Memory Reallocation with Polylogarithmic Overhead

Ce Jin*
UC Berkeley

Abstract

The *Memory Reallocation problem* asks to dynamically maintain an assignment of given objects of various sizes to non-overlapping contiguous chunks of memory, while supporting updates (insertions/deletions) in an online fashion. The total size of live objects at any time is guaranteed to be at most a $1-\epsilon$ fraction of the total memory. To handle an online update, the allocator may rearrange the objects in memory to make space, and the *overhead* for this update is defined as the total size of moved objects divided by the size of the object being inserted/deleted.

Our main result is an allocator with worst-case expected overhead $\mathrm{polylog}(\epsilon^{-1})$. This exponentially improves the previous worst-case expected overhead $\widetilde{O}(\epsilon^{-1/2})$ achieved by Farach-Colton, Kuszmaul, Sheffield, and Westover (2024), narrowing the gap towards the $\Omega(\log \epsilon^{-1})$ lower bound. Our improvement is based on an application of the sunflower lemma previously used by Erdős and Sárközy (1992) in the context of subset sums.

Our allocator achieves polylogarithmic overhead only in expectation, and sometimes performs expensive rebuilds. Our second technical result shows that this is necessary: it is impossible to achieve subpolynomial overhead with high probability.

1 Introduction

Memory allocation is one of the oldest problems in computer science, both in theory and in practice.

Consider an array of M memory slots indexed by $\{0, 1, \ldots, M-1\}$, and a dynamically changing collection of ***objects*** X,[1] where object $x \in X$ has ***size*** $\mu(x) \in \mathbb{Z}^+$. An ***allocation*** is an assignment of the objects to non-overlapping contiguous regions in memory, or formally, an allocation map $\phi\colon X \to \{0, 1, \ldots, M-1\}$ such that $\phi(x) + \mu(x) \le M$, and the intervals $[\phi(x), \phi(x) + \mu(x))$ are disjoint for all $x \in X$. An ***allocator*** needs to maintain the allocation while handling two types of ***updates*** to X in an online fashion (starting from the initial state $X = \varnothing$):

- ***Inserting*** a new object x of size $\mu(x)$ into X (memory allocation).
- ***Deleting*** an object $x \in X$ from X (memory deallocation / free).

We sometimes refer to objects currently in X as ***live objects*** for emphasis.

In the classic setting of memory allocation, objects cannot be moved after being allocated. Upon inserting an object x, the allocator decides its location $\phi(x)$, and x continues to occupy the memory interval $[\phi(x), \phi(x) + \mu(x))$ until its deletion. After a few updates, the free space in memory may become fragmented. When inserting a new object x, even though the total empty space is larger than $\mu(x)$, there may be no contiguous region that fits it. This causes a waste of memory.

*cejin@berkeley.edu. This work is supported by the Miller Research Fellowship at the Miller Institute for Basic Research in Science, UC Berkeley.

[1] *Objects* are also referred to as memory requests, items, or blocks in the literature.

Unfortunately, classic results show that any online allocator must inevitably waste *most of* the memory in this no-move setting. Formally, we say an update sequence has ***load factor*** $1-\epsilon$ if, at every point in time, the total size of live objects, $\sum_{x\in X}\mu(x)$, is at most $(1-\epsilon)M$.[2] The load factor is known to the allocator in advance. Then, any no-move allocator (even allowing randomization) can only handle load factor $1-\epsilon \le O(\frac{1}{\log M})$, which tends to 0 as M grows [Rob71, Rob74, LNO96, BCF+25].

Memory Reallocation. This situation has motivated the theoretical study of *Memory **Re**allocation* in various settings [HP04, SSS09, LGL15, BFF+15a, BFF+15b, BFF+17, Kus23, FKSW24], where the goal is to support much higher load factor (ideally arbitrarily close to 1), by allowing the allocator to move around existing objects in memory, while incurring a small reallocation overhead. We follow the formal setup of this problem considered by Farach-Colton, Kuszmaul, Sheffield, and Westover [FKSW24] (which was also studied earlier in [Kus23, BFF+17, NT01] with extra history-independent or cost-oblivious requirements; see Section 1.3), defined as follows:

In the *Memory Reallocation* problem, to handle an update, the allocator may move the existing objects, at an (unnormalized) ***switching cost*** defined as the total size of moved objects (including the object being inserted/deleted). The ***overhead factor*** (or ***overhead*** for short) is the unnormalized switching cost divided by the size of the object being inserted/deleted. Formally, when inserting (or deleting) an object x, changing X to $X' = X \sqcup \{x\}$ (or $X' = X \setminus \{x\}$), if the allocator changes the allocation map $\phi\colon X \to \{0,1,\dots,M-1\}$ to $\phi'\colon X' \to \{0,1,\dots,M-1\}$, then the overhead factor incurred for this update is

$$1 + \frac{1}{\mu(x)} \cdot \sum_{\substack{y\in X\cap X':\\ \phi(y)\neq\phi'(y)}} \mu(y).$$

When the load factor is $1-\epsilon$, the following folklore deterministic allocator [BFF+17, Kus23] achieves an $O(\epsilon^{-1})$ overhead factor (remarkably, independent of the memory size M) for every update: to insert an object x, one simply finds an interval of size $O(\epsilon^{-1}\mu(x))$ with at least $\mu(x)$ empty memory slots (which must exist by an averaging argument), and reorganizes all objects contained in this interval to create a contiguous empty space that fits x.

On the other hand, there has been no known lower bound that rules out constant overhead. This raises the following main question:

What is the best possible reallocation overhead as a function of the load factor $1-\epsilon$?

To be more precise, when we say an allocator $\mathcal{A}$ achieves overhead $f(\epsilon)$, we mean that for every $\epsilon > 0$, there exists a family of allocators $\{\mathcal{A}_M\}_{M\in\mathbb{Z}^+}$ such that for every $M \in \mathbb{Z}^+$, $\mathcal{A}_M$ achieves overhead $f(\epsilon)$ on all input instances with M memory slots and load factor $1-\epsilon$.

Previous results. Recent works have made progress on this question via *randomization*. Formally, we say a randomized allocator achieves (worst-case) ***expected overhead*** $f(\epsilon)$ (against an oblivious adversary), if for every update sequence that is fixed in advance with load factor $1-\epsilon$,

[2] A more relaxed definition of load factor, which appeared in some previous works such as [Kus23], allows the total size of live objects to be at most $\lfloor(1-\epsilon)M\rfloor + 1$ (where $\epsilon > 0$). These two definitions are equivalent up to changing ϵ by a constant factor; see Observation 2.1.

the overhead factor incurred for each update has expectation at most $f(\epsilon)$ over the randomness of the allocator.

In the special case where every object has size at most $\epsilon^4 M$, Kuszmaul [Kus23] designed a randomized allocator with $O(\log \epsilon^{-1})$ expected overhead, exponentially better than the folklore allocator. His allocator is based on a variant of uniform probing with strong history independence. For larger object sizes, however, he conjectured that the folklore $O(\epsilon^{-1})$ overhead was optimal. Surprisingly, this conjecture was later disproven by Farach-Colton, Kuszmaul, Sheffield, and Westover [FKSW24], who designed a randomized allocator with expected overhead $O\big((\frac{1}{\epsilon})^{1/2} \operatorname{polylog}(\frac{1}{\epsilon})\big)$.

The allocators of [Kus23] and [FKSW24] additionally satisfy a ***resizability*** property, namely that the live objects X are always allocated to the prefix $\big[0, \frac{1}{1-\epsilon}\sum_{x\in X}\mu(x)\big)$ of the memory. Although no lower bound was known for general allocators, [FKSW24] proved that resizable allocators must incur overhead $\Omega(\log \epsilon^{-1})$ (even if the entire update sequence is known in advance). However, this still leaves an exponential gap from their $O\big((\frac{1}{\epsilon})^{1/2} \operatorname{polylog}(\frac{1}{\epsilon})\big)$ upper bound.

Despite the lack of progress in the general case, several special classes of update sequences are known to admit logarithmic expected overhead. Examples (in addition to the aforementioned tiny-object case [Kus23]) include: update sequences where all objects have power-of-two sizes [Kus23], update sequences with only a constant number of distinct object sizes, all within a constant factor of each other (this was mentioned in [FKSW24] as a corollary of their techniques), as well as a certain distribution of random update sequences [FKSW24]. Inspired by these successful examples, the authors of [FKSW24] speculated that the general case could possibly be tackled by a structure-versus-randomness dichotomy, as is often featured in additive combinatorics. To add to this optimism, additive combinatorial techniques have led to advances in several algorithmic problems involving packing various-sized objects (such as Knapsack and Bin Packing; see Section 1.3). Memory Reallocation appears to be another problem of this type, albeit with an additional dynamic flavor.

1.1 Our results

Upper bound. We present the first allocator with expected overhead factor $\operatorname{polylog}\big(\frac{1}{\epsilon}\big)$, exponentially improving the previous state-of-the-art result by Farach-Colton, Kuszmaul, Sheffield, and Westover [FKSW24]. Like [FKSW24] and [Kus23], our allocator is resizable.

Theorem 1.1 (Section 4)**.** *The Memory Reallocation problem can be solved by a randomized resizable allocator with worst-case expected overhead $O(\log^4 \epsilon^{-1} \cdot (\log\log \epsilon^{-1})^2)$ against an oblivious adversary.*

As a confirmation of the conjecture in [FKSW24], Theorem 1.1 exploits the additive combinatorics of the object sizes, but the actual design of the allocator turns out to be less complicated than we expected. The only combinatorial tool we need is a simple but beautiful theorem of Erdős and Sárközy about subset sums [ES92], which they proved using the celebrated sunflower lemma pioneered by Erdős and Rado [ER60]. See Section 1.2 for a brief overview of the techniques.

By a simple transformation (see Observation 2.1), Theorem 1.1 also implies an allocator with $\operatorname{polylog}(M)$ overhead, even when all the M memory slots can be full.

Corollary 1.2. *The Memory Reallocation problem (where the memory can be full) can be solved by a randomized resizable allocator with worst-case expected overhead $O(\log^4 M \cdot (\log\log M)^2)$ against an oblivious adversary.*

Lower bound. Our next result is a logarithmic lower bound for the overhead factor, which is also the first known super-constant lower bound that applies to general allocators without extra constraints. Previously, a logarithmic lower bound was known for resizable allocators [FKSW24],[3] and a near-logarithmic lower bound was known for strongly history-independent allocators [Kus23].

Theorem 1.3 (Section 5)**.** *In the Memory Reallocation problem, the worst-case expected overhead of any allocator (against an oblivious adversary) must be at least* $\Omega(\log \epsilon^{-1})$.

Closing the near-quartic gap between our upper bound and lower bound is an interesting open question.

High-probability guarantee? Theorem 1.1 only achieves worst-case *expected* logarithmic overhead. On each update, our allocator may incur large overhead with non-negligible probability, due to periodically performing expensive rebuilds. This raises a natural question: can we improve Theorem 1.1 to achieve $\operatorname{polylog} \epsilon^{-1}$ overhead on every update *with high probability* in ϵ^{-1}, or even deterministically?

Unfortunately, our next theorem implies that this is not possible. In the following, the *squared overhead* incurred for an update is defined as the square of the overhead factor for that update.

Theorem 1.4 (Section 5)**.** *In the Memory Reallocation problem, the worst-case expected squared overhead of any allocator (against an oblivious adversary) must be at least* $\Omega(\epsilon^{-1/7})$*, even when all objects have size* $\Theta(\epsilon^{3/7}M)$.

In the scenario of Theorem 1.4, the overhead factor of any update is always at most $O(\epsilon^{-3/7})$ (which corresponds to reorganizing the entire memory). Hence, if there is an allocator for load factor $1-\epsilon$ that achieves, for each update, an overhead at most f with at least $1-\delta$ probability, then either $f \geq \Omega(\sqrt{\epsilon^{-1/7}}) = \Omega(\epsilon^{-1/14})$ or $\delta \geq \Omega\big(\frac{\epsilon^{-1/7}}{(\epsilon^{-3/7})^2}\big) = \Omega(\epsilon^{5/7})$ must hold.

The exponent $1/7$ in Theorem 1.4 can be slightly improved by a more complicated refinement of our proof, which we omit in this paper. We leave it to future work to determine the tight bound.

Similarly to Corollary 1.2, we can show variants of Theorems 1.3 and 1.4 with lower bounds stated in terms of the number of memory slots M (which may be full); see details in Section 5.

1.2 Technical overview

Upper bound. We briefly describe the technical ingredients behind our proof of Theorem 1.1.

Our starting point is a generic substitution strategy employed by Farach-Colton, Kuszmaul, Sheffield, and Westover [FKSW24] in their allocator beating the folklore $O(1/\epsilon)$ overhead. Their allocator is resizable, i.e., the live objects X are always allocated within the prefix $[0, \frac{1}{1-\epsilon}\sum_{x\in X}\mu(x))$ of memory. For a resizable allocator, it is always safe to append any inserted object immediately after the rightmost live object in memory. The challenge is to restore resizability whenever a deletion happens and creates a gap in memory. If the deleted object x is near the right end of the allocation, then it is cheap to shift leftward all the objects after x, filling in the gap. However, this becomes expensive if the deleted object x is far from the right end.

A key idea of [FKSW24] is a substitution strategy: find another object y near the right end such that $\mu(x)-\mu(y)$ is non-negative but small, and move y into x's original place, nearly filling

[3] An advantage of [FKSW24]'s lower bound is that it also applies to offline resizable allocators, whereas ours only applies to online allocators.

in the gap. The original place of y then becomes a gap, but it is cheaper to fill it since it is closer to the right. In order to ensure that such a good substitute object y can always be found near the right end, [FKSW24]'s allocator carefully sorts and organizes the objects in memory, and performs periodic rebuilds. Intuitively, if many updates have happened and consumed most of the available substitute objects on the right, then it performs an expensive rebuild to replenish the supply.

The allocator in [FKSW24] guarantees that the substitute object y and the deleted object x have relative size difference $O(\epsilon^{1/2})$. While moderately small, this size difference is still non-negligible and accumulates over time, causing significant waste of memory. This is the main reason why [FKSW24] only achieves polynomial overhead instead of polylogarithmic. On the positive side, [FKSW24]'s substitution strategy with rebuilds can imply the following interesting corollary (which was noted in the conclusion section of their paper):

Proposition 1.5 (based on [FKSW24])**.** *If there are only k distinct object sizes, all in $[\mu, 2\mu]$, then an allocator can achieve worst-case expected overhead $O(k \log \frac{M}{k\mu})$.*

When very few distinct object sizes exist, Proposition 1.5 offers a significantly improved overhead factor compared to the general case. Roughly speaking, Proposition 1.5 is possible because one can ensure that the substitute object y always has exactly the same size as x.

Our strategy for proving Theorem 1.1 can be viewed as reducing from the general case to the special case of few distinct object sizes in Proposition 1.5. The key tool that enables this reduction is an additive combinatorial result of Erdős and Sárközy [ES92], which they proved using the sunflower lemma: in any (multi)set S of integers from $[n]$, one can find at least $\Omega(\frac{|S|}{\log^2 n})$ *disjoint* subsets $\{B_i\}_i$, each of cardinality $|B_i| \le O(\log n)$, which all have equal sum $\sum_{a \in B_i} a$.[4]

When the object sizes are within a constant factor of each other, we can appropriately round each object size up by a factor of at most $1+\epsilon$ (which is allowable when the load factor is $1-O(\epsilon)$), and then iteratively apply the above lemma with $O(\epsilon^{-1})$ in place of n. This partitions the objects into *bundles* of $O(\log \epsilon^{-1})$ objects each, with at most $\text{polylog}(\epsilon^{-1})$ distinct bundle sizes. We allocate each bundle contiguously in memory. In this way, our situation resembles the special case of few distinct object sizes as in Proposition 1.5. The difference is that we operate on bundles rather than individual objects; this increases the overhead factor by the number of objects in each bundle, which is only logarithmic.

Our proof of Theorem 1.1 proceeds not by a black-box reduction to Proposition 1.5, but rather by modifying its proof and handling additional implementation details related to bundling. For example, we need to unbundle a bundle when one of its objects is deleted, and bundle new objects as they are inserted. During each rebuild operation, we will rebundle the affected objects, so that the number of distinct bundle sizes remains bounded by $\text{polylog}(\epsilon^{-1})$. (We also need to lift the bounded-ratio assumption in Proposition 1.5; this was already addressed by [FKSW24], and we omit the details in this overview.)

Since [FKSW24] did not explicitly prove Proposition 1.5, we include a proof below for convenience. This also serves as a warm-up for our main proof of Theorem 1.1.

Proof of Proposition 1.5 (based on [FKSW24]). We maintain the *prefix property*, namely that all live objects occupy a prefix of the memory without any gaps in between. The objects are partitioned

[4] The actual theorem in [ES92] was not this statement, but rather its direct corollary that the subset sums of S contain an arithmetic progression of length $\Omega(\frac{|S|}{\log^2 n})$. The latter statement is of more historical interest in the literature, but less useful for us.

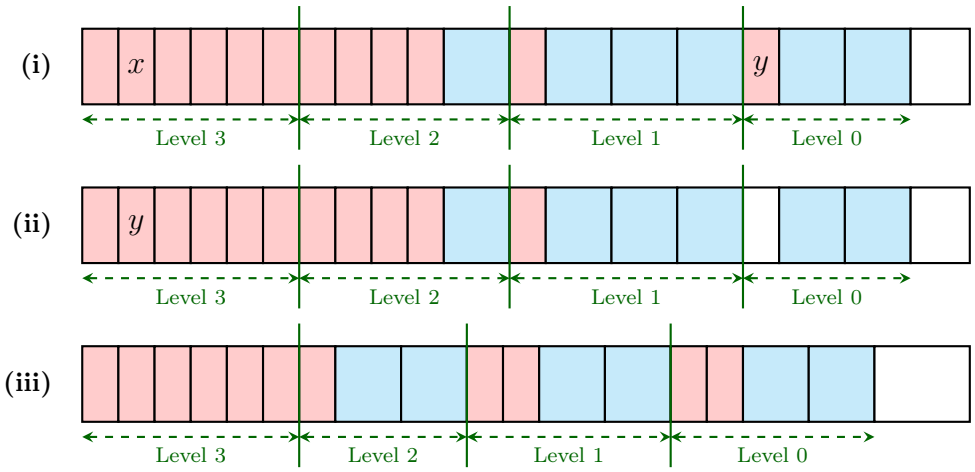

Figure 1: An illustration for the proof of Proposition 1.5 with two types of object sizes. (i) Initial state at $c = 3$, where object x is to be deleted. We find another object y of the same size in level 0. (ii) Swap x and y before deleting x. (iii) Increase the counter to $c = 4$, and rebuild levels $2, 1$, and 0.

into $\ell \coloneqq \lceil \log(\frac{M}{k\mu}) \rceil$ *levels* numbered $\ell - 1, \dots, 1, 0$ from left to right, each occupying a contiguous interval of memory.[5]

Intuitively, levels further to the left can have larger total size and are less frequently rebuilt. Formally, we maintain the following two invariants, where the counter c is initialized to a random integer and incremented by one per update (i.e., insertion or deletion).

