A 3SUM algorithm on the binary addition level of atomic generated real numbers

Carolin Zöbelein[*]

September 6, 2020

DRAFT

Abstract

We present a simple algorithm for the `3SUM` problem for a list L of n real numbers. For this, we generate real numbers with the help of a set $\mathbb{I}_a$ of so called 'Atomic irrational numbers'. We can show that our algorithm, which use the binary representation of integers, takes time complexity $\mathcal{O}\left(nb + U^{1.58}\right)$ for an universe of $U := 2^b$ distinguishable rational numbers and with our so generated real numbers it takes a time complexity of $\mathcal{O}\left(nb|\mathbb{I}_a| + U_{\mathbb{I}_a}^{1.58}\right)$ on an universe of $U_{\mathbb{I}_a} := 2^{b|\mathbb{I}_a|}$ distinguishable real numbers.

Keywords: 3SUM Problem, Real Numbers, Irrational Numbers, Data Structures, Binary Representation, Data Storage Representation, Nonnumerical Algorithm, Complexity
ACM Subject Classes: E1, E2, F1.3, F2.2

Preamble

The following content is a sketch for discussion purposes only, without warranty for mathematical completeness.

1 Introduction

We consider the `3SUM` problem and its limited to only integers version `Int3SUM`, a variant of this, one of the most basic problems in algorithm design, which are defined as follows:

Definition 1 (`3SUM`)**.** *Given a set $S \subset \mathbb{R}$, $|S| := n$, determine if there exists $n_1, n_2, n_3 \in S$ such that $n_1 + n_2 + n_3 = 0$.*

Definition 2 (`Int3SUM`)**.** *Given a set $S \subseteq \{-U, \ldots, U\} \subset \mathbb{Z}$, $|S| := n$, determine if there exists $z_1, z_2, z_3 \in S$ such that $z_1 + z_2 + z_3 = 0$.*

Since numerous problems can be reduced from `3SUM` [11] [3], it has raised wide interest. Such problems which are at least as hard as `3SUM` are called *3SUM-Hardness*. Some of them are ...

- Given a n-point set in the plane, determine if it contains three collinear points. It was shown by Gajentaan and Overmars [11] that it can be solved in $\mathcal{O}\left(n^2\right)$ time.

[*]The author believes in the importance of the independence of research and is funded by the public community. If you also believe in this values, you can find ways for supporting the author's work here: `https://research.carolin-zoebelein.de/funding.html`, Email: `contact@carolin-zoebelein.de`, PGP: D4A7 35E8 D47F 801F 2CF6 2BA7 927A FD3C DE47 E13B, `https://research.carolin-zoebelein.de` LaTeXsource availabe at `https://github.com/Samdney/paper-3sum`, `id: paper_0003`, ☺

- Given a set of n triangles in the plane, determine if their union contains a hole or not, respectively compute the area of their union which can be also solved in $\mathcal{O}\left(n^2\right)$ time [11].
- Given two n-poin sets $X, Y \subset \mathbb{R}$, each of size n, determine if all elements in $X + Y = \{x + y \,|\, x \in X, y \in Y\}$ are distinct. It can be solved in $\mathcal{O}\left(n^2 \log(n)\right)$ time [3]. This problem, including its stronger version with sorting $X + Y$, is used for the conditional lower bounds of problems by Barequet and Har-Peled in [3] and are classified as *Sorting $X + Y$-Hard.* It's an open question if it can be solve in $o\left(n^2 \log(n)\right)$ [5] [6] [7].
- Given two n-edge convex polygons, determine if one can be placed inside the other by translation and rotation which can also be solved in $\mathcal{O}\left(n^2 \log(n)\right)$ time [3].
- ... .

So, it turns out that examinations of `3SUM` are a starting point for wide studies of problem statements in computational geometry.

1.1 Related Work

The lower bound of computational complexity of `3SUM` is a long standing unsolved question. Over the years, work has been done and several results were achieved.

- A trivial hash table algorithm can solve `3SUM` in $\mathcal{O}\left(n^2\right)$ time.
  All elements $s \in S$ are hashed into a table T_h by $h(s)$. Then we go for all $(h(s_i) + h(s_j))$ through the hash table and check for $-(h(s_i) + h(s_j))$.
- Erickson [9] credited Seidel [8] that `Int3SUM` can be solved with fast Fourier transform in $\mathcal{O}(n + U \log(U))$ time, for a universe size U (an integer range $[-U, \ldots, U]$) [14].
- Baran, Demaine and Pătraşcu [2] showed that `Int3SUM` can be solve in $\mathcal{O}\left(n^2 / (\log(n) / \log\log(n))^2\right)$ time with high probability on the word RAM with $U = 2^\omega$ and $\omega > \log(n)$ is the machine word size. It uses a mixture of randomized universe reduction by hashing, word packing and table lookups.
- Erickson [9] and Ailon and Chazelle [1] proved that any 3-linear decision tree for `3SUM` has depth $\Omega\left(n^2\right)$ and matching $\Omega\left(n^{\lceil k/2 \rceil}\right)$ lower bounds in some particular models.
- Grønlund and Pettie [13] showed that `3SUM` can be solved in subquadratic time by a deterministic algorithm which runs in $\mathcal{O}\left(n^2 (\log(n) / \log\log(n))^{2/3}\right)$ time and a randomized algorithm which runs in $\mathcal{O}\left(n^2 (\log\log(n))^2 / \log(n)\right)$ expected time with high probability. They also showed that it exists a 4-linear decision tree with deph $\mathcal{O}\left(n^{3/2}\sqrt{\log(n)}\right)$.
- Freund [10] developed a `3SUM` algorithm by eliminating one of Grønlund and Pettie's conceptual building blocks and lead to a bound for running time of $\mathcal{O}\left(n^2 \log\log(n) / \log(n)\right)$.
- Gold and Sharir [12] showed that the randomized 4-linear decision tree complexity of `3SUM` is $\mathcal{O}\left(n^{3/2}\right)$.
- Chan [4] gave an algorithm that solved `3SUM` in $\mathcal{O}\left(\left(n^2 / \log(n)\right)(\log\log(n))^{\mathcal{O}(1)}\right)$ time and obtained subquadratic results on some `3SUM`-hard problems.
- Kane, Lovett, and Moran [18] constructed a linear decision tree that solves the `3SUM` problem with $\mathcal{O}\left(n \log^2(n)\right)$ queries which compare the sums of two 3-subsets.
- Kopelowitz, Pettie and Porat [19] conjectured that `3SUM` is unsolvable in $\mathcal{O}\left(n^{2-\omega(1)}\right)$ expected time.

In this work, we want to present a new kind of algorithm design, different compared to the mentioned works. We can see, that apart from Seidel's [8] Fast Fourier transformation and Baran, Demaine and Pătraşcu's [2] work which used randomized universe reducation, word packing and table lookups, for `Int3SUM`, these works are all mainly based on decision trees and queries depending on the numbers of S in decimal representation. We not want to do this kind of approach again. Instead, we will use a new angle of view based on the binary representation of the numbers of S and the characteristics of binary addition.

1.2 Our Results

In this work, we start with the basic idea of using an algorithm for the `3SUM` problem on the binary representation level of numbers in a computer system, which means using the nature of binary number addition on a bit level instead of comparing the numbers in their total representation.

The first problem which raises are the set of irrational numbers, since their can't be represented exactly by a finite number of bits. We show that we can solve this problem by a mapping of irrational numbers to a zero sum question of their rational factors, instead. To reach this, we introduce a concatenation representation of real numbers n_i, based on a finite set $\mathbb{I}_a$ of so called *Atomic irrational numbers.*

Since, we see that we are able to solve the real number problem by solving the problem for rational numbers, we examine existing representations like *Floating-point arithmetic* and *Fixed-point arithemetic.* Since, both of this representations would raise the complexity of our algorithm idea exponentially, we introduce so called *Fixed-point integer arithmetic.* This way of representation of a rational number consists simple by represent the pre radix point part of our number by one integer and the post radix point part of our number by a second additional integer. Although, obviously standard binary addition for the whole number in this way is no longer possible, this way of representation gives us the possiblity to save space and time.

We examine the properties of the zero sum addition of three integer numbers and receive a set of rules $\mathcal{C}$ which tells us how much 0s and how much 1s we need, by a given carrier $c[q]$ at bit position q, to receive a zero sum bit at position q. Since, the successors of such rules turn out to be unique, we can take this rules to determine valid soulutions.

For this we define two kinds of binary trees: a list entries tree T which contains the given list numbers n_i and string trees T^{string} which constains the given rules and the final solutions after $b+1$ bit depth steps. By connecting this two tree kinds we receive for the `3SUM` problem of a list L of n elements, a time complexity of $\mathcal{O}\left(nb + U^{1.58}\right)$ and space complexity of $\mathcal{O}_{Space}\left(nb + U^{1.58}\right)$ for an universe of $U := 2^b$ distinguishable rational numbers and $\mathcal{O}\left(nb|\mathbb{I}_a| + U_{\mathbb{I}_a}^{1.58}\right)$ on an universe of $U_{\mathbb{I}_a} := 2^{b|\mathbb{I}_a|}$ distinguishable real numbers n_i with a space complexity $\mathcal{O}_{Space}\left(nb|\mathbb{I}_a| + U_{\mathbb{I}_a}^{1.58}\right)$, too.

1.3 Organization

Our paper is organized as follows. In section 2, we start with giving some necessary basic definitions and notations. In section 3, we discuss basic properties of real numbers, the necessities of a particular way of irrational number representation and deriving from this, we introduce *Atomic irrational numbers.* Then, we determine the characteristics of zero sum addition on a binary representation level of numbers, to derive a set of rules, which we use for our `3SUM` algorithm in section 4. Finally, in section 5 we discuss the belonging complexities of our presented algorithm.

2 Preliminaries

We give some definitions and notation agreements. Since our algorithm will work on the binary representation level of numbers, we will repeat the basic properties of numbers, addition and bit manipulation in binary numeral system.

