0-1 Knapsack in Nearly Quadratic Time[*]

Ce Jin[†]
cejin@mit.edu
MIT
Cambridge, MA, USA

ABSTRACT

We study pseudo-polynomial time algorithms for the fundamental *0-1 Knapsack* problem. Recent research interest has focused on its fine-grained complexity with respect to the number of items n and the *maximum item weight* $w_{\max}$. Under $(\min,+)$-convolution hypothesis, 0-1 Knapsack does not have $O((n+w_{\max})^{2-\delta})$ time algorithms (Cygan-Mucha-Węgrzycki-Włodarczyk 2017 and Künnemann-Paturi-Schneider 2017). On the upper bound side, currently the fastest algorithm runs in $\widetilde{O}(n+w_{\max}^{12/5})$ time (Chen, Lian, Mao, and Zhang 2023), improving the earlier $O(n+w_{\max}^3)$-time algorithm by Polak, Rohwedder, and Węgrzycki (2021).

In this paper, we close this gap between the upper bound and the conditional lower bound (up to subpolynomial factors): The 0-1 Knapsack problem has a deterministic algorithm in $O(n+w_{\max}^2\log^4 w_{\max})$ time.

Our algorithm combines and extends several recent structural results and algorithmic techniques from the literature on knapsack-type problems:

(1) We generalize the "fine-grained proximity" technique of Chen, Lian, Mao, and Zhang (2023) derived from the additive-combinatorial results of Bringmann and Wellnitz (2021) on dense subset sums. This allows us to bound the support size of the useful partial solutions in the dynamic program.

(2) To exploit the small support size, our main technical component is a vast extension of the "witness propagation" method, originally designed by Deng, Mao, and Zhong (2023) for speeding up dynamic programming in the easier unbounded knapsack settings. To extend this approach to our 0-1 setting, we use a novel pruning method, as well as the two-level color-coding of Bringmann (2017) and the SMAWK algorithm on tall matrices.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**.

KEYWORDS

knapsack, dynamic programming, additive combinatorics

[*]The full version of this paper is available at https://arxiv.org/abs/2308.04093.
[†]Supported by NSF grants CCF-2129139, CCF-2127597, and a Siebel Scholarship.

ACM Reference Format:
Ce Jin. 2024. 0-1 Knapsack in Nearly Quadratic Time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC '24), June 24–28, 2024, Vancouver, BC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3618260.3649618

1 INTRODUCTION

In the *0-1 Knapsack* problem, we are given a knapsack capacity $t\in\mathbb{Z}^+$ and n items $(w_1,p_1),\dots,(w_n,p_n)$, where $w_i,p_i\in\mathbb{Z}^+$ denote the *weight* and *profit* of the i-th item, and we want to select a subset $X\subseteq[n]$ of items satisfying the capacity constraint $W(X):=\sum_{i\in X}w_i\le t$, while maximizing the total profit $P(X):=\sum_{i\in X}p_i$.

Knapsack is a fundamental problem in computer science.[1] It is among Karp's 21 NP-complete problems [43], and the fastest known algorithm runs in $O(2^{n/2}n)$ time [39, 57]. However, when the input integers are small, it is more preferable to use *pseudopolynomial time* algorithms that have polynomial time dependence on both n and the input integers. Our work focuses on this pseudopolynomial regime. A well-known example of pseudopolynomial algorithms is the textbook $O(nt)$-time Dynamic Programming (DP) algorithm for Knapsack, given by Bellmann [7] in 1957. Finding faster pseudopolynomial algorithms for Knapsack became an important topic in combinatorial optimization and operation research; see the book of Kellerer, Pferschy, and Pisinger [45] for a nice summary of the results known by the beginning of this century. In the last few years, research on Knapsack (and the easier Subset Sum problem, which is the special case of Knapsack where $p_i=w_i$) has been revived by recent developments in fine-grained complexity (e.g, [2, 6, 10, 29, 48, 49]) and integer programming (e.g., [33, 54]), and the central question is to understand the best possible time complexities for solving these knapsack-type problems.

Cygan, Mucha, Węgrzycki, and Włodarczyk [29] and Künnemann, Paturi, and Schneider [49] showed that the $O(nt)$ time complexity for Knapsack is essentially optimal (in the regime of $t=\Theta(n)$) under the $(\min,+)$-convolution hypothesis. To cope with this hardness result, recent interest has focused on parameterizing the running time in terms of n and the *maximum item weight* $w_{\max}$ (or the *maximum item profit* $p_{\max}$), instead of the knapsack capacity t. This would be useful when the item weights are much smaller than the capacity, and results along this line would offer us a more fine-grained understanding of knapsack-type problems. This parameterization is also natural from the perspective of integer linear programming (e.g., [33]): when formulating Knapsack as an integer linear program, the maximum item weight $w_{\max}$ corresponds to the standard parameter Δ, maximum absolute value in the input matrix.

[1]In this paper we use the term Knapsack to refer to 0-1 Knapsack (as opposed to other variants such as Unbounded Knapsack and Bounded Knapsack).

However, despite extensive research on 0-1 Knapsack along these lines, our understanding about the dependence on $w_{\max}$ is still incomplete. Known fine-grained lower bounds only ruled out $(n + w_{\max})^{2-\delta}$ algorithms for Knapsack [29, 49] (for $\delta > 0$). In comparison, Bellman's dynamic programming algorithm only runs in $O(nt) \le O(n^2 w_{\max})$ time. Several papers obtained the bound $\widetilde{O}(nw_{\max}^2)$ via various methods [5, 6, 33, 44].[2] Polak, Rohwedder, and Węgrzycki [54] carefully combined the *proximity technique* of Eisenbrand and Weismantel [33] from integer programming with the concave $(\max, +)$-convolution algorithm ([44] or [3]), and obtained an $O(n + w_{\max}^3)$ algorithm for Knapsack. These algorithms have cubic dependence on $(n + w_{\max})$. Finally, the very recent work by Chen, Lian, Mao, and Zhang [25] broke this cubic barrier with an $\widetilde{O}(n + w_{\max}^{12/5})$-time algorithm, which was based on additive-combinatorial results of Bringmann and Wellnitz [16].[3]

None of the above algorithms match the $(n + w_{\max})^{2-o(1)}$ conditional lower bound. The following question has been asked by [12, 25, 54]:

Main question: Can 0-1 Knapsack be solved in $\widetilde{O}(n + w_{\max}^2)$ time?

We remark that this $\widetilde{O}(n + w_{\max}^2)$ running time is known to be achievable for the easier *Unbounded Knapsack* problem (where each item has infinitely many copies available) [5, 21, 31], matching the $(n + w_{\max})^{2-\delta}$ conditional lower bound for Unbounded Knapsack [29, 49]. As argued by [54], the 0-1 setting appears to be much more difficult, and most of the techniques for Unbounded Knapsack do not appear to apply to the 0-1 setting.

1.1 Our Contribution

In this paper, we affirmatively resolve this main question, closing the gap between the previous $\widetilde{O}(n + w_{\max}^{12/5})$ upper bound [25] and the quadratic conditional lower bound [29, 49].

Theorem 1.1. *The 0-1 Knapsack problem can be solved by a deterministic algorithm with time complexity $O(n + w_{\max}^2 \log^4 w_{\max})$.*

In our paper we only describe an algorithm that outputs the total profit of the optimal knapsack solution. It can be modified to output an actual solution using the standard technique of back-pointers, without affecting the asymptotic time complexity.

By a reduction described in [54, Section 4], we have the following corollary which parameterizes the running time by the largest item profit $p_{\max}$ instead of $w_{\max}$.

Corollary 1.2. *The 0-1 Knapsack problem can be solved by a deterministic algorithm with time complexity $O(n + p_{\max}^2 \log^4 p_{\max})$.*

Independent works. Independently and concurrently to our work, Bringmann [11] also obtained an $\widetilde{O}(n + w_{\max}^2)$ time algorithm for 0-1 Knapsack (more generally, Bounded Knapsack).

Chronological remarks. The current paper is a substantially updated version of an earlier manuscript (posted to arXiv in July 2023). This earlier manuscript contained much weaker results, and is obsolete now. Our current paper incorporates part of the techniques from our earlier manuscript, and also builds on the very recent work by Chen, Lian, Mao, and Zhang [25] (posted to arXiv in July 2023).

1.2 Technical Overview

Our Knapsack algorithm combines and extends several recent structural results and algorithmic techniques from the literature on knapsack-type problems. In particular, we crucially build on the techniques from two previous papers by Chen, Lian, Mao, and Zhang [25], and by Deng, Mao, and Zhong [31]. Now we review the techniques in prior works and describe the ideas behind our improvement.

Fine-grained proximity based on additive combinatorics. There was a long line of work in the 80's and 90's on designing Subset Sum algorithms using techniques from *additive combinatorics* [17–19, 34, 35, 37], and more recently these techniques have been revived and applied to not only Subset Sum [16, 48, 52, 54] but also the more difficult Knapsack problem [25, 30]. Ultimately, these algorithms directly or indirectly rely on the following powerful result in additive combinatorics, pioneered by Freiman [36] and Sárközy [56] and tightened by Szemerédi and Vu [58], and more recently strengthened by Conlon, Fox, and Pham [27]: Let $\mathcal{S}(A) = \{\sum_{b \in B} b : B \subseteq A\}$ denote the subset sums of A. Then, if set $A \subseteq [N]$ has size $|A| \gg \sqrt{N}$, then $\mathcal{S}(A)$ contains an arithmetic progression of length N (and this arithmetic progression is homogeneous, meaning that each element is an integer multiple of the common difference).

Another technique used in recent knapsack algorithms is the *proximity technique* from the integer programming literature, see e.g., [28, 33]. When specialized to the Knapsack case (1-dimensional integer linear program), a proximity result refers to a distance upper bound between the optimal knapsack solution and the *greedy solution* (sort items in decreasing order of efficiencies p_i/w_i, and take the maximal prefix without violating the capacity constraint). Polak, Rohwedder, and Węgrzycki [54] exploited the fact that these two solutions differ by at most $O(w_{\max})$ items, which allowed them to shrink the size of the dynamic programming (DP) table from t down to $O(w_{\max}^2)$ (by performing DP on top of the greedy solution to find an optimal exchange solution). They achieved $O(n + w_{\max}^3)$ time by batch-updating items of the same weight w using the SMAWK algorithm [3] (see also [5, 44]).

