Vector Addition Chains

Sam Leder 1360442

Bachelor Final Project Applied Mathematics

Eindhoven University of Technology

Supervisor: Benne de Weger

June 2021

Contents

1 Introduction

For many cryptographic purposes the user has to compute one or multiple exponentiations. Because many users have to do this many times, it is useful to have an efficient method to do these computations quickly. If we need to compute x^k for a fixed k and many different values of x (as is the case in the RSA cryptosystem), addition chains are a helpful tool to greatly reduce the number of calculations needed.
If instead a multi-exponentiation (a product of multiple exponentiations) has to be computed, we can use vector addition chains to speed up this process.

In this report, we briefly look at integer addition chains before focusing on vector addition chains. For this we will consider the existing algorithms by Shamir [Elgamal, 1985] and De Rooij [de Rooij, 1995] and look at their performance. Next we look to create a new algorithm for finding addition chains. As a starting point we use the LLL algorithm for lattice reduction [Lenstra et al., 1982] and modify it to compute addition chains. By choosing a specific starting basis we end up with a vector in the reduced basis such that the steps taken precisely form an addition chain. Then we compare these three algorithms to each other to see which perform best under different circumstances.
After that we look at two general problems regarding addition chains and discuss two interesting papers on specific parts of addition chains.

2 Integer addition chains

Before we consider vector addition chains, let us look at simple integral addition chains. These are sequences that reach a target integer t by adding numbers previously in the chain together, starting at 1. Formally, it is a sequence $v = (v_0, v_1, \ldots, v_s)$ such that

$$v_0 = 1,\ v_s = t \text{ and}$$

$$\text{for all } 1 \leq k \leq s \text{ there are } 0 \leq i, j < k \text{ such that } v_k = v_i + v_j.$$

Here s is the length of the chain. Note that we do not count $v_0 = 1$ in the length, as this element is always given at the start of any chain.
For example, a chain for $t = 25$ could be $(1, 2, 3, 6, 7, 9, 18, 16, 32, 25)$, where

$$\begin{aligned} 1 + 1 &= 2, \\ 2 + 1 &= 3, \\ 3 + 3 &= 6, \\ 6 + 1 &= 7, \\ 7 + 2 &= 9, \\ 9 + 9 &= 18, \\ 9 + 7 &= 16, \\ 16 + 16 &= 32, \\ 16 + 9 &= 25. \end{aligned}$$

Two things stand out in this example: the 18 and 32 are not used to make any other numbers and 32 is even larger than the target. These numbers can be deleted as they do not add anything useful. This brings two new rules for addition chains: every element (other than the target) has to be used to create at least one other element, and no element may be larger than the target.
A better chain would therefore be $(1, 2, 3, 6, 7, 9, 16, 25)$. Note that an addition chain is always ordered, so switching two elements around would also make it invalid.
While it is not difficult to find addition chains, we would like an algorithm that finds short chains efficiently.

2.1 Link with exponentiation

As mentioned, addition chains are very useful for exponentiation. If one wants to compute the power x^k for some x (which does not need to be an integer, it can be anything that can be raised to a power) and $k \in \mathbb{N}$, they can use any addition chain to do this, since multiplying powers comes down to adding exponents together, i.e. $x^a \cdot x^b = x^{a+b}$.
For example, if we want to compute x^{25}, we could calculate $x \cdot x = x^2$, $x^2 \cdot x = x^3$, up to $x^{24} \cdot x = x^{25}$. However, this is clearly very inefficient. Instead, using the addition chain for 25 we just found is much faster: $x \cdot x = x^2$, $x \cdot x^2 = x^3$,

$x^3 \cdot x^3 = x^6$, up to $x^9 \cdot x^{16} = x^{25}$. This only takes 7 multiplications instead of 24, which is an improvement.

2.2 Double-and-add

Double-and-add (also known as square-and-multiply when used for exponentiations) is a method to obtain short integer addition chains. As its name suggests, it consists of doubling and adding numbers in a particular order. The algorithm makes use of the binary representation of the target and simply doubles the result every step and adds 1 for each 1 in the binary representation. The algorithm formulated below is more specifically called left-to-right double-and-add, because it parses the binary representation of t from left to right (i.e. from the most significant bit to the least). There is also right-to-left double-and-add, but this is not as useful in the rest of this report.

Algorithm 1: Left-to-right double-and-add

Input: $t = [t_{k-1} \dots t_0]_2$ binary representation of $t \in \mathbb{N}$, $t_1 = 1$
Output: v addition chain for t

```
v ← {1};
s ← 1;
for i from k − 2 down to 0 do
    s ← 2 · s;
    append s to v;
    if t_i = 1 then
        s ← s + 1;
        append s to v;
    end
end
return v
```

As an example, let again $t = 25$ which is 11001 in binary. We start with $s = 1$ and double it to 2. Then because $t_2 = 1$ we increment it to 3. Both next bits are 0 so we double twice to find 6 and 12. Finally we double and add to get 24 and 25. We get the chain $\{1, 2, 3, 6, 12, 24, 25\}$ of length 6.
In the remainder of the report we will use this approach to produce addition chains for integers and refer with $\mathrm{DA}(t)$ to the addition chain for t created by this algorithm.

One might ask if this algorithm always finds the shortest chain possible, but this is not the case. The minimal counterexample is $t = 15$, for which the double-and-add chain is $\{1, 2, 3, 6, 7, 14, 15\}$ of length 6, while $\{1, 2, 3, 5, 10, 15\}$ only has length 5.
More generally, if $\nu(t)$ denotes the Hamming weight of t (the number of ones in its binary representation), the double-and-add addition chain has length exactly

$$L = \lfloor \log_2(t) \rfloor + \nu(t) - 1.$$

Moreover, since $\nu(t) \leq \lfloor \log_2(t) \rfloor$ trivially holds, we have the bounds

$$\lfloor \log_2(t) \rfloor - 1 \leq L \leq 2 \cdot \lfloor \log_2(t) \rfloor - 1,$$

which does not depend on the exact bits of t, only its size.
While shorter chains exist for most targets, there is no known efficient way to find them, so double-and-add is very useful in that regard.

3 Vector addition chains

A more general problem is that of vector addition chains: instead of an integer we have a target vector $t = (t_1, t_2, \ldots, t_n),\ t_i \in \mathbb{N}$ and instead of starting with 1 we start with the standard basis vectors of $\mathbb{R}^n$. A vector addition chain is a sequence v such that

$$v_{-n+1} = (1, 0, \ldots, 0)$$
$$v_{-n+2} = (0, 1, \ldots, 0)$$
$$\vdots$$
$$v_0 = (0, 0, \ldots, 1)$$
$$\vdots$$
$$v_s = t \text{ and}$$
$$\text{for all } 1 \leq k \leq s \text{ there are } -n+1 \leq i, j < k \text{ such that } v_k = v_i + v_j.$$

Note again that the first element v_1 is the result of the first addition, so the basis vectors have nonpositive indices.
While we could of course create integer addition chains for each t_i and then add them together, there are far more efficient approaches. We will look at an existing algorithm and then formulate a new algorithm.

