

Fast and Simple Modular Subset Sum

Kyriakos Axiotis

Massachusetts Institute of Technology

Arturs Backurs

Toyota Technological Institute at Chicago

Karl Bringmann

Saarland University and Max-Planck Institute for Informatics

Ce Jin

Institute for Interdisciplinary Information Sciences, Tsinghua University, China

Vasileios Nakos

Saarland University and Max-Planck Institute for Informatics

Christos Tzamos

University of Wisconsin-Madison

Hongxun Wu

Institute for Interdisciplinary Information Sciences, Tsinghua University, China

Abstract

We revisit the Subset Sum problem over the finite cyclic group \mathbb{Z}_m for some given integer m . A series of recent works has provided asymptotically optimal algorithms for this problem under the Strong Exponential Time Hypothesis. Koiliaris and Xu (SODA'17, TALG'19) gave a deterministic algorithm running in time $\tilde{O}(m^{5/4})$, which was later improved to $\mathcal{O}(m \log^7 m)$ randomized time by Axiotis et al. (SODA'19).

In this work, we present two simple algorithms for the Modular Subset Sum problem running in near-linear time in m , both efficiently implementing Bellman's iteration over \mathbb{Z}_m . The first one is a *randomized* algorithm running in time $\mathcal{O}(m \log^2 m)$, that is based solely on rolling hash and an elementary data-structure for prefix sums; to illustrate its simplicity we provide a short and efficient implementation of the algorithm in Python. Our second solution is a *deterministic* algorithm running in time $\mathcal{O}(m \text{polylog } m)$, that uses dynamic data structures for string manipulation.

We further show that the techniques developed in this work can also lead to simple algorithms for the All Pairs Non-Decreasing Paths Problem (APNP) on undirected graphs, matching the asymptotically optimal running time of $\tilde{O}(n^2)$ provided in the recent work of Duan et al. (ICALP'19).

2012 ACM Subject Classification Theory of computation → Algorithm design techniques

Keywords and phrases Modular Subset Sum, rolling hash, dynamic strings

Funding *Karl Bringmann and Vasileios Nakos*: This work is part of the project TIPEA that has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No. 850979).

1 Introduction

In the *Subset Sum* problem, one is given a multiset $X = \{x_1, x_2, \dots, x_n\}$ of integers along with an integer target t , and is asked to decide if there exists a subset of X that sums to the target t . In the *Modular Subset Sum* generalization of the problem, all sums are taken over the finite cyclic group \mathbb{Z}_m for some given integer m .

Subset Sum is a fundamental problem in Computer Science known to be NP-complete but only weakly as it admits pseudo-polynomial time algorithms. In particular, the Dynamic Programming algorithm of Bellman [7] solves the problem in $\mathcal{O}(nt)$ time. It works by iteratively computing all attainable subset sums when using only the first i integers. More

specifically, it starts with $S^0 = \{0\}$ and computes S^i as $S^{i-1} \cup (S^{i-1} + x_i)$, where $S^{i-1} + x_i = \{s + x_i \mid s \in S^{i-1}\}$.

The above algorithm can be straightforwardly applied to give an $\mathcal{O}(nm)$ time algorithm for the modular case. Recent work by Koiliaris and Xu [25] obtained an improved deterministic algorithm running in $\tilde{\mathcal{O}}(m^{5/4})$ that relies on structural results from number theory [21]. A follow up work by Axiotis et al. [4] presented a randomized algorithm that improves the running time to $\mathcal{O}(m \log^7 m)$ using ideas based on linear sketching. The obtained running time matches (up to subpolynomial factors) the conditional lower bound of Abboud et al. [1] based on the Strong Exponential Time Hypothesis which implies that no $\mathcal{O}(m^{1-\varepsilon})$ algorithms exist for any constant $\varepsilon > 0$. Another near-linear time algorithm follows from [9], where the authors show that given sets $A_1, A_2, \dots, A_n \subseteq \mathbb{Z}_m$, one can compute the sumset $A_1 + A_2 + \dots + A_n$ in near-linear output-sensitive time; Modular Subset Sum is a special case where all A_i are of size 2.

While prior work obtained near-optimal algorithms for Modular Subset Sum, the resulting algorithms are complex and their analysis is relatively involved. In this work, we present two simple near-optimal algorithms. Our simplest algorithm (see Section 2) is *randomized* and runs in time $\mathcal{O}(m \log^2 m)$. More precisely, the algorithm produces the whole set X^* of attainable subset sums of the multiset X in time $\mathcal{O}(|X^*| \log^2 m)$. The idea behind our algorithm is a fast implementation of Bellman’s iteration and requires only two elementary techniques, rolling hashing and a data structure for maintaining prefix sums. These techniques are already taught in undergraduate level algorithms classes. We believe that our simple algorithm can serve as an example application when these techniques are introduced.

Our second algorithm (see Section 3) is *deterministic* and solves Modular Subset Sum in time $\tilde{\mathcal{O}}(m) = \mathcal{O}(m \text{polylog } m)$. More precisely, the algorithm produces the set X^* of attainable subset sums in time¹ $\tilde{\mathcal{O}}(|X^*|) = \mathcal{O}(|X^*| \text{polylog } |X^*|)$. This algorithm is based on a classic data structure for string manipulation, and apart from this data structure the algorithm is simple. The idea of solving Modular Subset Sum via dynamic string data structures has already been suggested in [4], however, the algorithm proposed in [4] runs in time $\mathcal{O}(|X^*| \text{polylog } m)$, which we improve to $\mathcal{O}(|X^*| \text{polylog } |X^*|)$.