The very recent paper by Chen, Lian, Mao, and Zhang [25] developed a new "fine-grained proximity" technique that combines these two lines of approach. They used the additive-combinatorial results of Bringmann and Wellnitz [16] (which built on works of Sárközy [55, 56] and Galil and Margalit [37]) to obtain several powerful structural lemmas involving the support size of two multisets A, B (with integers from $[w_{\max}]$) that avoid non-zero common subset sums, and these structural lemmas were translated into proximity results using exchange arguments. These fine-grained proximity results of [25] are more powerful than the earlier proximity bounds used in [33, 54]; the following lemma from [25] is one example: given a Knapsack instance, we can partition the item weights into two subsets $[w_{\max}] = \mathcal{W}_1 \uplus \mathcal{W}_2$, such that $|\mathcal{W}_1| \le \widetilde{O}(\sqrt{w_{\max}})$, and the differing items between the greedy solution and the optimal solution whose weights belong to $\mathcal{W}_2$ can only have total weight $O(w_{\max}^{3/2})$. This lemma immediately led to a simple $\widetilde{O}(n + w_{\max}^{5/2})$

[2]We use $\widetilde{O}(f)$ to denote $O(f \operatorname{poly}\log f)$.
[3]An earlier work by Bringmann and Cassis [13] obtained an algorithm in $\widetilde{O}(nw_{\max}p_{\max}^{2/3})$ time, which was the first algorithm for 0-1 Knapsack with subcubic dependence on $(n + w_{\max} + p_{\max})$.

algorithm [25]. A bottleneck step in this algorithm is to use DP to compute partial solutions consisting of items with weights from $\mathcal{W}_1$: they need to perform the batch DP update (based on SMAWK) $|\mathcal{W}_1|$ times, and the size of the DP table is still $O(w_{\max}^2)$ as in [54], so the total time for this step is $\widetilde{O}(w_{\max}^{2.5})$. To overcome this bottleneck, [25] used more refined proximity results based on the multiplicity of item weights, and obtained an improved running time $\widetilde{O}(n+w_{\max}^{2.4})$.

DP strategy based on multiplicity. In our work, we completely overcome this bottleneck of [25]: we can implement the DP for items with weights from $\mathcal{W}_1$ in only $\widetilde{O}(w_{\max}^2)$ time. This is the main technical part of our paper. (The other bottleneck in [25]'s simple $\widetilde{O}(n + w_{\max}^{5/2})$ time algorithm is to deal with items whose weights come from $\mathcal{W}_2$, but this part can be improved more easily by dividing into $O(\log w_{\max})$ partitions with smoothly changing parameters. See Section 3.1.) Now we give an overview of our improvement.

We rely on another additive-combinatorial lemma (Lemma 3.5) which can be derived from the results of Bringmann and Wellnitz [16]; it is analogous and inspired by the fine-grained proximity results of [25], but is not directly comparable to theirs. It implies the following proximity result: Let D denote the set of differing items between the greedy solution and the optimal solution. Then, for any $r \geq 1$, there can be at most $\widetilde{O}(\sqrt{w_{\max}/r})$ many weights $w \in [w_{\max}]$ such that D contains at least r items of weight w (i.e., w has multiplicity $\geq r$ in the item weights of D). In other words, if we figuratively think of the histogram of the weights of items in D, then the number of columns in the histogram with height $\geq r$ should be at most $\widetilde{O}(\sqrt{w_{\max}/r})$. As a corollary, the total area below height r in this histogram is at most $\sum_{r'=1}^{r} \widetilde{O}(\sqrt{w_{\max}/r'}) = \widetilde{O}(\sqrt{r w_{\max}})$.

Our DP algorithm exploits the aforementioned structure of D as follows. We perform the DP in $O(\log w_{\max})$ phases, where in the j-th phase ($j \geq 1$) we update the current DP table with all items of *rank* in $[2^{j-1}, 2^j)$. Here, the *rank* of a weight-w item is defined as the rank of its profit among all weight-w items (an item with rank 1 is the most profitable item among its weight class). By the end of phase j, our DP table should contain the partial solution consisting of all items in D of rank $< 2^j$, i.e., the partial solution that corresponds to the part below height 2^j in the histogram representing D. As we mentioned earlier, this partial solution only has $\widetilde{O}(\sqrt{2^j w_{\max}})$ items, and hence $\widetilde{O}(w_{\max} \cdot \sqrt{2^j w_{\max}})$ total weight, so the size of the DP table at the end of phase j only needs to be $L_j := \widetilde{O}(w_{\max} \cdot \sqrt{2^j w_{\max}})$.

To efficiently implement the DP in each phase, we need to crucially exploit the aforementioned fact that the number of weights $w \in \mathcal{W}_1$ with multiplicity $\geq 2^{j-1} - 1$ in D is at most $b_j := \widetilde{O}(\sqrt{w_{\max}/2^j})$. (Note that in phase $j = 1$ this threshold is $2^{j-1} - 1 = 0$, and the upper bound $b_1 = \widetilde{O}(\sqrt{w_{\max}})$ simply follows from $|\mathcal{W}_1| = \widetilde{O}(\sqrt{w_{\max}})$ guaranteed by [25]'s partition.) Our goal is to perform each phase of the DP updates in $\widetilde{O}(b_j \cdot L_j) = \widetilde{O}(w_{\max}^2)$ time. To achieve this goal, we surprisingly adapt a recent technique introduced in the much easier *unbounded* knapsack settings by Deng, Mao, and Zhong [31], called "witness propagation". In the following we briefly review this technique.

Transfer of techniques from the unbounded setting. The *unbounded* knapsack/subset sum problems, where each item has infinitely many copies available, are usually easier for two main reasons: (1) Since there are infinite supply of items, we do not need to keep track of which items are used so far in the DP. (2) There are more powerful structural results available, in particular the Carathéodory-type theorems [21, 31, 32, 46] which show the existence of optimal solution vectors with only logarithmic support size.

Deng, Mao, Zhong [31] recently exploited the small support size to design near-optimal algorithms for several unbounded-knapsack-type problems, based on their key new technique termed "witness propagation". The idea is that, since the optimal solutions must have small support size (but possibly with high multiplicity), one can first prepare the "base solutions", which are partial solutions with small support and multiplicity at most one. Then, they gradually build full solutions from these base solutions, by "propagating the witnesses" (that is, increase the multiplicity of some item with non-zero multiplicity). The time complexity of this approach is low since the support sizes are small.

Now we come back to our DP framework for 0-1 knapsack described earlier, and observe that we are in a very similar situation to the unbounded knapsack setting of [31]. In our case, if we intuitively view our DP as gradually growing the columns of the histogram representing D, then after phase $j - 1$, there can be only $\leq b_j$ columns in the histogram that may continue growing in subsequent phases. This means the "active support" of our partial solutions has size $\leq b_j$: when we extend a partial solution in the DP table during phase j, we only need to consider items from b_j many weight classes, namely those weights that have "full multiplicity" in this partial solution by the end of phase $j - 1$. (If there are more than b_j many such weights, then the proximity result implies that this partial solution cannot be extended to the optimal solution, and we can safely discard it.) This gives us hope of implementing the DP of each phase in $\widetilde{O}(b_j \cdot L_j) = \widetilde{O}(w_{\max}^2)$ using the witness propagation idea from [31].

However, we still need to overcome several difficulties that arise from the huge difference between 0-1 setting and unbounded setting. In particular, the convenient property (1) for unbounded knapsack mentioned above no longer applies to the 0-1 setting. In the following we briefly explain how we implement the witness propagation idea in the 0-1 setting.

Witness propagation in the 0-1 *setting.* In each phase j of our DP framework, we are faced with the following task (from now on we drop the subscript j and denote $b = b_j, L = L_j$): we are given a DP table $q[\,]$ of size L, in which each entry $q[z]$ is associated with a set $S[z] \subseteq \mathcal{W}_1$ of size $|S[z]| \leq b$ (this is the "active support" of the partial solution corresponding to $q[z]$). For each entry $q[z]$, we would like to extend this partial solution by adding items whose weights come from $S[z]$. More specifically, letting $x_w \geq 0$ denote the number of weight-w items to add (where $w \in S[z]$), we should update the final DP table entry $q'[z + \sum_{w\in S[z]} x_w w]$ with the new profit $q[z] + \sum_{w\in S[z]} Q_w(x_w)$. Here $Q_w(x)$ is the total profit of the top x remaining items of weight w (note that $Q_w(\cdot)$ is concave). Our goal is to compute the final DP table $q'[\,]$ (which should capture the optimal ways to extend from $q[\,]$) in $\widetilde{O}(bL)$ time. (Note that in the idealistic setting where all $S[z]$ are contained in a common superset

$\hat{S}$ of size $|\hat{S}| \le b$, this task can be solved via standard applications of SMAWK in $O(bL)$ total time in the same way as [5, 44, 54]. The key challenge in our setting is that, although each $S[z]$ has size $\le b$, their union over all z may have much more than b types of weights.)

We first focus on an interesting basic case where each set $S[z]$ has size at most $b = 1$. In this case, for each DP table entry $q[z]$ with $S[z] = \{w\}$ we would like to perform the DP update $q'[i] \leftarrow \max(q'[i], q[z] + Q_w((i-z)/w))$ for all i such that $i \ge z$ and $i \equiv z \pmod{w}$. Similarly to [5, 44, 54], we try to use the SMAWK algorithm to perform these DP updates. However, since these sets $S[z]$ may contain different types of weights w, we need to deal with them separately. This means that for each weight w, there may be only sublinearly many indices z with $S[z] = \{w\}$. Hence, in order to save time, we need to do SMAWK for each w in time complexity sublinear in the entire DP table size L, and only near-linear in $n_w = \left|\{z : S[z] = \{w\}\}\right|$. So we need to let SMAWK return a compact output representation, which partitions the DP table into n_w segments, or more precisely, n_w arithmetic progressions (APs) of difference w, where each $z \in \{z : S[z] = \{w\}\}$ is associated with an AP containing the indices i for which $q'[i]$ is maximized by $q[z] + Q_w((i-z)/w)$. This is an very interesting scenario where we actually need to use the tall-matrix version of SMAWK.

Then, we need to update these APs returned by these SMAWK algorithm invocations (for various different weights w) to the DP table $q'[\,]$. That is, for each i, we would like to pick the AP that contains i and maximizes the profit $q[z] + Q_w((i-z)/w)$ mentioned earlier. Naively going through each element in every AP would take time proportional to the total length of these APs. This would be too slow: although the total number of APs is only $O(L)$, their total length could still be very large. To solve this issue, we design a novel skipping technique, so that we can ignore suffixes of some of the APs, while still ensuring that we do not lose the optimal solution, so that the total time is reduced to $\widetilde{O}(L)$.

We first explain the key insight behind our skipping technique, through the following example. Suppose index i is contained in two APs computed by SMAWK for two different weights $w_1 \ne w_2$, denoted by $I_1 = \{z_1 + xw_1 : \ell_1 \le x \le r_1\}$ and $I_2 = \{z_2 + xw_2 : \ell_2 \le x \le r_2\}$. The final DP table entry $q'[i]$ is updated using $\max\{q[z_1] + Q_{w_1}((i-z_1)/w_1), q[z_2] + Q_{w_2}((i-z_2)/w_2)\}$, and we suppose the first option is larger. Then, we claim that all elements in $I_2 \cap (i, +\infty)$ are useless. To see this, consider any $i^* \in I_2 \cap (i, +\infty)$, and denote $i^* = z_2 + x^* w_2$, $i = z_2 + x w_2$ $(x^* > x)$, so $i^* \in I_2$ represents a solution of total weight i^* and profit $q[z_2] + Q_{w_2}(x^*)$. However, we can show this solution represented by $i^* \in I_2$ is dominated by another solution defined as follows: add $(x^* - x)$ many weight-w_2 items to the solution represented by $i \in I_1$, achieving the same total weight $i + (x^* - x)w_2 = i^*$ but higher (or equal) total profit $q[z_1] + Q_{w_1}((i-z_1)/w_1) + Q_{w_2}(x^* - x) \ge q[z_2] + Q_{w_2}(x) + Q_{w_2}(x^* - x) \ge q[z_2] + Q_{w_2}(x^*)$ (recall $Q_{w_2}(\cdot)$ is concave). Hence, we can safely ignore the solution represented by $i^* \in I_2$ without affecting optimality.[4,5]

[4]We need more tie-breaking arguments to deal with the possibility that $i^* \in I_2$ is not strictly dominated (i.e., they have equal profit). We omit them in this overview.

