

Today we will discuss algorithms for the k -SUM problem. In the following, let $k \geq 2$ be an integer.

Problem: k -SUM
Input: n integers (positive and negative)
Decide: Are there k (distinct) numbers in the input which sum to zero?

Sometimes people study k -SUM over the *real numbers*, in the Real RAM model. This is a more suitable model for computational geometry. For now, we'll look at the integer case; the real-valued case will come later.

We'll generally assume that our numbers are small enough that additions, comparisons, and subtractions of them take $O(1)$ time. (For example, if our computational model is the Word RAM, we could say that each number fits into one word.) We will use the notation k -SUM $_n$ to denote k -SUM instances with n numbers.

1 From k -SUM to $(k-1)$ -SUM

Let's start with some a simple (folklore) reduction from k -SUM to $(k-1)$ -SUM.¹

Theorem 1.1 *There is an $O(n^2)$ time reduction from k -SUM $_n$ to n instances of $(k-1)$ -SUM $_{n-1}$.*

Proof. Assume we have an oracle for solving $(k-1)$ -SUM $_n$, we design an algorithm for k -SUM. Given an instance $S = \{a_1, \dots, a_n\}$ of k -SUM $_n$, we do the following:

For all numbers x in S , make a new set S_x which does not contain x , and contains $(k-1) \cdot a_i + x$ for all $a_i \neq x$ in S .² Call $(k-1)$ -SUM on S_x . If some call returns "yes" then stop and return "yes". Otherwise, if no S_x has a $(k-1)$ -SUM, return "no".

Note the total overhead of the algorithm (not counting the cost of solving $(k-1)$ -SUM) is at most $O(n^2)$ time, assuming constant time additions: for each of the n numbers x , it takes $O(n)$ time to create the set S_x by adding and subtracting. (We could multiply each number in S by $(k-1)$ at the beginning, so that the only cost per number x is $O(n)$ additions and subtractions.)

Why does this algorithm work? We claim that there is a $(k-1)$ -SUM solution in S_x if and only if x is in a k -SUM solution of S . Suppose WLOG that $x = a_1$ and the k -SUM solution is a_1, \dots, a_k , so $\sum_i a_i = 0$. Then we have

$$S_x = \{(k-1)a_2 + a_1, \dots, (k-1)a_n + a_1\}.$$

The subset of $k-1$ numbers $(k-1)a_2 + a_1, \dots, (k-1)a_k + a_1$ has total sum

$$\sum_{i=2}^k ((k-1)a_i + a_1) = (k-1) \sum_{i=1}^k a_i = 0.$$

In the other direction, suppose S_x has a $(k-1)$ -SUM solution. This solution must be of the form $(k-1)a_{i_1} + x, \dots, (k-1)a_{i_{k-1}} + x$, where all $a_{i_j} \neq x$. Summing them up yields $(k-1)(x + a_{i_1} + \dots + a_{i_{k-1}}) = 0$, so $x, a_{i_1}, \dots, a_{i_{k-1}}$ is a k -SUM solution in S . \square

¹In math, a *folklore* result is something that was known by multiple parties and the reference is hard to track down. There might still be a reference, though...

²Note that, since the set of a_i s are distinct integers, the set of $(k-1) \cdot a_i + x$ are distinct integers.

2 Fast 2-SUM

Next, we show how to solve 2-SUM fast. (As far as we know, this is also folklore.)

Theorem 2.1 *2-SUM is in $O(n)$ time.*

Proof. Given a list L of n numbers, make a new list $L' = \{-a_i \mid a_i \in L\}$. Observe that there is a 2-SUM solution in L if and only if $L \cap L' \neq \emptyset$.

First, let's give an $O(n \log n)$ time algorithm: in $O(n \log n)$ time (assuming constant-time comparisons of integers), we can sort L' . Then for each $a_i \in L$, we can binary search to determine if $-a_i$ is in L' , in $O(\log n)$ time (again, assuming constant-time comparisons of integers).

