| + ... + |s_r| = mr$, and let T_i be one of the trees. Then, we can do the shift-tree operations on T_i with the following amortized time complexities:

- INIT: $\mathcal{O}(m \cdot \alpha(K))$;
- SET: $\mathcal{O}(\log m \cdot \alpha(K))$;
- SHIFT(k): $\mathcal{O}(m/2^j \cdot \alpha(K))$, where j is the largest integer such that $2^j \mid k$;
- DIFF: $\mathcal{O}((d+1) \log m \cdot \alpha(K))$, where d is the number of differences.

Moreover, the data structures use $\mathcal{O}(K)$ memory overall. The only constraint that we put on alphabet Σ is the support for equality tests. This contrasts with the randomized variant, where an integer alphabet of polynomial size is required.

3.2 Tags

We replace hashing with the concept of *tags*. Tags are simply identifiers associated with strings. Unlike hashes, they are not unique: one string can be represented by multiple tags. Each inner node of the shift-tree stores a tag instead of a hash. A new tag is created every time a node is updated.

We denote the string associated with a tag t by $\mathrm{str}(t)$. The strings associated with tags are not stored in memory. Instead, we maintain an equivalence relation $\mathcal{R}$ over the set of tags used in all the shift-trees. The relation $\mathcal{R}$ satisfies the following property:

▶ **Invariant 9.** *For each tag a and tag b such that $a \equiv_{\mathcal{R}} b$, the strings $\mathrm{str}(a)$ and $\mathrm{str}(b)$ are equal.*

Note that the inverse doesn't need to hold. In other words, the relation $\mathcal{R}$ partially captures the equality relation between strings associated with tags. The relation is refined during operations on the shift-trees using the following operations:

- NewTag(): Create a new tag x with its own singleton equivalence class, and return x.
- Find(x): Given a tag x, return identifier of its equivalence class.
- Union(x, y): Given tags x and y, union their equivalence classes (i.e. insert $x \equiv_{\mathcal{R}} y$ and close the relation transitively).
- DeleteTag(x): Given a tag x, remove it from its equivalence class and free its memory.

In order to support these operations efficiently, we represent the equivalence classes of $\mathcal{R}$ using a union-find data structure. A simple extension of standard disjoint-set forest implementation with support for element removal has been proposed by Kaplan et al. in [13].

▶ **Theorem 10** ([13])**.** *There exists a data structure that maintains an equivalence relation $\mathcal{R}$ using linear memory under operations* Find, Union *and* DeleteTag *in amortized $\mathcal{O}(\alpha(n))$ time, and* NewTag *in $\mathcal{O}(1)$ time, where n is the number of maintained elements.*

Their approach is based on lazy deletions: elements to be deleted are marked and the union-find trees are rebuilt if the fraction of marked elements is greater than half. More involved approaches with constant time deletions have been known in literature [2, 6], but such improvement doesn't change the amortized time complexity of our data structure.

The idea to use union-find data structure for detecting mismatches has been already proposed by Gawrychowski et al. in [11].

▶ Remark. In general, the DeleteTag operation is not only useful for space optimization. If unused tags are not removed, the complexity of operations is dependent on the total number of NewTag calls, which can be large. This is not an issue if only $\mathcal{O}(\mathrm{poly}(K))$ tags are created in total, because $\mathcal{O}(\alpha(\mathrm{poly}(K))) = \mathcal{O}(\alpha(K))$.

3.3 Operations

We now adapt the Update procedure and Diff operation to work with tags. The Init, Set and Shift operations use the Update primitive and don't require changes.

Update. We simply create a new tag for the updated node. If the node already contains a tag (i.e. the Update is called after initialization), we delete it in order to maintain linear memory usage. The newly created tag is in a singleton equivalence class of $\mathcal{R}$, so it trivially satisfies the invariant.