[5]The solution we showed that dominates the solution represented by $i^* \in I_2$ is not represented by any AP element, as it uses items of both types of weights w_1 and w_2. It is possible that $i^* \in I_2$ is still the best weight-i^* solution among those represented by the AP elements (which use only one type of weight), but it is fine to omit it since eventually it is not useful for the optimal knapsack solution.

The key insight above can be naturally used to design the following skipping technique: We initialize an empty bucket $B[i]$ for each index i in the DP table. For each of the $O(L)$ many APs returned by SMAWK, we insert the (description of the) AP into the bucket indexed by the beginning element of this AP. Then we iterate over the buckets $B[i]$ in increasing order of i. For each $B[i]$, we pick the AP from this bucket that maximizes the profit value at i, and update the profit value $q'[i]$ accordingly. Then, we copy this maximizing AP from bucket $B[i]$ to the bucket indexed by the successor of i in this AP; the other non-maximizing APs in bucket $B[i]$ will not be copied. In this way, the total time is $O(L)$, since we start with $O(L)$ APs and each bucket only copies one AP to another bucket.

Now we briefly describe how to generalize from the $|S[z]| \le 1$ case to $|S[z]| \le b$ for larger b. First we make an ideal assumption that we can partition all possible weights into b parts, $\mathcal{W}_1 = \mathcal{W}^{(1)} \uplus \mathcal{W}^{(2)} \uplus \cdots \uplus \mathcal{W}^{(b)}$, so that $|S[z] \cap \mathcal{W}^{(k)}| \le 1$ for all z and k. In this ideal case, we can iteratively perform b rounds, where in the k-th round we restrict the sets $S[z]$ to $S[z] \cap \mathcal{W}^{(k)}$, and perform the DP updates using the $b = 1$ case algorithm described above in $\widetilde{O}(L)$ time. (Note that after each round we should modify the active supports $S[z]$ accordingly: if $q'[i]$ is updated using $q[z] + Q_w((i-z)/w)$ for some w in this round, then the new $S[i]$ for the next round should be the old $S[z]$.) Hence the total time is $\widetilde{O}(bL)$. In the non-ideal case, we use the two-level color-coding technique originally used by Bringmann [10] in his subset sum algorithm. This technique gives us some properties that are weaker than the ideal assumption but still allow us to apply basically the same idea as the ideal case.

The correctness of our algorithm described above (namely that our skipping technique does not lose the optimal knapsack solution) is intuitive and is based on exchange arguments, but it takes some notations and definitions to formally write down the proof. In the main text of the paper, we formalize the intuition above, and abstract out a core problem called HintedKnapsackExtend$^+$ (Problem 1) that captures the scenario described above in a more modular way, and prove some helper lemmas for Problem 1 (for example, to allow us to decompose an instance with large b to multiple instances with smaller b).

1.3 Further Related Works

In contrast to our 0-1 setting, the *unbounded* setting (where each item has infinitely many copies available) has also been widely studied in the literature of Knapsack and Subset Sum algorithms, e.g., [5, 21, 31, 40, 41, 46, 50].

For the easier Subset Sum problem, an early result for Subset Sum in terms of n and $w_{\max}$ is Pisinger's deterministic $O(nw_{\max})$-time algorithm for Subset Sum [53]. This is not completely subsumed by Bringmann's $\widetilde{O}(n+t) \le \widetilde{O}(nw_{\max})$ time algorithm [10], due to the extra log factors and randomization in the latter result. More recently, Polak, Rohwedder, and Węgrzycki [54] observed that an $\widetilde{O}(n + w_{\max}^2)$ time algorithm directly follows from combining their proximity technique with Bringmann's $\widetilde{O}(n+t)$ Subset Sum algorithm [10]. They improved it to $\widetilde{O}(n + w_{\max}^{5/3})$ time, by further incorporating additive combinatorial techniques by [16]. Very recently, [25] obtained $\widetilde{O}(n + w_{\max}^{3/2})$-time algorithm for Subset Sum, using their fine-grained proximity technique based on additive combinatorial results of [16].

Recently there has also been a lot of work on approximation algorithms for Knapsack and Subset Sum (and Partition) [15, 20, 24, 30, 42, 51, 52]. Prior to this work, the fastest known $(1-\varepsilon)$ approximation algorithm for 0-1 Knapsack had time complexity $\widetilde{O}(n+1/\varepsilon^{2.2})$ [30]. Notably, [30] also used the additive combinatorial results of [16] to design knapsack approximation algorithms; this was the first application of additive combinatorial techniques to knapsack algorithms. In August 2023, Mao [51] and Chen, Lian, Mao, and Zhang [24] independently improved the time complexity to $\widetilde{O}(n+1/\varepsilon^2)$, which is nearly tight under the $(\min,+)$-convolution hypothesis [29, 49].

1.4 Open Problems

There are several interesting open questions.

- In the regime where n is much smaller than $w_{\max}$, can we get faster algorithms for 0-1 Knapsack? The independent work of He and Xu [38] achieved $\widetilde{O}(n^{1.5}w_{\max})$ time. By combining with our result, one can also bound the running time as $\widetilde{O}(n+\min\{n^{1.5}w_{\max}, w_{\max}^2\}) \le \widetilde{O}(nw_{\max}^{4/3})$. Can we achieve $\widetilde{O}(nw_{\max})$ time (which would also match the $(n+w_{\max})^{2-o(1)}$ conditional lower bound based on $(\min,+)$-convolution hypothesis [29, 49])?
- Can we solve 0-1 Knapsack in $O((n+w_{\max}+p_{\max})^{2-\delta})$ time for any constant $\delta > 0$? Bringmann and Cassis [12] gave algorithms of such running time for the easier unbounded knapsack problem. They also showed that such algorithms require computing bounded-difference $(\min,+)$-convolution [22, 26].
- Can we solve 0-1 Knapsack in $O(n + w_{\max}^2/2^{\Omega(\sqrt{\log w_{\max}})})$ time, matching the best known running time for $(\min,+)$-convolution [9, 23, 59]? Algorithms with such running time are known for the easier unbounded knapsack problem [5, 21, 31].
- Can Subset Sum be solved in $\widetilde{O}(n+w_{\max})$ time? This question has been repeatedly asked in the literature [2, 4, 12, 16, 54]. Currently the best result is the very recent $\widetilde{O}(n+w_{\max}^{3/2})$-time randomized algorithm by Chen, Lian, Mao, and Zhang [25].
- Can our techniques be useful for other related problems, such as scheduling [1, 14, 47] or low-dimensional integer linear proramming [33]?

1.5 Paper Organization

Section 2 contains definitions, notations, and some lemmas from previous works, which are essential for understanding Section 3. Then, in Section 3 we describe our algorithm for 0-1 Knapsack. A key subroutine of our algorithm is deferred to Section 4.

2 PRELIMINARIES

2.1 Notations and Definitions

We use $\widetilde{O}(f)$ to denote $O(f \operatorname{poly}\log f)$. Let $[N] = \{1, 2, \ldots, N\}$.

Multisets and subset sums. For an integer multiset X, and an integer x, we use $\mu_X(x)$ to denote the multiplicity of x in X. For a multiset X, the *support* of X is the set of elements it contains, denoted as $\operatorname{supp}(X) := \{x : \mu_X(x) \ge 1\}$. We say a multiset X is *supported on* $[N]$ if $\operatorname{supp}(X) \subseteq [N]$. For multisets A, B we say A is a subset of B (and write $A \subseteq B$) if for all $a \in A$, $\mu_B(a) \ge \mu_A(a)$. We write $A \uplus B$ as the union of A and B by adding multiplicities.

The *size* of a multiset X is $|X| = \sum_{x\in\mathbb{Z}} \mu_X(x)$, and the *sum of elements* in X is $\Sigma(X) = \sum_{x\in\mathbb{Z}} x \cdot \mu_X(x)$. The set of all *subset sums* of X is $\mathcal{S}(X) := \{\Sigma(Y) : Y \subseteq X\}$. We also define $\mathcal{S}^*(X) := \{\Sigma(Y) : Y \subseteq X, Y \ne \varnothing\}$ to be the set of subset sums formed by *non-empty* subsets of X.

The *r-support* of a multiset X is the set of items in X with multiplicity at least r, denoted as $\operatorname{supp}_r(X) := \{x : \mu_X(x) \ge r\}$.

Vectors and arrays. We will work with vectors in $\mathbb{Z}^{\mathcal{I}}$ where $\mathcal{I}$ is some index set. We sometimes denote vectors in boldface, e.g., $\boldsymbol{x} \in \mathbb{Z}^{\mathcal{I}}$, and use non-boldface with subscript to denote its coordinate, e.g., $x_i \in \mathbb{Z}$ (for $i \in \mathcal{I}$). Let $\operatorname{supp}(\boldsymbol{x}) := \{i \in \mathcal{I} : x_i \ne 0\}$, $\|\boldsymbol{x}\|_0 := |\operatorname{supp}(\boldsymbol{x})|$, and $\|\boldsymbol{x}\|_1 := \sum_{i\in\mathcal{I}} |x_i|$. Let $\mathbf{0}$ denote the zero vector. For $i \in \mathcal{I}$, let $\boldsymbol{e}_i$ denote the unit vector with i-th coordinate being 1 and the remaining coordinates being 0.

We use $A[\ell\,..\,r]$ to denote an array indexed by integers $i \in \{\ell, \ell+1, \ldots, r\}$. The i-th entry of the array is $A[i]$. Sometimes we consider arrays of vectors, denoted by $\boldsymbol{x}[\ell\,..\,r]$, in which every entry $\boldsymbol{x}[i] \in \mathbb{Z}^{\mathcal{I}}$ is a vector, and we use $x[i]_j$ to denote the j-th coordinate of the vector $\boldsymbol{x}[i]$ (for $j \in \mathcal{I}$).

0-1 Knapsack. In the 0-1 Knapsack problem with n input items $(w_1, p_1), \ldots, (w_n, p_n)$ (where *weights* $w_i \le w_{\max}$ and *profits* $p_i \le p_{\max}$ are positive integers) and knapsack capacity t, an *optimal knapsack solution* is an item subset $X \subseteq [n]$ that maximizes the total profit

$$P(X) := \sum_{i\in X} p_i, \tag{1}$$

subject to the capacity constraint

$$W(X) := \sum_{i\in X} w_i \le t. \tag{2}$$

We will frequently use the following notations:

- Let $\mathcal{W} = \operatorname{supp}(\{w_1, w_2, \ldots, w_n\}) \subseteq [w_{\max}]$ be the set of input item weights.
- For $\mathcal{W}' \subseteq \mathcal{W}$, let $I_{\mathcal{W}'} := \{i \in [n] : w_i \in \mathcal{W}'\}$ denote the set of items with weights in $\mathcal{W}'$.
- For $I = \{i_1, \ldots, i_{|I|}\} \subseteq [n]$, let $\operatorname{weights}(I) = \{w_{i_1}, \ldots, w_{i_{|I|}}\}$ be the *multiset* of weights of items in I.