3.1 Link with multi-exponentiation

Similarly to integer addition chains, we can use vector addition chains to more quickly compute multi-exponentiations, i.e. calculations of the form $x_1^{k_1} \cdot x_2^{k_2} \cdots x_n^{k_n}$, where $n, x_i, k_i \in \mathbb{N}$. For example, if we want to compute $x^8 \cdot y^5$, we could compute x^8 and y^5 separately and multiply them together, which takes 7 calculations:

$$x^2,\ x^4,\ x^8,\ y^2,\ y^4,\ y^5,\ x^8 \cdot y^5,$$

but we can also create the following addition chain for $(8, 5)$ with only 5 calculations:

$$\{(1,0), (0,1), (1,1), (2,1), (4,2), (8,4), (8,5)\}$$

In this case, $(1, 0)$ represents x, $(0, 1)$ represents y, $(4, 2)$ represents $x^4 \cdot y^2$ and so on. Hence adding $(8, 4) + (0, 1) = (8, 5)$ is equivalent to computing $(x^8 \cdot y^4) \cdot y = x^8 \cdot y^5$.

4 Shamir's algorithm

We introduce an algorithm for computing short vector addition chains, due to Shamir [Elgamal, 1985]. This algorithm can be seen as a generalization of double-and-add for integers described in Section 2, only for vectors.
See below a basic version of the algorithm, which does not produce full addition chains but is a good place to start. Therefore we call the output v' a pseudo addition chain.

Algorithm 2: Shamir's basic algorithm

Input: Target $t = (t_1, \ldots, t_n) \in \mathbb{N}^n$, $t_i = [t_{i,k-1} \ldots t_{i,0}]_2$ for $1 \leq i \leq n$
Output: Pseudo addition chain v' for t

$v' \leftarrow \{\}$;
$s \leftarrow (t_{1,1}, \ldots, t_{n,1})$;
$E \leftarrow \{(1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$;
for j from $k-2$ down to 0 **do**
    $s \leftarrow 2 \cdot s$;
    append s to v';
    $a \leftarrow (t_{1,j}, \ldots, t_{n,j})$;
    if $a \notin E$ and $a \neq 0$ **then**
        append a to E;
    end
    $s \leftarrow s + a$;
    append s to v';
end
return v', E

While the double subscript notation for t might look confusing, it is easily cleared up by an example. Let $t = (9, 6, 5) = (1001, 0110, 0101)$, then s gets initialised as $(1, 0, 0)$ (by taking the most significant bit of the coordinates of t). After doubling s, we have $a = (0, 1, 1)$ (the second bit of t's coordinates) and we continue this for every bit in t to find the pseudo chain

$$v' = \{(2, 0, 0), (2, 1, 1), (4, 2, 2), (4, 3, 2), (8, 6, 4), (9, 6, 5)\},$$

$$\text{and } E = \{(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 1), (1, 0, 1)\}.$$

However, we are not done here. As seen in the example above, $(2, 1, 1)$ is not the sum of two preceding vectors as is required. Instead it is the sum of one previous vector and another vector a. It is clear that the latter always exists exclusively of zeroes and ones. We call these vectors $a \in \{0, 1\}^n = \{(b_1, \ldots, b_n) : b_i \in \{0, 1\}, 1 \leq i \leq n\}$ extended basis vectors. While we could just add these vectors to the chain as soon as they occur, this is very inefficient considering there are only $2^n - n - 1$ such vectors (not counting basis vectors and the zero vector), so we can do much better in most cases. Furthermore, we cannot even add a vector $(0, 1, 1)$ immediately, since we first have to take the step

$(0,1,1) = (0,1,0) + (0,0,1)$.
This problem is further discussed in the next section, where we will use the set E to create proper addition chains. For now, we assume that every extended basis vector has been precomputed, so we can use them in our addition chains without issue.

The theoretical value of this approach is quite simple. Let t be the target of dimension n and size b bits, i.e. the largest coordinate of t has b bits in its binary representation. Every adding step and doubling step together cover one bit, so it takes $2b$ steps to reach the target. However, if all bits in a are 0, we do not need to add it, which saves a step. This occurs with probability $\frac{1}{2^n}$ if the numbers are arbitrary, so it might save a few steps if t is large in comparison to n. The number of steps is simply bounded above by $2b$ or equivalently $2 \cdot \lfloor \log_2(t_{\max}) \rfloor$.

4.1 Extended basis vectors

As we saw in Shamir's algorithm, we have to incorporate extended basis vectors in our addition chains. An approach to do so is discussed here.
We collect all used extended basis vectors in a set E (which does not contain duplicates) and use the following algorithm to "complete" the set. A complete set in this context means that every vector is either a regular basis vector or the sum of two other vectors in the set. This completed set can then be added to the start of the chain given in Algorithm 2 to make a proper addition chain. Here the Hamming weight is the number of ones in a vector, so this is equal to the sum of its coordinates. Also, "Break" means moving on to the next element $c \in C$ in the outer loop, as a sum for the current element has already been found.

Algorithm 3: Completion algorithm

Input: Set E containing distinct extended basis vectors
Output: Complete set C

$C \leftarrow E$;
Sort C by decreasing Hamming weight;
for c in C **do**
    $z \leftarrow ()$;
    $M \leftarrow \infty$;
    for a in C **do**
        $b \leftarrow c - a$;
        if $b \in C$ **then**
            Break
        end
        if $b_i \geq 0$ for all $1 \leq i \leq n$ **then**
            if HammingWeight$(b) < M$ **then**
                $z \leftarrow b$;
                $M \leftarrow$ HammingWeight(b);
            end
        end
    end
    insert z into C according to its Hamming weight;
end
reverse C;
return C

We can now define Shamir's algorithm in full. It is very simple, as it only combines the two parts that were created in the previous two algorithms. Recall that an addition chain is an ordered sequence, so we have to put the extended basis before the rest of the chain, as C also already contains the standard basis. This is also why we have to reverse C above to get it sorted by increasing Hamming weight.

Algorithm 4: Shamir's algorithm

Input: Target $t \in \mathbb{N}^n$
Output: Addition chain v for t

$v', E \leftarrow$ ShamirBasicAlgorithm(t);
$C \leftarrow$ CompletionAlgorithm(E);
$v \leftarrow C$ concatenate v';
return v

As an example, let

$$E = \{(1,0,0,0,0),(0,1,0,0,0),(0,0,1,0,0),(0,0,0,1,0),\\ (0,0,0,0,1),(1,1,1,1,1),(0,0,0,1,1),(1,0,1,1,0)\}.$$

First it is sorted into

$$C = \{(1,1,1,1,1),(1,0,1,1,0),(0,0,0,1,1),(0,0,0,0,1),\\ (0,0,0,1,0),(0,0,1,0,0),(0,1,0,0,0),(1,0,0,0,0)\}.$$

Then the first element $c = (1,1,1,1,1)$ is selected. For $a = (1,0,1,1,0)$ the Hamming weight of $b = (0,1,0,0,1)$ is 2, so b and 2 get stored in the variables z and M respectively. Actually M will not be lowered after this, so $(0,1,0,0,1)$ gets inserted into C behind $(0,0,0,1,1)$. This process is repeated for $c = (1,0,1,1,0) = (1,0,0,0,0) + (0,0,1,1,0)$ and $(0,0,1,1,0)$ gets inserted. Then follow $(0,0,0,1,1)$, $(0,1,0,0,1)$ and $(0,0,1,1,0)$ which are all sums of elements of C, so we end up with the completed set

$$C = \{(1,0,0,0,0),(0,1,0,0,0),(0,0,1,0,0),(0,0,0,1,0),\\ (0,0,0,0,1),(0,0,1,1,0),(0,1,0,0,1),\\ (0,0,0,1,1),(1,0,1,1,0),(1,1,1,1,1)\}.$$

