Addition Chains

efficient computing of powers

Bachelor Project

Sander van der Kruijssen
1335081
June 2005 – oktober 2007, Amsterdam

Abstract

This paper covers the subject of addition chains. Addition chains are used to ease calculations on power, by minimizing the total number of computations needed to generate such a power. A chain for a given number *n* is a row of integers, for which

$1 = a_0, a_1, a_2, \dots , a_r = n$

with the additional restriction that

$a_i = a_j + a_k$, where $k \leq j < i$ for all $i = 1, 2, ..., r$

Many different algorithms have been devised to calculate both relatively short and minimal addition chains. However, no *known* algorithm can calculate the minimal chainlength within both polinomial time and/or memory constraints.
Using a computer program, the binary 'approach algorithm' proved to perform worst of all tested algorithms. Of these methods, the power tree algorithm and window algorithm perform best.
The constraints of the those algorithms which produce minimal chains becomes apparant when looking at their implementation in the computer program: they are incapable of producing a result for every goal. However, when looking at those goals for which they do produce a result, the depthfirst method seems to perfom best.
Further work on this subject might improve the implementation on computer as well as (parts of) the minimal algorithms. This would enable us to find addition results or to find known results using less time and memory.

1 Table of Contents

2. Introduction

This bachelor thesis concludes my bachelor studies at last. I started working on this project in the summer of 2005. After working on it for some periods of time spread out over the past two years, I started working on it again in the summer of 2007 and finally managed to finish it.

I first came in contact with addition chains, the subject of this paper, when I was helping out at a mathematics day on my old high school [a]. The students were supposed to discover and play with the basics of the topic. I found the subject to be interesting, and it remained on my mind for a while after that. When it was time to choose a subject for the bachelor project, addition chains immediately crossed my mind as an option. The reason why addition chains first appealed to me is because its basics are quite simple, but after the first introduction there is still plenty to discover. For example, there are still some open problems remaining to be solved.

In this paper, I will try to present some of the most important findings on addition chains. First, I will give a general introduction on the subject, explaining the basics of the chains. The next chapters include some of the most basic and important findings and terms on addition chains, as well as some closely related problems.
Furthermore, some alternative representations of addition chains will be given.

A lot of algorithms are known to calculate short chains. The most important and well-known ones are given in chapter 8. This is followed by a chapter on the upper and lower bounds of chains and some algorithms that compute minimal chains.

I created a computer program in order to be able to compare the different algorithms. This program can be used to calculate addition chains using some of the algorithms presented in chapters 8 and 10. More on the computer program itself can be found in chapter 11. In chapter 12, the results of using the program are discussed. The program itself can be found at www.few.vu.nl/~savdkrui/bp/AdditionChains.zip. The code of the program is also added in appendix III.

Which of the algorithms works best can be found in the conclusion in chapter 13. Some possible improvements of the presented algorithms are added in chapter 14.

This paper itself can be found at www.few.vu.nl/~savdkrui/bp/AdditionChains.doc and www.few.vu.nl/~savdkrui/bp/AdditionChains.pdf

3. General introduction to addition chains

In the introduction, the term addition chain has been mentioned a couple of times. Now it is time to take a closer look at these chains. Addition chains form a subject that has been studied over a long time. As early as 200 years BC, material can be found that is related to the problem [f(Knuth)]. In Indian and Arabic works of the tenth and eleventh century, the problem is also mentioned, together with some proposed solutions. Since the first half of the last century, quite an amount of papers has been published on this subject (see chapter 15 for just a small selection of interesting papers). More on the history of addition chains can be found in [f(Knuth)]

Now what is actually meant by the term addition chain? Addition chains are, perhaps not that surprisingly, chains of numbers which are created using additions. The term 'addition chain' was first mentioned by Arnold Scholz in 1937 (Jahersbericht der deutschen Mathematiker-Vereinigung, class II, 47 (1937), 41-42) [f(knuth)].
Not every chain of numbers can be called an addition chain: there are certain rules that need to be complied to. But let us first have a short peek at the use of the chains, before digging into these rules.

In many areas, it is needed to compute the power of a certain number. Computing an integer x^n can be done this by multiplying x with itself n times. In this way, $n-1$ multiplications are needed in total to calculate x^n. In general however, multiplication is a costly operation on a computer: multiplying two numbers uses more instruction than, for instance, adding them. Therefore, it can be useful to try and minimize the total number of operations that are needed, whenever this is possible. And this is where addition chains come into play: they can be used for efficient computation of powers [b(wattel) and others].

Finally, it is time to look at the addition chains themselves. As mentioned, these chains consist of numbers. A chain works up to a certain goal: the number n. The rules that the chains need to comply to are as follows: [a(Wiskunde B-dag), b(wattel), d(bos), f(knuth) and many others]

- The first element of the chain is 1,
- The last element is the goal number, n,
- And each element, except for the first, is created by adding two other (not necessarily different) numbers that occur earlier in the chain.

This somewhat informal definition can be made a bit more strict and formal: ([f(knuth), g(bergeron) and many others]

An addition chain for a given number n is a row of integers, for which

$1 = a_0, a_1, a_2, \dots , a_r = n$

with the additional restriction that

$a_i = a_j + a_k$, where $k \leq j < i$ for all $i = 1, 2, ..., r$

Now, the question remains: how can chains such as these be used to compute powers efficiently? Addition chains use the fact that $x^p * x^q = x^{(p+q)}$ [c(sauerbrey) and others]. This simple rule enables us to calculate powers far more efficiently than with the naive method, which multiplies by x itself repeatedly.
This definition for the correspondence between addition chains and computing powers can also be given more formally:

For the given number n, take the addition chain $a_0, a_1, ..., a_k.$
Now, one can form a^n as follows: $x = x^{n0}, x^{n1}, ..., x^{nk} = x^n$ [b (wattel)], where the chain $x^{n0}, x^{n1}, ..., x^{nk}$ abides the rules for addition chains mentioned before.

Thus, the shorter the addition chain the less multiplications are needed.

Time to look at an example. Let us try to create an addition chain for *n = 55.* The following chain is presented in the form a_r = b (a_i + a_j). a_x is the identification of the current step. b is the number that is calculated in the current step. (a_i + a_j) is used to represent which two numbers are added in the current step. (1)

a_0 = 1
a_1 = 2 (a_0 + a_0)
a_2 = 3 (a_1 + a_0)
a_3 = 4 (a_2 + a_0)
a_4 = 7 (a_3 + a_2)
a_5 = 8 (a_3 + a_3)
a_6 = 16 (a_5 + a_5)
a_7 = 24 (a_6 + a_5)
a_8 = 48 (a_7 + a_7)
a_9 = 55 (a_8 + a_4)

This chain can easily be converted to the corresponding chain for x^{55}:
x, $xx = x^2$, $x^2x = x^3$, $x^3x = x^4$, $x^4x^3 = x^7$, $x^4x^4 = x^8$, $x^8x^8 = x^{16}$, $x^{16}x^8 = x^{24}$, $x^{24}x^{24} = x^{48}$, $x^{48}x^7 = x^{55}$

It should be noted that in some steps, the choice of which two numbers to add is not trivial. For instance, the step $a_3 = 4$ can be taken by adding a_2 and a_0, as shown in the chain at (1). However, one can also add a_1 and a_1 (remember that it is allowed to add the same number to itself twice). This yields the same result. The resulting addition chain is the same. More on this can be found in chapter 5

The chain given at (1) uses 9 addition steps in total to reach the goal, 55. The length of the addition chain is equal to the total number of steps that are needed. The chain length is represented by *r* ([f(knuth), g(bergeron), and others]). This chain length is equal to the amount of elements in the chain minus 1. This is logical, since no step is needed to reach the first element, 1. Therefore, this step is not counted and this is also the reason that in our earlier example the number 1 is labeled with a_0.

Using the naive method, by multiplying with x over and over again, also yields an addition chain. It is clear however, that this method will never give rise to a short chain: for our example 54 steps would be needed.

There is a restriction for the addition chains, which does not follow directly from the rules presented before: The numbers in an addition chain need to be sorted in ascending order.
If this rule would not be applied, it would be possible to build the following chain (2):

a_0 = 1
a_1 = 2 (a_0 + a_0)
a_2 = 4 (a_1 + a_1)
a_3 = 8 (a_2 + a_2)
a_4 = 16 (a_3 + a_3)
a_5 = 24 (a_4 + a_3)
a_6 = 48 (a_5 + a_5)
a_7 = 3 (a_1 + a_0)
a_8 = 7 (a_7 + a_1)
a_9 = 55 (a_8 + a_6)

This chain would be valid when considering just the basic rules given at first. However, it can be converted to the chain given in (1), without changing the essence of the chain [f(knuth)]. The only difference is the order of the elements. The extra rule poses no extra restrictions on the chain.

While there are no downsides to this added rule, sorting the chain does give some benefits. The chains can be compared more easily, which makes it easier to see whether two chains are equal. Also, the total amount of possible chains is brought down.

Yet another rule is straightforward and logical: the addition should stop at the goal, *n.* Of course, it would be possible to continue calculations after this number. However, logically, this will never lead to a shorter chain for the goal *n*. Therefore, any number in the chain that is added after *n* can be freely omitted.
Another rule, which is not a direct and clear consequence of the previously stated rules, is that elements are allowed to be present in the chain only once.

So far, all seems clear. Now, one might ask, what is the big fuzz about these chains? Why is there so much literature about this relatively simple mechanism of creating chains of numbers?
When we now remember the use of the addition chains once more (to diminish the total number of multiplications when calculating powers), it becomes clear that the addition chains should preferably be as short as possible. The shorter the chain, the less multiplications are needed. The term *minimal chain* is important here. A minimal chain for a given *n* is a chain that is as short as possible. The length of a minimal chain for *n* is represented by *l(n)*. No chain exists that reaches *n* in less steps then the minimal chain.
And this is where the real problems begin. The goal is to find chains that are as short as possible. However, no clear algorithms exist (or at least, are known) which find the shortest chain within a reasonable timing bound. Also, when a short chain is found using fast algorithms, it is not always clear whether this chain is minimal.
The chain that was given at (1) for instance, for $n = 55$, is not minimal. Although this might not be apparent of first sight, it becomes clear when looking at the following chain $^{(3)}$:

$a_0 = 1$
$a_1 = 2\ (a_0 + a_0)$
$a_2 = 4\ (a_1 + a_1)$
$a_3 = 8\ (a_2 + a_2)$
$a_4 = 9\ (a_3 + a_0)$
$a_5 = 18\ (a_4 + a_4)$
$a_6 = 36\ (a_5 + a_5)$
$a_7 = 54\ (a_6 + a_5)$
$a_8 = 55\ (a_7 + a_0)$

This chain has a length of 8, one less then the chain given at (1). So, by using the chain of (3), one can calculate x^{55} in just 8 multiplication steps. Also, this is obviously a lot less then the 54 steps that would be needed when using the naive method, which multiplies *x* by itself repeatedly.
Again, it might not be clear on first sight, but the chain given in (3) is actually minimal: no other chain will be able to reach our goal of 55 in fewer steps. This however does not mean that the minimal chain we found here is unique. A minimal chain is not per definition the only chain that reaches the goal in that specific amount of steps; in most cases, other minimal chains can be found.

Short addition chains give a quick and economic way of computing powers. The length of the addition chain equals the number of multiplications that are needed.
Because of the way the addition chains enable efficient calculations, they are applicable in a range of fields. For instance, the chains can be used in a number of encryption techniques [l(bleichenbacher), amongst others]. Especially with the RSA-coding algorithm, which is based on computations of powers, addition chains can play a role of importance [d(bos)].
One of the properties of the RSA algorithm is that the keys, which are used for the encryption, are not changed often. Therefore, the addition chains that can be used for the computations using this key do not need to be recomputed often. This means that it becomes worthwhile to

do calculations on the addition chain beforehand in order to use them multiple times afterwards [h(li)].
There are yet other areas in which addition chains can be used. One of them is the field of elliptic curves [j(morain)]. Another application is to test whether a certain number is prime [q(kunihiro). More on this can be found in chapter 7.

So far, the system of addition chains does not sound too complicated. The basic principles of the chains are quite simple. For small numbers, finding short chains can be done relatively easy by hand. Also, checking whether a given chain is minimal can be done by hand if *n* is not too large. However, for larger *n*, it becomes infeasible to do so. In general, finding a minimal addition chain is not that easy. No simple and univocal algorithm is known that can compute a minimal chain for any given *n* within a reasonable timeframe. Also, no algorithm is known which does terminate within reasonable time that performs better than any other known algorithm for *each n.* Because of this fact, in the course of history many algorithms have been invented for calculating reasonably short chains.

However, it is assumed that the problem of computing minimal chains is NP-complete [c (sauerbrey), d(bos), e(yacobi), h(li), m(flahmenkamp), q(kunihiro)]. A problem is in NP, if no known algorithm can solve it within polynomial complexity and if there is a polynomial algorithm which checks to see if a possible solution meets the requirements. For addition chains, this second algorithm could check whether for goal *n* a chain c with length l(c) exists. Now, a problem is NP complete, if it is in NP and any other problem in NP can be reduced, or translated, to it by using a polynomial algorithm.
Now, other sources ([l(bleichenbacher), m(flahmenkamp)]) seem to disagree: they don't mark the addition-chain problem to be NP-complete.
A closely related problem, finding the shortest chain for a set of numbers (the so-called addition sequence, see chapter 6), is known to be NP-complete [[g(bergeron), l(bleichenbacher), p(bleichenbacher)]. According to Achim Flammenkamp on his website:

> *"Computing* l(n) *for given* n *is NP-hard. Downey, Leony and Sethi didn't prove anything about this statement in their 1981 SIAM article. They proved a similar one, if a set of numbers is given! {addition sequences, SvdK} Many people (and even experts) overlooked this important difference!"*
> [w (referring to the article of P. Downey, B. Leong and R. Sethi; "computing sequences with addition chains". SIAM journal on Computing, 10 (3), p 638-646, 1981]

I have not been able to find any evidence or proof in the literature that computing a minimal addition chain is NP-complete, except for the fact that many sources cite this as being a fact, without giving an actual proof. This corresponds to a notion in [l(bleichenbacher)], were it is mentioned that *"we are not aware of a proof that the […] problem of computing l(n) given n is NP-complete"*. It might however well be possible that the problem of addition chains is actually NP-complete, and that the many authors who write so are right.

In chapter 8, the most important algorithms that give an approach of the minimal addition chain (will be called 'approach algorithms' from now on) are treated. Possibly, still some room for improvement can be found in the currently known algorithms, and an algorithm might be found that calculates shortest chains within reasonable timing and memory constraints.

4. Terms and Conditions

Before moving on with describing different representations of addition chains and some of the most important algorithms, first some of the most often used terms concerning the subject are introduced. Some of the terms have been mentioned before, but will be repeated for the sake of completeness.

As mentioned before: an *addition chain* is a chain of numbers, integers, for a given number *n*, for which it holds that

$1 = a_0, a_1, a_2, \dots , a_r = n$

with the additional restriction that

$a_i = a_j + a_k$, where $k \leq j < i$ for all $i = 1, 2, ..., r$

Let's first define the function that specifies the length of a chain: *l(c)*. Here, *c* is a chain. The length of a chain equals the number of elements in the chain minus one. This corresponds to the number of calculation steps that are needed, which is the same as the number of multiplications used to go from *a* to a^n. [b(wattel)].

A *minimal addition chain* is a chain for a given number *n* that is as short as possible. This means, that no other chain can be found, which complies with the rules for addition chains and which needs less steps to reach the goal, *n.*

- The length of such a minimal chain for the number *n* is given by the integer *r* [f(knuth)].
- The function that describes the length of the minimal chain for *n* is *l(n) [b (wattel), c(sauerbrey), f(knuth) and many others]*. Note that there is an overlap here: when x is a chain, the function l(x) returns the length of the chain. When x is an integer however, l(x) returns the length of the shortest possible chain for this goal *x*.

  The problem of determining l(n) appears to be addressed for the first time by Dellac (H. Dellac, "l'Intermédiaire des Mathématiciens", 1984) in 1894. In that same year, De Jonquières gives a partial solution (E. De Jonquières, "l'Intermédiaire des Mathématiciens", p 162-164), which mentions the factor method (see chapter 8, a method that is still often cited up until today). Jonquières also gave a list with what he thought to be l(n) for all prime numbers beneath *n = 200*. Later on, it appeared he was off by one in four cases: 107, 149, 163 and 179. [f(knuth)]

Two functions prove to be particularly helpful when it comes to addition chains: $\lambda(n)$ and *v(n).* These two functions are extensively used throughout the literature [f(knuth), g(bergeron) n(thurber) to name just a few].

- $\lambda(n) = \lfloor \log_2 n \rfloor$. This number is equal to:

  (the length of the binary representation of n) - 1

- *v(n)* = the number of ones in the binary representation of *n*. This is also known under the name 'Hamming weight', a term plentifully used in the world of cryptography.

These two functions can also be given recurrently [f(knuth)]

- $\lambda(1) = 0$, $\lambda(2n)= \lambda(2n+1)= \lambda(n)+1$
- $v(1) = 1$, $v(2n)=v(n)$, $v(2n+1) = v(n) +1$

One of the fields in which these two functions can be applied is in calculating the upper and lower bound for minimal chains. These bounds are the margins for the length of the minimal addition chains. The bounds can be used in a variety of ways, for example to test or improve algorithms, which try to generate short chains. In chapter 9 more information on these bounds can be found.

Addition chains can be made up of a set of different kinds of steps. These steps and the corresponding names are also extensively described in various sources ([f(knuth), m(flammenkamp), amongst others]):

- First of all we have the *doubling step*. Here, the last number in the chain is added to itself, which effectively doubles it:

  $a_i = a_{i-1} + a_{i-1}$.
- In a *star step,* the last number of the chain is used as one of the two additives. Therefore:

  $a_i = a_j + a_k$ en $a_j = a_{i-1}$, $a_k <= a_{i-1}$.

  A doubling step is logically always a star step. This does not hold the other way round though.
- Furthermore, there is the '*plus-one-step*', in which one is added to a number in the chain. This number will logically always be the last element of the chain. Otherwise, the chain would either not be sorted strictly ascending anymore or elements would be present in the chain more than once. This step can be represented as

  $a_i = a_{i-1} + a_0.$
- In a *small step*, to conclude, two numbers are added in such a way that $\lambda(n)$ is not increased. Informally stated, the next power of 2 is not 'reached'. Perhaps, a better description would be that, when taking a small step, the length of the binary representation of the added element in the chain is equal to that of the element left of it [m(flammenkamp)]. A small step can be represented as:

  $\lambda(a_i) = \lambda(a_i - 1).$

There are also some separate classes of addition chains. For instance, there is the class of star chains. As the name might suggest, such a chain consists solely of star steps; in each step, the previous element of the chain is used as one of the additives. Such a chain can be represented in a format which is starting to look familiar:

A chain of integers, for which it holds that:

$1 = a_0, a_1, a_2, \dots , a_r = n$

with the additional restriction that

$a_i = a_{i-1} + a_k$, *where* $k < i$

Notice the difference with the definition of a regular addition chain: instead of adding $\boldsymbol{a_j} + a_k$, where $k \leq \boldsymbol{j} < i$ for all $i = 1, 2, ..., r,$ we now add $\boldsymbol{a_{i-1}} + a_k$, *where* $k < i.$
For every *n,* a star chain can be found which reaches its goal. For a long time, up until halfway through the twentieth century, it was thought that for every *n* a minimal chain could be found which consists only of star steps. [m(flammenkamp)]. W. Hansen was the first who proved this to be wrong (W. Hansen, "Zum Scholz-Brauer Problem", Journal für reine und angewandte Mathematik, V. 202 1959, p. 129-136).
The first *n* for which there exists no minimal chain built up solely from star steps is 12509 [f(knuth)]. The minimal chain for 12509 has a length of 17 (please note that we are using a different, shorter representation than the one used in chapter 3. For more information on the representation of chains, see chapter 6):

1 2 4 8 16 17 32 64 128 256 512 1024 1041 2082 4164 8328 8345 12509

Note that the step from 17 to 32 is not a star step. No star chain can reach this *n* in so few steps.

A minimal star chain length is the smallest length of a star chain for goal *n.* This number is represented by l*(n) [c(sauerbrey), f(knuth)]. Logically, no star chain exists for which the minimal length l*(n) is smaller then the 'general' minimal length, l(n).
Therefore, l(n) ≤ l*(n) [c(sauerbrey), f(knuth), m(flammenkamp) and others].
While one cannot always find a minimal chain by considering only star chains, there still are some benefits in using them. One such benefit is that star chains can often be created using less memory then non-star chains [c(sauerbrey)]. Chapter 8 pays more attention to certain algorithms for calculating chains that are reasonably short. Some of these algorithms create only star chains.

Yet another function, *N(c)* gives the number of small steps that are used in a chain [n,o(thurber)]. The value given by *N(c)* depends on the specific chain that is used. However, just as functions for the length of a chain, *N(n)* can also be used with an integer *n* instead of a chain *c*. *N(n)* returns the minimal number of small steps for the number *n* (which is equal to the number of small steps in the minimal chain).
The number of steps that is *not* small (so-called big steps, [n(thurber)]) is fixed for each chain for *n:* this is equal to *λ(n)* [many sources, including f(knuth), m(flammenkamp)]. Since the total chain length is equal to the number of small steps and non-small steps combined, this means that *l(c) = N(c) + λ(n).*
This implies that a minimum chain can be found by minimizing the number of small steps taken. Every chain without any small step consists of only doubling steps and will form a chain for n = 2^m, where m is a natural number.

Let's recapitulate some of the terms we encountered so far in this chapter, applied to the example we used before of *n = 55*:

- λ(55) = 5. Since ^{2}log (55) = 5.78, $\lfloor {}^2\log n \rfloor$= 5.
- v(55) = 5. The binary representation of 55 is 110111, in which 5 bits are set to 1.
- l(55) = 8, see chapter 3
- N(55) = 3 for any of the minimal chains. For the chain given in (1) in chapter 3, *N(c) = 4*.

In the next part of this chapter, some general properties of addition chains are described. There are a couple of, sometimes trivial, properties that hold for each (minimal) addition chain. One of these is that the first step of each chain is a doubling step [f(knuth)]. Since in this step only one number is available, namely 1, this number is added to itself, thus giving the second number, 2. Thus, every chain is of the same format: 1, 2, …, n. By expanding this reasoning, one can see that the third element in the chain will either be 3 or 4. It follows from this that *l(1) = 0, l(2) = 1, l(3) = 2, l(4) = 2*. [b (wattel)]. This reasoning can of course be expanded even further, but since the number of integers that need to be added in each step grows exponentially, this approach is not very workable. See chapter 8 and 10 for some algorithms which use similar tactics.