Algorithm 6 The Update procedure.

```
1: function Update(i)
2:     if H[i] ≠ null then
3:         DeleteTag(H[i])
4:     H[i] ← NewTag()
```

Diff. We adapt the FindDifferences procedure as follows. Assume that we are looking for differences in interval $[a; b]$. Let t_1 and t_2 be tags in compared tree nodes, and $[x; y]$ the interval associated with these nodes. If $t_1 \equiv_{\mathcal{R}} t_2$ then the compared substrings are equal, so we exit instantly. Otherwise, we search for differences recursively in the left and right subtrees. If no differences are found and $[x; y] \subseteq [a; b]$, we know that $\mathrm{str}(t_1) = \mathrm{str}(t_2)$, so we can safely add $t_1 \equiv_{\mathcal{R}} t_2$ to relation $\mathcal{R}$ via $\textsc{Union}(t_1, t_2)$.

Algorithm 7 The FindDifferences procedure.

```
1:  function FindDifferences(T, Q, a, b, i, j, x, y)
2:      if [a; b] ∩ [x; y] = ∅ then                ▷ Check if we are outside of the search interval
3:          return ∅
4:      if x = y then        ▷ The nodes i and j are leaves if they represent an unit interval
5:          if T.H[i] ≠ Q.H[j] then    ▷ The leaves contain letters – compare them directly
6:              return {x}
7:          else
8:              return ∅
9:      if Find(T.H[i]) = Find(Q.H[j]) then        ▷ The inner nodes contain tags – use R.
10:         return ∅                          ▷ T.H[i] ≡_R Q.H[j] holds, so the strings are equal
11:     z ← (x+y+1)/2   ▷ The left nodes represent [x; z − 1] and the right nodes represent [z; y]
12:     A ← FindDifferences(T, Q, a, b, T.left(i), Q.left(j), x, z − 1)
13:     B ← FindDifferences(T, Q, a, b, T.right(i), Q.right(j), z, y)
14:     if A ∪ B = ∅ and [x; y] ⊆ [a; b] then
15:         Union(T.H[i], Q.H[j])
16:     return A ∪ B
```

We now briefly address the correctness of the FindDifferences procedure. Observe that if the condition $[a; b] \cap [x; y] \neq \emptyset$ holds then:

(i) the invariant 9 guarantees that the procedure will recur if substrings are not equal;

(ii) the procedure returns a difference for leaves iff their corresponding characters differ.

It follows by an easy induction on the level that the procedure returns all positions in $[a; b]$ where the strings differ and nothing more. Moreover, the procedure doesn't break the invariant when modifying the relation $\mathcal{R}$: if the condition in line 14 is true, then there are no differences between compared substrings.

3.4 Running time

Let K be the sum of lengths of strings maintained by all shift-trees and let $m = 2^n$ be the length of each string. We first note that the number of elements maintained by relation $\mathcal{R}$ never exceeds the total number of nodes in shift-trees, which is $\mathcal{O}(K)$, so any operation on $\mathcal{R}$ works in amortized $\mathcal{O}(\alpha(K))$ time.

We now argue the amortized running time of the Update and Diff operations. Let the actual cost of the i-th operation be a number c_i of operations on the relation $\mathcal{R}$. Let q_i be the number of equivalence classes of relation $\mathcal{R}$ after i operations. We define potential Φ_i to be $9q_i$. Clearly, $\Phi_i \geq \Phi_0 = 0$. The amortized cost of the i-th operation is $\widehat{c_i} = c_i + \Phi_i - \Phi_{i-1}$.

The actual cost of an Update operation is $c_i \leq 2$. The operation creates at most one new equivalence class, thus the amortized cost is $\widehat{c_i} = c_i + \Phi_i - \Phi_{i-1} \leq 2 + 9 \cdot 1 = 11$. It follows that the amortized time complexity of an Update operation is $\mathcal{O}(\alpha(K))$.

