Space-Efficient Randomized Algorithms for K-SUM

Joshua R. Wang

Stanford University, Stanford CA 94305, USA
joshua.wang@cs.stanford.edu

Abstract. Recent results by Dinur et al. (2012) on random SubsetSum instances and by Austrin et al. (2013) on worst-case SubsetSum instances have improved the long-standing time-space tradeoff curve. We analyze a family of hash functions previously introduced by Dietzfelbinger (1996), and apply it to decompose arbitrary k-Sum instances into smaller ones. This allows us to extend the aforementioned tradeoff curve to the k-Sum problem, which is SubsetSum restricted to sets of size k. Three consequences are:

- a Las Vegas algorithm solving 3-Sum in $O(n^2)$ time and $\tilde{O}(\sqrt{n})$ space,
- a Monte Carlo algorithm solving k-Sum in $\tilde{O}(n^{k-\sqrt{2k}+1})$ time and $\tilde{O}(n)$ space for $k \geq 3$, and
- a Monte Carlo algorithm solving k-Sum in $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta(\sqrt{2k}-2)})$ time and $\tilde{O}(n^\delta)$ space, for $\delta \in [0,1]$ and $k \geq 3$.

Keywords: k-sum, subset-sum, hashing, time-space tradeoffs.

1 Introduction

The k-Sum problem on n numbers is as follows: Given k sets $S_1, S_2, \ldots, S_k$ with at most n integers each and a target t, find $a_1, a_2, \ldots, a_k$ such that for all i, $a_i \in S_i$ and $\sum_{i=1}^{k} a_i = t$. One common variant of the problem has only a single set S from which all elements in the solution are chosen from, but the two are easily reducible to each other. The k-Sum problem can be trivially solved in $O(n^k)$ arithmetic operations by trying all possibilities, and a more sophisticated solution runs in $O(n^{\lceil k/2 \rceil} \log n)$ time (this log factor can be avoided for k odd). However, this solution also requires $O(n^{\lceil k/2 \rceil})$ space, while the trivial solution only needed $O(1)$ space. Is some trade-off between time and space possible? Schroeppel an Shamir [12] provide an algorithm for 4-Sum that runs in $\tilde{O}(n^2)$ time[1] and $\tilde{O}(n)$ space. In a survey of the time and space complexity of exact algorithms, Woeginger [13] studied the k-Sum problem and questioned whether an algorithm similar to the Schroeppel-Shamir 4-Sum algorithm can be constructed for 6-Sum.

[1] See Section 2.1 for an explanation of $\tilde{O}$ and O^* notation.

Gajentaan and Overmars [6] classified many problems from computational geometry as "3-Sum-hard" (i.e. there exists a $o(n^2)$ reduction from 3-Sum to the problem in question) in order to indirectly demonstrate their difficulty. Finding a subquadratic algorithm for any problem in this class of problems would immediately produce a subquadratic algorithm for 3-Sum. One example of such a problem is 3-POINTS-ON-LINE: Given a set of points in the plane, are there three collinear points? To reduce 3-Sum to this problem, map each $x \in S$ (using the single-set variation of 3-Sum) to the point (x, x^3), with the idea that $a_1 + a_2 + a_3 = 0$ iff the points (a_1, a_1^3), (a_2, a_2^3), and (a_3, a_3^3) are collinear.

k-Sum is also fundamentally connected to several NP-hard problems. For example, Pătraşcu and Williams [11] show that solving k-Sum over n numbers in $n^{o(k)}$ time would imply that 3-Sat with n variables can be solved in $2^{o(n)}$ time. Schroeppel and Shamir [12] showed how the SubsetSum problem can be reduced to an (exponential-sized) k-Sum problem (Recall that in SubsetSum, we are given a set S of n integers and a target t, and want to find a subset $S' \subseteq S$ such that $\sum_{a \in S'} a = t$). Therefore, more efficient k-Sum algorithms can be used to derive faster SubsetSum algorithms. Indeed, Schroeppel and Shamir use their 4-Sum algorithm to produce a $O^*(2^{0.5n})$ time and $O^*(2^{0.25n})$ space SubsetSum algorithm. They also showed that SubsetSum is solvable in time T and space S where $T \cdot S^2 = O^*(2^n)$ for $T(n) \geq \Omega^*(2^{n/2})$.

This 30-year old time-space tradeoff for SubsetSum was recently improved. In 2010, Howgrave-Graham and Joux [7] derived an algorithm for random Subset-Sum instances that runs in time $O^*(2^{0.337n})$ and memory $O^*(2^{0.256n})$. Becker, Coron, and Joux [3] then derived two algorithms for random instances, one running in time $O^*(2^{0.291n})$ and space $O^*(2^{0.291n})$ and one running in $O^*(2^{0.72n})$ time and $O^*(1)$ space. Dinur et al. [5]. presented a time-space tradeoff curve that dominates the Schroeppel-Shamir curve and matches it at its endpoints, for random instances. Austrin et al. [1] matched the Dinur curve *for worst case instances* with a randomized algorithm.

1.1 Our Results

The best known algorithm for 3-Sum takes $\tilde{O}(n^2)$ time (Baran, Demaine, and Pătraşcu [2] have found polylogarithmic improvements over $O(n^2)$ which were further improved by Gronlund and Pettie [9]), but also requires $\tilde{\Omega}(n)$ space (to hold a sorted copy of the input). Can the same running time be achieved with significantly less space? The primary difficulty here lies in the unsortedness of the input. What about for k-Sum in general? In his 2004 survey [13], Woeginger asked such questions, with the hopes of encouraging further progress on solving SubsetSum.

In this paper, we lower the space requirement for 3-Sum while maintaining the same running time, with a zero-error randomized algorithm:

Theorem 1. *The* 3-Sum *problem on n numbers can be solved by a Las Vegas* [2] *algorithm in time $O(n^2)$ and space* [3] $\tilde{O}(\sqrt{n})$.

We also investigate time-space tradeoffs for the general k-Sum problem. Given a fixed space budget S, for what T can k-Sum be solved in time T and space S? We prove the following general self-reduction for k-Sum:

Theorem 2. *Let A be a Las Vegas algorithm that solves k-*Sum *($k \geq 3$) on n numbers in $T(n)$ time and $S(n)$ space where $T(n), S(n) \in poly(n)$, and let $\delta \leq 1$ be an arbitrary constant. Then there is a Las Vegas algorithm A' that solves k-*Sum *on n numbers in $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1}T(n^\delta))$ time and $O(n^\delta + S(n^\delta))$ space.*

When $S(n) = \tilde{O}(n)$, the reduction of Theorem 2 optimizes its space usage.

Independently of Theorem 2, we also provide a family of Monte Carlo randomized algorithms for k-Sum that use linear space.

Theorem 3. *There exists a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-f(k)-1})$ time and $\tilde{O}(n)$ space, where $f(k)$ is the largest integer p such that the $(p+1)$-th triangular number is at most one less than k.*

Theorem 3 provides algorithms for: 4-Sum in $\tilde{O}(n^2)$ time, 5-Sum in $\tilde{O}(n^3)$ time, 6-Sum in $\tilde{O}(n^4)$ time, 7-Sum in $\tilde{O}(n^4)$ time, and so on, *all in linear space.* Note that the 4-Sum time matches the Schroeppel-Shamir [12] 4-Sum result.