We assume $w_{\max} \le t$ by ignoring items that are too large to fit into the knapsack. We assume $w_1 + \cdots + w_n > t$, since otherwise the trivial optimal solution is to include all the items. We assume $w_{\max} \le n^2$, because when $w_{\max} > n^2$ it is faster to run the textbook dynamic programming algorithm [7] in $O(nt) \le O(n \cdot nw_{\max}) \le O(w_{\max}^2)$ time. We use the standard word-RAM computation model with $\Theta(\log n)$-bit words, and we assume $p_i \le p_{\max}$ fits into a single machine word.[6]

[6]If this assumption is dropped, we simply pay an extra $O(\log p_{\max})$ factor in the running time for adding integers of magnitude $(np_{\max})^{O(1)}$.

The *efficiency* of item i is p_i/w_i. We always assume the input items have *distinct* efficiencies p_i/w_i. This assumption is justified by the following tie-breaking lemma proved in the full version.

Lemma 2.1 (Break ties). *Given a Knapsack instance I, in $O(n)$ time we can deterministically reduce it to another Knapsack instance I' with $n, w_{\max}$ and t unchanged, and $p'_{\max} \le \mathrm{poly}(p_{\max}, w_{\max}, n)$, such that the items in I' have* distinct efficiencies *and distinct profits.*

2.2 Greedy Solution and Proximity

Greedy solution. Sort the n input items in decreasing order of efficiency,

$$p_1/w_1 > p_2/w_2 > \cdots > p_n/w_n. \tag{3}$$

The *greedy solution* (or *maximal prefix solution*) is the item subset

$$G = \{1, 2, \ldots, i^*\}, \text{ where } i^* = \max\{i^* : w_1+w_2+\cdots+w_{i^*} \le t\}, \tag{4}$$

i.e., we greedily take the most efficient items one by one, until the next item cannot be added without exceeding the knapsack capacity. Since the input instance is nontrivial, we have $1 \le i^* \le n-1$, and $W(G) \in (t - w_{\max}, t]$. Denote the remaining items as $\overline{G} = [n] \setminus G = \{i^*+1, i^*+2, \ldots, n\}$.

Remark 2.2. As noted by [54], the greedy solution G can be found in deterministic $O(n)$ time using linear-time median finding algorithms [8] (if we only need the set G rather than the order of their elements), as opposed to a straightforward $O(n \log n)$-time sorting according to Eq. (3).

Every item subset $X \subseteq [n]$ can be written as $X = (G \setminus B) \cup A$ where $A \subseteq \overline{G}$ and $B \subseteq G$. Finding an optimal knapsack solution X is equivalent to finding an *optimal exchange solution*, defined as a pair of subsets (A, B) ($A \subseteq \overline{G}, B \subseteq G$) that maximizes $P(A) - P(B)$ subject to $W(A) - W(B) \le t - W(G)$. Since any optimal knapsack solution X satisfies $W(X) \in (t - w_{\max}, t]$, we have

$$0 \le W(A) - W(B) = W(X) - W(G) < w_{\max} \tag{5}$$

for any optimal exchange solution (A, B).

Proximity. For any optimal exchange solution (A, B), a simple exchange argument shows that the weights of items in A and in B do not share any non-zero common subset sum, i.e.,

$$\mathcal{S}^*(\text{weights}(A)) \cap \mathcal{S}^*(\text{weights}(B)) = \varnothing. \tag{6}$$

Indeed, for an optimal knapsack solution $X = (G \setminus B) \cup A$, if non-empty item sets $A' \subseteq A$ and $B' \subseteq B$ have the same total weight, then $(X \cup B') \setminus A'$ is a set of items with the same total weight as X but *strictly* higher total profit (since efficiencies of items in $B' \subseteq G$ are strictly higher than efficiencies of items in $A' \subseteq \overline{G}$ due to Eqs. (3) and (4)), contradicting the optimality of X.

The following proximity bound Eq. (7) is consequence of Eq. (5) and Eq. (6), and was used in previous works such as [25, 54] (see e.g., [54, Lemma 2.1] for a short proof): for any optimal exchange solution (A, B), it holds that

$$|A| + |B| \le 2w_{\max}. \tag{7}$$

In other words, any optimal knapsack solution X differs from the greedy solution G by at most $2w_{\max}$ items. The bound of Eq. (7) immediately implies

$$W(A) + W(B) \le 2w_{\max}^2 \tag{8}$$

for any optimal exchange solution (A, B).

Weight classes and ranks. We rank items of the same weight w according to their profits, as follows:

Definition 2.3 (Rank of items). For each $w \in \mathcal{W}$, consider the weight-w items outside the greedy solution, $\overline{G} \cap I_{\{w\}} = \{i_1, \ldots, i_m\}$, where $p_{i_1} > p_{i_2} > \cdots > p_{i_m}$. We define $\mathrm{rank}(i_1) = 1, \mathrm{rank}(i_2) = 2, \ldots, \mathrm{rank}(i_m) = m$. Similarly, consider the weight-w items in the greedy solution, $G \cap I_{\{w\}} = \{i'_1, i'_2, \ldots, i'_{m'}\}$, where $p_{i'_1} < p_{i'_2} < \cdots < p_{i'_{m'}}$. We define $\mathrm{rank}(i'_1) = 1, \mathrm{rank}(i'_2) = 2, \ldots, \mathrm{rank}(i'_{m'}) = m'$. In this way, every item $i \in [n]$ receives a $\mathrm{rank}(i)$.

Then, a standard observation is that an optimal solution should always take a prefix from each weight class:

Lemma 2.4 (Prefix property). *Consider any optimal exchange solution (A, B). If $i \in A$, then $\{i' \in \overline{G} \cap I_{w_i} : \mathrm{rank}(i') \le \mathrm{rank}(i)\} \subseteq A$, and $\mathrm{rank}(i) \le 2w_{\max}$.*

Similarly, if $i \in B$, then $\{i' \in G \cap I_{w_i} : \mathrm{rank}(i') \le \mathrm{rank}(i)\} \subseteq B$, and $\mathrm{rank}(i) \le 2w_{\max}$.

We defer the proof to the full version.

We remark that all items $i \in [n]$ with $\mathrm{rank}(i) \le 2w_{\max}$ can be deterministically selected and sorted in $O(n + w_{\max}^2 \log w_{\max})$ time using linear-time median selection algorithms [8].

2.3 Dynamic Programming and Partial Solutions

Our algorithm uses dynamic programming (DP) to find an optimal exchange solution (A, B) ($A \subseteq \overline{G}, B \subseteq G$). Now we introduce a few terminologies that will help us describe our DP algorithm later.

Definition 2.5 (Partial solutions and I-optimality). A *partial exchange solution* (or simply a *partial solution*) refers to a pair of item subsets (A', B') where $A' \subseteq \overline{G}, B' \subseteq G$. The *weight* and *profit* of the partial solution (A', B') are defined as $W(A') - W(B')$ and $P(A') - P(B')$ respectively.

Let $I \subseteq [n]$ be an item subset. We say the partial solution (A', B') is *supported on I* if $A' \cup B' \subseteq I$. We say (A', B') is *I-optimal*, if there exists an optimal exchange solution (A, B) such that $A' = A \cap I$ and $B' = B \cap I$.

Definition 2.6 (DP tables). A *DP table of size L* is an array $q[-L \,..\, L]$ with entries $q[z] \in \mathbb{Z} \cup \{-\infty\}$ for $z \in \{-L, \ldots, L\}$.[7] (By convention, assume $q[z] = -\infty$ for $|z| > L$.) We omit its index range and simply write $q[\,]$ whenever its size is clear from context or is unimportant.

For an item subset $I \subseteq [n]$, we say $q[\,]$ is an *I-valid DP table*, if for every entry $q[z] \ne -\infty$ there exists a corresponding partial solution (A', B') supported on I with weight $W(A') - W(B') = z$ and profit $P(A') - P(B') = q[z]$. An I-valid DP table $q[\,]$ is said to be *I-optimal* if it contains some entry $q[z]$ that corresponds to an I-optimal partial solution.

For example, the trivial DP table with $q[0] = 0, q[z] = -\infty (z \ne 0)$ is $\varnothing$-optimal (it contains the empty partial solution $(\varnothing, \varnothing)$). In dynamic programming we gradually extend this $\varnothing$-optimal DP table to an $[n]$-optimal DP table which should contain an optimal exchange solution. As a basic example, given an I-optimal DP table

[7]For brevity we call it size-L despite its actual length being $(2L + 1)$.

$q[-L\,..\,L]$, for $i \notin I$ we can obtain an $(I \cup \{i\})$-optimal DP table $q'[-L-w_i\,..\,L+w_i]$ in $O(L+w_i)$ time via the update rule $q'[z] \leftarrow \max\{q[z], q[z \mp w_i] \pm p_i\}$ (where $\pm$ is $+$ if $i \in \overline{G}$, or $-$ if $i \in G$).

Previous dynamic programming algorithms for 0-1 Knapsack [5, 25, 44, 54] used the following standard lemma based on the SMAWK algorithm [3]:

Lemma 2.7 (Batch-updating items of the same weight)**.** *Let $I \subseteq [n]$ and $J \subseteq \overline{G}$ be disjoint item subsets, and all $j \in J$ have the same weight $w_j = w$. Suppose an upper bound L' is known such that all $(I \cup J)$-optimal partial solutions (A', B') satisfy $|W(A') - W(B')| \le L'$.*

Then, given an I-optimal DP table $q[-L\,..\,L]$, we can compute an $(I \cup J)$-optimal DP table $q'[-L'\,..\,L']$ in $O(L+L'+|J|+w_{\max}\log w_{\max})$ time.

The same statement holds if the assumption $J \subseteq \overline{G}$ is replaced by $J \subseteq G$.

We defer a proof sketch to the full version.

3 ALGORITHM FOR 0-1 KNAPSACK

In this section we present our algorithm for 0-1 Knapsack (Theorem 1.1). In Section 3.1, we recall a crucial weight partitioning lemma from [25] based on fine-grained proximity, which naturally gives rise to a two-stage algorithm framework. The second stage can be easily performed using previous techniques [25, 54] and is described in Section 3.1, while the first stage contains our main technical challenge and is described in Sections 3.2, 3.3 and 4: In Section 3.2, we give a rank partitioning lemma based on another proximity result. Given this lemma, in Section 3.3 we abstract out a core subproblem called HintedKnapsackExtend$^+$, and describe how to implement the first stage of our algorithm assuming this core subproblem can be solved efficiently. Our algorithm for HintedKnapsackExtend$^+$ will be described in Section 4.

3.1 Weight Partitioning and the Second-Stage Algorithm

Chen, Lian, Mao, and Zhang [25] recently used additive-combinatorial results of Bringmann and Wellnitz [16] to obtain several powerful structural lemmas involving the support size of two integer multisets A, B avoiding non-zero common subset sums. These structural results (called "fine-grained proximity" in [25]) allowed them to obtain faster knapsack algorithms than the earlier works [33, 54] based on ℓ_1-proximity (Eq. (7)) only. Here we recall one of the key lemmas from [25].[8]

Lemma 3.1 ([25, Lemma 3.1], paraphrased)**.** *There is a constant C such that the following holds. Suppose two multisets A, B supported on $[N]$ satisfy*

$$|\operatorname{supp}(A)| \ge C\sqrt{N \log N}$$

and

$$\Sigma(B) \ge \frac{CN^2\sqrt{\log N}}{|\operatorname{supp}(A)|}.$$

Then, $\mathcal{S}^*(A) \cap \mathcal{S}^*(B) \neq \emptyset$.

