Efficient reductions and algorithms for Subset Product

Pranjal Dutta[1], Mahesh Sreekumar Rajasree[2]

[1] Chennai Mathematical Institute, India pranjal@cmi.ac.in
[2] Indian Institute of Technology, Kanpur, India mahesr@cse.iitk.ac.in

Abstract. Given positive integers $a_1, \ldots, a_n$ and a target integer t, the Subset Product problem asks to determine whether there exists a subset $S \subseteq [n]$ such that $\prod_{i \in S} a_i = t$. It differs from the Subset Sum problem where the multiplication operation is replaced by addition. There is a pseudopolynomial-time dynamic programming algorithm which solves the Subset Product in $O(nt)$ time and $\Omega(t)$ space.
In this paper, we present a simple and elegant randomized algorithm for Subset Product in $\tilde{O}(n + t^{o(1)})$ expected-time. Moreover, we also present a poly(nt) time and $O(\log^2(nt))$ space deterministic algorithm.
In fact, we solve a more general problem called the SimulSubsetSum. This problem was introduced by Kane 2010. Given k instances of Subset Sum, it asks to decide whether there is a 'common' solution to all the instances. Kane gave a logspace algorithm for this problem. We show a polynomial-time reduction from Subset Product to SimulSubsetSum and also give efficient algorithm for the latter. Our algorithms use multivariate FFT, power series and number-theoretic techniques, introduced by Jin and Wu (SOSA 2019) and Kane (2010).

Keywords: simultaneous · power series · subset product · logspace · FFT · pseudo-prime-factor

1 Introduction

The Subset Sum problem (in short, SSUM) is a well-known NP-complete problem [19, p. 226], where given $(a_1, \ldots, a_n, t) \in \mathbb{Z}_{\geq 0}^{n+1}$, the problem is to decide whether there exists $S \subseteq [n]$ such that $\sum_{i \in S} a_i = t$. In the recent years, this problem has gained significant attractions due to applications in provable-secure cryptosystems [20,11] and remarkable algorithmic improvements both in classical and quantum world [5,15,14,10,3,13,9]. In this paper, we study a well-known variant of the subset sum, called Subset Product.

*Problem 1 (*Subset Product*).* Given $(a_1, \ldots, a_n, t) \in \mathbb{Z}_{\geq 1}^{n+1}$, the Subset Product problem asks to decide whether there exists an $S \subseteq [n]$ such that $\prod_{i \in S} a_i = t$.

Subset Product is known to be NP-complete [12, p. 221]. It has a trivial $O(nt)$ time (*pseudo-polynomial* time) dynamic programming algorithm which requires $\Omega(t)$ space [2].

Subset Product has been studied and applied in many different forms. For e.g. 1) constructing a *smooth hash* (VSH) by Contini et al. [6], 2) attack on the Naccache-Stern Knapsack (NSK) public key cryptosystem [8]. Similar problem has also been studied in optimization, in the form of product knapsack problem [22], multiobjective knapsack problem [1].

Next, we define a seemingly unrelated problem. It asks to decide whether there is a 'common' solution to the given many instances of subset sum. This was first introduced by [16, Section 3.3] (but no formal name was given).

Problem 2 (SimulSubsetSum). Given subset sum instances $(a_{1j}, \ldots, a_{nj}, t_j) \in \mathbb{Z}_{\geq 0}^{n+1}$, for $j \in [k]$, where k is some parameter, the Simultaneous Subset Sum problem (in short, SimulSubsetSum) asks to decide whether there exists an $S \subseteq [n]$ such that $\sum_{i \in S} a_{ij} = t_j, \forall j \in [k]$.

Remarks. 1. When k is fixed parameter (independent of n), we call this $k-$SimulSubsetSum. There is a trivial $O(n(t_1+1)\ldots(t_k+1))$ time deterministic algorithm for the SimulSubsetSum problem with k subset sum instances (k not necessarily a constant) by extending the dynamic programming algorithm for SSUM.

2. It suffices to work with $t_j \geq 1, \forall j \in [k]$. To argue that, let us assume that $t_j = 0$ for some $j \in [k]$ and $I_j := \{i \in [n] \,|\, a_{ij} = 0\}$. Observe that if SimulSubsetSum has a solution set $S \subseteq [n]$, then $S \subseteq I_j$. Therefore, for every $\ell \in [k]$, instead of looking at $(a_{1\ell}, \ldots, a_{n\ell}, t_\ell)$, it suffices to work with $\{a_{i,\ell} \,|\, i \in I_j\}$ with the target t_ℓ. Thus, we can trivially ignore the j^{th} SSUM instance.

Hardness depends on k. Linear algebraically, Problem 2 is asking to solve a system of k-linear equations, in n-variables with 0/1 constraints on the variables. If we assume that the set of vectors $\{(a_{1j}, \ldots, a_{nj}) \,|\, \forall j \in [k]\}$ is linearly independent; then we can perform Gaussian elimination to find a relation between the free variables (exactly $n-k$) and dependent/leading variables. Then, by enumerating over all possible 2^{n-k} values of the free variables and finding the corresponding values for leading variables, we can check whether there is a 0/1 solution, hence solving it in $\mathsf{poly}(n,k) \cdot 2^{n-k}$ time. This implies that when $k \geq n - O(\log(n))$, SimulSubsetSum (with assuming linear independence) has a polynomial time solution. Whereas, we showed (in Theorem 6) that given a subset sum instance, we can convert this into a SimulSubsetSum instance in polynomial time even with $k = O(\log(n))$.

1.1 Our contributions

Theorem 1 (Time-efficient algorithm for Subset Product). *There exists a randomized algorithm that solves* Subset Product *in* $\tilde{O}(n + t^{o(1)})$ *expected-time.*

Remarks. 1. The result in the first part of the above theorem is reminiscent of the $\tilde{O}(n+t)$ time randomized algorithms for the subset sum problem [15,4], although the time complexity in our case is the expected time, and ours is better.

2. The expected time is because to factor an integer t takes expected $\exp(O(\sqrt{\log(t)\log\log(t)}))$ time [18]. If one wants to remove expected time analysis (and do the worst case analysis), the same problem can be solved in $\tilde{O}(n^2 + t^{o(1)})$ randomized-time. For details, see the end of subsection 3.1.

3. While it is true that Bellman's algorithm gives $O(nt)$ time algorithm, the state-space of this algorithm can be improved to (expected) $nt^{o(1)}$-time for Subset Product, using a similar dynamic algorithm with a careful analysis. For details, see D.

Theorem 2 (Space-efficient algorithm for Subset Product). *Subset Product can be solved deterministically in $O(\log^2(nt))$ space and $\mathsf{poly}(nt)$-time.*

Remark. We *cannot* directly invoke the theorem in [16, Section 3.3] to conclude Theorem 2 since the reduction from Subset Product to SimulSubsetSum requires $O(n\log(nt))$ space. Essentially, we use the same identity lemma as [16] and carefully use the space; for details see section A.

Using a pseudo-prime-factorization decomposition, we show that given a target t in Subset Product, it suffices to solve SimulSubsetSum with at most $\log t$ many instances, where each of the targets is also 'small', at most $O(\log\log t)$ bits.

Theorem 3 (Reducing Subset Product to SimulSubsetSum). *There is a deterministic polynomial time reduction from Subset Product to SimulSubsetSum.*

Remark. The reduction uses $\tilde{O}(n\log t)$ space as opposed to the following chain of reductions: Subset Product $\leq_{\mathsf{P}}$ SSUM $\leq_{\mathsf{P}}$ SimulSubsetSum. The first reduction is a *natural* reduction, from an input $(a_1,\ldots,a_n,t)$, which takes log both sides and adjusts (multiply) a 'large' M (it could be $O(n\log t)$ bit [17,22]) with $\log a_i$, to reduce this to a SSUM instance with $b_i := \lfloor M\log a_i\rfloor$. Therefore, the total space required could be as large as $\tilde{O}(n^2\log t)$. The second reduction follows from Theorem 6. Thus, ours is more space efficient. Motivated thus, we give an efficient randomized algorithm for SimulSubsetSum.

We also show that SimulSubsetSum, even with 2 instances, is as hard as SSUM. Though the proof is very standard, we sketch this for the completeness. For details, see section B.