The actual savings over $O(n^k)$ time in Theorem 3 are subtle, and studied in detail later in the paper (for now, we note that $f(k) \geq \sqrt{2k} - 2$). Theorem 3 strengthens the time-space tradeoff results of Dinur et al. and Austrin et al. in the following sense. Applying the original Schroeppel-Shamir reduction from SubsetSum to k-Sum, and running the algorithm of Theorem 3, we can recover the endpoints of the time-space tradeoff curve previously obtained. In particular, this occurs when k is one more than a triangular number.

Combining Theorem 2 and Theorem 3, we obtain the following time-space tradeoff curve for k-Sum in the "sub-linear" space setting:

Corollary 1. *Let $\delta \in [0,1]$ be an arbitrary constant. There is a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta f(k)})$ time and $\tilde{O}(n^\delta)$ space, where $f(k)$ is the largest integer p such that the $(p+1)$-th triangular number is at most one less than k.*

1.2 Intuition

To illustrate some of the ideas in this paper, let us consider 3-Sum. The naive algorithm checks all triples of numbers to see if they sum to the target, in $O(n^3)$

[2] Recall that algorithms are Las Vegas randomized if they always give correct results, but may take additional running time depending on the random numbers generated (but not depending on the choice of input).

[3] We consider a model of computation where the input is given in read-only memory while the machine works in read/write memory, measuring the space usage by the *working memory* size.

time. Note that choosing two numbers in the solution determines the third. A more careful algorithm will store the third set in a data structure so that after choosing the first two numbers of the solution, the third can be quickly checked. Because *the last number is determined*, this only requires $O(n^2)$ time.

Now consider algorithms that only use $\tilde{O}(\sqrt{n})$ space. One naive approach is to partition each set into $\sqrt{n}$ buckets of $\sqrt{n}$ numbers and solve 3-Sum on all triples of buckets. There are $n^{1.5}$ such triples, and since solving 3-Sum on these smaller instances will take $O(n)$ time, the running time of this naive algorithm is $O(n^{2.5})$. However, this algorithm does redundant work checking buckets, for the same reason that the $O(n^3)$ algorithm does redundant work checking numbers: both check all possible triples.

We avoid this work via hashing. We apply a particular hash family H^{lin} of Dietzfelbinger [4] to create our buckets. We show that with this hash family, choosing the first two buckets *fixes the third bucket*. Hence we now only check $O(n)$ triples of buckets, and the running time drops to $O(n^2)$. With H^{lin}, we can ensure that the sizes of the buckets also remain $O(\sqrt{n})$, to avoid increasing the running time for solving a subproblem.

We can generalize this technique in two different ways. Firstly, this technique works for general k: guessing the first $(k-1)$ buckets *fixes the last bucket* in a general k-Sum solution. Secondly, it works for sizes other than $O(\sqrt{n})$, although additional work is necessary to guarantee the bucket size. In fact, this generalization yields a self-reduction for k-Sum problems, since the resulting subproblems are smaller instances of k-Sum.

The running time and space usage of the final k-Sum algorithm depend on how the subproblems from the self-reduction are solved. The space usage in the self-reduction is optimized when a linear-space k-Sum algorithm is applied to the resulting subproblems. Hence we take particular interest in linear-space k-Sum algorithms and derive faster linear-space k-Sum algorithms, by adapting SubsetSum techniques to the k-Sum setting.

1.3 Organization

In Section 2, we discuss some basic notation and several standard k-Sum algorithms. In Section 3, we study a family of hash functions, introduced by Dietzfelbinger [4], and analyze its properties. In particular, we show the family is "almost-affine". In Section 4, we use this hash family to develop a self-reduction on k-Sum problems to reduce memory usage. In Section 5, we analyze the k-Sum time-space tradeoff curves produced by this reduction.

In Appendix A, we present the proof for our general k-Sum self-reduction theorem. In Appendix B, we use the previously mentioned hash family to also derive linear-space Monte Carlo algorithms for k-Sum.

2 Preliminaries

2.1 Randomized Algorithms and Running Time

This paper describes both Las Vegas and Monte Carlo randomized algorithms. Las Vegas algorithms always give correct results, but their running times hold in expectation over internal randomness (the input is still worst-case). Monte Carlo algorithms may give incorrect results with some (small) probability, but their running time is deterministic and worst-case. It is worth noting that a Las Vegas algorithm can be converted into a Monte Carlo algorithm with the same running time up to a constant factor, via a Markov bound.

We use $\tilde{O}$ to indicate suppression of polylog factors, and O^* to indicate suppression of polynomial factors.

When determining running time, we will use the standard word RAM model, assuming that operations on integers take constant time. Note that polylog time operations would still fold into the $\tilde{O}$ notation and do not affect the polynomial exponent, which is the primary focus of this paper.

2.2 Sets and Triangular Numbers

Definition 1. $[m]$ *denotes the set* $\{0, 1, \ldots, m-1\}$.

Definition 2. *Given sets S and T, the Minkowski sum of S and T, denoted $S + T$, is defined as the set* $\{s + t \mid s \in S, t \in T\}$.

Definition 3. *Given a set S and a function f that can operate on the elements of S, the image of S under f, denoted $f(S)$, is defined as the set* $\{f(s) \mid s \in S\}$.

Definition 4. *The n^{th} triangular number, T_n, is given by* $\sum_{i=1}^{n} i = \binom{n+1}{2}$.

2.3 Basic k-Sum Algorithms

We review standard algorithms for k-Sum on n numbers where $k \in [2, 4]$.

Theorem 4. 2-Sum *on n numbers can be solved in $O(n \log n)$ time and $\tilde{O}(n)$ space.*

The key idea is to sort one set in nondecreasing order and the other in nonincreasing order. Starting at the beginning of each set, advance the element in the first set if the current sum is too small and advance the element in the second set if the current sum is too large.

Theorem 5. 3-Sum *on n numbers can be solved in $O(n^2)$ time and $\tilde{O}(n)$ space.*

3-Sum proceeds similarly, sorting the first two sets as above and then brute-force guessing the element from the third set to use.

Theorem 6 (Schroeppel Shamir '79). 4-Sum *on n numbers can be solved in $O(n^2 \log n)$ time and $\tilde{O}(n)$ space.*

4-Sum is solved by reducing to the 2-Sum case, treating $S_1 + S_2$ as one set and $S_3 + S_4$ as another. In order to avoid $\tilde{O}(n^2)$ space usage, Schroeppel and Shamir [12] use a priority queue for each set sum, each holding at most a linear number of elements.