Definition 3 (Positive integers)**.** *Each natural number $z^+ \in \mathbb{Z}_0^+$ can be described by the bit string $B_{z^+} := \left(0, b_{[N-2]}, b_{[N-3]}, \cdots, b_{[1]}, b_{[0]}\right)$ with size $N := \lceil \log_2\left(z^+\right)\rceil$ given by the binary representation of a decimal number $z^+ := \sum_{q=0}^{N-1} b_{[q]}2^q$ with $b_{[q]} \in \{0,1\}$.*

Definition 4 (Bit toggle(b))**.** *The toggle function* $\operatorname{toggle}(B)$ *inverts the bits* $b_{[q]}$ *of the bit string* B *bitwise by* $0 \mapsto 1$ *and* $1 \mapsto 0$*.*

Definition 5 (Two's complement B^{-1})**.** *The two's complement* B^{-1} *of a bit string* B *with highest bit index* $N-1$ *is defined as its complement of* B *so that* $B + B^{-1} = 2^N - 1$*. The two's complement* B^{-1} *of* B *can be determined by* $B^{-1} = \operatorname{toggle}(B) + 1$ *and the two's complement* B *of* B^{-1} *by* $B = \operatorname{toggle}\left(B^{-1}\right) + 1$*.*

Definition 6 (Negative integers)**.** *Each negative integer number* $z^- \in \mathbb{Z}^-$ *can be described by the bit string* $B_{z^-} := \left(1, b_{[N-2]}, b_{[N-3]}, \cdots, b_{[1]} b_{[0]}\right)$ *with size* $N := \lceil \log_2 (|z^-|) \rceil$ *given by the binary representation of the belonging positive decimal number by* $z^- := (-1)\, z^+$*. Negative integers are represented by the two's complement in binary representation.*

Definition 7 (Integer binary addition)**.** *The addition of integers* z_i *in their binary representation is given by a bitwise addition in base two of their bit strings* B_{z_i} *with carriers* $c_{[q]} \in \{0,1\}$*,* $q = 0, \dots N-1$*.*

3 Zero Sum Bit Addition

In this paper, we want to introduce a binary representation based algorithm. Mostly, if people talk about binary algorithms, they mean two decision algorithms, like binary trees, which divide possible solutions sets into two subsets. We want explicitly focuse on the binary representation of integers and real numbers in typical computer designs, today. Regarding the `3SUM` problem we are particularly interested in the zero sum bit addition of three numbers. We look at this problem statement for integers and for the set of real numbers to deduce a set of constraint rules. Let's start with a general treatment of the set of real numbers, at first.

3.1 Real Numbers

Considering a set of given real numbers $n \in \mathbb{R}$, we have to differ between several kinds of numbers, which are integers $z \in \mathbb{Z}$, rational $q \in \mathbb{Q}$ and irrational $i \in \mathbb{R} \setminus \mathbb{Q}$ numbers. Since integers are a subset of rational numbers which can be written by $z/1$, we only have to differ in practice between rational and irrational numbers, regarding the problem of finite precision of representation of a number on a computer system and the question how we can decide if the sum of three numbers of possible irrational cases is zero or not.

At first, we need to take a short look at the character of irrational numbers, in generally. Irrational numbers are numbers which can not be presented by a fraction $\frac{z_1}{z_2}$, $z_1, z_2 \in \mathbb{Z}$. The most known irrational numbers are π, e but also $\sqrt{2}$. Although this numbers play an important role, there are still some unsolved questions regarding them. At first it is not know if for all cases of $i_1 i_2$ with $i_1, i_1 \in \mathbb{R} \setminus \mathbb{Q}$, the product is still also an irrational number. Until now, there couldn't be found a counterexample but also no general proof that all cases are leading to irrational numbers [20]. As second, it is also not known if the linear combination of two irrational numbers by $q_1 i_1 + q_2 i_2$ with $q_1, q_2 \in \mathbb{Q}$ is always an irrational number [20] [21], too. Finally, it exists just an assumption but no proof for the irrational nature of the numbers 2^e, π^e, $\pi^{\sqrt{2}}$, π^π, e^e and γ (the Euler–Mascheroni constant) [20]. Because of this unsolved questions, we want to introdude irrational numbers regarding the `3SUM` problem in a specific way. For this, we introduce `Atomic irrational numbers`.

Definition 8 (Atomic irrational numbers)**.** *Given a finite set* $\mathbb{I}_a$ *of irrational numbers* $i_a \in \mathbb{R} \setminus \mathbb{Q}$*. We call this numbers* ***Atomic irrational numbers*** *if they statisfy all of the following properties:*

1. *For all* $i_{a,1}, i_{a,2} \in \mathbb{I}_a$*, we have* $i_{a,2} \neq q i_{a,2}$*,* $\forall q \in \mathbb{Q}$*.*
2. *For all* $i_{a,1}, i_{a,2}, i_{a,3} \in \mathbb{I}_a$*, we have* $i_{a,3} \neq q_1 i_1 + q_2 i_2$*,* $\forall q_1, q_2 \in \mathbb{Q}$*.*
3. *We are in the know of the properties* $q_1 i_{a,1} + q_2 i_{a,2} = q_{a,3} \in \mathbb{Q}$ *for all* $i_{a,1}, i_{a,2}, i_{a,3} \in \mathbb{I}_a$ *if such cases exist. Means we know the complete tuples* $(q_1, i_{a,1}, q_2, i_{a,2}, q_3)$*.*

Now, we want to take attention on the possibility to present a real number n itself by a given combination of several single numbers, too. Following from this, we can write for a complete description of possible cases of a real number $n = q_1q_2 + q_3i_1 + i_2i_3$, with $q_j \in \mathbb{Q}$ and $i_j \in \mathbb{R} \setminus \mathbb{Q}$.

Let's look at the sum of the three real numbers n_1, n_2 and n_3 with keeping this description in mind. We know that an irrational number i can only disappear as the result of the addition of several numbers n, if this number i respectively its multiples are element of some of this numbers n. An irrational number i_1 can not be eliminated by neither an other irrational number $i_2 \neq i_1$ and their multiples nor by any rational number. In addition we know that $q_1q_2, \in \mathbb{Q}$ and $q_3i_1 \in \mathbb{R} \setminus \mathbb{Q}$. If i_2 and $i_3 \in \mathbb{I}_a$, we also know that $i_2i_3 \in \mathbb{R} \setminus \mathbb{Q}$. This leads to the point that for further steps of solving the 3SUM problem, we have to split n into two parts: the one with the rational summands q_1q_2 and the one with the irrational summands q_3i_1 and i_2i_3. We see for the irrational summands that we either have again the product of several irrational numbers or the product of one irrational and one rational number. Hence, in generally we can write for the three sum of this kind of summands $i_ii_j \cdots i_ki_l\,(q_1 + q_2 + q_3)$ and we see, that if we are able to identify and flag an irrational number clearly as irrational, it becomes not necessary to save its specific value and our zero sum problem can be reduced to a rational zero sum problem for this specific summand, too. Now, with this, we define our real numbers n.

Definition 9 (Real numbers n of a 3SUM list L)**.** *We define all numbers $n_i \in \mathbb{R}$ of a 3SUM list L by $n_i := q_{i,0} + \sum_{\forall i_{a,i,j}\mathbb{I}_a} i_{a,i,j}q_{i,j}$, with $q_{i,j} \in \mathbb{Q}$ and $|\mathbb{I}_a| < \infty$. If cases of kind $q_1i_{a,1} + q_2i_{a,2} = q_{a,3} \in \mathbb{Q}$ exist, we solve them each for one of its two given irrational numbers $i_{a,1}$ and substitute all appearances of $i_{a,1}$ in all numbers n_i by this. In this way we get everywhere left with $i_{a,2}$ and we add the rational part $q_{a,3}$ to q_0.*

Later in our algorithm, we will see, why this last part of our definition is necessary to get all possible solutions and to avoid troubles with the mentioned special cases.

With this insights, we will examine the zero sum binary addition for integers and for rational numbers, in the following.

3.2 Integer Binary Addition

We assume that given are three integers z_1, z_2 and z_3 and their belonging bit strings $B_{n_1}^{N-1}$, $B_{n_2}^{N-1}$ and $B_{n_3}^{N-1}$. We want to analyse the zero sum binary addition of this three numbers. Negative numbers are given in two's complement representation and hence the universe of values for each integer is given by $[-2^{N-1}, 2^{N-1} - 1]$. For this, we go through the binary addition of three integers (see also table 1) and determine several rules regarding possible addition partners for getting a zero sum.

Table 1: Binary addition of three integers with maximal carrier.

		1		1				
1	0	1	0	1	0	1		
···	···	1	1	1	1	1	1	
···	···	1	1	1	1	1	1	
···	···	0	0	0	0	1	0	
···	···	0	0	0	0	0	0	
···	···	5	4	3	2	1	0	q

Theorem 3.1 (Carriers $c_{[q]}$)**.** *The zero sum binary addition $z_1 + z_2 + z_3 = 0$ of three integers has the possible carriers $c_{[q]} \in \{0, 1, 10\}$.*

Proof. We look at bit position $q = 0$ which has by sure no carrier from a precursor bit addition. If we add the three bits $\left(b_{1,[0]}, b_{2,[0]}, b_{3,[0]}\right)$ we get for $b_{1,[0]} + b_{2,[0]} + b_{3,[0]} = 0$ the case $(0, 0, 0)$ with carrier $c_{[0]} = 0$ and the cases $(1, 1, 0)$, $(1, 0, 1)$ and $(0, 1, 1)$ with carrier $c_{[0]} = 1$ for a zero sum. Now, we go to bit position $q > 0$ which gets a carrier $c_{[q-1]}$ from its direct precursor bit addition. In the case of $c_{[q-1]} = 0$ we have the same situation like before. In the case of $c_{[q-1]} = 1$ we get for $\left(c_{[q-1]}, b_{1,[q]}, b_{2,[q]}, b_{3,[q]}\right)$ the valid case $(1,1,1,1)$ with carrier $c_{[q]} = 10$ and the cases $(1,1,0,0)$, $(1,0,1,0)$ and $(1,0,0,1)$ with carrier $c_{[q]} = 1$. □

Theorem 3.2 (Number of single bit addition bits)**.** *Each bit addition during a zero sum binary addition of three integers consists of either* 3 *or* 4 *or* 5 *bits in a single bit addition step* q.