- For every integer $j \in [1, \ell]$, the total number of objects in levels $\{j - 1, \dots, 1, 0\}$ is at most $k \cdot 2^j + (c \bmod 2^j) < (k + 1)2^j$.
- For every object size τ and integer $j \in [1, \ell - 1]$, if levels $\{\ell - 1, \dots, j + 1, j\}$ contain at least one size-τ object, then the total number of size-τ objects in levels $\{j - 1, \dots, 1, 0\}$ is at least $2^j - (c \bmod 2^j) \geq 1$.

Clearly, the two invariants hold at the beginning with empty memory.

To insert an object, we add it to level 0 immediately after the current rightmost object.

To delete a size-τ object x, if x is not already in level 0, we swap x with another size-τ object y from level 0, which must exist due to the second invariant. See Fig. 1. Hence, the deletion only creates a gap in level 0, which will be filled during the rebuild operation described below.

After every update, we increment the counter c and then perform a rebuild as follows. Pick the largest integer $j^* \leq \ell - 1$ such that 2^{j^*} divides c. Then, we collect all objects in levels $\{j^*, \dots, 1, 0\}$, and repartition them by the following greedy rule: Define $(n_0, n_1, \dots, n_{j^*-1}) \coloneqq (2, 2, 4, 8, 16, 32, \dots)$ and $n_{j^*} \coloneqq +\infty$. For each object size τ, we put n_0 size-τ objects in level 0, then put n_1 of the remaining size-τ objects in level 1, and so on, until the number of remaining size-τ objects is smaller than n_j for the current level j, at which point we finish by putting all of them in level j.

To prove the first invariant for $j \in [1, \ell - 1]$ (the $j = \ell$ case vacuously holds by our definition of ℓ), consider the most recent rebuild that involves level j, which happened $c \bmod 2^j$ updates before (if no such rebuild happened before, the proof is similar). Right after this rebuild, the number of

[5] This is slightly different from [FKSW24], who defined levels as nested subsets instead of disjoint subsets.

objects in levels $\{j-1,\ldots,1,0\}$ is at most $n_0+n_1+\cdots+n_{j-1}=2^j$ for each object size τ, and hence at most $k\cdot 2^j$ in total. Each subsequent update increases this count by at most one, so the first invariant holds. The second invariant can be proved similarly (we omit the details here).

For each update, a rebuild of levels $\{j,\ldots,1,0\}$ is triggered with probability $O(2^{-j})$, incurring a switching cost bounded by their total size, which is $O(k\cdot 2^j\cdot\mu)$ by the first invariant. Hence, the expected overhead factor is $\sum_{j=0}^{\ell-1}O(2^{-j})\cdot O(k\cdot 2^j\cdot\mu)/\mu=O(k\ell)=O(k\log(\frac{M}{k\mu}))$ as claimed. □

Lower bounds (general strategy). We begin with a minor technical simplification for all our lower bound proofs: by a rounding argument (see Observation 2.1), it suffices to prove overhead lower bounds in the scenario where the number of slots is $M=\Theta(1/\epsilon)$ and the memory may be full. See details in Section 5.

Now we discuss the general proof ideas for Theorem 1.3 and Theorem 1.4. Recall the allocator in Theorem 1.1 exploits the "additive coincidences" of object sizes, namely different small subsets of equal sum, to perform cheap substitutions. The proofs of lower bounds proceed in the opposite direction: we design hard instances that lack additive coincidences, so that cheap substitutions are impossible. This idea was already reflected in the previous lower bound by [FKSW24] against resizable allocators, and we will further build on this idea.

We will frequently use the following simple but crucial notion implicitly introduced by [FKSW24]: between two full memory states ϕ,ϕ', a *maximal changed interval* is an inclusion-maximal interval of memory such that no object in ϕ intersecting this interval retains its location in ϕ'. Let $\Delta(\phi,\phi')$ denote the total length of all the maximal changed intervals. Then, transforming ϕ into ϕ' requires a total switching cost at least $\Delta(\phi,\phi')$. See Fig. 2 for an illustration and Section 5.1 for formal definitions.

A maximal changed interval either permutes the objects inside, or substitutes them by a different set of objects with the same total size. At a high level, by designing object sizes with few additive coincidences, we have control of the possible substitutions that might occur. This helps us show limitations of low-overhead allocators, which can only produce short maximal changed intervals.

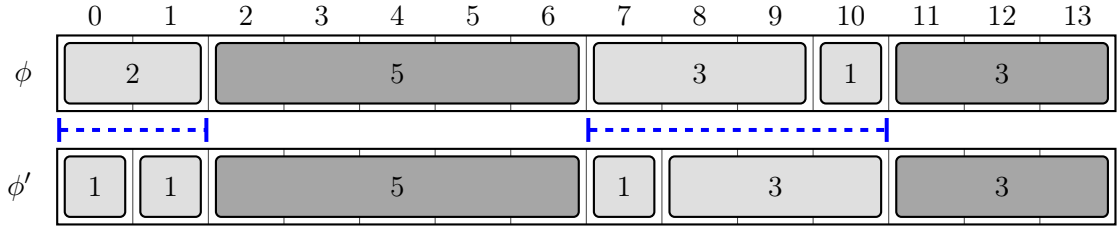

Figure 2: Visualization of two full memory states ϕ,ϕ' with $M=14$ slots. The unchanged objects in ϕ and ϕ' (highlighted in dark gray) have sizes 5 and 3. The maximal changed intervals between them (highlighted in dashed blue) are $[0,2)$ and $[7,11)$.
To transform ϕ to ϕ', the allocator must incur a total switching cost of at least $\Delta(\phi,\phi')=2+3+1=6$.

Logarithmic lower bound. Now we sketch the proof of Theorem 1.3, which is relatively simple. Recall that it suffices to prove a worst-case expected $\Omega(\log M)$ overhead lower bound when all M memory slots can be full. We construct a set $S=\{s_1,s_2,\ldots,s_k\}\subset\mathbb{Z}^+$ such that for every pair of distinct $a,b\in[k]$, there exist $s'_a,s'_b\in\mathbb{Z}^+$ with $s'_a+s'_b=s_a+s_b$ so that the following property holds:

- Let $S' = (S \setminus \{s_a, s_b\}) \sqcup \{s'_a, s'_b\}$. If $X \subseteq S$ and $X' \subseteq S'$ satisfy $\sum_{x \in X} x = \sum_{x' \in X'} x'$ and $s_a \in X$, then $s_b \in X$ must hold as well.

One can verify that the following construction works and ensures every integer is bounded by $O(2^{2k})$: $s_i \coloneqq 2^{i+k} + 2^i$ for all $i \in [k]$, $s'_a \coloneqq 2^{a+k} + 2^b$ and $s'_b \coloneqq 2^{b+k} + 2^a$. We further modify this construction to make every integer in $\Theta(2^{2k})$ by padding (see details in Section 5.2). Define the total number of memory slots to be $M \coloneqq s_1 + \cdots + s_k$ (hence, $k = \Theta(\log M)$ and $s_i \in \Theta(M/\log M)$). In the hard instance, we first insert k objects of sizes $s_1, s_2, \ldots, s_k$ to completely fill the memory, and denote the current memory state by ϕ. Then, we pick two distinct indices $a, b \in [k]$ uniformly at random, delete the objects of sizes s_a, s_b and insert objects of sizes s'_a, s'_b as defined above. Denote the current memory state by ϕ'. Now, consider the maximal changed interval $[L, R)$ between ϕ, ϕ' that contains the size-s_a object of ϕ. Let X denote the set of sizes of objects in ϕ contained in $[L, R)$, so $s_a \in X$. Then, the property above implies that $s_b \in X$ as well. Therefore, $R - L$ is at least the distance between the size-s_a and size-s_b objects in ϕ, which has expectation $\Omega(M)$ over the randomness of a and b. Thus, the total expected switching cost for the two insertions and two deletions is at least $\mathbf{E}[\Delta(\phi, \phi')] \geq \mathbf{E}[R - L] \geq \Omega(M)$, so the worst-case expected overhead is $\Omega(M)/\Theta(M/\log M) = \Omega(\log M)$ as claimed.

High-probability lower bound. Our proof of Theorem 1.4 is more involved. As a warm-up, it is instructive to see a proof sketch of the following worst-case lower bound against deterministic allocators with the prefix property (which is more or less the same as resizability):

Proposition 1.6. *Any deterministic allocator that satisfies the prefix property, namely that all live objects must occupy a prefix of the memory without any gaps in between, must incur worst-case overhead $\Omega(M^{1/4})$ when there are M memory slots which can be full.*

Compared to Theorem 1.4, here we are restricting to deterministic allocators to avoid probabilistic technicalities, but the main simplification of the proof comes from the prefix property assumption.

Proof of Proposition 1.6. We pick two integers $a, b \in [\mu, 2\mu]$ with $\mu = \Theta(\sqrt{M})$, so that all integers $ia + jb$ where $i, j \in \mathbb{Z} \cap [0, M/\mu]$ are distinct. For example, $a \coloneqq \mu \coloneqq \lceil \sqrt{M} \rceil$ and $b \coloneqq a + 1$ satisfy this property. In our hard update sequence, we first insert $\lfloor M/b \rfloor$ objects of size b. Then, we gradually replace all of them by size-a objects as follows: in every iteration, we delete one size-b object, and insert $O(1)$ size-a objects until the total size of live objects is in $(M - a, M]$. (This is almost the same as the update sequence in [FKSW24]'s proof of the amortized logarithmic lower bound against resizable allocators.) Denote the memory state right after the i-th iteration ($i \leq \lfloor M/b \rfloor$) by ϕ_i. For a size-a object at location λ, define its *potential* to be $M - \lambda \geq 0$; the potential of a memory state is simply the total potential of all the size-a objects in it. At the beginning, the total potential is zero. In the end, among all the $\lfloor M/a \rfloor$ size-a objects, at least half of them have potential $\Omega(M)$, with total potential $\Omega(M^2/a) = \Omega(M^{3/2})$.

Between two adjacent memory states ϕ_i, ϕ_{i+1}, consider all the maximal changed intervals $[L_k, R_k)$, with the exception that the interval $[L_f, R_f)$ containing the rightmost object of ϕ_{i+1} is redefined with right boundary $R_f \coloneqq M$. See an illustration in Fig. 3. Since the rightmost gap in ϕ_{i+1} is smaller than a, the total switching cost to transform ϕ_i into ϕ_{i+1} is at least $\sum_{k \neq f} (R_k - L_k) + (R_f - L_f - a)$. Assuming the allocator incurs worst-case switching cost S per update, this implies $\sum_k (R_k - L_k) \leq O(S) + a$.

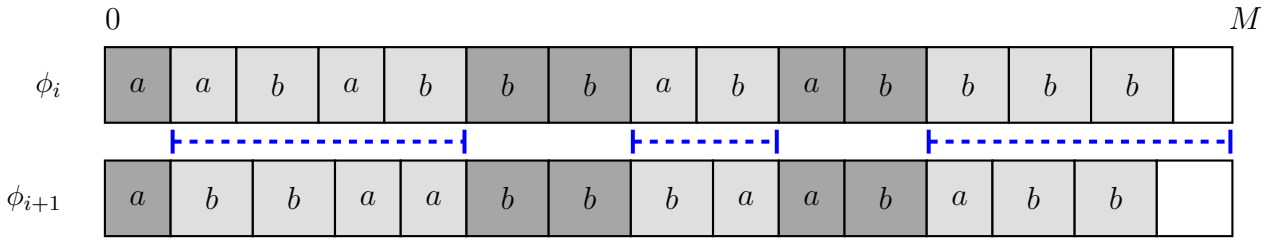

Figure 3: An illustration for the proof of Proposition 1.6. Between memory states ϕ_i, ϕ_{i+1} satisfying the prefix property, every maximal changed interval $[L_k, R_k)$ (marked in dashed blue), except for the rightmost one, permutes the objects inside.

Every $[L_k, R_k)$, except for the rightmost one, is completely filled by objects, and hence must contain the same number of size-a and size-b objects in both states, due to the property of a and b stated at the beginning. Now we analyze the total potential difference between ϕ_i and ϕ_{i+1}:

- Inside every $[L_k, R_k)$ $(k \neq f)$, we can perfectly pair up the a-objects in ϕ_i with those in ϕ_{i+1}, forming at most $(R_k - L_k)/a$ pairs. Each pair contributes potential difference less than $R_k - L_k$. Hence, the total contribution to the potential difference is $\leq (R_k - L_k)^2/a$.
- Inside the rightmost maximal changed interval $[L_f, R_f)$ in ϕ_i (and in ϕ_{i+1}), there are at most $(R_f - L_f)/a$ size-a objects, each with potential at most $M - L_f = R_f - L_f$. Hence, the total contribution to the potential difference is also $\leq (R_f - L_f)^2/a$.

Summing up, the total potential difference between ϕ_i, ϕ_{i+1} is at most

$$\sum_k (R_k - L_k)^2/a \leq \big(\sum_k (R_k - L_k)\big)^2/a \leq (O(S) + a)^2/a = O(S^2/a + a).$$

Summing up over all $i \leq \lfloor M/b \rfloor$, we obtain that the potential difference between the initial and final states is $O(\frac{M}{b}(\frac{S^2}{a} + a)) = O(S^2 + M)$. Since we showed earlier that this potential difference is $\Omega(M^{3/2})$, this implies $S \geq \Omega(M^{3/4})$ for sufficiently large M. Hence, the worst-case overhead factor is at least $\Omega(S/\mu) = \Omega(M^{1/4})$ as claimed. □

Relaxing the prefix requirement in Proposition 1.6 poses more technical challenges. In particular, the update sequence in the proof of Proposition 1.6 becomes easy: an allocator could always store size-a objects in a prefix of the memory, and size-b objects in a suffix, leaving a gap somewhere in the middle. Each update can be handled with constant overhead, by performing insertions and deletions near the gap.

We now sketch how to modify the proof of Proposition 1.6 to obtain an $M^{\Omega(1)}$ worst-case bound for general deterministic allocators (without details on the calculation of parameters). We pick object sizes $a, b \in \Theta(\mu)$ with the same roles as before, and another object size $c \in \Theta(\mu)$ whose role will be explained later. In our construction, we ensure the total size of size-a, size-b, and size-c objects stays in $M - \Theta(\mu)$, and we always insert another special object f (which we call the "finger object") so that all M memory slots are fully occupied. By selecting a, b, c to avoid additive coincidences, we can achieve the following crucial property (formally stated in Lemma 5.14):

- Between two full memory states ϕ and ϕ', if a maximal changed interval does not contain the finger object of either state, then it can only permute the objects within it.

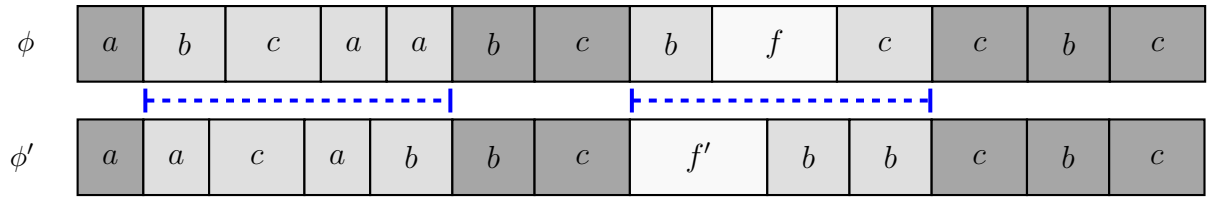

Figure 4: An illustration for Lemma 5.14 with two memory states ϕ, ϕ' (corresponding to $S_{3,5}$ and $S_{3,4}$ in Definition 5.10 respectively). Two maximal changed intervals are marked in dashed blue. The first interval contains objects of sizes $\{a, a, b, c\}$ in both ϕ, ϕ'. The second interval contains the finger objects of both ϕ, ϕ'.

This property is reminiscent of our previous proof for the prefix setting (Proposition 1.6): local permutation of objects may happen anywhere in memory, but insertions and deletions can only happen near the rightmost gap (or, in the non-prefix setting here, the finger object). However, unlike in the prefix setting, here the finger object may not always stay at the same place throughout the process. If we could somehow constrain the finger object to remain close to a location p, then we could adapt the previous potential-based proof by defining the potential of a size-a object at location λ to be $\approx |p - \lambda|$ (see the proof of Lemma 5.24).

It remains to describe how to constrain the finger object in a relatively small interval. We modify our earlier hard instance to include Q size-c objects, in addition to the $\Theta(M/b)$ size-b objects at the beginning, where Q is polynomially smaller than M/b. As before, we gradually replace the size-b objects by size-a objects. However, we interrupt this process at a random time step unknown to the allocator, and start a "surprise inspection" instead, where we gradually delete all the size-c objects. To stay prepared for the surprise inspection, the allocator must always keep all the size-c objects close to the finger object; otherwise, it would be expensive to delete all the size-c objects as demanded, since deletions can only happen near the finger object as we mentioned earlier (this strategy is formally implemented in the proof of Lemma 5.20). Consequently, it is too expensive for the allocator to move the finger object significantly far away, since it would have to carry along a large number of size-c objects to move together with the finger object (this is formally proved in Lemma 5.21 via a similar potential argument). Therefore, the finger object is always constrained in a small interval as desired.

The proof strategy described above yields a deterministic worst-case $M^{\Omega(1)}$ lower bound on the overhead factor. Since our hard update sequence is oblivious, the same proof also applies to randomized allocators with high-probability guarantees against an oblivious adversary. In fact, our proof gives a polynomial lower bound already for the L^2 norm of the overhead factor, as stated in Theorem 1.4.

1.3 Related works

Memory (Re-)allocation. Bender, Farach-Colton, Fekete, Fineman, and Gilbert [BFF+17] defined and studied the Memory Reallocation problem in the *cost-oblivious* setting, where the cost of inserting, deleting, and moving an object of size μ is an arbitrary unknown subadditive function $c(\mu)$. This is substantially more general than many other works (including ours), e.g., [NT01, Kus23, FKSW24], which assume $c(\mu) = \mu$.

Naor and Teague [NT01] and Kuszmaul [Kus23] studied Memory Reallocation (for both unit-size and variable-size objects) in *history-independent* settings. In the strongly history-independent

setting, the allocation of a set S of labeled objects should only depend on the set S and the randomness of the allocator. Kuszmaul [Kus23] showed that, even for the unit-size case, a strongly history-independent allocator must incur expected reallocation overhead $\Omega(\log \epsilon^{-1} / \log\log \epsilon^{-1})$, nearly matching the previous $O(\log \epsilon^{-1})$ upper bound by Berger, Kuszmaul, Polak, Tidor, and Wein [BKP+22].

The very recent work of Bender, Conway, Farach-Colton, Komlós, Kuszmaul, and Wein [BCF+25] studied a variant of memory allocation (without reallocation) in which each object being inserted is allowed to be *fragmented* into up to k contiguous pieces. In this request fragmentation setting, they bypassed the logarithmic barrier for the memory competitive ratio in the classic $k = 1$ case [Rob71, Rob74, LNO96, BCF+25]. They also determined the optimal memory competitive ratio in this setting.