1.2 Prior works and limitation of the obvious attempts

There have been a very few attempts to classically solve Subset Product or its variants. It is known to be NP-complete and the reduction follows from the Exact Cover by 3-Sets (X3C) problem [12, p. 221]. Though the knapsack and its approximation versions have been studied [17,22], we do not know many classical algorithms and attempts to solve this, unlike the recent attention for the subset sum problem [4,15,14,5]. In this paper, we start investigating similar questions in the Subset Product regime.

Why the obvious methods fail. Since subset sum can be solved in randomized $\tilde{O}(n+t)$ time [15], as mentioned before, one obvious way to solve Subset Product would be to work with $b_i := \lfloor M \log a_i \rfloor$ and a $\mathcal{R}$, a range of target values t' which could be as large as $M \log t$ such that Subset Product is YES iff subset sum instance with b_i and $t' \in \mathcal{R}$ is YES. But M could be as large as $O(n \cdot (\prod_i a_i)^{1/2})$. Therefore, although there is a randomized near-linear time algorithm for subset sum, when one reduces the instance of Subset Product to a subset sum instance, the target becomes very large, failing to give an $\tilde{O}(n+t)$ algorithm.

Similarly, Theorem 3 along with the reduction in Theorem 7, which reduces the Subset Product to SSUM, actually blows up the target, and fails to give near-linear time algorithm.

We also mention that it is not clear if non-algebraic techniques, as used in [4], could be extended for SimulSubsetSum or not. Moreover, the general techniques, used for subset sum [4,15,14] seem to fail to 'directly' give algorithms for Subset Product. This is exactly why, in this work, the efficient algorithms have been indirect, *via* solving SimulSubsetSum instances.

2 Preliminaries

Notations. $\mathbb{N}, \mathbb{Z}$ and $\mathbb{Q}$ denotes the set of all natural numbers, integers and rational numbers respectively. Let a, b be two m-bit integers. Then, $a//b$ denotes a/b^e where e is the largest non-negative integer such that $b^e \mid a$. Observe that $a//b$ is not divisible by b and the time to compute $a//b$ is $O(m \log(m) \cdot \log(e))$. Also, $\tilde{O}(N)$ denotes $N \cdot \mathsf{poly}(\log N)$.

For any positive integer $n > 0$, $[n]$ denotes the set $\{1, 2, \ldots, n\}$ while $[a, b]$ denotes the set of integers i s.t. $a \leq i \leq b$. $\mathbb{F}[x_1, \ldots, x_k]$ denotes the ring of k-variate polynomials over field $\mathbb{F}$ and $\mathbb{F}[[x_1, \ldots, x_k]]$ is the ring of power series in k-variables over $\mathbb{F}$. We will use the short-hand notation $\boldsymbol{x}$ to denote the collection of variables $(x_1, \ldots, x_k)$ for some k. For any non-negative integer vector $\boldsymbol{e} \in \mathbb{Z}^k$, $\boldsymbol{x}^{\boldsymbol{e}}$ denotes $\prod_{i=1}^{k} x_i^{e_i}$. Using these notations, we can write any polynomial $f(\boldsymbol{x}) \in \mathbb{Z}[\boldsymbol{x}]$ as $f(\boldsymbol{x}) = \sum_{\boldsymbol{e} \in S} f_e \cdot \boldsymbol{x}^{\boldsymbol{e}}$ for some suitable set S. We denote $\mathrm{coef}_{\boldsymbol{x}^{\boldsymbol{e}}}(f)$, as the coefficient of $\boldsymbol{x}^{\boldsymbol{e}}$ in the polynomial $f(\boldsymbol{x})$ and $\deg_{x_i}(f)$ as the highest degree of x_i in $f(\boldsymbol{x})$.

Lemma 1 (Kane's Identity [16]). *Let $f(x) = \sum_{i=0}^{d} c_i x^i$ be a polynomial of degree at most d with coefficients c_i being integers. Let $\mathbb{F}_q$ be the finite field of order $q = p^k > d + 2$. For $0 \leq t \leq d$, define*

$$r_t = \sum_{x \in \mathbb{F}_q^*} x^{q-1-t} f(x) = -c_t \in \mathbb{F}_q$$

Then, $r_t = 0 \iff c_t$ is divisible by p.

Theorem 4 ([21]). *For $n \geq 25$, there is always a prime in $[n, 6/5 \cdot n]$.*

The following is a naive bound, but it is sufficient for our purpose.

Lemma 2. *For integers $a \geq b \geq 1$, we have $(a/b)^b \leq 2^{2\sqrt{ab}}$.*

Proof. Let $x = \sqrt{a/b}$. We need to show that $x^{2b} \leq 2^{2bx}$, which is trivially true since $x \leq 2^x$, for $x \geq 1$.

3 Time-efficient algorithm for Subset Product

In this section, we give a randomized $\tilde{O}(n + t^{o(1)})$ expected time algorithm for Subset Product. Essentially, we factor all the entries in the instance in $\tilde{O}(n+t^{o(1)})$ expected time. Once we have the exponents, it suffices to solve the corresponding SimulSubsetSum instance. Now, we can use the efficient randomized algorithm for SimulSubsetSum (Theorem 5) to finally solve Subset Product. So, first we give an efficient algorithm for SimulSubsetSum.

Theorem 5 (Algorithm for SimulSubsetSum). *There is a randomized $\tilde{O}(kn+\prod_{i\in[k]}(2t_i+1))$-time algorithm that solves SimulSubsetSum, with target instances $t_1, \ldots, t_k$.*

Proof. Let us assume that the input to the SimulSubsetSum problem is k SSUM instance of the form $(a_{1j}, \ldots, a_{nj}, t_j)$, for $j \in [k]$. Define a k-variate polynomial $f(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, \ldots, x_k)$, as follows:

$$f(\boldsymbol{x}) \;=\; \prod_{i=1}^{n} \left(1 + \prod_{j=1}^{k} x_j^{a_{ij}}\right) .$$

Here is an immediate but important claim. We denote the monomial $\boldsymbol{m} := \prod_{i=1}^{k} x_i^{t_i}$ and $\mathrm{coef}_{\boldsymbol{m}}(f)$ as the coefficient of $\boldsymbol{m}$ in the polynomial $f(\boldsymbol{x})$.

Claim 1. There is a solution to the SimulSubsetSum instance, i.e., $\exists S \subseteq [n]$ such that $\sum_{i\in S} a_{ij} = t_j, \forall j \in [k]$ *iff* $\mathrm{coef}_{\boldsymbol{m}}(f(\boldsymbol{x})) \neq 0$.

Therefore, it is enough to compute the coefficient of $f(\boldsymbol{x})$. The rest of the proof focuses on computing $f(\boldsymbol{x})$ efficiently, to find $\mathrm{coef}_{\boldsymbol{m}}(f)$.

Let p be prime such that $p \in [N+1, (n+N)^3]$, where $N := \prod_{i=1}^{k}(2t_i+1)$. Define an ideal $\mathcal{I}$, over $\mathbb{Z}[\boldsymbol{x}]$ as follows: $\mathcal{I} := \langle x_1^{t_1+1}, \ldots, x_k^{t_k+1}, p\rangle$. Since, we are interested in $\mathrm{coef}_{\boldsymbol{m}}(f)$, it suffices to compute $f(\boldsymbol{x}) \bmod \langle x_1^{t_1+1}, \ldots, x_k^{t_k+1}\rangle$, and we do it over a field $\mathbb{F}_p$ (which introduces error); for details, see the proof in the end (Randomness and error probability paragraph).

Using Lemma 6, we can compute all the coefficients of $\ln(f(\boldsymbol{x})) \bmod \mathcal{I}$ in time $\tilde{O}(kn + \prod_{i=1}^{k} t_i)$. It is easy to see that the following equalities hold.

$$f(\boldsymbol{x}) \mod \mathcal{I} \;\equiv\; \exp\left(\ln(f(\boldsymbol{x}))\right) \mod \mathcal{I} \;\equiv\; \exp\left(\ln(f(\boldsymbol{x})) \bmod \mathcal{I}\right) \mod \mathcal{I} .$$

Since, we have already computed $\ln(f(\boldsymbol{x})) \bmod \mathcal{I}$, the above equation implies that it is enough to compute the exponential which can be done using Lemma 5. This also takes time $\tilde{O}(kn + \prod_{i=1}^{k}(2t_i+1))$.