2.4 Hash Functions

Here are some definitions concerning hash functions:

Definition 5. *A family of hash functions $H = \{h : U \to [m]\}$ is said to be universal if for every $x, y \in U$, if $x \neq y$ then $Pr_{h\in H}[h(x) = h(y)] \leq \frac{1}{m}$.*

Definition 6. *Given a family of hash functions $H = \{h : U \to [m]\}$, and a set $S \subseteq U$, let the bucket of h with value v be $h^{-1}(\{v\}) \cap S$ (i.e. all elements in S with hash value v). Also, define $\mathcal{B}_h(x) := h^{-1}(\{h(x)\}) \cap S$ (the bucket of h with value $h(x)$).*

We will be hashing the elements in our k-Sum instance. We note that we can assume elements (and hence $|U|$) are at most $\tilde{O}(n^k)$, since we can take all numbers modulo a random prime on the order of $\tilde{O}(n^k)$; there are only $O(n^k)$ sums to consider, each with at most $O(\log n)$ prime factors (the prime number theorem guarantees there are enough primes to choose from). Note that we can verify solutions to guard against collisions, so our algorithms are Las Vegas randomized.

3 Almost Affine Hashing

Hashing has long been useful in k-Sum algorithms and reductions. Baran, Demaine, and Pătraşcu [2] proved a key lemma about the load balancing property of universal families of hash functions. Baran et al. use this to show an upper-bound for 3-Sum, Pătraşcu [10] uses it to reduce 3-Sum to 3-SumConvolution, and Abboud and Lewi show a more general reduction from k-Sum to k-SumConvolution for $k \geq 2$.

Lemma 1 (Baran Demaine Pătraşcu '05). *Given any universal family of hash functions $H = \{h : [u] \to [m]\}$, some set $S \subset [u]$ of size n, and an integer $t > 2n/m - 2$, the expected number of elements $x \in S$ with $|\mathcal{B}_h(x)| \geq t$ is at most $\frac{2n}{t-2n/m+2}$.*

However, Jafargholi and Viola [8] recently pointed out that it appears that the family of hash functions used by Baran et al. with this lemma is not known to be universal. They do suggest that similar hash functions studied by Dietzfelbinger [4] might work, but do not explore the issue further. We show that this is indeed the case; we can use the following family of hash functions:

Definition 7. *Let u, m, and k positive integers be given. For $a, b \in [km]$, let the hash function $h_{a,b} : [u] \to [m]$ be defined as $h_{a,b}(x) = ((ax + b) \mod km) \operatorname{div} k$, where div is integer division.*

Let the family of hash functions $H^{lin}_{u,m,k}$ be defined as $\{h_{a,b} \mid a, b \in [km]\}$.

Theorem 7 (Dietzfelbinger '96). *If m, u, and k are all powers of 2, and $k \geq u/2$, then $H^{lin}_{u,m,k}$ is universal. In fact, it is two-wise independent, i.e. $Pr_{h \in H^{lin}_{u,m,k}}[h(x_1) = i_1 \wedge h(x_2) = i_2] = 1/m^2$ for arbitrary $i_1, i_2 \in [m]$ and distinct $x_1, x_2 \in [u]$.*

This family of hash functions is particularly interesting with the constraint that all sizes are powers of two, since it can be implemented with bit shift operations, does not require a large prime, and uses relatively few operations, all of which were noted by Dietzfelbinger [4].

With this bound on k in mind, denote $H^{lin}_{u,m,\lceil u/2 \rceil}$ as $H^{lin}_{u,m}$.

Baran, Demaine, and Pătraşcu [2] also relied the fact that the hash function they chose was "almost-linear". We prove a similar property for $H^{lin}_{u,m}$. Call a family of hash functions H that map from $[u]$ to $[m]$ *almost-affine* if for all $h \in H$ and $x, y \in [u]$, $h(x+y) \in \{h(x) + h(y) - h(0) + z \pmod m \mid z \in \{-1, 0, 1\}\}$.

Lemma 2. *The family of hash functions $H^{lin}_{u,m}$ is almost-affine.*

Proof. The main idea is that dividing by k before addition can only influence the result by at most 1 due to losing a carry. Suppose we have some integers a, b. Then we can write a as $ka_1 + a_2$ and b as $kb_1 + b_2$, where $a_2, b_2 \in [k]$. Notice that:

$$\begin{aligned}
(a \ \operatorname{div} k) + (b \ \operatorname{div} k) &= a_1 + b_1 \\
&= \begin{cases} ((ka_1 + kb_1 + a_2 + b_2) \ \operatorname{div} k) & \text{if } a_2 + b_2 < k \\ ((ka_1 + kb_1 + a_2 + b_2) \ \operatorname{div} k) - 1 & \text{if } a_2 + b_2 \geq k \end{cases} \\
&\in \{((a+b) \ \operatorname{div} k) + z \mid z \in \{-1, 0\}\}.
\end{aligned}$$

Hence, we can observe that:

$$\begin{aligned}
h(x+y) + h(0) \pmod m &= (((ax + ay + b) \mod km) \ \operatorname{div} k) \\
&\quad + ((b \mod km) \ \operatorname{div} k) \pmod m \\
&\in \{(((ax + ay + 2b) \mod km) \ \operatorname{div} k) \\
&\quad + z \pmod m \mid z \in \{-1, 0\}\} \\
h(x) + h(y) \pmod m &= (((ax + b) \mod km) \ \operatorname{div} k) \\
&\quad + ((ay + b \mod km) \ \operatorname{div} k) \pmod m \\
&\in \{(((ax + ay + 2b) \mod km) \ \operatorname{div} k) \\
&\quad + z \pmod m \mid z \in \{-1, 0\}\}.
\end{aligned}$$

Hence, $h(x+y) \in \{h(x) + h(y) - h(0)\} + \{-1, 0, 1\} \pmod m$, as desired. □

Lemma 2 guarantees that if $(k-1)$ sets have their hash buckets fixed, any solution that uses elements from those buckets could only have its last element in one of $2k-1$ buckets of the last set. Hence, hashing can be used to shrink the problem size with some limited growth in the number of cases. It is worth noting that this hash works best on 3-Sum, since for larger values of k, applying the hash tends to increase the running time of the algorithm.

It can be seen that for large enough m, large buckets can be completely avoided by simply inspecting a constant number of hashes (in expectation).

Corollary 2. *Consider a universal family of hash functions $H = \{h : [u] \to [m]\}$, a set $S \subset [u]$ of size n, where $m \le \sqrt{n}$, and an arbitrary constant $c \ge 1$. Then:*

$$Pr_{h\in H}\left[\forall x \in S : |\mathcal{B}_h(x)| \le (c+2)\frac{n}{m}\right] \ge 1 - \frac{2}{c^2}$$

Proof. Let $t = (c+2)\frac{n}{m}$. Let $b(h)$ be the number of elements $x \in S$ with $|\mathcal{B}_h(x)| \ge t$. Applying Lemma 1 yields that $E[b(h)] \le \frac{2n}{c(n/m)+2} \le \frac{2m}{c}$. Applying a Markov bound yields $Pr_h[b(h) \ge cm] \le \frac{2}{c^2}$. However, if $b(h) < cm$ then in fact $b(h) = 0$, since $b(h)$ counts the number of elements in buckets of h with at least $(c+2)\frac{n}{m}$ elements ($m \le \sqrt{n}$ implies $\frac{n}{m} \ge m$). Hence,

$$Pr_h\left[\forall x \in S : |\mathcal{B}_h(x)| \le (c+2)\frac{n}{m}\right] \ge 1 - \frac{2}{c^2}.$$

This completes the proof. □

The next lemma is analogous to a result proved by Baran, Demaine, and Pătraşcu [2] for 3-Sum and their hash family, but it holds for general k and our almost-affine family. It will be used to limit the number of false positives after hashing.