This running time can be improved to $O(n)$, by using hash functions and word tricks. In particular, if:

- Each number can be stored in a word,
- we can populate a hash table of $O(n^2)$ size with $O(n)$ elements in $O(n)$ time, such that none of the entries of the table include some special character, and
- we can randomly access any entry of a table in $O(1)$ time,

then we can get an $O(n)$ -time randomized algorithm.

Here are more details. *Suppose all numbers in our list L have m -bit representations, so we can think of each number as a bit vector of length m , where the first bit is the sign of the integer.* Then, we want to find two vectors in L whose first bits differ but all other bits are the same.

Our hash functions will be constructed from $t \times m$ matrices $M \in \{0, 1\}^{t \times m}$, for $t = 10 + 2 \log(n)$. In particular, pick a uniform random such M , and define

$$h_+(x) := (M \cdot x) \bmod 2, \quad h_-(x) = (M \cdot (-x)) \bmod 2,$$

where $-x$ is the same as x but the first bit is flipped. So, both hash functions are hashing an m -bit vector x down to a t -bit vector.

Exercise: Prove that for all m -bit vectors $x \neq y$, $\Pr[M \cdot x = M \cdot y \bmod 2] \leq 1/2^t$.

Now suppose there's a 2-SUM solution a_1, a_2 in L . Then $a_1 = -a_2$, and we know that $h_+(a_1) = h_-(a_2)$. Conversely, if there's no 2-SUM solution in L , then by the Exercise and the Union Bound,

$$\Pr[(\exists a_1, a_2 \in L) h_+(a_1) = h_-(a_2)] \leq n^2/2^t \leq 1/2^{10}.$$

Thus, to get an algorithm with high success probability, it suffices for us to determine if there are a_1, a_2 such that $h_+(a_1) = h_-(a_2)$. To this end, we make lists L' and L'' of vectors from $\{0, 1\}^t$, where

$$L' = \{h_+(a_i) \mid a_i \in L\} \text{ and } L'' = \{h_-(a_i) \mid a_i \in L\},$$

and we want to determine if $L' \cap L''$ is empty or not. Consider a hash table T of $2^t \leq O(n^2)$ size, indexed by vectors $\{0, 1\}^t$. Go through each $v \in L''$, and mark $T[v]$ with a special character not appearing in the table (we could also use another hash function to choose this character, randomly). Finally, we output "yes" if and only if there is some $u \in L'$ such that $T[u]$ contains the special character. Assuming we can access any entry of T in $O(1)$ time, and we can populate a hash table of $O(n^2)$ size with $O(n)$ elements in $O(n)$ time, this algorithm runs in $O(n)$ time. \square

3 3-SUM and k-SUM Algorithms

Theorem 3.1 *3-SUM is in $O(n^2)$ time.*

We can get a randomized 3-SUM algorithm running in $O(n^2)$ time, by simply combining the 2-SUM algorithm and the reduction from 3-SUM to 2-SUM, both of which were already given above. But we can get a deterministic algorithm by using another strategy based on sorting.

Proof. Given a list L of n numbers, here is an algorithm:

Sort L in $O(n \log n)$ time.
 For each a in L ,
 Make two pointers on the sorted list L : p_1 at the beginning of L , and p_2 at the end.
 Repeat until the pointers reach each other:
 Let b be the current number at p_1 and c be the number at p_2 .
 If $a = b$, move p_1 to the right (*we want a distinct triple of numbers*)
 If $a = c$, move p_2 to the left (*same reason*)
 If $a + b + c = 0$ then return (a, b, c) .
 If $a + b + c > 0$, then move p_2 to the left (*to get a smaller 3-sum, we have to decrease c*)
 If $a + b + c < 0$, then move p_1 to the right (*to get a larger 3-sum, we have to increase b*)
 Return “no solution”.

Exercise: Why is this algorithm correct? (Intuitively, if a is in a 3-SUM, then b is the smaller number and c will be the larger number such that $a + b + c = 0$.)