Techniques for the First Algorithm

We first explain the technical innovation behind our randomized $\mathcal{O}(m \log^2 m)$ algorithm (Theorem 1 in Section 2). At the core of our argument is a new method for computing the symmetric difference $S_1 \Delta S_2$ between two sets $S_1, S_2 \subseteq [m]$ in output-sensitive time upon specific updates on those two sets. The idea is to use hashing to compare the indicator vectors of the two sets. If the two hashes are the same, then the two sets are the same w.h.p. If not, we compute the symmetric difference of the sets S_1 and S_2 by recursing on the first and the second half of the universe, $\{1, \dots, \lceil m/2 \rceil\}$ and $\{\lceil m/2 \rceil + 1, \dots, m\}$. In total, at most $\log m + 1$ hashes need to be computed per element of $S_1 \Delta S_2$. Each hash that needs to be computed corresponds to a contiguous interval of the indicator vectors. It can be evaluated in $\mathcal{O}(\log m)$ time given access to a data structure that maintains prefix sums of a polynomial rolling hash function for the indicator vectors of each of the sets.

We show that this idea can be applied to other problems beyond Modular Subset Sum. In particular, we consider the problem of all-pairs non-decreasing paths (APNP) in undirected

¹ All our running time bounds assume that the usual arithmetic operations on $\log(m)$ -bit numbers can be performed in constant time.

graphs, where we obtain near-optimal algorithms improving the state-of-the-art for this problem, see Appendix A.

These two algorithms for Modular Subset Sum and APNP are simple to describe and to analyze. To illustrate their simplicity, we provide short but detailed implementations in Python for both algorithms in the appendix (see Appendix B and C).

Techniques for the Second Algorithm

Now let us describe our deterministic $\mathcal{O}(m \text{ polylog } m)$ algorithm (Theorem 2 in Section 3). The core of this algorithm is again a fast method for computing the symmetric difference $S_1 \Delta S_2$ for sets $S_1, S_2 \subseteq [m]$. Consider the indicator vectors of S_1 and S_2 and interpret them as length- m strings z_1, z_2 over alphabet $\{0, 1\}$. Then the symmetric difference $S_1 \Delta S_2$ corresponds to all positions at which the strings z_1, z_2 differ. We thus obtain the first element of the symmetric difference by computing the longest common prefix of z_1 and z_2 . Generalizing this idea, we can enumerate the symmetric difference using one longest common prefix query per output element. We implement such queries by using a classic data structure for dynamically maintaining a family of strings under concatenations, splits, and equality tests due to Mehlhorn et al. [27].

Implementing this idea naively leads to a running time of $\mathcal{O}(|X^*| \text{ polylog } m)$. By working on the run-length encoding of the strings z_1, z_2 , we further improve the running time to $\mathcal{O}(|X^*| \text{ polylog } |X^*|)$.

Further Related Work

In addition to Modular Subset Sum, there has recently been a lot of interest in obtaining faster algorithms for other related problems, like non-modular Subset Sum [8, 25, 24] and Knapsack [29, 10, 23, 6, 5, 22, 17], and providing conditional lower bounds [1, 12, 26].

2 Algorithm I: Rolling Hash and Polynomial Identity Testing

To describe our implementation, we consider Bellman's iteration² $S^i = S^{i-1} \cup (S^{i-1} + x_i)$. Our goal is to compute $X^* = S^n$, that is, the set of all attainable modular subset sums. Note that, if we could efficiently compute the new sums $C^i \triangleq (S^{i-1} + x_i) \setminus S^{i-1}$, we would be able to implement the Bellman iteration as $S^i = S^{i-1} \cup C^i$. We will shortly show that C^i can be computed in output-sensitive time $\mathcal{O}(|C^i| \log^2 m)$. This implies that the total time to evaluate $S^n = C^1 \cup \dots \cup C^n$ is $\mathcal{O}(|C^1| \log^2 m) + \dots + \mathcal{O}(|C^n| \log^2 m) \leq \mathcal{O}(|X^*| \log^2 m) \leq \mathcal{O}(m \log^2 m)$ since the sets C^i are disjoint and their union is of size $|X^*| \leq m$.

We now argue how to compute these new subset sums, $(S^{i-1} + x_i) \setminus S^{i-1}$, efficiently. Instead of considering the set difference between the sets $S^{i-1} + x_i$ and S^{i-1} , we will consider their symmetric difference $(S^{i-1} + x_i) \Delta S^{i-1} = C^i \cup D^i$, where $D^i = S^{i-1} \setminus (S^{i-1} + x_i)$. An important observation made in [4] is that since the sets S^{i-1} and $S^{i-1} + x_i$ have the same size, the symmetric difference will have size exactly $2|C^i|$ as $|C^i| = |D^i|$. Thus, recovering this larger set $C^i \cup D^i$ in output sensitive time is asymptotically the same as recovering only the elements of C^i . For notational convenience, we call elements of D^i "ghost sums" in the sense that they are not new subset sums.

² Here and in the remainder of this paper, we write $S + x := \{s + x \bmod m \mid s \in S\}$, for a given modulus m .

We now provide a recursive function (Algorithm 1) that given a set of integers S and integers a, b, x , computes $((S + x) \setminus S) \cap [a, b]$. Calling the function with $S = S^{i-1}$, $a = 0$, $b = m$ and $x = x_i$, we can recover C^i . We will show that the function outputs C^i in time $\mathcal{O}(|C^i| \log^2 m)$, which is what we need.

Algorithm 1 Find New Subset Sums in range $[a, b]$

```

1: function FIND-NEW-SUMS( $a, b, x, S$ )
2:   if  $(S + x) \cap [a, b] = S \cap [a, b]$  then return  $\emptyset$ 
3:   if  $b = a + 1$  then
4:     if  $a \in (S + x) \setminus S$  then return  $\{a\}$   $\triangleright a$  is a new subset sum
5:     else return  $\emptyset$   $\triangleright a \in S \setminus (S + x)$  is a ghost sum
6:   else return FIND-NEW-SUMS( $a, \lfloor \frac{a+b}{2} \rfloor, x$ )  $\cup$  FIND-NEW-SUMS( $\lfloor \frac{a+b}{2} \rfloor, b, x$ )

```

We implement the function efficiently by maintaining a data structure for the characteristic vector of the set S that allows efficient membership queries, updates and equality checks between different parts of the vector as required in line 2.