Lemma 3. *Given a constant k and integers $a_1, a_2, \ldots, a_k$ and $b_1, b_2, \ldots, b_k$ where $\sum_{i=1}^k a_i \ne \sum_{i=1}^k b_i$, the probability that $\sum_{i=1}^k h(a_i) = \sum_{i=1}^k h(b_i)$ after picking a random $h \in H^{lin}_{u,m}$ is upper-bounded by $\frac{O(1)}{m}$.*

Proof. By repeated application of Lemma 2, for all $h \in H^{lin}_{u,m}$:

$$h(\sum_{i=1}^k a_i) \in \left\{\sum_{i=1}^k h(a_i) - (k-1)h(0) + z \pmod m \mid z \in \{-k+1, \ldots, k-1\}\right\}$$
$$h(\sum_{i=1}^k b_i) \in \left\{\sum_{i=1}^k h(b_i) - (k-1)h(0) + z \pmod m \mid z \in \{-k+1, \ldots, k-1\}\right\}$$

Suppose that $\sum_{i=1}^k h(a_i) = \sum_{i=1}^k h(b_i)$. This could only occur if $h\left(\sum_{i=1}^k a_i\right)$ and $h\left(\sum_{i=1}^k b_i\right)$ are within $2k-2$ of each other.

However, $H^{lin}_{u,m}$ is two-wise independent by Theorem 7. There are only $m(4k-3)$ ways to assign the values of $h\left(\sum_{i=1}^{k} a_i\right)$ and $h\left(\sum_{i=1}^{k} b_i\right)$ because they are within $2k-2$ of each other. This leads to an upper bound on the probability of $\sum_{i=1}^{k} h(a_i) = \sum_{i=1}^{k} h(b_i)$ of $\frac{4k-3}{m}$. But k is constant, completing the proof. □

4 A *k*-Sum Self-reduction

This section uses the hashing results to derive space-efficient randomized algorithms for k-Sum. Specifically, we demonstrate how to reduce the space usage of k-Sum algorithms using Corollary 2. These reductions are Las Vegas randomized. It is worth noting that without Corollary 2, the same running times could be attained, but via Monte Carlo algorithms and the universality of $H^{lin}_{u,m}$.

We begin by illustrating the idea with the 3-Sum problem, and then present a theorem for general k. This family of hash functions does particularly well when applied to 3-Sum, since it does not increase the running-time cost.

Reminder of Theorem 1. *The* 3-Sum *problem on n numbers can be solved by a Las Vegas algorithm in time $O(n^2)$ and space $\tilde{O}(\sqrt{n})$.*

Proof. Algorithm 1 is the desired algorithm.

Algorithm 1. SpaceEfficient3Sum(S_1, S_2, S_3, t)

```
1: Set $m \leftarrow \sqrt{n}$.
2: Randomly choose a hash function $H^{lin}_{u,m}$.
3: for $v \in [m]$ and $i \in \{1,2,3\}$ do
4:     Count the $a_i \in S_i$ where $h(a_i) = v$, and store the result in $c$.
5:     if $c > 5\sqrt{n}$ then
6:         Restart the algorithm.
7:     end if
8: end for
9: for $sum \in \{h(t) - 2h(0) + z \pmod{m} \mid z \in \{-2,\ldots,2\}\}$ do
10:    for $v_1 \in [m]$ do
11:        for $v_2 \in [m]$ do
12:            Set $v_3 \leftarrow sum - v_1 - v_2 \pmod{m}$.
13:            Let $S'_i = \{a_i \in S_i \mid h(a_i) = v_i\}$ for $i = 1,2,3$.
14:            Run the basic 3-Sum algorithm on $S'_1, S'_2, S'_3, t$. Return any found solution.
15:        end for
16:    end for
17: end for
18: Report no solution.
```

Correctness: Since $H^{lin}_{u,m}$ is almost-affine (Lemma 2), for any solution (a_1,a_2,a_3):

$$h(a_1) + h(a_2) + h(a_3) \in \{h(t) - 2h(0) + z \pmod{m} \mid z \in \{-2,\ldots,2\}\}.$$

Hence at some point $v_i = h(a_i)$ for $i = 1, 2, 3$ and the algorithm will find this solution. When no solutions exist, the algorithm cannot find one.

Running Time: Since $m = \sqrt{n}$, applying Corollary 2 with $c = 3$ yields that there is at least a $\frac{7}{9}$ chance that all buckets for a specific set S_i are at most $5\sqrt{n}$ elements in any bucket. By a union bound, there is at least a $\frac{1}{3}$ chance that all sets S_i have at most $5\sqrt{n}$ elements in any bucket. Hence in expectation the algorithm picks at most three hashes before it gets past the bucket size check.

Checking bucket sizes takes $O(n^{1.5})$ time, since a linear scan is done for each of $\sqrt{n}$ values of v. There are $O(n)$ choices for v_1, v_2, v_3, and each iteration runs the basic 3-Sum algorithm on instances of size $O(\sqrt{n})$, taking $O(n)$ time. The total running time is hence $O(n^2)$. Note that Baran, Demaine, Pătraşcu [2] could be used for subproblems to shave off additional log factors.

Memory Usage: Since bucket sizes are guaranteed not to be too large, storing S'_i only requires $\tilde{O}(\sqrt{n})$ space. Running the basic 3-Sum algorithm on instances of $O(\sqrt{n})$ elements also uses $\tilde{O}(\sqrt{n})$ space.

This completes the proof. □

This technique holds for general k as well as sizes other than $\tilde{O}(\sqrt{n})$. Theorem 1 is actually an application of the following general theorem:

Reminder of Theorem 2. *Let A be a Las Vegas algorithm that solves k-Sum ($k \geq 3$) on n numbers in $T(n)$ time and $S(n)$ space where $T(n), S(n) \in poly(n)$, and let $\delta \in [0, 1]$ be an arbitrary constant. Then there is a Las Vegas algorithm A' that solves k-Sum on n numbers in $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1}T(n^\delta))$ time and $O(n^\delta + S(n^\delta))$ space.*

The proof of this theorem is more complex and nuanced, and can be found in Appendix A. Notice when $\delta = 0$, we recover the brute force algorithm's running time and space usage. When $\delta = 1$, we recover the input algorithm's (given that the input algorithm uses at least linear space).