Proof. We get a 3 bit bit addition in step q if $\left(c_{[q-1]} = 0 \wedge c_{[q-2]} = 0\right)$. This is clearly always satisfied for bit position $q = 0$. It's also always fulfilled if $\left(c_{[q-1]} = 10 \wedge c_{[q-2]} = 1\right)$ or $\left(c_{[q-1]} = 10 \wedge c_{[q-2]} = 1 \wedge c_{[q-3]} = 10\right)$. We get a 4 bit bit addition in step q if $c_{[q-1]} = 1$ which can be generated by $c_{[q-2]} = 0$ or by $c_{[q-2]} = 1$. Finally, we get a 5 bit bit addition in step q if $c_{[q-1]} = 1$ and $c_{[q-2]} = 10$. □

Theorem 3.3 (Number of 0s and 1s in integers for zero sum)**.** *Each bit addition during a zero sum binary addition of three integers gets zero sum if for a* 3 *bit addition the integers consist of three* 0*s or of one* 0 *and two* 1*s at position* q*, if for a* 4 *bit addition the integers consist of two* 0*s and one* 1 *or of three* 1*s at position* q *and if for a* 5 *bit addition the integers consist of three* 0*s or one* 0 *and two* 1*s at position* q.

Proof. To get a zero bit from a bit addition of 3 bits its clear that this is statisfied if we receive 0 with $c_{[q]} = 0$ which we get for the addition of three 0s, or 10 with $c_{[q]} = 1$ which we get for the addition of two 1s. We get a zero bit from a bit addition of 4 bits, one of them the carriere $c_{[q-1]} = 1$, if we receive 10 with $c_{[q]} = 1$ which we receive for two 0s and one 1, or 100 with $c_{[q]} = 10$ which we get for the additional addition of three 1s. Finally, we get a zero bit from a bit addition of 5 bits, two of them from the carriers $c_{[q-1]} = 1$ and $c_{[q-2]} = 10$, if we receive 10 with $c_{[q]} = 1$ which we get for the addition of three additional 0s, or 100 with $c_{[q]} = 10$ which we get for the addition of additional one 0 and two 1. □

Theorem 3.4 (Carriers for 3, 4 and 5 bit bit additions)**.** *The possible carriers of a* 3 *bit bit addition at position* q *are* $c_{[q]} = 0$ *or* $c_{[q]} = 1$*, for a* 4 *bit bit addition* $c_{[q]} = 1$ *or* $c_{[q]} = 10$*, and for a* 5 *bit bit addition* $c_{[q]} = 1$ *or* $c_{[q]} = 10$.

Proof. That proof directly follows from the proof above. □

With the results of theorem 3.1 to 3.4, we can put together a set of rules regarding the properties of integers for zero sum addition of three numbers. We will write short $\mathcal{C}^{c_{[q]}}_{[q],b} := \left(\mathcal{C}_{[q],0}, \mathcal{C}_{[q],1}, c_{[q]}, b\right)$ with $C_{[q],0}$ be the number of 0s, $C_{[q],1}$ be the number of 1s, $c_{[q]}$ be the generated carriere and b be the number of bits of the belonging case, for our possible bit cases.

Definition 10 (Zero sum bit addition properties)**.** *The zero sum bit addition of three numbers for position* q *is statisfied for*

$$\begin{array}{lll} \mathcal{C}^{0}_{[q],3} := (3,0,0,3) & \mathcal{C}^{10}_{[q],4} := (0,3,10,4) & \mathcal{C}^{10}_{[q],5} := (1,2,10,5) \\ \mathcal{C}^{1}_{[q],3} := (1,2,1,3) & \mathcal{C}^{1}_{[q],4} := (2,1,1,4) & \mathcal{C}^{1}_{[q],5} := (3,0,1,5) \end{array} \tag{1}$$

Definition 11 (Zero sum bit addition rules)**.** *The complete zero sum addition of three numbers, each of length* N*, is given by the rules*

$$\begin{array}{l} \mathcal{C}^{0}_{[q],3} \to \{\mathcal{C}^{0}_{[q+1],3}, \mathcal{C}^{1}_{[q+1],3}\} \\ \mathcal{C}^{1}_{[q],3} \to \{\mathcal{C}^{10}_{[q+1],4}, \mathcal{C}^{1}_{[q+1],4}\} \end{array} \tag{2}$$

$$\begin{array}{l} \mathcal{C}^{10}_{[q],4} \to \{\mathcal{C}^{0}_{[q+1],3} \to \{\mathcal{C}^{10}_{[q+2],4}, \mathcal{C}^{1}_{[q+2],4}\}, \mathcal{C}^{1}_{[q+1],3} \to \{\mathcal{C}^{10}_{[q+2],5}, \mathcal{C}^{1}_{[q+2],5}\}\} \\ \mathcal{C}^{1}_{[q],4} \to \{\mathcal{C}^{10}_{[q+1],4}, \mathcal{C}^{1}_{[q+1],4}\} \end{array} \tag{3}$$

$$\begin{array}{l} \mathcal{C}^{10}_{[q],5} \to \{\mathcal{C}^{0}_{[q+1],3} \to \{\mathcal{C}^{10}_{[q+2],4}, \mathcal{C}^{1}_{[q+2],4}\}, \mathcal{C}^{1}_{[q+1],3} \to \{\mathcal{C}^{10}_{[q+2],5}, \mathcal{C}^{1}_{[q+2],5}\}\} \\ \mathcal{C}^{1}_{[q],5} \to \{\mathcal{C}^{10}_{[q+1],4}, \mathcal{C}^{1}_{[q+1],4}\} \end{array} \tag{4}$$

Notations. We write $\mathcal{C}_{[q],b}^{c_{[q]}}(0)$ to get the value of $\mathcal{C}_{[q],0}$ from $\mathcal{C}_{[q],b}^{c_{[q]}}$, $\mathcal{C}_{[q],b}^{c_{[q]}}(1)$ to get the value of $\mathcal{C}_{[q],1}$ from $\mathcal{C}_{[q],b}^{c_{[q]}}$, $\mathcal{C}_{[q],b}^{c_{[q]}}(c)$ to get the value of $c_{[q]}$ from $\mathcal{C}_{[q],b}^{c_{[q]}}$ and $\mathcal{C}_{[q],b}^{c_{[q]}}(b)$ to get the value of b from $\mathcal{C}_{[q],b}^{c_{[q]}}$.

3.3 Rational Number Binary Addition

After the analysis of the integer zero sum case, we now want to move to rational numbers, in general. Since, we want to introduce a binary representation based algorithm, we are interested in approaches for zero sum addition representation which use as less bits as possible to represent as much as possible different numbers. We want to discuss different options for this, in the following.

3.3.1 Integer Reduction

The most obvious possibility to analyse the zero sum of three rational numbers $q_1 := \alpha_1/\beta_1$, $q_2 := \alpha_2/\beta_2$ and $q_3 := \alpha_3/\beta_3$, $\alpha_i \in \mathbb{Z}$ and $\beta_i \in \mathbb{N}_1$, is given by a reduction of rational numbers to integers. If we look at $0 \overset{!}{=} q_1+q_2+q_3 = \alpha_1/\beta_1+\alpha_2/\beta_2+\alpha_3/\beta_3 = (\alpha_1\beta_2\beta_3 + \alpha_2\beta_1\beta_3 + \alpha_3\beta_1\beta_2) \,/\, (\beta_1\beta_2\beta_3)$, we see that the zero sum question for rational numbers get reduced to the integer zero sum question $0 \overset{!}{=} \alpha_1\beta_2\beta_3 + \alpha_2\beta_1\beta_3 + \alpha_3\beta_1\beta_2$ which can be achived by multiplying all number by the product of denomiators of all q respectively their lowest common denominator. The necessary number of bits of each new summand gets b^3, if the number of bits of all α_i and β_i bit strings is b each. Additionally, in each summand, we get a mix of nominators and denominators of several numbers. Later, we will see that this makes things unnecessary complicated, so we look at other options.

3.3.2 Floating-Point Arithmetic Binary Addition

We assume that given are three numbers n_1, n_2 and n_3 in floating-point arithmetic following the *IEEE Standard for binary floating-point arithmetic* [15] [16] [17] which is used by the most computer systems, today. We want to analyse the zero sum binary addition of three numbers of this standard.

Definition 12 (Floating-point representation)**.** *The floating-point representation of a real number is defined by*

$$n := s \cdot m \cdot b^e \tag{5}$$

with the sign s, the mantissa m, the base b and an exponent e. The sign $s = (-1)^S$ is given by one bit S ($S = 0$ positive, $S = 1$ negative). The mantissa $1 \le m < 2$ consist of p mantissa bits of value M such that $m = 1.M = 1 + M/2^p$, because of normalization, so that the leading 1 doesn't have to be saved. Following the IEEE standard for floating-point arithmetic [17], b is always 2. Finally the exponent e is saved as not negative binary number E by adding a fixed bias value B with $E := e + B$. The bias value is determined by $2^{r-1} - 1$ for an exponent e represented by r bits.

The maximal exponent value $E = 11\ldots111_2 = 2^r - 1$ is reserved as special value for NaN and ∞, and the minimal exponent value $E = 00\ldots000_2 = 0$ is reserved for the floating-point numbers 0 and all denormalized values. NaN is used for undefined values, like $0/0$ or $\infty - \infty$. Denormalized, or also called subnormal numbers, are number in the range of the smallest absolute normalized floating-point number and 0. They are saved as fixed-point numbers and don't have the same precision like floating-point numbers.