[8]The original statement of [25, Lemma 3.1] had a worse $\log N$ factor than the $\sqrt{\log N}$ factor in Lemma 3.1. By inspection of their proof, they actually proved the stronger version stated here in Lemma 3.1.

Using this fine-grained proximity result, Chen, Lian, Mao, and Zhang obtained a weight partitioning lemma [25, Lemma 4.1], which is a key ingredient in their algorithm. Our algorithm also crucially relies on this weight partitioning lemma in a similar way, but for our purpose we need to extend it from the two-partition version in [25] to $O(\log w_{\max})$-partition.[9]

Recall the following notations from Section 2.1: $W(I) = \sum_{i \in I} w_i$, $\mathcal{W} = \operatorname{supp}(\{w_1, w_2, \dots, w_n\}) \subseteq [w_{\max}]$, and $I_{\mathcal{W}'} := \{i \in [n] : w_i \in \mathcal{W}'\}$.

Lemma 3.2 (Extension of [25, Lemma 4.1])**.** *The set $\mathcal{W}$ of input item weights can be partitioned in $O(n + w_{\max}\log w_{\max})$ time into $\mathcal{W} = \mathcal{W}_1 \uplus \mathcal{W}_2 \uplus \cdots \uplus \mathcal{W}_s$, where $s < \log_2(\sqrt{w_{\max}})$, with the following property:*

Denote $\mathcal{W}_{\le j} = \mathcal{W}_1 \cup \cdots \cup \mathcal{W}_j$ and $\mathcal{W}_{>j} = \mathcal{W} \setminus \mathcal{W}_{\le j}$. For every optimal exchange solution (A, B) and every $1 \le j \le s$,

- *$|\mathcal{W}_j| \le 4C\sqrt{w_{\max}\log w_{\max}} \cdot 2^j$, and*
- *$W(A \cap I_{\mathcal{W}_{>j}}) \le 4Cw_{\max}^{3/2}/2^j$ and $W(B \cap I_{\mathcal{W}_{>j}}) \le 4Cw_{\max}^{3/2}/2^j$,*

where C is the universal constant from Lemma 3.1.

The proof of Lemma 3.2 is similar to that of the original two-partition version [25, Lemma 4.1], and is deferred to full version.

Given this weight partitioning $\mathcal{W} = \mathcal{W}_1 \uplus \mathcal{W}_2 \uplus \cdots \uplus \mathcal{W}_s$, our overall algorithm runs in two stages: in the first stage, we only consider items whose weights belong to $\mathcal{W}_1$, and efficiently compute an $I_{\mathcal{W}_1}$-optimal DP table (see Definition 2.6) by exploiting the small size of $\mathcal{W}_1$. Then, the second stage of the algorithm updates the DP table using the remaining items $I_{\mathcal{W}_2} \uplus \cdots \uplus I_{\mathcal{W}_s}$. The second stage follows the same idea as [25] of using Lemma 3.2 to trade off the size of the DP table and the number of linear-time scans needed to update the DP table. In contrast, the first stage is more technically challenging; we summarize it in the following lemma, and prove it in subsequent sections:

Lemma 3.3 (The first stage)**.** *Let $\mathcal{W}_1 \subseteq [w_{\max}]$ from Lemma 3.2 be given. Then we can compute an $I_{\mathcal{W}_1}$-optimal DP table in $O(n + w_{\max}^2 \log^4 w_{\max})$ time.*

The overall $O(n + w_{\max}^2 \log^4 w_{\max})$ algorithm for 0-1 Knapsack then follows from Lemma 3.3 and Lemma 3.2, using arguments similar to [25]. In the full version, we include the proof of Theorem 1.1 assuming Lemma 3.3.

3.2 Rank Partitioning

Given $\mathcal{W}_1 \subseteq [w_{\max}]$ of size $|\mathcal{W}_1| \le O(\sqrt{w_{\max}\log w_{\max}})$ from Lemma 3.2, we partition the items whose weights belong to $\mathcal{W}_1$ into dyadic groups based on their ranks (Definition 2.3), as follows:

Definition 3.4 (Rank partitioning)**.** Let $k = \lceil \log_2(2w_{\max}+1) \rceil$. For each $1 \le j \le k$, define item subsets

$$J_j^+ := \{i \in \overline{G} \cap I_{\mathcal{W}_1} : 2^{j-1} \le \operatorname{rank}(i) \le 2^j - 1\},$$

and

$$J_j^- := \{i \in G \cap I_{\mathcal{W}_1} : 2^{j-1} \le \operatorname{rank}(i) \le 2^j - 1\}.$$

Note that $J_1^+ \uplus J_1^- \uplus \cdots \uplus J_k^+ \uplus J_k^-$ form a partition of $\{i \in I_{\mathcal{W}_1} : \operatorname{rank}(i) \le 2^k - 1\}$.

[9]We remark that [25, Lemma 5.3] also gave a three-partition extension of this lemma, but in a different way than what we need here.

Denote

$$J^+_{\le j} = J^+_1 \cup \cdots \cup J^+_j,$$

and

$$J^-_{\le j} = J^-_1 \cup \cdots \cup J^-_j.$$

Note the the rank partitioning defined in Definition 3.4 can be computed in $O(n + w_{\max} \log w_{\max})$ time.

Our rank partitioning is motivated by the following additive-combinatorial Lemma 3.5, which can be derived from the results of Bringmann and Wellnitz [16]. Recall the *r-support* $\operatorname{supp}_r(X)$ of a multiset X is the set of items in X with multiplicity at least r.

Lemma 3.5. *There is a constant C such that the following holds. Suppose two multisets A, B supported on $[N]$ satisfy*

$$|\operatorname{supp}_r(A)| \ge C\sqrt{N/r} \cdot \sqrt{\log(2N)} \tag{9}$$

for some $r \ge 1$, and

$$\Sigma(B) \ge \Sigma(A) - N. \tag{10}$$

Then, $\mathcal{S}^*(A) \cap \mathcal{S}^*(B) \ne \emptyset$.

Lemma 3.5 is partly inspired by [25, Lemma 3.2] which generalized their fine-grained proximity result (Lemma 3.1) from $\operatorname{supp}(A)$ to $\operatorname{supp}_r(A)$.[10] We include a proof of Lemma 3.5 in the full version.

Using Lemma 3.5, we obtain the following structural lemma for the rank partitioning. Recall the definition of $I_{\mathcal{W}_1}$-optimal partial solutions from Definition 2.5.

Lemma 3.6 (Rank partitioning structural lemma). *For a universal constant C, the partition $J^+_1 \uplus J^-_1 \uplus \cdots \uplus J^+_k \uplus J^-_k \subseteq I_{\mathcal{W}_1}$ from Definition 3.4 satisfies the following properties for every $I_{\mathcal{W}_1}$-optimal partial solution (A', B'):*

(1) *$A' \subseteq J^+_{\le k}$ and $B' \subseteq J^-_{\le k}$.*
(2) *For all $1 \le j \le k$, $|A' \cap J^+_{\le j}| \le m_j$ and $|B' \cap J^-_{\le j}| \le m_j$, where*

$$m_j := C \cdot 2^{j/2} \cdot \sqrt{w_{\max} \log(2w_{\max})}. \tag{11}$$

(3) *For all $1 \le j \le k$,*

$$|\{w \in \mathcal{W}_1 : I_{\{w\}} \cap J^+_{j-1} \subseteq A' \text{ and } I_{\{w\}} \cap J^+_j \ne \emptyset\}| \le b_j,$$

and similarly

$$|\{w \in \mathcal{W}_1 : I_{\{w\}} \cap J^-_{j-1} \subseteq B' \text{ and } I_{\{w\}} \cap J^-_j \ne \emptyset\}| \le b_j,$$

where

$$b_j := C \cdot 2^{-j/2} \cdot \sqrt{w_{\max} \log(2w_{\max})}, \tag{12}$$

and $J^+_0 := J^-_0 := \emptyset$.

We defer the proof of this lemma to the full version.

[10]Their generalization of Lemma 3.1 is not applicable in our first-stage algorithm. Note that Lemma 3.5 is incomparable to Lemma 3.1 even when $r = 1$.

3.3 The First-Stage Algorithm via Hinted Dynamic Programming

Based on our rank partitioning $J^+_1 \uplus J^-_1 \uplus \cdots \uplus J^+_k \uplus J^-_k \subseteq I_{\mathcal{W}_1}$, $k = \lceil \log_2(2w_{\max} + 1) \rceil$ (Definition 3.4) and its structural lemma (Lemma 3.6), our first-stage algorithm uses dynamic programming and runs in k phases. At the beginning of the j-th phase ($1 \le j \le k$), we have a $(J^+_{\le j-1} \cup J^-_{\le j-1})$-optimal DP table, and we first update it with the "positive items" J^+_j to obtain a $(J^+_{\le j} \cup J^-_{\le j-1})$-optimal DP table, and then update it with the "negative items" J^-_j to obtain a $(J^+_{\le j} \cup J^-_{\le j})$-optimal DP table. We will adjust the size of the DP table throughout the k phases based on Item 2 of Lemma 3.6. This is similar to the second-stage algorithm from Section 3.1, except that in Section 3.1 the DP table is shrinking whereas here it will be expanding.

To implement the DP efficiently, we crucially rely on Item 3 of Lemma 3.6, which gives an upper bound on the "active support" of the weights of items in every partial solution in the current DP table. More specifically, consider an $I_{\mathcal{W}_1}$-optimal partial solution (A', B') and its restriction (A'', B'') where $A'' = A' \cap J^+_{\le j-1}, B'' = B' \cap J^-_{\le j-1}$. Then Item 3 of Lemma 3.6 implies that the items in $A' \setminus A''$ (or $B' \setminus B''$) can only have at most b_j distinct weights. This means that, for any partial solution (A'', B'') in the DP table at the end of phase $j - 1$, in order to extend it to an $I_{\mathcal{W}_1}$-optimal partial solution (A', B') in future phases, we only need to update it with items from these b_j weight classes determined by Item 3 of Lemma 3.6. This idea is called *witness propagation*, and was originally introduced by Deng, Mao, and Zhong [31] in the context of unbounded knapsack-type problems. Implementing this idea in the more difficult 0-1 setting is a main technical contribution of this paper.

In the rest of this section, we will introduce a few more definitions to help use formally describe our algorithm, and we will abstract out a core subproblem called HintedKnapsackExtend$^+$ which captures the aforementioned idea of witness propagation. Then we will show how to implement our first-stage algorithm and prove Lemma 3.3, assuming HintedKnapsackExtend$^+$ can be solved efficiently.

In the following definition, we augment each entry of the DP table with hints, which contain the weight classes from which we need to add items when we update this entry, as we just discussed.

Definition 3.7 (Hinted DP tables). A *hinted DP table* is a DP table $q[\,]$ where each entry $q[z] \ne -\infty$ is annotated with two sets $S^+[z], S^-[z] \subseteq \mathcal{W}_1$. We say the table has *positive hint size b* if $|S^+[z]| \le b$ for all z, and has *negative hint size b* if $|S^-[z]| \le b$ for all z.