As was mentioned earlier, the length of a chain is equal to *N(c) + λ(n) [f(knuth)]*. This can easily be seen: *λ(1) = 0* and every step is by definition either a small step or a step that increases *λ(n)* by 1. Therefore, it must hold that each chain is as long as *N(c) + λ(n)*.
There is yet another way to show the validity of this rule: for every step i in an addition chain it holds that $a_{i-1} < a_i \leq 2a_{i-1}$. Doubling is the largest possible step, and a doubling step always increases $\lambda(a_i)$ by one. Therefore, $\lambda(a_i)$ is always equal to either $\lambda(a_{i-1})$ or $\lambda(a_{i-1})$+1. If $\lambda(a_i) = \lambda(a_{i-1})$, then step a_i is a small step. If $\lambda(a_i)$ is equal to $\lambda(a_{i-1})$+1 however, it is a big step. Since every step now is either small or big, the length r of a chain will always equal to *λ(n) + N(c)* [f(knuth)].

As was also mentioned before, a doubling step is always a star step. This is a direct result of the definition of the doubling step: in which "the last number in the chain is added to itself". If one chooses to add any other number than the last to itself, this is of course also allowed (provided that the resulting chain does not violate the demand that chains should be strictly increasing. See for example chain (1) in chapter 2, where the third element is added to itself in order to create the fifth. However, this is not called a doubling step, due to the definition given before. A doubling step is never a small step [f(knuth)].

Another rule for addition chains is, that a doubling step is always followed by a star step. If this would not be the case, then we would get the following chain:

$1, ..., a_i, a_j = 2*a_i, a_k = a_l + a_m, ..., n$

in which $l \leq m < j$.

Now, it will always be the case that $a_l + a_m < a_j$, and therefore $a_k < a_j$ will hold. This however violates the definition of addition chains, which requires a strictly ascending order of its elements. Thus, a doubling step will always have to be followed by a star step. [f(knuth)]

Another 'universal addition chain rule' is that when step a_i is *not* a small step, then either a_{i+1} must be a small step, a star step, or both. Even though this might sound surprising at first, this rule also proves to be logical. Whenever step a_i is not small, $\lambda(n)$ is increased by 1 in this step. In order to understand the reasoning, it might be easier to look at the following chain:

$1, ..., a_i, a_j, ..., n$

Now, if step a_j is not a star step (or, put differently, the number that is reached in step *i* is not used in the calculation of a_j), then this step must be a small step. This is a direct result from the fact that a_i was not a small step and a_j was not a star step. It is impossible to increase $\lambda(n)$ by 1in step *j*, with *i* being a big step, without using the number a_i. This means, that *j* is a small step (and does not increases $\lambda(n)$) or a star step. Note, that the 'or' in this condition is not exclusive: j can be a star step as well as a small step.

On first sight, this fiddling with special cases of addition chains does not seem to be very helpful. However, the observations presented in this chapter might be used later-on, when looking at the algorithms which produce an approach of optimal addition chains. The rules as presented just now might enable us to prune part of the search tree.

As was stated previously, there are often multiple and different minimal addition chains for one and the same *n*. In literature, the function *NMC(n)* (Number of Minimal Chains) is used to describe the number of minimal chains for *n* [n,o(thurber)]. The number of minimal chains fluctuates strongly for different values of *n*. First of all, the number of minimal chains for every power of 2 (NMC(2^r) is equal to 1. For this set of chains, the only way to produce a minimal chain is by applying the doubling step repeatedly. It appears to be so that *NMC(n)* = 1 for the powers of 2 and for *n = 3* only. While this statement has of yet not been proven [n(thurber)], so far no counterexample has been found. A possible explanation for this is that in many chains two calculation steps can be swapped in such a way that this does not influence the total number of steps. Let's take a look at the following sequence, as an example:

$1, …, a_i, a_j = 2 * a_i, a_k = a_j+2, … n$

This results in a chain of the same length as

$1, …, a_i, a_j = a_i + 1, a_k = 2 * a_{j.}, …$ n

provided that the number a_j in the first chain is not used in any of the steps following the calculation of *k*. In the two chains above, the doubling step and the addition step have been swapped.

With the same restrictions in mind, we can also exchange such a chain as

$1, … a_i, a_j = a_i + a_g, a_k = a_j + a_h, … n$

where $h, g < j$ and $h \neq g$ with the following chain:

$1, … a_i, a_j = a_i + a_h, a_k = a_j + a_g, … n$

without affecting the final chain length.

As *v(n),* the number of ones in the binary representation of *n*, increases, it might be expected that these conditions occur more frequently.

Knowing the total number of minimal addition chains for *n* can be useful for several reasons. In some cases, algorithms are used which try to find all of the minimal addition chains. When *NMC(n)* is known, it is easy to see whether this algorithm has indeed found all minimal chains. Using this knowledge can diminish the search time for additional minimal chains. In this way one can also state, that if none of the minimal chains is a star chain, that no such minimal star chain exists.
NMC can also be used in one of the calculation-algorithms: the factor method. The number of minimal chains can be used here to see whether the factor method produces a minimal chain. For more information on the factor method, see chapter 8.3. [n(thurber)]

There are plenty of other, less used, terms that apply to addition chains, some of which will be treated here.

- *L(r):* The set of numbers that has a minimal chain of length r. For *r = 1, L(r) = {2}*, for *r = 2 L(r) = {3, 4}*, etc.
- *c(r):* This function gives the smallest number *n* with a minimal chain of length *r.* This is equal to the smallest value of *L(r).* So, the first couple of values are: *c(1) = 2*, *c(2) = 3* [f(knuth), m(flammenkamp)]. While this function sounds easy enough, according to [f(knuth)] *"it seems to be most difficult to compute l(n) for these values of n".* For r ≤ 11, the value for c(r) appears to be approximately equal to c(r-1) + c(r-2). Some people thought that c(r) would grow like Φ^r, but Knuth however proved them wrong. Yet others believed that c(r) would always return a prime number. However, c(15) = 11 * 173 and c(18) = 11 * 1021. This has led Knuth to say that *"perhaps no conjecture about addition chains is safe."* The first 23 values for c(r) can be found in the following table:

r	**c(r)**
1	2
2	3
3	5
4	7
5	11
6	19
7	29
8	47
9	71
10	127
11	191
12	379
13	607
14	1087
15	1903
16	3583
17	6271
18	11231
19	18287
20	34303
21	65131
22	110591
23	196591

Table 4.1: Values for c(r) for 1 ≤ r ≤ 23 [f(knuth), m(flammenkamp)]

- *d(r):* The total amount of numbers that have a minimal chain of length r. Notice the similarity with L(r). Where L(r) returns the set of numbers which have a minimal chain of length r, d(r) returns the corresponding amount of numbers. So, *d(1) = 1*, *d(2) = 2. [f(knuth), m(flammenkamp)]*

r	***d(r)***
1	1
2	2
3	3
4	5
5	9
6	15
7	26
8	44
9	78
10	136
11	246
12	432
13	772
14	1382
15	2481

Table 4.1: Values for d(r) for $1 \leq r \leq 15$ [f(knuth)]

Clearly, d(r) seems to be an increasing function. There is no evident way to prove this however, or to determine growth factor.

The most important function we have encountered so far is by far l(n), which gives the length of a minimal addition chain for n. Many theories about this function have been thought of. Some of the most important will be treated here. Some others, about the bounds of this function, can be found in chapter 9.

- One of the most important conjectures about l(n) is known as the Scholz-Brauer conjecture: $l(2^n-1) \leq n + l(n) - 1$. This conjecture, stated in 1937, is still an open problem. Computer calculations have shown that the conjecture holds for $n \leq 14$, and it is also known to hold for *n = 15, 16, 17, 18, 20, 24 and 32* [f(knuth)]. Also, the conjecture is proven to hold for several special cases. For instance, whenever l(n) is reached by a star chain, the conjecture holds. This was proven by Brauer in 1939: $l^*(2^n-1) \leq n + l^*(n) - 1$. The result of Brauers proof is that, in order to prove the Scholz-Brauer conjecture, one only has to look at those chains for which $l(n) < l^*(n)$, the first of which occurs only at n = 12509. In general though, a proof still has to be found.
  The numbers of the form 2^n-1 are of particular interest. The reason for this is that the binary method, one of the best known and most commonly cited algorithms concerning addition chains (see chapter 8.2), gives the worst performance for these numbers.

Another conjecture states that $l(n) \geq \lambda(n) + \log_2(v(n))$. Although this conjecture has not been proven in general, there are certain special cases for which it does hold.
For instance, if $v(n) \geq 2^m+1$, then $l(n) \geq \lambda(n) + m + 1$, *for m = 0, 1, 2 and 3.*
When $v(n) \leq 16$, then $l(n) \geq \lambda(n) + \log_2 v(n)$ holds. On the other hand, when $v(n) \geq 17$, *then* $l(n) \geq \lambda(n) + 4$ [n(thurber)].
Schönhage came close to proving the general conjecture (A. Schönhage, *A lower bound for the length of addition chains*, Theoret. Comput. Sci., 1): he showed that $l(n) \geq \log_2(n) + \log_2(v(n)) - 2.13$. More information on such kind of bounds for the length l(n) of minimal chains can be found in chapter 9.

There are quite some ideas about addition chains which have been though to hold, but proved to be false later on. We already saw some examples of this. Another false believe held quite a while was that the binary algorithm (presented in chapter 8.2) returned minimal chains. Yet another false idea was, as we saw, that for every *n* one could find a minimal addition chain consisting of solely star steps (written as a formula l(n) = l*(n) does not hold for all n). A third example, which was also mentioned in this chapter, that c(r) would consist only of prime numbers, proved to be false as well.
There is another example of such an idea that at first was thought to be valid, but appeared to be wrong after all: the rule that l(2n) = l(n) + 1. This rule might sound logical. If one has a minimal chain for *n*, then the fastest way of reaching *2n* is by taking the chain for *n* and add a doubling step at the end. This conjecture was first mentioned by A. Goulard and a 'proof' for it was given by E. De Jonquières (l'Interm. des Math. 2 (1895), 125-126). However, as can be shown by computer calculations, this conjecture fails as well: since l(191) = l(382) = 11. It is difficult to prove by hand however that l(191) > 10. The smallest four values for which l(2n) = l(n) are: n = 191, 701, 743, 1111.
It might seem reasonable then to think that l(2n) ≥ l(n), but even this might not hold. Kevin R. Hebb has shown [(Notices Amer. Math. Soc. 21 (1974), A-294)] that l(n) – l(mn) can get

arbitrarily large for all fixed integers m not a power of 2. The smallest case for which l(mn)< l(n) is $l((2^{13}+1)/3) = l(2731) = 15$ and $l(2^{13}+1) = 14$[f(knuth)].

And then of course there is the question whether or not the addition chain problem is NP-complete, as was already mentioned in chapter 3. While this has been cited in many sources, others counter it.

Perhaps this further clarifies why Knuth sighed that it seems like no conjecture about addition chains is safe [f(knuth)].

There are, however, certain rules which do apply to the chains.

- $l(n) \leq l(r) + l(s)$ for $n = r*s$. When you have the following two chains:

  $1, a_1, \ldots, a_r = r$

  and

  $1, b_1, \ldots, b_s = s$

  you can always construct the chain $1, a_1, \ldots, a_r, a_r*b_1, \ldots, a_rb_s = r*s$, which has the length l(r) + l(s). In some cases one can optimize chains that are generated in such a way. Therefore, $l(n) \leq l(r) + l(s)$ [b(wattel), f(knuth)].
- $l(n+1) \leq l(n)+1$ and $l(n+2) \leq l(n)+1$. This is simply a result of the fact that each chain starts with the numbers 1 and 2. Therefore, when you have a minimal chain for *n,* a chain for both $n + 1$ and $n + 2$ with a length of $l(n) + 1$ can always be found. Here too, of course, sometimes shorter chains are possible.

There are some other rules, which give upper and lower boundaries for the length of an addition chain l(n). For a more elaborate coverage, see chapter 9.

- $l(n) \leq \lambda(n) + v(n) - 1$ [m(flammenkamp)]. Since $l(n) = \lambda(n) + N(n)$, it follows from this rule that $N(n) \leq v(n) - 1$.

Of course, the last step of a minimal addition chain is always a star step. If not, the number a_{n-1}, which is not used, could easily be removed from the chain. The resulting chain would still be a valid chain for n, with a length one shorter than the original. Therefore, the original chain could never have been a minimal chain. So for each minimal chain, the last step is a star step. If n is odd, the last step is non-doubling, since a doubling step can create even numbers only.

5 Different kinds of chains

In the course of history, a range of different types of addition chains was created. These different kinds of chains are in some ways different from the 'standard' addition chain, but they do mostly follow the same rules and have a comparable format. Some of the most well known alternatives for the regular addition chain will be discussed in this chapter. However, this is mostly for the sake of completeness; in the rest of this paper not be much attention will be paid to these alternatives.

First of all, two alternatives cited in literature most often are addition sequences and vector addition chains. Both of these techniques change the basic start of the chains, but use the standard rules. Whereas regular addition chains are built up using a series of numbers with a single goal, *n*, addition sequences start with a sequence of multiple goals and vector addition chains create chains consisting of vectors instead of numbers.

5.1 Addition sequences

With an addition sequence, a set of integers of the form $\{n_1, n_2, \ldots, n_m\}$ is taken as goal, replacing the single number for which a chain normally has to be found. The resulting addition sequence is supposed to contain all of these numbers. The resulting addition sequence itself looks similar to a regular addition chain.

Put more formally, the addition sequence is:
A chain of numbers, where for a given sequence of numbers $\{n_1, n_2, \ldots, n_m\}$

$1 = a_0, a_1, a_2, \ldots, a_r = n_m$

with the restriction that:

$a_i = a_j + a_k$, where $k \le j < i$

and each number of the given sequence occurs in the chain:

$n_g = a_h$, where $g \le m$ and $h \le r$.

Other restrictions for addition chains, which were mentioned in chapter 3, still hold. So, for instance, the addition sequences have to be strictly increasing as well.

Different techniques might be used to calculate the addition sequences efficiently. Once again, the ultimate goal is to find addition sequences, which are as short as possible. One possible approach is to calculate a chain for each of the individual goals first, and then combine these different chains into one. However, this does not guarantee at all that a minimal addition sequence will be found. Better solutions will need to try and combine the calculations for each of the numbers of the goal sequence in an optimal way.

As was mentioned in chapter 3, the problem of addition sequences has been proven to be NP-complete. Although it is closely related to the problem of calculating minimal addition chains, I have not been able to find a proof that states that the problem of calculating addition chains is NP-complete itself too, although this is often claimed to be the case.
Addition sequences are treated a bit more extensively when covering the window method to calculate addition chains (see chapter 8.4). Part of this method uses addition sequences to calculate the shortest possible chain.

5.2 Vector addition chains

Vector addition chains have a single goal. However, each of the elements of a vector addition chain is a vector instead of a single number. A vector has the form $(x_1, x_2, \ldots, x_m)$.
A regular addition chain starts with the number one and builds up to goal *n*. A vector addition chain starts with a goal with the following shape: $(x_1, x_2, \ldots, x_m)$. Instead of one initial element,

there will be *m* vectors to start with, of the form (1, 0, … 0), (0, 1, …, 0), …, (0, 0, …, 1). Each of these vectors consists of m elements: *m-1* zeroes and a single one.

In [r(bergeron)], the following definition for a vector addition chain c with the two elements (a, b) is given:

c = $(a_{-1,}\ b_{-1}), (a_{0,}\ b_0)\ (a_{1,}\ b_1), \ldots, (a_{r,}\ b_r)$, where

$(a_{-1}, b_{-1}) = (0, 1)$,

$(a_0, b_0) = (1, 0)$ and

$(a_r, b_r)= (a, b)$,

where for each i, $1 \leq i \leq r$, there exists j and k, $-1 \leq j, k < i$, such that $(a_i, b_i) = (a_j + a_k, b_j+ b_k)$

This definition can be extended to that for a vector addition chain with m elements: (a, b, c, … m).

c = $(a_{-m+1}, b_{-m+1}, c_{-m+1}, \ldots, m_{-m+1}), (a_{-m+2}, b_{-m+2}, c_{-m+2}, \ldots, m_{-m+2}), \ldots, (a_{-1}, b_{-1}, c_{-1}, \ldots, m_{-1}), (a_0, b_0, c_0, \ldots, m_0), (a_1, b_1, c_1, \ldots, m_1), \ldots, (a_r, b_r, c_r, \ldots, m_r)$, where

$(a_{-m+1}, b_{-m+1}, c_{-m+1}, \ldots, m_{-m+1}) = (0, 0, ..., 0, 1)$,

$(a_{-m+2}, b_{-m+2}, c_{-m+2}, \ldots, m_{-m+2}) = (0, 0, ..., 1, 0)$,

$(a_{-1}, b_{-1}, c_{-1}, \ldots, m_{-1}) = (0, 1, \ldots, 0, 0)$,

$(a_0, b_0, c_0, \ldots, m_0) = (1, 0, \ldots, 0, 0)$,

$(a_r, b_r, c_r, \ldots, m_r) = (a, b, \ldots, m)$

where, for each i, $1 \leq i \leq r$, there exists j and k, $-1 \leq j, k < i$, such that $(a_i, b_i, \ldots, m_i) = (a_j + a_k, b_j+ b_k, \ldots, m_j + m_k)$

Perhaps the principle of vector addition chains is best explained using an example. Let us take a simple example, with (40, 7) as goal. Just as with addition sequences, there is an easy solution. One could first calculate (40, 0) and then (0, 7), using the same techniques that can be applied to standard addition chains. These two vector addition chains can afterwards be combined to a chain reaching the goal (40, 7). However, as one might expect, this will generally not produce a minimal vector addition chain. Just as with the addition sequences, the simplicity of this algorithm comes at a price.

Techniques that can be used to calculate regular addition chains efficiently can certainly be beneficial to the calculation of the two alternative problems mentioned just before. However, both do need specialized algorithms of their own in order to provide optimal chains at the lowest costs. For both these problems, the key to success is to combine the different calculation steps for parts of the chain. Again, this somewhat cryptic description can be clarified using an example. Let's have another look at our example vector addition chain, mentioned above: (40, 7). A chain can be given by first calculating one for (40, 0) and one for (0, 7) and combining these afterwards. This could result in the following chain (remember from the definition: this chain has to start with the elements (0, 1) and (1, 0):

(0, 1); (1, 0); (2, 0); (4, 0); (8, 0); (16, 0); (32, 0); (40, 0); (0, 2); (0, 3); (0, 4); (0, 7); (40, 7)

This chain has a length of 11 (in an vector addition chain, none of the initial elements is taken into account when calculating its length), just as with regular addition chains.
A better result can be obtained when we try to combine some of the steps in this chain:

(0, 1); (1, 0); (0, 2); (2, 0); (4, 0); (8, 0); (8, 1); (16, 2); (32, 4); (40, 5); (40, 7)

resulting in a chain of length 9.

With addition sequences, similar improvements can be made: instead of calculating an addition chain for each element of the sequence separately and combining these afterwards, one could try to combine certain steps in order to create a shorter chain.

It should be noted that many of the terms for addition chains, mentioned in chapter 4, can also be applied to addition sequences and vector addition chains. For the minimal length of an addition sequence for example, we could use $l(a_1, a_2, \ldots, a_m)$ [d(bos)].

5.3 Addition-subtraction chains

Apart from changes to the problem definition itself, other changes also have been suggested. For instance, one could think about certain adaptations to the rules of creating the chains. One important class is formed by the addition-subtraction chains. [many sources, including i(gordon), k(bosma)]. There are certain numbers *n* for which the minimal (regular) addition chain is relatively long. For example, the set of numbers $n = 2^r - 1$ has a particularly poor performance, when compared to other numbers of approximately equal size (see also appendix II with the minimal length of addition chains)
We can now change the rules for creating addition chains, by allowing not only additions but also subtractions. This will shorten the minimal chains for some *n*, especially for those where $n = 2^r - 1$. Logically, when working with these so-called addition-subtraction chains, the demand that chains should have a strictly increasing ordering has to be dropped [k(bosma)].

Let's look at an example again. As mentioned, addition chains perform particularly poor for the set of numbers $2^r - 1$. Therefore, it might be worthwhile to take a look at one of these numbers: *n = 31*, for instance. Using regular addition chains, a minimal chain with a length of 7 can be found, for instance:

1 2 4 8 9 18 27 31

However, by allowing subtraction as well, a chain with a length of 6 can be found:

1 2 4 8 16 32 31

While this looks promising, allowing subtraction does not decrease *l(n)* for every *n*. If we look at the example of *n = 55*, given in chapter 2. A possible addition-subtraction chain for this *n* would be

1 2 4 8 9 16 32 64 55

This chain has a length of 9. Note that this chain is not minimal (remember that we showed in chapter 2 that chains exist for *n = 55* with a length of 8). Allowing subtraction will not necessarily lead to a decrease in minimal chain length for every number. It should be noted however, that the minimal addition-subtraction chain for a number *n* will never be longer than the regular minimal addition chain. This is easy to see: one is not obliged to use a subtraction step in a addition-subtraction chain. Therefore, you could always create a chain that is exactly equal to the minimal addition chain, by using only addition steps.
Why then would we bother using those regular addition chains at all? Since using addition-subtraction chains shorten our minimal chain length, let's stick to those, right? Things however are not that simple. First of all, let's remember the original use of addition chains: to decrease the complexity of calculating powers. We do this by decreasing the number of multiplication steps that are needed: each step in the addition chain corresponds to a multiplication step when computing powers.
Now, if an addition step corresponds to multiplication, then a subtraction step corresponds to division. Generally, division is quite a costly operation, which needs more computing time than most other operations, including multiplication [j(morain)]. So, by allowing subtraction, we might need fewer steps, at the cost of these steps being more complex. Not an easy tradeoff.
Yet another downside of allowing subtraction is that the number of possible chains is increased greatly. Our search field is broadened intensively: the total number of possible

chains leading to *n* using addition-subtraction chains is larger than the number of chains using only addition steps. Therefore we will, in general, need more computations before we can be absolutely positive we have found a minimal chain.
For these two reasons, in general the practical use for addition-subtraction chains is smaller than that for normal addition chains.

5.4 Q-addition chains

There are also less-commonly quoted alternatives, like the so-called q-addition chains [s(von zur gathen)]. Put very simply, a q-addition chain is a normal addition chain, with the difference that multiplication by the number q is free.
For example, let's look at our previous example of *n = 55* again. When we use *q = 5*, we can create the following chain:

1, $*^1$ 5, 10, 11, $*^2$ 55 $^{(4)}$

In the q-addition chain in (4), we have marked all the 'multiplication steps' with a *. Such a q-addition chain can easily be converted to a regular addition chain. To do so, we first need to come up with a chain for *n = q*. So, in our example, we need a chain for *n = 5:*

1 2 4 5 $^{(5)}$

Now, in order to create a regular addition chain, we need to 'insert' the chain for *q* (5) into each of those places in the q-addition chain (5), where we used a multiplicative step (all the steps marked with a *). For our example, this would result in the following chain.