Sunflower lemma. The classic Erdős–Rado sunflower lemma [ER60] was quantitatively improved in a recent breakthrough by Alweiss, Lovett, Wu, and Zhang [ALWZ21], and further refined by [Rao20, Tao20, FKNP21, BCW21]; see the surveys by Rao [Rao23, Rao26]. The recent advances on sunflower lemmas led to the resolution of the Kahn–Kalai conjecture [PP24].

In theoretical computer science, the sunflower lemma has found applications in data structure lower bounds [FMS97, GM07, RR18], circuit lower bounds [Raz85, AB87, Ros14, CKR22, BM25, dRV25, CGR+25] and lifting theorems [LMM+22], parameterized algorithms [Mar05], sparsifying or analyzing DNF formulas [GMR13, LZ19, LSZ19, AW21, Tan22], etc. See [ALWZ20, Section 1.2 of the conference version] for a more comprehensive list of references.

Subset sums and algorithm design. Our key lemma is extracted from Erdős and Sárközy's proof that the subset sums of any $S \subseteq [n]$ must contain an arithmetic progression of length $\Omega(|S|/\log^2 n)$ [ES92]. Schoen [Sch11] subsequently improved this bound to $\Omega(|S|/\log n)$, but his proof does not yield the equal-sum disjoint subsets (crucial for our approach) as [ES92]'s proof does. The results of [ES92, Sch11] can be greatly improved in the regime where $|S| \geq n^c$ for constant $c > 0$, e.g., [AF88, Fre93, Sár94, SV06b, SV06a, CFP21]. Some of these results have found algorithmic applications in recent pseudopolynomial and approximation algorithms for 0-1 Knapsack and related problems, e.g., [GM91, BW21, CLMZ24b, Bri24, Jin24, CLMZ24a, CLMZ24c, CMZ25]. Additive combinatorics of subset sums has also led to faster algorithms for Bin Packing [NPSW23, JSS21].

Very recently, motivated by questions related to Subset Sum algorithms, Chen, Mao, and Zhang [CMZ26] designed a fast algorithm for constructing the equal-sum disjoint subsets as given by [ES92]'s proof. Their result may potentially be helpful in designing a time-efficient implementation of our allocator; see further discussions in Section 6.

Paper organization

Section 2 gives some useful preliminaries. Section 3 proves the key combinatorial lemmas used by our allocator. Section 4 describes our allocator. Section 5 proves the lower bounds. We conclude with several open questions in Section 6.

2 Preliminaries

Notations. Denote $[n] = \{1, 2, \ldots, n\}$ and $\mathbb{Z}^+ = \{1, 2, \ldots\}$. All logarithms are base two unless otherwise specified.

For $b > 0$, let $a \bmod b$ denote the unique number $r \in [0, b)$ such that $a - r$ is an integer multiple of b.

We often write $A \sqcup B$ instead of $A \cup B$ to emphasize that the sets A and B are disjoint.

On the definition of load factor. We have defined the load factor $1 - \epsilon$ (where $\epsilon > 0$) to mean that the total size of live objects at any time is at most $(1-\epsilon)M$, where M is the number of memory slots. Another definition in the literature (e.g., [Kus23]) relaxes this upper bound to $\lfloor (1-\epsilon)M \rfloor + 1$; in particular, when $M \leq 1/\epsilon$, this upper bound becomes M, i.e., the memory is allowed to be full. The following Observation 2.1 shows that these two definitions are actually equivalent up to changing ϵ by a constant factor. This fact will be convenient for proving lower bounds in Section 5.

Observation 2.1. *For $\epsilon > 0$, if there is an allocator $\mathcal{A}$ that (for all $M \in \mathbb{Z}^+$) handles total live objects size $\leq (1-\epsilon)M$ with overhead $f(\epsilon)$, then,*

1. *there is an allocator $\mathcal{A}'$ that (for all $M \in \mathbb{Z}^+$) handles total live objects size $\leq \lfloor (1-\epsilon)M \rfloor + 1$ with overhead $f(\epsilon/3)$, and*
2. *there is an allocator $\mathcal{A}''$ that (for all $M \in \mathbb{Z}^+$) works even when all the M memory slots can be full, with overhead $f(\frac{1}{3M})$.*

Proof. We first prove Item 1. Similar proof ideas already appeared in [Kus23].

We describe how to design the desired allocator $\mathcal{A}'$ given an implementation of $\mathcal{A}$. Suppose $\mathcal{A}'$ receives an input instance I' with M' memory slots $\{0, 1, \ldots, M'-1\}$ and maximum total live objects size $\leq \lfloor (1-\epsilon)M' \rfloor + 1$.

Based on instance I', define an input instance I for $\mathcal{A}$ as follows: there are $M \coloneqq 2M' + 1$ memory slots $\{0, 1, \ldots, M-1\}$. For every insertion/deletion of size $\mu'(x) \in \mathbb{Z}^+$ in the instance I', we correspondingly create an insertion/deletion of size $\mu(x) \coloneqq 2 \cdot \mu'(x)$ in I.

We run $\mathcal{A}$ on the constructed instance I, and translate the allocation of $\mathcal{A}$ to our allocator $\mathcal{A}'$ by the following rule: whenever $\mathcal{A}$ decides to allocate object x to the memory slots $[\phi(x), \phi(x) + \mu(x))$ (where integer $\phi(x) \in [0, M - \mu(x)] = [0, 2M' + 1 - 2\mu'(x)]$), we let $\mathcal{A}'$ allocate x to the memory slots $[\phi'(x), \phi'(x) + \mu'(x))$ where $\phi'(x) \coloneqq \lfloor \phi(x)/2 \rfloor$. One can verify that:

- $\phi'(x) \in [0, M' - \mu'(x)]$, i.e., the allocated memory slots for x in I' are in $[0, M')$, and
- $[\phi(x_1), \phi(x_1)+\mu(x_1)) \cap [\phi(x_2), \phi(x_2)+\mu(x_2)) = \varnothing$ implies $[\phi'(x_1), \phi'(x_1)+\mu'(x_1)) \cap [\phi'(x_2), \phi'(x_2)+\mu'(x_2)) = \varnothing$, i.e., the disjointness of the allocated intervals is preserved.

Thus, $\mathcal{A}'$ is a valid allocator for the instance I'.

The maximum total live object size of I equals twice that of I', and the memory size in I is $M = 2M' + 1$. Therefore, the load factor of I is

$$\frac{2 \cdot (\lfloor (1-\epsilon)M' \rfloor + 1)}{2M' + 1} = 1 - \frac{\lceil \epsilon M' \rceil - 0.5}{M' + 0.5} \leq 1 - \frac{\epsilon M'/2}{M' + 0.5} \leq 1 - \epsilon/3.$$

Hence, $\mathcal{A}$ can achieve overhead $f(\epsilon/3)$ on instance I.

Since the movement of our allocator $\mathcal{A}'$ is caused by the movement of $\mathcal{A}$ with the same moving objects (with size scaled by two, which does not affect the overhead factor), the overhead factor of $\mathcal{A}'$ on instance I' is also $f(\epsilon/3)$. This finishes the proof of Item 1.

We now derive Item 2 as an immediate consequence of Item 1. Given an input instance with M memory slots and maximum total live objects size $\leq M$, we can directly solve it using the allocator in Item 1 with ϵ set to $1/M$ so that $\lfloor (1-\epsilon)M \rfloor + 1 = M$. By Item 1, the overhead factor is $f(\epsilon/3) = f(\frac{1}{3M})$. □

Real-interval setting. When proving our upper bound in Section 4, to avoid excessive use of floors and ceilings, we use the following real-interval memory model introduced in [Kus23] and [FKSW24]: The memory is the *real interval* $[0, M)$ instead of discrete memory slots $\{0, 1, \ldots, M-1\}$. Each object x has a real-number size $\mu(x)$, so that the total size of live objects at any time is at most $(1-\epsilon)M$, where $1-\epsilon$ ($\epsilon > 0$) is the load factor. We are allowed to allocate x at any real location $\phi(x) \in [0, M - \mu(x)]$, subject to the constraint that the occupied intervals $[\phi(x), \phi(x) + \mu(x))$ are pairwise disjoint for all live objects x. The exact value of $M > 0$ is not important in this setting, and can be rescaled to any positive real number.

Observation 2.2. *If there is an allocator $\mathcal{A}$ in the real-interval setting with overhead $f(\epsilon)$, then there is an allocator $\mathcal{A}'$ that (for all $M \in \mathbb{Z}^+$) solves the original setting with M discrete memory slots with overhead $f(\epsilon)$.*

Proof. Represent the memory by the real interval $[0, M)$. We now proceed in a similar way to the proof of Observation 2.1. Given an update sequence with load factor $1-\epsilon$ for the original discrete setting with M memory slots, we feed the same sequence to the allocator $\mathcal{A}$ which operates on the real interval $[0, M)$. In particular, every object in this update sequence has integer size. Whenever $\mathcal{A}$ allocates an object x to the interval $[\phi(x), \phi(x) + \mu(x))$, in the discrete setting we allocate it to the memory slots indexed by $\lfloor \phi(x) \rfloor, \lfloor \phi(x) \rfloor + 1, \ldots, \lfloor \phi(x) \rfloor + \mu(x) - 1$. Using $\mu(x) \in \mathbb{Z}$, one can verify that this transformation preserves the disjointness of the allocated intervals, and all the used memory slots are from $\{0, 1, \ldots, M-1\}$.

The load factor of this update sequence in the real-interval setting is also $\frac{(1-\epsilon)M}{M} = 1-\epsilon$, so $\mathcal{A}$ achieves overhead factor $f(\epsilon)$. Thus, our allocator in the discrete setting also achieves overhead factor $f(\epsilon)$. □

3 Combinatorial lemmas

A *sunflower with p petals* is a family of p sets whose pairwise intersections are identical (called the *core*). The following is the state-of-the-art bound on the sunflower lemma, proved by Bell, Chueluecha, and Warnke [BCW21] (based on the recent breakthrough of [ALWZ21] and subsequent refinements [Rao20, Tao20, FKNP21]).

Lemma 3.1 (Sunflower lemma [BCW21])**.** *There is a constant $C \geq 4$ such that the following holds for all integers $p, k \geq 2$. Any family of at least $(Cp \log k)^k$ distinct k-element sets must contain a sunflower with p petals.*

Following Erdős and Sárközy [ES92], we use the sunflower lemma to prove the following Lemma 3.2. (If one uses the original Erdős–Rado bound of $1 + k!(p-1)^k$ [ER60] instead of the improved bound in Lemma 3.1, the last bound in Lemma 3.2 would worsen to $p \geq \frac{n}{C \log^2 w}$.)

Lemma 3.2. *There is a constant C such that for any $w \geq 4$ and any sequence of n positive integers $a_1, a_2, \ldots, a_n \in [w]$, there exist disjoint subsets $B_1, B_2, \ldots, B_p \subseteq [n]$ such that*

- *The sums $\sum_{i \in B_j} a_i$ are equal for all $j \in [p]$,*
- *$|B_1| = |B_2| = \cdots = |B_p| \leq C \log w$, and*
- *$p \geq \frac{n}{C \log w \log\log w}$.*

Proof. The proof is due to Erdős and Sárközy [ES92] (see also recent exposition in [Rao23]). We may assume w and n are large enough (otherwise, the lemma immediately holds for a large enough constant C). Let $k = \lceil 2 \log w \rceil$. For $s \leq nw$, define set family

$$\mathcal{S}_s := \left\{ B \in \binom{[n]}{k} : \sum_{i \in B} a_i = s \right\}.$$

Since $0 < \sum_{i \in B} a_i \leq kw$ holds for all $B \in \binom{[n]}{k}$, we clearly have $\binom{n}{k} = \sum_{1 \leq s \leq kw} |\mathcal{S}_s|$. By averaging, there exists s such that

$$|\mathcal{S}_s| \geq \frac{\binom{n}{k}}{kw} \geq \frac{(n/k)^k}{kw} \geq \left(\frac{n}{2k}\right)^k,$$

where the last inequality follows from $2^k \geq kw$ by our choice of $k = \lceil 2 \log w \rceil$.

Choose $p = \lfloor \frac{n}{2Ck \log k} \rfloor \geq \frac{n}{C' \log w \log\log w}$, where $C' > 0$ is some constant. Then, the above inequality implies $|\mathcal{S}_s| \geq (Cp \log k)^k$. Hence, by Lemma 3.1, $\mathcal{S}_s$ contains a sunflower with p petals $B'_1, B'_2, \ldots, B'_p$. Let the core be $R = B'_1 \cap \cdots \cap B'_p$, and define $B_1 := B'_1 \setminus R, \ldots, B_p := B'_p \setminus R$. Clearly, for all $j \in [p]$, $\sum_{i \in B_j} a_i = \sum_{i \in B'_j} a_i - \sum_{i \in R} a_i = s - \sum_{i \in R} a_i$ and $|B_j| = k - |R|$, so the requirements are indeed all satisfied. □

The following lemma is a simple refinement of the previous one. It allows us to shave off a logarithmic factor in our later application.

Lemma 3.3. *In the same setup as Lemma 3.2, one can additionally achieve $|B_1| + |B_2| + \cdots + |B_p| \geq \frac{n}{C \log\log w}$.*

Proof. We again assume n, w are large enough. We iteratively apply Lemma 3.2 as follows. Define $p^* = \lfloor \frac{n}{2C \log w \log\log w} \rfloor$, where C is the constant from Lemma 3.2.

First, apply Lemma 3.2 to $a_1, \ldots, a_n$, and obtain disjoint subsets $B_1^{(1)}, \ldots, B_{p_1}^{(1)} \subseteq [n]$ such that $|B_j^{(1)}| = k_1$ and $\sum_{i \in B_j^{(1)}} a_i = s_1$ for all $j \in [p_1]$. Since $p_1 \geq p^*$, we can decrease p_1 down to p^*.

Then, apply Lemma 3.2 to the remaining numbers in $a_1, \ldots, a_n$ indexed by $[n] \setminus (B_1^{(1)} \sqcup \cdots \sqcup B_{p_1}^{(1)})$, and obtain disjoint subsets $B_1^{(2)}, \ldots, B_{p_2}^{(2)} \subseteq [n] \setminus (B_1^{(1)} \sqcup \cdots \sqcup B_{p_1}^{(1)})$ such that $|B_j^{(2)}| = k_2$ and $\sum_{i \in B_j^{(2)}} a_i = s_2$ for all $j \in [p_2]$. As long as there remain at least $n/2$ numbers, we still have $p_2 \geq p^*$, so we can decrease p_2 to p^*.

Repeat this procedure until the m-th iteration, where m is the smallest integer such that $k_1 + k_2 + \cdots + k_m \geq C \log w$. We have $k_1 + k_2 + \cdots + k_m < C \log w + k_m \leq 2C \log w$. Note that the total size of subsets removed so far is

$$\sum_{i=1}^{m} \sum_{j=1}^{p^*} |B_j^{(i)}| = p^*(k_1 + k_2 + \cdots + k_m) \leq \frac{n}{2C \log w \log\log w} (2C \log w) \leq \frac{n}{\log\log w} \leq \frac{n}{2},$$

assuming w is large enough. Hence, when invoking Lemma 3.2, we always have at least $n/2$ objects remaining, so $p_1, p_2, \dots$ are indeed lower-bounded by p^*.

Finally, we return $B_j := \bigcup_{i=1}^{m} B_j^{(i)}$ for all $j \in [p^*]$. Clearly, we still have that all $\sum_{i \in B_j} a_i$ are equal, and that $|B_1| = \cdots = |B_{p^*}| = k_1 + k_2 + \cdots + k_m \in [C \log w, 2C \log w]$. Thus, $|B_1| + \cdots + |B_{p^*}| \geq p^* \cdot C \log w \geq \frac{n}{C' \log \log w}$ for some constant $C' > 0$. Therefore, all requirements are satisfied (with a possibly different constant C). □

Finally, we iteratively apply Lemma 3.3 to partition *all* the given integers into logarithmic-sized subsets which have logarithmically many distinct sums. This is the lemma that we will use for designing our allocator.

Lemma 3.4. *There exist constants C_1, C_2 such that the following holds. For any $w \geq 4$, and any sequence of n positive integers $a_1, a_2, \dots, a_n \in [w]$, there exist a partition*

$$[n] = T_1 \sqcup T_2 \sqcup \cdots \sqcup T_\ell, \ \ \ell \leq C_1 \log 2n \log \log w, \tag{1}$$

and ℓ positive integers $s_1, \dots, s_\ell$, where each T_j is further partitioned into p_j subsets,

$$T_j = B_{j,1} \sqcup B_{j,2} \sqcup \cdots \sqcup B_{j,p_j}, \tag{2}$$

such that for all j:

1. *For all $1 \leq k \leq p_j$, $\sum_{i \in B_{j,k}} a_i = s_j$.*
2. *For all $1 \leq k \leq p_j$, $|B_{j,k}| \leq C_2 \log w$.*
3. *For all $1 \leq m \leq n$, the number of $j \in [\ell]$ such that $p_j \geq m$ is at most $C_1 \log(2n/m) \log \log w$.*

We remark that the first two properties in Lemma 3.4 are more important. Item 3 only serves to shave a near-logarithmic factor in our later application.

Proof. Starting from all the indices $[n]$, we iteratively apply Lemma 3.3 to build the partition Eq. (1). Each application of Lemma 3.3 returns one set T_j together with its decomposition Eq. (2) which satisfies Item 1 and Item 2 by definition. After each application, we remove the already processed part T_j, and continue with the remaining indices, until all of $[n]$ have been partitioned.

By the guarantee of Lemma 3.3, each iteration removes a subset T_j which contains an $\Omega(1/(\log \log w))$ fraction of the remaining indices. Consequently, after $O(\log(2n/m) \log \log w)$ iterations, the number of remaining indices drops below m, and after that we must have $p_j \leq |T_j| < m$. This proves Item 3. In particular, the total number of iterations is $\ell = O(\log n \log \log w)$. □

4 The allocator

In this section, we present our allocator with polylogarithmic overhead, proving Theorem 1.1. By Observation 2.2, we model the memory as the real interval $[0, M)$ throughout this section.

Our main lemma is an allocator that achieves polylogarithmic overhead when the objects are not tiny.

Lemma 4.1. *Let $\ell \geq 2$ be an integer. In the real-interval setting with load factor $1 - \epsilon$ where all objects x have sizes $\mu(x) > 2^{-\ell} M$, there is a resizable allocator with (worst-case) expected overhead $O(\ell^3 (\log \epsilon^{-1})(\log \ell + \log \log \epsilon^{-1})^2)$.*

Combined with previously known results, Lemma 4.1 implies our main theorem.