We interpret the set S as a characteristic vector and write $S_i = 1$ if $i \in S$ and $S_i = 0$ if $i \notin S$. To check that $(S + x) \cap [a, b] = S \cap [a, b]$, we need to check that the binary sequences $(S+x)_a, \dots, (S+x)_{b-1}$ and S_a, \dots, S_{b-1} are equal. To check the equality of the two sequences we will use polynomial identity testing. In particular, let r be a uniformly random integer from $\{0, \dots, p-1\}$ for a large enough prime p (which we will choose later). Then, with high probability, it is sufficient to check that $\sum_{i=a}^{b-1} (S+x)_i r^i = \sum_{i=a}^{b-1} S_i r^i \pmod{p}$ to conclude the equality of the sequences. The latter condition is equivalent to $r^x \sum_{i=a-x}^{b-x-1} S_i r^i = \sum_{i=a}^{b-1} S_i r^i \pmod{p}$, which is the same as $r^x (f(b-x) - f(a-x)) = f(b) - f(a) \pmod{p}$, where $f(t) \triangleq \sum_{i=0}^{t-1} S_i r^i \pmod{p}$ for all $t = 0, \dots, m$.

Correctness

To argue the correctness, we observe that for any two binary sequences $x, y \in \{0, 1\}^t$, prime p and a random integer $r \in \{0, \dots, p-1\}$ we have $\Pr[\sum_i x_i r^i = \sum_i y_i r^i \pmod{p}] \leq t/p$ if $x \neq y$ and $\Pr[\sum_i x_i r^i = \sum_i y_i r^i \pmod{p}] = 1$ if $x = y$. This is also known as the Rabin-Karp rolling hash function. Choosing $p = \Theta(m^2 \log(m)/\delta)$, suffices to have the algorithm fail with probability at most δ . This is because a single randomized comparison fails with probability at most $\frac{\delta}{m \log m}$ and by a union bound the probability that any of the $m \log m$ comparisons performed in the algorithm fails is at most δ . Assuming basic arithmetic operations between $\mathcal{O}(\log m)$ -bit numbers take constant time, we can choose $\delta = 1/\text{poly}(m)$ to obtain a high probability of success.

Running Time

We will show that the prefix sums $f(t) = \sum_{i=0}^{t-1} S_i r^i \pmod{p}$ can be evaluated in time $\mathcal{O}(\log n)$, which will lead to the required running time for computing C^i as we will see later. Additionally, we need that the data structure can update the characteristic vector of the set S in $\mathcal{O}(\log n)$ time (to be able to implement the Bellman iteration $S^i = S^{i-1} \cup C^i$ efficiently). These requirements can be abstracted as follows. We have a sequence of integers T_0, \dots, T_{m-1} and in each step we either want to compute the prefix sum $g(t) \triangleq \sum_{i=0}^{t-1} T_i$ for some integer $t \in \{1, \dots, m\}$ or we want to update an arbitrary integer T_i for some $i \in \{0, \dots, m-1\}$. Our goal is to implement the queries and updates in $\mathcal{O}(\log m)$ time.

This indeed can be done by a simple binary tree.³ Such a data structure implies that we can check the condition on line 2 in $\mathcal{O}(\log m)$ time. To bound the final running time, consider a particular position where S^{i-1} and $S^{i-1} + x_i$ differ. This position can cause the condition $(S + x) \cap [a, b) = S \cap [a, b)$ to fail (and the algorithm to proceed to line 3) at most $\mathcal{O}(\log m)$ times. Each time we spend $\mathcal{O}(\log m)$ time to check the condition and the total number of positions where S^{i-1} and $S^{i-1} + x_i$ differ is $2|C^i|$. In total, this implies that the function outputs C^i in time $\mathcal{O}(\log m) \cdot \mathcal{O}(\log m) \cdot 2|C^i| = \mathcal{O}(|C^i| \log^2 m)$, which is what we wanted. Finally, we observe that we can perform the Bellman iteration $S^i = S^{i-1} \cup C^i$ in $\mathcal{O}(|C^i| \log m)$ time.

Combining the above, we arrive at our first result.

► **Theorem 1.** *Modular Subset Sum can be solved in $\mathcal{O}(m \log^2 m)$ time with high probability.*

A sample implementation in Python is given in Appendix B. It uses a simple and efficient implementation of binary trees for maintaining prefix sums [18].

3 Algorithm II: Dynamic Strings

This section is devoted to the second algorithm for Modular Subset Sum. In particular, we prove the following theorem.

► **Theorem 2.** *Modular Subset Sum can be solved by a deterministic algorithm in time $\mathcal{O}(|X^*| \text{polylog } |X^*|)$, where X^* denotes the set of attainable subset sums of X modulo m .*

We first set up the necessary notation on strings. A string z of length $|z|$ is a sequence of letters from alphabet Σ referred to as $z[0], \dots, z[|z| - 1]$. By $z[i..j]$ we denote the substring from letter $z[i]$ up to letter $z[j]$. We write $z[..j]$ as shorthand for $z[0..j]$ and similarly $z[i..]$ for $z[i..|z| - 1]$.

3.1 Data Structure for Dynamic Strings

We start by reviewing a classic tool in string algorithms. This is a data structure for efficiently maintaining a family \mathcal{F} of strings over alphabet Σ under the following update operations.

- **ADDSTRING(c):** Given a letter $c \in \Sigma$, this operation adds the 1-letter string c to \mathcal{F} .
- **CONCATENATE(s, s'):** Given strings $s, s' \in \mathcal{F}$, concatenate them and add the resulting string to \mathcal{F} . The two strings s, s' remain in \mathcal{F} .
- **SPLIT(s, i):** Given a string $s \in \mathcal{F}$ and a number i , split s into two strings $s[..i - 1]$ and $s[i..]$ and add these strings to \mathcal{F} . The string s remains in \mathcal{F} .
- **EQUAL(s, s'):** Given strings $s, s' \in \mathcal{F}$, return true if $s = s'$.