1 2 4 5 10 11 22 44 55 $^{(6)}$

Let's have a closer look how we go from (4) to (6). The chain for q (5):

- adds the first element to itself (doubling step), in order to create the second element.
- adds the second element to itself (doubling step), in order to create the third element.
- adds the first element to the third element, in order to create the fourth element.

Now, we need to copy these steps at every marked spot in (5), where we used a multiplication step. For $*^1$, our steps are clear:

- we double the first element (1), to get 2
- we double the second element (2), to get 4
- we add the first element to the third element (1 + 4), to get 5.

For $*^2$, we use the exact same algorithm:

- we double the first element (11), to get 22
- we double the second element (22), to get 44
- we add the first element to the third element (11 + 44), to get 55.

It is obvious that by using q-addition chains we can greatly decrease the length of the minimal chains. However, when looking at our original problem definition, it is clear that its direct use, at least for our purposes, is somewhat limited.
More on these q-addition chains can be found in chapter 6.

5.5 Lucas Chains

A special class of addition chains is formed by the lucas chains. While both kinds of chains are somewhat similar, there are also some important differences. A lucas chain is also a chain of integers, starting with zero and ending at goal *n*. Each of the elements in a lucas chain is created by adding two other elements in the chain. However, lucas chains have the extra demand that the difference between these two additives is also in the chain.

So, a lucas chain for a positive integer *n* is an ascending list of integers

a_{-1}, a_0, …, a_r,

such that

$a_{-1} = 0$, $a_0 = 1$ and $a_r = n$

and such that, for all $1 \leq k \leq r$, there exists $0 \leq h, i, j < k$, with $a_k = a_i + a_j$ and $a_h = |a_i - a_j|$. [L(bleichenbacher)]

Here, element a_{-1} is added, to enable doubling steps. Otherwise, when adding a number to itself to create a new element in the chain ($a_k = a_i + a_i$), the demand that $a_h = |a_i - a_i| = 0$ would not hold. An alternative definition would be:
a lucas chain for a positive integer *n* is an ascending list of integers

$a_0, a_1 \ldots, a_r$,

such that

$a_0 = 1$ and $a_r = n$

and such that, for all $1 \leq k \leq r$, there exists $0 \leq h, i, j < k$, with $a_k = a_i + a_j$ and ($a_i = a_j$ OR $a_h = |a_i - a_j|$). [u]
which would result in a chain starting at 1 instead of zero.
The length of such a chain is given by r. The function $l_L(n)$ gives the minimal length of a lucas chain for *n.* A lucas chain for a set S of nonnegative numbers has minimal length $l_L(S)$.

There are some famous examples of lucas chains, for example (in a slightly altered notation), the fibonacci chain

0, 1, 2, 3, 5, 8, 13, 21, 34, …

and the chain for the powers of 2:

0, 1, 2, 4, 8, 16, …

For our running example of *n = 55* a lucas chain could be:

0, 1, 2, 3, 4, 7, 8, 9, 16, 23, 32, 41, 48, 55

with a length of 12 (both 0 and 1 are not taken into account when calculating the length of a lucas chain).
However, this chain is far from minimal. Shorter chains can be given, such as this one with a length of 8 (which is, as a matter of fact, just the start of the fibonacci chain!):

0, 1, 2, 3, 5, 8, 13, 21, 34, 55

Just as with regular addition chains, many algorithms to calculate minimal lucas chains exist. However, this falls outside the scope of this paper. For more information on lucas chains, see [L(bleichenbacher)].

5.6 L^0 chains

Hansen defined the concept of a l^0 chain. In an l^0 chain certain elements are underlined. Each element fulfills the condition that $a_i = a_j + a_k$, where a_j is the largest underlined element less than a_i. [f(knuth)].

An example of such a l^0 chain is:

$\underline{1}, \underline{2}, \underline{4}, 5, \underline{8}, 10, 12, \underline{18}$

For these chains, the difference between each element l_x and the largest underlined element smaller than l_x is also in the chain.
A nice property of these l^0 chains is that the minimal length of this set of chains lies 'between' the regular l-chains and the l*-chains:

$l(n) \leq l^0(n) \leq l^*n$

Remember now the scholz-brauer conjecture mentioned in chapter 4. As mentioned, this conjecture holds for l* chains. Hansen however proved this theorem to hold also for l^0-chains: $l^0(2^n-1) \leq n - 1 + l^0(n)$.
Since $l^0(n) \leq l^*(n)$, this result is stronger then the Scholz-Brauer conjecture for *l** chains. It is unknown whether $l(n) = l^0(n)$ for all *n*. If this equation would hold, the Scholz-Brauer conjecture would be proven. [f(knuth), n(Thurber)]

6. Chain-representation

In chapter 3, we started with quite an elaborate representation for the addition chains (see the chains given at (1), (2) and (3) in chapter 3). In the chapters afterwards, we quickly switched to a shorter and more synoptic representation. Let's repeat both representations for the same chain, in order to refresh our memory:

Elaborate representation
$a_0 = 1$
$a_1 = 2$ $(a_0 + a_0)$
$a_2 = 3$ $(a_0 + a_1)$
$a_3 = 4$ $(a_2 + a_0)$
$a_4 = 7$ $(a_3 + a_2)$
$a_5 = 8$ $(a_3 + a_3)$
$a_6 = 16$ $(a_5 + a_5)$
$a_7 = 24$ $(a_6 + a_5)$
$a_8 = 48$ $(a_7 + a_7)$
$a_9 = 55$ $(a_8 + a_4)$

Shortened representation

1, 2, 3, 4, 7, 8, 16, 24, 48, 55

This shorter representation however also gives less information than the more elaborate one. Where the latter one provides information about the additives used in each of the steps, the shorter representation only gives the chain elements themselves. However, the extra information about the additives is normally not needed.
In all literature, the shortened representation (either with or without commas) is used.
In the rest of this paper, we will adhere to this representation as well, unless we specifically need the more elaborate one.

6.1 Directed Acyclic Graphs

Addition chains can also be represented in a totally different way. The chains correspond in a natural way with Directed Acyclic Graphs [f(knuth), amongst others].
Figure 6.1 below shows how the chain (1), given in chapter 3, can be represented visually.

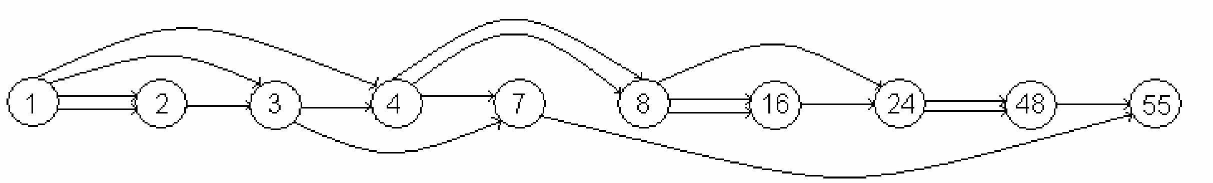


Fig 6.1: DAG for (2)

The syntax of such a chain is quite straightforward. It can be grasped in a couple of simple rules. Each node of the graph corresponds to an element of the addition chain. The nodes are ordered in ascending order, from left to right. Every node, except for the first, has two incoming edges. The first node has none. Also, every node, except for the last one, has minimally one outgoing edge. The last node has none. There are no cycles in the DAG, just as there are no cycles in the chain itself. The incoming edges of a node correspond to the additives used in that step.
Such a DAG-representation exists for every addition chain. Following Bleichenbachers [l(bleichenbacher)] definition, for each addition chain $a_0, a_1, \ldots a_r$ there exists a DAG with $r + 1$ nodes and $2r$ edges. Let it be noted that this also holds the other way around, as

[l(bleichenbacher)] proved; for each DAG there exists an addition chain, which has a length of at most *(the number of edges) – (the number of nodes) + 1*. The length of the chain can easily be inferred from such a DAG. Since each element of the chain is represented by a node in the DAG, the length of the chain will be equal to *(the number of nodes – 1)*, just as it is equal to *(the number of elements in the chain – 1).* The length of the chain is also equal to *(the number of edges / 2).* This can easily be seen, since every addition step is represented by two arrows and the length of the chain is equal to the total number of addition steps.
Not only addition chains themselves can be represented by DAGs, but also many of the related problems (some of which were presented in chapter 5) can be visualized in the same way.

We saw that our simplified representation gets rid of the information about the specific additives that are used. Sometimes, this can be useful; when looking at minimal chains for example, often the chain length is most important, not the specific addition steps. Now, with our DAG-representation, this extra information is back again. When you want to rewrite a chain that is given in the short notation to a DAG however, this cannot always be done without ambiguity. Take, for instance, the minimal chain for *n = 7:* (7)

1 2 3 4 7

This chain can be transformed in any of the following two DAGs [p(bleichenbacher)].

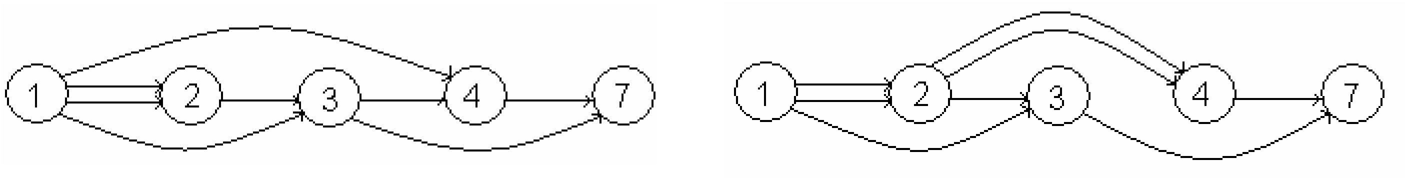


Fig 6.2.1 and 6.2.2: two possible DAGs for the addition chain 1 2 3 4 7

These two DAGs correspond to the following addition chains:

6.2.1:
$a_0 = 1$
$a_1 = 2\ (a_0 + a_0)$
$a_2 = 3\ (a_1 + a_0)$
$a_3 = 4\ (a_2 + a_0)$
$a_4 = 7\ (a_3 + a_2)$

6.2.2:
$a_0 = 1$
$a_1 = 2\ (a_0 + a_0)$
$a_2 = 3\ (a_0 + a_1)$
$a_3 = 4\ (a_1 + a_1)$
$a_4 = 7\ (a_3 + a_2)$

This makes it clear that one can find multiple DAGs for principally one and the same addition chain. This property sometimes is unwanted. For instance, it becomes unclear what the number of minimal chains should be. Should this be equal to all those minimal chains which have a different simple notation, or a different DAG-notation? Also, it becomes more difficult to compare chains when they are essentially the same but their notation is different. Therefore, in general this property is unwanted, even though there are cases in which it might be preferred.

There is yet another representation that corresponds directly to the use of the DAGs. If you look at the DAG in figure 1, you could describe each of its nodes by the two numbers that are added to 'create' that node. The number four in the DAG of figure 6.1 is then represented as (0, 2), because it is built using adding the numbers 1 and 3, which are at positions 0 and 2. The first number of the chain, 1, that is not the result of an addition, is not represented when using this representation. The entire addition chain then becomes

(0, 0), (0, 1), (0, 2), (2, 3), (3, 3), (5, 5), (5, 6), (7, 7), (4, 8). (8)

This bracket-notation however suffers from the same downside as the original DAGs: one needs to specify the exact steps that are used, while this is not needed using the simplest representation.

6.2 Simplifying DAGs

So, DAGs are sensitive to the exact additives that are used, while the standard representation is not. Let's take a look at ways in which we might be able to reduce this and other sensitivities.
DAGs can be rewritten, or perhaps one should say redrawn, to a 'standard format'. The first step towards this standard is by reducing any node without any outgoing edges, except of course for the goal-node *n.* Such nodes can easily be removed from the chain, since they are not used later on in the chain. This means, that each of the other elements of the chain can still be reached when this element is removed. Using such a simplification step shows the benefits of using DAGs, they can be used to identify useless chain elements easily.

The DAGs can be standardized further. When a node (a_l) has only one outgoing edge, this node will be used as an additive for only one other node (a_m). This means, that in order to reach this latter node (a_m), one basically adds three numbers, $a_i + a_j + a_k$, where $a_i + a_j = a_l$ and $a_l + a_k = a_m$. These numbers, a_i, a_j and a_k, can, but do not have to be different.
Now, principally, the order in which we add these numbers is not important. The result of all the additions $(a_i + a_j) + a_k$, $a_i + (a_j + a_k)$ and $(a_i + a_k) + a_j$ is, of course, the same. When the intermediate result of this threefold addition is used only in one other step, to reach a_m, these three different additions are interchangeable in the addition chain. However, when specifying an addition chain, we are forced to distinguish between the three basically identical chains [f(knuth)]. We can however standardize our visual representation in such a way that this 'moment of choice' is eliminated. This ambiguity was also described in chapter 4, page 13, when describing the function for the number of minimal chains, NMC.

Take a look at the chain for *n = 55* in figure 1 for example; here we see that the nodes 2, 7, 16 and 48 all have a single outgoing edge. Now let's take a look at how we might simplify the graph at these nodes. As mentioned, whenever a node has only one outgoing edge, this basically results in a threefold addition. We can visualize this by removing a node with one outgoing edge and diverting both of its incoming edges to the goal-node of the outgoing one. If there are multiple nodes in a DAG which have a single outgoing edge, one should start rewriting at the rightmost node and repeat the rewrite step leftwards until no such nodes are left. When applying this rule to the DAG in figure one, we get the DAG of figure 6.3 as result.

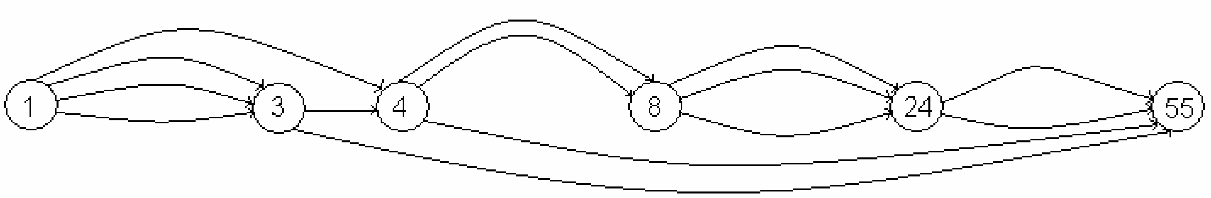


Fig 6.3.: reduced graph for the addition chain of figure 1.

It will be clear that the rules given before, for the regular DAGs cannot be applied directly for such a reduced graph. However, for these graphs the length of the chain can still be retrieved. Now, a slightly less obvious formula should be used:

(number of edges) – (number of nodes) + 1. [f(knuth)]

Taking a closer look, we can easily see the logic behind this formula though. In the original, full-blown figure, we already saw that the total number of steps is equal to

(number of nodes) – 1,

as well as

(number of edges) / 2.

These two formulas logically yield the same result in the original graph. Or, stated differently

(number of nodes) - 1 = (number of edges) / 2 = l(c)

We can combine this to the following formula:

(number of edges) – ((number of nodes) - 1) = (number of edges) – (number of nodes) + 1 = (number of steps in the addition chain).

Now, this latter formula can also be used for the reduced graphs. This behaviour is simply explained: when reducing the graph, we remove a nodes and its single outgoing edge, one at a time. Each of the incoming edges of the removed node is connected to another node, and therefore is still present in the reduced graph. So, in every reduced graph, as many edges as vertices are removed. [p(bleichenbacher)], which means the difference between the number of nodes and the number of edges remains constant throughout the process of reducing.

It should be noted though, that removing a useless step with no outgoing edges in general will alter the outcome of the formula. This is as expected, since this step corresponds to the actual removal of an element of the original addition chain (compared to just altering the representation, as is done while erasing a node with a single outgoing edge from the graph).

Now it's time to take a look at why we went through all this trouble of graph reduction. It enables us to compare DAGs, and thereby addition chains. We can say that two DAGs (and therefore chains) are equivalent if their reduced graphs are the same. Now, different addition chains can be equivalent, since they can have one and the same reduced graph. It might even happen that non-star chains are equivalent to star chains.

Now what was the original purpose for giving the reduced graphs: trying to eliminate some of the unwanted sensitivities of the DAGs. Unfortunately, this problem is not entirely eliminated. When trying to reduce the two graphs for *n = 7*, given previously in figure 6.2, we still end up with two separate reduced graphs (see figure 6.4). So, the 'weakness' of DAGs is persistent, it is still present even after reducing the graphs. Of course, when the definition of addition chains would force us to give the precise calculation of each of the elements of an addition chain, then this weakness would turn out to be a feature instead of a (mild) nuisance. [p(bleichenbacher)].

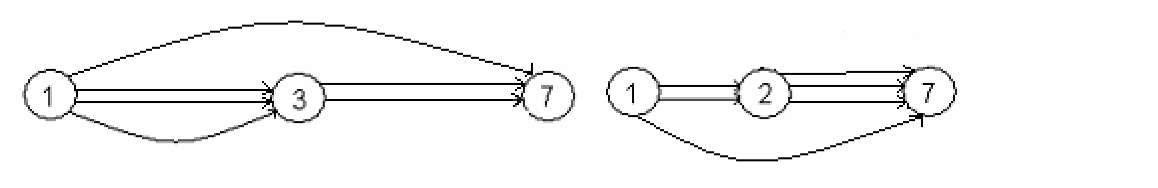


Fig 6.4.1 en 6.4.2: reduced graphs for figures 6.2.1 and 6.2.2

There is yet another use of the graph representation of the addition chains. As is mentioned in [f(Knuth)], the label of each node is equal to the number of paths from the source to that node. This holds both for the regular as well as the reduced graphs. Now, by reversing the directions of all of the arcs and labeling the nodes accordingly, a DAG is created that represents yet another addition chain. An important property is that the label of each vertex is exactly equal to the number of oriented paths from the source to that vertex. Thus, the problem of finding an optimal addition chain is equivalent to minimizing the quantity over all directed graphs that have one source vertex and one sink vertex and exactly n oriented paths from the source to the sink.
Applying this algorithm to the reduced graph in figure 6.3 yields the following result (see figure 6.5).

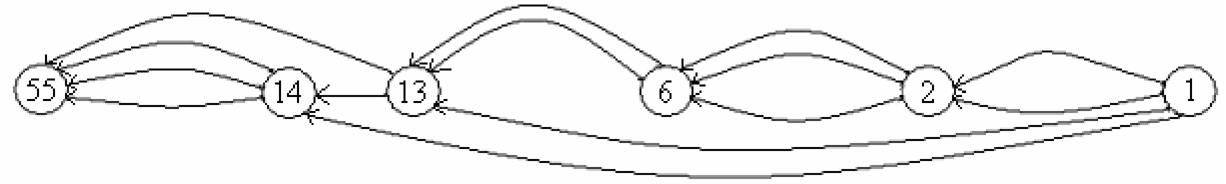


Fig 6.5: reversed DAG for the reduced DAG in figure 6.3

This reversed DAG could correspond to the following addition chain

1 2 4 6 12 13 14 28 42 55

Note that this is a different chain then the one we originally started with (1), though it does have the same length as the original. Also note that this algorithm is only applicable to reduced graphs. When we would try to apply it to the graph of figure 6.1, for example, we run into problems right from the start: nodes which have only one outgoing edge cannot be reversed properly, since this would result in a node with a single incoming edge. Such a DAG could never correspond to a proper addition chain, since some steps in the graph do not result in an addition of two elements.

6.3 Q-addition DAGs

In the previous chapter, q-addition chains were introduced. Now, we can take a closer look at the process of creating regular addition chains out of the q-chains. As was mentioned in chapter 5, we need to 'insert' the chain for *q,* the multiplying factor, into each of those places in the chain where we used a multiplicative step.
Let's reconsider the example for *n = 55* given before, but treat it graphically instead of verbally now.
For *n = 55* and *q = 5*, we created the chain

1, 5, 10, 11, 55

This chain is represented in figure 6.6, where each multiplicative step is marked with a *, just as before.

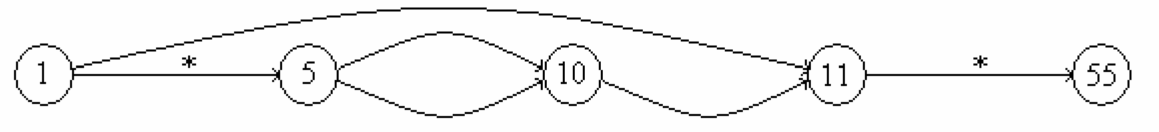


Fig 6.6: q-addition chain for n = 55, q = 5

Now, we have to come up with a chain for q, which is five:

1 2 4 5

Graphically, this can be represented as

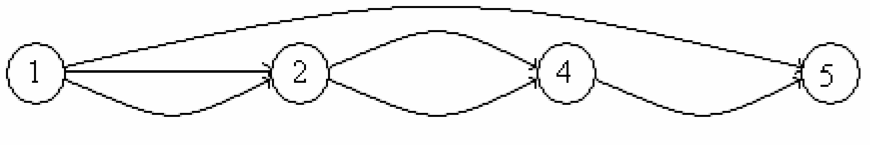


Fig 6.7: Addition chains for n = 5

Now, we need to insert the chain for *q = 5* in the q-addition chain for *55*. Visually, this can be represented as

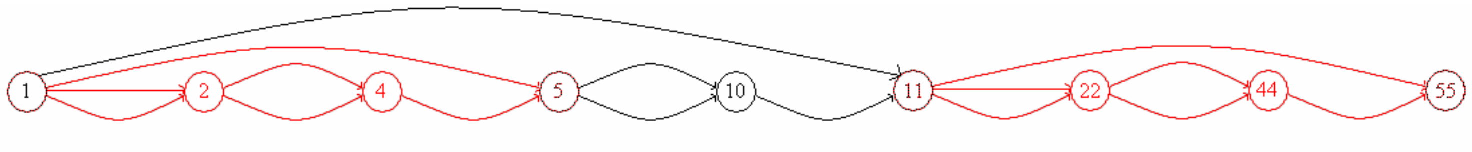


Fig 6.8: Addition chain for n =5 inserted into q-addition chain for n = 55 and q = 5

Here, the inserted parts are marked red. Note that the first and last element of the q-chain overlap with elements from the regular chain. We have created a regular addition chain, starting with the q-addition chain and the chain for q in exactly the same way as before, in chapter 5.

Our DAG representation allows gaining insight in certain operations on addition chains. Let's take a look at squaring the goal of an addition chain. How can such a operation be represented in addition chains themselves; how can we go from the original addition chain to one for which the goal is squared? The DAG representation gives us a fine solution. Whenever we have an addition chain for *n* and the corresponding DAG, we can easily create a chain for n^2 by repeating all the steps of the original chain. Or, equally, by simply copying and pasting all the nodes and vertices of the DAG. Let us look at an example we have used before: *n = 7*. We saw two DAGs for this goal in figure 6.2. In this example, we will focus on the DAG given in figure 6.2.1. Now, in order to create a DAG (and thus a chain) for $n^2 = 49$, all we need to do is copying and pasting this figure, adapt the labels of the nodes where necessary and we are done.