Randomness and error probability. Note that there are $\Omega(n+N)^2$ primes in the interval $[N+1,(n+N)^3]$. Moreover, since $\mathrm{coef}_{\boldsymbol{m}}(f) \leq 2^n$, at most n prime factors can divide $\mathrm{coef}_{\boldsymbol{m}}(f(\boldsymbol{x}))$. Therefore, we can pick a prime p randomly from this interval in $\mathsf{poly}(\log(n+N))$ time and the probability of p dividing the coefficient is $O(n+N)^{-1}$. In other words, the probability that the algorithm fails is bounded by $O\left((n+N)^{-1}\right)$. This concludes the proof. □

We now compare the above result with some obvious attempts to solve SimulSubsetSum, before moving into solving Subset Product.

A detailed comparison with time complexity of [16]. Kane [16, Section 3.3] showed that the SimulSubsetSum problem can be solved deterministically in $C^{O(k)}$ time and $O(k \log C)$ space, where $C := \sum_{i,j} a_{ij} + \sum_j t_j + 1$, which could be as large as $(n+1) \cdot (\sum_{j\in[k]} t_j) + 1$, since a_{ij} can be as large as t_j. As argued in [14, Corollary 3.4 and Remark 3.5], the constant in the exponent, inside the order notation, can be as large as 3 (in fact directly using [16] gives a larger constant; but modified algorithm as used in [14] gives 3). Use AM-GM inequality to get

$$\left((n+1)\cdot(\sum_j t_j)+1\right)^{3k} > \left(\frac{2}{k}\cdot\sum_j t_j + 1\right)^{3k} \overset{\text{AM-GM}}{\geq} \prod_{j=1}^{k}(2t_j+1)^3 \ .$$

Assuming $N = \prod_{j=1}^{k}(2t_j+1)$, our algorithm is near-linear in N while Kane's algorithm [16] takes $O(N^3)$ time; thus ours is almost a cubic improvement.

Comparison with the trivial algorithm. It is easy to see that a trivial $O(n\cdot(t_1+1)(t_2+1)\ldots(t_k+1))$ time *deterministic* algorithm for SimulSubsetSum exists. Since, $t_i \geq 1$, we have

$$\frac{n}{2}\cdot\prod_{i\in[k]}(1+t_i) \geq \frac{n}{2}\cdot 2^k \geq kn\,, \text{ and } \frac{n}{2}\cdot\prod(1+t_i) \geq \frac{n}{2^{k+1}}\cdot\prod(2t_i+1)\,.$$

Here, we used $2(1+x) > (2x+1)$, for any $x \geq 1$. Therefore, $n\cdot\prod_{i\in[k]}(1+t_i) \geq kn + n/2^{k+1}\cdot\prod(2t_i+1)$. Thus, when $k = o(\log n)$, our complexity is better.

3.1 Proof of Theorem 1

Once we have designed the algorithm for SimulSubsetSum, we design a time-efficient algorithm for Theorem 1.

Proof. Let $(a_1,\ldots,a_n,t) \in \mathbb{Z}_{\geq 0}^{n+1}$ be the input for Subset Product problem. Without loss of generality, we can assume that all the a_i divides t because if some a_i does not divide t, it will never be a part of any solution and we can discard it. Let us first consider the prime factorization of t and a_j, for all $j \in [n]$. We will discuss about its time complexity in the next paragraph. Let

$$t \;=\; \prod_{j=1}^{k} p_j^{t_j}\,, \qquad a_i \;=\; \prod_{j=1}^{k} p_j^{e_{ij}}, \forall\, i \,\in\, [n]\,,$$

where p_j are distinct primes and t_j are positive integers and $e_{ij} \in \mathbb{Z}_{\geq 0}$. Since, $p_i \geq 2$, trivially, $\sum_{i=1}^{k} t_i \leq \log(t)$, and $\sum_{i=1}^{k} e_{ij} \leq \log(t), j \in [n]$. Also, the number of distinct prime factors of t is at most $O(\log(t)/\log\log(t))$; therefore, $k = O(\log(t)/\log\log(t))$.

Time complexity of factoring To find all the primes that divide t, we will use the factoring algorithm given by Lenstra and Pomerance [18] which takes expected $t^{o(1)}$ [3] time to completely factor t into prime factors p_j (including the exponents t_j). Using the primes p_j and the fact that $0 \leq e_{ij} \leq \log(t)$, computing e_{ij} takes $\log^2(t)\log\log(t)$ time, by performing binary search to find the largest x such that $p_j^x \mid a_i$. So, the time to compute all exponents $e_{i,j}, \forall i \in [n], j \in [k]$ is $O(nk\log^2(t)\log\log(t))$. Since, $k \leq O(\log t/\log\log(t))$, the total time complexity is $\tilde{O}(n + t^{o(1)})$.

Setting up SimulSubsetSum Now suppose that $S \subseteq [n]$ is a solution to the Subset Product problem, i.e., $\prod_{i\in S} a_i = t$. This implies that

$$\sum_{i\in S} e_{ij} \;=\; t_j\,, \quad \forall\, j \,\in\, [k]\,.$$

In other words, we have a SimulSubsetSum instance where the j^{th} SSUM instance is $(e_{1j}, e_{2j}, \ldots, e_{nj}, t_j)$, for $j \in [k]$. The converse is also trivially true. We now show that there exists an $\tilde{O}(kn + \prod_{i\in[k]}(2t_i+1))$ time algorithm to solve SimulSubsetSum.

Randomized algorithm for Subset Product Using Theorem 5, we can decide the SimulSubsetSum problem with targets $t_1, \ldots, t_k$ in $\tilde{O}(kn + \prod_{i\in[k]}(2t_i+1))$ time (randomized) while working over $\mathbb{F}_p$ for some suitable p (we point out towards the end). Since $k \leq O(\log(t)/\log\log(t))$, we need to bound the term $\prod_{i\in[k]}(2t_i+1)$. Note that,

$$\begin{aligned}\prod_{i\in[k]} (2t_i+1) \;&=\; \sum_{S\subseteq[k]} 2^{|S|} \cdot \left(\prod_{i\in S} t_i\right) \\ &\leq\; 2^{2k} \cdot \left(\prod_{i\in[k]} t_i\right)\,.\end{aligned}$$

[3] Expected time complexity is $\exp(O(\sqrt{\log t \log\log t}))$, which is smaller than $t^{O(1/\sqrt{\log\log t})} = t^{o(1)}$, which will be the time taken in the next step. Moreover, we are interested in *randomized* algorithms, hence expected run-time is

We now focus on bounding the term $\prod_{i\in[k]} t_i$. By AM-GM,

$$\begin{aligned}\prod_{i\in[k]} t_i \leq \left(\frac{\sum_{i\in[k]} t_i}{k}\right)^k &\leq \left(\frac{\log(t)}{k}\right)^k \\ &\leq 2^{O\left(\sqrt{k\log(t)}\right)} \quad [Lemma\ 2] \\ &\leq 2^{O\left(\sqrt{\log(t)^2/\log\log(t)}\right)} \\ &\leq t^{O(1/\sqrt{\log\log(t)})} = t^{o(1)}\end{aligned}$$

Note that the prime p in the Theorem 5 was $p \in [N+1, (n+N)^3]$, where $N := \prod_{i=1}^{k}(2t_i+1)-1$. As shown above, we can bound $N = t^{o(1)}$. Thus, $p \leq O((n+t^{o(1)})^3)$, as desired. Therefore, the total time complexity is $\tilde{O}(n\log(t)/\log\log(t)+t^{o(1)}) = \tilde{O}(n+t^{o(1)})$. This finishes the proof. □

Removing the expected-time If one wants to understand the worst-case analysis, we can use the polynomial time reduction from Subset Product to SimulSubsetSum in section 4. Of course, we will not get prime factorization; but the pseudo-prime factors will also be good enough to set up the SimulSubsetSum with similar parameters as above, and the SimulSubsetSum instance can be similarly solved in $\tilde{O}(n+t^{o(1)})$ time. Since the reduction takes $n^2\mathsf{poly}(\log t)$ time, the total time complexity becomes $\tilde{O}(n^2+t^{o(1)})$.

4 An efficient reduction from Subset Product to SimulSubsetSum

In this section, we will present a deterministic polynomial time reduction from Subset Product to SimulSubsetSum. In section 3, we have given a pseudo-polynomial time reduction from Subset Product to SimulSubsetSum by performing prime-factorization of the input $(a_1,\ldots,a_n,t)$. The polynomial time reduction also requires to factorize the input, but the factors are not necessarily prime. To be precise, we define pseudo-prime-factorization which can be achieved in polynomial time.