As mentioned previously, there is a naive self-reduction that shrinks space usage. It splits each set up into buckets of size $O(n^\delta)$, and runs another algorithm on each possible combination of buckets. This would result in an algorithm that runs in $\tilde{O}(n^{k-\delta k}T(n^\delta))$ time and $\tilde{O}(S(n^\delta))$ space. This naive reduction also recovers brute-force for $\delta = 0$ and the input algorithm for $\delta = 1$, and in fact interpolates the exponents of the two algorithms for values of δ in between. Our reduction via hashing beats this naive reduction for all $\delta \in [0, 1]$, with equality only at the endpoints (given that the input algorithm uses at least linear space).

5 Time-Space Tradeoffs for k-Sum

This section explores the results we get by applying Theorem 2. The following theorem provides a family of k-Sum algorithms to use on subproblems:

Reminder of Theorem 3. *There exists a Monte Carlo algorithm that solves k-Sum on n numbers in $\tilde{O}(n^{k-f(k)-1})$ time and $\tilde{O}(n)$ space, where $f(k)$ is the largest integer p such that $T_{p+1} + 1 \leq k$.*

The proof of Theorem 3 is in Appendix B. This theorem yields:

Reminder of Corollary 1. *Let $\delta \in [0,1]$ be an arbitary constant. There is a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta f(k)})$ time and $\tilde{O}(n^{\delta})$ space, where $f(k)$ is the largest integer p such that the $(p+1)$-th triangular number is at most one less than k.*

Proof. This follows directly from applying the reduction from Theorem 2 to the algorithm guaranteed by Theorem 3. □

Corollary 1 gives a tradeoff curve that consists of two linear pieces. Suppose we want a k-Sum algorithm that runs in $\tilde{O}(n^{\delta})$ space. For the region $\delta \in [0, \frac{1}{(k-1)-f(k)}]$, we have an algorithm that runs in $\tilde{O}(n^{k-\delta(k-1)})$ time. In the region $\delta \in [\frac{1}{(k-1)-f(k)}, 1]$, we have one that runs in $\tilde{O}(n^{k-1-\delta(f(k))})$ time. At the shared point in these two intervals, the running time is $\tilde{O}(n^{k-(k-1)/(k-1-f(k))})$.

We also note that $f(k)$ has the following (coarse) lower bound:

Lemma 4. $f(k) \geq \sqrt{2k} - 2$

Proof. Notice that $T_{\lceil \sqrt{2k}-1 \rceil} \leq \frac{\sqrt{2k}(\sqrt{2k}-1)}{2} \leq k$. But $f(k)$ is the largest integer p for which $T_{p+1} + 1 \leq k$, so it must be at least $\lceil \sqrt{2k} - 2 \rceil$. □

This immediately gives upper-bounds for Theorem 3 and Corollary 1:

Corollary 3. *There exists a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-\sqrt{2k}+1})$ time and $O(n)$ space.*

Corollary 4. *Let $\delta \in [0,1]$ be an arbitary constant. There is a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-\delta(k-1)} + n^{k-1-\delta f(k)})$ time and $\tilde{O}(n^{\delta})$ space, where $f(k)$ is the largest integer p such that the $(p+1)$-th triangular number is at most one less than k.*

6 Conclusion

Our results also extend to the k-Xor problem [8], which is identical except that the elements are vectors from $\mathbb{F}_2^n$ instead of integers. For this variant there is a simple linear universal family of hash functions (let M be a random $k \times n$ matrix over $\mathbb{F}_2$, and define $h_M(x) = Mx$). Hence the hashing properties we need easily hold in this case, and the same techniques work.

One open problem is whether our family of linear-space Monte Carlo algorithms for k-Sum can be derandomized or be made to work for real inputs. The Schroeppel-Shamir algorithm for 4-Sum matches the $\tilde{O}$ running-time, so it seems plausible that this might hold for larger values of k.

An especially interesting open problem is whether the algorithms presented by Howgrave-Graham and Joux as well as Becker, Coron, and Joux can be moved from random-instances to worst-case instances and randomized algorithms. Giving better worst-case bounds for SubsetSum has been an open problem for more than 30 years.

Acknowledgements. Many thanks to Ryan Williams for providing helpful pointers to existing literature, insightful discussions, and proofreading.

References

1. Austrin, P., Kaski, P., Koivisto, M., Määttä, J.: Space–time tradeoffs for subset sum: An improved worst case algorithm (2013)
2. Baran, I., Demaine, E.D., Pǎtraşcu, M.: Subquadratic algorithms for 3SUM. In: Dehne, F., López-Ortiz, A., Sack, J.-R. (eds.) WADS 2005. LNCS, vol. 3608, pp. 409–421. Springer, Heidelberg (2005)
3. Becker, A., Coron, J.-S., Joux, A.: Improved Generic Algorithms for Hard Knapsacks. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 364–385. Springer, Heidelberg (2011)
4. Dietzfelbinger, M.: Universal hashing and k-wise independent random variables via integer arithmetic without primes. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 569–580. Springer, Heidelberg (1996)
5. Dinur, I., Dunkelman, O., Keller, N., Shamir, A.: Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 719–740. Springer, Heidelberg (2012)
6. Gajentaan, A., Overmars, M.H.: On a class of $O(n^2)$ problems in computational geometry. Comput. Geom. Theory Appl. 45(4), 140–152 (2012)
7. Howgrave-Graham, N., Joux, A.: New Generic Algorithms for Hard Knapsacks. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 235–256. Springer, Heidelberg (2010)
8. Jafargholi, Z., Viola, E.: 3sum, 3xor, triangles. CoRR, abs/1305.3827 (2013)
9. Jørgensen, A.G., Pettie, S.: Threesomes, degenerates, and love triangles. CoRR, abs/1404.0799 (2014)
10. Patrascu, M.: Towards polynomial lower bounds for dynamic problems. In: STOC, pp. 603–610 (2010)
11. Pǎtraşcu, M., Williams, R.: On the possibility of faster sat algorithms. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 1065–1075. Society for Industrial and Applied Mathematics, Philadelphia (2010)
12. Schroeppel, R., Shamir, A.: A $T \cdot S^2 = O(2^n)$ Time/Space Tradeoff for Certain NP-Complete Problems. In: Proceedings of the 20th Annual Symposium on Foundations of Computer Science, SFCS 1979, Washington, DC, USA, pp. 328–336. IEEE Computer Society Press, Los Alamitos (1979), `http://dx.doi.org/10.1109/SFCS.1979.3`, doi:10.1109/SFCS.1979.3
13. Woeginger, G.J.: Space and time complexity of exact algorithms: Some open problems. In: Downey, R.G., Fellows, M.R., Dehne, F. (eds.) IWPEC 2004. LNCS, vol. 3162, pp. 281–290. Springer, Heidelberg (2004)

A Appendix: Proof of Theorem 2

Reminder of Theorem 2. *Let A be a Las Vegas algorithm that solves k-*Sum *($k \geq 3$) on n numbers in $T(n)$ time and $S(n)$ space where $T(n), S(n) \in poly(n)$,* *and let $\delta \in [0,1]$ be an arbitrary constant. Then there is a Las Vegas algorithm A' that solves k-*SUM *on n numbers in $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1}T(n^\delta))$ time and $O(n^\delta + S(n^\delta))$ space.*

Proof. The key idea is to use hashing to reduce the size of each set by a square root factor at each step. However, storing any of the intermediate sets of this computation defeats the purpose of hashing any further. To avoid this, the algorithm first determines all hash functions and values to shrink each set to the desired size, and then computes the final sets in one step.