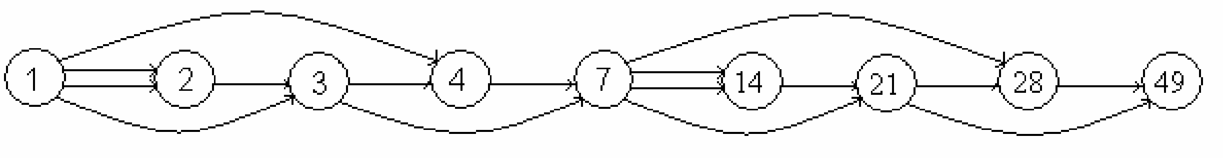


Fig 6.9: The DAG for n = 49, created by using the DAG from 6.2.1

While this method does give an easy procedure to generate addition chains for n^2, it will in general not produce minimal chain, even when the original chain for *n* is minimal. Our example in figure 6.9 illustrates this.
The initial chain for *n = 7* is minimal; no chain exists which reaches *7* in less then *4* steps. The squaring algorithm yields a chain for $n^2 = 49$ with a length of 8 (which is, of course, twice the length of the original chain). However, *l(49) = 7*:

1 2 3 6 12 24 28 49

So, as a conclusion we might add that

$l(n^2) \leq 2 \cdot l(n)$

7 Addition chains applied

In previous chapters, we gave a first introduction on the practical use of addition chains. In this chapter we will elaborate somewhat on this subject. However, the coverage will remain quite limited, since the emphasis of this paper is on the problem statement, not its practical application. Still, this chapter is useful to give a brief glance on practical uses, in order to stress the benefits that minimal chain-calculation can give.

In the introduction already, some of the most important uses of addition chains were mentioned.
Encryption algorithms, RSA in particular, form an important group of such applications. RSA is an algorithm (named after its founders, Ron Rivest, Adi Shamir and Leonard Adleman) which uses public-key encryption. This basically means that messages can be encrypted using a publicly known key. However, to decrypt such decoded messages, one needs another, private key.
In order to encrypt and decrypt the messages using the RSA algorithm, computations using powers are needed. A RSA calculation is a modular exponentiation. In order to encrypt messages, the RSA algorithm encrypts message m by computing $E = m^e \bmod N$, where N is p·q, for the two prime numbers p and q. Decryption goes alike; in order to retrieve message m, one needs to compute $m = E^d \bmod N$. Here, e and N are public keys; d is the private decryption key. In order to ensure the security of the algorithm, p, q and therefore also N should be at least 512 bits large. This also means that d is automatically large.
So, in order to perform the encryption and decryption, we need to carry out quite complex computations using exponentiation with large numbers. This sounds like a proper job for addition chains. And things get even better: in the RSA algorithm, keys do not change often. This means that we can use the same addition chains more often. This makes the use of addition chains worthwhile: computing an addition chain once can result in a speedup in many other computations [c (sauerbrey), h(li)].
The encryption calculations can be speeded up in particular. By choosing special encryption keys of the form $2^{2^n} + 1$ (for instance, 3 or 17), the encryption can be done especially efficient. Note however, that currently only the first four of these numbers, better known as fermat-numbers, are known to be prime. See [x] for more information on fermat primes.
These fermat numbers have the nice property that their minimal chain can easily be found using the binary method (see chapter 8.2). By using these numbers as public encryption keys, the encryption can be done particularly efficient. Obviously, one should not set the decryption key to be one of such fermat numbers, since the encryption technique could then be cracked easily. Decryption therefore is slower than encryption and other algorithms or heuristics are needed [h(li)].

There are yet other fields which can gain in speed thanks to addition chains. One of these is calculations on cyclic groups, such as elliptic curves [d(bos)]. Also, as seen before in chapter 5, calculations on fibonacci and other lucas chains can benefit from the presence of good algorithms to calculate minimal addition chain [d(bos)].

Another use of the addition chains can be found in the field of data compression. According to [e(yacobi)], there exists a large correspondence between calculations with large numbers and data compression. An example of this is that in compression, often used messages are assigned a short code. In fast computations, frequently used computation steps are stored for reuse.
Messages can be compressed by sending the difference between consecutive messages instead of the messages. This is knows as delta modulation. In the computation of powers, we can use $x^{\Delta + n} = x^{\Delta} \cdot x^{n}$, which allows a speedy computation when x^n is known beforehand and when Δ is significantly smaller than n. Yacobi proposed an adaptation to the Lempel-Ziv data compression algorithm, in which he uses this correspondence between compression and

power calculation. For the algorithm he proposed, Yacobi used a variant of a well-known algorithm to compute addition chains efficiently (the m-ary method, see chapter 8.6).

Apart from the fields mentioned so far, addition chains are applicable in any field in which exponentiation is used. One of the problems where exponentiation is heavily used is for instance fermat primality testing, when based on fermats little theorem that $a^{p-1} = 1$ (mod p) for $1 \leq a < p$, which we already briefly encountered when covering RSA in chapter .
The addition chain problem can furthermore be compared to the smallest grammar problem [v(charikar)]. The problem statement for the smallest grammar is *What is the smallest context-free grammar that generates exactly the given string σ?*
In [v(charikar)], the similarity with addition chain problem is used in order to prove the hardness of the smallest grammar problem.

8. Algorithms for short chains

As mentioned before, many methods are known to compute short addition chains. However, all of these methods have their downside. They either just calculate short (instead of minimal) chains, or they cannot compute minimal chains in polynomial time and memory space. Both these kinds of algorithms can still be very useful however. In this chapter, we will focus on those methods that give short (but not per definition minimal) chains: the 'approach algorithms'.
Algorithms that give short chains can, even though the chain is not optimal, still diminish the total amount of calculations steps by a great deal, since all such algorithms give chains that are shorter than when using the naïve method introduced in chapter 3. A second benefit of these 'approach algorithms' is the structured approach which forms their basis. Because of this, the method of using the chains can be understood and implemented easily, for example using computer programs.

There are, on the downside, also some negative consequences of the use of these algorithms. They do not give actual minimal chains, but only a relatively short one. Therefore, the resulting exponentiations are not optimal. There is however, no known algorithm which can compute minimal chains in polynomial time; such an algorithm might not even exist at all (see chapter 3).

8.1 Naive method

This algorithm was introduced before in chapter 3. Although it was already stated that its performance is far from optimal, we will still repeat it here, for the sake of completeness.
The naive method uses a very simple algorithm:

Given a goal *n*, create such chain where

$1 = a_0, a_1, a_2, ..., a_n = n$

and

$a_i = a_{i-1} + 1$

Or, put differently:

- take the last element of the chain
- add one to it
- add the result to the end of the chain
- Repeat this until *n* is reached

Using this algorithm however does not yield any benefits at all: the chains are as long as they can possibly get. Therefore it is never used.

8.2 Binary Method

8.2.1 The algorithm

The binary algorithm for calculating addition chains is one of the earliest known methods to try and solve our problem. It appeared as early as 200 BC in Hindustani mathematical papers [f(Knuth)]. The algorithm can be described in different ways, each of which yield exactly the same result:

Binary Algorithm

1. Take the goal *n,* for which the chain needs to be calculated
2. Rewrite *n* into the binary representation of that number
3. Use the number *X*. Initialize *X* to 1. Now we start at the leftmost side of the binary representation of *n*

4. Skip the leftmost 1 in this binary representation
5. Double *X.*
6. If the current position of the binary representation of *n* contains a *1*, then increment *X* by *1*.
7. Move one position to the right in the binary representation of *n.*
8. Repeat this algorithm from step 5 onwards, until the end of the chain is reached.

When using this algorithm, one can create a chain for any *n* which reaches its goal in a limited number of steps and which obeys the rules for addition chains. Let's look at an example. To keep things simple *n = 55* once more. We simply follow the steps given at the binary algorithm.

1. n = 55
2. n = 11.0111
3. X = 1, n=$\underline{1}$.0111
4. n = 1.0111
5. X = 2, n =$\underline{1}$.0111
6. X = 3, n =$\underline{1}$.0111
7. X = 3, n = 1.$\underline{0}$111
5. X = 6, n = 1.$\underline{0}$111
6. X = 6, n =1.$\underline{0}$111
7. X = 6, n=1.0$\underline{1}$11
5. X = 12, n=1.0$\underline{1}$11
6. X = 13, n=1.0$\underline{1}$11
7. X = 13, n=1.01$\underline{1}$1
5. X = 26, n=1.01$\underline{1}$1
6. X = 27, n=1.01$\underline{1}$1
7. X = 27, n=1.011$\underline{1}$
5. X = 54, n=1.011$\underline{1}$
6. X = 55, n=1.011$\underline{1}$
7. X = 55, n=1.0111
8. X = 55, n=1.0111, done.

Applying this binary method therefore yields the following chain:

1 2 3 6 12 23 26 27 54 55

The length of this chain is 9. The benefits of using this algorithm are clear: using a simple and straightforward method, we can create a relatively short chain. We can apply this algorithm in a limited amount of time and using a limited amount of storage space.

There are also other ways of calculating an addition chain for a given number, which produce the exact same chain as the binary method. One of these is by doing the calculations 'backwards'. Instead of starting at the left end of the chain, we now build it starting from the right. This is done as follows: start with the goal and either divide it by two if it is even or subtract 1 when it is odd. By repeating this simple algorithm, adding each of the intermediate results to the chain until 1 is reached, the same chain is created as when using the binary algorithm. The advantage of this method over the original binary method is, that less memory space is needed. Only the remaining value of n needs to be remembered. It also eliminates the need to first convert the number to a binary representation, a simplification mainly convenient for us humans.

Another algorithm, which produces different chains than the binary method but whose chains are of equal length, can be called 'doubling method'. This method simply adds the highest possible number to the last element of the chain.
For the example of *n = 55*, the chain becomes

1 2 4 8 16 32 48 52 54 55

This chain is once again of length 9. Note that the doubling method in every step adds the highest possible integer to the rightmost number in the chain and therefore that it is also a star chain.

For a long time, the binary method was thought to produce optimal chains. However, as our example of $n = 55$ shows, this is not always true. The smallest goal for which the binary method gives a non optimal chain is $n = 15$ [f(knuth)], where the binary method needs 6 steps, but the minimal chain has a length of 5.

Due to its simplicity though, the binary method has been widely cited throughout literature and is one of the most well known methods for calculating short addition chains.
The algorithm does have some downsides though. As mentioned, the algorithm is not capable of calculating truly minimal chains for every goal. However, as no known algorithm can do so within a reasonable timeframe, this downside could be forgiven. Another downside of the binary algorithm, especially when using the backwards methods, is that some bookkeeping is needed. Therefore, it is not worthwhile to use this method for small n, say $n \leq 10$ [f(knuth)]. Using this method only becomes useful for larger n or when the cost of multiplications (measured in time) is large enough.
Some other critical remarks on the use of the binary methods can be found in [f(knuth)]. A comparison between the binary method and other known methods can be found in chapter 12.

8.2.2 Length of the chain

By examining the way in which the binary chains are created, we can try to come up with a formula to calculate the length of the binary addition chains. First of all, we need one calculation step for each bit in the binary representation (this is the doubling step, step 5, in the binary algorithm), except for the leftmost 1-bit. The length of this binary representation can be easily calculated using the second base logarithm of the number. When this number is rounded downwards, you get the length of the binary representation. This can represented as:

$\lambda(n) = \lfloor \log_2 n \rfloor$

Clearly, here the notions given in chapter 4 come in handy.
Added to that, we need one step for each 1 in the binary representation:

$v(n) =$ *number of ones in binary representation of n.*

The total length of the binary addition chain then becomes

$l(n) = \lambda(n) + v(n) - 1$ [f(Knuth)]

The last '-1' is needed, because when counting the number of ones in the binary representation, the first 1 is also counted. However, no extra calculation step is needed for this 1-bit. The result would therefore be off by one.
Although this formula does not reproduce the actual addition chain, it does give an easy and quick way of calculating its length. This formula will be of further use later on.

Since the length of chains produced by the binary method, the backwards method and the doubling method are all equal, the same formula can be used to calculate the length of chains produced by these other algorithms as well.

8.3 Factor method

8.3.1 The algorithm

There are also algorithms that work entirely different from the binary one. One of them is the factor method. This algorithm is based on factorization, and uses the product-rule.
When trying to calculate the chain for n, the factor algorithm tries to compute a short chain by first calculating chains for the smallest divider of n. This is done by taking $n = p \cdot q$, where p is

the smallest prime number dividing *n* and *q > 1*. Now, we can calculate x^n by first calculating x^p and subsequently taking the *q*-th power of this outcome of x^p. In case x^n is a prime number itself, we can calculate x^{n-1} using the same tactics, after which we can compute x^n by multiplying x^{n-1} by *x*. If *n = 1,* we do of course need no steps at all to create a proper chain.
So far, so good, for the theory. We have split our goal *n* by using $n = p \cdot q$. But how do we calculate proper chains for *p* and for *q* when these numbers are bigger then 1? By simply applying this algorithm recursively: reinitialize the algorithm for any *p* and *q* unequal to 1.

Let us have a peek at what this algorithm would do for our running example of *n = 55*:
The smallest divider p of 55 is 5. This means that q will be 11. Now, we have to split each of these numbers into subsequent factors. Since 5 is a prime number, we take p = 1 and q = 4. Subsequent factors are 2 and 1. How 11 is divided into factors can be seen below:

For x^{55}:

$y = x^5 = x^4 \cdot x = (x^2)^2 \cdot x$
$y^{11} = y^{10} \cdot y = (y^2)^5 \cdot y$ (example taken from [f(knuth)]).

So far for the theory of this algorithm. Let us have a closer look at it. If we would like to create an addition chain for *n* using the description above, we first need to calculate the factors, recursively. This corresponds naturally to the creation of a binary graph. The number *n* is added as the root of this tree. If a node in the tree is bigger than 1, two children are added: the numbers *p* and *q* calculated as above. If a node has a value of 1, it will be a leaf node. Exemplary trees are shown in figures 8.1 and 8.2.
After the tree is created, we are left with the hard part of the algorithm: the creation of the corresponding addition chain. This part of the factor method is somewhat more difficult. It is a typical example of an algorithm that is relatively easy to apply yourself in a sloppy way, but which is rather difficult to specify to such a degree that a computer can execute it.
A detailed description of this part of the algorithm is not widely cited. In fact, I haven't been able to find any source which provides enough detail to base the algorithm on. In [f(knuth)], only a very general description with corresponding general example can be found. Therefore, I have added my own implementation of this algorithm.

So, the question remains: how do we go from the factor tree we created to the corresponding addition chain? Basically, by starting at the root and recursively adding the correct values for each of the branches.

Some design decisions can be made for this algorithm. First of all, when building the factor tree, we need to add *p* and *q* as children of node *n*. We can either choose to add *p* to the left and *q* to the right, or vice versa. I have chosen to add *p* to the right. We need to make yet another choice when converting the tree to the corresponding chain. We have chosen to plunge through the tree in a depth-first manner, first going down the rightmost branch, and afterwards working through the left branches.
These choices are however somewhat trivial. When making other choices here, we might end up with a different chain. However, the length of the chains will always be equal, no matter which design choice is made.

Let us have a look at the algorithms to build the factor tree and to calculate the corresponding addition chain, which together form the factor method:

Factor algorithm:

```
    1. Add n to the tree as its root node

BuildTree(n)
    2. If n does not equal 1:
        a. Take p and q, such that n = $p \cdot q$ and p is the smallest prime dividing n bigger
           than 1
        b. if p equals n and q equals 1 (in other words, if n was a prime itself), then set q
           to p – 1 and p to 1.
        c. Add p as the right-hand child of n and q as the left-hand child of n.
        d. If p is bigger than 1, call BuildTree(p)
        e. if q is bigger than 1, call BuildTree(q)
end of BuildTree

    3. Set the root of the tree as the current node
    4. Set x to 1
    5. Add 1 as the first element of the chain

CalculateChain(n)
    6. if the right hand side of the current node equals 2
        a. Double x
        b. Add x to the end of the chain
    7. else
        a. if the right hand side of the current node does not equals 1
            i. call calculateChain(right-hand side of node n)
    8. store the current value of x in $y_{node}$
    9. if the left hand side of the current node equals 2
        a. double x
        b. add x to the chain
    10. else
        a. if the left hand side of the current node does not equal 1
            i. call calculateChain(left hand side of node n)
    11. if this node is not the root of the tree and the right hand side child of this node is 1
        a. add the stored data in the current node to x
        b. add x to the chain
    12. else if this node is the root of the tree and the right hand side of the root is a one
        a. add the remembered value of the right hand side node to x and add x to the
           chain
```

Even though this algorithm contains only 12 steps, a full-blown run-through of the algorithm becomes quite elaborate when *n* increases. This is due to the fact that recursive calls are made at multiple points in the algorithm. Specifically the amount of pages needed to show the exact workings of this algorithm explodes with an increasing *n.* Therefore, we have decided not to add our running example of *n = 55* here*,* but to use *n = 5* instead.
However, the results for *n = 55* are included here (see figure 8.3 and the corresponding chain), and a full run-through of the tree algorithm can be found in appendix I.
Let us take a look at the factor method for *n = 5* now. First, building the tree:

```
FactorMethod(5)
1. Add 5 to the tree: 5
2. n is not equal to 1, so
    a. p = 5, q = 1
    b. Since p is equal to n (5) and q is equal to 1, n is a prime. So set q to 4 and p to 1
    c. Add p and q to the tree:
```

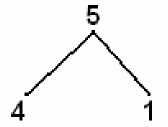


Fig 8.1.1: Building the factor tree for n = 5

```
    d. p is not bigger than 1, so do nothing.
    e. q is bigger than 1, so call BuildTree (4)
    2. 4 is not equal to 1, so
        a. p = 2, q = 2
```

b. p does not equal n and q does not equal 1, so do nothing
c. Add p and q to the tree:

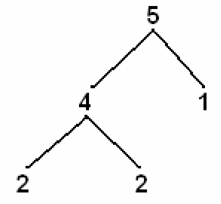


Fig 8.1.2: Building the factor tree for n = 5

d. p is bigger than 1, so call BuildTree(2)
2. 2 is not equal to 1, so
    a. p = 1, q = 1
    b. p does not equal n, so do nothing
    c. add p and q to the tree

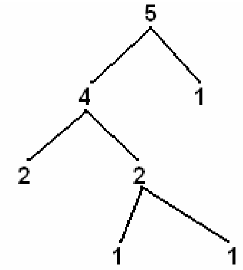


Fig 8.1.3: Building the factor tree for n = 5

    d. p is not bigger than 1, so do nothing
    e. q is not bigger than 1, so do nothing

e. q is bigger than 1, so call BuildTree(2)
2 2 is not equal to 1, so
    a. p = 1, q = 1
    b. p does not equal n, so do nothing
    c. add p and q to the tree

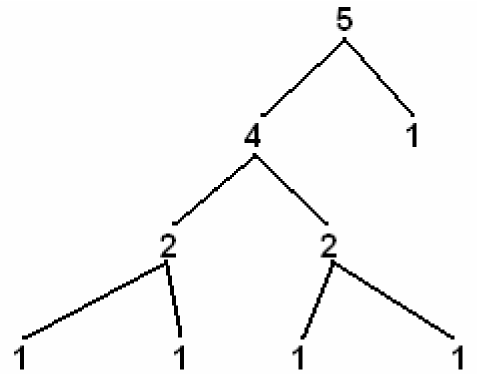


Fig 8.1.4: Building the factor tree for n = 5 (final result)

    d. p is not bigger than 1, so do nothing
    e. q is not bigger than 1, so do nothing

This ends the first part of our algorithm: building the factor tree. As can be seen, all the leaves have 1 as value.
Now, let’s apply the rest of the algorithm.

3. Set 5 to the root of the node
4. Set x to 1
5. Add x as the first element of the chain

CalculateChain:

6. The right hand side does not equal 2
7. So, else:

    a. The right hand side of the current node equals 1
8. store the current value of x in the current node:

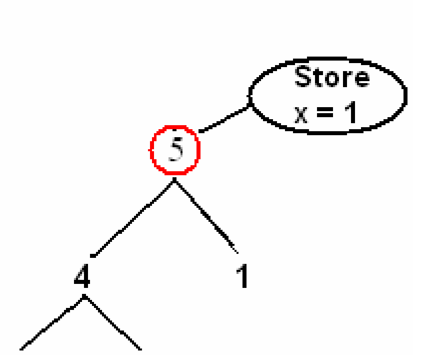


Fig 8.2.1: Store the value of x in the node

9. The left hand side of the current node does not equal 2
10. So, else:
    a. The left hand side of the current node does not equal 1, so
        i. call calculateChain(4) and set x to its returning value

        6. The right hand side now does equal 2.
            a. So x = 2
            b. Add 2 to the chain
        8. store the value of x in the current node:

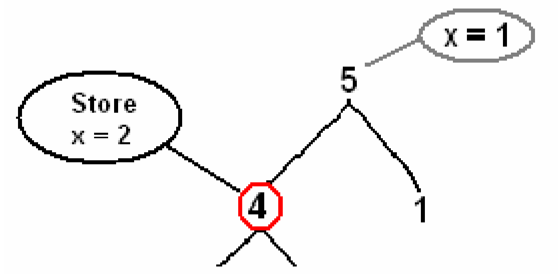


Fig 8.2.2: Store the value of x in the node.

        9. The left hand side of the current node equals 2, so:
            a. x = 4
            b. add 4 to the chain
        Steps 10, 11 and 12 do not apply
11. The current node is the root node, so do nothing
12. Else: the current node is the root node, and the right hand side of the root is a one, so:
    a. add the remembered value of the right hand side, 1, to x. So x = 5. Add 5 to the chain.

After this, the algorithm terminates, which gives us the following chain:

1 2 4 5 [9]

This result might not come as a surprise. The question might arise why we went through all this trouble, when we could have built this chain using the binary method, or even our common sense. However, the factor method does allow us to shorten chains on many occasions; it beats the binary method on numerous occasions. However, sometimes the binary method does give better results, the smallest case in which this happens is *n = 33*. [f(knuth)]. Have a closer look at these results in chapter 12.

Let's see the algorithms results for *n = 55* (the full run-through of the factor method for *n = 55* can be found in appendix I)

First, the tree is constructed (see fig 8.3). It is not hard to see how this tree was constructed. The smallest divider of 55 is 5, the remainder is 11. So 5 and 11 are added to the tree. This process is repeated until the entire tree in figure 8.3 is built.