For every a in L , observe that the repeat loop takes $O(n)$ time to find the (b, c) pair, if it exists. (For a fixed a , the total number of times that a pointer moves is at most n , since we quit if the two pointers reach each other.) Therefore the algorithm runs in $O(n^2)$ time. \square

In general, the k -SUM problem can be reduced to 2-SUM, as follows:

Theorem 3.2 *Let $k \geq 2$. There is an $O(n^{\lceil k/2 \rceil})$ -time reduction from k -SUM on n numbers to 2-SUM on $O(n^{\lceil k/2 \rceil})$ numbers.*

Proof. WLOG, we may assume that the instance of k -SUM has k parts, where we want to pick exactly one number from each part such that the k numbers sum to zero. (The setting of $k = 3$ was called “Colorful 3-SUM” on your problem set.) We enumerate all $O(n^{\lceil k/2 \rceil})$ choices of $\lfloor k/2 \rfloor$ numbers, one number from each of the first $\lfloor k/2 \rfloor$ parts of the instance, forming a list

$$L = \left\{ \sum_i a_i \mid a_i \text{ is in part } i, \text{ for all } i = 1, \dots, \lfloor k/2 \rfloor \right\}.$$

Similarly, for all $O(n^{\lceil k/2 \rceil})$ choices from the last $\lceil k/2 \rceil$ parts of the instance, form a list

$$L' = \left\{ \sum_i a_i \mid a_i \text{ is in part } \lfloor k/2 \rfloor + i, \text{ for all } i = 1, \dots, \lceil k/2 \rceil \right\}.$$

Now there is a k -sum in the original instance if and only if there is a number in L and a number in L' which sum to zero; the latter is equivalent to 2-SUM. \square

Corollary 3.1 4-SUM is in $O(n^2)$ time, and $k\text{-SUM}$ is in $O(n^{\lceil k/2 \rceil})$ time.

A popular conjecture in fine-grained complexity is that this running time for $k\text{-SUM}$ cannot be improved:

k-SUM Conjecture: For every $k \geq 2$ and $\varepsilon > 0$, $k\text{-SUM}$ cannot be solved in $O(n^{\lceil k/2 \rceil - \varepsilon})$ (randomized) time.

Note this implies that for **odd values of k** , $k\text{-SUM}$ and $(k + 1)\text{-SUM}$ have essentially the same time complexity. On the one hand, the conjecture seems rather strong. You can use it to prove strong lower bounds for many other problems (some examples are [AL13, ALW14, ABHS19]; see [VW15] for a bunch of references on the 3-SUM Conjecture itself). We will see some of these consequences over the next few lectures! On the other hand, we don't really know good algorithmic improvements to solving $k\text{-SUM}$, beyond small log factors, so maybe the conjecture is reasonable...

4 A Faster Algorithm for 3-SUM

In the last part of the lecture, we'll show one way to get an algorithm for 3-SUM running in $o(n^2)$ time. Unlike OV and APSP, the current best known algorithms for 3-SUM only get polylogarithmic improvements over the "easy" running time. (We don't know how to apply the polynomial method to solve 3-SUM faster!) Baran, Demaine, and Patrascu [BDP08] gave a 3-SUM algorithm running in $n^2 \cdot \text{poly}(\log \log n) / (\log^2 n)$ time, which is essentially the best known running time for 3-SUM over the integers. (For the real-valued version of 3-SUM, there are other references with similar log-speedups but very different techniques, starting with the work of [GP14].)

Below is an alternative (unpublished) algorithm for 3-SUM, applying some work of Andrea Lincoln, Joshua Wang, and your two instructors [LVWW16]. (That paper shows there is a deterministic algorithm for 3-SUM running in $O(n^2(\log \log n) / \log n)$ time and $\tilde{O}(\sqrt{n})$ space.) Assuming we can do $O(1)$ -time lookups into tables, the algorithm we give below runs in $O(n^2 \cdot (\log \log n)^2 / \log^2 n)$ randomized time, matching the best known bounds.