To estimate the amortized cost of Diff operation, we consider the FindDifferences calls that recurred. We say that the call is *wasted* if the compared strings are equal, but comparison of tags reported that they are not. Otherwise, if the compared strings are not

equal and the procedure recurred, we say that the call is *required*. Let w be the number of wasted calls and r be the number of required calls. We can charge the calls that didn't recur to their parents, so the total number of FindDifferences calls is bounded by $3(r+w)$. Each call does at most three operations on relation $\mathcal{R}$, so the cost of Diff operation is $c_i \leq 9(r+w)$.

Let d be the number of differences that have been found. We can bound the number of required calls r by $\mathcal{O}((d+1)\log m)$, the same way as in hashing-based shift-trees. We now focus our attention on the wasted calls. Assume that we are looking for differences in interval $[a;b]$. Let w' be the number of wasted calls such that the interval associated with compared nodes is contained within $[a;b]$. Notice that upon return, each such call unions two different equivalence classes, thus decreasing the potential by 9. It follows that Diff operation reduces the potential in total by $9w'$. On the other hand, the intervals associated with the remaining wasted calls must contain a or b, so there are at most 2 such calls for each tree level. It follows that the number of all wasted calls is bounded by $w' + 2\log m$. The amortized cost of Diff operation is then:

$$\widehat{c_i} \leq 9r + 9w - 9w' \leq 9r + 18\log m = \mathcal{O}((d+1)\log m)$$

and the amortized time complexity is $\mathcal{O}((d+1)\log m \cdot \alpha(K))$.

We complete analysis by providing amortized running time of Init, Set and Shift operations. The Init operation calls Update operation $\mathcal{O}(m)$ times, so its amortized running time is $\mathcal{O}(m \cdot \alpha(K))$. By the same argument we obtain the required amortized complexities for Set and Shift operation.

4 Traversing all cyclic shifts

In this section, we consider a problem of going over all the cyclic shifts of the shift-tree efficiently, in some order. This means that we want to consider all shifts $s^{+\sigma(0)}, ..., s^{+\sigma(|s|-1)}$, for σ being some permutation of $\{0, ..., |s|-1\}$. This requires invoking $\textsc{Shift}(\sigma(i) - \sigma(i-1))$ for $i = 1, 2, ..., |s|-1$, assuming the shift-tree initially represents $s^{+\sigma(0)}$. We claim that there exists a permutation σ such that the total complexity of these operations amortizes to $\mathcal{O}(m \log m)$ time. For simplicity, we consider the hashing-based shift-trees; the complexity for deterministic variant is just multiplied by $\alpha(K)$. This technique is crucial for our Modular Subset Sum algorithm and might be used for other problems, where the order of operations doesn't matter.

The time complexity of a single Shift operation depends heavily on the value of shift. Recall that the running time of $\textsc{Shift}(k)$ is $\mathcal{O}(m/2^j)$, where j is the largest integer such that $2^j \mid k$, and $m = 2^n$ is the size of the shift-tree. For example, a shift by 1 requires rebuilding the entire tree, while shift by $m/2$ takes constant time. It means that the complexity of going over all the cyclic shifts heavily depends on the permutation σ.

It turns out that a good permutation is a bit-reversal permutation, which we define as follows. We denote the bit-reverse of n-bit number j by $\text{bitrev}_n(j)$, i.e. if $j = \sum_{i=0}^{n-1} c_i 2^i$ then $\text{bitrev}_n(j) = \sum_{i=0}^{n-1} c_i 2^{n-i-1}$. We say that σ is a *bit-reversal permutation* if $\sigma(i) = \text{bitrev}_n(i)$.

▶ **Lemma 11.** *Let σ be a bit-reversal permutation of length $m = 2^n$ and T a shift-tree of length $\mathcal{O}(m)$. The sequence of operations $T.\textsc{Shift}(\sigma_i - \sigma_{i-1})$ for $i = 1, ..., m-1$ takes total time $\mathcal{O}(m \log m)$.*