To answer the question how many extended basis vectors we need to compute on average for an addition chain of given length, we apply some probability theory. This question is equivalent to "What is the expected number of distinct items Y when drawing with replacement m items from a pool of N equally probable items?"
Let for each item $1 \leq i \leq N$ the indicator be given by

$$X_i = \mathbb{1}\{\text{drawing } i \text{ at least once}\},$$

so that

$$Y = \sum_{i=1}^{N} X_i.$$

Then we can use the rules for the expected value to see

$$\mathbb{E}[Y] = \sum_{i=1}^{N} \mathbb{E}[X_i]$$

and for arbitrary $1 \leq i \leq N$ it holds that

$$\mathbb{E}[X_i] = \mathbb{P}\{\text{drawing } i \text{ at least once}\}$$

$$= 1 - \mathbb{P}\{\text{not drawing } i\}$$

$$= 1 - \left(\frac{N-1}{N}\right)^m,$$

so we find

$$\mathbb{E}[Y] = \sum_{i=1}^{N} \mathbb{E}[X_i] = N \cdot \mathbb{E}[X_1] = N\left(1 - \left(\frac{N-1}{N}\right)^m\right).$$

Going back to the situation of addition chains, $N = 2^n - n - 1$ is the number of extended basis vectors (not counting the standard basis or zero vector) and m is the number of times such a vector gets selected in Algorithm 2 (i.e. k times). Then Y is the number such vectors that get added to E in the algorithm (so not counting duplicates). We would rather say something about the size of C instead of E, but this is beyond the scope.
For example if again $n = 5$, we get $N = 26$. If then the target t has $k = \text{HammingWeight}(t_{\max})$ equal to m, we would like to predict the size of E. The expected value of $Y = |E|$ is shown in Figure 1 and is obviously bounded above by N. We see that $E[Y]$ grows close to N, but never reaches it.

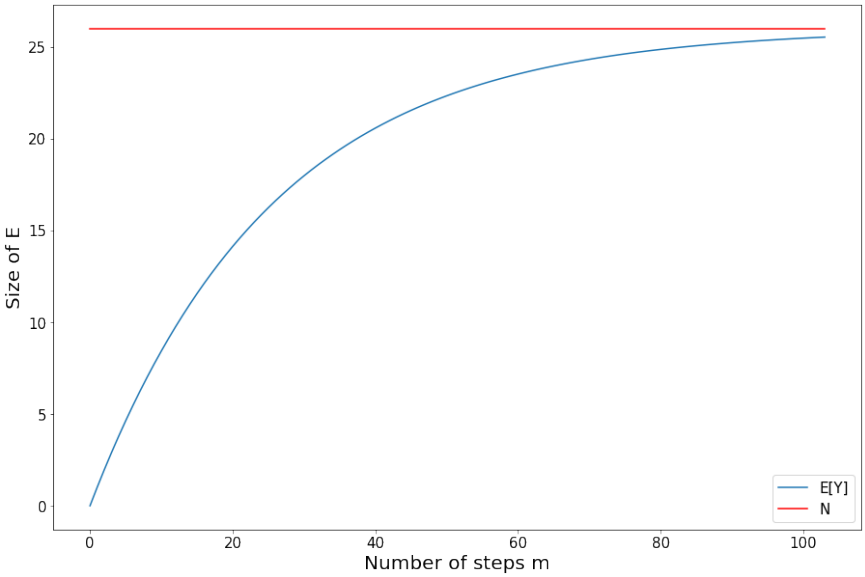


Figure 1: Expected value of the size of E, $n = 5$

5 De Rooij (1995)

The next useful algorithm for vector addition chains is due to De Rooij [de Rooij, 1995]. While the paper is about efficient exponentiation and the main algorithm is described in terms of exponentiations, we slightly adapt this algorithm to make addition chains instead. In this report we will refer to the adapted algorithm for addition chains simply as "De Rooij's algorithm".
The algorithm looks at the two largest coordinates in t and reduces them similarly to Euclid's algorithm.
See the algorithm below. Here max and $next$ are the indices corresponding to the largest coordinates of t, i.e.

$$t_{\max} \geq t_{\text{next}} \geq t_i \text{ for all } i \neq max,\ next.$$

Note that these indices are not necessarily unique, e.g. when $t = (3, 5, 3)$ we have a choice between $next = 1$ and $next = 3$, but this does not matter for the performance of the algorithm. These indices get updated after every step. Also, DA refers to the double-and-added method discussed in Section 2.

Algorithm 5: De Rooij's algorithm

Input: Target $t \in \mathbb{N}^n$
Output: Addition chain v for t

$v \leftarrow \{(1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$;
$u \leftarrow ((1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1))$;
while $t_{\text{next}} > 0$ **do**
  $q \leftarrow t_{\max}$ div t_{next};
  $t_{\max} \leftarrow t_{\max}$ mod t_{next};
  for i in DA(q) **do**
    append $i \cdot u_{\max}$ to v;
  end
  $u_{\text{next}} \leftarrow q \cdot u_{\max} + u_{\text{next}}$;
  append u_{next} to v;
  update indices max and $next$;
end
for i in DA(t_{next}) **do**
  append $i \cdot u_{\max}$ to v;
end
return v

Note that at every point the equality

$$t_1 u_1 + t_2 u_2 + \cdots + t_n u_n = t$$

holds, where t is the initial target vector but the t_i denote the coefficients that get changed throughout the algorithm. We can see this is true as follows.
In the initial state we have

$$t_1 \cdot (1, 0, \ldots, 0) + t_2 \cdot (0, 1, \ldots, 0) + \cdots + t_n \cdot (0, 0, \ldots, 1) = t,$$

which is trivially true. Then in every step there are some $1 \leq i, j \leq n$ such that

$$t_i \leftarrow t_i - q \cdot t_j \text{ and } u_j \leftarrow u_j + q \cdot u_i,$$

so the new value of $t_i u_i + t_j u_j$ is

$$\begin{aligned}
&(t_i - q \cdot t_j)u_i + t_j(u_j + q \cdot u_i) \\
&= t_i u_i - q \cdot t_j u_i + t_j u_j + q \cdot t_j u_i \\
&= t_i u_i + t_j u_j.
\end{aligned}$$

We see that this value does not change, and since the values of t_k and u_k ($k \notin \{i, j\}$) do not change either, the equality

$$t_1 u_1 + t_2 u_2 + \cdots + t_n u_n = t$$

still holds true after every step. So eventually we reach the situation where $t_{\max} = 1$ and every other t_i is zero, so

$$0 \cdot u_1 + 0 \cdot u_2 + \cdots + 1 \cdot u_{\max} + \cdots + 0 \cdot u_n = t$$

and $u_{\max}$ must be equal to t. This is where the algorithm terminates.

Let us look at an example. If $t = (22, 18, 3)$ we get the following results.