For an item subset $J \subseteq I_{\mathcal{W}_1}$, we say a hinted DP table $q[\,]$ is *hinted-J-optimal*, if $q[\,]$ is J-valid (see Definition 2.6), and it has an entry $q[z]$ such that both of the following hold:

(1) There exists an $I_{\mathcal{W}_1}$-optimal partial solution (A', B') such that $W(A' \cap J) - W(B' \cap J) = z$ and $P(A' \cap J) - P(B' \cap J) = q[z]$.
(2) Every $I_{\mathcal{W}_1}$-optimal partial solution (A', B') with $W(A' \cap J) - W(B' \cap J) = z$ should satisfy $A' \setminus J \subseteq I_{S^+[z]}$ and $B' \setminus J \subseteq I_{S^-[z]}$.

Note that if a hinted DP table is hinted-J-optimal, then in particular it is J-optimal in the sense of Definition 2.6 (due to Item 1 of Definition 3.7).

The following lemma summarizes each of the $k = \lceil \log_2(2w_{\max} + 1) \rceil$ phases in our first-stage algorithm.

Lemma 3.8. *Let k, m_j, b_j be defined as in Lemma 3.6. Let $L_j := m_j \cdot w_{\max}$. For every $1 \le j \le k$, the following hold:*

(1) *Given a hinted-$(J^+_{\le j-1} \cup J^-_{\le j-1})$-optimal DP table of size L_{j-1} with positive and negative hint size b_j, we can compute a hinted-$(J^+_{\le j} \cup J^-_{\le j-1})$-optimal DP table of size L_j with positive hint size b_{j+1} and negative hint size b_j, in $O(L_j b_j \cdot \log^2(L_j b_j) + |J^+_j|)$ time.*

(2) *Given a hinted-$(J^+_{\le j} \cup J^-_{\le j-1})$-optimal DP table of size L_j with positive hint size b_{j+1} and negative hint size b_j, we can compute a hinted-$(J^+_{\le j} \cup J^-_{\le j})$-optimal DP table of size L_j with positive and negative hint size b_{j+1}, in $O(L_j b_j \cdot \log^2(L_j b_j) + |J^-_j|)$ time.*

Lemma 3.8 immediately implies our overall first-stage algorithm. We defer the proof of Lemma 3.3 assuming Lemma 3.8 to the full version.

It remains to prove Lemma 3.8. In the following, we will reduce it to a core subproblem called HintedKnapsackExtend$^+$, which captures the task of updating a hinted size-L DP table with positive hint size b using "positive items" whose weights come from some positive integer set U (here we can think of $U = \mathcal{W}_1$). Similarly to the proof of the batch-update lemma (Lemma 2.7) based on SMAWK, here we also use a function $Q_w : \mathbb{Z}_{\ge 0} \to \mathbb{Z}$ to represent the total profit of taking the top-x items of weight w.

Problem 1 (HintedKnapsackExtend$^+$). *Let $U \subseteq \mathcal{W}_1$. For every $w \in U$, suppose $Q_w : \mathbb{Z}_{\ge 0} \to \mathbb{Z}$ is a concave function with $Q_w(0) = 0$ that can be evaluated in constant time. We are given a DP table $q[-L \,..\, L]$ (where $q[i] \in \mathbb{Z} \cup \{-\infty\}$), annotated with $S[-L \,..\, L]$ where $S[i] \subseteq U$.*

Consider the following optimization problem for each $-L \le i \le L$: find a solution vector $\boldsymbol{x}[i] \in \mathbb{Z}^U_{\ge 0}$ that maximizes the total profit

$$r[i] := q\big[z[i]\big] + \sum_{w \in U} Q_w(x[i]_w), \tag{13}$$

where $z[i] \in \mathbb{Z}$ is uniquely determined by

$$z[i] + \sum_{w \in U} w \cdot x[i]_w = i. \tag{14}$$

The task is to solve this optimization problem for each $-L \le i \le L$ with the following relaxation*:*

- *If* all *maximizers $(\boldsymbol{x}[i], z[i])$ of Eq. (13) (subject to Eq. (14)) satisfy*
$$\operatorname{supp}(\boldsymbol{x}[i]) \subseteq S\big[z[i]\big], \tag{15}$$
then we are required to correctly output a maximizer for i.
- *Otherwise, we are allowed to output a suboptimal solution for i.*

Remark 3.9. We give a few remarks to help get a better understanding of Problem 1:

(1) In Eq. (14), $z[i] \le i$ must hold, since $w \in U \subseteq [w_{\max}]$ is always positive and $\boldsymbol{x}[i]$ is a non-negative vector.
(2) If we do not have the relaxation based on hints $S[i]$, then Problem 1 becomes a standard problem solvable in $O(|U|L)$ time using SMAWK algorithm (basically, repeat the proof of Lemma 2.7 for every $w \in U$; see also [5, 25, 44, 54]).
(3) Under this relaxation, without loss of generality, we can assume the output of Problem 1 always satisfies $\operatorname{supp}(\boldsymbol{x}[i]) \subseteq S\big[z[i]\big]$ (Eq. (15)) for all $-L \le i \le L$. (If we had to output an $\boldsymbol{x}[i]$ that violates Eq. (15), then we must be in the "otherwise" case for i, and should be allowed to output anything). In particular, if $|S[i]| \le b$ for all $-L \le i \le L$, then we can assume the output $\boldsymbol{x}[-L \,..\, L]$ of Problem 1 has description size $O(bL)$ words.
(4) Note that Problem 1 is different from (and easier than) the task of maximizing Eq. (13) for every i subject to Eq. (15). The latter version would make a cleaner definition, but it is a harder problem which we do not know how to solve.

The following Theorem 3.10 summarizes our algorithm for Problem 1, which will be given in Section 4.

Theorem 3.10. *HintedKnapsackExtend$^+$ (Problem 1) with $|S[i]| \le b$ for all $-L \le i \le L$ can be solved deterministically in $O(Lb \log^2(Lb))$ time.*

In the full version of the paper, we show how to prove Lemma 3.8 using Theorem 3.10.

4 ALGORITHM FOR HintedKnapsackExtend$^+$

In this section we solve the HintedKnapsackExtend$^+$ problem (Problem 1), proving Theorem 3.10. In Lemma 4.1, we solve the special case where the hints are singleton sets. In the full version of the paper, we provide several helper lemmas that allow us to decompose an instance into multiple instances with smaller hint sets, and then put the pieces together to solve the general case.

4.1 The Base Case with Singleton Hint Sets

The following lemma is the most interesting building block of our algorithm for Problem 1.

Lemma 4.1. *HintedKnapsackExtend$^+$ (Problem 1) with $|S[i]| \le 1$ for all $-L \le i \le L$ can be solved deterministically in $O(L \log L)$ time.*

More precisely, the algorithm runs in $O(L + L_1 \log L)$ time, where $L_1 = \{-L \le i \le L : S[i] \ne \emptyset\}$.

The pseudocode of our algorithm for Lemma 4.1 is given in Algorithm 1. Here we first provide an overview. Algorithm 1 contains two stages:

- In the first stage, we enumerate $w \in [w_{\max}]$ and $c \pmod w$, and collect indices $j \equiv c \pmod w$ such that $w \in S[j]$. Then we try to extend from these collected indices j by adding integer multiples of w (which does not interfere with other congruence classes modulo w): using SMAWK algorithm [3], for every $i \equiv c \pmod w$, find j among the collected indices to maximize $q[j] + Q_w(\frac{i-j}{w})$. This is the same idea as in the proof of the standard batch-update Lemma 2.7 (used in e.g., [5, 20, 44, 54]). However, in our scenario with small sets $S[j]$, the number of collected indices j is usually sublinear in the array size L, so in order to save time we need to let SMAWK

Algorithm 1: Solving HintedKnapsackExtend$^+$ with singleton hint sets

```
   Input: $q[-L\,..\,L]$ and $S[-L\,..\,L]$, where
          $S[i] \subseteq [w_{\max}], |S[i]| \le 1, q[i] \in \mathbb{Z} \cup \{-\infty\}$ for all $i$
   Output: $(\boldsymbol{x}[-L\,..\,L], z[-L\,..\,L], r[-L\,..\,L])$ as a solution to
           Problem 1
1  SMAWKAndScan($q[-L\,..\,L], S[-L\,..\,L]$):
2  begin
       /* Stage 1: use SMAWK to find all candidate
          updates $q[j] + Q_w(\frac{i-j}{w})$ where $w \in S[j]$,
          expressed as difference-$w$ APs consisting
          of indices $i$                                  */
3      Initialize $\mathcal{P} \leftarrow \varnothing$
4      for $w \in [w_{\max}]$ and $c \in \{0, 1, \dots, w-1\}$ do
5          $J := \{j : w \in S[j] \text{ and } j \equiv c \pmod w, -L \le j \le L\}$
6          $I := \{i : i \equiv c \pmod w, -L \le i \le L\}$
7          Run SMAWK on matrix $A_{I \times J}$ defined as
             $A[i,j] := q[j] + Q_w\left(\frac{i-j}{w}\right)$.
8          for $j \in J$ do
9              Suppose SMAWK returned the AP $P_j \subseteq I$ of
                 difference $w$, such that for every $i \in P_j$,
                 $j = \arg\max_{j' \in J} A[i, j']$
10             $P_j \leftarrow P_j \cap \{i \in \mathbb{Z} : i > j\}$        // focus on
                 candidate updates where $\frac{i-j}{w}$ is a
                 positive integer
11             Suppose $P_j = \{c + kw, c + (k+1)w, \dots, c + \ell w\}$,
                 and insert $(j; c, w, k, \ell)$ into $\mathcal{P}$
       /* Stage 2: combine all candidate updates by a
          linear scan from left to right, extending
          winning APs and discarding losing APs        */
12     Initialize empty buckets $B[-L], B[-L+1], \dots, B[L]$
13     for $(j; c, w, k, \ell) \in \mathcal{P}$ do
14         Insert $(j; c, w, k, \ell)$ into bucket $B[c + kw]$
             // insert to the bucket indexed by the
             beginning element of the AP
15     for $i \leftarrow -L, \dots, L$ do
16         $r[i] \leftarrow q[i], z[i] \leftarrow i, \boldsymbol{x}[i] \leftarrow \mathbf{0}$.   // the trivial
             solution for $i$
17         if $B[i] \ne \varnothing$ then
18             Pick $(j; c, w, k, \ell) \in B[i]$ that maximizes
                 $q[j] + Q_w\left(\frac{i-j}{w}\right)$
19             if $q[j] + Q_w\left(\frac{i-j}{w}\right) > r[i]$ then
20                 $r[i] \leftarrow q[j] + Q_w\left(\frac{i-j}{w}\right), z[i] \leftarrow j, \boldsymbol{x}[i] \leftarrow$
                     $\frac{i-j}{w}\boldsymbol{e}_w$.               // solution for $i$
21             if $i + w \le c + \ell w$ then
22                 Insert $(j; c, w, k, \ell)$ into bucket $B[i + w]$
                     // extend this winning AP by one
                     step, and all other APs in the
                     bucket $B[i]$ are discarded
23     return $(\boldsymbol{x}[-L\,..\,L], z[-L\,..\,L], r[-L\,..\,L])$
```

return a compact output representation, described as several arithmetic progressions (APs) with difference w, where each AP contains the indices i that have the same maximizer j.