Proof. Let $\delta_i = \sigma_i - \sigma_{i-1}$ and consider a single $\textsc{Shift}(\delta_i)$ operation. Let j be the largest integer such that $2^j \mid \delta_i$. The complexity of this operation is then $\mathcal{O}(m/2^j) = \mathcal{O}(2^{n-j})$. If $2^j \mid \delta_i$ and $2^{j+1} \nmid \delta_i$ then j is the least significant bit that is different between σ_i and σ_{i-1}. Since σ_i is a bit-reverse of i, it means that $n-j-1$ is the most significant bit that is different between i and $i+1$. Such situation happens only if $i+1$ is of form $k \cdot 2^{n-j-1}$, where k is odd. There are 2^j such numbers in range $[1; 2^n - 1]$. It follows that shifts for a given value of j take overall $\mathcal{O}(2^{n-j} \cdot 2^j) = \mathcal{O}(2^n)$ time. There are $\mathcal{O}(n)$ possible values of j, so the whole sequence of shifts takes $\mathcal{O}(2^n \cdot n) = \mathcal{O}(m \log m)$ time. ◀

5 Modular Subset Sum

In this section, we provide an algorithm for the Modular Subset Sum problem that uses the shift-tree data structure. Let $X = \{x_1, ..., x_n\}$ be a multiset of integers from $\mathbb{Z}_m$. Our algorithm computes the set $X^* \subseteq \mathbb{Z}_m$ such that $k \in X^*$ if and only if there exists a subset of X that sums to the value k modulo m. We assume that the input multiset X is provided in a compact form: as a list of $\mathcal{O}(m)$ distinct elements along with their multiplicities.

The algorithm is based on the so-called Bellman's iteration. Consider sets $S_0, ..., S_n \subseteq \mathbb{Z}_m$ such that $S_0 = \{0\}$ and $S_i = S_{i-1} \cup (S_{i-1} + x_i)$, where $S_{i-1} + x_i = \{a + x_i : a \in S_{i-1}\}$. It is easy to see that the set S_i is a set of all attainable subset sums of $\{x_1, ..., x_i\}$ and the final result is $X^* = S_n$. In order to compute the set S_i from S_{i-1}, it is sufficient to find the set $C_i = S_i \setminus S_{i-1} = (S_{i-1} + x_i) \setminus S_{i-1}$. If one can compute the set C_i in time $\mathcal{O}(|C_i| \cdot f(m))$, then the set X^* can be computed in total time $\mathcal{O}(m \cdot f(m))$.

The key idea of [4] is to notice that instead of computing $C_i = (S_{i-1} + x_i) \setminus S_{i-1}$, we can compute the symmetric difference $D_i = (S_{i-1} + x_i) \triangle S_{i-1}$. Computing the set D_i doesn't break the time complexity, because its size is only two times larger than C_i. We can now interpret problem of finding the set D_i as a text problem [3, 9]. Let $s_i \in \{0,1\}^m$ be the characteristic vector of the set S_i, i.e. $s_i[j] = 1$ iff $j \in S_i$. The problem of finding the set D_{i+1} is then reduced to problem of finding differences between the strings s_i and $s_i^{+x_{i+1}}$. The set D_{i+1} is exactly the set of indices, where these strings differ.

We now describe our algorithm. Let S be the set of all subset sums attainable using elements processed so far, and let s be its characteristic vector. Initially, the set S contains only 0. We maintain the characteristic vector s and its cyclic shift using two shift-trees, T_1 and T_2 respectively. The length of a string maintained by a shift-tree is required to be a power of two, but that may not be the case with the string s. We address this issue in the following way. Let L be the smallest power of two such that $L \geq 2m$. We consider the following auxiliary strings:

$$s' = s0^{L-m}$$
$$s'' = s0^{L-2m}s$$

The strings s' and s'' have length L, which is a power of two. Moreover, the string s'' has the following property: the string s^{+d} is a prefix of $(s'')^{+d}$, for any $d \in \{0, ..., m\}$. This property allows us to find differences between s and s^{+d} by comparing a prefix of s' with a prefix of $(s'')^{+d}$.