Step	u_1	u_2	u_3	t_1	t_2	t_3	q
1	(1, 0, 0)	(0, 1, 0)	(0, 0, 1)	$\underline{\underline{22}}$	$\underline{18}$	3	1
2	(1, 0, 0)	(1, 1, 0)	(0, 0, 1)	$\underline{4}$	$\underline{\underline{18}}$	3	4
3	(5, 4, 0)	(1, 1, 0)	(0, 0, 1)	$\underline{4}$	2	$\underline{3}$	1
4	(5, 4, 0)	(1, 1, 0)	(5, 4, 1)	1	$\underline{2}$	$\underline{\underline{3}}$	1
5	(5, 4, 0)	(6, 5, 1)	(5, 4, 1)	1	$\underline{\underline{2}}$	$\underline{1}$	2
6	(5, 4, 0)	(6, 5, 1)	(17, 14, 3)	$\underline{\underline{1}}$	0	$\underline{1}$	1
7	(5, 4, 0)	(6, 5, 1)	(22, 18, 3)	0	0	1	

In every step $t_{\max}$ is underlined twice and t_{next} is underlined once. We see that the indices *max* and *next* get updated after every step. In every step we check how often t_{next} goes into $t_{\max}$, which we call q, and then subtract t_{next} q times from $t_{\max}$ and add $u_{\max}$ q times to u_{next}.
To make it into a proper addition chain, we need to create an integer addition chain for q and multiply it with the correct vector. For example in step 2 we need to add $4 \cdot u_2$ to u_1. We have the simple addition chain $\{1, 2, 4\}$ for q, so we add $\{(1,1,0), (2,2,0), (4,4,0)\}$ to our chain before we can add $(5, 4, 0)$ as well. We find this chain of length 9:

$$\{(1,0,0), (0,1,0), (0,0,1), (1,1,0), (2,2,0), (4,4,0),$$

$$(5,4,0), (5,4,1), (6,5,1), (12,10,2), (17,14,3), (22,18,3)\}.$$

Now that we know how the algorithm works, we will discuss its performance in terms of t. This will depend on n, the dimension of t, and $t_{\max}$, the largest coordinate of t (not to be confused with the variable in the algorithm).
If we fix the dimension to an arbitrary integer, say $n = 5$, we can graph the chain length against $t_{\max}$ for a large number of random targets, see Figure 2. For this 1000 random vectors have been generated, for which an addition chain and its length have been calculated.

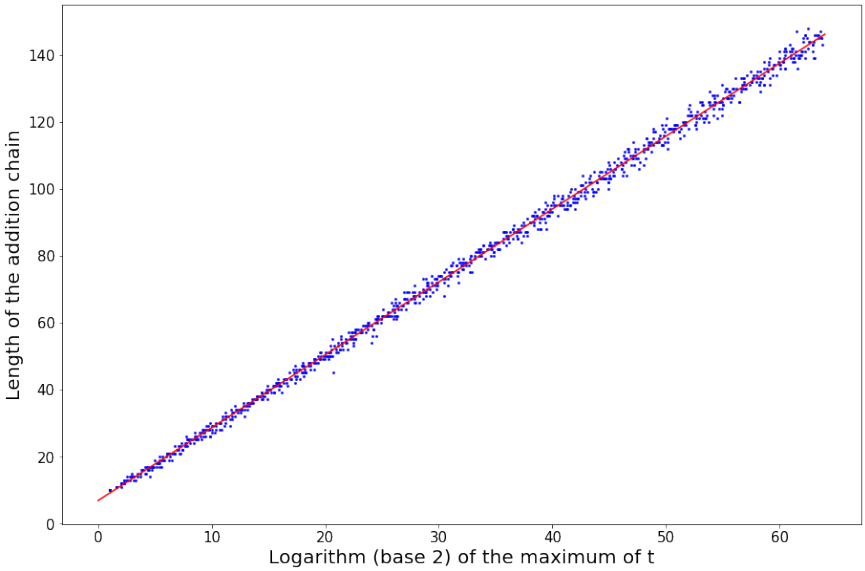


Figure 2: De Rooij, $n = 5$

Note that the x-axis has a logarithmic scale and the values of $t_{\max}$ range from 0 to 2^{64}. Also plotted is the best fit line, and we see that this relation is strongly linear. The slope of the line in this case is 2.18.
Graphs for other values of n give a very similar picture, so we have that

$$L = \alpha_n \log_2(t_{\max}),$$

where L is the length of the addition chain. We have that $\alpha_5 = 2.18$, but we might be interested to see more of these values. In Figure 3 we plot the experimental values for α_n for n up to 250.

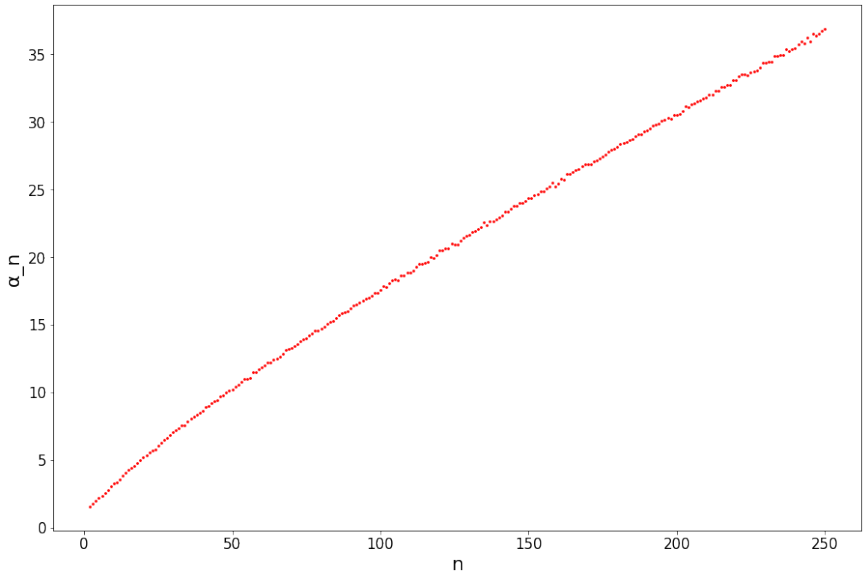


Figure 3: Values for α_n, $n \leq 250$

While the graph may look slightly linear again, this is definitely not the case: it significantly bends. It is an interesting problem to find a formula for α_n and therefore a function for L in terms of t, and to this end we will now consider a specific target $t \in \mathbb{N}^n$ in order to find a theoretical approximation.

Let $1 < \tau < 2$ such that $\tau^n - \tau^{n-1} - 1 = 0$. It is clear that such a τ exists and is unique for any $n \geq 2$, because

$$1^n - 1^{n-1} - 1 = -1 \text{ and } 2^n - 2^{n-1} = 2^{n-1} - 1 > 1$$

and $f_n(x) = x^n - x^{n-1} - 1$ is continuous and $f_n'(x) = nx^{n-1} - (n-1)x^{n-2} > 0$ for $x \geq 1$, so the existence follows from the Mean Value Theorem and the uniqueness from the fact that f_n is increasing on $(1, 2)$.

Now let $t = (\tau^p, \tau^{p-1}, \ldots, \tau^{p-n+1})$ for some $p > n$, which has a special property. The algorithm will go as follows:

$$q = \tau^p \text{ div } \tau^{p-1} = \lfloor \tau \rfloor = 1$$

$$\text{and } \tau^p - \tau^{p-1} = \tau^{p-n} \text{ by the definition of } \tau,$$

so t becomes $(\tau^{p-1}, \tau^{p-2}, \ldots, \tau^{p-n})$, which is the same as it was but with a decremented exponent. This process continues for p steps until the algorithm terminates. We see that $q = 1$ in every step of the algorithm, which is unique for this choice of t and causes the addition chain to be relatively long.
One thing to note is that powers of τ are not integers, so they will have to be

rounded, but the effect is still the same.