There are roughly three parts to our algorithm:

1. **A self-reduction for 3-SUM.** Roughly speaking, for an integer parameter s , we can reduce 3-SUM on n numbers to $O(n^2/s^2)$ instances of 3-SUM on at most $3s$ numbers. This statement is very similar in spirit to the self-reduction we gave for OV, which was used in the OV algorithm. (However, the actual reduction is very different from the OV one.)
2. **A randomized reduction for 3-SUM.** Given that we can reduce the instances to be "small", of size $O(s)$, a randomized reduction will let us reduce the *sizes of the numbers* in the small instances, by working modulo a random prime.
3. **Fast look-up table.** Once the instances are small *and* the numbers are small, we can store the answers to all small instances on small numbers in a look-up table, to solve them quickly.

We have deliberately taken this route to obtaining a 3-SUM algorithm, so that it can be compared and contrasted with the OV algorithm. In the OV algorithm, we also ran a self-reduction reducing OV on n vectors to $O(n^2/s^2)$ instances of OV on $2s$ vectors, but then we used probabilistic polynomials and matrix multiplication to show how to solve all those OV instances simultaneously in $\tilde{O}(n^2/s^2)$ time, for decent sized s (when the dimensionality was $O(\log n)$, we could set $s = n^\varepsilon$ for a tiny $\varepsilon > 0$). In the case of 3-SUM, we don't know how to get a good-enough polynomial to solve all the $O(n^2/s^2)$ instances quickly, so instead we choose s to be much smaller, like $\text{poly}(\log n)$: small enough that we can store all possible instances we might need to solve into a $\text{poly}(n)$ -sized look-up table.

Let's now go through the three parts in turn.

4.1 Self-Reduction

Lincoln *et al* [LVWW16] give a deterministic $O(n \log n + n^2/s^2)$ -time reduction from 3-SUM on n numbers to $O(n^2/s^2)$ instances of 3-SUM on $3s$ numbers. Such a reduction was quite easy to do for OV, but is highly nontrivial to do for 3-SUM! This self-reduction works in the Real RAM as well (where registers can hold real numbers, which we can do additions and comparisons on, in unit time).

Here we just sketch how the reduction goes, and why it works. Start by sorting the n numbers in $O(n \log n)$ time. Then, partition the sorted order into (n/s) contiguous “chunks” of s numbers each. There are (n^3/s^3) triples of chunks (each corresponding to a set of at most $3s$ numbers), but one can prove that there are at most $O(n^2/s^2)$ triples of chunks that could possibly contain a 3-SUM solution. (This is subtle, and applies Dilworth’s theorem in an interesting way. See the paper if you’re interested!) Moreover, we can calculate which triples could possibly contain a 3-SUM solution in $O(n^2/s^2)$ time.

4.2 Randomized Reduction

We want to show that there is a randomized reduction from the 3-SUM problem on s numbers to the 3-SUM problem on s numbers modulo a “small” prime. The following theorem shows how to hash any set S of m -bit integers into $O(\log |S| + \log \log \log m)$ -bit integers modulo a prime, in a way that preserves 3-SUM solutions in S with high probability.

In the following, we’ll use the notation $[n] = \{-n, -n+1, \dots, 0, 1, \dots, n\}$ which is a little non-standard.

Theorem 4.1 *For all positive integers m , suppose we choose a random prime p in the interval $\{2, \dots, s^7 \cdot m\}$. Then for every set S of $3s$ numbers in $[2^m]$,*

- If S has a 3-SUM, then $\Pr_p[S \text{ has a 3-SUM solution modulo } p] = 1$.
- If S doesn’t have a 3-SUM, then $\Pr_p[S \text{ has a 3-SUM solution modulo } p] \leq O(\log m + \log s)/s^4$.