Proof of Theorem 1.1 assuming Lemma 4.1. Kuszmaul [Kus23] gave a resizable allocator for objects of size $\le \epsilon^4 M$ with worst-case expected overhead $O(\log \epsilon^{-1})$. Farach-Colton, Kuszmaul, Sheffield, and Westover [FKSW24, Section 4.2] showed that, given another resizable allocator that handles objects of size larger than $\epsilon^4 M$ with worst-case expected overhead $f(\epsilon)$, one can combine it with Kuszmaul's allocator to obtain a resizable allocator that handles arbitrary object sizes and achieves worst-case expected overhead $O(f(\epsilon) + \log(\epsilon^{-1}))$. Since Lemma 4.1 achieves $f(\epsilon) \le O(\log^4 \epsilon^{-1} \cdot (\log\log \epsilon^{-1})^2)$ for $\ell = O(\log \epsilon^{-1})$, together they imply Theorem 1.1. □

The rest of this section is devoted to the proof of Lemma 4.1. As introduced in the overview section, the proof is a combination of our bundling idea based on [ES92] (see Section 3) and the substitution strategy with periodic rebuilds from [FKSW24] (as illustrated in the warm-up proof of Proposition 1.5). Additionally, we need to relax the bounded-ratio assumption of Proposition 1.5 in the same way as [FKSW24].

For simplicity, we assume the load factor is $1 - 2\epsilon$ instead of $1 - \epsilon$. We also assume ϵ^{-1} is an integer. Both assumptions can be justified by scaling ϵ by at most a constant factor.

We rescale the length of the real memory interval $[0, M)$ to $M := 2^\ell$. Then, the assumption in Lemma 4.1 implies that each object x has size $\mu(x) > 1$.

4.1 Basic definitions

Scales and inflation. A *scale* refers to an interval $(2^i, 2^{i+1}]$ of object sizes, where $i \in \mathbb{Z}$. For the purpose of Lemma 4.1, the relevant scales are $(1, 2], (2, 4], \ldots, (2^{\ell-1}, 2^\ell]$, i.e., $i \in \{0, 1, \ldots, \ell - 1\}$.

For an object x with size $\mu(x) = s \in (2^i, 2^{i+1}]$, we inflate its size to $\mu(x) := \lceil \frac{s}{\epsilon \cdot 2^{i+1}} \rceil \cdot \epsilon \cdot 2^{i+1}$. Since ϵ^{-1} is an integer, the inflated size is $\epsilon \cdot 2^{i+1}$ multiplied by an integer in $(\epsilon^{-1}/2, \epsilon^{-1}]$, and still lies in the interval $(2^i, 2^{i+1}]$. The inflation increases $\mu(x)$ by at most a factor of $(1 + 2\epsilon)$, so the new load factor becomes $(1 - 2\epsilon) \cdot (1 + 2\epsilon) < 1$, i.e., the total size of live objects remains smaller than M. The inflation affects the overhead factor by at most $1 + 2\epsilon = O(1)$ multiplicatively. We always use $\mu(x)$ to denote the size of x after inflation. Clearly, an allocation of the inflated objects automatically gives an allocation of the original objects.

After inflating the objects, our allocator maintains the prefix property, namely that all objects are stored without any gaps in between, starting from the left boundary of the memory. This property immediately implies that our allocator is resizable.

Bundles and types. Our allocator partitions the live objects into *bundles* of various *types*. The type of a bundle B is denoted by $\text{type}(B)$. Objects in the same bundle are always allocated contiguously in memory. Objects in the same bundle must have the same scale $(2^i, 2^{i+1}]$. Any two bundles B, B' of the same type must have equal size, $\sum_{y \in B} \mu(y) = \sum_{y \in B'} \mu(y)$. From time to time, the allocator may unbundle old bundles and form new bundles.

Given a subset Y of objects, we use the procedure CreateBundles(Y) (Algorithm 1) to partition Y into bundles, and assign newly created types to these bundles. This procedure invokes Lemma 3.4 for each scale $(2^i, 2^{i+1}]$ separately to bundle the objects of that scale in Y. In Lemma 3.4, each subset $B_{j,k} \subset T_j$ in Eq. (2) represents a bundle of objects, and T_j corresponds to the type of that bundle.

Algorithm 1: CreateBundles(Y), where Y is a set of objects

$\mathcal{B} \leftarrow \varnothing$ `// the collection of created bundles`
for $i \in \{0, 1, \ldots, \ell - 1\}$ **do**
  $Y_i \coloneqq \{y \in Y : \mu(y) \in (2^i, 2^{i+1}]\}$
  Let $Y_i = \{y_1, y_2, \ldots, y_{|Y_i|}\}$, and define $a_m \coloneqq \mu(y_m)/(\epsilon \cdot 2^{i+1})$.
  Apply Lemma 3.4 with $w \coloneqq \epsilon^{-1}$ to integers $a_1, \ldots, a_{|Y_i|} \in [w]$, and obtain the partition of $[|Y_i|]$ as in Eqs. (1) and (2)
  for *each* T_j *in Eq.* (1) **do**
    $\tau_{\text{new}} \leftarrow \tau_{\text{new}} + 1$ `// Create a new type indexed by` τ_{new} `(a global counter).`
    for *each* $B_{j,k} \subset T_j$ *in Eq.* (2) **do**
      Create a bundle $B \coloneqq \{y_m : m \in B_{j,k}\}$ with $\text{type}(B) \coloneqq \tau_{\text{new}}$
      $\mathcal{B} \leftarrow \mathcal{B} \sqcup \{B\}$.
return $\mathcal{B}$

Levels. The live objects allocated in memory are partitioned into ℓ *levels*, where each level consists of a contiguously allocated (possibly empty) set of bundles. The levels are indexed decreasingly by $\ell - 1, \ell - 2, \ldots, 1, 0$ from left to right.

We maintain that level j can only contain objects x with $\mu(x) \le 2^{j+1}$. In other words, an object of scale $(2^i, 2^{i+1}]$ can only be placed in levels $\ell - 1, \ell - 2, \ldots, i + 1, i$. In particular, an object of the smallest scale $(1, 2]$ may be placed in any level.

For a bundle type τ, we define its *leftmost level*, denoted $\text{leftlevel}(\tau)$, as the largest $j \le \ell - 1$ such that level j contains a bundle of type τ. (If no type-τ bundles currently exist in memory, then $\text{leftlevel}(\tau) \coloneqq -\infty$.)

4.2 Implementation

At the very beginning, run Initialize (Algorithm 2).

Insertion and deletion. Now we describe how our allocator handles updates (i.e., insertions and deletions). The implementations are given in Algorithm 3 and Algorithm 4, respectively. To slightly unify the descriptions of these two procedures, we introduce a global variable Y to represent the set of objects that are yet to be added to memory. More specifically:

- For insertion, we set Y to contain the current object being inserted.
- In the case of deleting x, whose scale is $(2^{i^*}, 2^{i^*+1}]$, we first perform the substitution strategy, so that the bundle B containing x now appears in level i^* (namely the rightmost level allowed to contain an object of this scale). Then, remove the entire bundle B from memory (creating a temporary gap in level i^*), and set $Y \leftarrow B \setminus \{x\}$, indicating that these objects should be moved back into memory later.

Both types of updates finish by invoking the Rebuild(j^*) procedure (Algorithm 5) for some $j^* \in \{0, 1, \ldots, \ell - 1\}$, whose behavior is to reorganize objects in levels $j^*, j^* - 1, \ldots, 1, 0$ together with the objects from Y, without modifying the levels to the left of j^*. Here, j^* is determined in a way so that levels further to the left are less frequently rebuilt, as follows: For each scale $(2^i, 2^{i+1}]$,

maintain a counter c_i, initialized uniformly at random from $\mathbb{Z} \cap [0, 2^{\ell-1-i})$.[6] At the end of each update of object scale $(2^{i^*}, 2^{i^*+1}]$, we first increment the counter c_{i^*} and then invoke REBUILD(j^*) for the largest integer $j^* \in [i^*, \ell-1]$ such that $2^{j^*-i^*}$ divides c_{i^*}.

Algorithm 2: INITIALIZE

```
for i ∈ {0, 1, ..., ℓ − 1} do
    c_i ← random integer from [0, 2^{ℓ−1−i}) // a global counter
τ_new ← 0 // a global counter
```

Algorithm 3: INSERT(x)

```
Suppose μ(x) ∈ (2^{i*}, 2^{i*+1}]
c_{i*} ← c_{i*} + 1
Y ← {x} // a global variable
REBUILD(j*), where j* is the largest integer j* ∈ [i*, ℓ − 1] such that 2^{j*−i*} divides c_{i*}
```

Algorithm 4: DELETE(x)

```
Suppose μ(x) ∈ (2^{i*}, 2^{i*+1}]
c_{i*} ← c_{i*} + 1
Let B be the bundle containing x
if B is not in level i* then
    // B must be in levels {ℓ − 1, ℓ − 2, ..., i* + 1}
    Let B' be a bundle in level i* such that type(B') = type(B) (which must exist by
      Corollary 4.4)
    Swap B and B' in memory
// Now, B is in level i*, and x is in B.
Remove B from memory
Y ← B \ {x} // a global variable
REBUILD(j*), where j* is the largest integer j* ∈ [i*, ℓ − 1] such that 2^{j*−i*} divides c_{i*}
```

Rebuild. During REBUILD(j^*) (Algorithm 5), we rebuild the levels $j^*, j^*-1, \ldots, 1, 0$ and also add the objects from Y into them, without modifying the remaining levels $\ell-1, \ell-2, \ldots, j^*+1$ on the left. For a bundle B in levels $\{j^*, j^*-1, \ldots, 1, 0\}$, if there is another bundle B' in levels $\{\ell-1, \ell-2, \ldots, j^*+1\}$ with $\text{type}(B') = \text{type}(B)$, then we keep B as an old bundle (since it may still be useful for performing the substitution strategy in the future); otherwise, we unbundle B and add its objects to Y. We partition the objects in Y into new bundles. We then place the bundles (both new and old) back into memory following a right-to-left greedy rule similar to that in the warm-up

[6] This is the only randomized part in our proof of Lemma 4.1. If we instead initialize $c_i \leftarrow 0$ for all i, then we would obtain a *deterministic* allocator for object sizes $[\text{poly}(\epsilon)M, M)$ with polylogarithmic *amortized* overhead. We do not know how to extend this amortized derandomization to all object sizes in $(0, M)$, since the tiny objects are handled by Kuszmaul's allocator [Kus23] which seems inherently randomized.

Algorithm 5: REBUILD(j^*) where $0 \le j^* \le \ell - 1$

$\mathcal{T}_{\text{old}} \coloneqq \{\text{type } \tau : \text{leftlevel}(\tau) > j^*\}$
$\mathcal{B} \leftarrow \varnothing$ `// a collection of bundles, initially empty`
for *each bundle* B *in levels* $\{j^*, j^* - 1, \dots, 0\}$ **do**
    Remove B from the memory
    if $\text{type}(B) \notin \mathcal{T}_{old}$ **then** $Y \leftarrow Y \sqcup B$ `// Unbundle` B `and add all its objects to` Y
    else $\mathcal{B} \leftarrow \mathcal{B} \sqcup \{B\}$ `//` B `remains as an old bundle, which we add to` $\mathcal{B}$
`// Now, the memory in levels` $\{j^*, j^* - 1, \dots, 0\}$ `has become empty.`
$\mathcal{B} \leftarrow \mathcal{B} \sqcup \text{CREATEBUNDLES}(Y)$, and then reset $Y \leftarrow \varnothing$.
for *each bundle type* τ **do**
    $\mathcal{B}_\tau \coloneqq \{B \in \mathcal{B} : \text{type}(B) = \tau\}$
    Let $(2^i, 2^{i+1}]$ be the scale of objects in type-τ bundles
    Define sequence $(n_i, n_{i+1}, \dots, n_{j^*})$ by $n_j = \begin{cases} 2^{\max\{j-i,1\}} & j \neq j^* \\ +\infty & j = j^* \end{cases}$
        `//` $(n_i, n_{i+1}, \dots, n_{j^*}) = (2, 2, 4, 8, 16, \dots, +\infty)$
    for $j \leftarrow i, i+1, \dots, j^*$ **do**
        If there are n unassigned bundles in $\mathcal{B}_\tau$, assign $\min\{n, n_j\}$ of them to level j in memory

proof of Proposition 1.5. Here, we assign each bundle to one of the levels in $\{j^*, j^* - 1, \dots, 1, 0\}$; bundles within the same level may be placed in arbitrary order.

We have the following basic observations:

Observation 4.2. *REBUILD(j^*) does not modify levels $\{\ell - 1, \ell - 2, \dots, j^* + 1\}$ in memory.*

Observation 4.3. *After every update (INSERT($\cdot$) or DELETE($\cdot$)) finishes, the memory satisfies the prefix property.*

Proof. Right before invoking REBUILD(j^*) in DELETE(x), the levels $\ell - 1, \ell - 2, \dots, i^* + 1$ satisfy the prefix property, because the deletion may only creates a gap in level i^*. Since $j^* \ge i^*$, after REBUILD(j^*), the memory satisfies the prefix property again.

For INSERT(x), the proof is straightforward. □

4.3 Invariants

Let $C_1 \ge 1, C_2 \ge 1$ be the constants from Lemma 3.4. Let C_3 be a constant (depending on C_1, C_2) such that

$$C_3 \ge 3C_1 \log(12C_2C_3). \tag{3}$$

Below, we state the three main invariants which the allocator satisfies at the end of every update (INSERT($\cdot$) or DELETE($\cdot$)). Property 1 ensures that there are few distinct types of bundles at any time. The remaining two invariants are analogous to those appearing in the warm-up proof of Proposition 1.5: Property 2 gives an upper bound on the total size of each level, and Property 3 guarantees enough supply of bundles of each type for performing the substitution strategy.

Property 1. *For every i, j with $\ell \geq j \geq i \geq 0$, the number of distinct types among bundles of scale $(2^i, 2^{i+1}]$ in levels $\{\ell - 1, \ell - 2, \ldots, j\}$ is at most*

$$C_3(\ell - j)(\log \ell + \log\log \epsilon^{-1})^2.$$

Property 2. *Let $s(i, j)$ denote the total size of objects of scale $(2^i, 2^{i+1}]$ that are in levels $\{j-1, j-2, \ldots, i\}$.*

Then, for every i, j with $\ell \geq j > i \geq 0$,

$$s(i, j) \leq 2^{j+1} \cdot C_2 \log \epsilon^{-1} \cdot C_3 \ell(\log \ell + \log\log \epsilon^{-1})^2 + (c_i \bmod 2^{j-i}) \cdot 2^{i+1}. \tag{4}$$

In particular,

$$s(i, j) \leq 2^j \cdot 3C_2C_3\ell(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2 - 2^{i+1}.$$

Property 3. *Let $v(\tau, j)$ denote the number of type-τ bundles in levels $\{j - 1, j - 2, \ldots, 0\}$.*

Then, for every bundle type τ with object scale $(2^i, 2^{i+1}]$, and every j such that leftlevel$(\tau) \geq j > i$,

$$v(\tau, j) \geq 2^{j-i} - (c_i \bmod 2^{j-i}) \geq 1. \tag{5}$$

Corollary 4.4. *At Algorithm 4 of* Delete(x) *(Algorithm 4), the substitute bundle B' always exists.*

Proof. Let $\tau = \text{type}(B)$, and recall that the scale of τ is $(2^{i^*}, 2^{i^*+1}]$. Since B can only be in levels $\{\ell - 1, \ell - 2, \ldots, i^* + 1\}$, we have leftlevel$(\tau) \geq i^* + 1$. Hence, by Property 3 with τ and $j = i^* + 1$, we have $v(\tau, i^* + 1) \geq 1$. Note that $v(\tau, i^* + 1)$ counts exactly the number of type-τ bundles in level i^*, since objects of scale $(2^{i^*}, 2^{i^*+1}]$ can only appear in levels $\{\ell - 1, \ell - 2, \ldots, i^*\}$. Therefore, there exists at least one type-τ bundle B' in level i^*. □

We use time step t to refer to the state at the end of the t-th update (Insert$(\cdot)$ or Delete$(\cdot)$) of the update sequence. Initially at time 0, the memory is empty and Properties 1 to 3 all hold vacuously. We now use induction on t to prove that these invariants hold at every time step $t \geq 1$.

Proof of Property 1. To prove Property 1 for i, j at any time step t, suppose the most recent execution of Rebuild(j^*) such that $j^* \geq j$ finished at time $t' \leq t$; if no such execution of Rebuild(j^*) has occurred, let $t' = 0$. Since the levels $\{\ell - 1, \ell - 2, \ldots, j\}$ have never been modified after time t', it suffices to prove the claimed upper bound at time t'. If $t' = 0$, then the bound immediately holds. Henceforth, assume $t' \geq 1$.

Focus on the execution of Rebuild(j^*) at time t'. We now separately bound the number of old types $\tau \in \mathcal{T}_{\text{old}}$ and new types $\tau \notin \mathcal{T}_{\text{old}}$ of scale $(2^i, 2^{i+1}]$, and add them up.

- Recall that τ is an old type if and only if leftlevel$(\tau) \geq j^* + 1$. Since Property 1 held at time step $t' - 1$ for the levels $\{\ell - 1, \ell - 2, \ldots, j^* + 1\}$, this implies that the number of old types of scale $(2^i, 2^{i+1}]$ is at most $C_3(\ell - j^* - 1)(\log \ell + \log\log \epsilon^{-1})^2$.
- It remains to bound the number of new types of scale $(2^i, 2^{i+1}]$ created by CreateBundles(Y). We focus on the subset $Y_i \subseteq Y$ containing objects of scale $(2^i, 2^{i+1}]$ only. By definition, all objects of Y_i were in levels $\{j^*, j^* - 1, \ldots, i\}$ at time $t' - 1$, with the possible exception of one additional object if the t'-th update is an insertion. Since Property 2 held for scale $(2^i, 2^{i+1}]$ and levels $\{j^*, j^* - 1, \ldots, i\}$ at time $t' - 1$, this implies that the objects in Y_i have total size at most

$$2^{j^*+1} \cdot 3C_2C_3\ell(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2.$$

Consequently,

$$|Y_i| \le 2^{j^*-i} \cdot 6C_2C_3\ell(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2.$$

For a new type τ, if a level $j \in [i, j^*]$ receives at least one type-τ bundle at the end of the **for** loop in Algorithm 5, then the number of type-τ bundles, $|\mathcal{B}_\tau|$, must be at least $1 + n_{j-1} + n_{j-2} + \cdots + n_i = \begin{cases} 1 + 2^{j-i} & i < j \\ 1 & i = j \end{cases} \ge 2^{j-i}$. On the other hand, by definition of CreateBundles(Y) and Lemma 3.4, Item 3, the number of types τ with at least 2^{j-i} bundles is at most

$$\begin{aligned}
&C_1 \log\big(2|Y_i|/2^{j-i}\big) \log\log \epsilon^{-1} \\
&\le C_1 \log\Big(2^{j^*-j} \cdot 12C_2C_3\ell(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2\Big) \log\log \epsilon^{-1} \\
&\le C_1(j^* - j + \log(12C_2C_3) + 3\log \ell + 3\log\log \epsilon^{-1}) \log\log \epsilon^{-1} \\
&\le C_3(j^* - j + 1)(\log \ell + \log\log \epsilon^{-1})^2. && \text{(by } C_3 \ge 3C_1 \log(12C_2C_3)\text{)}
\end{aligned}$$

Thus, the number of new types of scale $(2^i, 2^{i+1}]$ in level j is at most $C_3(j^* - j + 1)(\log \ell + \log\log \epsilon^{-1})^2$.