We have $f_n(x) = x^n - x^{n-1} - 1 = (x-1)x^{n-1} - 1$ and $f_n(\tau_n) = 0$.
We claim that $\tau_n \approx 1 + \frac{\log(n)}{n}$ with the following argument:

$$\text{Let } \tau_n = 1 + c_n \cdot \frac{\log(n)}{n}$$

$$\log\left(1 + c_n \cdot \frac{\log(n)}{n}\right)^n = n \log\left(1 + c_n \cdot \frac{\log(n)}{n}\right)$$

$$\approx n \cdot c_n \cdot \frac{\log(n)}{n} = c_n \log(n) = \log\left(n^{c_n}\right)$$

Now

$$0 = f_n(\tau_n) = (\tau_n - 1)\tau^n \cdot \frac{1}{\tau} - 1$$

$$\approx c_n \cdot \frac{\log(n)}{n} \cdot n^{c_n} \cdot \frac{1}{1 + c_n \cdot \frac{\log(n)}{n}} - 1 = \frac{c_n \cdot \log(n) \cdot n^{c_n - 1}}{1 + c_n \cdot \frac{\log(n)}{n}} - 1$$

Now, if $c_n < 1$ this expression becomes negative and if $c_n > 1$ it becomes positive. Since the expression is zero, we conclude that $c_n \approx 1$.

The addition chain for $(\tau^p, \tau^{p-1}, \ldots, \tau^{p-n+1})$ has length p, so

$$p \approx \alpha_n \log_2((\tau_n)^p)$$

$$\implies \alpha_n \approx \frac{1}{\log_2(\tau_n)} \approx \frac{1}{\log_2(1 + \frac{\log_2(n)}{n})} \approx \frac{1}{\left(\frac{\log_2(n)}{n}\right)} = \frac{n}{\log_2(n)}.$$

In Figure 4, we see a comparison between the theoretical values described above and some randomly generated values for $n = 5$. In this case $\tau \approx 1.325$, and the length of the addition chains of $(\tau^p, \tau^{p-1}, \ldots, \tau^{p-n+1})$ is shown for $2 \leq p \leq 160$, as well as addition chains some randomly generated targets. These theoretical values are on the high side, but do not form an upper bound for this algorithm. Furthermore, we see that these values curve down at the end of the graph, however this is most likely due to an implementation issue than actual performance of the algorithm.

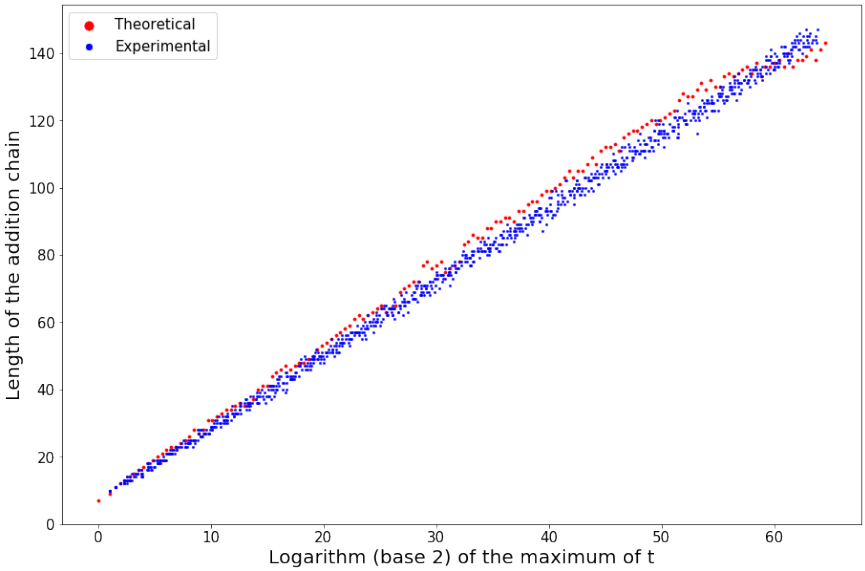


Figure 4: Theoretical values compared to experimental values, $n = 5$

A further interesting question is the following. The algorithm always subtracts the second largest coordinate from the largest, but we could instead subtract a smaller coordinate from the largest one, which would give smaller remainders and therefore presumably finish quicker. However, in practice this does not seem to be the case. It is beyond the scope of this report to find out and properly explain why this is true.

6 A new algorithm

6.1 LLL lattice basis reduction

In 1982 Lenstra, Lenstra and Lovász created an algorithm to quickly reduce a lattice basis [Lenstra et al., 1982]. A lattice that has a basis $\mathbf{b} = \{b_1, \ldots, b_n\}$ (where $b_i \in \mathbb{R}^n$ for $1 \leq i \leq n$) can be written as the set

$$\mathcal{L} = \left\{ \sum_{i=0}^{n} \lambda_i b_i : \lambda_i \in \mathbb{Z} \right\}.$$

If for the basis $\mathbf{b}$ the Gram-Schmidt basis is $\mathbf{b}^* = \{b_1^*, \ldots, b_n^*\}$ and its Gram-Schmidt coefficients are

$$\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle} \text{ for all } 1 \leq j < i \leq n,$$

then $\mathbf{b}$ is called LLL-reduced if it satisfies

1. $|\mu_{i,j}| \leq \frac{1}{2}$ for all $1 \leq j < i \leq n$,
2. $\frac{3}{4}||b_{k-1}^*||^2 \leq ||b_k^*||^2 + \mu_{k,k-1}^2 ||b_{k-1}^*||^2$ for all $2 \leq k \leq n$.

The second condition is called the Lovász condition. In words, these conditions demand that the vectors are relatively short and nearly orthogonal.
Below is a formulation the algorithm, based on [Hoffstein, 2008].

Algorithm 6: LLL algorithm

Input: Basis $\mathbf{b}$ of a lattice
Output: Reduced basis $\mathbf{b}$ of the same lattice

$\mathbf{b}^* \leftarrow \text{GramSchmidt}(\mathbf{b}) = \{b_1^*, \ldots, b_n^*\}$ (without normalization);
$k \leftarrow 2$;
$\mu_{i,j} \leftarrow \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}$ for all $1 \leq j < i \leq n$;
while $k \leq n$ **do**
  for j from k to 1 **do**
    if $|\mu_{k,j}| > \frac{1}{2}$ **then**
      $b_k \leftarrow b_k - \text{round}(\mu_{k,j}) \cdot b_j$;
      $\mathbf{b}^* \leftarrow \text{GramSchmidt}(\mathbf{b}) = \{b_1^*, \ldots, b_n^*\}$;
    end
  end
  if $\frac{3}{4}||b_{k-1}^*||^2 \leq ||b_k^*||^2 + \mu_{k,k-1}^2 ||b_{k-1}^*||^2$ **then**
    $k \leftarrow k + 1$;
  else
    $b_{k-1}, b_k \leftarrow b_k, b_{k-1}$;
    $b_{k-1}^*, b_k^* \leftarrow b_k^*, b_{k-1}^*$;
    $k \leftarrow \max\{k - 1, 2\}$;
  end
end
return b

As an example, consider the basis $\mathbf{b} = \{(9, 5, 1), (3, 5, 2), (7, 7, 8)\}$. To reduce $\mathbf{b}$, we go through the algorithm.
First, $\mu_{2,1} = 0.505$ which gets rounded to 1, so we get $b_2 = (3, 5, 2) - (9, 5, 1) = (-6, 0, 1)$. The Lovász condition is not satisfied, so we swap b_1 and b_2.
Now $\mu_{2,1} = -1.43$, so $b_2 = (9, 5, 1) + (-6, 0, 1) = (3, 5, 2)$ and the Lovász condition is satisfied, so we move on to $\mu_{3,2} = 1.84$ and $b_3 = (7, 7, 8) - 2 \cdot (3, 5, 2) = (1, -3, 4)$. We have ended up with $\mathbf{b} = \{(-6, 0, 1), (3, 5, 2), (1, -3, 4)\}$, which is an LLL-reduced basis.