The IEEE standard [17] differs between four possible datatypes: single (32 bits, $r = 8$ bits, $p = 23$ bits, $-126 \le e \le 127$, $B = 127$), single extended (≥ 43 bits, $r \ge 11$ bits, $p \ge 31$ bits, $e_{min} \le -1022$ and $e_{max} \ge 1023$, B is not specified), double (64 bits, $r = 11$ bits, $p = 52$ bits, $-1022 \le e \le 1023$, $B = 1023$) and double extended (≥ 79 bits, $r \ge 15$ bits, $p \ge 63$ bits, $e_{min} \le -16382$ and $e_{max} \ge 16383$, B is not specified) with precisions from 7 to 20 decimals. The specific representation of floating-point parameters isn't consistent from system to system. The most distributed representation consists of a big endian version with most left bit b_{N-1} being defined as the sign bit, then the reserved exponent bits and finally the reserved mantissa bits as most right bit block.

For analysing the zero sum addition of three given floating-point numbers in normalized representation, we have to determine the exponent e of each number and bringing all three numbers to the same exponent, so we can do an analog addition like for integers. If we want to examine this addition bitwise like for integers, we are forced to temporary store each value with its full representation after a possible shifting by an exponent adjustment, to be able to align the radix point separation of all three numbers. For a floating-point datatype of size b bits, mantissa p bits and r exponent bits, we get for $2^1 2^{r-1} 2^p = 2^b$ possible floating-point values with $b = r + p$ bits, a necessary mapped bit number b' of $1 + 2^{r-1} - 1 + p = 2^{r-1} + p$ bits for temporary storing. With this the maximal possible value which can be represented by the floating-point representation is given by $(2 - 2^{-p})\, 2^{2^{r-1}-1}$. An other option is still to save only the original floating-point representation and working with additional information giving the number of additional zeros because of exponential shift and their positions by pointers or bit index savings during further parts of the algorithm, to return the correct bit value at a specific position, even without saving the shifted value itself.

3.3.3 Fixed-Point Arithmetic Binary Addition

Although, today mostly floating-point numbers are used, because of their large range of values, in some cases also fixed-point numbers are in usage, because of their higher precision and faster calculation without the leak of necessary exponential shifts for arithmetic calculations like additions.

Definition 13 (Fixed-point representation). *A binary fixed point number is defined by its total number of bits b, an optional sign bit s and b^{pre} digits before the radix point and $b^{post} = b - 1 - b^{pre}$ decimal places. The radix point position is fixed. The b^{pre} digits are given as standard integer. The bits of the b^{post} decimal places are given by the fractional parts of the given value to b^{post}.*

For example 0.25_{10} is presented exactly by 0.01_2. Instead 0.7_{10} can only be approximated by $\approx 0.00001011_2$ in fixed-point arithmetic. Basic arithmetic operations like addition or subtraction can be done in the usual case like for integer numbers. Since, the radix point has a fixed position for all numbers, the necessary alignment can be trivially done. The total number of bits of a fixed-point number is $b = 1 + b^{pre} + b^{post}$ to present 2^b different numbers. The maximal possible value depends on the specific choice of b^{pre} and b^{post} and is given by $\left(2^{b^{pre}} - 1\right).\left(2^{b^{post}} - 1\right)$. Like also already for floating-point numbers, instead of saving the new shifted value, we can also save shifting informations, to save memory space.

3.3.4 Fixed-Point Integer Arithmetic Binary Addition

We want to introduce a so-called *fixed-point integer arithmetic* as variation of the standard fixed-point arithmetic.

Definition 14 (Fixed-point integer arithmetic). *A number n in* ***fixed-point integer arithmetic*** *is given by a bit string $B^{b-1} := (i_1, i_2)$, consisting of the two integers i_1 with $|i_1| := 1 + b^{pre}$ and i_2 with $|i_2| := 1 + b^{post}$ with the same sign $s_1 = s_2$ and total length $b = 1 + b^{pre} + 1 + b^{post}$.*

It is defined by its total number of bits b, optional sign bits $s_1 = s_2$ and b^{pre} digits before the radix point and $b^{post} = b - 1 - b^{pre} - 1$ decimal places, too. The radix point position is always fixed. Also like the fixed point arithmetic, the b^{pre} digits are given as standard integer. In constrast to the fixed-point arithmetic the b^{post} decimal places are also given by an integer, interpreting all decimal places as one integer, like if we would simple ignore the radix point and all digits before, and not by a fractional part. For example 2.25_{10} is presented by $10_2.11001_2$. In contrast to the fixed-point arithmetic, a correct representation of all decimal places within the precision of the number of bits b^{post} is possible, but instead usual standard arithmetic operations like addition or substraction can no longer be done like before.

Example 1. *Consider the addition of the numbers $0.90_{10} = 0_2.1011010_2$ and $0.20_{10} = 0_2.0010100_2$ from which we subtract $0.10_{10} = 0_2.0001010_2$ by $-0.10_{10} = 0_2.1110110_2$ in fixed-point integer arithmetic (we left out the leading sign bits for the moment). We will look at the pre and the post radix point zero sum addition separately. For the post radix point addition we get* $110 = 90_{10} + 20_{10} = 1011010_2 + 0010100_2 = 1101110_2$. *It* *is obviously that substracting* -0.10_{10} *will not give us* $0_{10} = 0000000_2$*, instead we will get* $100_{10} = 1100100_2$*. This means, if we are interested for the zero sum case, we have to look for* 1100100_2 *and moving a leading* 1 *carrier bit to our pre radix point addition, now.*

With the help of this small example, we can steady properties of zero sum fixed-point integer arithemtic addition of three numbers.

Theorem 3.5 (Post radix point zero sum)**.** *The fixed-point integer addition of three numbers* $n_1 := \alpha_1.\beta_1$, $n_2 := \alpha_2.\beta_2$ *and* $n_3 := \alpha_3.\beta_3$ *gets a post radix point zero sum part if* $\beta_1 + \beta_2 + \beta_3 \in \{-10_{10}^{b^{post}}, 0_{10}^{b^{post}}, 10_{10}^{b^{post}}\}$ *for* $b^{post} \geq 1$.

Proof. For a sum of three numbers in fixed-point integer arithmetic, the maximum range of value for each positiv digit in base 10 is $[0, 9]$. Hence, the maximum positive value of the sum of three number lies within the range $[0, 3 \cdot 10_{10}^{b^{post}}[$. For the case of zero sum, we know that we can either have three numbers each with 0 as post radix point value, whose post radix point sum clearly gets 0. Or we have two positive values plus one negative value, respectively one positive value and two negative values. The sum of two values of the same sign always lies in the range $[0, |2 \cdot 10^{b^{post}}|[$. Hence, because we are looking for the zero sum case by the addition of the third value, we always get a zero sum case for a post radix point value of 0 or $|10_{10}^{b^{post}}|$. □

Theorem 3.6 (Post radix point addition carriers)**.** *The possible carriers in base* 10 *for a post radix point zero sum addition of three numbers are* $\{-1, 0, 1\}$.

Proof. This follows trivially from theorem 3.5. □

With this new properties of zero sum addition, we see that we have a similiar case to the integer addition one. We also has the zero sum which leads to 0s for each bit, but here we have the two additional cases of $-10_{10}^{b^{post}}$ and $10_{10}^{b^{post}}$, now. Hence, we have to think about this two additional situations in which we have to get 1 at particular bit positions, now. We can use our results from integer addition helpfully.

Theorem 3.7 (Carriers $c_{[q]}$)**.** *The zero sum post radix point binary addition* $n_1 + n_2 + n_3 = 0$ *of three numbers in fixed-point integer representation has the possible carriers* $c_{[q]} \in \{0, 1, 10\}$.

Proof. If we look at bit position $q = 0$, we have the carrier $c_{[q]} = 0$ for in the case of getting bit 0 analog to the integer case. For getting a 1 bit we have $(0, 0, 1), (0, 1, 0)$ and $(1, 0, 0)$. We get the carrier $c_{[q]} = 1$, if we want to get a 0 bit, we have again the analog case like for integers and we get a 1 bit for $(1, 1, 1)$. If we go to position $q = 1$, the 0 bit case is again like for integers and if we want to get a 1 bit with having a carrier $c_{[q-1]} = 1$, we need $(1, 0, 0, 0)$ with carrier $c_{[q]} = 0$ or $(1, 0, 1, 1), (1, 1, 0, 1)$ or $(1, 1, 1, 0)$ with carrier $c_{[q]} = 1$. Finally, at position $q = 2$, we can get again the analog case like for integers for 0 bits, and if we want to get a 1 bit with having a carrier $c_{[q-1]} = 1$ and $c_{[q-2]} = 10$, we need $(1, 1, 1, 0, 0), (1, 1, 0, 1, 0)$ or $(1, 1, 0, 0, 1)$ with carrier $c_{[q]} = 1$ or $(1, 1, 1, 1, 1)$ with carrier $c_{[q]} = 10$. □

Theorem 3.8 (Number of single bit addition bits)**.** *Each bit addition during a zero sum post radix point binary addition of three numbers consists of either* 3 *or* 4 *or* 5 *bits in a single bit addition step* q.

Proof. Because of the same carrier situation like for the integer case, we have the analog proof for our statement like for theorem 3.2. □

Theorem 3.9 (Carriers for 3, 4 and 5 bit bit additions)**.** *The possible carriers of a* 3 *bit bit addition at position* q *are* $c_{[q]} = 0$ *or* $c_{[q]} = 1$*, for a* 4 *bit bit addition* $c_{[q]} = 1$ *or* $c_{[q]} = 10$*, and for a* 5 *bit bit addition* $c_{[q]} = 1$ *or* $c_{[q]} = 10$.

Proof. That proof directly follows from the proofs above. □

Form this, we can follow the properties for getting a 1 bit during post radix point zero sum bit addition of three numbers.