Summing up the counts of old and new types proves the claimed upper bound $C_3(\ell - j)(\log \ell + \log\log \epsilon^{-1})^2$. □

Proof of Property 2. To prove the claimed bound on $s(i, j)$ at any time step t, suppose the most recent execution of Rebuild(j^*) such that $j^* \ge j$ finished at time $t' \le t$; if no such execution of Rebuild(j^*) has occurred, let $t' = 0$. By definition, t' is no earlier than the most recent time when the counter c_i became divisible by 2^{j-i}. Hence, after time t', there have been at most $(c_i \bmod 2^{j-i})$ updates of object scale $(2^i, 2^{i+1}]$, where c_i denotes the counter at current time t.

Let $s'(i, j)$ denote the total size of objects of scale $(2^i, 2^{i+1}]$ in levels $\{j-1, j-2, \ldots, i\}$ at time t'. If $t' = 0$, then $s'(i, j) = 0$. Now we bound $s'(i, j)$ in the $t' \ge 1$ case. Focus on the execution of Rebuild(j^*) at time t'. By the placement rule at the end of Rebuild(j^*), since $j^* > j - 1$, each bundle type τ of object scale $(2^i, 2^{i+1}]$ contributes at most $n_i + n_{i+1} + \cdots + n_{j-1} = 2^{j-i}$ bundles to the levels $\{j-1, j-2, \ldots, i\}$. By Property 1, the number of distinct bundle types of scale $(2^i, 2^{i+1}]$ is at most $C_3\ell(\log \ell + \log\log \epsilon^{-1})^2$. By Lemma 3.4, every bundle created by CreateBundles($\cdot$) contains at most $C_2 \log \epsilon^{-1}$ objects. By multiplying these quantities together, we get the following upper bound:

$$s'(i, j) \le C_3\ell(\log \ell + \log\log \epsilon^{-1})^2 \cdot 2^{j-i} \cdot C_2 \log \epsilon^{-1} \cdot 2^{i+1}.$$

After time t', the levels $\{\ell-1, \ell-2, \ldots, j\}$ have never been modified; in particular, we have not moved any objects from levels $\{\ell-1, \ell-2, \ldots, j\}$ to levels $\{j-1, j-2, \ldots, 0\}$. Thus, the increase $s(i, j) - s'(i, j)$ can only be caused by the at most $(c_i \bmod 2^{j-i})$ objects of scale $(2^i, 2^{i+1}]$ inserted after time t'. Thus, $s(i, j) \le s'(i, j) + (c_i \bmod 2^{j-i}) \cdot 2^{i+1}$, finishing the proof of Property 2. □

Proof of Property 3. To prove the claimed bound on $v(\tau, j)$ at any time step t, suppose the most recent execution of Rebuild(j^*) such that $j^* \ge j$ finished at time $t' \le t$; if no such execution of Rebuild(j^*) has occurred, let $t' = 0$. Recall from the assumptions that type-τ bundles have objects of scale $(2^i, 2^{i+1}]$, where $i < j$. As in the proof of Property 2, we know that after time t' there have been at most $(c_i \bmod 2^{j-i})$ updates of object scale $(2^i, 2^{i+1}]$.

After time t', the levels $\{\ell-1, \ell-2, \ldots, j\}$ have never been modified. Hence, by our assumption that leftlevel$(\tau) \geq j$ holds at time t, we know leftlevel$(\tau) \geq j$ also held at time t'. In particular, this means $t' \geq 1$.

Let $v'(\tau, j)$ denote the number of type-τ bundles in levels $\{j-1, j-2, \ldots, 0\}$ at time t'. We now show that $v'(\tau, j) \geq 2^{j-i}$. To show this, focus on the execution of REBUILD(j^*) at time t', and separately analyze two cases depending on whether τ is a new type:

- Case $\tau \notin \mathcal{T}_{\text{old}}$ (i.e., τ is a new type created by CREATEBUNDLES):
  Suppose the **for** loop at Algorithm 5 put all the type-τ bundles in the levels $\{j_0, j_0-1, \ldots, i\}$ but not in level j_0+1. Then, leftlevel$(\tau) = j_0 \in [i, j^*]$. Since we assumed $j \leq$ leftlevel$(\tau) = j_0$, we must have $v'(\tau, j) = n_i + n_{i+1} + \cdots + n_{j-1} = 2^{j-i}$.
- Case $\tau \in \mathcal{T}_{\text{old}}$:
  We know leftlevel$(\tau) \geq j^* + 1$ by the definition of $\mathcal{T}_{\text{old}}$. By induction, we can assume as an inductive hypothesis that Property 3 held at time t' for $v'(\tau, \cdot)$ whenever the second argument is strictly larger than j. In particular, since leftlevel$(\tau) \geq j^* + 1 > j$, this implies
  $$v'(\tau, j^* + 1) \geq 2^{j^*+1-i} - (c'_i \bmod 2^{j^*+1-i}),$$
  where c'_i is the counter at time t'. By definition of j^* at time t', we know c'_i is divisible by 2^{j^*-i} but not by 2^{j^*-i+1}, so $c'_i \bmod 2^{j^*+1-i} = 2^{j^*-i}$. Hence $v'(\tau, j^* + 1) \geq 2^{j^*+1-i} - 2^{j^*-i} = 2^{j^*-i}$; in other words, in the execution of REBUILD(j^*), the number of old type-τ bundles in levels $\{j^*, j^*-1, \ldots, 0\}$ is at least $|\mathcal{B}_\tau| \geq 2^{j^*-i}$. Thus, by the placement rule at the end of REBUILD(j^*), the number of bundles assigned to levels $\{j-1, j-2, \ldots, i\}$ is $v'(\tau, j) = \min\{|\mathcal{B}_\tau|, n_i + n_{i+1} + \cdots + n_{j-1}\} = \min\{|\mathcal{B}_\tau|, 2^{j-i}\} \geq \min\{2^{j^*-i}, 2^{j-i}\} = 2^{j-i}$.

Thus, in both cases we have $v'(\tau, j) \geq 2^{j-i}$ as claimed.

After time t', the levels $\{\ell-1, \ell-2, \ldots, j\}$ have never been modified; in particular, we have not moved any objects from levels $\{j-1, j-2, \ldots, 0\}$ to levels $\{\ell-1, \ell-2, \ldots, j\}$. Thus, the decrease $v'(\tau, j) - v(\tau, j)$ can only be caused by the at most $(c_i \bmod 2^{j-i})$ objects of scale $(2^i, 2^{i+1}]$ deleted after time t', each of which may destroy one type-τ bundle. Hence, $v(\tau, j) \geq v'(\tau, j) - (c_i \bmod 2^{j-i})$, finishing the proof of Property 3. □

Therefore, our allocator is correct. Finally, we bound the worst-case expected overhead.

Proof of Lemma 4.1. By definition, REBUILD(j^*) only moves objects in levels $\{j^*, j^*-1, \ldots, 0\}$, whose total size can be bounded by summing up Property 2 for $j = j^* + 1$ over all scales $(2^i, 2^{i+1}]$ where $0 \leq i \leq j^*$, giving the upper bound $O(2^{j^*}\ell^2(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2)$. The total size of objects moved by INSERT$(\cdot)$ or DELETE$(\cdot)$ is dominated by that of REBUILD(j^*).

For every update of scale $(2^{i^*}, 2^{i^*+1}]$ in the update sequence, since we randomly initialized the counter c_{i^*}, the probability that it triggers REBUILD(j^*) equals $\begin{cases} 2^{i^*-j^*-1} & i^* \leq j^* < \ell - 1 \\ 2^{i^*-\ell+1} & j^* = \ell - 1 \end{cases} \leq 2^{i^*-j^*}$. Hence, the expected switching cost for this update is

$$\begin{aligned} &\sum_{i^* \leq j^* \leq \ell-1} 2^{i^*-j^*} \cdot O(2^{j^*}\ell^2(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2) \\ &= O(2^{i^*}\ell^3(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2). \end{aligned}$$

Dividing by the size $\Theta(2^{i^*})$ of the updated object gives the expected overhead bound $O(\ell^3(\log \epsilon^{-1})(\log \ell + \log\log \epsilon^{-1})^2)$ as claimed. □

5 Lower bounds

In this section, we prove Theorem 1.3 and Theorem 1.4. Both our lower bounds are proved with an oblivious adversary, i.e., our hard update sequences are fixed before starting the allocator, instead of being generated adaptively based on the current memory state.

Bounded ratio property. The object sizes in our hard update sequences are always in the interval $[\mu, C\mu]$ for some $\mu > 0$ and an absolute constant $C > 1$. This allows us to work with the more convenient notion of (unnormalized) *switching cost*, namely the total size of changed (moved/inserted/deleted) objects for handling an update x, and only lose a constant factor C when normalized by the size of x to get the overhead factor.

Full-memory assumption. In our proofs, we design hard instances in the original discrete setting (instead of the real-interval setting used in Section 4) with a memory of M slots indexed by $0, 1, \ldots, M-1$. Moreover, by Observation 2.1, Item 2, we are allowed to make *all the M memory slots fully occupied* in the hard instances. Formally, we actually prove the following two theorems which imply Theorem 1.3 and Theorem 1.4. Here, all hidden constants are absolute.

Theorem 5.1. *For every $M_0 \in \mathbb{Z}^+$, there exists an integer $M \in \Theta(M_0)$ such that any allocator that handles instances with M memory slots and object sizes $\Theta(M/\log M)$ must have worst-case expected switching cost $\Omega(M)$ (against an oblivious adversary).*

Theorem 5.2. *For every $M_0 \in \mathbb{Z}^+$, there exists an integer $M \in \Theta(M_0)$ such that any allocator that handles instances with M memory slots and object sizes $\Theta(M^{4/7})$ must have worst-case expected squared switching cost $\Omega(M^{9/7})$ (against an oblivious adversary).*

We suspect that the theorems above could be strengthened to hold for *all* $M \geq M_0$, but our current proofs do not seem to directly imply that.

We can immediately derive Theorem 1.3 from Theorem 5.1 as follows.

Proof of Theorem 1.3 assuming Theorem 5.1. Given $\epsilon > 0$ sufficiently small, pick an integer $M \in \Theta(1/\epsilon)$ so that $M \leq 1/(3\epsilon)$ and Theorem 5.1 holds for M. Since the object size in Theorem 5.1 is $\Theta(M/\log M)$, the required expected overhead factor is $\Omega(M)/\Theta(M/\log M) = \Omega(\log M) = \Omega(\log \epsilon^{-1})$. By Observation 2.1, Item 2 (which can apply to expected overhead as well), we conclude that any allocator that supports load factor $1 - \epsilon \geq 1 - \frac{1}{3M}$ must also incur expected overhead at least $\Omega(\log \epsilon^{-1})$. □

Analogously, Theorem 5.2 implies Theorem 1.4 (we omit the details).

Theorem 5.1 and Theorem 5.2 are proved in Section 5.2 and Section 5.3 respectively.

5.1 Basic definitions

We start by introducing a few definitions which will be used in both Section 5.2 and Section 5.3.

Size profile S and difference $\delta(S_1, S_2)$. In order to design hard input sequences of updates, we actually design hard sequences of *full-memory size profiles* (which could then be translated into sequences of updates).

A *full-memory size profile* (or *size profile* for short) is a multiset S of positive integers whose sum equals M, which correspond to the sizes of all live objects at a certain time step. (The labels of the objects are unimportant, since we do not have to distinguish two objects of the same size.) Define the *difference* between two multisets S_1, S_2 as $\delta(S_1, S_2) := \sum_x |m_{S_1}(x) - m_{S_2}(x)|$, where $m_S(x)$ denotes the multiplicity of x in the multiset S.

A sequence of size profiles $(S_1, S_2, \ldots, S_k)$ naturally induces a shortest update sequence that realizes them in order (starting from an empty memory), whose length is given by $\delta(\varnothing, S_1) + \sum_{i=1}^{k-1} \delta(S_i, S_{i+1})$.[7]

As an example, for two size profiles $S_1 = \{2, 3\}, S_2 = \{1, 2, 2\}$ with memory size $M = 5$, the sequence (S_1, S_2) can be realized by the following shortest update sequence starting from an empty memory, whose length is $\delta(\varnothing, S_1) + \delta(S_1, S_2) = 2 + 3 = 5$:

- Insert x with $\mu(x) = 2$.
- Insert y with $\mu(y) = 3$ (realizing size profile S_1).
- Delete y.
- Insert z with $\mu(z) = 1$.
- Insert w with $\mu(w) = 2$ (realizing size profile S_2).

Memory state ϕ, difference $\Delta(\phi, \phi')$, and maximal changed intervals. To analyze the necessary cost incurred by the allocator, we are primarily interested in the memory states (i.e., allocations) made by the allocator when the memory is full.

At full memory, the *memory state* ϕ can be uniquely described by the list of live object sizes ordered by their locations from left to right. For example, in the full memory state $\phi = (2, 5, 3, 1, 3)$ (with $M = 14$ and size profile $\{1, 2, 3, 3, 5\}$), the size-5 object has location 2, i.e., it is allocated to the memory slots with indices in the interval $[2, 2 + 5)$. See visualization in Fig. 2.

We now define the *difference* between two full memory states, which is a lower bound on the total switching cost required to transform one state to the other.

Recall that a size-μ object is said to have *location* p if it is allocated to the interval $[p, p + \mu)$.

Definition 5.3 ($\Delta(\phi, \phi')$)**.** Given two full memory states ϕ, ϕ', we say a size-μ object at location p in ϕ is *unchanged*, if ϕ' also has a size-μ object at location p; otherwise, we say the object is *changed*. Then, define the *difference* $\Delta(\phi, \phi')$ as the total size of changed objects in ϕ.

Observation 5.4. *For two full memory states ϕ, ϕ', the total switching cost to transform ϕ to ϕ' is at least $\Delta(\phi, \phi')$.*

We now define the useful notion of *maximal changed intervals*, which we borrow from the previous work [FKSW24].

Definition 5.5 (Maximal changed intervals)**.** In the same setup as Definition 5.3, we say $[L, R) \subseteq [0, M)$ is a *maximal changed interval* if it is an inclusion-maximal interval such that every object in ϕ intersecting this interval is changed.

[7] If there are multiple such shortest update sequences, we fix an arbitrary one of them.

By definition, $[0, M)$ is partitioned into the disjoint union of the unchanged objects and the maximal changed intervals. In particular, $\Delta(\phi, \phi')$ equals the total length of the maximal changed intervals.

We remark that the definitions of $\Delta(\phi, \phi')$, unchanged objects, and maximal changed intervals are symmetric with respect to ϕ and ϕ'.

See Fig. 2 for an example with visualization that illustrates the definitions above.

5.2 Logarithmic lower bound for expected overhead

In this section, we prove Theorem 5.1 (which implies Theorem 1.3).

We will define a hard distribution over sequences of full-memory size profiles. We first specify the memory size M, and the family of size profiles that we will use.

Let parameter k be a positive integer to be determined later.

Definition 5.6 (Size profile S_π)**.** For a permutation $\pi\colon [k] \to [k]$, define the size profile

$$S_\pi = \{2^{2k+1} + 2^{k+i} + 2^{\pi(i)} : i \in \{1, 2, \ldots, k\}\},$$

which consists of k distinct-sized objects of total size

$$M := k \cdot 2^{2k+1} + \sum_{i=1}^{k} 2^{k+i} + \sum_{i=1}^{k} 2^i = (k+1)2^{2k+1} - 2.$$

Note that all object sizes are in the interval $[2^{2k+1}, 2^{2k+2})$.

By the definition above, the number of memory slots is $M \in \Theta(k2^{2k})$. Hence, given $M_0 \geq 1$, we can choose some $k \in \frac{1}{2}(\log M_0 - \log\log M_0) + O(1)$ so that $M \in \Theta(M_0)$. Then, the object sizes are in $[\mu, 2\mu)$ where $\mu \in \Theta(M/\log M)$. Recall that our goal is to prove a worst-case expected switching cost lower bound of $\Omega(M)$ in this full-memory setting.

We have the following simple but crucial lemma:

Lemma 5.7. *For two permutations $\pi, \pi'\colon [k] \to [k]$, suppose two non-empty subsets $X \subseteq S_\pi, X' \subseteq S_{\pi'}$ have equal sum $\sum_{x\in X} x = \sum_{x'\in X'} x'$. Then, there is a set $J \subseteq [k]$ such that*

$$X = \{2^{2k+1} + 2^{k+i} + 2^{\pi(i)} : i \in J\}, \tag{6}$$

$$X' = \{2^{2k+1} + 2^{k+i} + 2^{\pi'(i)} : i \in J\}, \tag{7}$$

and

$$\{\pi(i) : i \in J\} = \{\pi'(i) : i \in J\}. \tag{8}$$

Proof. Consider the binary representation of $\sum_{x\in X} x$, restricted to the part between the $(k+1)$-st bit and the $2k$-th bit (inclusive; the i-th bit has binary weight 2^i). By definition of S_π, the positions of 1s among the bits in this part uniquely determine the set $J \subseteq [k]$ that satisfies Eq. (6), since there are no carries from the lower binary bits in the summation. Since this sum is the same as $\sum_{x'\in X'} x'$, we know that the same J should also satisfy Eq. (7).

Similarly, the part between the 1-st bit and the k-th bit in the binary representation of $\sum_{x\in X} x$ can uniquely determine the set $\{\pi(i) : i \in J\}$. Since this sum is the same as $\sum_{x'\in X'} x'$, we conclude that Eq. (8) must hold. □

Now we are ready to prove Theorem 5.1.

Proof of Theorem 5.1. We define a distribution over length-two sequences of size profiles, $(S_\iota, S_{\pi'})$, where $\iota\colon [k] \to [k]$ is the identity permutation $\iota(i) = i$, and let $\pi'\colon [k] \to [k]$ be modified from ι by swapping two random coordinates, that is, we independently pick two uniformly random *distinct* indices $a, b \in [k]$ and define

$$\pi'(i) \coloneqq \begin{cases} a & \text{if } i = b, \\ b & \text{if } i = a, \\ i & \text{otherwise.} \end{cases}$$

Note that $\delta(S_\iota, S_{\pi'}) = 4$, since the adversary can transform S_ι to $S_{\pi'}$ by first deleting the two objects of sizes

$$x \coloneqq 2^{2k+1} + 2^{k+a} + 2^a, y \coloneqq 2^{2k+1} + 2^{k+b} + 2^b$$

, and then inserting another two objects of sizes

$$x' \coloneqq 2^{2k+1} + 2^{k+b} + 2^a, y' \coloneqq 2^{2k+1} + 2^{k+a} + 2^b.$$

Initially, the size profile S_ι can be realized by inserting k objects into the empty memory.

We run the randomized allocator on the random update sequence induced by the sequence of size profiles $(S_\iota, S_{\pi'})$. We compare the full memory states ϕ_ι and $\phi_{\pi'}$ corresponding to the size profiles S_ι and $S_{\pi'}$ respectively.