Note that no string is ever removed from \mathcal{F} . Mehlhorn et al. [27] were the first to design a data structure supporting these operations in polylogarithmic time. Their time bounds have been further improved [2, 3, 20], but since we will ignore logarithmic factors we shall not make use of those improvements.

► **Theorem 3** ([27]). *There is a deterministic data structure for maintaining a family of strings under the operations ADDSTRING, CONCATENATE, SPLIT, and EQUAL such that any sequence of k operations resulting in total size $N = \sum_{s \in \mathcal{F}} |s|$ runs in time $\mathcal{O}(k \text{polylog}(kN))$.*

³ The bounds are known to be tight up to a $\log \log m$ factor in the cell-probe model [28]. In particular, if the query (update) time is $\log^{\mathcal{O}(1)} m$, then the update (query) time is $\Omega(\log(m)/\log \log m)$.

The data structure even works for very large alphabet Σ , as long as Σ is ordered and we can compare any two letters in time $\mathcal{O}(1)$.

We observe that as an application of the above we obtain the following data structure.

► **Lemma 4.** *There is a deterministic data structure that maintains a length- m string z over alphabet $\{0, 1\}$, initialized as $z = 0^m$, under the following operations, where any sequence of k operations runs in time $\mathcal{O}(k \text{ polylog}(km))$:*

- **ADD(i):** *Given $0 \leq i < m$, set $z[i] := 1$,*
- **LCP(i, j):** *Return the length of the longest common prefix of $z[i..]$ and $z[j..]$.*

Proof. For the initialization of $z = 0^m$, we first run **ADDSTRING**(0) and then, using $\mathcal{O}(\log m)$ concatenations, we generate the strings 0^{2^i} and we combine them according to the binary representation of m to obtain the string 0^m .

For **ADD**(i), we split z at i and at $i + 1$ to obtain the strings $z[..i - 1]$ and $z[i + 1..]$. We then run **ADDSTRING**(1), and finally we concatenate twice to obtain the resulting string $z' = \text{CONCATENATE}(\text{CONCATENATE}(z[..i - 1], 1), z[i + 1..])$.

For a longest common prefix query **LCP**(i, j), we first split z at i and at j to obtain the strings $y_1 := z[i..]$ and $y_2 := z[j..]$. Then we perform a binary search for the largest ℓ such that $y_1[.. \ell] = y_2[.. \ell]$. Each step of the binary search uses two splits, to construct the strings $y_1[.. \ell]$ and $y_2[.. \ell]$, and one equality test.

Hence, we can simulate k operations among **ADD** and **LCP** using $\mathcal{O}(k \log m)$ operations among **EQUAL**, **ADDSTRING**, **CONCATENATE**, and **SPLIT**. The total string length of the constructed family \mathcal{F} is $N = \mathcal{O}(mk \log m)$, and thus the total time is $\mathcal{O}(k \log m \text{ polylog}(kN)) = \mathcal{O}(k \text{ polylog}(km))$. ◀

3.2 From Dynamic Strings to Modular Subset Sum

Recall that given $X = \{x_1, \dots, x_n\} \subseteq \mathbb{Z}_m$ our aim is to compute the set $X^* \subseteq \mathbb{Z}_m$ consisting of all subset sums of X . As in our first algorithm for Modular Subset Sum, we follow Bellman's approach by initializing $S^0 := \{0\}$ and iteratively computing $S^i := (S^{i-1} + x_i) \cup S^{i-1}$ for $i = 1, \dots, n$. As we have seen in the first algorithm, it suffices to compute the symmetric difference $E^i := (S^{i-1} + x_i) \Delta S^{i-1}$ in time $\mathcal{O}(|E^i| \text{ polylog } m)$; then over all iterations we compute $S^n = X^*$ in time $\mathcal{O}(|X^*| \text{ polylog } m)$.

It remains to show how to compute the symmetric difference E^i in each iteration. To this end, let z be the indicator vector of S^i copied twice, that is, z is a string of length $2m$ over alphabet $\{0, 1\}$ where $z[j]$ indicates whether $j \bmod m$ is in S^i , for any $0 \leq j < 2m$. We maintain the data structure from Lemma 4 for the string z . Since this data structure initializes z as 0^{2m} , we call **ADD**(0) and **ADD**(m) to initialize z correctly according to $S^0 = \{0\}$.

At the beginning of the i -th iteration, note that $z[m - x_i .. 2m - x_i]$ is the indicator vector of $S^{i-1} + x_i$. Therefore, the query **LCP**(0, $m - x_i$) yields a number d' such that $d := d' + 1$ is minimal with $z[d] \neq z[m - x_i + d]$. In other words, d is the smallest element of the symmetric difference E^i of S^{i-1} and $S^{i-1} + x_i$ (unless $d \geq m$, in which case we have that E^i is the empty set). We find the next element of E^i by calling **LCP**($d + 1, m - x_i + d + 1$). Repeating this argument, we compute the set E^i using $\mathcal{O}(|E^i|)$ **LCP** operations. This finishes the description of how to compute the symmetric difference E^i . We maintain the string z for the next iteration by setting $z[d] = z[d + m] = 1$ for each $d \in E^i$ with $d \notin S^{i-1}$. This uses $\mathcal{O}(|E^i|)$ **ADD** operations.

In total, we run $\mathcal{O}(|X^*|)$ **LCP** and **ADD** operations on a string of length $2m$. This takes total time $\mathcal{O}(|X^*| \text{ polylog}(|X^*| m)) = \mathcal{O}(|X^*| \text{ polylog } m)$ according to Lemma 4. In

particular, this running time is bounded by $\tilde{\mathcal{O}}(m)$. We further improve this running time in Section 3.4 below. For pseudocode see Algorithm 2.