A' will recursively construct a list L whose elements are of the form $(h, v_1, v_2, \ldots, v_k)$, i.e. a hash function followed by k hash values (one for each S_i). At any step, define the active set of S_i to be $\tilde{S}_i = \{s \in S_i \mid h(s) = v_i \forall (h, v_1, v_2, \ldots, v_k) \in L\}$. Each element appended to L reduces the size of all active sets, so elements can be repeatedly appended until the active sets are only $O(n^\delta)$ in size, at which point it is safe to invoke A. To handle the possibility that δ is not a perfect power of $\frac{1}{2}$, define the function $s(x) := \max((\frac{1}{2})^x, \delta)$. Step i of the algorithm will reduce the size of all active sets from $O(n^{s(i)})$ to $O(n^{s(i+1)})$.

The recursive helper function R will construct L and then invoke A. It has access to all sets S_i and is given a partially constructed L. Algorithm A' simply calls R with $L = \emptyset$.

Algorithm 2. $R(L, S_1, \ldots, S_k)$

```
Require: The active sets $\tilde{S}_1, \ldots, \tilde{S}_k$ each contain at most $(k+2)^2 n^{s(\ell)}$ elements.
 1: Let $\ell \leftarrow |L|$.
 2: if $s(\ell) = \delta$ then
 3:     Call $A(\tilde{S}_1, \ldots, \tilde{S}_k)$.
 4:     return
 5: end if
 6: Set $m_\ell \leftarrow (k+2)n^{s(\ell)-s(\ell+1)}$.
 7: Pick a random hash function $h \in H^{lin}_{u,m_\ell}$.
 8: for $v \in [m_\ell]$ and $i \in \{1, \ldots, k\}$ do
 9:     Count the $a_i \in \tilde{S}_i$ where $h(a_i) = v$, and store the result in $c$.
10:     if $c > (k+2)^2 n^{s(\ell+1)}$ then
11:         Pick another hash and try again.
12:     end if
13: end for
14: for $sum \in \{h(t) - (k-1)h(0) + z \pmod{k_\ell} \mid z \in \{-k+1, \ldots, k-1\}\}$ do
15:     for $v_1, \ldots, v_{k-1} \in [m_\ell]$ do
16:         Set $v_k \leftarrow sum - \sum_{i=1}^{k-1} v_i \pmod{m_\ell}$.
17:         Let $L'$ be $L$ appended with $(h, v_1, \ldots, v_k)$.
18:         Call $R(L', S_1, \ldots, S_k)$.
19:     end for
20: end for
```

Correctness: We first prove the size guarantee made when calling R. A' initially calls R with $\ell = 0$ and sets of size $n \leq (k+2)^2 n^{s(0)}$. R ensures that the hash it has chosen creates buckets that are no larger than $(k+2)^2 n^{s(\ell+1)}$ in size, so it may safely append an additional element to L before recursing.

We also want to show that if a solution exists, the algorithm will find it. Since $H^{lin}_{u,m}$ is almost affine, a call to R where each element of the solution is active will in turn make some recursive call where the solution elements are still active. Since the top-level call to R is made with all elements active, all elements of a solution will be found by the algorithm.

Running Time: Checking that the buckets of a randomly-selected hash function are not too large takes $O(n^{1+s(\ell)-s(\ell+1)})$ time since the algorithm needs to perform a linear scan for each hash value $v \in [m_\ell]$. Applying Corollary 2 with $c = k$, the chance of a hash failing over a specific S_i is at most $\frac{2}{k^2}$; the chance of it failing over any S_i, by a union bound, is at most $\frac{2}{k}$. Since $k \geq 3$, the expected number of hashes the algorithm needs to pick and check is at most three. Hence our expected time checking for hashes during a single call to R, not including recursive subcalls, is $O(n^{1+s(\ell)-s(\ell+1)})$.

There is a single call where $\ell = 0$. Each recursive level of R makes $O(n^{(k-1)(s(\ell)-s(\ell+1))})$ calls to the level below it. Hence, there are $O(n^{(k-1)(1-s(\ell))})$ calls to R for a fixed ℓ. The total expected time checking for hashes during all calls with a given ℓ is therefore $O(n^{(k-1)(1-s(\ell))+(1+s(\ell)-s(\ell+1))})$. But notice that:

$$\begin{aligned}(k-1)(1-s(\ell)) + (1+s(\ell)-s(\ell+1)) &= k - s(\ell)(k-2) - s(\ell+1)\\ &\leq k - s(\ell+1)(k-1)\\ &\leq k - \delta(k-1).\end{aligned}$$

Hence, the total expected time checking for hashes during all calls with a given ℓ is also $O(n^{k-\delta(k-1)})$. Since the algorithm only searches for hash functions for at levels up to $\lceil \log_2 \frac{1}{\delta} \rceil - 1$, the total expected running time checking for hashes overall is $O(n^{k-\delta(k-1)})$.

When $s(\ell) = \delta$, the algorithm needs to compute all $\tilde{S}_i$. From the previously-derived formula, there are only $O(n^{(k-1)(1-\delta)})$ calls where this occurs. Computing all $\tilde{S}_i$ only requires a linear scan of each S_i, so this takes time $O(n^{k-\delta(k-1)})$.

Finally, the algorithm invokes A $O(n^{(k-1)(1-\delta)})$ times on sets of size at most $(k+2)^2 n^\delta$, so in total the algorithm uses $O(n^{(k-1)(1-\delta)} T(n^\delta))$ time making calls to A.

The total time taken is $O(n^{k-\delta(k-1)} + n^{k-\delta(k-1)-1} T(n^\delta))$.

Memory Usage: Notice that L contains at most $\lceil \log_2 \frac{1}{\delta} \rceil$ elements of size $(k+1)$ each, so it takes $O(1)$ space. The space needed to verify there are no large buckets is also $O(1)$, since the algorithm only computes a count for only a single hash value at a time.

Invoking A on sets of size at most $(k+2)^2 n^\delta$ requires only $O(n^\delta + S(n^\delta))$ space (to store the inputs along with the space needed by A).

This completes the proof. □

B Appendix: Linear Space Algorithms for k-Sum

The two factors in the space usage of the algorithm derived from applying Theorem 2 are balanced when the original algorithm requires only linear space. In this appendix, we utilize almost affine hashing to produce a family of linear-space algorithms for k-Sum.

Theorem 8, Theorem 9, and Corollary 5 are based on results produced by Austrin, Kaski, Koivisto, and Määttä [1]. We shift from the SubsetSum problem to the k-Sum problem and use almost affine hashing in place of carefully chosen moduli. The switch from chosen moduli to this family of hash functions is justified by the fact that, as mentioned before, this hashing can be done with bit shift operations, without the availability of large primes, and with relatively few operations. As mentioned before, if the Schroeppel-Shamir reduction from SubsetSum to k-Sum is used on this set of algorithms, every endpoint of their piecewise-linear time-space tradeoff curve for SubsetSum is recovered, so this adaptation is lossless.