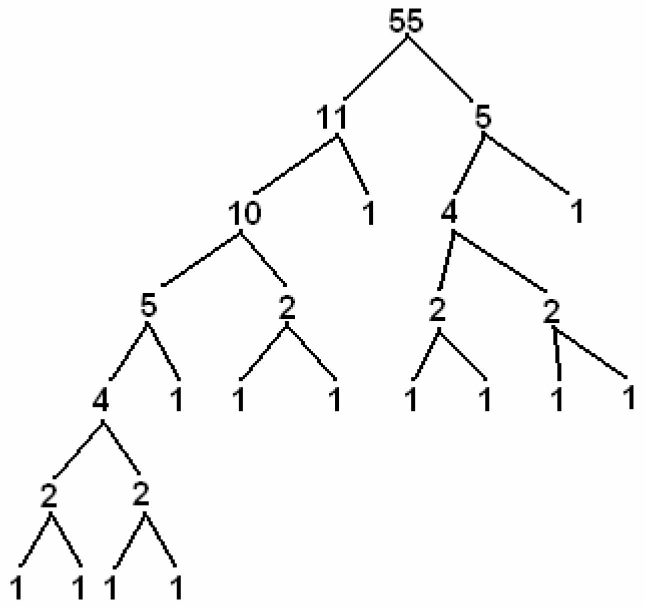


Fig 8.3: the factor tree for *n = 55*

Afterwards, we need to go from this tree to the corresponding chain.
According to the algorithm, we first set x to one and add this to the start of the chain. The algorithm then descends down the right path of the tree. When it encounters its first 1 (the right child of the 5), it stores 1 (the current value of x) in the node with value 5.
Afterwards, it takes the left branch of the 5-node. Since both of its children have the value 2, it doubles the current value of x and adds it to the end of the chain twice. Then, it adds the remembered value in the 5-node (1) to x and adds it to the chain. Now, we have the following chain:

1 2 4 5

Note that so far, the algorithm takes the exact same steps as in our previous example for *n = 5*. Now, the algorithm takes the left branch of the root node. It travels to the 11-node. There, it sees that the right child is a 1, so it stores the current value of x, which is 5. It goes on to the 10-node, and now sees the right hand child equals 2. So it doubles x, which becomes 10, and adds the new value to the chain. It travels on towards the 5-node, and stores the value of x here. At the next node (the 4-node), it doubles x and adds the new value to the chain for both its children. So, by traveling downwards, we have taken three doubling steps. Now, the chain looks like

1 2 4 5 10 20 40

All that is left to do now is travel back up the chain and add the remembered values. So at node 5, we add the value 10 and at node 11 we add 5 to the end of the chain.
The final chain then is

1 2 4 5 10 20 40 50 55.

with a total length of 8. Note that the binary method needed 9 steps in total.
Now, let us take a look at what happens when we adapt the algorithm slightly. For instance, what would happen if we would first go down the left branch of a node, instead of the right one? As mentioned before, the chain might be different, but the chain length will be the same. This is logical, for we use the same tree and nodes are added to tree under the same restriction as with the regular factor algorithm. Let's have a look at what happens when we use this left-oriented algorithm for the example of *n = 55*.

First, we take the left branch. This will yield the chain 1 2 4 5 10 11 (which is, not coincidently, a chain consisting of the elements of the leftmost branch itself). Now, we take a walk down the right branch. We store 11 at node 5, since it right child is a 1. We go down to node 4. Here, the left child is a two, so we use a doubling step. Since the right child is a 2 as well, we use yet another doubling step. Thereafter, the algorithm goes one step upwards, and adds the stored value 11 to the last element of the chain. Now, the algorithm is finished and can terminate. This gives us the chain

1 2 4 5 10 11 22 44 55

This chain is different from the one using the right oriented algorithm, but it reaches its goal in 8 steps as well, as could be expected.

8.3.2 Length of the chain

Even though, on average, the factor method produces shorter chains than the binary method, it also has its downsides. With the binary method, we have a formula which can be used to calculate the chain length, without actually producing the chains themselves. No such formula seems to exist for the factor method, or at least, no such formula is known.

On the other hand, when looking at the graph containing all the factors, we can come up with a formula for the chain length. Let's have a closer look at the algorithm once more. More specific, let's have a look at how the algorithm converts the factor tree to the corresponding chain. It does so by walking through the tree in a depth first manner. Whenever either a 1 or a 2 is encountered, the algorithm does some calculations and adds the next number to the chain. Since adding an element to the chain increases its length by one and these steps are the only occasions in which this chain length is increased, the final length of the chain is equal to the number of times we encounter a 1 or a 2 when walking through the chain.

On first sight, one might think that the length of the chain is now simply obtained by counting all the ones and two's in the tree. Life is, however, not *that* simple. Let's have another look at the tree for our example of *n = 55* (fig 8.4).

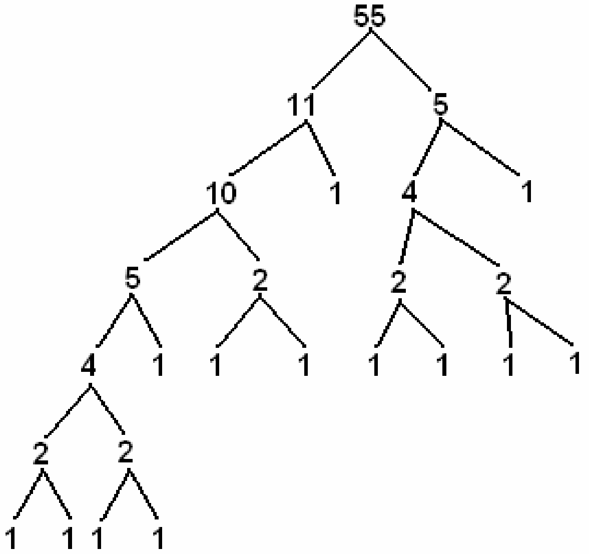


Fig 8.4: factor tree for *n = 55*

Each of the twos in the figure has two ones as its children. However, no calculation steps will be taken for these two ones. So, if we would simply count the number of ones and twos in the graph, we would get a chain length that is off by twice the number of 2's in the graph.
The correct formula therefore is:

l(n) = (number of 1's in the factor tree) – (number of 2's in the factor tree)

This is equal to

(number of nodes in the factor tree bigger than 2) + 1

This also is logical. Each node in chain, bigger than 1, has two children. If a child of a node is either equal to 1 or two, an extra step will be added to the chain. If a child is bigger than 2 however, no step is added to the chain at this time. Since children are always smaller than their father and the algorithm keeps running until a 1 is added as the leaf node, the tree building algorithm will always terminate and each branch in the tree will end in a 1-node.
If a node is bigger than 2, two nodes will be added as its children. When these two children are added, the (current) number of branches nodes is increased by one. To each branch, eventually a two or a one will be added. So, for each branch, eventually a step in the addition chain will be taken. So, whenever an extra branch is added, eventually this will result in an extra step in the addition chain. Therefore, the number of nodes that are bigger than 2 directly corresponds to the number of steps in the chain.

8.3.3 Factor vector chains

A method similar to the factor method can be used for calculating short vector addition chains. For instance, let us look at a two-dimensional vector, (a, b), for a > b it holds that $a = n \cdot b + s$ and $0 \leq s < b$.

Now, the vector addition chain can be calculated as follows:
First build vector chains for (n, 0) and (s, 0) and a chain for b. These chains can be calculated by using any of the known algorithms. Next, calculate the vector chain for $(n \cdot x, x)$. The final step that needs to be taken is adding the vector chain for [s, 0] to the final result.
This gives a total chain length of *l([a,b]) = l(n) + l(b) + l(s) + 2*, in many cases shorter than the original *l(a) + l(b) + 1* which would result from using the naive method. [r(bergeron)]. Often however, the final resulting chain might be even shorter, since steps from calculating (n, 0) and (s, 0) may overlap.
For some clarity, let's look at an example: (43,5)
43 = 8*5 + 3
Let's first come up with the chains for (8,0), (3,0) and 5:

(8,0): (1,0), (2,0), (4,0), (8,0)
(3,0): (1,0), (2,0), (3,0)
5: 1, 2, 4, 5

By combining these results, the final chain would be

(0,1), (1,0), (2,0), (3,0), (4,0), (8,0), (8,1), (16, 2), (32,4), (40,5), (43,5)

of length 9 (note that this is even shorter than l(n) + l(b) + l(s) + 2, which is 3 + 3 + 2 +2, since the step (2,0) is present in both the chains of (8,0) and (3,0).

Using the naïve method would result in the chain

(0,1), (1,0), (0,2), (0,4), (0,5), (2,0), (3,0), (5,0), (10,0), (20,0), (40,0), (43,0), (43,5)

of length 11 (which is equal to l(a) + l(b) + 1 = 3 + 7 +1 = 11)

For the general understanding, let's look at yet another example, in which there is no overlap between (n, 0) and (s, 0) and in which we can use the knowledge of our running example of chains for 55:
(55,6)
55 = 9 * 6 + 1
(9, 0): (1,0), (2,0), (4,0), (8,0), (9,0)
(1,0): (1,0)
6: 1, 2, 4, 6

Resulting in the chain

(0,1), (1,0), (2,0), (4,0), (8,0), (9,0), (9,1), (18,2), (36, 4), (54,6), (55,6)
This chain is of length 9 (l(n) + l(b) + l(s) + 2, which is 4 + 0 + 3 +2), still a better result than the chain of length 12, obtained when using the naïve method (l(a) + l(b) + 1, which is 8 + 3 + 1)

8.4 Window method

8.4.1 The algorithm

Yet another algorithm is called the window method. The basic working of the window method is as follows. It uses the binary representation of the goal *n* and cuts it into parts, called windows. Each of these windows has a maximum length, the window length. Afterwards, an addition sequence is built, containing the values of each of these windows. Thereafter, this resulting chain is extended in order to cover the goal *n*, by using specific rules that will be described in a moment. Many sources describe the window method [c(sauerbrey), f(knuth), amongst many others]

Let's have a closer look at the exact working of this algorithm.
First of all, an appropriate window length *w* needs to be chosen. Typical values for this window length are small odd integers, like 3 or 5. The maximum size of the windows then becomes 2^w.

In order to calculate addition chains using the window method, one can use the following algorithm.

Window Algorithm

1. Choose a window length w
2. Rewrite *n* to its binary representation.
3. Divide the binary representation of *n* into windows, starting from the left to the right. Each of these windows has to start and end with a 1. The length of each of the windows is at most the window length.
   To divide the binary representation in the windows properly, the following sub-algorithm is used. Start at the leftmost bit of the binary representation
   a. Is the current bit set to 0? Then move one bit to the right.
   b. Is the current bit set to 1?
      i. Is this the first bit of the current window that is encountered? Then the current bit should be added as the start of the window.
      ii. If this is not the first bit of *n*, then check whether any 0's were skipped in the last steps. Check to see if the possibly skipped zeroes and the current one-bit, when added to the current window, exceed the maximum window size.
         1. if so, forget about the zeroes at all. The current one-bit is the start of a new window.
         2. If not, add the zeroes and the current one-bit to the current window.
4. Repeat step 3, until the right hand side of the binary representation of *n* is reached.
5. Calculate the value for each of the windows (for example, a window 101 has the corresponding value 5)
6. Build an addition sequence that covers all the windows.
7. Start at the first bit to the right of the leftmost window (note that this is not necessarily the window with the largest value). Set X to the value of this leftmost window.
8. Double X and add X to the chain.
9. If the right hand side of a window is reached, add the value of this window to X and add the new value of X to the chain.
10. Move one bit to the right in the binary representation
11. Repeat steps 8, 9 and 10 until the end of the binary representation is reached.

Once again, we see an algorithm that is quite elaborate. An example might help to clarify things. Let's start with our usual example of *n = 55,* with a window length *w = 3*.

1. Set the window length w to 3.
2. The binary representation of 55 is 11.0111
3 & 4. Start at the left of the binary representation. Each window starts and ends with a 1. Therefore, the first window starts with the leftmost one. The next bit is once again a 1. Since the total window size (2) is smaller then the window length (3) , this 1 can be added to the current window. Now a 0 is reached. This is skipped for now. The next bit is a 1 again. There is one skipped 0 still 'in memory'. If this zero and the current one would be added to the current window, the total window length would become 4 and would thus be larger than the maximum window length. This is not allowed and therefore, we can forget about this 0 for now. The 1 is the start of the new window. Hereafter, two more 1's are added to this window. Finally, *n*, when divided into windows, looks like this:
$n = \underline{11}\ 0\ \underline{111}$.
5. The leftmost window (11) has the value of 3, the right-most window (111) has the value of 7.
6. We need to build a addition sequence for {3, 7}. This sequence could be 1 2 3 6 7.
7. We need to start at the bit to the right of the leftmost window. This is at the zero. n= 11 $\underline{0}$ 111. X should be set to the value of the left-most window. Therefore: X = 3.
8. X = 6. n = 11 $\underline{0\ }$111.
9. No right hand side of a window is reached. X = 6.
10. n = 11 0 $\underline{1}$11
8. X = 12
9. No right hand side of a window is reached. X = 12.
10. n = 11 0 1$\underline{1}$1
8. X = 24
9. No right hand side of a window is reached. X = 24.
10. n = 11 0 11$\underline{1}$.
8. X = 48.
9. The right hand side of the right-most window (111) is reached. Therefore, the value of this window (7) should be added to the current value of X. Now, *X = 55*.
10&11 The end of n is reached. *X = 55*.

The total chain now becomes

1 2 3 6 7 12 24 48 55

Thus, applying this algorithm yields a chain of length 8. Even though this is a shorter chain than the one given by the binary method, one could ask if using this more complex algorithm is worthwhile when gaining only so little.
Normally however, the window method is used when calculating chains for relatively large *n*'s. Therefore, one more example will be given for, let's say, *n = 438479*. This time, the algorithm will not be described as elaborately as before though, and in-between steps are left for the reader to check.

For *n = 438479*, the according binary notation is

1101011000011001111

With a window length set to 5, the distribution of the windows becomes

$\underline{1101}$ 0 $\underline{11}$ 0000 $\underline{11001}$ $\underline{111}$
13 3 25 7

The addition sequence should therefore cover {3 7 13 25}. The addition sequence then becomes:

1 2 3 5 7 12 13 25

With a total length of 7.
Now, the squaring and adding begins. The final chain then becomes

1 2 3 5 7 12 13 25 26 52 104 107 214 428 856 1712 3424 6848 13696 27392 54784 54809 109618 219236 438472 438479

This algorithm thus gives a chain of length 25 for *n = 438379*.

8.4.2 Length of the chain

When using the window method, it is not as easy to give a formula that produces the length of the resulting chains as it is when using the binary method. We can try to give an estimate of the length however. When looking at the algorithm, there are several different parts that are present in the calculation method. By trying to give the length for each of these parts, we can come up with a formula for the total length of the chain.

First of all, there is the length of the addition sequence (step 6). Thereafter, the number of doubling steps (step 8) needs to be added. As a third part, the number of additional calculations (step 9) needs to be added.
The total number of steps needed by this algorithm then becomes:

(Length of addition sequence) + (number of doubling steps) + (number of additional steps)

(formula 8.4.2.1)

The number of doubling steps can be calculated relatively easy. When looking at the binary representation of the goal *n*, notice that a doubling step is used for each bit, except for those bits that are part of the leftmost window. Also, the number of additional steps can be specified. For each window, one addition step is needed, except for the first one. The previous formula then can be rewritten to:

(length of addition sequence) + (length of binary representation –length of first window) + (number of windows – 1) (formula 8.4.2.2)

When applying this formula to the example of *n = 438379*, the total length of the addition chain becomes 7 + (19 – 4) + (4 – 1) = 25. So, for the window method, *l(438379) = 25*, which corresponds to the value found in 8.4.1.
The *binary* chain for 438379 would have had a length of 28, which can be checked using the formula presented in chapter 8.2.2.

However, while the formula seems to calculate the length of the chains correctly, there are a couple of problems that need to be mentioned. To start with, there is no easy formula for calculating the length of the addition sequence, used in step 6 of the window algorithm. As a matter of fact, up until now we have not yet specified the actual algorithm for calculating the addition sequence at all.

***

8.3.2.1 Intermezzo: A closer look at addition sequences

In this intermezzo, an algorithm will be given that can be used to calculate addition sequences. Remember that, as was mentioned in chapter 5, while regular addition chains have a single goal, the goal of an addition sequence is (not surprisingly) a sequence: $\{n_1, n_2, \ldots, n_m\}$

In order to calculate an addition sequence, there are several tactics that can be used. In this intermezzo, we will present only a small selection.

One could calculate addition chains for each goal n_i in the addition sequence, using any of the algorithms described in this paper (for instance, the binary algorithm), and

subsequently combining al these chains into one. However, this evidently will not always provide the most efficient chain possible.
An alternative is to store all sequences for up to a certain bit length and simply load the needed chain for the current sequence. This sounds easier than it might be in reality. First of all, the range of possible addition sequences is far wider than the amount of regular addition chains. This number is especially large when the amount of goal integers ({n_1, n_2, …, n_m}) is not known in advance. When working with the window method, we do have this problem: beforehand we do not know how many goals the addition sequence of step 6 of the window algorithm will have. An added difficulty comes from the fact that the used window might not be known in advance too. This makes it difficult to decide what the maximum size of the addition sequences to be stored should be. On the bright side however, only sequences producing odd numbers will need to be stored, since every window has a 1 in the rightmost bit. Still, due to the incredibly large amount of chains that would need to be stored, this approach seems to be doomed to remain just a theory.

Now, what would be an appropriate method to build an addition sequence?
First, let's have a look at some known facts for the sequences. Any of the goals of the sequence will need to be present in the final resulting chain. In the example of *n = 438479* mentioned before, an addition sequence had to be found for the numbers 3, 7, 13, 25.
Apart from that, it is known that any addition chain or sequence will need to start with the numbers 1 and 2.
Therefore, part of the resulting chain is already known. For our example, the chain will at least contain the following parts:

1 2 3 7 13 25

Clearly, this chain is not finished yet, as not every number can be created by adding two previous elements. What is left to do here is to find an algorithm that can be used to fill in the gaps:

Addition sequence algorithm

1. First, build a pseudo-chain, containing the numbers 1, 2 and all of the goals.
2. Start at the right end of this chain-under-construction
3. Check whether adding two other numbers already present in the sequence add up to create this number.
4. If not, an extra element needs to be added to the chain, in such a way that now the current element can be created by addition of two other numbers. In order to do so, take the current number and subtract the integer to its left from it. Add the result to the addition sequence.
5. Go one element to the left in the chain and repeat steps 3 and 4 until the leftmost number of the addition sequence (1) is reached.

Let's have a look at our example and apply the algorithm.

For the sequence (3, 7, 13, 25):

1. 1 2 3 7 13 25
2. Start at the number 25
3. 25 cannot be created by adding to other elements of the chain
4. So, we need to add an element to the chain. The number left to 25 is 13. *25 – 13 = 12*. So, we need to add 12 to the chain, which now becomes

   1 2 3 7 12 13 <u>25</u>.

   To make clear that 25 has already been handled, it has been underlined.
5. Next, repeat steps 3 and 4 for the element 13:
3. 13 can be created by adding 12 and 1; 1, 2, 3, 7, 12, <u>13</u>, <u>25</u>
5. Now, take a look at element 12.
3. 12 cannot be created by adding two other elements of the chain.
4. The first element left of 12 is 7. So add 12 – 7 = 5 to the chain. The chain now becomes

   1 2 3 5 7 <u>12</u> <u>13</u> <u>25</u>.

Now, all the other elements of the chain can be created by adding two other elements of the chain (7 = 5+2, 5 = 3+2, 3 = 2+1 and 2 = 1+1). Our algorithm therefore does not need to add any other elements.

While this method works fine for this example, it will not be efficient for every addition sequence. The algorithm gives especially poor results when calculating on partial sequences with large gaps between the elements. Take for example the partial sequence that needs to include the numbers

47, 117, 343, 499, 933, 5689. (example taken from [t(Gordon)], which does present the partial and the resulting chains, but not how this chain is built)
When calculating the addition sequence with the algorithm given just now, the result is far from optimal. The number 5689 is that much bigger than the second largest number (933), that one needs to use the subtraction step 4 of the addition sequence algorithm multiple times before reaching the next biggest number. The rear end of the chain would then become

933 1024 1957 2890 3823 4756 5689.

However, this sub-chain is not optimal. A chain of lesser length can be fabricated (see below). The algorithm becomes less and less efficient as the gap between two subsequent numbers increases.

In order to overcome this problem, the algorithm needs to be adapted. Whenever an element of chain is more than three times as large as the preceding number, the ‘backwards binary algorithm’ (see chapter 8.2.1) is used. Taking the example with the numbers 933 and 5689 as the two largest numbers of a partial chain, the right part of the sequence then becomes

933 1422 2844 5688 5689

Notice that this sequence is two shorter than the one used by the original algorithm. However, this might be a bit deceiving, because now another number needs to be added to go from 933 to 1422. This could have further implications when applying the remainder of the algorithm. Though the improvement might not be shown by this example, one needs to remind that the benefits of this adapted algorithm become more apparent when the gap between two subsequent numbers increases.

When using this adapted algorithm, the addition sequence is calculated as follows (the first few steps, using the backwards algorithm, are not specified here):
1, 2, 47, 117, 343, 499, 933, 1422, 2844, 5688, 5689
1, 2, 47, 117, 343, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 47, 117, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 10, 47, 117, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 10, 47, 55, 117, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 10, 47, 55, 91, 117, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 10, 47, 55, 91, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 10, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 8, 10, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 8, 10, 18, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 8, 10, 18, 36, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 8, 10, 18, 36, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 8, 10, 11, 18, 36, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689

1, 2, 8, 10, 11, 18, 36, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689
1, 2, 4, 8, 10, 11, 18, 36, 47, 55, 91, 109, 117, 226, 343, 434, 489, 499, 933, 1422, 2844, 5688, 5689

giving an addition sequence with a length of 22. The sequence created by using this algorithm is exactly equal to the one given in [d(Bos & Coster), t(gordon)]. As a comparison, the original, not adapted algorithm would produce the chain

1 2 3 5 8 10 18 21 26 47 65 91 109 117 226 343 434 499 933 1024 1957 2890 3823 4756 5689

with a length of 24.

While the adapted algorithm is fairly easy to use and does not require an extensive amount of difficult calculations, one does wonder whether it is easy to improve it even further.
A possible downside of this algorithm might be, that it considers mainly star steps. Although it does provide a check to see whether a number can be created using any two present chain elements, and thereby allows the use of non-star steps, in the case no two such numbers are present a star step is always added.
A possible improvement would be to keep track of all possible sequences while calculating backwards. This way, more optimal chains might be created that use non-star steps more often. However, the complexity of this algorithm increases dramatically. More calculation time and storage space is needed in order to keep track of all the possible addition sequences. The algorithm might create shorter sequences, but since here it is used only as a part of a broader algorithm, the added complexity might not be worthwhile. Apart from that, further improving the addition sequence algorithm falls outside the scope of this paper.