Claim 5.8. *There is a maximal changed interval between ϕ_ι and $\phi_{\pi'}$ that contains both the size-x and size-y objects of ϕ_ι.*

Proof. The size-x and size-y objects appear in ϕ_ι but no longer appear in $\phi_{\pi'}$, so both of them are changed objects, and each should be in some maximal changed interval (Definition 5.5). Let $[L, R)$ be the maximal changed interval that contains the size-x object in ϕ_ι. This interval consists of a subset of changed objects in ϕ_ι including the size-x object. Hence, the interval length $R - L$ equals the sum of some subset $X \subseteq S_\iota$, where $X \ni x$. Similarly, this interval length $R - L$ also equals the sum of some subset $X' \subseteq S_{\pi'}$. Hence, $\sum_{x \in X} x = \sum_{x' \in X'} x'$.

We can now apply Lemma 5.7 to ι, π' and X, X', and obtain $J \subseteq [k]$. From $x \in X$ and Eq. (6), we get $a \in J$; in particular, $\iota(a) \in \{\iota(i) : i \in J\}$. Then, by Eq. (8), we have $\iota(a) \in \{\pi'(i) : i \in J\}$ as well. Since $\iota(a) = a = \pi'(b)$, this implies $\pi'(b) \in \{\pi'(i) : i \in J\}$, and thus $b \in J$. By $b \in J$ and Eq. (6), we get $y \in X$. Therefore, the size-y object is also contained in the interval $[L, R)$, which proves the claim. □

By the claim above, all objects in ϕ_ι located between the size-x and size-y objects (inclusive) are changed objects, whose sizes should contribute to $\Delta(\phi_\iota, \phi_{\pi'})$. Since $a \neq b \in [k]$ are uniformly randomly chosen by the adversary and are unknown to the allocator in advance, one can show that the expected number of objects in ϕ_ι located between the size-x and size-y objects (inclusive) equals $\frac{k+4}{3}$. Since all object sizes are in $[\mu, 2\mu] = [2^{2k+1}, 2^{2k+2}]$, we get that $\Delta(\phi_\iota, \phi_{\pi'})$ has expectation at least $\frac{k+4}{3}\mu$. Hence, by Observation 5.4, the expected total switching cost to transform ϕ_ι to $\phi_{\pi'}$ is at least $\frac{k+4}{3}\mu$. Since this transformation is completed within $\delta(S_\iota, S_{\pi'}) = 4$ updates, we know the worst-case expected switching cost achieved by the allocator must be at least $\frac{k+4}{3}\mu/\delta(S_\iota, S_{\pi'}) \geq \Omega(M)$. This finishes the proof of Theorem 5.1. □

5.3 Polynomial lower bound for expected squared overhead

In this section, we prove Theorem 5.2 (which implies Theorem 1.4). We described the intuition in the overview section (Section 1.2). Now we outline the structure of the formal proof:

- In Section 5.3.1, we construct the family $\{S_{i,j}\}$ of size profiles used in our proof. We establish the property of finger objects as mentioned in the overview section.
- In Section 5.3.2, we use $\{S_{i,j}\}$ to design hard update sequences. These sequences are organized as a tree $\mathcal{T}$, where each node corresponds to an $S_{i,j}$. The tree $\mathcal{T}$ has a spine and many branches. The "surprise inspection" mentioned in the overview intuitively corresponds to branching off from a random point on the spine.
- In Section 5.3.3, assuming a good randomized allocator $\mathcal{A}^{\text{rand}}$ exists, we use Yao's principle to fix a certain good deterministic allocator $\mathcal{A}$. Then, each node on the tree $\mathcal{T}$ is associated with a memory state, namely the state reached by running $\mathcal{A}$ on the update sequence induced by the root-to-node path on $\mathcal{T}$. From the performance of $\mathcal{A}$, we obtain certain inequalities involving the differences between these memory states.
- In Section 5.3.4, we use the properties of finger objects to show that these inequalities cannot hold, reaching a contradiction. Hence the purported allocator $\mathcal{A}^{\text{rand}}$ cannot exist, finishing the proof.

5.3.1 Size profiles and their properties

We construct the family of size profiles that we will use in our hard instances, and prove its main property in Lemma 5.14. The construction is based on two integer parameters $P > Q \geq 1$, which will be determined later.

We need a standard construction of several integers whose small-coefficient linear combinations are all distinct:

Lemma 5.9. *Given $n_1 \geq n_2 \geq n_3 \geq n_4 \geq 1$, there exist positive integers $a_1, a_2, a_3, a_4 \in \Theta(n_2 n_3 n_4)$, such that the integers $k_1 a_1 + k_2 a_2 + k_3 a_3 + k_4 a_4$ for $k_i \in \mathbb{Z} \cap [-n_i, n_i]$ are all distinct. Moreover, $a_1 < a_2 < a_3 < a_4 \leq 2a_1$.*

Proof. Let $n_i' \coloneqq 2n_i + 1$ for all $i \in [4]$. Let

$$\begin{aligned} a_1 &\coloneqq n_2' n_3' n_4' \\ a_2 &\coloneqq a_1 + n_3' n_4' \\ a_3 &\coloneqq a_2 + n_4' \\ a_4 &\coloneqq a_3 + 1, \end{aligned}$$

which clearly satisfy the "moreover" part of the lemma statement.

Suppose to the contrary that $k_1 a_1 + \cdots + k_4 a_4 = k_1' a_1 + \cdots + k_4' a_4$ for some $(k_1, \ldots, k_4) \neq (k_1', \ldots, k_4')$ where $k_i, k_i' \in \mathbb{Z} \cap [-n_i, n_i]$. Then, $m_1 a_1 + \cdots + m_4 a_4 = 0$, where $m_i = k_i - k_i' \in [-2n_i, 2n_i]$ and m_i are not all zero. Since $a_1 \equiv a_2 \equiv a_3 \equiv 0 \pmod{n_4'}$ and $a_4 \equiv 1 \pmod{n_4'}$, we obtain $0 = m_1 a_1 + \cdots + m_4 a_4 \equiv m_4 \pmod{n_4'}$. Since $|m_4| \leq 2n_4 = n_4' - 1$, we must have $m_4 = 0$.

Then, $m_1 a_1 + m_2 a_2 + m_3 a_3 = 0$. We divide both sides by n_4', and note that $\frac{a_1}{n_4'} \equiv \frac{a_2}{n_4'} \equiv 0 \pmod{n_3'}$, $\frac{a_3}{n_4'} \equiv 1 \pmod{n_3'}$. Using the same argument as the previous paragraph, we get $m_3 = 0$. Iterating the argument again gives $m_2 = 0$, at which point we conclude $m_1, \ldots, m_4$ must all be zero, a contradiction. Thus a_1, a_2, a_3, a_4 satisfy the desired conditions. □

Let $P > Q \geq 1$ be integer parameters to be determined later.

Use Lemma 5.9 to construct four positive integers a, b, c, d such that the integers

$$k_1 a + k_2 b + k_3 c + k_4 d \ : \ k_1, k_2 \in \mathbb{Z} \cap [-3P, 3P], k_3 \in \mathbb{Z} \cap [0, 2Q], k_4 \in \{0, 1\} \tag{9}$$

are all distinct, and

$$\mu \coloneqq a < b < c < d \leq 2\mu \tag{10}$$

where $\mu \in \Theta(PQ)$. We define the number of memory slots in our hard instance to be

$$M \coloneqq Pa + Qc + d \in [P\mu, 3P\mu] \subset \Theta(P^2 Q). \tag{11}$$

Definition 5.10 (Size profile $S_{i,j}$)**.** For integers $0 \leq i \leq P, 0 \leq j \leq Q$, define the full-memory size profile $S_{i,j}$ as the multiset consisting of:

- i size-a objects,
- j size-c objects,
- h size-b objects, where $h \coloneqq h(i,j) \coloneqq \lfloor (M - d - ia - jc)/b \rfloor$ (by Eq. (11), $0 \leq h \leq 3P$ must hold), and
- one remaining object (termed the *finger object*) of size $M - ia - jc - hb$.

The finger object size f in Definition 5.10 satisfies

$$f - d = (M - d - ia - jc) - \lfloor (M - d - ia - jc)/b \rfloor \cdot b \in [0, b),$$

that is, $f \in [d, d + b)$. Combining this bound with Eq. (10) yields the following basic observations:

Observation 5.11. *Every object in $S_{i,j}$ has size in $[\mu, 4\mu]$.*

Observation 5.12. *The finger object in $S_{i,j}$ has size different from a, b, and c (which justifies its distinguished role).*

We now describe the arithmetic structure of subset sums of object sizes in $S_{i,j}$:

Lemma 5.13. *Let X be any subset of the multiset $S_{i,j}$. Then, there exists a unique tuple (k_1, k_2, k_3, k_4) in the range $k_1 \in \mathbb{Z} \cap [0, 2P], k_2 \in \mathbb{Z} \cap [-3P, 3P], k_3 \in \mathbb{Z} \cap [0, 2Q], k_4 \in \{0, 1\}$, such that the sum of X equals $k_1 a + k_2 b + k_3 c + k_4 d$. Moreover, $k_4 = 1$ if and only if X contains the finger object of $S_{i,j}$.*

Proof. Given $X \subseteq S_{i,j}$, we first show that there exists *some* (k_1, k_2, k_3, k_4) from the specified range such that the sum of X equals $k_1 a + k_2 b + k_3 c + k_4 d$.

Recall from Definition 5.10 that the numbers of size-a, size-b, and size-c objects in $S_{i,j}$ are $i \leq P$, $h \leq 3P$, and $j \leq Q$ respectively. Denote the number of size-a, size-b, and size-c objects contained in $X \subseteq S_{i,j}$ by $m_1 \in [0, P], m_2 \in [0, 3P]$, and $m_3 \in [0, Q]$, respectively. Now consider two cases:

- If X does not contain the finger object of $S_{i,j}$, then the sum of X equals $m_1 a + m_2 b + m_3 c$, so $(k_1, k_2, k_3, k_4) \coloneqq (m_1, m_2, m_3, 0)$ satisfies the requirement and is in the specified range.
- Otherwise, X contains the finger object of size f. By Definition 5.10 and Eq. (11), $f = M - ia - jc - hb = (P - i)a - hb + (Q - j)c + d$. Thus, the sum of X equals $f + m_1 a + m_2 b + m_3 c = (P - i + m_1)a + (m_2 - h)b + (Q - j + m_3)c + d$, so $(k_1, k_2, k_3, k_4) \coloneqq (P - i + m_1, m_2 - h, Q - j + m_3, 1)$ satisfies the requirement and is in the specified range.

Hence, the desired (k_1, k_2, k_3, k_4) in the specified range always exists. The uniqueness of this tuple in this range then follows from the fact that all integers in Eq. (9) are distinct.

The "moreover" part of the lemma statement follows from the case distinction above and the uniqueness of the tuple (k_1, k_2, k_3, k_4). □

Lemma 5.13 implies the following crucial lemma, which is key to our proof:

Lemma 5.14. *Let ϕ, ϕ' be two memory states corresponding to two size profiles $S_{i,j}, S_{i',j'}$ respectively. Let $[L, R)$ be a maximal changed interval between ϕ, ϕ'.*

If $[L, R)$ in ϕ does not contain the finger object of $S_{i,j}$, then the multiset of object sizes in $[L, R)$ in ϕ' is the same as that in ϕ (in particular, $[L, R)$ in ϕ' does not contain the finger object of $S_{i',j'}$).

See Fig. 4 for an example. By taking the contrapositive and using the symmetry between ϕ and ϕ' in the statement above, we have the following corollary: $[L, R)$ in ϕ contains the finger object of $S_{i,j}$ if and only if $[L, R)$ in ϕ' contains the finger object of $S_{i',j'}$.

Proof of Lemma 5.14. Let $X \subseteq S_{i,j}$ and $X' \subseteq S_{i',j'}$ denote the multisets of sizes of the objects contained in $[L, R)$ in ϕ and ϕ' respectively. Then the sum of X and the sum of X' both equal $R - L$.

Suppose $[L, R)$ in ϕ does not contain the finger object. Then, $R - L = k_1 a + k_2 b + k_3 c + k_4 d$, where $k_4 = 0$, and $k_1 \le P, k_2 \le 3P, k_3 \le Q$ denote the number of size-a, size-b, size-c objects in X respectively. By the "moreover" part of Lemma 5.13 applied to X' and $k_4 = 0$, we conclude X' does not contain the finger object of $S_{i',j'}$. Therefore, $R - L = k_1' a + k_2' b + k_3' c + 0 \cdot d$ where $k_1' \le P, k_2' \le 3P, k_3' \le Q$ denote the number of size-a, size-b, size-c objects in X' respectively. By the uniqueness in Lemma 5.13, we must have $k_i = k_i'$ for all $i \in [3]$, i.e., the multisets X and X' are equal. □

5.3.2 Hard update sequences

We now proceed to the construction of hard sequences of size profiles.

Definition 5.15 (Tree $\mathcal{T}$ and sequence $\sigma_{i,j}$)**.** Define a rooted tree $\mathcal{T}$ with $(P+1)(Q+1)$ nodes uniquely labeled by the size profiles $S_{i,j}$ with $0 \le i \le P, 0 \le j \le Q$ (Definition 5.10), as follows (see an illustration in Fig. 5): The root node is $S_{0,Q}$. For each $1 \le i \le P$, the parent of $S_{i,Q}$ is $S_{i-1,Q}$. For each $0 \le i \le P$ and $0 \le j \le Q - 1$, the parent of $S_{i,j}$ is $S_{i,j+1}$.

Define $\sigma_{i,j}$ as the sequence of the size profiles corresponding to the path on $\mathcal{T}$ from the root to $S_{i,j}$, i.e.,

$$\sigma_{i,j} \coloneqq (S_{0,Q}, S_{1,Q}, \ldots, S_{i,Q}, S_{i,Q-1}, S_{i,Q-2}, \ldots, S_{i,j+1}, S_{i,j}).$$

Observe that the difference $\delta(\cdot, \cdot)$ between two adjacent size profiles on $\mathcal{T}$ is always $O(1)$:

Observation 5.16. *For all $0 \le i \le P, 1 \le j \le Q$, $\delta(S_{i,j}, S_{i,j-1}) \le 5$.*

For all $1 \le i \le P$, $\delta(S_{i,Q}, S_{i-1,Q}) \le 5$.

Proof. We prove the first claim only (the second claim follows from the same argument). By Definition 5.10, $S_{i,j}$ has one more size-c object than $S_{i,j-1}$, and the same number of size-a objects. The difference between the counts of size-b objects is

$$0 \le h(i, j-1) - h(i, j) = \lfloor (M - d - ia - jc + c)/b \rfloor - \lfloor (M - d - ia - jc)/b \rfloor \le 2,$$

where the last step is due to $c/b \le 2\mu/\mu = 2$. Finally, $S_{i,j}$ and $S_{i,j-1}$ each have a finger object of possibly different size. Summing these differences together gives $\delta(S_{i,j}, S_{i,j-1}) \le 1 + 2 + 2 = 5$. □

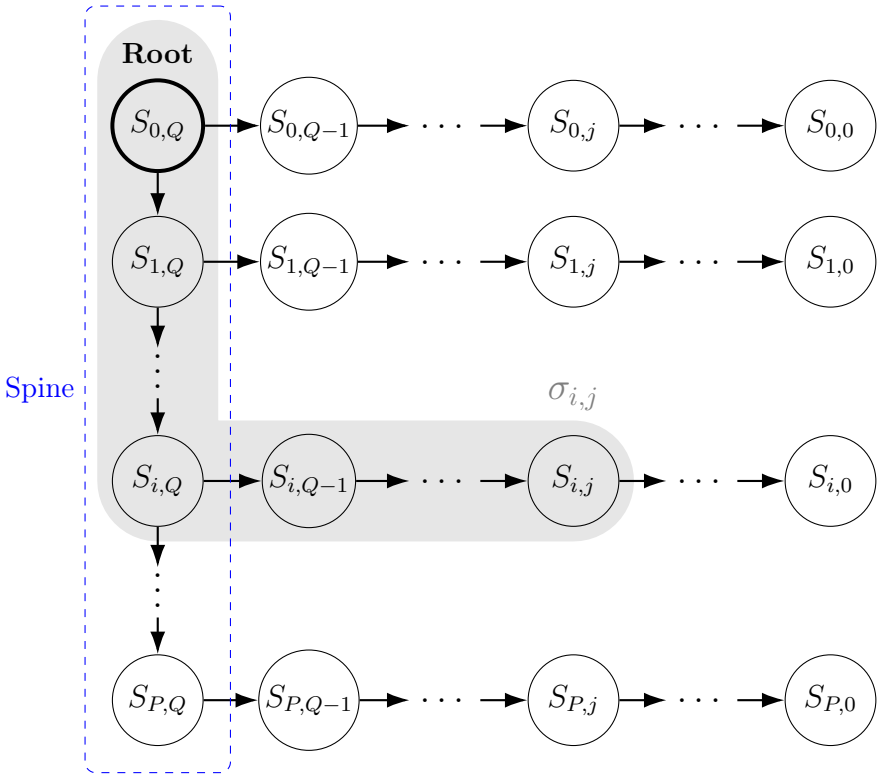

Figure 5: The rooted tree $\mathcal{T}$ defined in Definition 5.15. The sequence $\sigma_{i,j}$ (shaded in gray) is the tree path from the root node $S_{0,Q}$ to the node $S_{i,j}$. The tree has a *spine* (shown in blue dashed box) $(S_{0,Q}, S_{1,Q}, \ldots, S_{P,Q})$, and each node on the spine is the root of a *branch* $(S_{i,Q}, S_{i,Q-1}, \ldots, S_{i,0})$.

5.3.3 Fixing a deterministic allocator

We will use Yao's principle to find a good deterministic allocator, and then analyze its behavior. Before that, we make a few more definitions.

In a consecutive sequence of δ updates, if the allocator incurs switching costs $C_1, C_2, \ldots, C_\delta$, respectively, then we can relate the total squared switching cost to the total switching cost by Cauchy–Schwarz inequality:

$$(C_1 + \cdots + C_\delta)^2 \le \delta(C_1^2 + \cdots + C_\delta^2). \tag{12}$$

Definition 5.17 (Memory state $\phi_{i,j}$ and total squared switching cost $\kappa(\cdot,\cdot)$)**.** For a deterministic allocator $\mathcal{A}$, we use $\phi_{i,j}^{\mathcal{A}}$ to denote the memory state after running $\mathcal{A}$ on the update sequence induced by the size profile sequence $\sigma_{i,j}$ (which ends at $S_{i,j}$; see Definition 5.15).

If $S_{i',j'}$ is the parent of $S_{i,j}$ on the tree $\mathcal{T}$, then we define $\kappa^{\mathcal{A}}(\phi_{i',j'}^{\mathcal{A}}, \phi_{i,j}^{\mathcal{A}})$ to be the total squared switching cost incurred by $\mathcal{A}$ when transforming state $\phi_{i',j'}^{\mathcal{A}}$ to state $\phi_{i,j}^{\mathcal{A}}$.