The technique requires that there are only a constant number of solutions to the k-Sum instance, which can be ensured by some standard preprocessing:

Theorem 8. *There is a $O(n \log n)$ time Monte Carlo algorithm to process instances of k-Sum which takes as input an instance $(S_1, \ldots, S_k, t)$ of size n and outputs $O(\log n)$ k-Sum instances of the same size. If the original instance has a solution, then at least one of the output instances will have at least one solution and at most $O(1)$ solutions. Otherwise, none of the instances will have any solutions.*

Proof. Consider a k-Sum instance $S_1, \ldots, S_k$ with target t. Without loss of generality, all sets contain only nonnegative elements (it is safe to add a positive constant to all elements in any particular set S_i and to the target t at the same time).

Let S be the set of all solutions. The algorithm will guess that the size of S is in the range $[2^s, 2^{s+1})$ for $s = 0, 1, \ldots, k \log n$ (try them all, one will be correct). Let $m = 2^s$, and for each S_i choose uniformly at random a function $f_i : S_i \to [m]$. Also, randomly choose a $u \in [m]$.

For a fixed solution $(a_1, \ldots, a_k) \in S$, there is a $\frac{1}{m}$ probability that:

$$\sum_{i=1}^{k} f_i(a_i) \equiv u \pmod{m} \qquad (1)$$

Also, any two distinct solutions both satisfy (1) with probability $\frac{1}{m^2}$. When s is a correct guess,
$1 \leq \frac{|S|}{m} < 2$. Let X be a random variable denoting the number of solutions that satisfy (1). Then:

$$\begin{aligned}\mathbb{E}[X] &= \frac{|S|}{m}\\ \mathbb{E}[X^2] &= \mathbb{E}[X] + \frac{|S|(|S|-1)}{m^2} < \frac{|S|}{m} + \frac{|S|^2}{m^2}.\end{aligned}$$

The first and second moment methods give:

$$\begin{aligned}Pr(X > 10) &< \frac{\mathbb{E}[X]}{10} < \frac{1}{5}\\ Pr(X > 0) &> \frac{\mathbb{E}[X]^2}{\mathbb{E}[X^2]} > \frac{1}{1+m/|S|} > \frac{1}{2}.\end{aligned}$$

By a union bound, there is at least one correct solution and at most $O(1)$ solutions that satisfy (1) with constant probability. If a solution satisfies (1), then in fact $\sum_{i=1}^{k} f_i(a_i) = u + jm$ for some $j \in [k]$ (there are at most $k-1$ carries). Guess this j by iterating over all possibilities.

Let A be the largest element in any S_i. For all S_i, let
$S'_i = \{a_i + (kA+1)f_i(a_i) \mid a_i \in S_i\}$ and let $t' = t + (kA+1)(u+jm)$. Notice that this maps invalid solutions to invalid solutions and correct solutions that satisfy (1) to correct solutions provided that j was guessed correctly. The k-Sum instances $S'_1, \ldots, S'_k$ with target t' are output, over all choices of s and j, for $k \log n = O(\log n)$ total instances.

If the original instance has a solution, then at least one guess of s is correct, and there is a constant probabability that there will be an output instance has at least one solution and at most $O(1)$ solutions. Otherwise, none of the instances will have solutions, as desired.

The algorithm takes $O(n \log n)$ time since there are $\log n$ values of s to guess and modifying every element takes linear time.

This completes the proof. □

We now inductively construct algorithms to solve k-Sum for increasing k, assuming that at least one and at most $O(1)$ solutions exist. For this proof, k will take on values one more than a triangular number.

Theorem 9. *For every integer $p \geq 0$, there exists a Monte Carlo algorithm that solves $(T_{p+1}+1)$-Sum on n numbers in $\tilde{O}(n^{T_p+1})$ time and $\tilde{O}(n)$ space, assuming that at least one solution and at most $O(1)$ solutions exist.*

Proof. It will be convenient to define a recursive function HashReduction that takes k sets $S_1, \ldots, S_k$ and a modulus m and finds up to *num* solutions (stopping with fewer if not that many solutions exist) to k-Sum in the modular setting.

We wish to show that Algorithm 3 meets the desired requirements when run with $k = T_{p+1} + 1$, $num = 1$, and m large enough to avoid wrap-around (begin with arithmetic over the integers).

Note that HashReduction makes calls to other functions, passing the images of S_i under some hash function h. It is assumed that these sets are implemented as vectors, and that the results can be returned as indices, so that the original elements can be recovered.

Algorithm 3. HashReduction$(k, S_1, \ldots, S_k, t, num, m)$

```
Require: $k = T_j + 1$ for some $j \geq 1$.
1: if $k = 2$ then
2:     Run the basic 2-Sum algorithm on $S_1$ and $S_2$, stopping at $num$ solutions.
3:     return
4: end if
5: Let $m' \leftarrow \Theta(n^{j-1})$.
6: Randomly choose a hash function $h \in H^{lin}_{m,m'}$.
7: for $v_\ell \in [m']$ do
8:     Initialize an empty lookup table $T$.
9:     Sort $h(S_1)$.
10:    for $a_2 \in S_2, \ldots, a_j \in S_j$ do
11:        Do a binary search for $v_\ell - \sum_{i=2}^{j} h(a_i) \pmod{m'}$ in $h(S_1)$.
12:        If a solution $(a_1, \ldots, a_j)$ is found, store it in $T$ as $(\sum_{i=1}^{j} a_i \pmod{m}) \rightarrow$
           $(a_1, \ldots, a_j)$, but store $\Theta(n)$ entries at most.
13:    end for
14:    for $sum \in \{h(t) - (k-1)h(0) + z \pmod{m'} \mid z \in \{-k+1, \ldots, k-1\}\}$ do
15:        Set $v_r \leftarrow sum - v_\ell \pmod{m'}$.
16:        Call HashReduction$(T_{j-1} + 1, h(S_{j+1}), \ldots, h(S_k), v_r, \Theta(n^{T_{j-2}+1}), m')$.
17:        For each solution $(a_{j+1}, \ldots, a_k)$, lookup $t - \sum_{i=j+1}^{k} a_i \pmod{m}$ in $T$.
18:        for entry $t - \sum_{i=j+1}^{k} a_i \pmod{m} \rightarrow (a_1, \ldots, a_j)$ in $T$ do
19:            Record $(a_1, \ldots, a_j, a_{j+1}, \ldots, a_k)$ as a solution.
20:            if $num$ solutions have been found then
21:                return
22:            end if
23:        end for
24:    end for
25: end for
```

Correctness: We begin by proving that HashReduction would run correctly if each call actually returned all solutions, not just the requested *num* at each recursive call. We will later show that it suffices to only return the requested number of solutions.

HashReduction divides the sets into two groups, left and right, and guesses the sum of a solution's hash values for each group. It relies on the almost affine-property of H^{lin} in order to reduce the number of cases it needs to guess.