- The second stage is to combine all the APs found in the first stage, and update them onto a single DP array. Ideally, we would like to take the entry-wise maximum over all the APs, that is, for each i we would like to maximize $q[j] + Q_w(\frac{i-j}{w})$ over all APs containing i, where w is the difference of the AP and j is the maximizer associated to that AP. Unfortunately, the total length of these APs could be much larger than the array size L, which would prevent us from getting an $\widetilde{O}(L)$ time algorithm. To overcome this challenge, the idea here is to crucially use the relaxation in the definition of Problem 1, so that we can skip a lot of computation based on the concavity of $Q_w(\cdot)$. We perform a linear scan from left to right, and along the way we discard many APs that cannot contribute to any useful answers. In this way we can get the time complexity down to near-linear.

Proof of Lemma 4.1. The algorithm is given in Algorithm 1.

Time complexity. We first analyze the time complexities of the two stages of Algorithm 1.

- The first stage contains a **for** loop over $w \in [w_{\max}]$ and $c \in \{0, 1 \dots, w-1\}$ (Line 4), but we actually only need to execute the loop iterations such that the index set $J := \{j : w \in S[j] \text{ and } j \equiv c \pmod w, -L \le j \le L\}$ (defined at Line 5) is non-empty. Since $|S[j]| \le 1$ for all j, these sets J over all (w, c) form a partition of the size-L_1 set $\{-L \le j \le L : S[j] \ne \varnothing\}$, and can be prepared efficiently. Then, for each of these sets J, at Line 7 we run SMAWK algorithm [3] to find all row maxima (with compact output representation) of an $O(1 + L/w) \times |J|$ matrix in $O(|J| \log L)$ time. The output of SMAWK is represented as $|J|$ intervals on the row indices of this matrix, which correspond to $|J|$ APs of difference w. These $|J|$ APs are then added into $\mathcal{P}$. Thus, in the end of the first stage, set $\mathcal{P}$ contains at most $\sum_J |J| \le L_1 \le 2L + 1$ APs (each AP only takes $O(1)$ words to describe), and the total running time of this stage is $O(\sum_J |J| \log L) = O(L_1 \log L)$.
- In the second stage, we initialize $(2L + 1)$ buckets $B[-L\,..\,L]$, and first insert each AP from $\mathcal{P}$ into a bucket (Line 13). Then we do a scan $i \leftarrow -L, \dots, L$ (Line 15), where for each i we examine all APs in the bucket $B[i]$ at Line 18, and then copy at most one winning AP from this bucket into another bucket (Line 22). Hence, in total we only ever inserted at most $|\mathcal{P}| + (2L + 1) = O(L)$ APs into the buckets. So the second stage takes $O(L)$ overall time.

Hence the total time complexity of Algorithm 1 is $O(L_1 \log L + L) \le O(L \log L)$.

Correctness. We prove that the return values $(\boldsymbol{x}[i], z[i], r[i])$ correctly solve Problem 1.

Fix any $i \in \{-L, \dots, L\}$, and let $(\boldsymbol{x}^*[i], z^*[i], r^*[i])$ be an maximizer of Eq. (13) (subject to Eq. (14)). If $\boldsymbol{x}^*[i] = \mathbf{0}$, then it is the trivial solution, which cannot be better than our solution, due to Lines 16 and 19. So in the following we assume $|\text{supp}(\boldsymbol{x}^*[i])| \ge 1$, which means $i > z^*[i]$. If the support containment condition

$\mathrm{supp}(\boldsymbol{x}^*[i]) \subseteq S\big[z^*[i]\big]$ (Eq. (15)) is violated, then by definition of Problem 1 we are not required to find a maximizer for i. Hence, we can assume $\mathrm{supp}(\boldsymbol{x}^*[i]) \subseteq S\big[z^*[i]\big]$ holds. Since $\big|S\big[z^*[i]\big]\big| \le 1 \le |\,\mathrm{supp}(\boldsymbol{x}^*[i])|$, we assume $\mathrm{supp}(\boldsymbol{x}^*[i]) = S\big[z^*[i]\big] = \{w^*\}$.

In the **for** loop iteration of the first stage where $w = w^*$ and $c = i \bmod w$, we have $z^*[i] \in J$ and $i \in I$. The input matrix $A_{I\times J}$ to SMAWK encodes the objective values of extending from j by adding multiples of w^*; in particular, $A[i, z^*[i]]$ equals our optimal objective $r^*[i] = q\big[z^*[i]\big] + Q_{w^*}\left(\frac{i-z^*[i]}{w^*}\right)$. So SMAWK correctly returns an AP $P_{z^*[i]} = \{c+kw^*, c+(k+1)w^*, \dots, c+\ell w^*\}$ that contains i (unless there is a tie $A[i, z^*[i]] = A[i, j]$ for some other $j \in J$, and i ends up in the AP P_j, but in this case we could have started the proof with $(\boldsymbol{x}^*[i], z^*[i])$ being this alternative maximizer $z^*[i] \leftarrow j$ and $\boldsymbol{x}^*[i] \leftarrow \frac{i-z^*[i]}{w^*}\boldsymbol{e}_{w^*}$). Since $i > z^*[i]$, we know i is not removed from $P_{z^*[i]}$ at Line 10. This AP $P_{z^*[i]}$ containing i is then added to $\mathcal{P}$.

In the second stage, each AP in $\mathcal{P}$ starts in the bucket indexed by the leftmost element of this AP (Line 13), and during the left-to-right linear scan this AP may win over others in its current bucket (at Line 18) and gets advanced to the bucket corresponding to its next element in the AP (at Line 22), or it may lose at Line 18 and be discarded. (Note that any AP can only appear in buckets whose indices belong to this AP.) Our goal is to show that the AP $P_{z^*[i]}$ can survive the competitions and arrive in bucket $B[i]$, so that it can successfully update the answer for i at Line 20.

Suppose for contradiction that $P_{z^*[i]}$ lost to some other AP $P'_{j'}$ at Line 18 when they were both in bucket $B[i_0]$ (for some $i_0 < i$). Suppose this AP $P'_{j'}$ has common difference w', and corresponds to the objective value $q[j'] + Q_{w'}\left(\frac{i'-j'}{w'}\right)$ for $i' \in P'_{j'}$. Note that $i_0 \in P'_{j'}$ satisfies $i_0 > j'$ due to Line 10. Note that $w' \ne w^*$ must hold, since two APs produced in stage 1 with the same common difference cannot intersect (because SMAWK returns disjoint intervals), and hence cannot appear in the same bucket $B[i_0]$. Now we consider an alternative solution for index i defined as

$$(\boldsymbol{x}', j') := \left(\tfrac{i_0-j'}{w'}\boldsymbol{e}_{w'} + \tfrac{i-i_0}{w^*}\boldsymbol{e}_{w^*}\,,\ j'\right).$$

Note that $j' + \sum_{w\in[w_{\max}]} w \cdot x'_w = i$, and it has objective value

$$\begin{aligned}
&q[j'] + \sum_{w\in[w_{\max}]} Q_w(x'_w)\\
&= q[j'] + Q_{w'}\left(\tfrac{i_0-j'}{w'}\right) + Q_{w^*}\left(\tfrac{i-i_0}{w^*}\right)\\
&\ge q\left[z^*[i]\right] + Q_{w^*}\left(\tfrac{i_0-z^*[i]}{w^*}\right) + Q_{w^*}\left(\tfrac{i-i_0}{w^*}\right)\\
&\qquad\qquad (\text{since } P_{j'} \text{ wins over } P_{z^*[i]} \text{ in bucket } B[i_0])\\
&\ge q\left[z^*[i]\right] + Q_{w^*}\left(\tfrac{i_0-z^*[i]}{w^*} + \tfrac{i-i_0}{w^*}\right) + 0 \quad (\text{by concavity of } Q_{w^*}(\cdot))\\
&= r^*[i].
\end{aligned}$$

Now there are two cases:

- $q[j'] + \sum_{w\in[w_{\max}]} Q_w(x'_w) > r^*[i]$.
  This contradicts the assumption that $r^*[i]$ is the optimal objective value for index i.
- $q[j'] + \sum_{w\in[w_{\max}]} Q_w(x'_w) = r^*[i]$.
  Then, $(\boldsymbol{x}', j')$ is also a maximizer for index i, but it has support size $|\,\mathrm{supp}(\boldsymbol{x}')| = 2$ due to $i > i_0 > j'$ and $w' \ne w^*$, and hence violates the support containment condition $\mathrm{supp}(\boldsymbol{x}') \subseteq S[j']$ (Eq. (15)). By definition of Problem 1, we are not required to find a maximizer for index i.

Hence, we have shown that the AP $P_{z^*[i]}$ can arrive in bucket $B[i]$. This finishes the proof that Algorithm 1 correctly solves HintedKnapsackExtend$^+$ for index i. □

ACKNOWLEDGEMENTS

I thank Ryan Williams and Virginia Vassilevska Williams for useful discussions. I thank anonymous reviewers for useful comments on previous versions of this paper.

REFERENCES

[1] Amir Abboud, Karl Bringmann, Danny Hermelin, and Dvir Shabtay. 2022. Scheduling lower bounds via AND subset sum. *J. Comput. Syst. Sci.* 127 (2022), 29–40. https://doi.org/10.1016/j.jcss.2022.01.005
[2] Amir Abboud, Karl Bringmann, Danny Hermelin, and Dvir Shabtay. 2022. SETH-based Lower Bounds for Subset Sum and Bicriteria Path. *ACM Trans. Algorithms* 18, 1 (2022), 6:1–6:22. https://doi.org/10.1145/3450524
[3] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter W. Shor, and Robert E. Wilber. 1987. Geometric Applications of a Matrix-Searching Algorithm. *Algorithmica* 2 (1987), 195–208. https://doi.org/10.1007/BF01840359
[4] Kyriakos Axiotis, Arturs Backurs, Ce Jin, Christos Tzamos, and Hongxun Wu. 2019. Fast Modular Subset Sum using Linear Sketching. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*. SIAM, 58–69. https://doi.org/10.1137/1.9781611975482.4
[5] Kyriakos Axiotis and Christos Tzamos. 2019. Capacitated Dynamic Programming: Faster Knapsack and Graph Algorithms. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019 (LIPIcs, Vol. 132)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:13. https://doi.org/10.4230/LIPIcs.ICALP.2019.19
[6] MohammadHossein Bateni, MohammadTaghi Hajiaghayi, Saeed Seddighin, and Cliff Stein. 2018. Fast algorithms for knapsack via convolution and prediction. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018*. ACM, 1269–1282. https://doi.org/10.1145/3188745.3188876
[7] Richard Bellman. 1957. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA.
[8] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. 1973. Time Bounds for Selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461. https://doi.org/10.1016/S0022-0000(73)80033-9
[9] David Bremner, Timothy M. Chan, Erik D. Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Mihai Pătraşcu, and Perouz Taslakian. 2014. Necklaces, Convolutions, and X+Y. *Algorithmica* 69, 2 (2014), 294–314. https://doi.org/10.1007/s00453-012-9734-3
[10] Karl Bringmann. 2017. A Near-Linear Pseudopolynomial Time Algorithm for Subset Sum. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*. SIAM, 1073–1084. https://doi.org/10.1137/1.9781611974782.69
[11] Karl Bringmann. 2023. Knapsack with Small Items in Near-Quadratic Time. *arXiv preprint arXiv:2308.03075* (2023). arXiv:2308.03075 To appear in STOC 2024.
[12] Karl Bringmann and Alejandro Cassis. 2022. Faster Knapsack Algorithms via Bounded Monotone Min-Plus-Convolution. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022 (LIPIcs, Vol. 229)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 31:1–31:21. https://doi.org/10.4230/LIPIcs.ICALP.2022.31
[13] Karl Bringmann and Alejandro Cassis. 2023. Faster 0-1-Knapsack via Near-Convex Min-Plus-Convolution. In *31st Annual European Symposium on Algorithms, ESA 2023 (LIPIcs, Vol. 274)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:16. https://doi.org/10.4230/LIPICS.ESA.2023.24
[14] Karl Bringmann, Nick Fischer, Danny Hermelin, Dvir Shabtay, and Philip Wellnitz. 2022. Faster Minimization of Tardy Processing Time on a Single Machine. *Algorithmica* 84, 5 (2022), 1341–1356. https://doi.org/10.1007/s00453-022-00928-w
[15] Karl Bringmann and Vasileios Nakos. 2021. A Fine-Grained Perspective on Approximating Subset Sum and Partition. In *Proc. 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1797–1815. https://doi.org/10.1137/1.9781611976465.108
[16] Karl Bringmann and Philip Wellnitz. 2021. On Near-Linear-Time Algorithms for Dense Subset Sum. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*. SIAM, 1777–1796. https://doi.org/10.1137/1.9781611976465.107 arXiv:2010.09096
[17] Mark Chaimovich. 1999. New algorithm for dense subset-sum problem. Number 258. xvi, 363–373. Structure theory of set addition.