6.2 LLL-based vector addition chains

Now that we have seen an addition chain algorithm and the lattice reduction algorithm, we propose a new approach which uses the LLL algorithm to find vector addition chains in reasonable time. By choosing a basis that has a specific vector as one of the shortest vectors in the lattice, we are almost guaranteed to find an addition chain for t by reducing the basis. Concretely, if $t = (t_1, \ldots, t_n)$ is our target, let the basis be given by

$$B_t = \begin{pmatrix} 1 & Kt_2 & 0 & 0 & \ldots & 0 & 0 \\ 0 & -Kt_1 & Kt_3 & 0 & \ldots & 0 & 0 \\ 0 & 0 & -Kt_2 & Kt_4 & \ldots & 0 & 0 \\ \vdots & & & & & & \\ 0 & 0 & 0 & 0 & \ldots & -Kt_{n-2} & Kt_n \\ 0 & 0 & 0 & 0 & \ldots & 0 & -Kt_{n-1} \end{pmatrix}$$

where $K > 0$ is a constant to be determined and b_i denotes the i^{th} row of B and these rows form a basis of $\mathbb{R}^n$. Now

$$s := tB = \sum_{i=1}^{n} t_i b_i = (t_1, 0, 0, \ldots, 0)$$

is a short vector. Denote the operations on B in the LLL algorithm by the matrix U so that $UB = B^*$ is the reduced basis. If s is indeed the first reduced basis vector (the first row of B^*), we have that t is the first row of U and we can find an addition chain for t.

Next we come to the choice of K. We have

$$\det(B_t) = (-1)^{n-1} K^{n-1} \prod_{i=1}^{n-1} t_i \approx (Kt_{\max})^{n-1}.$$

Since the length of the reduced basis vectors is approximately

$$(\det(B_t))^{\frac{1}{n}} \approx (Kt_{\max})^{\frac{n-1}{n}},$$

we expect that $(t_1, 0, \ldots, 0)$ is in the reduced basis if

$$t_{\max} \ll (Kt_{\max})^{\frac{n-1}{n}}$$

$$(t_{\max})^n \ll (Kt_{\max})^{n-1}$$

$$t_{\max} \ll K^{n-1}$$

$$K \gg (t_{\max})^{\frac{1}{n-1}}.$$

See below the algorithm. Here B_t denotes the basis described above.

Algorithm 7: LLL-based addition chain

Input: Target $t \in \mathbb{N}^n$
Output: Addition chain v for t

$v \leftarrow \{(1,0,\ldots,0),\ldots,(0,0,\ldots,1)\}$;
$u \leftarrow ((1,0,\ldots,0),\ldots,(0,0,\ldots,1))$;
$\mathbf{b} \leftarrow B_t$;
$\mathbf{b}^* \leftarrow \text{GramSchmidt}(\mathbf{b}) = \{b_1^*,\ldots,b_n^*\}$ (without normalization);
$k \leftarrow 2$;
$\mu_{i,j} \leftarrow \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}$ for all $1 \leq j < i \leq n$;
while $k \leq n$ **do**
  for j from k to 1 **do**
    if $|\mu_{k,j}| > \frac{1}{2}$ **then**
      for i in $\text{DA}(-\text{round}(\mu_{k,j}))$ **do**
        append $i \cdot u_j$ to v;
      end
      $b_k \leftarrow b_k - \text{round}(\mu_{k,j}) \cdot b_j$;
      append $u_k - \text{round}(\mu_{k,j}) \cdot u_j$ to v;
      $\mathbf{b}^* \leftarrow \text{GramSchmidt}(\mathbf{b}) = \{b_1^*,\ldots,b_n^*\}$;
    end
  end
  if $\frac{3}{4}||b_{k-1}^*||^2 \leq ||b_k^*||^2 + \mu_{k,k-1}^2||b_{k-1}^*||^2$ **then**
    $k \leftarrow k+1$;
  else
    $b_{k-1}, b_k \leftarrow b_k, b_{k-1}$;
    $b_{k-1}^*, b_k^* \leftarrow b_k^*, b_{k-1}^*$;
    $u_{k-1}, u_k \leftarrow u_k, u_{k-1}$;
    $k \leftarrow \max\{k-1, 2\}$;
  end
end
return v

Let us now consider the same example as we did for De Rooij's algorithm, $t = (22, 18, 3)$.
The basis we use is $\mathbf{b} = \{(1, 792, 0), (0, -968, 132), (0, 0, -792)\}$ where we choose $K = 44$. By reducing this basis and saving the adding and swapping steps we get the addition chain

$$\{(1,0,0),(0,1,0),(0,0,1),(1,1,0),(2,2,0),(3,3,0),$$

$$(4,3,0),(4,3,1),(6,5,1),(12,10,2),(18,15,3),(22,18,3)\}.$$

A problem of the algorithm is that it does not always work. The LLL algorithm stops when the Lovász condition is satisfied, which is always the case when we reach the vector s, but sometimes it is satisfied earlier. This means that the algorithm stops before the addition chain has reached its target t. This is the case even when we take K high enough.
Furthermore, the LLL algorithm allows negative numbers, which addition chains do not. This means we end up with an addition-subtraction chain, which can be useful in some situations but are also impractical in others. One solution is to always round down coefficients instead of rounding to the nearest integer, which results in only positive numbers. However, this causes the algorithm to stop too early even more often. Therefore we try to balance these two options with a custom rounding method: Choose a constant $\alpha \in [0.5, 1)$ (we will use $\alpha = 0.9$) and only round up if the decimal part of a number is larger than α.

To illustrate this, we generate 1000 random targets up to ten million and calculate the addition chain for each of them. When simply rounding to the closest integer, in 3% of cases the target was not reached and in 12% there were negative numbers included. On the other hand, if we use custom rounding, 13% does not reach the target and 11% uses negative numbers. We conclude that the custom method performs slightly better in terms of negative number, but is much worse in reaching the target. Hence it is not an improvement on the previous method, which we will keep using in the following.

7 Comparisons

7.1 Shamir and De Rooij

First we come to the comparison of the two existing algorithms. We compare them twice: once with precomputations and once without.
If we allow extended basis vectors to be precomputed as in Algorithm 2, Shamir's algorithm as an advantage. In Figure 5 we see that both algorithms have a strong logarithmic relation to the size of the target, but they differ in performance. While De Rooij does slightly better for $n = 2$ and $n = 3$, Shamir is better for every dimension $n \geq 5$, with the gap between the two growing bigger for larger n.