Definition 1 (Pseudo-prime-factorization). *A set of integers $\mathcal{P} \subset \mathbb{N}$ is said to be pseudo-prime-factor set of $(a_1,\ldots,a_n) \in \mathbb{N}^n$ if*

1. *the elements of $\mathcal{P}$ are pair-wise coprime, i.e., $\forall p_1,p_2 \in \mathcal{P}, gcd(p_1,p_2)=1$,*
2. *there are only non-trivial factors of a_i's in $\mathcal{P}$, i.e., $\forall p \in \mathcal{P}, \exists i \in [n]$ such that $p \mid a_i$,*
3. *every a_i's can be uniquely expressed as product of powers of elements of $\mathcal{P}$, i.e., $\forall i \in [n], a_i = \prod_{p\in\mathcal{P}} p^{e_p}, \forall i \in [n]$ where $e_p \geq 0$.*

For a given $(a_1,\ldots,a_n)$, $\mathcal{P}$ may not be unique. A trivial example of a pseudo-prime-factor set of $\mathcal{P}$ for $(a_1,\ldots,a_n)$ is the set of all distinct prime factors of $\prod_{i=1}^{n} a_i$. The following is an important claim which will be used to give a *polynomial* time reduction from Subset Product to SimulSubsetSum.

Claim 2. For any pseudo-prime-factor set $\mathcal{P}$ of $(a_1, \ldots, a_n)$, we have $|\mathcal{P}| \leq k$ where k is the number of distinct prime factors of $\prod_{i=1}^{n} a_i$.

Proof. The proof uses a simple pigeonhole principle argument. Let $g_1, \ldots, g_k$ be the distinct prime factors of $\prod_{i=1}^{n} a_i$. From the definition of $\mathcal{P}$, we know that $g_1, \ldots, g_k$ are the only distinct prime factors of $\prod_{p \in \mathcal{P}} p$. Therefore, if there are more than k numbers in $\mathcal{P}$, then there must exist $p_1, p_2 \in \mathcal{P}$ such that $gcd(p_1, p_2) \neq 1$ which violates pair-wise coprime property of $\mathcal{P}$. □

Constructing $\mathcal{P}$ suffices We now show that having a pseudo-prime-factor set $\mathcal{P}$ for $(a_1, \ldots, a_n, t)$ helps us to reduce a Subset Product instance $(a_1, \ldots, a_n, t)$ to SimulSubsetSum with number of instances $|\mathcal{P}|$, in polynomial time. Wlog, we can assume that $a_i \mid t$ and $a_i, t \leq 2^m, \forall i \in [n]$ for some m. Trivially, $m \leq \log t$. So, using Claim 2, we have $|\mathcal{P}| \leq (n+1) \cdot m = \mathsf{poly}(n \log t)$.

From Definition 1, we have unique non-negative integers e_{ij} and t_j such that $t = \prod_{j \in |\mathcal{P}|} p_j^{t_j}$ and $a_i = \prod_{j \in |\mathcal{P}|} p_j^{e_{ij}}, \forall i \in [n]$. Since, $a_i \mid t$, we have $e_{ij} \leq t_j \leq m, \forall i \in [n], j \in [|\mathcal{P}|]$ and they can be computed in $\mathsf{poly}(m, n)$ time.

Let us consider the $|\mathcal{P}| -$ SimulSubsetSum instance where the i^{th} SSUM instance is $(e_{1i}, e_{2i}, \ldots, e_{ni}, t_i)$. Then, due to factorization property (the third property in Definition 1) of $\mathcal{P}$, the Subset Product instance is YES, i.e., $\exists S \in [n]$ such that $\prod_{i \in S} a_i = t$ iff the SimulSubsetSum instance with number of instances $|\mathcal{P}|$, is a YES.

4.1 Polynomial time algorithm for computing pseudo-prime-factors

We will now present a deterministic polynomial time algorithm for computing a pseudo-prime-factor set $\mathcal{P}$ for $(a_1, \ldots, a_n)$. We will use the notation $\mathcal{P}(a_1, \ldots, a_n)$ to denote a pseudo-prime-factor set for $(a_1, \ldots, a_n)$. Also, let $\mathcal{S}(a_1, \ldots, a_n)$ be the set of all pseudo-prime-factor sets; this is a finite set.

The following lemma is a crucial component in algorithm 1. We use $a//b$ to denote a/b^e such that $b^{e+1} \nmid a$.

Lemma 3. *Let $(a_1, \ldots, a_n)$ be n integers. Then,*

1. *If a_1 is coprime with $a_i, \forall i > 1$, then for any $\mathcal{P}(a_2, \ldots, a_n) \in \mathcal{S}(a_2, \ldots, a_n)$, $\mathcal{P}(a_2, \ldots, a_n) \cup \{a_1\} \in \mathcal{S}(a_1, \ldots, a_n)$.*
2. *$\mathcal{P}(g, a_1//g, a_2//g, \ldots, a_n//g) \in \mathcal{S}(a_1, \ldots, a_n)$, for given a_i, $i \in [n]$ and any factor g of some a_i.*

Proof. The first part of the lemma is trivial. For the second part, let g be a non-trivial factor of some a_i and

$$\mathcal{P} \;:=\; \{p_1, \ldots, p_k\} \;\in\; \mathcal{S}(g, a_1//g, a_2//g, \ldots, a_n//g)\,,$$

be any pseudo-prime-factor set. Then, p_i's are pair-wise coprime and since each p_i divides either g or $a_i//g$ for some $i \in [n]$, it also divides some a_i because g is a factor of some a_i. Also, we have *unique* non-negative integers e_{ip}, e_{gp} s.t.

$$a_i//g = \prod_{p \in \mathcal{P}} p^{e_{ip}}, \forall i \in [n] \text{ and } g = \prod_{p \in \mathcal{P}} p^{e_{gp}}\,.$$

Combining these equation, we get $a_i = a_i//g * g^{f_{ig}} = \prod_{p\in\mathcal{P}} p^{e_{ip}+e_{gp}*f_{ig}}$. Here f_{ig} is the maximum power of g that divides a_i. Therefore, $\{p_1, \ldots, p_k\}$ is also a pseudo-prime-factor set for $(a_1, \ldots, a_n)$. □

Pre-processing Using Lemma 3, algorithm 1 performs a divide-and-conquer approach to find $\mathcal{P}(a_1, \ldots, a_n)$. Observe that we can always remove duplicate elements and 1's from the input since it *does not* change the pseudo-prime-factors. Also, we can assume without loss of generality that $a_i//a_1 =: a_i, \forall i > 1$ because of the second part in Lemma 3, with $g = a_1$, since it gives us $\mathcal{P}(a_1, a_2//g, \ldots, a_n//g)$ and we know it suffices to work with these inputs.

If a_1 is coprime to the rest of the a_i's, then the algorithm will recursively call itself on $(a_2, \ldots, a_n)$ and combine $\mathcal{P}(a_2, \ldots, a_n)$ with $\{a_1\}$. Else, there exists an $i > 1$ such that $gcd(a_1, a_i) \neq 1$. So, the algorithm finds a factor g of a_1 using Euclid's GCD algorithm and computes $\mathcal{P}(g, a_1//g, \ldots, a_n//g)$. At every step we remove duplicates and 1's. Hence, the correctness of algorithm 1 is immediate assuming it terminates.

To show the termination and time complexity of algorithm 1, we will use the *'potential function'* $\mathbb{P}(I) := \prod_{a\in I} a$, where I is the input and show that at each recursive call, the value of the potential function is halved. Initially, the value of the potential function is $\prod_{i=1}^{n} a_i$. We also remark that since the algorithm removes duplicates and 1's; the potential function can *never* increase by the removal step and so it never matters in showing the decreasing nature of $\mathbb{P}$.

1. a_1 is coprime to the rest of the a_i's: In this case, the recursive call has input $(a_2, \ldots, a_n)$. Since, $a_1 \geq 2$, the value of potential function is
$$\mathbb{P}(a_2, \ldots, a_n) \;=\; \prod_{i=2}^{n} a_i \;<\; (\prod_{i=1}^{n} a_i)/2 \;=\; \mathbb{P}(a_1, \ldots, a_n)/2\,.$$
2. a_1 shares a common factor with some a_i. Let $g = gcd(a_1, a_i) \neq 1$. Since, we have assumed $a_i//a_1 = a_i$, this implies that a_i is not a multiple of a_1. This implies that $2 \leq g \leq a_1/2$. Therefore, the new value of potential function is
$$\begin{aligned}
&\mathbb{P}(g, a_1//g, \ldots, a_n//g)\\
&= g\prod_{j=1}^{n} a_j//g\\
&\leq (a_1//g) \times ((a_i//g) \times g) \times \prod_{j\in[n]\setminus\{1,i\}} a_j\\
&\leq \frac{a_1}{g} \cdot \prod_{j=2}^{n} a_j\\
&\leq (\prod_{j=1}^{n} a_j)/2 \;=\; \mathbb{P}(a_1, \ldots, a_n)/2\,.
\end{aligned}$$
We used the fact that since, $2 \leq g \,|\, a_i$, therefore, $g \times (a_i//g) \leq a_i$.