*** end of intermezzo

Back to the formula (8.4.2.2) that calculates lengths of addition chains computed using the window method: *(length of addition sequence) + (length of binary representation –length of first window) + (number of windows – 1)*
In the intermezzo, we have looked at an algorithm to calculate the addition sequence part. Now what about its length? There does not appear to be an easy formula that gives the length of the addition sequence. An upper bound could be given though. Remember that an addition sequence can also be created by calculating a chain for each of its elements and subsequently combining those. Now, by using the binary method for each of the intermediate chains, we have a formula to calculate the length of each of these chains. Therefore, we also have an upper bound of the resulting addition sequence.
A second problem of this formula (*formula 8.4.2.2*) is, that there is no easy way to know the length of the first window or the total number of windows, without explicitly calculating them. Yet another problem is, that there might be overlap between the addition sequence and the rest of the calculations. Applying the formula to our running example of *n = 55*, yields the following result:

1 2 3 4 7

Since the leftmost window has the value 3, the following numbers are added to the chain:

6 12 24 48 55

giving the chain

1 2 3 4 6 7 12 24 48 55

The length of the addition sequence is 4, the number of doubling steps needed is also 4, and one additional step is needed. This gives a total chain length of 4 + 4 + 1 = 9.

As we have seen before though, a chain of length 8 could be given using the window method with an alternative addition sequence.

And then there is yet another remark to be made. So far, no special attention was given to the choice for a proper window length. There is not a single window value that performs best for all *n*. Also, there is no (known) formula which specifies what window values are optimal for certain (classes of) *n*. In general, using medium sized windows produces shorter chains. By using larger window values, the addition sequences become more difficult to construct. When using a small window value, storing tables with minimal chains for each of the sequences becomes an actual option. This would eliminate the need to calculate the sequence (step 6 of the addition sequence algorithm) each time the window method is used.
Even though our formula (8.4.2.2) does not allow us to immediately derive the chain length for any given *n*, it can still prove to be a useful one. It can shorten the amount of time needed for calculations, when only the length of the chain and not the actual chain itself is needed. When looking at the formula, we see that we need the length of the addition sequence, the binary representation and the length of the first window of *n.* This means that in order to be able to calculate *l(n)* we need to apply six steps of the window algorithm. Steps 7 – 11 can thus be eliminated when one is only interested in *l(n)* itself.

Note that in fact, the binary method is a special case of the window method, for the case that the maximum window length is set to 1. [d(bos), q(kunihiro)].

8.5 Power tree method

8.5.1 The algorithm

So far, we have taken a look at methods that use an algorithm to calculate a single chain. Now, it is time to try and take a look at a somewhat different approach: using a tree based algorithm.

The basic thought behind the algorithms considered so far tried to construct a chain as follows:

- consider the chain up till now
- Calculate the next value to add to the chain (either performing calculations starting at the front or the back of this chain)
- Add the calculated value to the chain
- Repeat the previous steps until a proper addition chain has been constructed.

All of these methods thereby construct a single chain for a single goal, *n*.
By using a tree-based method, this basic construction method can be altered.

The basic method of the power tree algorithm is to construct a tree, with a root of 1 and up to a certain depth, where each node in the tree has a number as value. For each node, the corresponding addition chain can be found by taking its path to the root up the tree. Nodes can only be added as child to a node in the tree if they correspond to taking a star-step in the chain. Let’s look at a picture of such a tree for the first couple of layers.

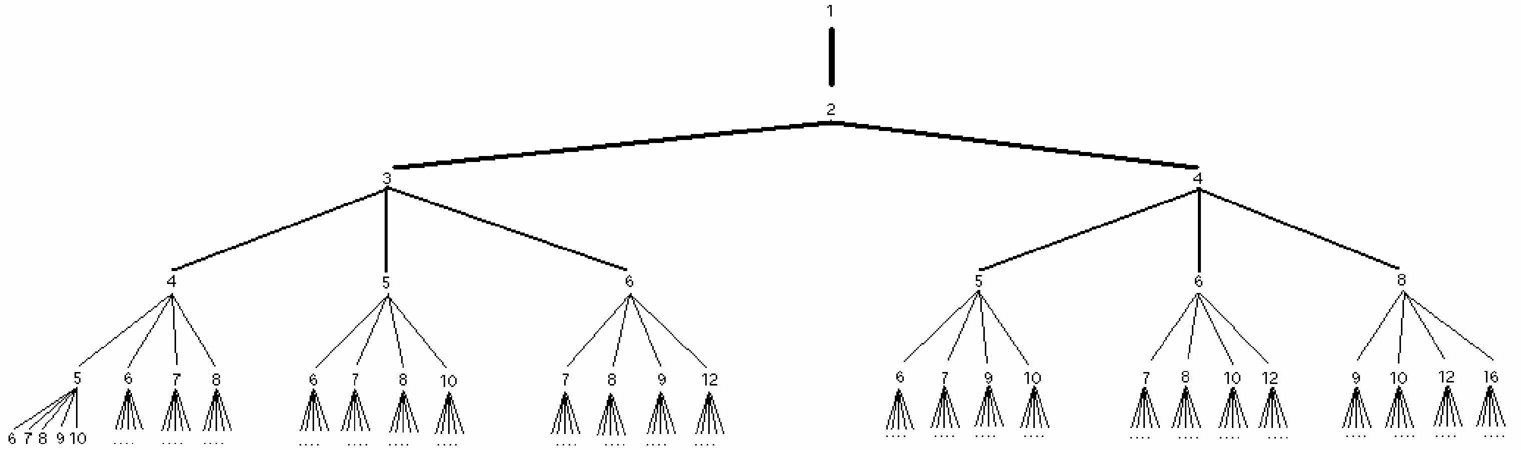


Fig 8.5: Full-blown power-method tree

In order to construct this tree, we started with a tree containing a single node; the root with value of 1. Now, we need to add all the possible nodes to this root, which can be reached using a star step. The only applicable number here is 2. Now, we go down a layer, and try to add more nodes. Here, the nodes 3 (1 + 2) and 4 (2 + 2) are added. The construction of the rest of this tree should now be clear.
Essentially, the tree is constructed in a breadth first manner.

While figure 8.5 does show how the full-blown power tree is constructed, it also shows its greatest weakness. The number of nodes added to this tree grows exponentially as the depth increases. For each of the *n* nodes at layer *l*, *l* nodes are added. So, at each layer l, ***l!*** nodes are added.
Not only does it proof to be cumbersome and time-consuming to construct such a tree by hand; a computer program will also need vast amounts of time and memory to construct such a tree of large depths.

One possible way to diminish the total amount of nodes that need to be stored is by allowing numbers to be present in the tree only once. This means that we do need to keep track of which number is already present in the tree, but that on the other side we can prune all those nodes which are already present. Using this adapted algorithm, we get the tree in figure 8.6 for the first 8 layers:

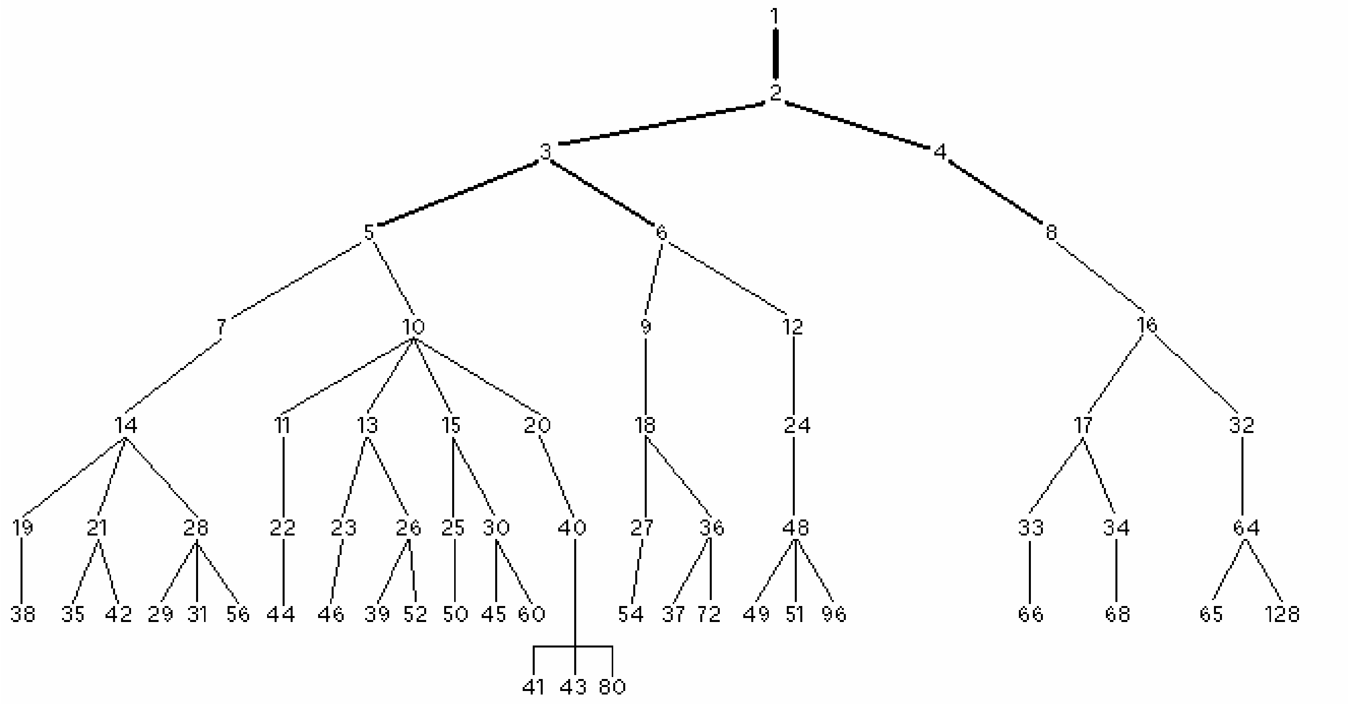


Fig 8.6: The first 8 layers of the power tree method.

The exact algorithm would then be as follows:

Power tree algorithm: up to level x

```
1. Add a node with value 1 as root of the tree
2. Set the current layer to 1
2. While the current layer is smaller than the needed layer:
        A. While there are still nodes to add to the current layer
                i. Find the leftmost node at layer l
                ii. For i = layer l down to 1
                        a. Get the i-th (grand)parent of the current node)
                        b. add a child to the current node, with as value the sum of the values of
                           its i-th parent and the current node.
                iii. Get the next node at the current layer
```

8.5.2 Length of the chain

The power-tree method does not seem to have a simple formula to determine the length of the chains it produces.

That, however, is not all there is to say about the length of the chains. The tree based method creates star-chains only. As was mentioned before, star chains do not provide minimal chains for each n, with *n = 12509* [bergeron] being the first example of a minimal star chain which is longer than the overall minimal chain.

The power tree method however does not provide minimal star chains. The first couple of numbers for which the power tree method provides less than optimal star chains are *n = 77*, *n = 154* and *n = 233*. The first case in which the power tree method defeats both the binary and the factor method is for n = 23. The first case in which the factor method beats the power tree method however is for *n = 19879* [f(Knuth)]. The factor method does not beat the power tree method often. According to [f(knuth)], for n ≤ 100.000, there are only 6 cases in which the power tree method loses to the factor method. On the other hand, it ties 11191 times and wins the majority (88803 times) [f(knuth)]. More on this can be found in chapter 12, when comparing the various approach algorithms.

Looking at the results as found in f(knuth), it appears as if the power tree method yields the shortest chains. While this might be true, it is worthwhile to look at some other aspects of this method. As mentioned before, the power tree method is different from the previously mentioned algorithms, in that it is both layer-based and tree-based.
This comes with both benefits and downsides. A plus of using a layer-based method is that applying the algorithm once will yield a short chain for all numbers *n* up to a certain length. Thus, running the algorithm once will give a result for a wide branch of numbers. On the downside, it is not possible to calculate a chain for a single goal *n* quickly. The power tree method will need a relatively large amount of both time and memory in order to compute all chains for larger layers. When you are only interested in one chain for a single number, this added complexity could be unwanted.
The algorithm could be altered in such a way that only a chain for a certain number is calculated. When doing so, parts of the tree from which the goal can't be reached can be pruned, saving memory and search time further down the tree. Still, this adapted method would have to search a relatively large search space when compared to simpler algorithms such as the binary one.

As a note, the full-blown power tree method, which does not use any pruning, does calculate minimal star chains, but suffers from the downsides mentioned before. More on the results of the power tree method can be found in chapter 12.

8.5.3 Adaptation to the algorithm

The algorithm as presented so far can be adapted in a couple of ways. For instance, one could look at the pruning rules and design a less crude algorithm. At this point however, we will not take a further look at that; in chapter 10 we will present alternative pruning algorithms, when presenting algorithms for producing absolutely minimal addition chains.

One way to adapt the power tree method is by adding the largest element first, instead of the smallest. For the full-blown power tree method, adapting the algorithm in this way would produce chains of exactly the same length. However, in the power tree method with pruning, after an element is added for the first time, it is not included in the tree anymore. As a result, the order in which nodes are added to the tree does matter. Altering this order will cause changes in the layers further down the tree.
One might wonder whether reversing the tree in such a way would yield better results. Let's take a look at how this reversed tree would look.

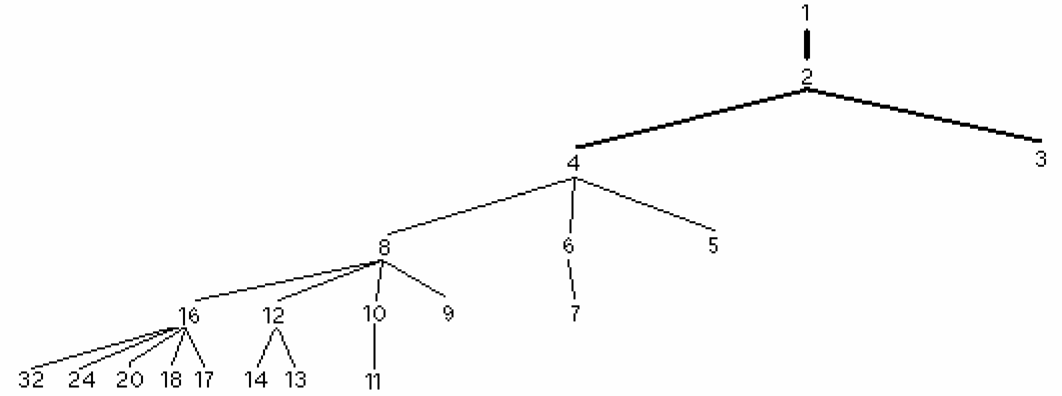


Fig 8.7: Reversed power tree.

Looking at this figure, it becomes clear that the reversed power tree method is strongly left-oriented; the right branches of each node hardly have children at all, if any, while the number of children of nodes on the left hand side increases at each layer. Note, that when comparing fig 8.7 to figure 8.6, that in the first 6 layers, the first method consists of 20 nodes where the second one has 21 (the node with value 15 is absent at level 6 of the reversed power tree method).

When comparing the results of the reversed power tree method to those of the doubling method, it shows that both algorithms generate the exact same chains. Giving this another thought reveals the logic behind this. When using the reversed power tree method, at each node we first add the largest possible number, and subsequently all other nodes we encounter when moving up the tree. Now, when comparing this with the description of the doubling method ('simply add the highest possible number to the last element of the chain', see chapter 8.2.1) we immediately see the resemblance between these two algorithms.
So, the reversed power tree method produces chains of the same length as the binary method. In chapter 12, we will have a closer look at the achievements of these and the other methods mentioned thus far.

8.6 Other methods:

There are yet some other methods which produce relatively short chains. These methods are mentioned here for the sake of completeness and for the interested reader to find out more about them, but they will not be treated in a great deal of detail here. Please note that none of the methods and adaptations presented in this section are present in the computer program, nor are they taken into account when discussing the results.
First of all, [g(bergeron)] mentions a couple of methods, some of which have not been presented so far in this paper.

The dyadic strategy [g(bergeron)] limits the choice of all possible candidates for the next step to

$$\left\lfloor \frac{N}{2^j} \right\rfloor, \text{ where } j = 1, \ldots, \lambda(n) - 1.$$

In order to improve the speed of this dyadic strategy, Bergeron also mentions the fermat strategy, which is of a similar design as the dyadic one:

$$\left\lfloor \frac{N}{2^{2\wedge j}} \right\rfloor, \text{ where } j = 1, \ldots, \lambda(\lambda(n) - 1).$$

When comparing these two strategies to those methods described before, one could note that these are not actual algorithms describing a detailed plan for how to calculate a specific chain. Instead, these two strategies rather limit the search space for possible elements of the chain.

8.6.1 M-ary method

Another often mentioned method is the m-ary algorithm [f(knuth), [e(yacobi) Cohen& Lenstra and many others]. This method could be seen as a generalization of the binary algorithm. Therefore, it is also somewhat similar to the window method. In order to calculate a chain using the m-ary method, first write n in the m-ary number system (for m = 2, this is the binary system) as follows

$n = d_0 \cdot m^t + d_1 \cdot m^{t-1} + ... + d_{\cdot t}$, for $m = 2^k$. Now, add the elements of the chain as follows:

$1, 2, 3, \ldots, m\text{-}2, m\text{-}1$

$2d_0, 4d_0, \ldots, md_0, md_0 + d_1$

$2(md_0 + d_1), 4(md_0 + d_1), \ldots, m(md_0 + d_1), m^2d_0 + md_1 + d_2.$

…

$m^td_0 + m^{t-1}d_1 + \ldots + d_t$

This algorithm will construct chains of a length of $m - 2 + (k + 1)\cdot t$.
The chain can be shortened however, by deleting elements from the first row that do not occur in the range of d_0 to d_t. Also, any elements of layers further down can be removed, if their elements are already present (for instance, it might occur that $2\cdot d_0$ is already present in the

range 1, 2, …, m. In order to detect such prunable steps, one could rewrite the chains to their DAG-equivalent and apply the simplification rules described in chapter 6.1.
The formula m – 2 + (k + 1)·t therefore gives an upper bound on chain length, not a formula for the actual length itself.

As mentioned, the m-ary method is somewhat similar to two methods described before. First of all, the binary method is just a special case of the m-ary method, for m being 2. The similarity to the window method might become clear after a short example. For consistency's sake, let's compute a chain for n = 55, say for m = 4 (2^2).
55 in the 4-ary representation is 313 ($3{\cdot}4^2 + 1{\cdot}4^1 + 3{\cdot}4^0$).
So let's compute:

1, 2, 3,

2·3, 4·3, 4·3+1

2·**(**4·3+1), 4·(4·3+1)+3

Rewriting this to a regular chain yields:

1, 2, 3, 6, 12, 13, 26, 52, 55

There is no need to prune any of the steps, so the final chain length is equal to the upper bound: m – 2 + (k+1) ·t = 4 – 2 + (2 + 1) ·2 = 8

Now, when looking at the binary representation of 55 (110111), we can see how the m-ary method and the window method correspond. One could say, that the m-ary method also uses windows to calculate the final chain. Only this method divides the windows in a very simple and straightforward way: starting at the right end of the binary representation, we cut the number up in chunks of size *k*. In our example with m = 4, the representation would look like $\underline{11}\ \underline{01}\ \underline{11}$. Note, that using this representation, zero's are allowed at the start and end of the window. Now, the final chain can be calculated by starting at the left end of the representation, doubling the last number and adding the values of the window when they are encountered. This second part of the algorithm is exactly equal to the window method.

Adapted window methods
A range of adaptations to the window method has been proposed.
One of the properties that is often subject to change is the aspect of a fixed window size. Some sources give an example for applying the algorithm in which they use a variable window size, without giving an actual algorithm to be used to calculate the window size to be used.

Sauerbrey [c(sauerbrey)] presents an adaptation that introduces a flexible window size. He proposed to use an increasing window, where each subsequent window is of bigger length than the previous one.
Li [l(li)] also proposed an adaptation to the window method. Instead of using a maximum window size, he suggested to use the following algorithm for window allocation:

1. Start at the left of the binary representation
2. Starting at the present bit, take the largest string that reoccurs later on in the representation and set this to be a window.
3. Skip any zeroes until the next 1 is reached.
4. Repeat steps 2 and 3 until the end of the chain is reached.

After the windows have been assigned using this algorithm, the chain is calculated using the same methods as with the regular window method. For n = 55, the windows would be assigned as follows $\underline{11}\ 0\ \underline{11}\ \underline{1}$. It should be noted, that using this method, windows are allowed to have a zero as the last bit.

8.7 Minimal tree methods

So far, we have looked at a diversity of methods that calculate *relatively* short addition chains. When looking at and discussing the power tree method, we got a first hunch of the results of this method. Now, one might wonder if using such a tree based method might be usable to generate minimal chains.

The reason that chains calculated by the power tree method are not minimal per se, is, as we saw, a result of the pruning rules we use. On the other hand, if we would apply no pruning whatsoever, the search space would become prohibitively large. The key to the success of such methods therefore seems to be a pruning algorithm that cuts off as much of the search tree as can be missed, without cutting off too much.

A key concept to this tradeoff is using minimal and maximal values for the minimal chain length. Of course, any branch in the tree that would produce a chain larger than the minimal chain length could be pruned. The lower can be used as a stopping algorithm: as soon as our algorithm has found a chain equal to the lower bound of the minimal chain length, we know that we have actually found a minimal chain and we can stop looking. Now, before going into any further details on the different minimal tree algorithms, we will take a closer look at the lower and upper bound of l(n) in chapter 9.

9 Bounds

In the previous chapter, the need of boundaries for the chain length was already mentioned. In this chapter, we will shortly look at the use of bounds, after which algorithms and formulas are presented to actually calculate both upper and lower bounds.

9.1 The use of bounds

As we saw in chapter 8.5, tree based algorithms can be used to calculate addition chains. After seeing Knuths results on comparing the factor method and the tree based method, it even looks like the results of using a tree based method are quite promising (we will also come back to this in chapter 12).
However, we also saw that using such a tree based method has its downsides: the search space grows explosively if no proper pruning algorithms are used. And that is where bounds are essential.
We define two kinds of bounds: upper bounds, which give a maximum for the minimal chain length l(n), and lower bounds, which, not that surprisingly, give a minimum for l(n). In other words,

Lower bound for n ≤ n ≤ upper bound for n

Though the exact use of the bounds will become clearer when discussing the actual methods using them, we will give a short explanation of it here. Perhaps the following figure (9.1) can help to explain the use of bounds. Please note, that this figure is fictional; it is intended purely as an example and not based on any actual algorithm or chain.