Definition 15 (Post radix point zero sum bit addition properties for 1 bits)**.** *The post radix point zero sum bit addition of three numbers for position q for getting a 1 bit is statisfied for*

$$\begin{array}{lll} \mathcal{C}^{0}_{[q],3} := (2,1,0,3) & \mathcal{C}^{1}_{[q],4} := (1,2,1,4) & \mathcal{C}^{1}_{[q],5} := (2,1,1,5) \\ \mathcal{C}^{1}_{[q],3} := (0,3,1,3) & \mathcal{C}^{0}_{[q],4} := (3,0,0,4) & \mathcal{C}^{10}_{[q],5} := (0,3,10,5) \end{array} \tag{6}$$

The total number of bits of a fixed-point integer number is $b = 1 + b^{pre} + 1 + b^{post}$ to present 2^b different numbers. The maximal possible value depends on the specific choice of b^{pre} and b^{post} and is given by $\left(2^{b^{pre}} - 1\right) . \left(2^{b^{post}} - 1\right)$.

4 The General Algorithm

We present the final algorithm. At first, we will give a general overview about the algorithm idea. Then we will describe necessary preprocessing data structuring and our `3SUM` frame algorithm itself.

4.1 The Idea

For the determination of valid solutions we split our algorithm into several parts. At first we look at the set of the given numbers n_i. We use a list entries binary tree T to sort them according their bit value 0 or 1 at bit position q. Next, we generate a second binary tree T^{string} which is build by the set of rules $\mathcal{C}$ of definitions 10 and 15 and their successor rules we determined in the last section. Finally, we go through the tree T^{string} from its root to the final maximal depth $b+1$. In each depth `bit`, we take the given rules $\mathcal{C}$s and the solutions from the direct previous depth `bit - 1` and check if the nodes of tree T which are necessary to statisfy this specific rule with previous solution combination, have at least as much elements as necessary. If they all have enough element, then add the new solution to the set of solutions of depth `bit` to the tree T^{string}. In the next sections, we will explain this more detailed.

4.2 Data Structuring

The determination of `3SUM` solutions starts with some preparation.

Step 0 (Assumptions)**.** *Given a list L of n real numbers $n_i \in \mathbb{R}$. Each number n_i can again consists of several real numbers $n_{i,j}$ (we will call them* ***sub numbers****). Without further checking, we assume that all n_i and their sub numbers $n_{i,j}$ are valid values, means no one of them is NaN or $\pm\infty$.*

Furthermore, we assume that the number of bits b^{pre} for the representation of pre radix point parts of rational numbers $q \in \mathbb{Q}$ is choosen in such a way, that its precision is sufficient enough to present all pre radix point parts of rational numbers of our list L exactly. Analog, we assume that the number of bits b^{post} for the representation of post radix point parts of rational numbers is choosen in such a way, that its precision is sufficient enough to present all post radix point parts of rational numbers of our list exactly, too.

The first step at all, consists of generating and saving the list L of n real numbers. We introduce our new data type `fixedint` (see listing 1) for fixed-point integer arithmetic.

Listing 1: Data type `fixedint`

```

datatype fixedint x.y {
    (int) x;
    (int) y;
};
```

Next, we define a new data structure `listnumber` (see listing 2) for one list element n_i. We assume that a number n can consists of one rational r and several irrational i parts. Since, we know from our consideration of section *Real numbers* 3.1 that it is not necessary to save the irrational numbers itself, we only safe a symbolic string for each irrational number i (see `datatype irrational`) and finally all the necessary irrational symbolic strings for our n in one list (see `irrational list`). Next, in `listnumber`, we save a possible rational sub number and the rational factors of possible irrational sub numbers with `fixedint` data type. Finally, we save a mapping `map`, to be still able to identify the saved rational factors of irrational sub numbers with their belonging irrational number string, later.

Listing 2: Data type `irrational`, data structure `listnumber` and mapping `map` from a list of irrational numbers to n

```

datatype irrational i {
    (string) i;
};

irrational list n_i = ( irrational i1,
                        irrational i2,
                        irrational i3,
                      ...);

struct listnumber n (    fixedint qx.qy,
                         fixedint i1_qx.i1_qy,
                         fixedint i2_qx.i2.qy,
                         fixedint i3_qx.i3_qy,
                      ...);

map n_types: n_i -> n[1:];
```

We save all numbers n_i in a list L, each as structure `listnumber`. To keep things better to manage, we define `irrational list` as complete list for all possible irrational sub numbers i overall n_i. Means, if for example the number n_1 consists of the irrational sub number i_1 and number n_2 consists of the irrational sub number $i_2 \neq i_1$, we save i_1 and i_2 in common in one `irrational list`, and define $i2\_qx.i2\_qy := 0.0$ for element n_1 and $i1\_qx.i1\_qy := 0.0$ for element n_2. In this way, we get an uniform list of the rational parts of irrational sub numbers overall numbers n_i with one fixed `map`, which is much easier to manage.

Step 1 (Saving list elements)**.** *We build a `listnumber list` L of real numbers $n_i \in \mathbb{R}$ by saving each element n_i as `listnumber` in an array at index i. The belonging irrational sub numbers are saved in one common `irrational list n_i` overall all elements n_i and a belonging mapping `map n_types`.*

4.3 The 3SUM Frame Algorithm

We start with the `3SUM` frame algorithm. At first, we want to consider the case of having only one rational sub number part for each n_i which means $n_i := q_i$ and no further sub numbers, at all. We use b^{pre} pre radix point bits and b^{post} post radix point bits. The frame algorithm consists of two binary trees which interacts with each other. We start with a new data structure `node_T` (see listing 3) for the binary tree T which administrates the list entries n_i of the given list L.

Listing 3: Data structure `node_T` for saving the information of one node of our binary tree T

```

struct node_T (_n = empty, _count = 0){
    struct listnumber list n = _n;
```

```
        int count = _count;
        struct node_T *zero_left = nil;
        struct node_T *one_right = nil;
    };
```

It consists of a list of numbers n_i by `struct listnumber list n` and a counting value `int count` for telling us the length of our list which is saved in this node. The default case is an empty list of length 0. Additionally, we initialize two null pointers `struct node_T *zero_left` and `struct node_T *one_right` which will be used for pointing to the child nodes, later. Optional we could augment our `listnumber list` data structure by also saving the list index of each n_i in the original list L. This information isn't necessary for our algorithm but can be interesting if we still want to be able to identify our solutions with the original list entries.

Our second binary tree T^{string} will be used to administrate the zero sum addition rules C as well their valid solutions in depth d of the binary tree. For this we will use for each node a new data structure `node_Tstring` (see listing 4).

Listing 4: Data structure `node_Tstring` for saving the information of one node of our binary tree T^{string}

```

    struct rule C (C0, C1, c, b);
    struct pt[3] vn (None, None, None);

    struct node_Tstring (_C = empty, _vn = empty) {
        struct rule C = _C;
        int list vn = _vn;
        struct node_Tstring *C_left = nil;
        struct node_Tstring *C_right = nil;
    };
```

In the sections *Integer Binary Addition* 3.2 and *Fixed-Point Integer Arithmetic Binary Addition* 3.3.4, we defined a set of rules regarding zero sum binary addition. For us, this means the addition of three numbers to get a zero sum is a deterministic question following a set of fixed rules $\mathcal{C}$. We also saw, that for real numbers, we have to differ between three possible cases for zero sum of a post radix point part of a rational number: $\mathcal{C}_{string}^{b^{post}} \in \{-10_{10}^{b^{post}}, 0_{10}^{b^{post}}, 10_{10}^{b^{post}}\}$.

Assume we want to determine the zero sum of three numbers with b^{post} post radix point bits. For each possible value of $C_{string}^{b^{post}}$ for this one fixed number b^{post} plus the additional b^{pre} bits which always have to sum to a 0 bit string, we can determine a binary tree T^{string}, which gives us all possible rule sequences for getting a zero sum of three numbers.

In `struct rule C` we save this rule for each node and in `int list vn` its belonging solutions by pointers to the nodes of T from which we have to take the solution entries which we will determine in the following. Like for tree `node_T`, we initialize two null pointers `struct node_Tstring *C_left` and `struct node_Tstring *C_right` which will be used for pointing to the child nodes, later.

Now, with this we can give our `3SUM` frame algorithm (see listing 5).

Listing 5: The `3SUM` frame algorithm

```

    new node_T T(L,n);
    new node_Tstring Tstring(empty, empty);

    determineTree_T(T, bit) {

        if bit <= (b-1):

            // Determine T for depth bit
            T.zero_left = new node_T (empty,0);
            T.one_right = new node_T (empty,0);

            if T.count != 0:
                for i=0...(T.count - 1) do
                    if T.n[i][bit] == 0:
                        T.zero_left.n.add(T.n[i]);
                        T.zero_left.count += 1;
                    elseif T.n[i][bit] == 1:
                        T.one_right.n.add(T.n[i]);
                        T.one_right.count += 1;

            determineTree_T(T.zero_left, bit+1);
            determineTree_T(T.one_right, bit+1);

    };


    determineTree_Tstring(pattern, Tstring, bit) {

        if bit < (b-1):

            // Determine Tstring for depth bit
            Tstring.C_left = new node_Tstring(empty,empty);
            Tstring.C_right = new node_Tstring(empty,empty);

            Tstring.C_left.C = successor1(pattern, Tstring.C);
            Tstring.C_right.C = successor2(pattern, Tstring.C);

            // Check for solutions for depth bit
            if bit == 0:
                solutions_left = checkForValidSol(Tstring.C_left.C);
                if solutions_left != empty:
                    Tstring.C_left.vn.add(sol_tuple(solutions_left));

                solutions_right = checkForValidSol(Tstring.C_right.C);
                if solutions_right != empty:
                    Tstring.C_left.vn.add(sol_tuple(solutions_right));
            elseif bit > 0 and Tstring.vn != empty:
                for ns in Tstring.vn do
                    solutions_left = checkForValidSol(ns, Tstring.C_left.C);
                    if solutions_left != empty:
                        Tstring.C_left.vn.add(sol_tuple(solutions_left));

                    solutions_right = checkForValidSol(ns, Tstring.C_right.C);
                    if solutions_right != empty:
                        Tstring.C_left.vn.add(sol_tuple(solutions_right));

            determineTree_Tstring(Tstring.C_left, b+1);
            determineTree_Tstring(Tstring,C_right, b+1);
```

```
60          else:
61              return Tstring.vn != empty;   // Final solutions
62
63      };
64
65
66      solve3SUM(T, Tstring) {
67          determineTree_T(T,0);
68          determineTree_Tstring(pattern1, Tstring,0);
69          determineTree_Tstring(pattern2, Tstring,0);
70          determineTree_Tstring(pattern3, Tstring,0);
71      };
```

The first main part of our frame algorithm is `determineTree_T(T,bit)` (see line 5).