Algorithm 2 Algorithm for Modular Subset Sum using dynamic strings.

```

1: function MODULARSUBSETSUMVIADYNAMICSTRINGS( $X, m$ )
2:    $S := \{0\}$ 
3:   Initialize  $z = 0^{2m}$  (as in Lemma 4)
4:    $z.\text{ADD}(0)$ 
5:    $z.\text{ADD}(m)$ 
6:   for  $i = 1, \dots, n$  do
7:      $E^i := \emptyset$ 
8:      $d := 1 + z.\text{LCP}(0, m - x_i)$ 
9:     while  $d < m$  do
10:       $E^i := E^i \cup \{d\}$ 
11:       $d := d + 1 + z.\text{LCP}(d + 1, m - x_i + d + 1)$ 
12:     for each  $d \in E^i$  do
13:       if  $d \notin S$  then
14:          $S := S \cup \{d\}$ 
15:          $z.\text{ADD}(d)$ 
16:          $z.\text{ADD}(d + m)$ 
17:   return  $S$ 

```

3.3 Solution Reconstruction

In order to reconstruct a subset $Y \subseteq X$ summing to a given target t , we augment the above algorithm as follows. We store the set S^i in a balanced binary search tree \mathcal{T}^i . For each number $d \in S^i \setminus S^{i-1}$, in the node corresponding to d in \mathcal{T}^i we store a pointer to the node corresponding to $d - x_i$. At the end of the algorithm \mathcal{T}^n stores $S^n = X^*$, the set of all subset sums of X . Note that computing \mathcal{T}^n augmented by these pointers takes total time $\mathcal{O}(|X^*| \log |X^*|)$ and thus does not increase the asymptotic running time of the algorithm.

With this bookkeeping, given any target integer $t \in \mathbb{Z}_m$ we first search for t in \mathcal{T}^n to check whether $t \in X^*$. If $t \in X^*$, then starting from the node corresponding to t in \mathcal{T}^n , we follow the stored pointers to reconstruct a subset $Y \subseteq X$ summing to t modulo m . The total running time of this solution reconstruction is $\mathcal{O}(|Y| + \log |X^*|)$.

Note that this solution reconstruction method visits any node of \mathcal{T}^n at most once, which implies $|Y| \leq |X^*| \leq m$. This is essentially the only control we have over the size $|Y|$, in particular we do not guarantee Y to be a smallest subset summing to t .

3.4 Improving the Running Time

We now improve the running time from $\mathcal{O}(|X^*| \text{polylog } m)$ to $\tilde{\mathcal{O}}(|X^*|) = \mathcal{O}(|X^*| \text{polylog } |X^*|)$, finishing the proof of Theorem 2. Observe that all steps of the algorithm (including the solution reconstruction) run in time $\tilde{\mathcal{O}}(|X^*|)$, except for Lemma 4. Hence, it suffices to replace this lemma by the following improved variant, which makes use of run-length encoding.

► **Lemma 5.** *There is a deterministic data structure that can initialize $z = 0^m$ and perform k ADD and LCP operations in total time $\mathcal{O}(k \text{polylog } k)$.*

Proof. Recall that we assume that arithmetic operations on $\mathcal{O}(\log m)$ -bit numbers can be performed in time $\mathcal{O}(1)$. In particular, the string length m can be processed in time $\mathcal{O}(1)$.

Denote by $S \subseteq \mathbb{Z}_m$ the set of which z is the indicator vector. That is, initially we have $S = \emptyset$ and on operation $\text{ADD}(i)$ we update $S := S \cup \{i\}$. We store S in a balanced binary search tree \mathcal{T} . We also augment \mathcal{T} to store at each node the size of its subtree. This allows us to perform the following queries in time $\mathcal{O}(\log |S|)$:

- *Rank:* Given a number v , determine the number of keys stored in \mathcal{T} that are smaller than v ,
- *Select:* Given a number v , determine the v -th number stored in \mathcal{T} (in sorted order).

Note that \mathcal{T} can be updated in time $\mathcal{O}(\log |S|)$ per operation. The total time for maintaining \mathcal{T} during k ADD and LCP operations is $\mathcal{O}(k \log k)$, since $|S| \leq k$.

We compress the string z by replacing each run of 0's by one symbol. Specifically, let $\Sigma := \{1\} \cup \{(0, L) \mid 0 \leq L \leq m\}$. Note that symbols in Σ can be read and compared in time $\mathcal{O}(1)$. We convert string $z \in \{0, 1\}^m$ to a string $C(z) \in \Sigma^*$ by replacing each maximal substring 0^L of z by the symbol $(0, L)$. For simplicity, we also add the symbol $(0, 0)$ between any two consecutive 1's in z . For example, the string $z = 10001100$ is converted to $C(z) = 1(0, 3)1(0, 0)1(0, 2)$. We use the data structure of Theorem 3 to store $C(z)$. We maintain $C(z)$ using the binary search tree \mathcal{T} , by implementing initialization, ADD , and LCP as follows.

Initialization. Given m , we initialize $z = 0^m$ and thus $C(z) = (0, m)$. This string is generated by calling $\text{ADDSTRING}(c)$ for $c = (0, m) \in \Sigma$, which takes time $\mathcal{O}(1)$.

Add. Given i , we want to set $z[i] := 1$. We denote by $a < i < b$ the predecessor and successor of i in S , so that $z[a..b] = 10^{b-a-1}1$. Note that a and b can be computed from \mathcal{T} . Using a rank query on i , we can infer the corresponding position h with $C(z)[h-1..h+1] = 1(0, b-a-1)1$. We split $C(z)$ at h and at $h+1$ to obtain the strings $C(z)[..h-1]$ and $C(z)[h+1..]$. We then construct the string $(0, i-a-1)1(0, b-i-1)$ using ADDSTRING thrice and CONCATENATE twice. Finally, we concatenate $C(z)[..h-1]$ and $(0, i-a-1)1(0, b-i-1)$ and $C(z)[h+1..]$ to form the new string $C(z)$ after setting $z[i] := 1$.