Time complexity In both the cases, the value of the potential function is halved. So, the depth of the recursion tree (in-fact, it is just a line) is at most $\log(\prod_{i=1}^{n} a_i) \leq m \cdot n$. Also, in each recursive call, the input size is increased at most by one but the integers are still bounded by 2^m. This implies that input size, for any recurrence call, can be at most $(m+1) \cdot n$. Since there is no branching, the total number of operations is $(m+1) \cdot n \times m \cdot n = O((mn)^2)$. Therefore, the total time complexity is $n^2 \cdot poly(m)$.

Algorithm 1: Algorithm for Pseudo-prime-factor set

Input: $(a_1, a_2 \ldots, a_n) \in \mathbb{N}^n$ where each a_i is an m-bit integer such that $a_i//a_1 = a_i > 1, \forall i > 1$

Output: Pseudo-prime-factor set $\mathcal{P}$ for $(a_1, a_2, \ldots, a_n)$

```
if n == 0 then
    return ∅;
end
if ∃i > 1 such that gcd(a_1, a_i) ≠ 1 then
    g = gcd(a_1, a_i);
    I = {g};
    for i ∈ [n] do
        a'_i = a_i//g;
        if a'_i ∉ I and a'_i ≠ 1 then
            I = I ∪ {a'_i}
        end
    end
    return P(I);
end
else
    return P(a_2, ..., a_n) ∪ {a_1};
end
```

5 Conclusion

In this paper, we give efficient algorithms for Problem 1-2 which are variants of SSUM problem. We also present an efficient reduction from Subset Product to SimulSubsetSum. Here are some immediate questions to investigate.

1. Can we improve the complexity of Theorem 5 to $\tilde{O}(n + \sum_{i=1}^{k} t_i)$?
2. What can we say about the hardness of SimulSubsetSum with k subset sum instances where $k = \omega(\log(n))$?

References

1. Bazgan, C., Hugot, H., Vanderpooten, D.: Solving efficiently the 0–1 multi-objective knapsack problem. Computers & Operations Research **36**(1), 260–279 (2009)

2. Bellman, R.E.: Dynamic programming (1957)
3. Bernstein, D.J., Jeffery, S., Lange, T., Meurer, A.: Quantum algorithms for the subset-sum problem. In: International Workshop on Post-Quantum Cryptography. pp. 16–33. Springer (2013)
4. Bringmann, K.: A near-linear pseudopolynomial time algorithm for subset sum. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1073–1084. SIAM (2017)
5. Bringmann, K., Wellnitz, P.: On near-linear-time algorithms for dense subset sum. In: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 1777–1796. SIAM (2021)
6. Contini, S., Lenstra, A.K., Steinfeld, R.: Vsh, an efficient and provable collision-resistant hash function. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 165–182. Springer (2006)
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
8. Draziotis, K.A., Martidis, V., Tiganourias, S.: Product subset problem: Applications to number theory and cryptography. arXiv preprint arXiv:2002.07095 (2020)
9. Dutta, P., Rajasree, M.S.: Algebraic algorithms for variants of subset sum. In: Conference on Algorithms and Discrete Applied Mathematics. pp. 237–251. Springer (2022)
10. Esser, A., May, A.: Low weight discrete logarithm and subset sum in 20. 65n with polynomial memory. memory **1**, 2 (2020)
11. Faust, S., Masny, D., Venturi, D.: Chosen-ciphertext security from subset sum. In: Public-Key Cryptography–PKC 2016, pp. 35–46. Springer (2016)
12. Garey, M.R., Johnson, D.S.: Computers and intractability, vol. 174. freeman San Francisco (1979)
13. Helm, A., May, A.: Subset sum quantumly in 1.17ˆ n. In: 13th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2018). pp. 1–15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
14. Jin, C., Vyas, N., Williams, R.: Fast low-space algorithms for subset sum. In: Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 1757–1776. SIAM (2021)
15. Jin, C., Wu, H.: A simple near-linear pseudopolynomial time randomized algorithm for subset sum. arXiv preprint arXiv:1807.11597 (2018)
16. Kane, D.M.: Unary subset-sum is in logspace. arXiv preprint arXiv:1012.1336 (2010)
17. Kovalyov, M.Y., Pesch, E.: A generic approach to proving np-hardness of partition type problems. Discrete applied mathematics **158**(17), 1908–1912 (2010)
18. Lenstra, H.W., Pomerance, C.: A rigorous time bound for factoring integers. Journal of the American Mathematical Society **5**(3), 483–516 (1992)
19. Lewis, H.R.: Computers and intractability. a guide to the theory of np-completeness (1983)
20. Lyubashevsky, V., Palacio, A., Segev, G.: Public-key cryptographic primitives provably as secure as subset sum. In: Theory of Cryptography Conference. pp. 382–400. Springer (2010)
21. Nagura, J.: On the interval containing at least one prime number. Proceedings of the Japan Academy **28**(4), 177–181 (1952)
22. Pferschy, U., Schauer, J., Thielen, C.: Approximating the product knapsack problem. Optimization Letters pp. 1–12 (2021)

A Space-efficient algorithm for Subset Product

The proof of Theorem 2 uses the idea of reducing Subset Product to SimulSubsetSum and then solving SimulSubsetSum by computing the coefficient of $f(\boldsymbol{x}) = \prod_{i=1}^{n} \left(1 + \prod_{j=1}^{k} x_j^{a_{ij}}\right)$ where $\boldsymbol{x} = (x_1, \ldots, x_k)$ using an extension of Lemma 1. We cannot directly use [16] as it requires large space ($O((n + \log(nt))\log(t))$ space to be precise) to store the SimulSubsetSum instance; for this discussion refer to the paragraph in the next page after the remark. So, here instead we compute the coefficient of $f(\boldsymbol{x})$ without storing the SimulSubsetSum instance using Lemma 4.

The low space algorithm presented in this proof depends on the generalisation of Lemma 1. Here we present Kane's identity for *bi*variate polynomials which can be easily extended to k-variate polynomials.

Lemma 4 (Identity lemma [16]). *Let $f(x,y) = \sum_{i=0}^{d_1}\sum_{j=0}^{d_2} c_{i,j}x^i y^j$ be a polynomial of degree at most $d_1 + d_2$ with coefficients $c_{i,j}$ being integers. Let $\mathbb{F}_q$ be the finite field of order $q = p^m > \max(d_1, d_2)+1$. For $0 \le t_1 \le d_1, 0 \le t_2 \le d_2$, define*

$$r_{t_1,t_2} = \sum_{x\in\mathbb{F}_q^*}\sum_{y\in\mathbb{F}_q^*} x^{q-1-t_1}y^{q-1-t_2}f(x,y) = c_{t_1,t_2} \in \mathbb{F}_q$$

Proof. Let n be a positive integer, then the two following identities hold:

Identity 1. $\sum_{x\in\mathbb{F}_q^*} x^n = -1$ if $q-1 \mid n$ because $x^n = x^{(q-1)m} = 1$ due to Fermat's Little theorem.

Identity 2. $\sum_{x\in\mathbb{F}_q^*} x^n = 0$, if $q-1 \nmid n$. This is because we can rewrite the summation as $\sum_{i=0}^{q-2} g^{i\cdot n} = \dfrac{g^{n(q-1)}-1}{g^n-1} = 0$ where g is a generator of $\mathbb{F}_q^*$.