Using Cauchy–Schwarz inequality (Eq. (12)) and Observation 5.4, we have

$$\Delta(\phi_{i',j'}^{\mathcal{A}}, \phi_{i,j}^{\mathcal{A}})^2 \le \delta(S_{i',j'}, S_{i,j}) \cdot \kappa^{\mathcal{A}}(\phi_{i',j'}^{\mathcal{A}}, \phi_{i,j}^{\mathcal{A}}). \tag{13}$$

Suppose there is a randomized allocator $\mathcal{A}^{\text{rand}}$ (i.e., a probability distribution over deterministic allocators $\mathcal{A}$) with worst-case expected squared switching cost at most $F^2\mu^2$ on each update. If $S_{i',j'}$ is the parent of $S_{i,j}$ on the tree $\mathcal{T}$, then running $\mathcal{A} \sim \mathcal{A}^{\text{rand}}$ on the deterministic update sequence induced by $\sigma_{i,j}$ yields

$$\mathop{\mathbf{E}}_{\mathcal{A}\sim\mathcal{A}^{\text{rand}}}[\kappa^{\mathcal{A}}(\phi_{i',j'}^{\mathcal{A}}, \phi_{i,j}^{\mathcal{A}})] \le \delta(S_{i',j'}, S_{i,j}) \cdot F^2\mu^2,$$

which, combined with Eq. (13), gives

$$\mathop{\mathbf{E}}_{\mathcal{A}\sim\mathcal{A}^{\text{rand}}}[\Delta(\phi_{i',j'}^{\mathcal{A}}, \phi_{i,j}^{\mathcal{A}})^2] \le \delta(S_{i',j'}, S_{i,j})^2 \cdot F^2\mu^2 \le 25F^2\mu^2, \tag{14}$$

where the last step is due to Observation 5.16.

Summing Eq. (14) along the spine of $\mathcal{T}$ gives

$$\mathop{\mathbf{E}}_{\mathcal{A}\sim\mathcal{A}^{\text{rand}}}\left[\sum_{i=1}^{P} \Delta(\phi_{i-1,Q}^{\mathcal{A}}, \phi_{i,Q}^{\mathcal{A}})^2\right] \le 25PF^2\mu^2. \tag{15}$$

Summing Eq. (14) over all edges on all the $(P+1)$ branches of $\mathcal{T}$ gives

$$\mathop{\mathbf{E}}_{\mathcal{A}\sim\mathcal{A}^{\text{rand}}}\left[\sum_{i=0}^{P}\sum_{j=0}^{Q-1} \Delta(\phi_{i,j+1}^{\mathcal{A}}, \phi_{i,j}^{\mathcal{A}})^2\right] \le 25(P+1)QF^2\mu^2. \tag{16}$$

By adding Eq. (15) and Q^{-1} times Eq. (16), applying the linearity of expectation, followed by averaging (Yao's principle), we conclude that there exists a deterministic allocator $\mathcal{A}$ such that

$$\sum_{i=1}^{P} \Delta(\phi_{i-1,Q}^{\mathcal{A}}, \phi_{i,Q}^{\mathcal{A}})^2 + \frac{1}{Q}\sum_{i=0}^{P}\sum_{j=0}^{Q-1} \Delta(\phi_{i,j+1}^{\mathcal{A}}, \phi_{i,j}^{\mathcal{A}})^2 \le 25(2P+1)F^2\mu^2 \le 100PF^2\mu^2.$$

In the following, we fix this deterministic allocator $\mathcal{A}$, and drop the superscript $\mathcal{A}$ for simplicity. In particular, the inequality above implies

$$\sum_{i=1}^{P} \Delta(\phi_{i-1,Q}, \phi_{i,Q})^2 \le 100PF^2\mu^2, \tag{17}$$

and

$$\sum_{i=0}^{P}\sum_{j=0}^{Q-1} \Delta(\phi_{i,j+1}, \phi_{i,j})^2 \le 100PQF^2\mu^2. \tag{18}$$

The rest of our proof amounts to showing that the above two inequalities would lead to a contradiction in a certain parameter regime. We summarize this as the following lemma:

Lemma 5.18. *Assume*

$$Q^2 \ge 5\sqrt{P}F, \tag{19}$$

$$\sqrt{P} \ge 800QF. \tag{20}$$

Then, there cannot exist memory states $\{\phi_{i,j}\}_{0\le i\le P, 0\le j\le Q}$ such that $\phi_{i,j}$ corresponds to the size profile $S_{i,j}$ and both Eqs. (17) and (18) hold.

We now quickly derive Theorem 5.2 from Lemma 5.18.

Proof of Theorem 5.2 assuming Lemma 5.18. Given $M_0 \ge 1$, we choose $F = \lceil M_0^{1/14} \rceil$. Then, we set the parameters $P > Q \ge 1$ (used for the definitions in Sections 5.3.1 and 5.3.2) as follows to satisfy Eq. (19) and Eq. (20):

$$Q \coloneqq 4000F^2, \tag{21}$$

$$P \coloneqq 160Q^3 = \Theta(F^6). \tag{22}$$

By Eq. (11), the number of memory slots in our construction is $M = \Theta(P^2Q) = \Theta(F^{14}) \in \Theta(M_0)$, and each object has size $\Theta(\mu) = \Theta(PQ) = \Theta(F^8) = \Theta(M^{4/7})$, as required.

Let $\mathcal{A}^{\text{rand}}$ be a randomized allocator for instances with M memory slots and object sizes $\Theta(M^{4/7})$. If it achieves worst-case expected squared switching cost at most $F^2\mu^2$, then by earlier discussions in Section 5.3.3, there exist memory states $\{\phi_{i,j}\}$ (corresponding to size profiles $S_{i,j}$) such that both Eqs. (17) and (18) hold, but this is impossible due to Lemma 5.18. Hence, $\mathcal{A}^{\text{rand}}$ must have worst-case expected squared switching cost larger than $F^2\mu^2 \ge \Omega(M^{9/7})$, as claimed. This finishes the proof of Theorem 5.2. □

5.3.4 Analysis

It remains to prove Lemma 5.18. We will assume the claimed memory states $\{\phi_{i,j}\}_{0\le i\le P, 0\le j\le Q}$ exist, and derive a contradiction. In doing so, we will crucially use the property of the finger object (Lemma 5.14).

Definition 5.19 ($p_{i,j}$)**.** Let $p_{i,j}$ denote the location of the finger object in the memory state $\phi_{i,j}$.

We first show that all the size-c objects in the memory state $\phi_{i,Q}$ should be close to the finger object:

Lemma 5.20. *Define a radius parameter*

$$\rho \coloneqq 10\sqrt{P}QF\mu. \tag{23}$$

Then, for all $0 \le i \le P$*, in the memory state* $\phi_{i,Q}$*, the interval* $[p_{i,Q} - \rho,\ p_{i,Q} + \rho)$ *must fully contain all the* Q *size-*c *objects.*

Proof. Compare the memory states $\phi_{i,Q}$ and $\phi_{i,0}$. Since $\phi_{i,0}$ contains no size-c objects, we know every size-c object in $\phi_{i,Q}$ must be contained in some maximal changed interval $[L_i, R_i)$ between these two states; moreover, since $[L_i, R_i)$ in $\phi_{i,0}$ contains no size-c objects, Lemma 5.14 implies that $[L_i, R_i)$ must contain the finger objects in both memory states. In other words, all the size-c objects of $\phi_{i,Q}$ are fully inside the maximal changed interval $[L, R)$ that contains the finger objects. Since $p_{i,Q} \in [L, R)$, it then suffices to show $R - L \le \rho$.

By Definition 5.5, $R - L \le \Delta(\phi_{i,Q}, \phi_{i,0})$. Then, a very crude application of Eq. (18) gives

$$\begin{aligned}
100PQF^2\mu^2 &\ge \sum_{j=0}^{Q-1} \Delta(\phi_{i,j+1}, \phi_{i,j})^2 && \text{(by Eq. (18))} \\
&\ge \frac{1}{Q}\Big(\sum_{j=0}^{Q-1} \Delta(\phi_{i,j+1}, \phi_{i,j})\Big)^2 && \text{(Cauchy–Schwarz)} \\
&\ge \frac{\Delta(\phi_{i,Q}, \phi_{i,0})^2}{Q} && \text{(triangle inequality)} \\
&\ge \frac{(R-L)^2}{Q}.
\end{aligned}$$

Therefore, $(R - L)^2 \le 100PQ^2F^2\mu^2 \le \rho^2$ as claimed. □

The following lemma intuitively says that it is expensive to move all of the size-c objects by a large total distance.

Lemma 5.21. *Let* λ_i *denote the sum of the locations of all the* Q *size-*c *objects in* $\phi_{i,Q}$*. Then, for every* $1 \le i \le P$*,*

$$|\lambda_{i-1} - \lambda_i| \le \Delta(\phi_{i-1,Q}, \phi_{i,Q})^2/\mu.$$

In particular, summing up this inequality and comparing with Eq. (17) *imply*

$$|\lambda_0 - \lambda_i| \le \sum_{i'=1}^{i} |\lambda_{i'-1} - \lambda_{i'}| \le 100PF^2\mu$$

for all $0 \le i \le P$*.*

Proof. Consider all the maximal changed intervals $[L_k, R_k)$ between $\phi_{i-1,Q}$ and $\phi_{i,Q}$. By Lemma 5.14, any $[L_k, R_k)$ *without* the finger objects must contain the same number of size-c objects in $\phi_{i-1,Q}$ and $\phi_{i,Q}$. Since $\phi_{i-1,Q}$ and $\phi_{i,Q}$ have the same number of size-c objects, by subtracting, we conclude that any $[L_k, R_k)$ (even if containing the finger objects) has the same number of size-c objects in both memory states.

Let $q_1 < q_2 < \cdots < q_Q$ and $q'_1 < q'_2 < \cdots < q'_Q$ denote the locations of size-c objects in $\phi_{i-1,Q}$ and $\phi_{i,Q}$, respectively. By the previous paragraph, the two objects at q_j in $\phi_{i-1,Q}$ and at q'_j in $\phi_{i,Q}$ either both belong to the same maximal changed interval, or are unchanged with $q_j = q'_j$.

We have

$$|\lambda_{i-1} - \lambda_i| = \left| \sum_{j=1}^{Q} q_j - \sum_{j=1}^{Q} q'_j \right| \le \sum_{j=1}^{Q} |q_j - q'_j|.$$

For each non-zero term $|q_j - q'_j|$, if the two corresponding objects are in $[L_k, R_k)$, we bound this term by $|q_j - q'_j| \le R_k - L_k$. Let m_k denote the number of size-c objects in $[L_k, R_k)$ in $\phi_{i-1,Q}$, which must satisfy $m_k \le (R_k - L_k)/c \le (R_k - L_k)/\mu$. Then, we obtain the upper bound

$$\sum_{j=1}^{Q} |q_j - q'_j| \le \sum_k m_k (R_k - L_k) \le \sum_k (R_k - L_k)^2/\mu.$$

Since the total length of maximal changed intervals equals the difference between two memory states (Definition 5.5), we have

$$\sum_k (R_k - L_k)^2 \le \Big(\sum_k (R_k - L_k) \Big)^2 = \Delta(\phi_{i-1,Q}, \phi_{i,Q})^2.$$

Chaining the three displayed inequalities finishes the proof. □

The previous two lemmas together imply that the finger object should always be confined to a small interval:

Lemma 5.22. *For every* $0 \le i \le P$, $|p_{0,Q} - p_{i,Q}| \le 4\rho$.

Proof. By Lemma 5.20, all size-c objects in $\phi_{i,Q}$ are fully inside $[p_{i,Q} - \rho, p_{i,Q} + \rho)$. Similarly, all size-$c$ objects in $\phi_{0,Q}$ are fully inside $[p_{0,Q} - \rho, p_{0,Q} + \rho)$.

Suppose to the contrary that $|p_{0,Q} - p_{i,Q}| > 4\rho$; here, assume $p_{0,Q} - p_{i,Q} > 4\rho$ without loss of generality (the other case follows similarly). Then, for any two size-c objects in $\phi_{0,Q}$ and in $\phi_{i,Q}$, at locations q and q' respectively, it holds that $q - q' \ge (p_{0,Q} - \rho) - (p_{i,Q} + \rho) > 2\rho$. Thus,

$$\lambda_0 - \lambda_i > Q \cdot 2\rho = 80\sqrt{P}Q^2 F\mu.$$

On the other hand, Lemma 5.21 states that

$$|\lambda_0 - \lambda_i| \le 100PF^2\mu.$$

Together, we get $20\sqrt{P}Q^2F\mu < 100PF^2\mu$, which contradicts our assumption Eq. (19) that $Q^2 \ge 5\sqrt{P}F$. Hence, we must have $|p_{0,Q} - p_{i,Q}| \le 4\rho$. □

Using Lemma 5.22, the final plan is to show that it is expensive to insert all the P size-a objects, as a large fraction of them will end up far from the finger object. To formalize this intuition, we need to define a suitable potential for the size-a objects, and show that the total potential cannot change significantly.

Definition 5.23 ($\Phi(\cdot)$)**.** Define interval $[L^*, R^*) \coloneqq [p_{0,Q} - 8\rho,\ p_{0,Q} + 8\rho)$. If a size-$a$ object has location q, we say its *potential* is

$$\Phi(q) \coloneqq \begin{cases} q - R^* & \text{if } q > R^*, \\ L^* - q & \text{if } q \le L^*, \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\Phi(q) \ge 0$, and

$$|\Phi(q) - \Phi(q')| \le |q - q'|. \tag{24}$$

Lemma 5.24. *For $0 \le i \le P$, let η_i denote the sum of potentials $\Phi(\cdot)$ of all the size-a objects in $\phi_{i,Q}$. Then, $|\eta_{i-1} - \eta_i| \le 4\Delta(\phi_{i-1,Q}, \phi_{i,Q})^2/\mu$.*

Proof. Consider all the maximal changed intervals $[L_k, R_k)$ between $\phi_{i-1,Q}$ and $\phi_{i,Q}$. By Lemma 5.14, any $[L_k, R_k)$ *without* the finger objects must contain the same number of size-a objects in $\phi_{i-1,Q}$ and $\phi_{i,Q}$. Let $[L_f, R_f)$ be the maximal changed interval that contains the finger objects in $\phi_{i-1,Q}$ and $\phi_{i,Q}$, whose locations are $p_{i-1,Q}$ and $p_{i,Q}$ respectively. Since $\phi_{i,Q}$ has one more size-a object than $\phi_{i-1,Q}$, we know $[L_f, R_f)$ contains one more size-a object in $\phi_{i,Q}$ than in $\phi_{i-1,Q}$.

Let $q_1 < q_2 < \cdots < q_r$ and $q'_1 < q'_2 < \cdots < q'_r$ denote the locations of size-a objects that are outside $[L_f, R_f)$ in $\phi_{i-1,Q}$ and $\phi_{i,Q}$, respectively. Let $q_{r+1} < q_{r+2} < \cdots < q_{i-1}$ and $q'_{r+1} < q'_{r+2} < \cdots < q'_i$ denote the locations of size-a objects inside $[L_f, R_f)$ in $\phi_{i-1,Q}$ and $\phi_{i,Q}$, respectively. By the previous paragraph, for $j \in [r]$, the two objects at q_j and at q'_j either both belong to the same maximal changed interval $[L_k, R_k)$ (for some $k \ne f$), or are unchanged with $q_j = q'_j$.

To bound $|\eta_{i-1} - \eta_i|$, we write

$$\begin{aligned} |\eta_{i-1} - \eta_i| = \left| \sum_{j=1}^{i-1} \Phi(q_j) - \sum_{j=1}^{i} \Phi(q'_j) \right| &\le \sum_{j=1}^{r} |\Phi(q_j) - \Phi(q'_j)| + \sum_{j=r+1}^{i-1} \Phi(q_j) + \sum_{j=r+1}^{i} \Phi(q'_j), \\ &\le \sum_{j=1}^{r} |q_j - q'_j| + \sum_{j=r+1}^{i-1} \Phi(q_j) + \sum_{j=r+1}^{i} \Phi(q'_j), \end{aligned} \tag{25}$$

where we used the non-negativity of $\Phi(\cdot)$ and Eq. (24).

To bound the first sum in Eq. (25), we proceed in the same way as in the proof of Lemma 5.21. For each non-zero term $|q_j - q'_j|$, if the two corresponding objects are in $[L_k, R_k)$, we bound this term by $|q_j - q'_j| \le R_k - L_k$. Let m_k denote the number of size-a objects in $[L_k, R_k)$ in $\phi_{i-1,Q}$, which must satisfy $m_k \le (R_k - L_k)/a \le (R_k - L_k)/\mu$. Then, we obtain the upper bound

$$\sum_{j=1}^{r} |q_j - q'_j| \le \sum_{k \ne f} m_k (R_k - L_k) \le \sum_{k \ne f} (R_k - L_k)^2/\mu.$$

Now, we bound the last two sums in Eq. (25). We only need to consider the case where they contain at least one positive term, i.e., there exists some $q^* \in \{q_{r+1}, \ldots, q_{i-1}\} \cup \{q'_{r+1}, \ldots, q'_i\}$ such that $\Phi(q^*) > 0$. By definition of $\Phi(\cdot)$, this means $q^* \notin [L^*, R^*) = [p_{0,Q} - 8\rho,\ p_{0,Q} + 8\rho)$, so $|q^* - p_{0,Q}| \ge 8\rho$. Since both q^* and $p_{i,Q}$ are in the interval $[L_f, R_f)$, we have

$$R_f - L_f \ge |q^* - p_{i,Q}| \ge |q^* - p_{0,Q}| - |p_{i,Q} - p_{0,Q}| \ge 8\rho - 4\rho = 4\rho, \tag{26}$$

where in the last inequality we used Lemma 5.22. On the other hand, for any $q \in \{q_{r+1}, \dots, q_{i-1}\} \cup \{q'_{r+1}, \dots, q'_i\} \subset [L_f, R_f)$, we have

$$\begin{aligned}
\Phi(q) &\le |q - p_{0,Q}| + \Phi(p_{0,Q}) && \text{(by Eq. (24))} \\
&= |q - p_{0,Q}| && \text{(by definition of } \Phi(\cdot)) \\
&\le |q - p_{i,Q}| + |p_{i,Q} - p_{0,Q}| \\
&\le (R_f - L_f) + 4\rho && \text{(by } q, p_{i,Q} \in [L_f, R_f), \text{ and Lemma 5.22)} \\
&\le 2(R_f - L_f). && \text{(by Eq. (26))}
\end{aligned}$$

Since there are at most $(R_f - L_f)/a$ size-a objects in $[L_f, R_f)$ in $\phi_{i-1,Q}$ (or, in $\phi_{i,Q}$), we conclude that the second sum and the third sum of Eq. (25) each can be upper-bounded by $2(R_f - L_f) \cdot (R_f - L_f)/a \le 2(R_f - L_f)^2/\mu$.

Summing up, Eq. (25) can be bounded as

$$\begin{aligned}
|\eta_{i-1} - \eta_i| &\le \sum_{k \ne f} (R_k - L_k)^2/\mu + 2(R_f - L_f)^2/\mu + 2(R_f - L_f)^2/\mu \\
&\le 4 \sum_k (R_k - L_k)^2/\mu \\
&\le 4\Big(\sum_k (R_k - L_k)\Big)^2/\mu \\
&= 4\Delta(\phi_{i-1,Q}, \phi_{i,Q})^2/\mu
\end{aligned}$$

as claimed. □

Now we are ready to finish the proof.