Step 2 (Creation of list entries tree T)**.** *We create a binary tree T of all list entries $n_i = q_i = q_i x.q_i y$ of L. We do this by splitting a given* `listnumber list n` *in depth* `bit` *of the tree, regarding the bit value $n_{i,[bit]}$ of each list entry n_i. If $n_{i,[bit]} = 0$ then add it to child node* `T.zero_left.n` *and increment* `T.zero_left.count` *by one, else add it to child node* `T.one_right.n` *and increment* `T.one_right.count` *by one.*

We finish with a tree T for $b = b^{pre} + b^{post}$ bits with a total depth $b+1$ and a width of 2^b in depth b (starting indexing the depth with zero). In each depth we have n list elements overall.

Additionally, to the list entries tree T, we also manage a second tree, the zero sum rules tree T^{string} by `determineTree_Tstring` (Tstring, bit) (see line 28|).

Step 3 (Creation of zero sum rules tree T^{string})**.** *We create a binary tree T^{string} of all zero sum rules $\mathcal{C}_{[q]}$ for one fixed $\mathcal{C}^{b^{post}}_{string}$ and the $\mathcal{C}^{b^{pre}}$ zero sum rule string of zeros, in common $\mathcal{C}^{b}_{string} := \mathcal{C}^{b^{pre}} \circ \mathcal{C}^{b^{post}}_{string}$, regarding the bit* `bit`*. For this we take for every* `Tstring.C` *its two successor rules according definitions 10, 11 and 15 by* `pattern` $\in \{-10^{b^{post}}_{10}, 0^{b^{post}}_{10}, 10^{b^{post}}_{10}\}$ *and add them to the two child nodes of* `Tstring` *by* `Tstring.C_left.C = successor1(pattern, Tstring.C)` *and* `Tstring.C_right.C = successor2(pattern, Tstring.C)` *(see lines 33 to 37). We define the two successors of the empty root node rule by the two 3 bit rules $\mathcal{C}^0_{[0],3}$ and $\mathcal{C}^1_{[0],3}$.*

We finish with a tree T^{string} for $b = b^{pre} + b^{post}$ bits with a total depth $b+1$ and a width of 2^b in depth b (also starting indexing the depth with zero). In each depth b we have 2^b zero sum rules.

Like you maybe already noticed, we left stuff out in our step explanation of the raw frame algorithm (see starting at line 39). We come to this, the solution checks, now.

Step 4 (Solution checks)**.** *For the identification of valid solutions in depth* `bit`*, we have to check for each valid solution of depth* `bit - 1`*, given by* `Tstring.vn` *(see line 49), if the new rule* `Tstring.C_left.C` *respectively* `Tstring.C_right.C` *can be statisfied taking* `ns` *into account, at all. This check is done in* `checkForValidSol(ns, Tstring.C_left.C)` *respectively* `checkForValidSol(ns, Tstring.C_right.C)` *(see line 50 and 54). If the rule* `Tstring.C_left.C` *respectively* `Tstring.C_right.C` *in consideration of* `ns` *can be statisfied, it is added to* `Tstring.C_left.vn` *respectively* `Tstring.C_left.vn` *by adding a tuple of three pointers* `sol_tuple` *to the nodes of T from which the three list elements which statisfy the rule of step* `bit` *have to be taken (see line 52 and 56). If no solution* `ns` *from depth* `bit - 1` *exists, given by* `T.string.vn == empty`*, there exists no possibility anymore for this rule chain to solve the problem, hence this particular branch has to die. For the two rules of the successors of the empty root node, we have the exception that no* `ns` *has to be considered, means is this case we have* `Tstring.vn = empty`*.*

Remark 4.1. We want to emphasize that the pointers of `sol_tuple` point to the nodes from which we have to take the solution n_i's and NOT to the n_i's itself!

Finally, we want to give the conditions for `checkForValidSol(ns := Tstring.vn[i], Tstring.C_{*}.C)`.

Definition 16 (checkForValidSol(Tstring.vn[i], Tstring.C_{*}.C))**.** *Given a solution tuple* ***sol_tuple*** *by* ***Tstring.vn[i]*** *of three pointers* ***(*pt1, *pt2, *pt3)*** *which points to the nodes of T of depth* ***bit - 1*** *from which we have to take the three numbers n_1, n_2 and n_3. A rule* ***Tstring.C_*.C*** *of depth* ***bit*** *leads to a valid solution of* ***Tstring.C_*.vn*** *of depth* ***bit*** *if all of the following conditions are statisfied.*

1. *Condition: The sum of numbers of list entries which are taken for the solution from two child nodes of a common direct parent node, has to be the same number like the number of list entries which where taken for the solution in the step before from their common direct parent node.*

$$\begin{aligned} & n_1 \in (\texttt{(pt1 -> T).zero\_left.n} \oplus \texttt{(pt1 -> T).one\_right.n}) \\ \wedge \quad & n_2 \in (\texttt{(pt2 -> T).zero\_left.n} \oplus \texttt{(pt2 -> T).one\_right.n}) \\ \wedge \quad & n_2 \in (\texttt{(pt3 -> T).zero\_left.n} \oplus \texttt{(pt3 -> T).one\_right.n}) \end{aligned} \tag{7}$$

2. *Condition: The sum of the number of 0s which are taken in the current step has to be equal to the value of zeros given by rule $\mathcal{C}$. Also the sum of the number of 1s which are taken in the current step has to be equal to the value of ones given by rule $\mathcal{C}$.*

$$\begin{aligned} & |\{\forall n_i \in (n_1, n_2, n_3)\, | n_i \in \texttt{(pt\{*\} -> T).zero\_left.n}\}| = \texttt{Tstring.C\_*.C[0]} \\ \wedge \quad & |\{\forall n_i \in (n_1, n_2, n_3)\, | n_i \in \texttt{(pt\{*\} -> T).one\_right.n}\}| = \texttt{Tstring.C\_*.C[1]} \end{aligned} \tag{8}$$

5 Algorithm Complexities

In the last section, we gave our 3SUM algorithm in a general way in which we still left out the open question how exactly we determine valid solutions in checkForValidSol(Tstring.vn[i], Tstring.C_{*}.C) and the belonging complexity for doing this. We will give answers to this, now.

5.1 Basic tree complexities

We assume a serial implementation of the four binary trees T, $T^{string}_{pattern1}$, $T^{string}_{pattern2}$ and $T^{string}_{pattern3}$.

Theorem 5.1 (Complexities of T)**.** *The time complexity to generate a list entries binary tree T of total depth $b+1$ is $\mathcal{O}(nb)$. Its space complexity is $\mathcal{O}_{Space}(nb)$.*

Proof. In the worst case, our given list L consists of n entries, all identically, with each bit either 0 or with each bit 1. In each tree depth we get one list with all elements n, and $2^{bit} - 1$ empty lists. Hence, in each depth we have a time complexity of n. In total this gives us $n(b+1)$ and hence $\mathcal{O}(nb)$. Since we already mentioned, in each depth, the sum of all single list lengths overall always equals n, we get a space complexity also by $n(b+1)$ and hence $\mathcal{O}_{Space}(nb)$. □

Next, we look at the complexities of a zero sum rule tree $T^{string}_{pattern}$ for pattern. We see that the complexity of our algorithm part depends on checkForValidSol(Tstring.vn[i], Tstring.C_{*}.C) for all ns and Tstring.C_{*}.C) in a particular depth bit. We want to examine the specific properties of this algorithm part.

Regarding our rules given by definitions 10, 11 and 15, we know that we can only have some specific combinations for taking our three numbers from zero and from one bit representing nodes in depth bit. Since, we just have the possible rules $\mathcal{C}(0,3)$, $\mathcal{C}(3,0)$, $\mathcal{C}(1,2)$ and $\mathcal{C}(2,1)$ we get the following possible ns with rule $\mathcal{C}$ combinations for which we have to check if ...