Let us now consider $\sum_{x\in\mathbb{F}_q^*}\sum_{y\in\mathbb{F}_q^*} x^{q-1-t_1}y^{q-1-t_2}f(x,y)$.

$$\begin{aligned}
\sum_{x\in\mathbb{F}_q^*}\sum_{y\in\mathbb{F}_q^*} x^{q-1-t_1}y^{q-1-t_2}f(x,y) &= \sum_{x\in\mathbb{F}_q^*}\sum_{y\in\mathbb{F}_q^*} x^{q-1-t_1}y^{q-1-t_2}\left(\sum_{i=0}^{d_1}\sum_{j=0}^{d_2} c_{i,j}x^i y^j\right)\\
&= \sum_{i=0}^{d_1}\sum_{j=0}^{d_2} c_{i,j}\left(\sum_{x\in\mathbb{F}_q^*}\sum_{y\in\mathbb{F}_q^*} x^{q-1-t_1+i}y^{q-1-t_2+j}\right)\\
&= \sum_{\substack{i\in[0,d_1]\setminus\{t_1\}\\ j\in[0,d_2]\setminus\{t_2\}}} c_{i,j}\left(\sum_{x\in\mathbb{F}_q^*}\sum_{y\in\mathbb{F}_q^*} x^{q-1-t_1+i}y^{q-1-t_2+j}\right)\\
&\quad + c_{t_1,t_2}\\
&= c_{t_1,t_2}
\end{aligned}$$

Observe that when $i \in [0,d_1]\setminus\{t_1\}$, we have $\sum_{x\in\mathbb{F}_q^*} x^{q-1-t_1+i} = 0$ because $|i-t_1| \le d_1 < q-1 \implies q-1+i-t_1$ is not a multiple of $q-1$. The same goes for $j \in [0,d_2]\setminus\{t_2\}$. ◻

▶ Remark. Lemma 4 can be easily extended to k variables which was used by the authors of [16] to solve SimulSubsetSum with k many SSUM instances in space $O(k \log(n \sum_{i=1}^{k} t_i))$ and time $(poly(n, t_1, \ldots, t_k))^{O(k)}$. In this case, the order of the finite field must be greater than $\max(d_1, \ldots, d_k) + 1$ where d_i's are the individual degrees of the polynomial. □

Now we briefly discuss the issue on directly using Kane's result [16] below.

Issue with directly invoking [16] We can reduce a Subset Product instance $(a_1, \ldots, a_n, t)$ to a SimulSubsetSum instance containing k SSUM instances $(e_{1i}, \ldots, e_{ni}, t_i), \forall i \in [k]$ where $k \leq \log(t)$, by using Theorem 3. The space required for the SimulSubsetSum instance is the number of bits in e_{ij}, t_j. We know that $a_i = \prod_{j=1}^{k} p_i^{e_{ij}} \implies 2^{\sum_{j=1}^{k} \log(e_{ij})} \leq 2^{\sum_{j=1}^{k} e_{ij}} \leq a_i$ because $p_i \geq 2, \forall i \in [k]$. Therefore, we have $\sum_{i,j} \log(e_{i,j}) \leq \sum_{i=1}^{n} \log(a_i) \leq n \log(t)$. Similarly, $\sum_i \log(t_i) \leq \log(t)$. Therefore, the space required for the SimulSubsetSum is $O(n \log(t))$. And, if we directly use the low-space algorithm for SimulSubsetSum from [16], the total space complexity would become $O((n + \log(nt)) \cdot \log(t))$.

A.1 Proof of Theorem 2

To avoid the n-factor in the space complexity, we will not store the entire SimulSubsetSum instance. Instead, for each summation in the k variate version of Lemma 4, we will compute the values of e_{ij} and t_j and discard them after using it. To be precise, for $\boldsymbol{g} = (t_1, \ldots, t_k)$, we have

$$c_{\boldsymbol{g}} = (-1)^k \sum_{\boldsymbol{x} \in (\mathbb{F}_q^*)^k} f(\boldsymbol{x}) \cdot \prod_{i=1}^{k} x_i^{q-1-t_i}$$

where $f(\boldsymbol{x}) = \prod_{i=1}^{n} \left(1 + \prod_{j=1}^{k} x_j^{e_{ij}}\right)$ and $c_{\boldsymbol{g}} = \text{coef}_{\boldsymbol{g}}(f(\boldsymbol{x}))$. The values of e_{ij} and t_i is only required in $f(\boldsymbol{x})$ and $\prod_{i=1}^{k} x_i^{q-1-t_i}$ respectively. Since, e_{ij} and t_i are the powers of p_i in a_j and t respectively, we can't use pseudo-prime-factorization as this would require us to use $O(n \log(t))$ space to compute a pseudo-prime-factor set. Therefore, we will use naive prime-factorization algorithm that runs in $\tilde{O}(t)$ time which is affordable because we are interested in poly(knt).

Choosing the prime q Observe that the total degree of $f(\boldsymbol{x})$ is $\sum_{ij} e_{ij} \leq n \cdot (\sum_i t_i) \leq n \log t$ because $0 \leq e_{ij} \leq t_j$ and $\sum_i t_i \leq \log(t)$. Therefore, the maximum individual degree is bounded by $n \log(t)$. Since, Lemma 4 requires a prime q that depends on the maximum individual degree of the polynomial, it suffices to work with $N = \lceil n \log(t) \rceil$ and $q > N$. Observe that we need to compute the coefficient modulo q, therefore, we need to ensure that q does not divide the coefficient. To achieve this, we will use Lemma 4 for different primes $q \in [N + 1, (n + N)^3]$ which contains $\Omega(n + N)^2$ prime. This works because the coefficient can be at most 2^n, therefore, it will have at most n prime factors. So, at least one prime in the range will not divide the coefficient.

Computing $f(\boldsymbol{x})$ and $\prod_{i=1}^{k} x_i^{q-1-t_i}$ using low space We will make sure that t_i is the exponent of the i^{th} *smallest* prime factor of t. To find an e_{ij}, we will first find the i^{th} smallest prime p_i that divides t and then compute the largest power of p_i that divides a_j. Once, we find e_{ij}, we can use it to compute $\prod_{j=1}^{k} x_j^{e_{ij}}$ part of $f(\boldsymbol{x})$ and discard it as shown in algorithm 2. Similarly, we can compute $\prod_{i=1}^{k} x_i^{q-1-t_i}$.

Space and Time complexity Observe that algorithm 2 uses only $O(\log(nt))$ space for variables that are used throughout the algorithm and reuses $O(\log(t))$ space while computing t_i, e_{ij}, p_ℓ values. It uses $k\log(nt) = O(\log^2(nt))$ space for $\boldsymbol{y}$, therefore, the total space complexity is $O(\log^2(nt))$. Whereas the time complexity is $\mathsf{poly}(nt)$ because each loop runs for $\mathsf{poly}(nt)$ iterations and finding the exponents take $\tilde{O}(t)$ time.

B Reductions between SimulSubsetSum and SSUM

In this section, we prove some hardness results. These proofs are very standard, still after different feedback, we give the details for the brevity.

Theorem 6 (Hardness of $2-$SimulSubsetSum). *There is a deterministic polynomial time reduction from* SSUM *to* $2-$SimulSubsetSum.

Proof. Let $(a_1, \ldots, a_n, t)$ be an instance of SSUM. Consider the following $2-$SimulSubsetSum instances, $\mathcal{S}_b = [(a_1, \ldots, a_n, t), (1, 0, \ldots, 0, b)]$, where $b \in \{0, 1\}$. If the SSUM instance is NO, then *both* the $2-$SimulSubsetSum are also NO. If the SSUM instance is a YES, then we argue that one of the S_b instance must be YES. If SSUM instance has a solution which contains a_1, then $\mathcal{S}_1$ is a YES instance whereas if it does not contain a_1, then S_0 is a YES instance. □

Extension to $\log-$SimulSubsetSum. The above reduction can be trivially extended to reduce SSUM to SimulSubsetSum, with number of SSUM instances $k = O(\log n)$. In that case we will work with instances $S_{\boldsymbol{b}}$, for $\boldsymbol{b} \in \{0,1\}^k$. Since the number of instances is $2^k = \mathsf{poly}(n)$, the reduction goes through.

We will now show that $2-$SimulSubsetSum reduces to SSUM which again can be generalised to SimulSubsetSum, for any number of SSUM instances k.

Theorem 7 ($2-$SimulSubsetSum is *easier* than SSUM). *There is a deterministic polynomial time reduction from* $2-$SimulSubsetSum *to* SSUM.

Proof. Let $[(a_1, \ldots, a_n, t_1), (b_1, \ldots, b_n, t_2)]$ be a $2-$SimulSubsetSum instance where without loss of generality $t_1 \leq t_2$. Also, we can assume that $t_1 \leq \sum_{i=1}^{n} a_i$, otherwise it does not have a solution.