Suppose there is a solution to the current HashReduction call, $a_1, \ldots, a_k$. If $k = 2$, then basic 2-Sum algorithm is called, which is a correct algorithm by Theorem 4. Otherwise, HashReduction chooses a hash and then runs its main for-loop.

In some iteration, v_ℓ is a correct guess for $\sum_{i=1}^{j} h(a_i)$. In this same iteration, $(\sum_{i=1}^{j} a_i) \to (a_1, \ldots, a_j)$ will be stored in T. By Lemma 2, $\sum_{i=1}^{k} h(a_i) \pmod{m'}$ must equal some value in the set $\{h(t) - (k-1)h(0) + z \pmod{m'} \mid z \in \{k-1, \ldots, k-1\}\}$. The algorithm guesses all possible values for this sum, and then computes what v_r must be to get this sum. Since $\sum_{i=1}^{k} a_i = t$, this solution will be correctly recorded by the algorithm and returned.

Next, we will show that it suffices to return only the requested number of solutions. Here, we will use the assumption that at most $O(1)$ solutions exist in the top level call. Fix some solution to the top level call, $a_1, a_2, \ldots, a_k$. Consider only the recursive branch where all v_ℓ and v_r are guessed correctly for this solution.

We claim that with probability arbitrarily close to 1, the value of *num* for every call along this branch is large enough to have all solutions returned. Since there are $O(1)$ function calls in this branch (this number depends only on the original value of k), it suffices to show this holds for any particular call.

At any recursive call, the current sets under consideration are a contiguous group of the original sets, $S_\ell, \ldots, S_r$, transformed by randomly chosen hash functions $h_1, h_2, \ldots, h_s$. We want to bound the probability of a false positive, i.e. some $b_\ell, \ldots, b_r$ such that $b_i \in S_i$ and

$$\sum_{i=\ell}^{r} (h_s \circ \cdots \circ h_1)(a_i) = \sum_{i=\ell}^{r} (h_s \circ \cdots \circ h_1)(b_i).$$

By Lemma 3, this probability has an upper bound of $\frac{O(1)}{\#\text{ values of } h_s}$ plus the probability of the event:

$$\sum_{i=\ell}^{r} (h_{s-1} \circ \cdots \circ h_1)(a_i) = \sum_{i=\ell}^{r} (h_{s-1} \circ \cdots \circ h_1)(b_i).$$

Repeating this s times gives an upper bound of $\sum_{i=1}^{s} \frac{O(1)}{\#\text{ values of } h_i}$ plus the probability of the event that:

$$\sum_{i=\ell}^{r} a_i = \sum_{i=\ell}^{r} b_i.$$

But any $b_\ell, \ldots, b_r$ for which this holds can be combined with the other a_i to make a different solution to the original top-level call: $(a_1, \ldots, a_{\ell-1}, b_\ell, \ldots, b_r, a_{r+1}, \ldots, a_k)$. Hence there are only $O(1)$ many $b_\ell, \ldots, b_r$ for which this event can occur. Ignoring these $O(1)$ solutions, the probability that any other $b_\ell, \ldots, b_r$ is a false positive is just $\frac{O(1)}{\#\text{ values of } h_s}$, since the number of possible hash values drops by a factor of roughly n in each recursive call. But the number of values of h_s was chosen to be some $m' = \Theta(n^{j-1})$. The algorithm should pick m' large enough to guarantee that non-solutions only have less than a $\frac{1}{n^{j-1}}$ probability of a false positive. It is then possible to use a Markov bound to pick a large enough value for $\Theta(n)$ and $\Theta(n^{T_{j-2}+1})$ to guarantee that with probability arbitrarily close to 1, the chosen value of *num* for any particular call along this solution branch is large enough.

Hence this fixed solution will eventually be found by the top level call, and the algorithm correctly finds some solution.

Running Time: We will inductively prove that for all $p \geq 0$, HashReduction takes $\tilde{O}(n^{T_p+1} + num)$ time when run with $k = T_{p+1} + 1$.

For the base case $p = 0$, $k = 2$ and the algorithm simply runs the basic 2-Sum algorithm. Theorem 4 guarantees it has the desired running time.

Assume that the inductive hypothesis holds for $p = q$. Consider when HashReduction is run with $k = T_{q+2} + 1$, $j = q + 2$. The algorithm chooses a hash function ($O(1)$ time) and then runs through $m' = O(n^{q+1})$ iterations of its main for-loop. Finding solutions to store in T takes $O(n^{T_q+1})$ time in all cases, since for all $q \geq 0$, $q + 1 \leq T_q + 1$. By the inductive hypothesis, calling HashReduction with $k = T_{q+1} + 1$ takes $\tilde{O}(n^{T_q+1})$ time.

Count the time taken to find and return solutions separately. Since only *num* solutions are requested, this requires $O(num)$ time. The total time for all iterations is hence $\tilde{O}(n^{T_{q+1}+1} + num)$, as desired. By induction, the hypothesis is true for all $p \geq 0$.

Memory Usage: Every recursive call uses $\tilde{O}(n)$ space for the lookup table T. The number of recursive calls depends only on k, not n, so the total memory usage is $\tilde{O}(n)$, as desired. □

Instances of the general k-Sum problem can be solved by preprocessing and then running this algorithm.

Corollary 5. *For a constant integer $p \geq 0$, there exists a Monte Carlo algorithm that solves* $(T_{p+1} + 1)$-Sum *on n numbers in $\tilde{O}(n^{T_p+1})$ time and $\tilde{O}(n)$ space.*

Algorithm 4. CompleteKSum($k, S_1, \ldots, S_k, t$)

```
Proof. 1: Preprocess $(S_1, \ldots, S_k, t)$ via the algorithm in Theorem 8.
2: for Resulting instances $(S'_1, \ldots, S'_k, t')$ do
3:     Run HashReduction on the instance to find a solution.
4: end for
```

By Theorem 8 and Theorem 9, Algorithm 4 has the desired properties. Notice that it has to run on $O(\log n)$ instances, but this is absorbed by the $\tilde{O}$ notation. $\square$

The following lemma produces algorithms for the remaining values of k:

Lemma 5. *Let A be an algorithm that solves k-*Sum *($k \geq 3$) on n numbers in $\tilde{O}(n^d)$ time and $\tilde{O}(n)$ space for some constant d. Then there is an algorithm A' that solves $(k+1)$-*Sum *on n numbers in $\tilde{O}(n^{d+1})$ time and $\tilde{O}(n)$ space.*

Proof. The algorithm A' is to guess one element $s \in S_{k+1}$ of the solution and then to run A on $S_1, \ldots, S_k$ for the remaining elements, which now need to sum to $t - s$. $\square$

Hence, for general k, we get the following Monte Carlo algorithm for k-Sum:

Reminder of Theorem 3. *There exists a Monte Carlo algorithm that solves k-*Sum *on n numbers in $\tilde{O}(n^{k-f(k)-1})$ time and $\tilde{O}(n)$ space, where $f(k)$ is the largest integer p such that $T_{p+1} + 1 \leq k$.*

Proof. This follows directly from Corollary 5 and Lemma 5. $\square$