LCP. Given i, j , let $y_1 := z[i..]$ and $y_2 := z[j..]$. We first construct the strings $C(y_1)$ and $C(y_2)$. This is similar to the last paragraph: Denote the predecessor and successor of i by $a < i \leq b$, so that $z[a..b] = 10^{b-a-1}1$. Using a rank query, we find the corresponding position h with $C(z)[h-1..h+1] = 1(0, b-a-1)1$. Splitting $C(z)$ at $h+1$ and concatenating it after $(0, b-i-1)$ yields $C(y_1)$. (If $b-i-1 = 0$ then we remove the initial $(0, 0) = (0, b-i-1)$.) We similarly generate $C(y_2)$. We now perform a binary search for the largest ℓ such that $C(y_1)[..\ell] = C(y_2)[..\ell]$, using two SPLIT and one EQUAL operation per binary search step. We use a rank and a select query to determine the length Δ of the string corresponding to $C(y_1)[..\ell]$, that is, $C(z[i..i+\Delta-1]) = C(y_1)[..\ell]$. If $C(y_1)[\ell+1] = 1$ or $C(y_2)[\ell+1] = 1$ or one of these symbols is undefined (i.e., out of bounds) then $\text{LCP}(i, j) = \Delta + 1$. Otherwise, we have $C(y_1)[\ell+1] = (0, L_1)$ and $C(y_2)[\ell+1] = (0, L_2)$, and then $\text{LCP}(i, j) = \Delta + \min\{L_1, L_2\} + 1$.

Hence, we can simulate k operations among ADD and LCP using $\mathcal{O}(k \log k)$ operations among EQUAL , ADDSTRING , CONCATENATE , and SPLIT . The total string length of the constructed family \mathcal{F} is $N = \mathcal{O}(k^2 \log k)$, since after k operations each constructed string has at most k 1's and thus has length $\mathcal{O}(k)$. By Theorem 3, the total time is $\mathcal{O}(k \text{polylog}(kN)) = \mathcal{O}(k \text{polylog } k)$. ◀

References

- 1 Amir Abboud, Karl Bringmann, Danny Hermelin, and Dvir Shabtay. SETH-based lower bounds for subset sum and bicriteria path. In *SODA*, pages 41–57. SIAM, 2019.
- 2 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Dynamic pattern matching. Technical report, Department of Computer Science, University of Copenhagen, 1998. DIKU Report 98/27, 16 pages, <http://cs.au.dk/~gerth/papers/diku-98-27.pdf>.
- 3 Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *SODA*, pages 819–828. ACM/SIAM, 2000.
- 4 Kyriakos Axiotis, Arturs Backurs, Ce Jin, Christos Tzamos, and Hongxun Wu. Fast modular subset sum using linear sketching. In *SODA*, pages 58–69. SIAM, 2019.
- 5 Kyriakos Axiotis and Christos Tzamos. Capacitated dynamic programming: Faster knapsack and graph algorithms. In *ICALP*, volume 132 of *LIPICs*, pages 19:1–19:13, 2019.
- 6 MohammadHossein Bateni, MohammadTaghi Hajiaghayi, Saeed Seddighin, and Cliff Stein. Fast algorithms for knapsack via convolution and prediction. In *STOC*, pages 1269–1282. ACM, 2018.
- 7 Richard E. Bellman. Dynamic programming. 1957.
- 8 Karl Bringmann. A near-linear pseudopolynomial time algorithm for subset sum. In *SODA*, pages 1073–1084. SIAM, 2017.
- 9 Karl Bringmann and Vasileios Nakos. Fast n -fold Boolean Convolution via Additive Combinatorics
- 10 Timothy M. Chan. Approximation schemes for 0-1 knapsack. In *SOSA@SODA*, volume 61 of *OASICS*, pages 5:1–5:12, 2018.
- 11 Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.
- 12 Marek Cygan, Marcin Mucha, Karol Wegrzycki, and Michal Włodarczyk. On problems equivalent to $(\min, +)$ -convolution. *ACM Trans. Algorithms*, 15(1):14:1–14:25, 2019.
- 13 Ran Duan, Yong Gu, and Le Zhang. Improved time bounds for all pairs non-decreasing paths in general digraphs. In *ICALP*, volume 107 of *LIPICs*, pages 44:1–44:14, 2018.
- 14 Ran Duan, Ce Jin, and Hongxun Wu. Faster algorithms for all pairs non-decreasing paths problem. In *ICALP*, volume 132 of *LIPICs*, pages 48:1–48:13, 2019.
- 15 Ran Duan, Ce Jin, and Hongxun Wu. Faster algorithms for all pairs non-decreasing paths problem. *arXiv preprint arXiv:1904.10701*, 2019.
- 16 Ran Duan and Seth Pettie. Fast algorithms for (\max, \min) -matrix multiplication and bottleneck shortest paths. In *SODA*, pages 384–391. SIAM, 2009.
- 17 Friedrich Eisenbrand and Robert Weismantel. Proximity results and faster algorithms for integer programming using the Steinitz lemma. In *SODA*, pages 808–816. SIAM, 2018.
- 18 Peter M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.
- 19 François Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303. ACM, 2014.
- 20 Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In *SODA*, pages 1509–1528. SIAM, 2018.
- 21 Yahya Ould Hamidoune, Anna S Lladó, and Oriol Serra. On complete subsets of the cyclic group. *Journal of Combinatorial Theory, Series A*, 115(7):1279–1285, 2008.
- 22 Klaus Jansen and Lars Rohwedder. On integer programming and convolution. In *ITCS*, volume 124 of *LIPICs*, pages 43:1–43:17, 2019.
- 23 Ce Jin. An improved FPTAS for 0-1 knapsack. In *ICALP*, volume 132 of *LIPICs*, pages 76:1–76:14, 2019.
- 24 Ce Jin and Hongxun Wu. A simple near-linear pseudopolynomial time randomized algorithm for subset sum. In *SOSA@SODA*, volume 69 of *OASICS*, pages 17:1–17:6, 2019.
- 25 Konstantinos Koiliaris and Chao Xu. Faster pseudopolynomial time algorithms for subset sum. *ACM Trans. Algorithms*, 15(3):40:1–40:20, 2019.