1. In step `bit - 1` we take all three numbers from one *node*.
   For step `bit` we have:
   (a) Two child nodes $child_1 := node.zero$ and $child_2 := node.one$.
      i. Case $\mathcal{C}(0,3)$:
         A. $child_2.count \geq 3$
      ii. Case $\mathcal{C}(3,0)$:
         A. $child_1.count \geq 3$
      iii. Case $\mathcal{C}(1,2)$:
         A. $child_1.count \geq 1 \wedge child_2.count \geq 2$
      iv. Case $\mathcal{C}(2,1)$:
         A. $child_1.count \geq 2 \wedge child_2.count \geq 1$
2. In step `bit - 1` we take one number from $node_1$ and two numbers from $node_2$.
   For step `bit` we have:
   (a) Four child nodes $child_1 := node_1.zero$, $child_2 := node_1.one$, $child_3 := node_2.zero$ and $child_4 := node_2.one$.
      i. Case $\mathcal{C}(0,3)$:
         A. $child_2.count \geq 1 \wedge child_4.count \geq 2$
      ii. Case $\mathcal{C}(3,0)$:
         A. $child_1.count \geq 1 \wedge child_3.count \geq 2$
      iii. Case $\mathcal{C}(1,2)$:
         A. $child_1.count \geq 1 \wedge child_4.count \geq 2$
         B. $child_2.count \geq 1 \wedge child_3.count \geq 1 \wedge child_4.count \geq 1$
      iv. Case $\mathcal{C}(2,1)$:
         A. $child_1 \geq 1 \wedge child_3.count \geq 1 \wedge child_4.count \geq 1$
         B. $child_2 \geq 1 \wedge child_3.count \geq 2$
3. In step `bit - 1` we take one number from $node_1$, one number from $node_2$ and one number from $node_3$.
   For step `bit` we have:
   (a) Six child nodes $child_1 := node_1.zero$, $child_2 := node_1.one$, $child_3 := node_2.zero$, $child_4 := node_2.one$, $child_5 := node_3.zero$ and $child_6 := node_3.one$.
      i. Case $\mathcal{C}(0,3)$:
         A. $child_2.count \geq 1 \wedge child_4.count \geq 1 \wedge child_6.count \geq 1$
      ii. For $\mathcal{C}(3,0)$:
         A. $child_1.count \geq 1 \wedge child_3.count \geq 1 \wedge child_5.count \geq 1$
      iii. Case $\mathcal{C}(1,2)$:
         A. $child_1.count \geq 1 \wedge child_4.count \geq 1 \wedge child_6.count \geq 1$
         B. $child_2.count \geq 1 \wedge child_3.count \geq 1 \wedge child_6.count \geq 1$
         C. $child_2.count \geq 1 \wedge child_4.count \geq 1 \wedge child_5.count \geq 1$
      iv. Case $\mathcal{C}(2,1)$:
         A. $child_1.count \geq 1 \wedge child_3.count \geq 1 \wedge child_6.count \geq 1$
         B. $child_1.count \geq 1 \wedge child_4.count \geq 1 \wedge child_5.count \geq 1$
         C. $child_2.count \geq 1 \wedge child_3.count \geq 1 \wedge child_5.count \geq 1$

We see that it is not necessary to check for specific n_i values. Instead, we only have to check if the particular lists have at least as much numbers as necessary to statisfy the given `ns` with rule $\mathcal{C}$ combination. This property makes our algorithm independent from the total number of n's of L and it only become hooked on the number of distinguishable elements of L which is given by 2^b.

So, we can split `checkForValidSol(Tstring.vn[i], Tstring.C_{*}.C)` intro three parts: The first one, in which we have to determine the valid combinations of nodes for which we have to check their *count* values for a given `ns` with $\mathcal{C}$ case. The second, in which we have to determine for all nodes if their *count* values are at least 1, 2 and 3. And finally the third part, in which we have to bring the first two parts together to determine which combinations of nodes can be statisfied, at all.

We want to start with the determination of the maximal number of `ns` tuples in a particular depth `bit` of $T^{string}_{pattern}$. Since, we part in each depth step in T our list of n numbers regarding their bit values, we can associate the `ns` solutions in depth `bit`, with the solutions of the `3SUM` problem statements with numbers of `bit` bits. Hence, we are interested in the maximal number of solutions for the `3SUM` problem of a list of $n := 2^{bit}$ pairwise distinguishable numbers represented by `bit` bits each in their binary representation.

Lemma 5.1 (Maximal number of solutions of pairwise distinguishable numbers)**.** *The maximal number of valid solution triples for the zero sum problem of n pairwise distinguishable numbers is $\frac{1}{4}n^2 - \frac{1}{2}n$.*

Proof. Regarding the zero sum question of three numbers, we know that the sum of three time zero, as well as the sum of two positive numbers and one negative number, and the sum of two negative numbers and one positive number can statisfy the question. We assume given is a sequence of n pairwise distinguishable numbers. Be number $n_1 > 0$, $n_2 := k$ and $n_3 = -(n_1 + k)$. Additionally, be $\min(|n_1|, |n_2|, |n_3|) = |n_1|$ and $n_1 + k \leq n$. Then, we have $n - 2n_1$ choices for k and the maximal number of valid solution triples is given by $\sum_{n_1=1}^{n/2}(n - 2n_1) = \frac{1}{2}n - \frac{1}{4}n^2 - \frac{1}{2}n = \frac{1}{4}n^2 - \frac{1}{2}n$. □

Next, we have to think about the maximal number of possible combinations to check which we have to do for one `ns`.

Lemma 5.2 (Maximal number of solution node combinations for one `ns`)**.** *The maximal number of solution node combinations for one `ns` in a particular depth `bit` regarding given rules takes four.*

Proof. From the results of the consideration of possible `ns` with $\mathcal{C}$ situations as well as from lemma 5.1 by $\approx \left(2^{bit+1}\right)^2 = 2^2 2^{2\,bit} = 2^2\left(2^{bit}\right)^2$, we can derive, that the maximal number of node combinations which we have to check for one `ns` turns out to be four. From symmetry aspects of $\mathcal{C}$ rules, we know that for each `ns` we always have two possible rules and that one of this rules is of the form in which all three numbers are either 0 or all three are 1, together with the second rule in which one number is either 0 or 1 and we take the two numbers from the other set. This leads to a maximal number of possible node combinations of four. □

Lemma 5.3 (Maximal number of `ns` per node)**.** *The maximal number of `ns` for each node in depth `bit` equals $2^{1.58 \cdot bit}$.*

Proof. From the former considerations we know that we have until three possible combinations for each `ns` with one particular $\mathcal{C}$. Hence, it follows for depth `bit` a maximal number of `ns`'s for each node by $3^{bit-1} = 2^{\log_2(3)(bit-1)} \approx 2^{1.58(bit-1)} \approx 2^{1.58 \cdot bit}$. □

Since, we now know how much `ns` tuple we have, we can determine the complexity to determine the combinations of nodes for which we have to check their *count* values in a particular depth `bit`.

Lemma 5.4 (Complexity of valid combinations determination)**.** *The time complexity to determine all possible valid combinations of nodes for depth `bit` takes $\mathcal{O}\left(2^{1.58 \cdot bit}\right)$ and its space complexity takes $\mathcal{O}_{Space}\left(2^{1.58 \cdot bit}\right)$.*

Proof. We know for depth `bit` that we have 2^{bit} nodes. Each of this nodes can have maximal $2^{1.58 \cdot bit}$ tuples `ns`, with maximal three new valid combinations in depth `bit + 1`. Since, in a binary tree we consider all nodes in parallel, we get $3 \cdot 2^{1.58 \cdot bit}$ valid combinations and a time complexity of $\mathcal{O}\left(3 \cdot 2^{1.58 \cdot bit}\right) = \mathcal{O}\left(2^{1.58 \cdot bit}\right)$. For each valid combination we have to save three node count value pointers. Hence, we get a space complexity by $\mathcal{O}_{Space}\left(3 \cdot 3 \cdot 2^{1.58 \cdot bit}\right) = \mathcal{O}_{Space}\left(2^{1.58 \cdot bit}\right)$. □

Now, we look at our second part of `checkForValidSol(Tstring.vn[i], Tstring.C_{*}.C)`. Considering a tree $T^{string}_{pattern}$. We are interested in the time and space complexity of doing all necessary checks of `T.count` values which we need finally for the determinded valid combination checks.

Lemma 5.5 (Checks of all *T.count* values complexities for a fixed depth `bit`)**.** *Given a binary tree T at depth* `bit`*. The time complexity for doing all necessary checks of T.count values for determining valid* `ns` *with* `Tstring.C` *combinations for a fixed depth takes $\mathcal{O}(1)$ time and $\mathcal{O}_{Space}\left(2^{bit}\right)$ space complexity.*

Proof. If we have a binary tree T, we know that the number of nodes in depth `bit` is given by 2^{bit}. Because of the definitions 10 and 15, we know that we have to check nodes for $T.count \geq 1$, $T.count \geq 2$ and $T.count \geq 3$. In the worst case, we have to check each node for each of this three values. This leads to $3 \cdot 2^{bit}$ necessary checks in depth `bit`. Since all checks happen independently from each other in a binary tree, we can do it with $\mathcal{O}(3) = \mathcal{O}(1)$ time and $\mathcal{O}_{Space}\left(3 \cdot 2^{bit}\right) = \mathcal{O}_{Space}\left(2^{bit}\right)$ space complexity. □

We save the results of our *T.count* checks for example in a simple 3×2^{bit} matrix and can now use it to check our determined valid combinations.

Lemma 5.6 (Checks of valid combinations complexity)**.** *The time complexity to check all valid combinations in a particular depth* `bit` *takes $\mathcal{O}\left(2^{1.58 \cdot bit}\right)$.*

Proof. We know for depth `bit`, that we have maximal $3 \cdot 2^{1.58 \cdot bit}$ possible valid combinations to check for at each node. Since, we have to check for each node three count values, we get a time complexity by $\mathcal{O}\left(3 \cdot 3 \cdot 2^{1.58 \cdot bit}\right) = \mathcal{O}\left(2^{1.58 \cdot bit}\right)$. □

With this we have the complexities of one $T^{string}_{pattern}$.

Theorem 5.2 (Complexities of $T^{string}_{pattern}$)**.** *The time complexity of a binary tree $T^{string}_{pattern}$ to determine all valid solutions of total depth $b+1$ for one fixed given* `pattern` *is $\mathcal{O}\left(2^{1.58 \cdot b}\right)$. Its space complexity is given by $\mathcal{O}_{Space}\left(2^{1.58 \cdot b}\right)$.*

Proof. Regarding the time complexity, we know that we take $2^{1.58 \cdot bit}$ for each depth `bit`. Hence, we need a total time of $\approx \sum_{bit=1}^{b} 2^{1.58 \cdot bit} \approx 2^{1.58 \cdot b}$ and with our former considerations we get for space usage $\approx \sum_{bit=1}^{b}\left(2^{bit} + 2^{1.58 \cdot bit}\right) \approx 2^{b} + 2^{1.58 \cdot b} \approx 2^{1.58 \cdot b}$. □

Finally, we can give the total complexities for our `3SUM` algorithm for a serial implementation of the binary trees T and $T^{string}_{pattern}$.