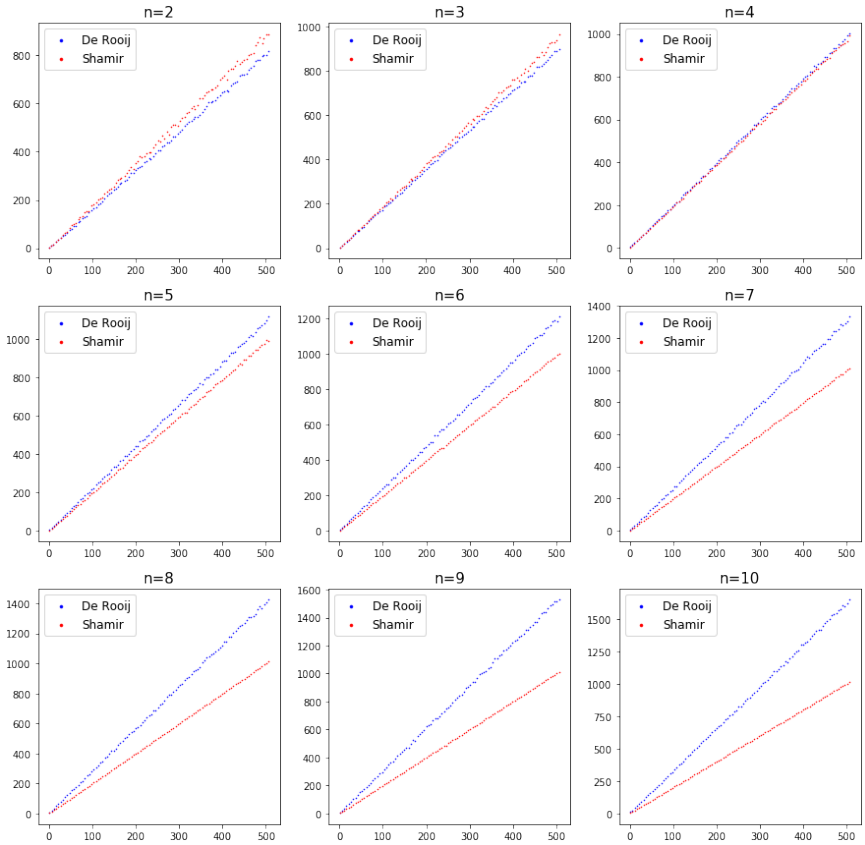


Figure 5: Comparison between De Rooij and Shamir (with precomputations)

In Figure 6 we see a somewhat similar picture. The only difference is that the extended basis vectors have not been precomputed, so they have to be added to the addition chain as well, which worsens the performance of Shamir's algorithm. The values for De Rooij do not change.

Where the plots followed a straight line above, there is a distinguishable downwards curve now. This can be explained by the probability theory described in Section 4. Initially the algorithm takes an additional step each time to add an extended basis vector, but as the chain grows longer the probability of having had the particular extended basis vector grows, so it does not take an extra step.
This means that while the algorithms are now closer together for small targets, the results for longer targets is almost the same as before, so that Shamir still outperforms De Rooij.

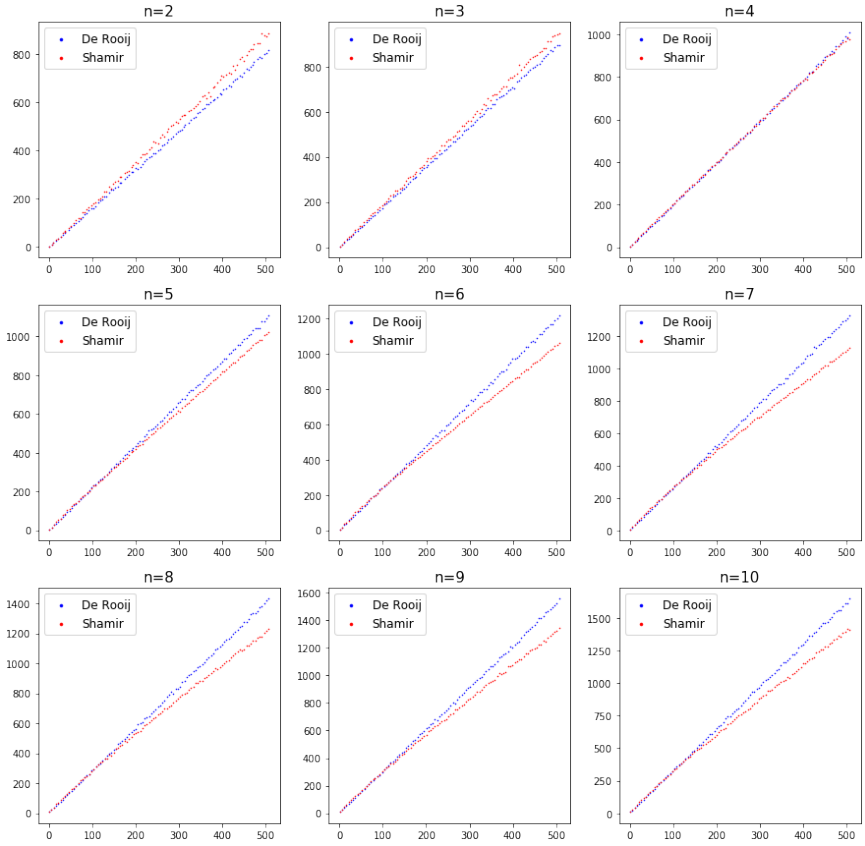


Figure 6: Comparison between De Rooij and Shamir (without precomputations)

7.2 De Rooij and LLL

Now we move on to comparing our own LLL algorithm against that of De Rooij. In Figure 7, we see a comparison between the two algorithms. The results for De Rooij are the same as in Figure 2, with the LLL results added as well.

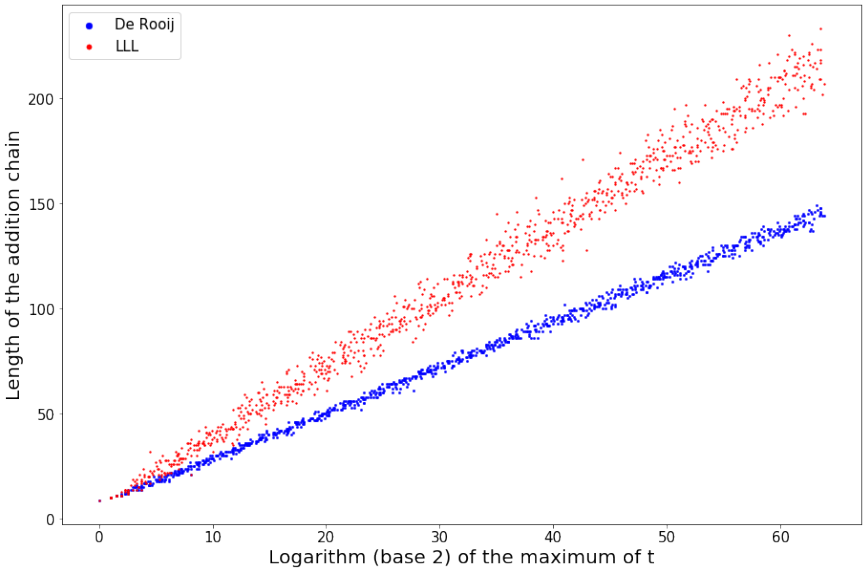


Figure 7: Comparison between De Rooij and LLL, $n = 5$

It is clear that De Rooij performs much better and on top of that has less variance in its length. We saw that the slope for De Rooij was 2.18, while for LLL it is 3.34. This is about half more, which is quite a lot. From this we can conclude that the LLL algorithm is not a useful way to find addition chains.
We might also ask what the relation is between the dimension and the slope of the graph (5 and 3.34 respectively in this case), similarly to the considerations we had for De Rooij's algorithm. However, this algorithm is more complex to predict and so we will not treat this here.

8 Various problems

8.1 Addition chain with given basis

Let the basis $\{b_1, ..., b_n\}$ and the target $t \in \mathbb{N}^n$ be given. The goal is to construct an addition chain for t starting from the given basis instead of the standard basis $\{(1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$.