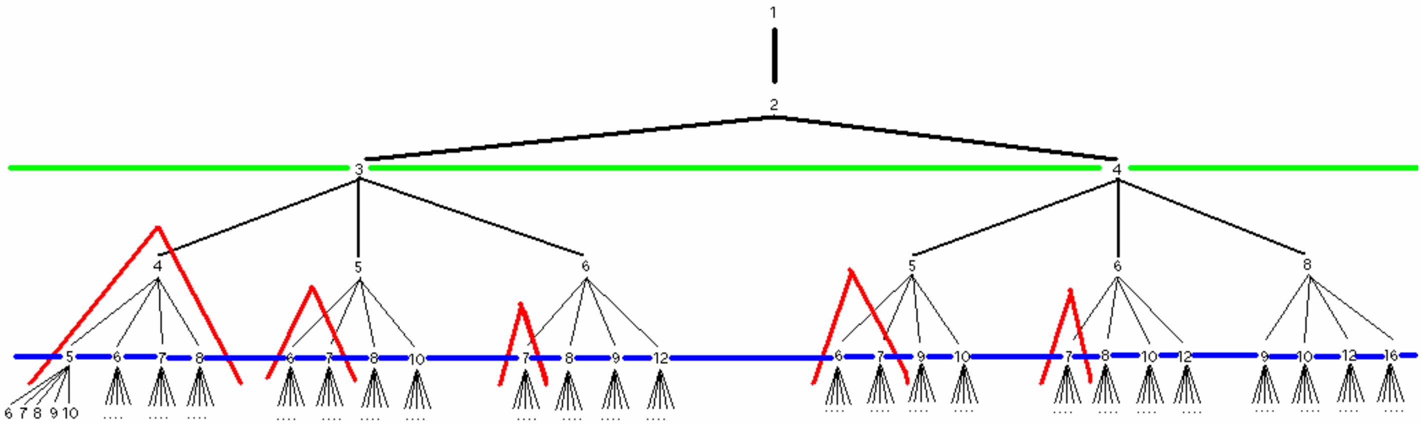


Fig 9.1: bound in a hypothetical search tree.

In figure 9.1, lines of three different colours are used.
The green line represents the lower bond on the chain length. This lower bound can be used as a stopping condition for the tree algorithms. As soon as a chain is found that has a length equal to the lower bound, the algorithm can stop, since then we know for sure that a chain of minimal length has been found.
Apart from that, we can prune any path in the tree that is longer than the upper bound (presented in figure 9.1 by the blue line). Since it is known that the minimal chain length is smaller or equal to the upper bound, no (sub)chain with a length larger than the upper bound will result in a minimal chain.
Apart form these two clear pruning rules, a third, derivate rule can be used to cut large parts from the tree. When for a certain number *n* we have a upper bound *ub = x*, then we know that at layer *x,* the number *n* should be added.
Now, addition chains are strictly monotonic rising and the doubling step is the largest possible step (because we can only use addition steps). Therefore, at layer *x – 1* in the tree, each node should have a value of at least *n/2.* Of course, this process can be repeated, until the root of the tree has been reached.

At each layer *l*, the minimal value of its nodes should be $\lceil mv(l+1)/2 \rceil$, where mv(l) is the minimum value of layer l and the minimum value the upper bound is *n*.
In figure 9.1, the red lines indicate the parts of the tree that can be skipped using the third pruning rule. Time to look at an example, of course for *n = 55*. Let's look at two situations: with an upper bound of 9 and an upper bound of 8.

Position in the chain	**Upper bound of 9**	**Upper bound of 8**
9	55	-
8	28	55
7	14	28
6	7	14
5	4	7
4	2	4
3	1	2
2	1	1
1	1	1
0	1	1

Table 9.1 upper bounds for *n = 55*

One can imagine that improvements to the upper bound calculations, no matter how small they might be, can have a great impact on the total number of nodes that are pruned from the search tree. In case an upper bound of 9 is used, the breadth first search algorithm (discussed in chapter 10) would need to visit 9960 nodes before a chain of length 8 is found. Using a bound of 8 would bring this number of nodes down to 3118. On the other hand, the limits should not be chosen too sharp. *"Of course, the trick when pruning a search tree is not to cut off too much"* as Thurber puts it [o(thurber)]. This would have as result that either non-minimal chains can be found (with a high lower bound) or possibly no chain can be found at all (with a low upper bound).

9.2 Upper bounds

Of course, we could give a first upper bound for *n: n* itself. This bound can be slightly improved to *n – 1,* since the number 1 is present in each chain without using an addition step [a].
However, it will be clear that this bound should be improved further to be useful. What we would like is to find a sharp upper bound, in other words an upper bound that is as close to *l(n)* as possible.

In the previous chapter, we tried to come up with formulas for calculating the length of the chain when using each of the different algorithms. Since these algorithms produce chains that are relatively short, such formulas could be well used as upper bounds.

9.2.1 Binary upper bound

However, as we saw before, only the binary method gives us a nice formula for the chain length, given the goal *n*:
$l(n) \leq \lambda(n) + v(n) - 1$
This formula for the length of the binary chain is generally used as an upper bound [f(knuth), r(bergeron) and many others].
Let's have a look at how this bound performs, following Knuth's theorems [f(knuth)]. In order to do so, we first need to give a first lower bound.
First of all, it is clear that any chain for *n* will require at least $\lceil {}^2log(n) \rceil$steps.
By induction, it is obvious, that step a_i in the chain will have as value at most 2^i:

- we always start with 1 (2^0) at position zero (a_0)

- At each step a_i, the largest possible step is to add a_{i-1} to itself twice. Since we know that the value of a_{i-1} is at most 2^{i-1}, the value of a_i can never be greater than $2 \cdot 2^{i-1} = 2^i$.

Therefore, we know that $^2\log(n) \leq r$, and thus that
$l(n) \geq \lceil {}^2\log(n) \rceil$

Now, we can have a look at how the binary algorithm performs under different circumstances.

- $v(n) = 1$: In case the binary representation of *n* contains only a single one, n will be of the form 2^a. Now, using our upper bound, we know that $l(n) \leq \lambda(n)$ (since $v(n) = 1$). Using our lower bound, we know that l(n) *)* $\geq \lceil {}^2\log(n) \rceil$. Therefore, $l(n) = {}^2\log 2^a = a$
- $v(n) = 2$: In this case, n will be of the form $2^a + 2^b$, for which we can state, without loosing generality, that a>b. Now, both the upper and the lower bound give us the result of $l(n) = a+1$.
- So, until now we have seen that for $v(n) = 1$ and $v(n) = 2$, the binary upper bound is actually equal to l(n). Now, with the values of v(n) increasing, the proofs for the length of l(n) become more difficult. Knuth gives a proof in which he shows that for those chains with $v(n) = 3$, $l(n) = a+2$. So, still, the binary method performs optimally.
- Starting from $v(n) = 4$ however, there are certain cases for which the binary upper bound is larger than l(n). Knuth identifies four groups of numbers for which the minimal chains have a length of $\lambda(n) + 2$, one step shorter than the binary bound. These groups are, for $n = (2^a + 2^b + 2^c + 2^d)$ and a>b>c>d:
    - Case 1: $a - b = c - d$
    - Case 2: $a - b = c - d + 1$
    - Case 3: $a - b = 3$, $c - d = 1$
    - Case 4: $a - b = 5$, $b - c = c - d = 1$

  So, for any number with $v(n) = 4$ matching any of these four cases, $l(n) = \lambda(n) + 2$, whereas the binary upper bound gives a length of $\lambda(n) + v(n) - 1 = \lambda(n) + 3$.

9.2.2 Wattels upper bound

Now, while this binary upper bound is much sharper than the bound of *n - 1* given before, we might be able to find a bound with yet better results.
In his paper, Wattel suggests the following upper bound:
$l(n) \leq {}^2\log(n) + ({}^2\log(n) / k) - k + 2^{k-1} + 1$, where $n \geq 2^{2k}$
The proof of this upper bound is extensive, and I will present here only a rough draft of it. For the exact proof, see [b(wattel)]:

Rewrite n to its binary representation: $a_0a_1\ldots a_r$.
With r being the length of the chain and t = r/k
Now, a chain for *n* is constructed in a way, quite similar to the window method. First, construct windows of length k, starting at the left end of the binary representation. Note that these windows are allowed to start and/or end with 0. Now, build an initial addition chain in the following manner:

Step 1:
1, 10, 11, 101, 111, …, (11…1)
$\Lsh$ length = k
This chain contains all odd numbers between 1 and 2^k, being 2^{k-1} in total.

Step 2: Now, we can add the number $a_0a_1\ldots a_{k-1}0$ to the initial chain, by adding two elements already present in the chain. Now, we'll look at the next window: $a_k, a_{k+1}, \ldots a_{2k-1}$. In case this window contains only bits set to 0, we can use k doubling steps to construct $a_0, a_1, \ldots, a_{2k-1}$. If the window however contains at least one 1-bit (say at position i in the window), then the number $(a_k, a_{k+1}, \ldots, a_i)$ is present in the initial chain (since this chain contains all odd numbers up to 2^k and the binary representation of $a_k = a_i$ is odd since $a_i = 1$). Now we can use i-1 doubling steps, then one step to add $(a_k, a_{k+1}, \ldots, a_i)$ to the chain, and subsequently another (k-i) doubling steps. So, to go from the initial chain to the chain for $a_0a_1\ldots a_{2k-1}$, we need at most k+1 steps.

Step 3: We can repeat step 2 for every window, needing k steps for each window with value 0 and k+1 steps for every other window.

Step 4: Now, let's say we have constructed a chain $a_0, a_1, \ldots, a_{t\cdot k-1}$ in step 3. We can now construct $a_0, a_1, \ldots, a_r$ from this chain by using r-(t·k-1) doubling steps plus at most one additional step. So, in step 4, we need at most r-(t·k)+2 steps.

Now, we can add up the results: $2^{k-1} + (t-1)(k+1) + r-(t\cdot k)+2$

$= 2^{k-1} + r + t - k + 1$

Since t = [r/l] and r = [^{2}log n], we get $l(n) \leq {}^2\log(n) + ({}^2\log(n) / k) - k + 2^{k-1} + 1$: the bound we mentioned before.

This method uses the same tactics as the m-ary method discussed in chapter 8.6. The method and its lower bound are also described by Thurber in [g(bergeron)]

In the following figure, the binary upper bound is compared to Wattels bound for a variety values for k.

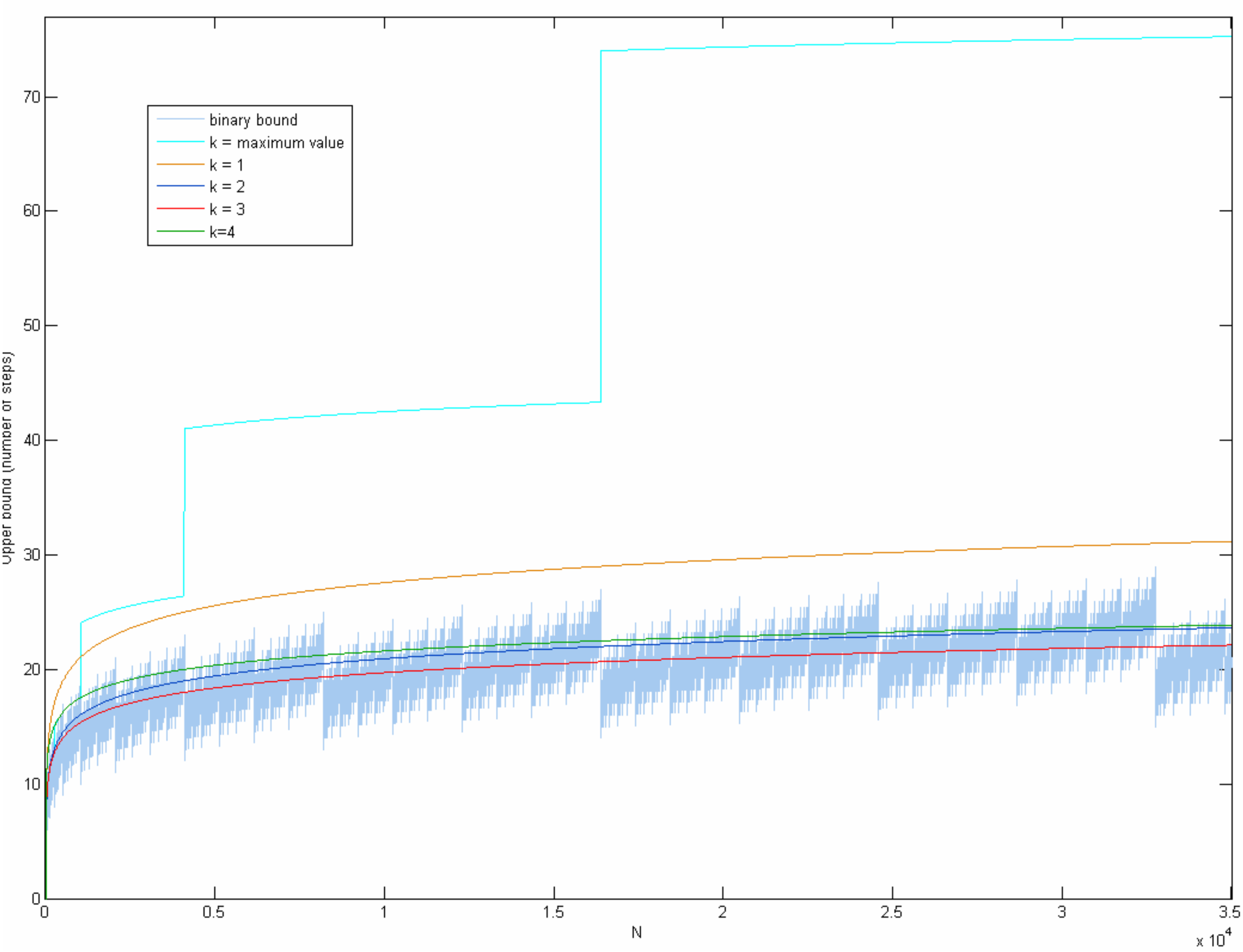


Fig 9.2: comparing the binary upper bound and Wattel's upper bound for a variety of values for k

Figure 9.2 shows that wattels bound is optimal for k = 3 for $2^{2\cdot 3} \leq n \leq 35.000$.

9.2.3 Recursive bounds

So far, we have only looked at upper bounds as a formula for the goal *n*. There are two other ways to go, though.
The first of these is calculating the upper bound based on other bounds.
A set of rules which can be used to calculate the bound in such a way is [l(bleichenbacher)]:

1. $l(n+1)\leq l(n) +1$
2. $L(n+2) \leq l(n)+1$ (if $n \geq 2$)
3. $L(mn) \leq l(m) + l(n)$
4. $L(mn +k) \leq l(\{k,n\}) + l(m) + 1$

Now, the question remains when to use which rule. [l(bleichenbacher)] suggests to use them as follows:

1. Used as a first bound, for $n \geq 2$
2. Used as a first bound, for $n \geq 2$
3. Used as a bound up to i (for example 100), depending on i divides n. If i is divisor of n, then we have $l(n) \leq l(i) + l(n/i))$
4. Again used as a bound up to i, depending on whether i divides n. If i is divisor of n, then we have $l(n) \leq l(i) + l(n/i)$, otherwise $l(n) \leq l(\{i, n \bmod i\})+ l(\lfloor n/i \rfloor) + 1)$

Note, that in this fourth step, an *addition sequence* is used in the result.

Let's look at benefits and downsides of this method for calculating upper bounds. On the negative side, using a simple formula for a given *n* (as with the binary upper bound) is easier to use and implement. Using relative rules such as suggested by Bleichenbacher introduces moments of choice: sometimes multiple rules can be applied. In order to find the smallest upper bound, we need to apply the different rules recursively, introducing a wider search space when searching for a bound. However, the bounds were introduced to limit the search space when calculating addition chains!
On the positive side however, applying these rules can be beneficial. Since even small improvements of the bounds can have quite an effect on the effectiveness of the minimal addition chain algorithms, going through some effort to improve the bound can be worthwhile.

9.2.4 Approach algorithm's bounds

Keeping this last remark in mind, it is time to look at the third possibility for calculating bounds. Looking at the third and fourth rule suggested by Bleichenbacher, similarities with the factor chain method can be seen. Now, while only the binary method has a simple and ready-to-use formula, this does not mean that the other methods are useless when it comes to generating bounds.
In order to calculate short bounds to be used for the minimal chain algorithms, one could first use each of the approach-algorithms. The minimal value for the chain length returned by these algorithms can then be used as an upper bound. While running each of the algorithms for each *n* will take some time, the result might be that the minimal chain algorithms will need less time and memory to complete.
At this point we can also use the data on the performance of the binary algorithm. We know that the binary bound algorithm returns the minimal chain length for $v(n) = 3$ and for certain cases of $v(n) = 4$. For these cases, it would be futile to run any of the other algorithms to try and decrease the upper bound. For any other case however, it might just be worthwhile.

9.2.5 Open problems

There are some still open problems concerning minimal chain length. In chapter 4, the Scholz-Brauer conjecture was already mentioned:

$l(2^n-1) \leq n - 1 + l(n)$

No proof for this conjecture has been found yet. As mentioned in chapter 4 though, the formula has been found to hold for some *n.* For star chain, the conjecture has been proven by Brauer in 1939:

$l^*(2^n-1) \leq n - 1 + l^*(n)$

The conjecture has also been proven for the l^0-chains, mentioned in chapter 5 [f(knuth)]

Since the general Scholz-Brauer problem is still unsolved, we will not use this conjecture in our program.

Apart from the bounds for regular addition chains, some sources also present bounds for related problems. In [c(sauerbrey)] for example, we can find an upper bound for addition sequences:

$l(n_0, n_1, \ldots, n_k) \leq \log(n_k) + c \sum_{i=0}^{k} ((\log n_i) / (\log \log (n_i + 2))$, with c being a constant.

However, we will not go into details about addition sequences and vector addition chains in this paper.

9.3 Lower bounds

Just as with the upper bounds, a couple of lower bounds can be given. However, the diversity of lower bounds is somewhat smaller. Let's first look at some first, very weak, lower bounds. Of course, for any $n \geq 1$, we can create an addition chain of at least 1 step. However, it will be clear that this bound is no good whatsoever.
When discussing the upper bounds, another, better, lower bound was introduced:
$\lceil {}^2\log(n) \rceil$
While this bound is clearly better than the simple bound of 1, we can try to come up with yet a better bound. Also with this bound, however, one has to be careful not to give a lower bound that is too sharp. A lower bound which lb(n) for which *lb(n) > l(n)* for certain *n* is useless.
An often mentioned lower bound is [c(sauerbrey), g(bergeron), y(flammenkamp), and others]:

${}^2\log (n) + {}^2\log(v(n)) - 2.13 \leq l(n)$

This bound has been proven by Schönhage in 1974. It follows the general lower bound $l(n) \geq {}^2\log(n) + 2\log(v(n))$ (f[Knuth]). However, for this conjecture, no actual proof has been found so far. Schönhage was able to find a proof for the weakened version. Since ${}^2\log (n) + {}^2\log(v(n)) - 2.13$ will often result in fractioned numbers, and the number of steps in an addition chain is always a round number, we can slightly improve this bound to

$\lceil {}^2\log (n) + {}^2\log(v(n)) - 2.13 \rceil \leq l(n)$

In the following figure, both the lower bound and the upper (binary) bound have been plotted. The minimal length of the addition chains lies somewhere in between.

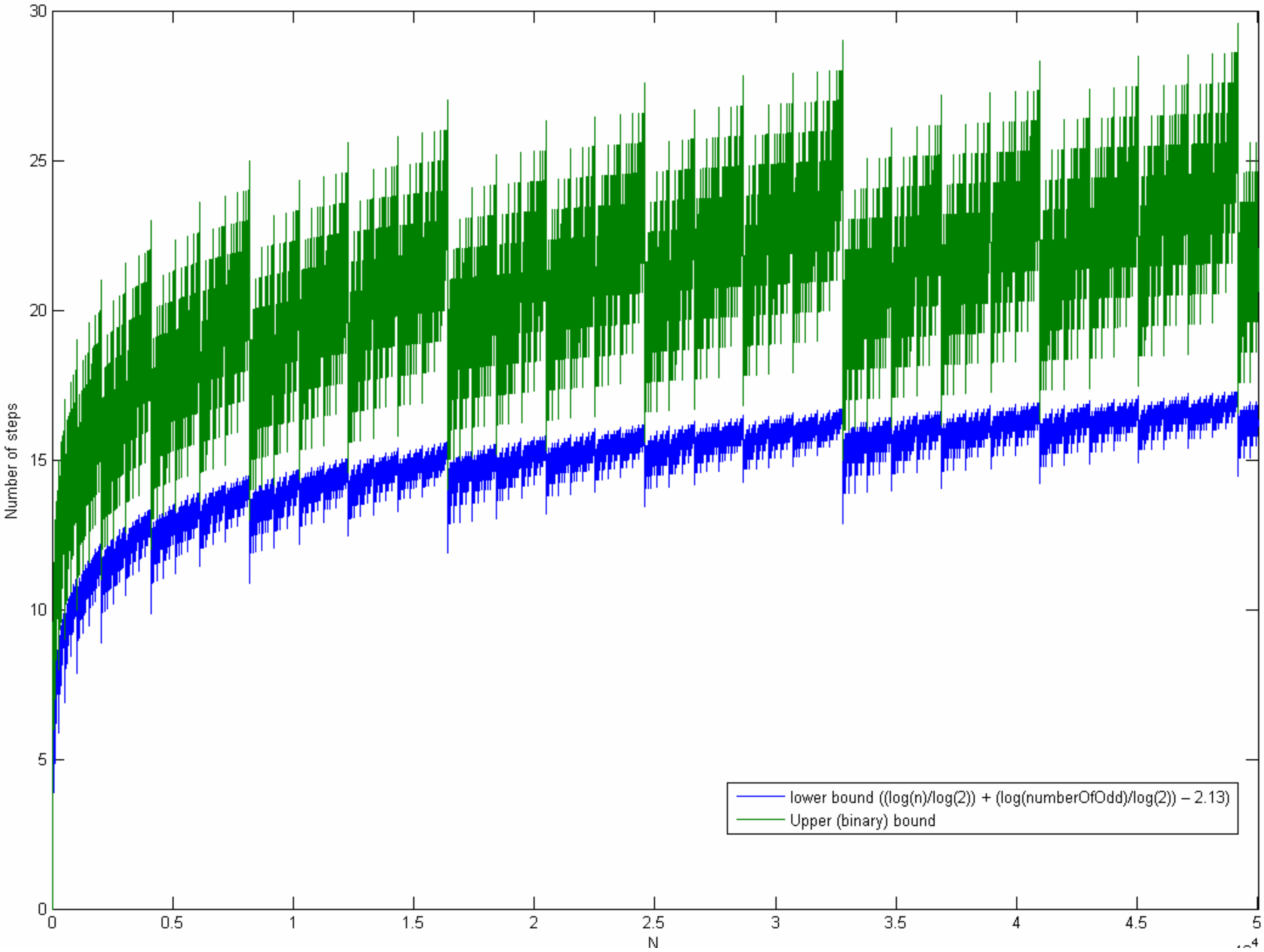


Fig 9.3: Upper and lower bound plotted together

10 Minimal algorithms

In this chapter, three different minimal algorithms will be described in detail. Each of these algorithms calculates l(n), the minimal length of an addition chain. All use a tree-based approach, each in a slightly different way. These three algorithms are also present in the computer program.
Apart from that, a couple of other tree based minimal algorithms will be mentioned. These, however, are not included in the computer program and will be described only briefly here.