Theorem 5.3 (Complexities of `3SUM` for rational numbers)**.** *Given a list L of n rational numbers $n_i := q_i \in \mathbb{Q}$ on an universe size of $U := 2^b$ distinguishable numbers with $b := 1 + b^{pre} + b^{post}$. The time complexity to determine all valid solutions of* `3SUM` *takes $\mathcal{O}\left(nb + U^{1.58}\right)$. Its belonging space complexity is $\mathcal{O}_{Space}\left(nb + U^{1.58}\right)$.*

Proof. We have one binary tree T and three binary trees $T^{string}_{pattern}$, for each of the three `pattern`s one. With our results of theorem 5.1 and 5.2 we get $nb + 3 \cdot 2^{1.58 \cdot b} = nb + 3 \cdot U^{1.58}$ and hence the complexities are given by $nb + U^{1.58}$. □

5.2 Irrational Numbers

So far, we considered a list L of n rational numbers $n_i := q_i \in \mathbb{Q}$. Now, we will analyse the case of $n_i \in \mathbb{R}$.

In section 3.1 we saw, that we can map the question of zero sum of irrational numbers to the question of zero sum of rational numbers. During our definition of the data type `fixedint` we used this, to define each number n_i by a list of sub numbers $n_{i,j}$ (see step 0) together with a mapping between the rational representing parts of irrational numbers to their belonging irrational numbers.

Theorem 5.4 (Complexities of `3SUM` for real numbers)**.** *Given a list L of n real numbers by $n_i := q_{i,0} + \sum_{\forall i_{a,i,j} \mathbb{I}_a} i_{a,i,j} q_{i,j}$, with $q_{i,j} \in \mathbb{Q}$ and $|\mathbb{I}_a| < \infty$ according our definition 9 and each $q_{i,j}$ on an universe of $U := 2^b$ distinguishable numbers with $b := 1 + b^{pre} + b^{post}$. The time complexity to determine all valid solutions of `3SUM` takes $\mathcal{O}\left(nb|\mathbb{I}_a| + U_{\mathbb{I}_a}^{1.58}\right)$ on an universe of $U_{\mathbb{I}_a} := 2^{b|\mathbb{I}_a|}$ distinguishable numbers n_i. Its belonging space complexity is $\mathcal{O}_{Space}\left(nb|\mathbb{I}_a| + U_{\mathbb{I}_a}^{1.58}\right)$.*

Proof. For rational numbers we only considered the case of $n_i := q_0$. If we extend our numbers to real numbers, we simple have to concatenate $|\mathbb{I}_a| + 1$ (the $|\mathbb{I}_a|$ ones representing atomic irrational ones, plus the one rational part q_0) single b bit long rational cases. We can consider them like one long $b\left(|\mathbb{I}_a| + 1\right)$ bit long integer representation. Have attention that even we go through it like through one integer, at the crossing points from one rational representing to the next, we don't move carriers and successor rules. Instead we start with the carrier and rule situation like for a bit position zero again.

For T it is clear that we go through n numbers of bit length $b\left(|\mathbb{I}_a| + 1\right)$ to sort our list of numbers regarding their bit representation, now.

If we look at $T_{pattern}^{string}$ we have to remember, that we have three `pattern` trees for each rational number. Assume we already went through the first b bits of our new tree T^{string} tree of totally $b\left(|\mathbb{I}_a| + 1\right)$ bits for one specific `pattern`. To get all possible solutions for one rational number, we have to do this also for the two other pattern cases, so that we get a time and space usage of $\approx 2^{1.58 \cdot b}$ until this point. Since, with this we have the complete solution set for our first rational number, we can put this three part solutions together to one large solution set, by putting all part solutions equal to one tree at depth b, so that at each tree node we have now maximal $2^{1.58 \cdot b}$ `ns` values at each node. Have attention, that we know that the maximal number of solutions of three patterns together can be maximal 2^{2b} and hence we still have maximal $2^{1.58 \cdot b}$ tuples `ns` at each of our nodes and not $3 \cdot 2^{1.58 \cdot b}$. We can put the three solution sets together efficiently by considering all three pattern trees as one tree with tree separated data structures for `ns`'s and $\mathcal{C}$'s. Because of symmetry properties of final solution sets given by symmetries in $\mathcal{C}$, we can merge this tree data structures if we reached depth b to one tree, to get our complete tree for the first rational number. In the same way, we now go further through our new merged tree until $2b$ with time and space usage of $2^{1.58 \cdot 2b}$, and so on.

With this we get for the necessary time and also each space $nb\left(|\mathbb{I}_a| + 1\right) + 2^{b\left(|\mathbb{I}_a|+1\right) \cdot 1.58}$. □

5.3 Possible Algorithm Variations

The mentioned algorithm is designed as serial implementation. Since, it is clear that in real applications, only a finite parallelization with also some additional restrictions is possible, we not want to look closer in improvement options and instead only give an overview of them, how our algorithm could be varied, depending on the given physical resources.

- In the presented version, if we have `Tstring.vn == empty`, we still take it to the next step until the end. To let a binary tree branch die and removing the whole branch in this case, would be a much more efficient version regarding time as well as space complexity.
- We receive a power of 1.58 for our universe $U := 2^b$ in our final complexity since we can get until three possible valid combinations for a particular `ns` tuple with a particular rule $\mathcal{C}^\alpha$, during the same time we get for the same `ns` tuple and the second rule $\mathcal{C}^\beta$ one valid combination because of always $\mathcal{C}^\alpha \in \{(1,2), (2,1)\}$ and $\mathcal{C}^\beta \in \{(0,3), (3,0)\}$. By putting all four valid combination determinations and their calculations at first into a common higher saving structure with information pointers regarding

their belonging rule, we can split instead the the total set of four into two sub parts with each two valid combination determinations, for parallel determination of two possiblities. In this way we can reduce the total complexity to U instead of $U^{1.58}$. The disadvantage of this approach is a much higher need of memory space and physical time consuming actions for managing the pointers of rule with node connection informations.

- If we have enough resources, we can use the trees T, $T^{string}_{pattern1}$, $T^{string}_{pattern2}$ and $T^{string}_{pattern3}$ in parallel. For this we have to consider that if we reach depth `bit` in tree T^{string}, we only still need depth `bit + 1` in tree T. So, we could run this two trees in parallel with one step delay, the elder dephts of both trees are no longer necessary and can be removed to save space. We can extend this parallelization of course also on all three pattern trees.

6 Conclusion

TODO

Acknowledgement

Thanks to the private donators who financially support this work.

License

References

[1] Nir Ailon and Bernard Chazelle, *Lower bounds for linear degeneracy testing*, Journal of the ACM (JACM) **52** (2005), no. 2, 157–171.

[2] Ilya Baran, Erik D Demaine, and Mihai Pătraşcu, *Subquadratic algorithms for 3sum*, Workshop on Algorithms and Data Structures, Springer, 2005, pp. 409–421.

[3] Gill Barequet and Sariel Har-Peled, *Polygon containment and tranlational in-hausdorf-distance between segment sets are 3sum-hard*, International Journal of Computational Geometry & Applications **11** (2001), no. 04, 465–474.

[4] Timothy M Chan, *More logarithmic-factor speedups for 3sum,(median,+)-convolution, and some geometric 3sum-hard problems*, ACM Transactions on Algorithms (TALG) **16** (2019), no. 1, 1–23.

[5] Erik D. Demaine, Joseph S. B. Mitchell, and Joseph O'Rourke, *The open problems project*, `http://cs.smith.edu/~jorourke/TOPP/`, 09 2017, (Accessed on 2020/07/30).

[6] ———, *The open problems project - problem 11: 3sum hard problems*, `http://cs.smith.edu/~jorourke/TOPP/P11.html#Problem.11`, 09 2017, (Accessed on 2020/07/30).

[7] ———, *The open problems project - problem 41: Sorting x + y (pairwise sums)*, `http://cs.smith.edu/~jorourke/TOPP/P41.html#Problem.41`, 09 2017, (Accessed on 2020/07/30).

[8] Herbert Edelsbrunner, Joseph O'Rourke, and Raimund Seidel, *Constructing arrangements of lines and hyperplanes with applications*, SIAM Journal on Computing **15** (1986), no. 2, 341–363.

[9] Jeff Erickson et al., *Lower bounds for linear satisfiability problems.*, SODA, 1995, pp. 388–395.

[10] Ari Freund, *Improved subquadratic 3sum*, Algorithmica **77** (2017), no. 2, 440–458.

[11] Anka Gajentaan and Mark H Overmars, *On a class of $O(n^2)$ problems in computational geometry*, Computational geometry **5** (1995), no. 3, 165–185.

[12] Omer Gold and Micha Sharir, *Improved Bounds for 3SUM, k-SUM, and Linear Degeneracy*, 25th Annual European Symposium on Algorithms (ESA 2017) (Dagstuhl, Germany) (Kirk Pruhs and Christian Sohler, eds.), Leibniz International Proceedings in Informatics (LIPIcs), vol. 87, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 42:1–42:13.

[13] Allan Grønlund and Seth Pettie, *Threesomes, degenerates, and love triangles*, 2014.

[14] LH Harper, Thomas H Payne, John E. Savage, and E Straus, *Sorting $x+ y$*, Communications of the ACM **18** (1975), no. 6, 347–349.

[15] IEEE, *IEEE Standard for binary floating-point arithmetic*, ANSI/IEEE Std 754-1985 (1985), 1–20.

[16] ———, *IEEE Standard for floating-point arithmetic*, IEEE Std 754-2008 (2008), 1–70.

[17] ———, *IEEE Standard for floating-point arithmetic*, IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019), 1–84.

[18] Daniel M. Kane, Shachar Lovett, and Shay Moran, *Near-optimal linear decision trees for k-sum and related problems*, 2017.

[19] Tsvi Kopelowitz, Seth Pettie, and Ely Porat, *Higher lower bounds from the 3sum conjecture*, 2014.

[20] Wolfram MathWorld, *Irrational number – from wolfram mathworld*, `https://mathworld.wolfram.com/IrrationalNumber.html`, (Accessed on 2020/09/05).

[21] ———, *Pi*, `https://mathworld.wolfram.com/Pi.html`, (Accessed on 2020/09/05).