To achieve this, solve the system

$$a_1 \cdot b_1 + ... + a_n \cdot b_n = t$$

for $a_i > 0$. The solution set of this equation forms a cone in $\mathbb{R}^n$, so there might or might not be a solution. If there is a solution t (or equivalently if t is in the cone) an addition chain is possible; otherwise it is not.
If it is, we make an addition chain with the b_i, this is the same as an addition chain for $(a_1, ..., a_n)$ with the standard base.

For example, let $t = (7, 1, 10)$, $b = \{(1, 1, 1), (2, 0, 3), (0, 1, 0)\}$.
Then we find

$$t = (7, 1, 10) = 1 \cdot (1, 1, 1) + 3 \cdot (2, 0, 3) + 2 \cdot (0, 1, 0).$$

With this, we can use De Rooij's algorithm to make an addition chain:
$\{(1, 1, 1), (2, 0, 3), (0, 1, 0), (2, 1, 3), (4, 2, 6), (6, 2, 9), (7, 3, 10)\}$.
Note that this follows the same steps as a "normal" addition chain for $(1, 3, 2)$:
$\{(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 1, 1), (0, 2, 2), (0, 3, 2), (1, 3, 2)\}$.

8.2 Addition chain with intermediate stop

Given targets s and t, find an addition chain through s to t.

For this, first make a "normal" addition chain for s.
Try using the last n vectors of this chain as a basis and try to make a chain for t as before.
If this is not possible, replace one of the vectors by an earlier vector from the chain and try again. By doing this we enumerate over all n-tuples in the chain. In this way we find the largest n independent vectors in the chain.
An important fact to note here is that we do not have to use multiples for this. That is, the addition chain for s will have different multiples of the same vector, like $(4, 1, 2)$, $(8, 2, 4)$, $(12, 3, 6)$. These cannot be used in the same basis, since they are not linearly independent. This means we can skip these multiples to greatly speed up the process.
Repeat this step until a valid basis is found. This always succeeds, because eventually one arrives at the standard basis again, as the chain for s starts here.

For example, let $s = (4, 3, 1)$, $t = (7, 6, 4)$.
An addition chain for s is

$$\{(1,0,0), (0,1,0), (0,0,1), (1,1,0), (2,2,0), (3,3,0), (4,3,0), (4,3,1)\}.$$

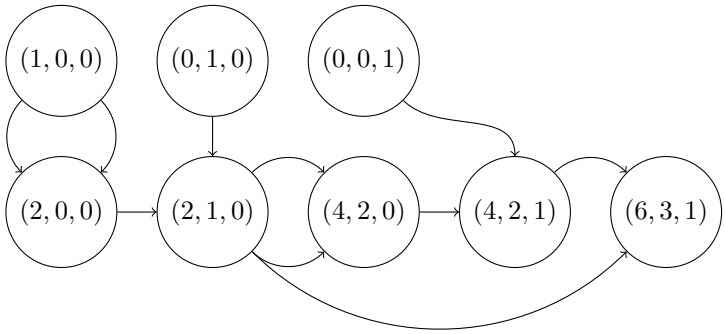
Then we try to find a valid basis that can form an addition chain for t, and we find $\{(4,3,1), (3,3,0), (0,0,1)\}$. This addition chain becomes

$$\{(4,3,1), (3,3,0), (0,0,1), (0,0,2), (0,0,3), (4,3,4), (7,6,4)\}.$$

8.3 Addition chains and graphs

In his 2011 paper *Calculating optimal addition chains,* Clift describes how integer addition chains can be expressed in terms of graphs and shows some useful properties [Clift, 2011]. We will now look at a similar approach for vector addition chains.

The graphs are created by making a node for every element of the addition chain and indicating with two directed edges which elements were added together to create this element.



Above is an example of a chain for $t = (6, 3, 1)$:

$$\{(1,0,0), (0,1,0), (0,0,1), (2,0,0), (2,1,0), (4,2,0), (4,2,1), (6,3,1)\},$$

where we can clearly see how the chain was created.

It is clear that every node (except the starting nodes) has indegree 2, since every element is created by adding exactly two other elements together. Also, every node except the ending node has an outdegree of at least one, since otherwise it would not be used in the addition chain and should be removed. The original paper goes into more depth on properties of these graphs, but we will not consider this further.

8.4 Link between integer and vector addition chains

In the 2021 paper *Addition chains, vector chains, and efficient computation*, Thurber and Clift describe an interesting link between integer addition chains and their vector counterparts [Thurber and Clift, 2021].
This is perhaps best illustrated by an example. Consider the following addition chain for $t = 177$:

$$\{1, 2, 3, 5, 10, 20, 40, 43, 86, 172, 177\}.$$

We can "unwind" the chain as follows, where the vector denotes the current coefficients:

$$\begin{aligned}
177 &= 177, \ [1] \\
&= 5 + 172, \ [1, 1] \\
&= 5 + 2 \cdot 86, \ [1, 2] \\
&= 5 + 4 \cdot 43, \ [1, 4] \\
&= 5 + 4 \cdot (3 + 40) = 5 + 4 \cdot 3 + 4 \cdot 40, \ [1, 4, 4] \\
&= 5 + 4 \cdot 3 + 4 \cdot (2 \cdot 20) = 5 + 4 \cdot 3 + 8 \cdot 20, \ [1, 4, 8] \\
&= 5 + 4 \cdot 3 + 8 \cdot (2 \cdot 10) = 5 + 4 \cdot 3 + 16 \cdot 10, \ [1, 4, 16] \\
&= 5 + 4 \cdot 3 + 16 \cdot (2 \cdot 5) = 4 \cdot 3 + 33 \cdot 5, \ [4, 33] \\
&= 4 \cdot 3 + 33 \cdot (3 + 2) = 33 \cdot 2 + 37 \cdot 3, \ [33, 37] \\
&= 33 \cdot 2 + 37 \cdot (2 + 1) = 70 \cdot 2 + 37, \ [70, 37] \\
&= 70 \cdot (2 \cdot 1) + 37 = 177, \ [177]
\end{aligned}$$

We see for example that $177 = 5 + 4 \cdot 3 + 4 \cdot 40$ corresponds to the vector $[1, 4, 4]$, so computing 177 from this point is equivalent (follows the same steps) as computing $[1, 4, 4]$. $[1, 0, 0] = 5$, $[0, 1, 0] = 3$, $[0, 0, 1] = 40$, $[0, 1, 1] = 43$, $[0, 2, 2] = 86$, $[0, 4, 4] = 172$ and $[1, 4, 4] = 177$.

References

[Clift, 2011] Clift, N. M. (2011). Calculating optimal addition chains. *Computing (Vienna/New York)*, 91:265–284.

[de Rooij, 1995] de Rooij, P. (1995). Efficient exponentiation using precomputation and vector addition chains. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 950:389–399.

[Elgamal, 1985] Elgamal, T. (1985). A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472.

[Hoffstein, 2008] Hoffstein, J. (2008). *An Introduction to Mathematical Cryptography*, chapter Lattices and Cryptography. Springer, New York, NY.

[Lenstra et al., 1982] Lenstra, A. K., Lenstra, H. W., and Lovász, L. (1982). Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534.

[Thurber and Clift, 2021] Thurber, E. G. and Clift, N. M. (2021). Addition chains, vector chains, and efficient computation. *Discrete Mathematics*, 344(2):1–15.