Now, consider the SSUM instance $(\gamma b_1 + a_1, \ldots, \gamma b_n + a_n, \gamma t_2 + t_1)$, where $\gamma := 1 + \sum_{i=1}^{n} a_i$. If the $2-$SimulSubsetSum instance is YES, this implies that there exists $S \subseteq [n]$ such that $\sum_{i\in S} a_i = t_1$ and $\sum_{i\in S} b_i = t_2$. This implies that $\sum_{i\in S} \gamma b_i + a_i = \gamma t_2 + t_1$ and hence the SSUM instance is also YES.

Algorithm 2: Algorithm for solving Subset Product using low space

Input: A Subset Product instance $(a_1, \ldots, a_n, t) \in \mathbb{N}^{n+1}$
Output: Decides whether the Subset Product instance has a solution

$k = 0$;
for *each prime* $p_i \mid t$ **do**
  $k = k + 1$;
end
$N = \lceil n \log(t) \rceil$;
for *each prime* $q \in [N+1, (n+N)^3]$ **do**
  $c_{\boldsymbol{g}} = 1$;
  for *each* $\boldsymbol{y} \in (\mathbb{F}_q^*)^k$ **do**
    $prodx_1 = 1$;
    for $i \in [k]$ **do**
      Compute i^{th} smallest prime that divides t and find t_i;
      $prodx_1 = prodx_1 * y_i^{q-1-t_i}$;
      Discard t_i;
    end
    $f = 1$;
    for $i \in [n]$ **do**
      $prodx_2 = 1$;
      for $j \in [k]$ **do**
        Compute j^{th} smallest prime p_j that divides t;
        Using p_j compute e_{ij} which is the largest integer such that $p_j^{e_{ij}} \mid a_i$;
        $prodx_2 = prodx_2 * y_j^{e_{ij}}$;
        Discard p_j and e_{ij};
      end
      $f = f * (1 + prodx_2)$;
    end
    $c_{\boldsymbol{g}} = f * prodx_1$;
  end
  if $c_{\boldsymbol{g}} \neq 0$ **then**
    return True;
  end
end
return False;

Now, assume that the SSUM instance is YES, i.e., there exists $S \subseteq [n]$ such that $\sum_{i\in S} \gamma b_i + a_i = \gamma t_2 + t_1$. This implies that $\gamma(t_2 - \sum_{i\in S} b_i) + (t_1 - \sum_{i\in S} a_i) = 0$. If $t_1 \neq \sum_{i\in S} a_i$, then from the previous equality, $(t_1 - \sum_{i\in S} a_i)$ is a non-zero multiple of $\gamma \implies |\, t_1 - \sum_{i\in S} a_i \,| \geq \gamma$. However, by our assumption,

$$t_1 \;\leq\; \sum_{i=1}^{n} a_i \;\implies\; t_1 - \sum_{i\in S} a_i \;\leq\; \sum_{i\in[n]\setminus S} a_i \;<\; 1 + \sum_{i\in[n]} a_i \;=\; \gamma\,.$$

Moreover, $t_1 - \sum_{i\in S} a_i > -\gamma$, holds trivially, since $\gamma > \sum_{i\in[n]} a_i$ and $t_1 > 0$. Therefore, $|\, t_1 - \sum_{i\in S} a_i \,| < \gamma$ which implies that *both* $t_1 - \sum_{i\in S} a_i = 0$ and $t_2 - \sum_{i\in S} b_i = 0$. Hence, the $2-$ SimulSubsetSum instance is also YES. □

C Fast multivariate polynomial multiplication

In this section, we will study the time required to compute

$$A(x_1, \ldots, x_k) \;:\equiv\; \prod_{i=1}^{n} \left(1 + \prod_{j=1}^{k} x_j^{a_{ij}}\right) \bmod \langle x_1^{t_1+1}, x_2^{t_2+1}, \ldots, x_k^{t_k+1}, p\rangle$$

for some prime p. The case when $k = 1$ has been studied in [15, Lemma 5] where the authors gave an $\tilde{O}(n + t_1)$ time algorithm for $p \in [t_1 + 1, (n + t_1)^3]$.

Since the degrees of the variables are different, one needs to be careful while doing multivariate FFT; a loose calculation would give $\tilde{O}((2 \max t_i + 1)^k)$ time solution which is *bad* for us. For the sake of completeness, we give the details. Most of the techniques below are essentially adapted and generalized from [15].

Below, we always work with the field $\mathbb{F}_p$, for a suitably chosen prime p. For the SimulSubsetSum with k many SSUM instances, let the target values be $t_1, \ldots, t_k$. Define $N := 2t_k \cdot \prod_{i=1}^{k-1}(2t_i + 1)$. Choose a prime $p \in [N+1, (n+N)^3]$. Since $\log p = O(\log(n + \prod_{i=1}^{k}(2t_i + 1)))$, it can be taken inside $\tilde{O}$, in all the time-complexity expressions.

Lemma 5 (Fast multivariate exponentiation). *Let* $\boldsymbol{x} = (x_1, \ldots, x_k)$ *and* $f(\boldsymbol{x}) = \sum_{i=1}^{t_1} f_i(\boldsymbol{x}) \cdot x_1^i \in \mathbb{F}_p[\boldsymbol{x}]$ *where* $f_i(\boldsymbol{x}) \in \mathbb{F}_p[x_2, \ldots, x_k]$ *such that*

1. $f(\boldsymbol{x}) \mod \langle x_1, \ldots, x_k\rangle = 0$*, i.e. the constant term of* $f(\boldsymbol{x})$ *is* 0*, and,*
2. $\deg_{x_j}(f) = t_j$*, for positive integers* t_j*.*

Then, there is an $\tilde{O}(\prod_{i=1}^{k}(2t_i + 1))$ *time deterministic algorithm that computes a polynomial* $g(\boldsymbol{x}) \in \mathbb{F}_p[\boldsymbol{x}]$ *such that* $g(\boldsymbol{x}) \equiv \exp(f(\boldsymbol{x})) \bmod \langle x_1^{t_1+1}, \ldots, x_k^{t_k+1}\rangle$ *over* $\mathbb{F}_p$*.*

Proof. Let $g(\boldsymbol{x}) = \exp(f(\boldsymbol{x})) = \sum_{i=0}^{\infty} g_i(x_2, \ldots, x_k) \cdot x_1^i$, where $g_i \in \mathbb{F}_p[[x_2, \ldots, x_k]]$. Differentiate wrt x_1 to get:

$$g'(\boldsymbol{x}) \;:=\; \frac{\partial g(\boldsymbol{x})}{\partial x_1} \;=\; g(\boldsymbol{x}) \cdot \frac{\partial f(\boldsymbol{x})}{\partial x_1}\,.$$

By comparing the coefficients of x_1^i on both sides, we get (over $\mathbb{F}_p$):

$$g_i \equiv i^{-1} \cdot \sum_{j=0}^{i-1} f_{i-j} \cdot g_j \mod \langle x_2^{t_2+1}, \ldots, x_k^{t_k+1} \rangle ,$$

□

where $g_0 = 1$. By initializing $g_0 = 1$, the rest g_i to 0 and calling $Compute(0, t_1)$ procedure in algorithm 3, we can compute all the coefficients up to $x_1^{t_1}$, in the polynomial $g(\boldsymbol{x}) \mod \langle x_2^{t_2+1}, \ldots, x_k^{t_k+1} \rangle$, over $\mathbb{F}_p$.

Algorithm 3: Algorithm for $Compute(\ell, r)$

Input: integers ℓ, r and polynomials f_i, g_i
Output: Updated values of g_i

```
if ℓ < r then
    m = ⌊(ℓ + r)/2⌋;
    Compute(ℓ, m);
    for i ∈ {m + 1, ..., r} do
        g_i = g_i + i^{-1} Σ_{j=ℓ}^{m} (i − j) f_{i−j} g_j  mod ⟨x_2^{t_2+1}, x_3^{t_3+1}, ..., x_k^{t_k+1}, p⟩;
    end
    Compute(m + 1, r);
end
return g_ℓ, ..., g_r;
```

To speed up this algorithm, we can set $A(\boldsymbol{x}) = \sum_{i=0}^{r-\ell} i f_i x_1^i$ and $B(\boldsymbol{x}) = \sum_{i=0}^{m-\ell} g_{i+\ell} x_1^i$; here the f_i and $g_{i+\ell}$ have been computed modulo $\langle x_2^{t_2+1}, \ldots, x_k^{t_k+1} \rangle$ already. Use multidimensional FFT [7, Chapter 30] to compute $C(\boldsymbol{x}) = A(\boldsymbol{x})B(\boldsymbol{x})$ to speed up the for loop which takes $O(\prod_{i=1}^{k}(2t_i+1)\log(\prod_{i=1}^{k}(2t_i+1)))$ time.