- 26 Marvin Künnemann, Ramamohan Paturi, and Stefan Schneider. On the fine-grained complexity of one-dimensional dynamic programming. In *ICALP*, volume 80 of *LIPICs*, pages 21:1–21:15, 2017.
- 27 Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.
- 28 Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006.
- 29 Donguk Rhee. Faster fully polynomial approximation schemes for knapsack problems, 2015. Masterthesis, Massachusetts Institute of Technology.
- 30 Virginia Vassilevska, Ryan Williams, and Raphael Yuster. All pairs bottleneck paths and max-min matrix products in truly subcubic time. *Theory Comput.*, 5(1):173–189, 2009.
- 31 Virginia Vassilevska Williams. Nondecreasing paths in a weighted graph or: How to optimally read a train schedule. *ACM Trans. Algorithms*, 6(4):70:1–70:24, 2010.
- 32 Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, pages 887–898. ACM, 2012.

A Simple and Fast Algorithm for All-Pairs Non-Decreasing Paths

In the APNP problem, given an edge-weighted graph, the goal is to compute for any pair of nodes a and b the minimum cost of a path from a to b that uses non-decreasing edge-weights. The cost of such a path is defined to be the largest edge-weight encountered on the path.

There has been a number of works that sequentially improved the running time for the directed and undirected case of APNP [31, 13, 14]. The directed case is a generalization of the max-min matrix product [30, 16] and the best known algorithm for both problems runs in time $\tilde{\mathcal{O}}(n^{(\omega+3)/2})$, where ω is the exponent of fast matrix multiplication [11, 32, 19]. In contrast, the undirected case is known to be solvable in $\tilde{\mathcal{O}}(n^2)$ time [14].

We show how to solve the undirected APNP problem by a simple algorithm in time $\mathcal{O}(n^2 \log n)$. This improves the previously best result in terms of log-factors, and it is asymptotically optimal up to a single log-factor. For simplicity, in the following we call the undirected case of APNP simply APNP. In this section we prove the following theorem.

► **Theorem 6.** *All-Pairs Non-Decreasing Paths can be solved in $\mathcal{O}(n^2 \log n)$ time w.h.p.*

Let $G = (V, E)$ be an undirected graph with $n = |V|$ nodes and $m = \mathcal{O}(n^2)$ edges having edge weights $w(e)$ for $e \in E$. A path is a sequence of edges e_1, e_2, \dots, e_ℓ , such that e_i, e_{i+1} share an endpoint for all $1 \leq i \leq \ell - 1$. A non-decreasing path is a path satisfying $w(e_i) \leq w(e_{i+1})$ for all $1 \leq i \leq \ell - 1$. The weight of this non-decreasing path is defined to be $w(e_\ell)$, the weight of the last edge. The All Pairs Non-Decreasing Paths Problem (APNP) asks to determine the minimum weight non-decreasing path between every pair of vertices.

For simplicity, we focus on the strictly increasing version of the problem where there are no edges of equal weight. The general case can be converted to the distinct weights case (see Lemma 23 in Duan et al. [15]), through a very simple reduction. The reduction looks for connected components formed by edges of the same weight and replaces these edges with new ones with distinct weights. This preprocessing step runs in $\mathcal{O}(n^2)$ time and the number of edges in the new graph at most doubles. It thus suffices to focus on the distinct weights case.

The algorithm starts by ordering all edges of the graph from the smallest weight to the largest and inspecting the edges in this order. For every vertex u of the graph we maintain a set of vertices v that can be reached from u by a non-decreasing path using only the edges that have been inspected so far. Initially the sets for all vertices are empty. The first time a

vertex v is added to a list corresponding to a vertex u determines the cost of minimum non-decreasing path from u to v . In particular, if the vertex v is added to the list corresponding to the vertex u in the phase when we are inspecting edge e , the weight of the minimum non-decreasing path from u to v is equal to the weight $w(e)$ of the edge e . Let C_e be the set of newly discovered reachability pairs added in the phase when inspecting the edge e . We will shortly describe how we can compute C_e in an output-sensitive time $\mathcal{O}(|C_e| \log n)$. This implies that the total running time is upper bounded by $\mathcal{O}(n^2 \log n)$ since the total number of node pairs is upper bounded by n^2 and each pair is discovered at most once.

Now we describe how to compute C_e in $\mathcal{O}(|C_e| \log n)$ time. Let $e = (a, b)$. Let u be a vertex that can reach a but cannot reach b only using the edges inspected so far (not including e) via a non-decreasing path. We observe that, by adding the edge $e = (a, b)$, the vertex u can now reach vertex b (by first going to a and then traversing the edge e). Similarly, if u can reach b but cannot reach a , after adding e , u can reach a . On the other hand, if u can reach both a and b (or cannot reach both), no new edges will be added from u after inspecting edge e . Let R^a be the set of vertices u that can reach a but cannot reach b (right before inspecting e), and R^b be the set of vertices that can reach b but not a . We conclude that $C_e = ((R^a \setminus R^b) \times \{b\}) \cup ((R^b \setminus R^a) \times \{a\})$. Therefore it is sufficient to be able to compute $R^a \setminus R^b$ and $R^b \setminus R^a$ in $\mathcal{O}(|C_e| \log n)$ time. If we spend $\mathcal{O}(\log n)$ time per single vertex from one of these two sets, we obtain the required running time. We use a similar idea as we used for Modular Subset Sum. Let $R_i^a = 1$ if the i -th vertex of the graph belongs to R^a and $R_i^a = 0$ otherwise. For a random integer $r \in \{0, \dots, p-1\}$ (for a large enough prime p) we build a tree data structure that stores partial sums of the sequence $R_0^a \cdot r^0, R_1^a \cdot r^1, R_2^a \cdot r^2, \dots$ in its internal leaves. In particular, we associate the leaves of a complete binary tree with the elements of the sequence and each node recursively stores the sum of values of its children. We can update an element of the sequence by spending $\mathcal{O}(\log n)$ time on the data structure. Furthermore, if we have data structures for R^a and R^b , we can recursively inspect subtrees (whose hash values disagree) of the two data structures to find all elements from $R^a \setminus R^b$ and $R^b \setminus R^a$. The time spent to find one element is $\mathcal{O}(\log n)$. Thus, if we store such a data structure for each vertex of the graph, we can update them efficiently and compute C_e in time $\mathcal{O}(|C_e| \log n)$ for any edge e .