Proof of Lemma 5.18. Recall η_i is the sum of potentials $\Phi(\cdot)$ of all the size-a objects in $\phi_{i,Q}$. We have

$$\begin{aligned}
\eta_P &\le \eta_0 + \sum_{i=1}^{P} |\eta_{i-1} - \eta_i| \\
&\le \eta_0 + \sum_{i=1}^{P} 4\Delta(\phi_{i-1,Q}, \phi_{i,Q})^2/\mu && \text{(by Lemma 5.24)} \\
&\le \eta_0 + 400PF^2\mu && \text{(by Eq. (17))} \\
&= 400PF^2\mu. && (\phi_{0,Q} \text{ contains no size-}a \text{ objects})
\end{aligned}$$

By definition of $\Phi(\cdot)$, if a size-a object is not fully contained in the interval $[L^* - 10\rho, R^* + 10\rho + a) = [p_{0,Q} - 18\rho,\ p_{0,Q} + 18\rho + a)$, then its potential is at least 10ρ. The number of size-a objects in $\phi_{P,Q}$ that are fully contained in this interval is at most

$$(36\rho + a)/a \le 36\rho/\mu + 1 \underbrace{\le}_{\text{by Eq. (23)}} 400\sqrt{P}QF \underbrace{\le}_{\text{by Eq. (20)}} P/2.$$

Therefore, among the P size-a objects in $\phi_{P,Q}$, at least $P - P/2 = P/2$ of them have potential at least 10ρ, giving $\eta_P \ge (P/2) \cdot (10\rho) = 50P^{1.5}QF\mu$. By comparing with the upper bound

$\eta_P \le 400PF^2\mu$ from the previous paragraph, we get $\sqrt{P}Q \le 8F$. In particular, $Q \le 8F$, but this contradicts $Q = 4000F^2$ by Eq. (21) and $F \ge 1$. This finishes the proof of Lemma 5.18 (which implies Theorem 5.2 and Theorem 1.4). □

6 Conclusion and open questions

Our upper bound result demonstrates the surprising power of the sunflower lemma for the Memory Reallocation problem. Our lower bound results rely on the additive structure of the object sizes in the designed hard instances. In general, we believe that applying additive combinatorial techniques to various resource allocation and scheduling problems is a fruitful direction for future research.

We conclude with several concrete open questions:

Closing the gap. One of the main open questions is to narrow the gap between our $O(\log^4 \epsilon^{-1} \cdot (\log\log \epsilon^{-1})^2)$ upper bound and the $\Omega(\log \epsilon^{-1})$ lower bound for the expected reallocation overhead. We conjecture that the lower bound is closer to the truth.

Time complexity. Kuszmaul [Kus23] provided a time-efficient implementation of his allocator for tiny objects. It would be interesting to make our allocator time-efficient as well.

A main obstacle towards time-efficiency is the lack of a fast algorithmic version of Lemma 3.2, which was proved via Erdős and Sárközy's non-constructive argument based on the sunflower lemma. As a preliminary attempt, one could compute the construction of Lemma 3.2 (with slightly worse logarithmic factors) in $\mathrm{poly}(n, w)$ time, by combining Erdős–Rado's proof of the sunflower lemma with the standard dynamic programming algorithm for the counting version of the Subset Sum problem. However, this is too slow for our application, where w is as large as ϵ^{-1}.

After completing this paper, we became aware of a recent paper by Chen, Mao, and Zhang [CMZ26], which, despite the different motivation, arrived at the same question and made significant progress. More specifically, [CMZ26, Theorem 1.6] implies that, for any $\delta \in (0, 1)$, the construction of our Lemma 3.2 can be implemented in $\widetilde{O}(n+w^\delta)$ time, at the cost of worsening all the $\log w$ factors in the lemma statement to $(\log w)^{O(1/\delta)}$. This suggests the possibility of designing an allocator with a time complexity overhead factor of $\exp(O(\sqrt{\log \epsilon^{-1} \log\log \epsilon^{-1}}))$.

High-probability guarantee. Although polylogarithmic overhead with high probability is impossible in general (by Theorem 1.4), it is still achievable in some interesting special cases. For example, one can show that Kuszmaul's allocator [Kus23] for power-of-two sizes actually achieves $O(\log(1/\epsilon))$ overhead with high probability in $1/\epsilon$. We conjecture that high-probability polylogarithmic overhead is also achievable for tiny object sizes (for example, less than $\epsilon^4 M$).

Another interesting direction is to design allocators achieving $O(1/\epsilon^\alpha)$ overhead with high probability in $1/\epsilon$ (or even deterministically) with small exponent α. Our lower bound (Theorem 1.4) implies $\alpha \ge 1/14$, but the best known upper bound is only $\alpha = 1$, achieved by the folklore deterministic allocator.

Other settings. It would also be interesting to further investigate Memory Reallocation in the cost-oblivious setting of [BFF+17], or the request fragmentation setting of [BCF+25].

Acknowledgements

I would like to thank Nathan Sheffield and Alek Westover for inspiring discussions that sparked my interest in this problem. Additionally, I thank Shyan Akmal, Lin Chen, Hongbo Kang, Tsvi Kopelowitz, Mingmou Liu, Jelani Nelson, Kewen Wu, and Renfei Zhou for helpful discussions, and Nicole Wein for sharing a manuscript of [BCF+25].

References

[AB87] Noga Alon and Ravi B. Boppana. The monotone circuit complexity of Boolean functions. *Combinatorica*, 7(1):1–22, 1987. doi:10.1007/BF02579196. 11

[AF88] Noga Alon and Gregory Freiman. On sums of subsets of a set of integers. *Combinatorica*, 8(4):297–306, 1988. doi:10.1007/BF02189086. 11

[ALWZ20] Ryan Alweiss, Shachar Lovett, Kewen Wu, and Jiapeng Zhang. Improved bounds for the sunflower lemma. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 624–630. ACM, 2020. doi:10.1145/3357713.3384234. 11

[ALWZ21] Ryan Alweiss, Shachar Lovett, Kewen Wu, and Jiapeng Zhang. Improved bounds for the sunflower lemma. *Ann. of Math. (2)*, 194(3):795–815, 2021. doi:10.4007/annals.2021.194.3.5. 11, 13

[AW21] Shyan Akmal and Ryan Williams. MAJORITY-3SAT (and related problems) in polynomial time. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1033–1043. IEEE, 2021. doi:10.1109/FOCS52979.2021.00103. 11

[BCF+25] Michael Bender, Alexander Conway, Martín Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. Tight bounds for memory allocation with and without request fragmentation, 2025. Manuscript. 2, 11, 37, 38

[BCW21] Tolson Bell, Suchakree Chueluecha, and Lutz Warnke. Note on sunflowers. *Discrete Math.*, 344(7):Paper No. 112367, 3, 2021. doi:10.1016/j.disc.2021.112367. 11, 13

[BFF+15a] Michael A. Bender, Martín Farach-Colton, Sándor P. Fekete, Jeremy T. Fineman, and Seth Gilbert. Cost-oblivious reallocation for scheduling and planning. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 143–154. ACM, 2015. doi:10.1145/2755573.2755589. 2

[BFF+15b] Michael A. Bender, Martín Farach-Colton, Sándor P. Fekete, Jeremy T. Fineman, and Seth Gilbert. Reallocation problems in scheduling. *Algorithmica*, 73(2):389–409, 2015. doi:10.1007/S00453-014-9930-4. 2

[BFF+17] Michael A. Bender, Martín Farach-Colton, Sándor P. Fekete, Jeremy T. Fineman, and Seth Gilbert. Cost-oblivious storage reallocation. *ACM Trans. Algorithms*, 13(3):38:1–38:20, 2017. doi:10.1145/3070693. 2, 10, 37

[BKP+22] Aaron Berger, William Kuszmaul, Adam Polak, Jonathan Tidor, and Nicole Wein. Memoryless worker-task assignment with polylogarithmic switching cost. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 19:1–19:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ICALP.2022.19. 11

[BM25] Jarosław Błasiok and Linus Meierhöfer. Hardness of clique approximation for monotone circuits. In *40th Computational Complexity Conference, CCC 2025, August 5-8, 2025, Toronto, Canada*, volume 339 of *LIPIcs*, pages 4:1–4:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.CCC.2025.4. 11

[Bri24] Karl Bringmann. Knapsack with small items in near-quadratic time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 259–270. ACM, 2024. doi:10.1145/3618260.3649719. 11

[BW21] Karl Bringmann and Philip Wellnitz. On near-linear-time algorithms for dense subset sum. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 1777–1796. SIAM, 2021. doi:10.1137/1.9781611976465.107. 11

[CFP21] David Conlon, Jacob Fox, and Huy Tuan Pham. Subset sums, completeness and colorings. *arXiv preprint arXiv:2104.14766*, 2021. arXiv:2104.14766. 11

[CGR+25] Bruno Cavalar, Mika Göös, Artur Riazanov, Anastasia Sofronova, and Dmitry Sokolov. Monotone circuit complexity of matching. *arXiv preprint arXiv:2507.16105*, 2025. arXiv:2507.16105. 11

[CKR22] Bruno Pasqualotto Cavalar, Mrinal Kumar, and Benjamin Rossman. Monotone circuit lower bounds from robust sunflowers. *Algorithmica*, 84(12):3655–3685, 2022. doi:10.1007/S00453-022-01000-3. 11

[CLMZ24a] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. Approximating partition in near-linear time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 307–318. ACM, 2024. doi:10.1145/3618260.3649727. 11

[CLMZ24b] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. Faster algorithms for bounded knapsack and bounded subset sum via fine-grained proximity results. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*, pages 4828–4848. SIAM, 2024. doi:10.1137/1.9781611977912.171. 11

[CLMZ24c] Lin Chen, Jiayi Lian, Yuchen Mao, and Guochuan Zhang. An improved pseudopolynomial time algorithm for subset sum. In *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, pages 2202–2216. IEEE, 2024. doi:10.1109/FOCS61266.2024.00129. 11

[CMZ25] Lin Chen, Yuchen Mao, and Guochuan Zhang. Long arithmetic progressions in sumsets and subset sums: Constructive proofs and efficient witnesses. In *Proceedings of the 57th*

Annual ACM Symposium on Theory of Computing, STOC 2025, Prague, Czechia, June 23-27, 2025, pages 2086–2097. ACM, 2025. doi:10.1145/3717823.3718281. 11

[CMZ26] Lin Chen, Yuchen Mao, and Guochuan Zhang. Long arithmetic progressions in sparse subset sums: A computational perspective. In *Proceedings of the 2026 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3611–3638. SIAM, 2026. doi:10.1137/1.9781611978971.132. 11, 37

[dRV25] Susanna F. de Rezende and Marc Vinyals. Lifting with colourful sunflowers. In *40th Computational Complexity Conference, CCC 2025, August 5-8, 2025, Toronto, Canada*, volume 339 of *LIPIcs*, pages 36:1–36:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICS.CCC.2025.36. 11

[ER60] P. Erdős and R. Rado. Intersection theorems for systems of sets. *J. London Math. Soc.*, 35:85–90, 1960. doi:10.1112/jlms/s1-35.1.85. 3, 11, 13

[ES92] P. Erdős and A. Sárközy. Arithmetic progressions in subset sums. *Discrete Math.*, 102(3):249–264, 1992. doi:10.1016/0012-365X(92)90119-Z. 3, 5, 11, 13, 14, 16

[FKNP21] Keith Frankston, Jeff Kahn, Bhargav Narayanan, and Jinyoung Park. Thresholds versus fractional expectation-thresholds. *Ann. of Math. (2)*, 194(2):475–495, 2021. doi:10.4007/annals.2021.194.2.2. 11, 13

[FKSW24] Martín Farach-Colton, William Kuszmaul, Nathan S. Sheffield, and Alek Westover. A nearly quadratic improvement for memory reallocation. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2024, Nantes, France, June 17-21, 2024*, pages 125–135. ACM, 2024. doi:10.1145/3626183.3659965. 2, 3, 4, 5, 6, 7, 8, 10, 13, 16, 24

[FMS97] Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *J. ACM*, 44(2):257–271, 1997. doi:10.1145/256303.256309. 11

[Fre93] Gregory A. Freiman. New analytical results in subset-sum problem. *Discrete Math.*, 114(1-3):205–217, 1993. doi:10.1016/0012-365X(93)90367-3. 11

[GM91] Zvi Galil and Oded Margalit. An almost linear-time algorithm for the dense subset-sum problem. *SIAM J. Comput.*, 20(6):1157–1189, 1991. doi:10.1137/0220072. 11

[GM07] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. *Theor. Comput. Sci.*, 379(3):405–417, 2007. doi:10.1016/J.TCS.2007.02.047. 11

[GMR13] Parikshit Gopalan, Raghu Meka, and Omer Reingold. DNF sparsification and a faster deterministic counting algorithm. *Comput. Complex.*, 22(2):275–310, 2013. doi:10.1007/S00037-013-0068-6. 11

[HP04] Nicholas G. Hall and Chris N. Potts. Rescheduling for new orders. *Oper. Res.*, 52(3):440–453, 2004. doi:10.1287/OPRE.1030.0101. 2

[Jin24] Ce Jin. 0-1 knapsack in nearly quadratic time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 271–282. ACM, 2024. doi:10.1145/3618260.3649618. 11

[JSS21] Vishesh Jain, Ashwin Sah, and Mehtaab Sawhney. Anticoncentration versus the number of subset sums. *Adv. Comb.*, pages Paper No. 6, 10, 2021. doi:10.19086/aic.24872. 11

[Kus23] William Kuszmaul. Strongly history-independent storage allocation: New upper and lower bounds. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 1822–1841. IEEE, 2023. doi:10.1109/FOCS57990.2023.00111. 2, 3, 4, 10, 11, 12, 13, 16, 18, 37

[LGL15] Wei Quan Lim, Seth Gilbert, and Wei Zhong Lim. Dynamic reallocation problems in scheduling. *arXiv preprint arXiv:1507.01981*, 2015. arXiv:1507.01981. 2

[LMM+22] Shachar Lovett, Raghu Meka, Ian Mertz, Toniann Pitassi, and Jiapeng Zhang. Lifting with sunflowers. In *13th Innovations in Theoretical Computer Science Conference, ITCS 2022, January 31 - February 3, 2022, Berkeley, CA, USA*, volume 215 of *LIPIcs*, pages 104:1–104:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ITCS.2022.104. 11

[LNO96] Michael Luby, Joseph Naor, and Ariel Orda. Tight bounds for dynamic storage allocation. *SIAM J. Discret. Math.*, 9(1):155–166, 1996. doi:10.1137/S089548019325647X. 2, 11

[LSZ19] Shachar Lovett, Noam Solomon, and Jiapeng Zhang. From DNF compression to sunflower theorems via regularity. In *34th Computational Complexity Conference, CCC 2019, July 18-20, 2019, New Brunswick, NJ, USA*, volume 137 of *LIPIcs*, pages 5:1–5:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.CCC.2019.5. 11

[LZ19] Shachar Lovett and Jiapeng Zhang. DNF sparsification beyond sunflowers. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 454–460. ACM, 2019. doi:10.1145/3313276.3316323. 11

[Mar05] Dániel Marx. Parameterized complexity of constraint satisfaction problems. *Comput. Complex.*, 14(2):153–183, 2005. doi:10.1007/S00037-005-0195-9. 11

[NPSW23] Jesper Nederlof, Jakub Pawlewicz, Céline M. F. Swennenhuis, and Karol Węgrzycki. A faster exponential time algorithm for bin packing with a constant number of bins via additive combinatorics. *SIAM J. Comput.*, 52(6):1369–1412, 2023. URL: https://doi.org/10.1137/22m1478112, doi:10.1137/22M1478112. 11

[NT01] Moni Naor and Vanessa Teague. Anti-persistence: history independent data structures. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 492–501. ACM, 2001. doi:10.1145/380752.380844. 2, 10

[PP24] Jinyoung Park and Huy Tuan Pham. A proof of the Kahn-Kalai conjecture. *J. Amer. Math. Soc.*, 37(1):235–243, 2024. doi:10.1090/jams/1028. 11

[Rao20] Anup Rao. Coding for sunflowers. *Discrete Anal.*, pages Paper No. 2, 8, 2020. doi:10.19086/da. 11, 13

[Rao23] Anup Rao. Sunflowers: from soil to oil. *Bull. Amer. Math. Soc. (N.S.)*, 60(1):29–38, 2023. doi:10.1090/bull/1777. 11, 14

[Rao26] Anup Rao. The story of sunflowers. *J. Lond. Math. Soc. (2)*, 113(1):Paper No. e70380, 2026. doi:10.1112/jlms.70380. 11

[Raz85] A. A. Razborov. Lower bounds on the monotone complexity of some Boolean functions. *Dokl. Akad. Nauk SSSR*, 281(4):798–801, 1985. 11

[Rob71] John Michael Robson. An estimate of the store size necessary for dynamic storage allocation. *J. ACM*, 18(2):416–423, 1971. doi:10.1145/321650.321658. 2, 11

[Rob74] John Michael Robson. Bounds for some functions concerning dynamic storage allocation. *J. ACM*, 21(3):491–499, 1974. doi:10.1145/321832.321846. 2, 11

[Ros14] Benjamin Rossman. The monotone complexity of k-clique on random graphs. *SIAM J. Comput.*, 43(1):256–279, 2014. doi:10.1137/110839059. 11

[RR18] Sivaramakrishnan Natarajan Ramamoorthy and Anup Rao. Lower bounds on non-adaptive data structures maintaining sets of numbers, from sunflowers. In *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA*, volume 102 of *LIPIcs*, pages 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICS.CCC.2018.27. 11

[Sár94] A. Sárközy. Finite addition theorems. II. *J. Number Theory*, 48(2):197–218, 1994. doi:10.1006/jnth.1994.1062. 11

[Sch11] Tomasz Schoen. Arithmetic progressions in sums of subsets of sparse sets. *Acta Arith.*, 147(3):283–289, 2011. doi:10.4064/aa147-3-7. 11

[SSS09] Peter Sanders, Naveen Sivadasan, and Martin Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009. doi:10.1287/MOOR.1090.0381. 2

[SV06a] E. Szemerédi and V. Vu. Long arithmetic progressions in sumsets: thresholds and bounds. *J. Amer. Math. Soc.*, 19(1):119–169, 2006. doi:10.1090/S0894-0347-05-00502-3. 11

[SV06b] E. Szemerédi and V. H. Vu. Finite and infinite arithmetic progressions in sumsets. *Ann. of Math. (2)*, 163(1):1–35, 2006. doi:10.4007/annals.2006.163.1. 11

[Tan22] Till Tantau. On the satisfaction probability of k-cnf formulas. In *37th Computational Complexity Conference, CCC 2022, July 20-23, 2022, Philadelphia, PA, USA*, volume 234 of *LIPIcs*, pages 2:1–2:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.CCC.2022.2. 11

[Tao20] Terence Tao. The sunflower lemma via shannon entropy. *What's New [Terence Tao Blog]*, 2020. URL: https://terrytao.wordpress.com/2020/07/20/the-sunflower-lemma-via-shannon-entropy/. 11, 13