Observe that $\sum_{j=\ell}^{m}(i-j)f_{i-j}g_j$ is the coefficient of $x_1^{i-\ell}$ in $C(\boldsymbol{x})$; importantly $\deg_{x_i}(C) \leq 2t_i$, for $i \geq 2$. The extraction of the coefficient of x_1^i in $C(\boldsymbol{x})$ for all i, mod $\langle x_2^{t_2+1}, x_3^{t_3+1}, \ldots, x_k^{t_k+1} \rangle$ can be performed in $O(\prod_{i=1}^{k}(2t_i+1))$ time. This is done by traversing through the polynomial and collecting coefficient along with monomials having the same x_1^i term (and there can be at most $\prod_{i=2}^{k}(2t_i+1)$ many terms). Thus, the total time complexity of computing $g(\boldsymbol{x}) \mod \langle x_2^{t_2+1}, x_3^{t_3+1}, \ldots, x_k^{t_k+1} \rangle$ is

$$\begin{aligned} T(t_1, t_2, \ldots, t_k) &= 2T(t_1/2, t_2, \ldots, t_k) + \tilde{O}(\prod_{i=1}^{k}(2t_i+1)) \\ &= \tilde{O}(\prod_{i=1}^{k}(2t_i+1)) , \end{aligned}$$

as desired. □

Lemma 6 (Fast logarithm computation). *Let* $A(\boldsymbol{x}) = \prod_{i=1}^{n}\left(1+\prod_{j=1}^{k} x_j^{a_{ij}}\right)$. *Then, there exists an* $\tilde{O}(kn+\prod_{i=1}^{k} t_i)$ *time deterministic algorithm that computes* $\mathrm{coef}_{\boldsymbol{x}^{\boldsymbol{e}}}(\ln(A(\boldsymbol{x})))\mod p$ *for all* $\boldsymbol{e}$, *such that* $\boldsymbol{e}=(e_1,\ldots,e_k)$ *with* $e_i \le t_i$.

Proof. Let us define $B(\boldsymbol{x}) := \ln(A(\boldsymbol{x}))$. Then,

$$\begin{aligned} B(\boldsymbol{x}) &= \ln\left(\prod_{i=1}^{n}(1+\prod_{j=1}^{k} x_j^{a_{ij}})\right) \\ &= \sum_{i=1}^{n}\ln\left(1+\prod_{j=1}^{k} x_j^{a_{ij}}\right) \\ &= \sum_{i=1}^{n}\sum_{\ell=1}^{\infty}\left(\frac{(-1)^{\ell-1}}{\ell}(\prod_{j=1}^{k} x_j^{a_{ij}})^{\ell}\right) . \end{aligned}$$

Without loss of generality, we can assume that $t_1 \le t_i, \forall i>1$. Let $C(\boldsymbol{x}) := B(\boldsymbol{x}) \mod \langle x_1^{t_1+1},\ldots,x_k^{t_k+1},p\rangle$. Since, we are interested where the individual degree of x_j can be at most t_j, the index ℓ in the above equation (for a fixed i) must satisfy $a_{ij}\cdot\ell \le t_j$ for each $j\in[k]$. This implies $\ell \le t_j/a_{ij}$, for $j\in[k]$. Therefore, define $M_i := min_{j=1}^{k}\lfloor t_j/a_{ij}\rfloor$. Now, one can express $C(\boldsymbol{x})$ using M_i since it suffices to look the index ℓ till M_i (for a fixed i), as argued before.

Importantly, note that the above equation involves $\prod_{j=1}^{k} x_j^{a_{ij}\ell}$ which has individual degree >0, since both $a_{ij},\ell \ge 1$. Thus, define $T := \{\boldsymbol{e}=(e_1,\ldots,e_k)\in \mathbb{Z}^k \mid 1\le e_i\le t_i, \forall i\in[k]\}$. Then,

$$\begin{aligned} C(\boldsymbol{x}) &= \sum_{i=1}^{n}\sum_{\ell=1}^{M_i}\left(\frac{(-1)^{\ell-1}}{\ell}(\prod_{j=1}^{k} x_j^{a_{ij}})^{\ell}\right) \\ &= \sum_{\boldsymbol{e}\in T}\sum_{\ell=1}^{t_1/e_1}\left(\frac{s_{\boldsymbol{e}}\times(-1)^{\ell-1}}{\ell}\prod_{j=1}^{k} x_j^{e_i\ell}\right) , \end{aligned}$$

where $s_{\boldsymbol{e}} = |\{i\in[n]\mid (a_{i1},\ldots,a_{ik})=\boldsymbol{e}\}|$. Essentially, for a given $s_{\boldsymbol{e}}$, the quantity computes how many times a_{ij} is equal to e_j, for all $j\in[k]$. Using $s_{\boldsymbol{e}}$, we can interchange the order of the summation as shown above. Moreover, we can pre-compute $s_{\boldsymbol{e}}$, for all $\boldsymbol{e}\in T$ in time $O\left(kn+\prod_{i=1}^{k} t_i\right)$.

Observe that $\mathrm{coef}_{\boldsymbol{x}^{\boldsymbol{e}}}(B(\boldsymbol{x})) = \mathrm{coef}_{\boldsymbol{x}^{\boldsymbol{e}}}(C(\boldsymbol{x}))$, for any $(e_1,\ldots,e_k)\in T$. Since, $\ell \le t_1 < p$, ℓ^{-1} exists and can be pre-computed in $\tilde{O}(t_1)$.

▶ Time complexity. Observe that we have

$$C(\boldsymbol{x}) = \sum_{\boldsymbol{e}\in T}\sum_{\ell=1}^{t_1/e_1}\left(\frac{s_{\boldsymbol{e}}\times(-1)^{\ell-1}}{\ell}\prod_{j=1}^{k} x_j^{e_i\ell}\right)$$

$$= \sum_{e_2=1}^{t_2} \sum_{e_3=1}^{t_3} \cdots \sum_{e_k=1}^{t_k} \left(\sum_{e_1=1}^{t_1} \sum_{\ell=1}^{t_1/e_1} \left(\frac{s_{\boldsymbol{e}} \times (-1)^{\ell-1}}{\ell} \prod_{j=1}^{k} x_j^{e_i \ell} \right) \right)$$

The time taken to compute all $\mathrm{coef}_{\boldsymbol{x}^{\boldsymbol{e}}}(C(\boldsymbol{x}))$, given $s_{\boldsymbol{e}}$, is the number of iterations over all $(e_2, \ldots, e_k)$, for $1 \leq e_i \leq t_i$, $i > 1$ and $\ell \in [t/e_1]$, which is at most $\sum_{j=1}^{t_1} \lfloor t_1/j \rfloor \times \prod_{i=2}^{k} t_i = \tilde{O}(\prod_{i=1}^{k} t_i)$, since $\sum_{j=1}^{t_1} t_1/j = O(t_1 \log t_1)$. Thus, the total time is $\tilde{O}(kn + \prod_{i=1}^{k} t_i)$. □

D Dynamic programming approach for Subset Product

In this section, we will briefly discuss the modification to Bellman's dynamic programming approach for SSUM to solve Subset Product in deterministic (expected) time $O(nt^{o(1)})$.

The algorithm starts by removing all a_i that does not divide t. Then using the factoring algorithm in [18], we can factor t into prime factor p_j, i.e., $t = \prod_{i \in [k]} p_j^{t_j} = t$, where $k = O(\log(t)/\log\log(t))$. We now compute the DP table T of size $n \times t_1 \times \cdots \times t_k$ such that

$T[i, x_1, \ldots, x_k] = 1$, if and only if there exists $S \in [i]$, such that $\prod_{j \in S} a_j = \prod_{j \in [k]} p_j^{x_j}$.

Observe that the time complexity of the algorithm is the time taken to populate the DP table with either 1 or 0. Since the size of the DP table is $n \times \prod_{i \in [k]} (1 + t_i)$, using the similar analyse mentioned in subsection 3.1, we can bound the term $\prod_{i \in [k]} (1 + t_i)$ by $t^{o(1)}$. Therefore, the total time complexity is $O(nt^{o(1)})$.