The tree T_1 maintains the string s' and the tree T_2 maintains a cyclic shift of the string s''. Specifically, we traverse all cyclic shifts of T_2 in bit-reversal order, as explained in the previous section. Assume that the current shift of T_2 is x, i.e. the string maintained by T_2 is $(s'')^{+x}$. Let μ be the multiplicity of x in the input multiset X. If $\mu = 0$, then $x \notin X$ and the algorithm proceeds to the next shift. Otherwise, we simulate μ Bellman's iterations for the

element x as follows. The x is contained in the multiset X, so $x \in \{0, ..., m-1\}$. It implies that we can find the set of differences D between s and s^{+x} by comparing a prefix of s' with a prefix of $(s'')^{+x}$. This is done using DIFF operation on T_1 and T_2. We then update the set of attainable subset sums S and both shift-trees appropriately. If no differences were found, we skip the rest of iterations for element x.

Every element of X corresponds to some cyclic shift, so all elements will be processed if all the cyclic shifts are considered. The set S is then the set of all attainable subset sums for X. We provide the pseudocode as Algorithm 8. In the pseudocode, we denote the multiplicity of element x in the set X by $\mu_X(x)$.

Algorithm 8 The MODULARSUBSETSUM algorithm.

```
1:  function ModularSubsetSum(X, m)
2:      S = {0}
3:      k ← min{d ∈ ℕ : 2^d ≥ 2m}
4:      L ← 2^k
5:      s_0 ← 10^{m-1}                          ▷ The characteristic vector of S_0 = {0}
6:      Initialize shift-tree T_1 with s_0 0^{L-m}
7:      Initialize shift-tree T_2 with s_0 0^{L-2m} s_0
8:      for i ← 1, ..., L − 1 do
9:          x ← bitrev_k(i)
10:         T_2.Shift(x − bitrev_k(i − 1))          ▷ Now T_2 represents the string (s'')^{+x}
11:         for j ← 1, ..., μ_X(x) do
12:             D ← T_1.Diff(T_2, 0, m − 1) ▷ D is the set of all differences between s and s^{+x}
13:             if D = ∅ then ▷ Skip the rest of iterations for x if no differences were found
14:                 break
15:             for d ∈ D \ S do
16:                 S ← S ∪ {d}
17:                 T_1.Set(d, 1)
18:                 T_2.Set((d + x) mod L, 1)
19:                 T_2.Set((d + x − m) mod L, 1)
20:     return S
```

We now analyse the running time of the algorithm. We assume the hashing-based shift-trees are used; the complexity for deterministic variant is just multiplied by $\alpha(m)$. The initialization of shift-trees takes $\mathcal{O}(m)$ time. The value of $\text{bitrev}_k(i)$ can be computed naively bit by bit in $\mathcal{O}(k) = \mathcal{O}(\log m)$ time, which in total takes $\mathcal{O}(m \log m)$ time. Moreover, the total time of all SHIFT operations amortizes to $\mathcal{O}(m \log m)$ time due to lemma 11. We now focus on the total running time of inner loops.

Consider a single Bellman's iteration. The complexity of a DIFF operation is $\mathcal{O}((|D| + 1) \log m)$. The algorithm adds $|D \setminus S| = |D|/2$ new elements to the set S and updates the shift-trees. Each tree update takes $\mathcal{O}(\log m)$ time. In total, a single Bellman's iteration takes $\mathcal{O}((|D| + 1) \log m)$ time.

The sum of sizes of all the sets of differences is at most $2m$. It means that if there are k Bellman's iterations in total, then their total running time is $\mathcal{O}((m + k) \log m)$. The condition in the line 13 ensures that the total number of executed Bellman's iterations is $\mathcal{O}(m)$ by skipping the iterations if the set is empty. It follows that all the iterations take $\mathcal{O}(m \log m)$ time in total.