[18] Mark Chaimovich. 1999. New structural approach to integer programming: a survey. Number 258. xv–xvi, 341–362. Structure theory of set addition.

[19] Mark Chaimovich, Gregory Freiman, and Zvi Galil. 1989. Solving dense subset-sum problems by using analytical number theory. *J. Complexity* 5, 3 (1989), 271–282. https://doi.org/10.1016/0885-064X(89)90025-3

[20] Timothy M. Chan. 2018. Approximation Schemes for 0-1 Knapsack. In *1st Symposium on Simplicity in Algorithms, SOSA 2018 (OASIcs, Vol. 61)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:12. https://doi.org/10.4230/OASIcs.SOSA.2018.5

[21] Timothy M. Chan and Qizheng He. 2022. More on change-making and related problems. *J. Comput. Syst. Sci.* 124 (2022), 159–169. https://doi.org/10.1016/j.jcss.2021.09.005

[22] Timothy M. Chan and Moshe Lewenstein. 2015. Clustered Integer 3SUM via Additive Combinatorics. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015*. ACM, 31–40. https://doi.org/10.1145/2746539.2746568

[23] Timothy M. Chan and R. Ryan Williams. 2021. Deterministic APSP, Orthogonal Vectors, and More: Quickly Derandomizing Razborov-Smolensky. *ACM Trans. Algorithms* 17, 1 (2021), 2:1–2:14. https://doi.org/10.1145/3402926

[24] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. 2023. A Nearly Quadratic-Time FPTAS for Knapsack. *CoRR* abs/2308.07821 (2023). arXiv:2308.07821 To appear in STOC 2024.

[25] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. 2024. Faster Algorithms for Bounded Knapsack and Bounded Subset Sum Via Fine-Grained Proximity Results. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 4828–4848. https://doi.org/10.1137/1.9781611977912.171 arXiv:2307.12582

[26] Shucheng Chi, Ran Duan, Tianle Xie, and Tianyi Zhang. 2022. Faster min-plus product for monotone instances. In *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 1529–1542. https://doi.org/10.1145/3519935.3520057

[27] David Conlon, Jacob Fox, and Huy Tuan Pham. 2021. Subset sums, completeness and colorings. *arXiv preprint arXiv:2104.14766* (2021). arXiv:2104.14766

[28] William J. Cook, A. M. H. Gerards, Alexander Schrijver, and Éva Tardos. 1986. Sensitivity theorems in integer linear programming. *Math. Program.* 34, 3 (1986), 251–264. https://doi.org/10.1007/BF01582230

[29] Marek Cygan, Marcin Mucha, Karol Węgrzycki, and Michał Włodarczyk. 2019. On Problems Equivalent to (min, +)-Convolution. *ACM Trans. Algorithms* 15, 1 (2019), 14:1–14:25. https://doi.org/10.1145/3293465

[30] Mingyang Deng, Ce Jin, and Xiao Mao. 2023. Approximating Knapsack and Partition via Dense Subset Sums. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*. SIAM, 2961–2979. https://doi.org/10.1137/1.9781611977554.ch113 arXiv:2301.09333

[31] Mingyang Deng, Xiao Mao, and Ziqian Zhong. 2023. On Problems Related to Unbounded SubsetSum: A Unified Combinatorial Approach. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*. SIAM, 2980–2990. https://doi.org/10.1137/1.9781611977554.ch114

[32] Friedrich Eisenbrand and Gennady Shmonin. 2006. Carathéodory bounds for integer cones. *Oper. Res. Lett.* 34, 5 (2006), 564–568. https://doi.org/10.1016/j.orl.2005.09.008

[33] Friedrich Eisenbrand and Robert Weismantel. 2020. Proximity Results and Faster Algorithms for Integer Programming Using the Steinitz Lemma. *ACM Trans. Algorithms* 16, 1 (2020), 5:1–5:14. https://doi.org/10.1145/3340322

[34] Gregory A. Freiman. 1988. On extremal additive problems of Paul Erdős. *Ars Combin.* 26, B (1988), 93–114.

[35] G. A. Freiman. 1990. Subset-sum problem with different summands, In Proceedings of the Twentieth Southeastern Conference on Combinatorics, Graph Theory, and Computing (Boca Raton, FL, 1989). *Congr. Numer.* 70, 207–215.

[36] Gregory A. Freiman. 1993. New analytical results in subset-sum problem. Vol. 114. 205–217. https://doi.org/10.1016/0012-365X(93)90367-3 Combinatorics and algorithms (Jerusalem, 1988).

[37] Zvi Galil and Oded Margalit. 1991. An Almost Linear-Time Algorithm for the Dense Subset-Sum Problem. *SIAM J. Comput.* 20, 6 (1991), 1157–1189. https://doi.org/10.1137/0220072

[38] Qizheng He and Zhean Xu. 2024. Simple and Faster Algorithms for Knapsack. In *2024 Symposium on Simplicity in Algorithms (SOSA)*. 56–62. https://doi.org/10.1137/1.9781611977936.6

[39] Ellis Horowitz and Sartaj Sahni. 1974. Computing Partitions with Applications to the Knapsack Problem. *J. ACM* 21, 2 (1974), 277–292. https://doi.org/10.1145/321812.321823

[40] Klaus Jansen and Lars Rohwedder. 2019. On Integer Programming and Convolution. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019 (LIPIcs, Vol. 124)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 43:1–43:17. https://doi.org/10.4230/LIPIcs.ITCS.2019.43

[41] Klaus Jansen and Lars Rohwedder. 2022. On Integer Programming, Discrepancy, and Convolution. *Mathematics of Operations Research* 0, 0 (2022), 1–15. https://doi.org/10.1287/moor.2022.1308

[42] Ce Jin. 2019. An Improved FPTAS for 0-1 Knapsack. In *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019 (LIPIcs, Vol. 132)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 76:1–76:14. https://doi.org/10.4230/LIPIcs.ICALP.2019.76

[43] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA (The IBM Research Symposia Series)*. Plenum Press, New York, 85–103. https://doi.org/10.1007/978-1-4684-2001-2_9

[44] Hans Kellerer and Ulrich Pferschy. 2004. Improved Dynamic Programming in Connection with an FPTAS for the Knapsack Problem. *J. Comb. Optim.* 8, 1 (2004), 5–11. https://doi.org/10.1023/B:JOCO.0000021934.29833.6b

[45] Hans Kellerer, Ulrich Pferschy, and David Pisinger. 2004. *Knapsack problems*. Springer. https://doi.org/10.1007/978-3-540-24777-7

[46] Kim-Manuel Klein. 2022. On the Fine-Grained Complexity of the Unbounded SubsetSum and the Frobenius Problem. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*. SIAM, 3567–3582. https://doi.org/10.1137/1.9781611977073.141

[47] Kim-Manuel Klein, Adam Polak, and Lars Rohwedder. 2023. On Minimizing Tardy Processing Time, Max-Min Skewed Convolution, and Triangular Structured ILPs. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*. SIAM, 2947–2960. https://doi.org/10.1137/1.9781611977554.ch112

[48] Konstantinos Koiliaris and Chao Xu. 2019. Faster Pseudopolynomial Time Algorithms for Subset Sum. *ACM Trans. Algorithms* 15, 3 (2019), 40:1–40:20. https://doi.org/10.1145/3329863

[49] Marvin Künnemann, Ramamohan Paturi, and Stefan Schneider. 2017. On the Fine-Grained Complexity of One-Dimensional Dynamic Programming. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017 (LIPIcs, Vol. 80)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21:1–21:15. https://doi.org/10.4230/LIPIcs.ICALP.2017.21

[50] Andrea Lincoln, Adam Polak, and Virginia Vassilevska Williams. 2020. Monochromatic Triangles, Intermediate Matrix Products, and Convolutions. In *11th Innovations in Theoretical Computer Science Conference, ITCS 2020 (LIPIcs, Vol. 151)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 53:1–53:18. https://doi.org/10.4230/LIPIcs.ITCS.2020.53

[51] Xiao Mao. 2023. (1-ϵ)-Approximation of Knapsack in Nearly Quadratic Time. *CoRR* abs/2308.07004 (2023). arXiv:2308.07004 To appear in STOC 2024.

[52] Marcin Mucha, Karol Węgrzycki, and Michał Włodarczyk. 2019. A Subquadratic Approximation Scheme for Partition. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*. SIAM, 70–88. https://doi.org/10.1137/1.9781611975482.5

[53] David Pisinger. 1999. Linear Time Algorithms for Knapsack Problems with Bounded Weights. *J. Algorithms* 33, 1 (1999), 1–14. https://doi.org/10.1006/jagm.1999.1034

[54] Adam Polak, Lars Rohwedder, and Karol Węgrzycki. 2021. Knapsack and Subset Sum with Small Items. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021 (LIPIcs, Vol. 198)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 106:1–106:19. https://doi.org/10.4230/LIPIcs.ICALP.2021.106 arXiv:2105.04035

[55] A. Sárközy. 1989. Finite addition theorems. I. *J. Number Theory* 32, 1 (1989), 114–130. https://doi.org/10.1016/0022-314X(89)90102-9

[56] A. Sárközy. 1994. Finite addition theorems. II. *J. Number Theory* 48, 2 (1994), 197–218. https://doi.org/10.1006/jnth.1994.1062

[57] Richard Schroeppel and Adi Shamir. 1981. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM J. Comput.* 10, 3 (1981), 456–464. https://doi.org/10.1137/0210033

[58] E. Szemerédi and V. H. Vu. 2006. Finite and infinite arithmetic progressions in sumsets. *Ann. of Math. (2)* 163, 1 (2006), 1–35. https://doi.org/10.4007/annals.2006.163.1

[59] R. Ryan Williams. 2018. Faster All-Pairs Shortest Paths via Circuit Complexity. *SIAM J. Comput.* 47, 5 (2018), 1965–1985. https://doi.org/10.1137/15M1024524

Received 13-NOV-2023; accepted 2024-02-11