Proof. Let p be a randomly chosen prime from $[2, 2^t]$, for a parameter $t := 7 \log(s) + \log(m)$. For every triple (a, b, c) of numbers from $[2^m]^3$, we have:

- If $a + b + c = 0$ then $a + b + c = 0 \pmod p$.
- If $a + b + c \neq 0$, then $a + b + c \leq 3 \cdot 2^m$ has at most $O(m)$ prime factors. The prime number theorem tells us that there are at least $\Omega(2^t/t)$ primes in the interval $[2, 2^t]$. Putting these two facts together,

$$\Pr_p[a + b + c = 0 \pmod p] \leq O(mt/2^t).$$

Fix any set S of $3s$ numbers. As there are $O(s^3)$ triples of numbers from S , the Union Bound says

$$\Pr_p[(\exists a, b, c \in S) a + b + c \neq 0 \text{ but } a + b + c = 0 \pmod p] \leq O(mts^3/2^t).$$

This is the probability that S has a 3-SUM solution modulo p , but S doesn’t have a 3-SUM solution. Finally, when $t = 7 \log(s) + \log(m)$, note that the error is at most $O(\log m + \log s)/s^4$. \square

Note that if we had *real-valued* inputs (and worked over the real RAM) then the above reduction wouldn’t work at all!

By building on the above reduction, we can essentially show that, WLOG, we can assume the 3-SUM problem on n integers contains only numbers in $\{-n^{O(1)}, \dots, n^{O(1)}\}$: there is a randomized reduction from 3-SUM on n numbers of m -bits to n numbers in $\{-n^{O(1)}f(m), \dots, n^{O(1)}f(m)\}$ where $f(m)$ is an extremely slow-growing function of m . In Theorem 4.1, we started with 3-SUM over integers and ended with 3-SUM modulo a prime number. But the “modulo prime” case can actually be reduced back to the small integer case. The idea is that we think of every number

in \mathbb{Z}_p as an integer in $\{0, 1, \dots, p-1\}$, and check if there are three numbers summing to p , if there are three summing to $2p$, or if there are three summing to 0. The total sum of any triple is less than $3p$, so these three checks cover all the possible cases over the integers! After having reduced the case of m -bit numbers to $O(\log s + \log m)$ -bit numbers, we can apply the reduction *again*, yielding $O(\log s + \log \log m)$ -bit numbers, and we can keep repeating the reduction as necessary to reduce the dependence on m .

From here on, we will assume that the random prime p chosen above in Theorem 4.1 is at most $p \leq s^k$ for some constant k . Since in Theorem 4.1 we actually need $p \leq s^7 \cdot m$, this means we are assuming that the *number of bits* m used to encode each number in our original 3-SUM instance is at most $\text{poly}(s)$. As we will eventually set $s \leq O(\log n / \log \log n)$, we are effectively assuming that our original n numbers are in the interval $[-2^{\text{poly}(\log n)}, 2^{\text{poly}(\log n)}]$. By the previous paragraph, this is (basically) without loss of generality.

4.3 Fast Lookup Table

Let us cite the lookup table fact that we'll need; it's very simple.

Fact 4.1 *For any prime $p \leq s^{O(1)}$, there is a data structure of size $s^{O(s)}$ that can answer any 3-SUM instance on s numbers, modulo p .*

Proof. There are at most $s^{O(s)}$ sequences of s numbers in $\{0, 1, \dots, p-1\}$. Write down all their yes/no answers for 3-SUM modulo p , one by one, and store the answers in a table of $s^{O(s)}$ bits. \square

4.4 The Final Algorithm

We are now ready to give our final 3-SUM algorithm. Let $s \in \{1, \dots, n\}$ be a parameter. The algorithm works as follows. Let L be a given list of n numbers.