We arrive at the total time complexity of $\mathcal{O}(m \log m)$. By replacing hashing-based shift-trees with their deterministic variant, we obtain a deterministic algorithm with running time of $\mathcal{O}(m \log m \cdot \alpha(m))$. We recall the theorems that summarize these results:

▶ **Theorem 1.** *There exists an algorithm that returns all attainable modular subset sums of a multiset of integers from $\mathbb{Z}_m$ with high probability, in time $\mathcal{O}(m \log m)$ and space $\mathcal{O}(m)$.*

▶ **Theorem 2.** *There exists a deterministic algorithm that returns all attainable modular subset sums of a multiset of integers from $\mathbb{Z}_m$ in time $\mathcal{O}(m \log m \cdot \alpha(m))$ and space $\mathcal{O}(m)$.*

References

1 Amir Abboud, Karl Bringmann, Danny Hermelin, and Dvir Shabtay. Seth-based lower bounds for subset sum and bicriteria path. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 41–57. SIAM, 2019. doi:10.1137/1.9781611975482.3.

2 Stephen Alstrup, Inge Li Gørtz, Theis Rauhe, Mikkel Thorup, and Uri Zwick. Union-find with constant time deletions. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *Lecture Notes in Computer Science*, pages 78–89. Springer, 2005. doi:10.1007/11523468_7.

3 Kyriakos Axiotis, Arturs Backurs, Karl Bringmann, Ce Jin, Vasileios Nakos, Christos Tzamos, and Hongxun Wu. Fast and simple modular subset sum. In Hung Viet Le and Valerie King, editors, *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, pages 57–67. SIAM, 2021. doi:10.1137/1.9781611976496.6.

4 Kyriakos Axiotis, Arturs Backurs, Ce Jin, Christos Tzamos, and Hongxun Wu. Fast modular subset sum using linear sketching. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 58–69. SIAM, 2019. doi:10.1137/1.9781611975482.4.

5 Richard E Bellman et al. Dynamic programming. *Cambridge Studies in Speech Science and Communication. Princeton University Press, Princeton*, 1957.

6 Amir M. Ben-Amram and Simon Yoffe. A simple and efficient union-find-delete algorithm. *Theor. Comput. Sci.*, 412(4-5):487–492, 2011. doi:10.1016/j.tcs.2010.11.005.

7 Karl Bringmann. A near-linear pseudopolynomial time algorithm for subset sum. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1073–1084. SIAM, 2017. doi:10.1137/1.9781611974782.69.

8 Karl Bringmann and Vasileios Nakos. Fast n-Fold Boolean Convolution via Additive Combinatorics. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41:1–41:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ICALP.2021.41.

9 Jean Cardinal and John Iacono. Modular subset sum, dynamic strings, and zero-sum sets. In Hung Viet Le and Valerie King, editors, *4th Symposium on Simplicity in Algorithms, SOSA 2021, Virtual Conference, January 11-12, 2021*, pages 45–56. SIAM, 2021. doi:10.1137/1.9781611976496.5.

10 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1509–1528. SIAM, 2018. doi:10.1137/1.9781611975031.99.

11 Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster longest common extension queries in strings over general alphabets. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 5:1–5:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.5.

12 Ce Jin and Hongxun Wu. A simple near-linear pseudopolynomial time randomized algorithm for subset sum. In Jeremy T. Fineman and Michael Mitzenmacher, editors, *2nd Symposium on Simplicity in Algorithms, SOSA 2019, January 8-9, 2019, San Diego, CA, USA*, volume 69 of *OASICS*, pages 17:1–17:6. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/OASIcs.SOSA.2019.17.

13 Haim Kaplan, Nira Shafrir, and Robert Endre Tarjan. Union-find with deletions. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*, pages 19–28. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545384.

14 Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972. doi:10.1007/978-1-4684-2001-2_9.

15 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

16 Konstantinos Koiliaris and Chao Xu. A faster pseudopolynomial time algorithm for subset sum. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 1062–1072. SIAM, 2017. doi:10.1137/1.9781611974782.68.

17 Konstantinos Koiliaris and Chao Xu. Faster pseudopolynomial time algorithms for subset sum. *ACM Trans. Algorithms*, 15(3):40:1–40:20, 2019. doi:10.1145/3329863.

18 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.