A sample implementation in Python is given in Appendix C.

B Python Implementation of Modular Subset Sum

Below we present a simple implementation of our first algorithm for Modular Subset Sum (Theorem 1) in Python.⁴ It maintains a binary indexed tree that keeps track of the prefix sums of polynomial hashes of the characteristic vector of the attainable subsets. To easily deal with rollover due to the cyclicity of the mod operation, a separate copy of the characteristic vector is kept translated by m .

It takes as an input a list of numbers W and the modulus m , and returns a list of length m , where the entry at position s is None if s is not a possible subset sum of W modulo m , or contains the last number from W that was added to create the subset sum s .

```
import random

def ModularSubsetSum(W, m):
    p = 1234567891 #large prime p > m^2 log m
```

⁴ The code can be also found at <https://ideone.com/YLLwMQ>.

12 Fast and Simple Modular Subset Sum

```
r = random.randint(0,p) #random number r in [0,p)
powr = [1] #Precompute powers of r
for i in range(2*m): #powr[i] ≐ ri (mod p)
    powr.append((powr[-1] * r) % p)

#Binary Indexed Tree for prefix sums
tree = [0] * (2*m)
def read(i): #Prefix sum of [0,i)
    if i<=0: return 0
    return tree[i-1] + read(i-(i&-i))
def update(i, v): #add v to position i
    while i < len(tree):
        tree[i] += v
        i += (i+1)&-(i+1)

#Functions for finding new subset sums and adding them
def FindNewSums(a,b,w):
    h1 = (read(b)-read(a))*powr[m-w] % p #hash of S ∩ [a,b)
    h2 = (read(b+m-w)-read(a+m-w)) % p #hash of (S+w) ∩ [a,b)
    if h1 == h2: return []
    if b == a+1:
        if sums[a] is None: return [a] #a is a new sum
        return [] #a is a ghost sum
    return FindNewSums(a,(a+b)//2,w) + FindNewSums((a+b)//2,b,w)
def AddNewSum(s, w):
    sums[s] = w
    update(s,powr[s]), update(s+m,powr[s+m])

#Main routine for computing subset sums
sums = [None] * m
AddNewSum(0,0)
for w in W:
    for s in FindNewSums(0,m,w):
        AddNewSum(s,w)

return sums
```

Example

Find all modular subset sums mod 8 with numbers 1, 3 and 6:

```
ModularSubsetSum([1,3,6], 8) #Returns [0, 1, 6, 3, 3, None, 6, 6]
```

Recovering the subset

To recover the subset making a particular subset sum, we repeatedly subtract the last number added in the subset sum s until we get down to 0.

```
def RecoverSubset(sums, s):
    if sums[s] is None: return None
    if s <= 0: return []
    return RecoverSubset(sums, (s-sums[s]) % len(sums)) + [sums[s]]

sums = ModularSubsetSum([1,3,6], 8)
RecoverSubset(sums, 7) #Returns [1, 6]
RecoverSubset(sums, 2) #Returns [1, 3, 6]
```

C Python Implementation of All-Pairs Non-Decreasing Paths

Below we present a simple implementation of our algorithm in Python for computing minimum weight non-decreasing path between all pairs of n vertices.⁵ It takes as an input a list E of edges of the graph in increasing order of their weights and the number n of vertices. Note that the actual weights of the edges do not matter besides their relative order. The algorithm returns an $n \times n$ matrix `path`. `path[u, v] = None` if there is no way to reach v from u by traversing edges with increasing weights. Otherwise `path[u, v] = par`, where `par` is the previous vertex on the minimum weight non-decreasing path from u to v . For every vertex of the graph the algorithm keeps track of partial hashes of vertices that can reach this vertex in a tree data structure.

```
import random

def AllPairsNonDecreasingPaths(E, n):
    p = 1234567891
    r = random.randint(0, p)
    powr = [1]
    for i in range(n):
        powr.append((powr[-1] * r) % p)

    N = 1 << (n-1).bit_length()
    tree = [ [0]*(2*N) for _ in range(n) ]
    def update(v, node, val):
        while node > 0:
            tree[v][node] += val
            node >>= 1

    #Functions for finding new paths and adding them
    def FindNewPaths(a, b, node):
        if tree[a][node] == tree[b][node]: return []
        if node >= N:
            u = node - N
            if path[u][a] is None:
                return [(u, a, b)]
            return [(u, b, a)]
        return FindNewPaths(a, b, 2*node) + FindNewPaths(a, b, 2*node+1)
    def AddNewPath(u, v, par):
        path[u][v] = par
        update(v, u+N, powr[u])

    #Main routine for finding all pairs non-decreasing paths
    path = [ [None] * n for _ in range(n) ]
    for i in range(n):
        AddNewPath(i, i, i)
    for (a, b) in E:
        for (u, v, par) in FindNewPaths(a, b, 1):
            AddNewPath(u, v, par)

    return path
```

Example

Find all pairs non-decreasing paths in a graph with n nodes and edges (1, 2) and (0, 1):

```
AllPairsNonDecreasingPaths([(1,2),(0,1)], 3)
#Returns [[0, 0, None], [1, 1, 1], [1, 2, 2]]
```

⁵ The code can be also found at <https://ideone.com/S9RAhX>.

Recovering the path between two vertices

To recover a specific path between two vertices, we repeatedly move to the last node visited before the destination until we reach the source.

```
def RecoverPath(path,u,v):
    if path[u][v] is None: return None
    if u == v: return [u]
    return RecoverPath(path, u, path[u][v]) + [ v ]

path = AllPairsNonDecreasingPaths([(1,2),(0,1)], 3)
RecoverPath(path, 2, 0) #Returns [2,1,0]
RecoverPath(path, 0, 2) #Returns None
```