3-SUM Algorithm:

0. Pick a random prime $p \leq s^k$, and construct an $s^{O(s)}$ -size lookup table for 3-SUM on s numbers modulo p , as in Fact 4.1.
1. Compute all n numbers in L modulo p , mapping them to the domain $\{0, 1, \dots, p-1\}$.
(Note that by Theorem 4.1, for every subset S of $3s$ numbers, there is probability less than $1/s^3$ that the reduction modulo p created an erroneous 3-SUM solution in S .)
2. Run the 3-SUM self-reduction. For each of the $O(n^2/s^2)$ calls to 3-SUM on $3s$ numbers, consult the lookup table for the answer.
(Note for each call to the self-reduction, the lookup table returns the correct yes/no answer with probability at least $1-1/s^3$. So we expect at most a $1/s^3$ -fraction of the answers to our $O(n^2/s^2)$ calls to be incorrect.)
3. If more than $100 \cdot n^2/s^5$ of the lookup table calls report “yes”, then **return “yes”**.
(If we were given a “no” instance, we would expect at most n^2/s^5 calls to say “yes”.)
4. Otherwise, less than $100 \cdot n^2/s^5$ calls report “yes”. In this case, we can directly check all of the yes calls for a 3-SUM solution: for all of the $O(n^2/s^5)$ “yes” calls, search all of the relevant subsets of $O(s)$ numbers directly for a 3-sum. This takes in $O(s^2 \cdot n^2/s^5) \leq O(n^2/s^3)$ time. **Return “no”** if no 3-sum solution is found, and **“yes”** otherwise.

Exercise: Prove that this algorithm outputs a correct yes or no answer with probability greater than $2/3$. The parenthetical remarks in the pseudocode should help!

Assume that the cost of lookup in a table of size T takes time $L(T)$. Typically, either $L(T) \leq O(\log T)$, or $L(T) \leq O(1)$. The total running time of the above algorithm can be calculated as follows:

- Step 0 needs $s^{O(s)}$ time, to set up the lookup table.
- Step 1 needs $\tilde{O}(n)$ time.
- Step 2 takes $O(n^2/s^2)$ time, to generate $O(n^2/s^2)$ calls to a lookup table. Each lookup costs $L(s^{O(s)})$ time.
- Step 3 is negligible, we just need a counter for that.
- As mentioned above, Step 4 takes $O(n^2/s^3)$ time.

In order for the algorithm to run in subquadratic time, we need $s^{O(s)} \ll n^2/(\log^2 n)$. Setting

$$s := \varepsilon(\log n)/(\log \log n)$$

for small enough $\varepsilon > 0$, we will accomplish that. The total running time is then upper-bounded by step 2, which becomes

$$\frac{n^2(\log \log n)^2}{(\log n)^2} \cdot L(n^{O(\varepsilon)}).$$

Now, the running time improvement depends on how fast we can look up stuff in our table. If $L(T) \leq O(\log T)$, we have $L(n^{O(\varepsilon)}) \leq O(\varepsilon \log n)$, so the running time is $O(n^2(\log \log n)^2/\log n)$. If $L(T) \leq O(1)$, we save a $(\log n)^2$ factor.

5 Open Problem

The following seems to still be open: *Is there an $o(n^2)$ -time algorithm for 4-SUM?* The problem with the above algorithm is that 3-SUM self-reduction above does not generalize nicely to 4-SUM: for 4-SUM, there will be $O(n^3/s^3)$ 4-tuples of chunks in the self-reduction.

References

- [ABHS19] Amir Abboud, Karl Bringmann, Danny Hermelin, and Dvir Shabtay. SETH-based lower bounds for subset sum and bicriteria path. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 41–57. SIAM, 2019.
- [AL13] Amir Abboud and Kevin Lewi. Exact weight subgraphs and the k-Sum conjecture. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, volume 7965 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2013.
- [ALW14] Amir Abboud, Kevin Lewi, and Ryan Williams. Losing weight by gaining edges. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, volume 8737 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2014.
- [BDP08] Ilya Baran, Erik D. Demaine, and Mihai Patrascu. Subquadratic algorithms for 3sum. *Algorithmica*, 50(4):584–596, 2008.

- [GP14] Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 621–630. IEEE, 2014.
- [LVWW16] Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. Deterministic time-space trade-offs for k-SUM. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPICs*, pages 58:1–58:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
- [VW15] Virginia Vassilevska Williams. Hardness of easy problems: Basing hardness on popular conjectures such as the strong exponential time hypothesis (invited talk). In *10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.