10.1 Breadth first search

When discussing the power tree method in chapter 8.5, we noted that this algorithm does produce relatively short chains, but not minimal ones. The reason for this is two-fold. First of all, the pruning rules that are used to limit the total search space are rather crude: the algorithm simply adds numbers to the tree only once. Without using this pruning rule, we get the full-blown power tree method. This method however does not produce minimal chains as well, since it only takes star steps into account.
So, for a tree based algorithm to create minimal chains, we would need to use the full-blown power tree method, adapted in such a way that it considers all possible steps.
Generally, this can be done as follows:
Full-blown breadth first search algorithm: up to a node with label *n*:

```
1. Add a node with value 1 as root of the tree
2. Set the current layer l to 1
3. While the goal n has not been found:
        a. While there are still nodes to add to the current layer
                i. Find the leftmost node at layer l
                ii. For i = layer l up to 1
                        a. For j = i up to 1
                                1. Get the i-th and j-th (grand)parent of the current node
                                2. Store the value to add (the sum of the values of its i-th and j-th
                                   parent) at v.
                                3. if v is bigger than the value of the current node
                                   AND v is not bigger than n AND the current node does not
                                   already have a child with value v,
                                        i. add v as child to the current node
                iii. Get the next node at the current layer
        b. increase l by 1.
```

Let’s see an example for *n = 16:*

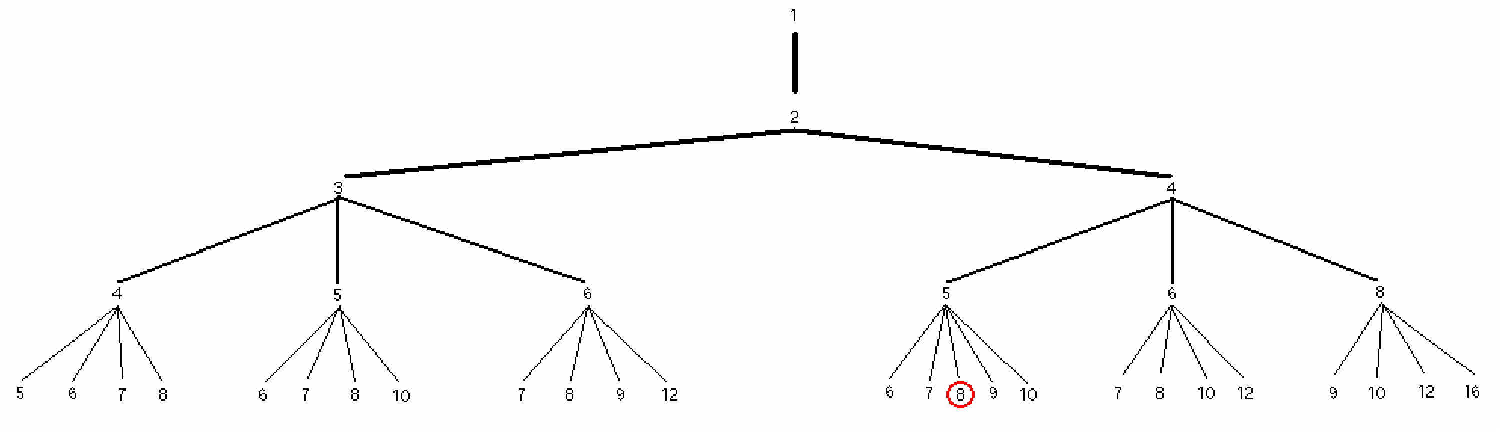


Fig 10.1: The breadth first search algorithm for *n = 16*

Notice the red-circled node with value 8 in the tree. When comparing the tree of figure 10.1 with that of the full blown power tree method (fig 8.5, chapter 8.5), one notices that this circled number is not present in the latter. The encircled node with value 8 in figure 10.1 is the result of a non-star step, and therefore it is absent in the full blown power tree.

So far, so good. We now have an algorithm which produces minimal chains. And that is all the good news for this algorithm. It suffers from the same downside as the full blown power tree method: a vast search space. As a matter of fact, things are even worse for this algorithm, since it considers all possible chains instead of ‘only’ star chains. Clearly, we need some pruning rules here.

Therefore, let’s adapt our algorithm as follows:

Breadth first search algorithm: up to node with number x

```
1. Add a node with value 1 as root of the tree
2. Set the current layer l to 1
3. Calculate the upper bound on the chain length and store this value in m
4. While the goal n has not been found:
    a. t = n
    b. for m down to l
        i. t = $\lceil t/2 \rceil$
    c. While there are still nodes to add to the current layer
        i. Find the leftmost node at layer l
        ii. For i = layer l up to 1
            a. For j = i up to 1
                1. Get the i-th and j-th (grand)parent of the current node
                2. Store the value to add (the sum of the values of its i-th and j-th
                   parent) at v.
                3. if v is bigger than or equal to t AND v is not bigger than n AND
                   the current node does not already have a child with value v,
                       i. add v as its child
        iii. Get the next node at the current layer
    b. increase l by 1.
```

At the third step of this algorithm, the upper bound on the chain length is calculated. One can choose to use either of the tactics described in the previous chapter. In the computer program accompanying this paper, the upper bound is calculated as the minimum of the binary upper bound, wattel’s bound, the factor method and the window method for a range of windows.

Let’s have a look at how this algorithm performs. First, let’s revisit the example of *n = 16*. The maximum length for this number (when calculated by the binary bound), is 4.

The following figure shows the resulting graph. At the right side of this graph, the value for t is displayed at each layer.

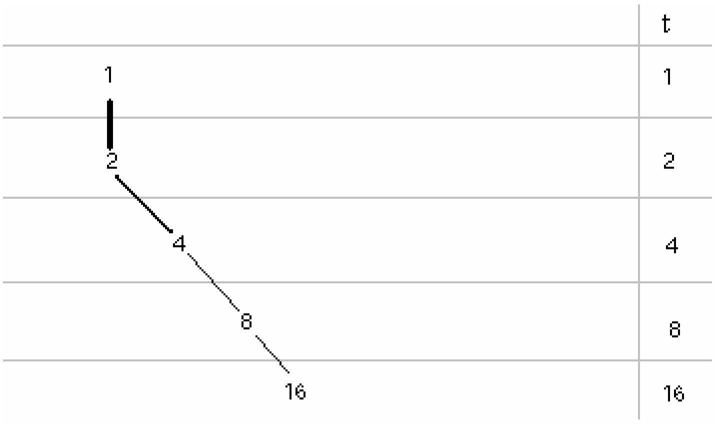


Fig 10.2: breadth first minimal search tree for *n = 16*

When comparing this result with that of figure 10.1, the effect of using proper pruning rules becomes apparent immediately. Without pruning, 35 nodes were added to the tree in total. With the use of the pruning rules, only 5 nodes were needed.
Of course, this is somewhat of a special case, with 16 being a power of 2. Let's look at yet another example, for *n = 15* (clearly, we will not use our regular example of n=55, since this would require us to build a tree with over 3000 nodes, even when using an optimal upper bound of 8). The binary upper bound algorithm would give a bound of 6, for which the breadth first search algorithm would create a tree like figure 9.3 below:

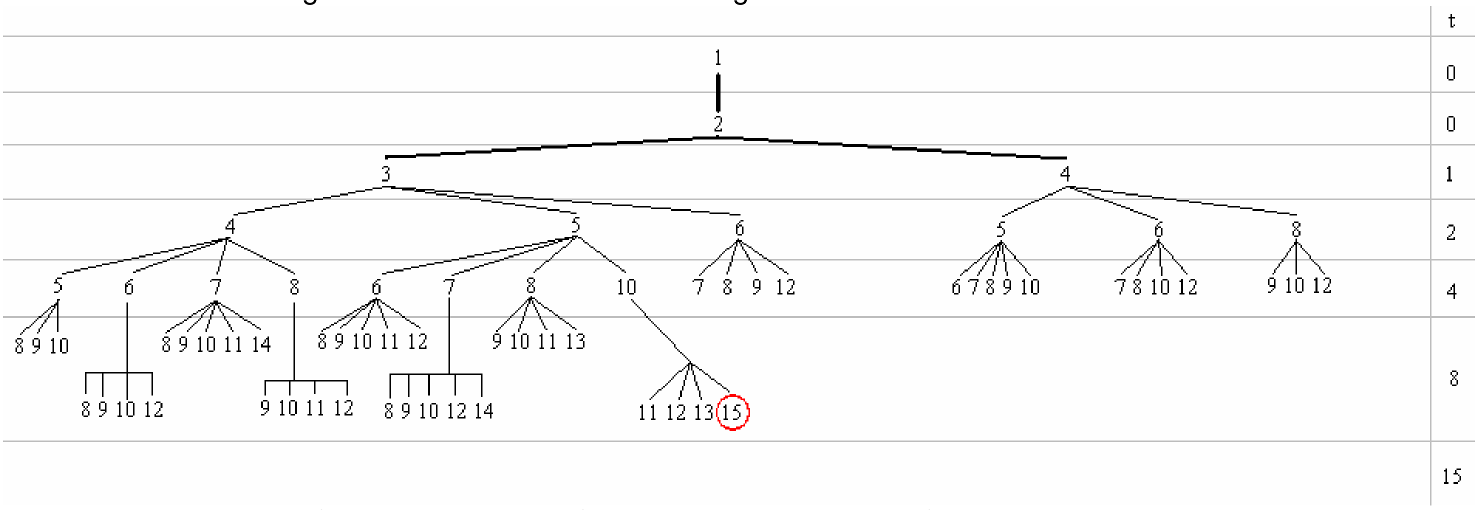


Fig 10.3: breadth first minimal search tree for *n = 15* with an upper bound of 6

In this figure, a total of 68 nodes is added.
However, with a bound of 5 (given by for instance the factor algorithm), the tree would look like this:

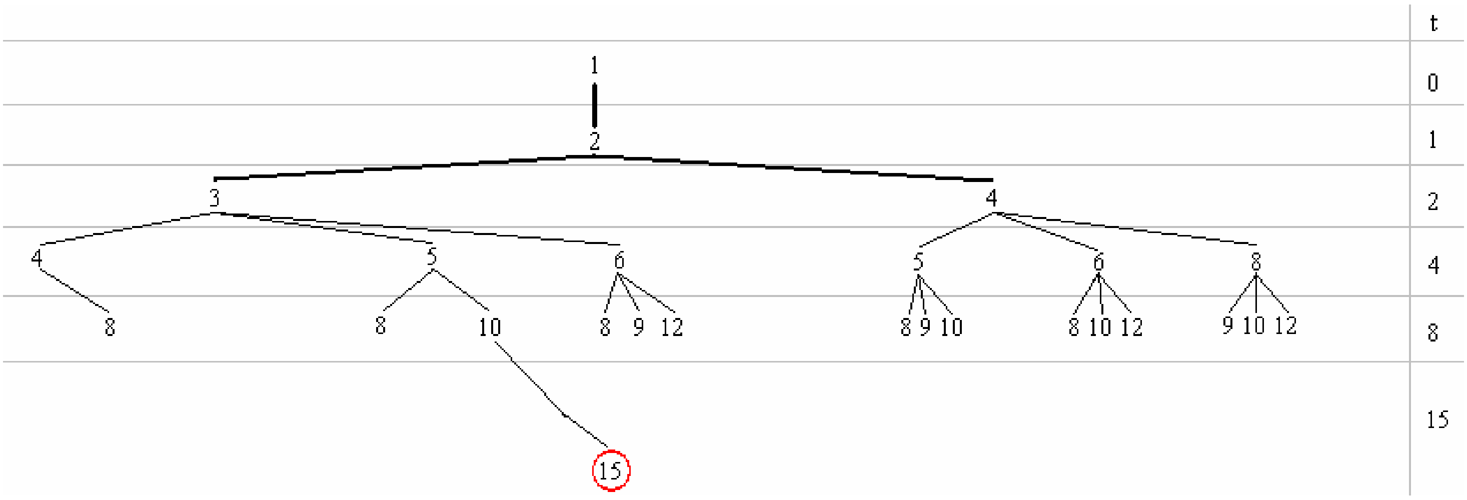


Fig 10.4: breadth first minimal search tree for *n = 15* with an upper bound of 5

Using this sharper bound reduces the number of nodes visited to 26. So, even for such small numbers and an upper-bound improvement of only one, almost two third of the nodes can be pruned from the tree, without loosing its minimal properties. Not surprisingly, for both bounds the resulting chain length is equal.

10.2 Depth first search

A second method to calculate minimal chains is to use a depth first search algorithm. Instead of calculating the tree layer for layer, one can build it branch by branch.
In short, the algorithm boils down to this:
Build the tree in a depth first way, with the upper bound as a maximum depth. If a node with the goal *n* is added, check to see whether the added node has a depth equal to the lower bound. If this is true, then the algorithm can finish: a minimal chain has been found. If not, then set the maximum depth to the layer of the added child and continue the search.

There are both benefits and downsides to this method. One of the possible benefits is that the depth first algorithm can terminate as soon as it has found a chain with length equal to the lower bound. Another benefit is that is capable of updating the upper bound along the way: as soon as a chain with length bigger than the lower bound but smaller than the upper bound is found, the upper bound can be renewed. On the downside however, the algorithm sometimes also needs more nodes than the breadth first algorithm. In those cases where the lower bound is smaller than the actual minimal chain, nothing can be gained from using this lower bound. In fact, in this case the depth first search method will need to continue to search the entire tree, in order to make sure that no chain shorter than the one found exists. It can happen that multiple minimal chains are found and therefore multiple nodes with value *n* need to be added.
Let's take a more detailed look at the algorithm.

Depth first search algorithm: up to node with number x

```
1. Add a node with value 1 as root of the tree, and set this node as the current node t.
2. Set the current layer l to 1
3. Calculate the maximum chain length and store this value in m
4. Calculate the minimum chain length and store this value in p
5. While no node for the goal n has been added at layer p of the tree:
        a. While no node for goal n has been added to the tree
                i. For i = layer l up to 1
                        a. For j = i up to 1
                                1. Get the i-th and j-th (grand)parent of the current node
                                2. Store the value to add (the sum of the values of its i-th and j-th
                                   parent) at v.
```

```
                    3. if v is bigger than or equal to t AND v is not bigger than n AND
                       the current node does not already have a child with value v
                           i. add v as its child
        ii. If any node was added
                a. Get the left-most child and set this as the current node t.
                b. If the current node is at a depth equal to the upper bound
                    1. Get the parent of the current node and set it as the current
                       node t.
                    2. While the current node has a parent and the current node is
                       the rightmost child of its parent:
                           i. Get the parent of the current node and set it as the
                              current node t.
                    3. If the current node is not the root of the tree
                           i. Get its first sibling to the right of the current node and
                              set it as the current node t
        iii. Else: (no node was added)
                a. If the current node t is not the root and the current node is the
                   rightmost child of its parent:
.                   1. Get the parent of the current node and set it as the current
                       node t.
                    2. While the current node t has a parent and the current node t is
                       the rightmost child of its parent:
                           i. Get the parent of the current node t and set it as the
                              current node t.
                    3. If the current node t is not the root of the tree
                           i. Get its first sibling to the right of the current node t and
                              set it as the current node t
                b. Else
                    1. Get the next sibling to the right of the current node t and set it
                       as the current node t
```

Applying this algorithm for *n = 16* would generate the same result as the breadth first algorithm. For *n = 15* and an upper bound of 6, the depth first search algorithm adds 83 nodes in total. For an upper bound of 5, it needs 28 nodes. The figures are not included here, but left for the reader as an exercise. Note that the algorithm adds all the children for a certain node before moving further down the tree. The first few steps of the tree would therefore look like this:

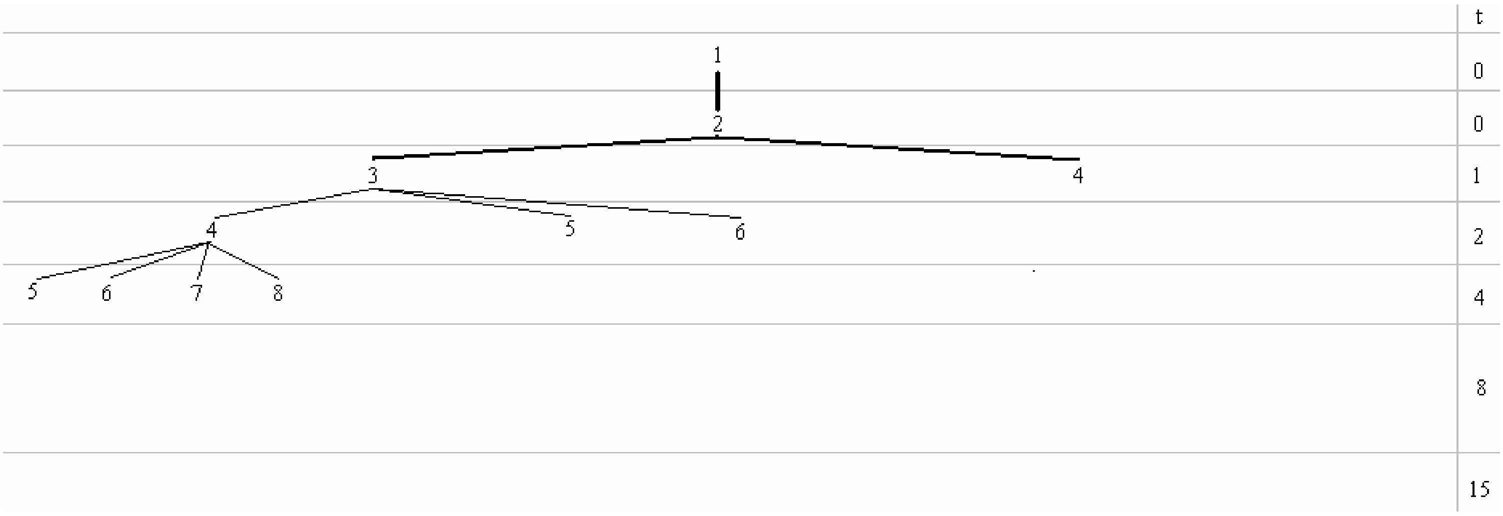


Fig 10.5: first few steps for depth first search algorithm for n = 15 and upper bound of 6

10.3 Combined method

So far, we have looked at two different methods to calculate minimal addition chains. In this section, we will look at a possible way of combining these two methods.
This combined method will first calculate the upper bound ub and lower bound lb. It then uses the depth first method up to a depth of lb. If it does not find a chain for *n*, it switches to the breadth first search method, using the same tree.
Since this method basically just uses the two algorithms described before, there is no need to repeat a more detailed plan of this algorithm at this point. More on this algorithm can be found in chapter 12 (results).

10.4 Other methods

10.4.1 Top down method

There are yet some other algorithms that have not been included in the computer program. One of these is a top-down method. In the top-down method, a tree is created with a root of *n* instead of 1. Now, all possible numbers from which this root can be reached should be added as its children. Basically, these are all the numbers in the range $\lceil n/2 \rceil$ to n-1. Now, it sounds simple enough to repeat this step for all the children until 1 has been reached.

However, this is not enough, since such an algorithm does not guarantee that a chain is built that obeys all the rules for addition chains. For instance, let's look at an example for *n = 7*.

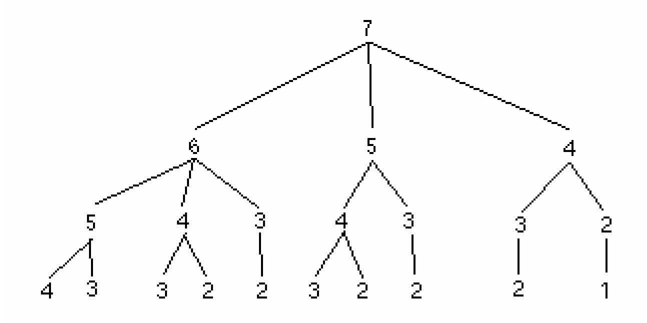


Fig 10.6: a top down algorithm which fails to produce a proper chain

Now, assuming that the algorithm is used in a breadth first manner, we see that the following chain is constructed:
1 2 4 7
Clearly, this is not a proper addition chain. In order to actually use a top down algorithm, we need to adapt the tree algorithm. Instead of adding all numbers in the range $\lceil n/2 \rceil$ to n-1 as children of node n, we need to add sets of numbers, which also contain the resting value. For the case of n = 7, the result of such an algorithm is given in figure 10.7:

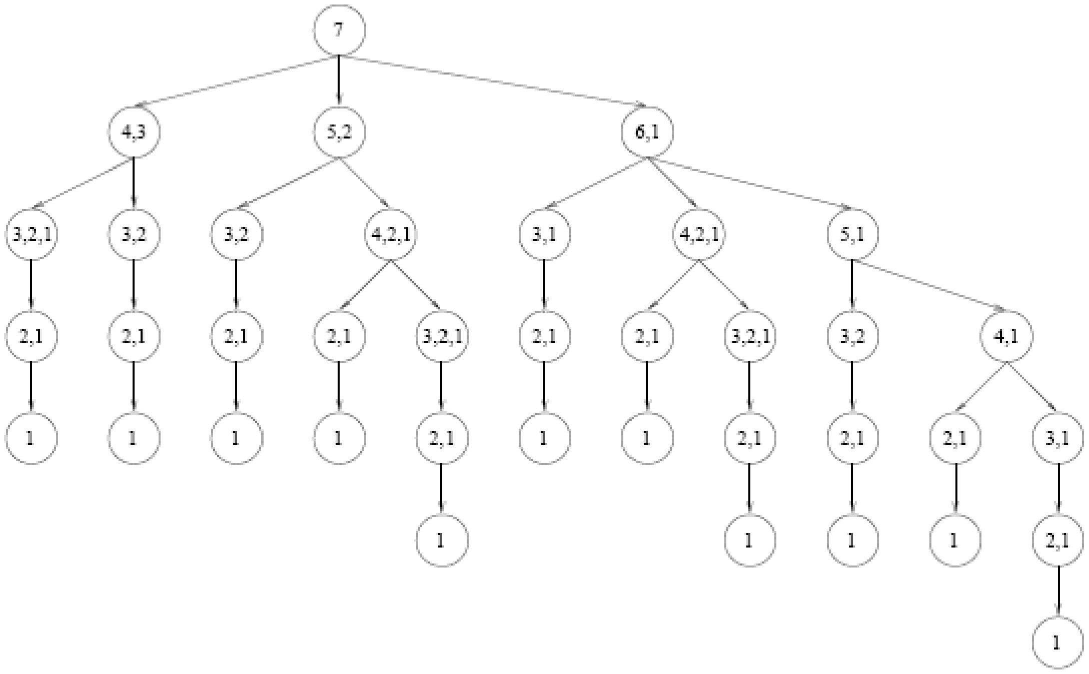


Fig 10.7 Top down search tree for *n = 7,* taken from l[(bleichenbacher)]

This method can also benefit greatly from pruning rules in order to limit the search space. Bleichenbacher also suggest some other improvements. First of all, by merging those nodes which contain the same set, the total number of nodes can be brought back by a great amount. In order to even increase the amount of overlap between the different nodes, one can add the numbers 1 and 2 to each set. Since every chain for $n \geq 2$ contains these two numbers, this can be done without affecting the chain length. It improves the efficiency of this algorithm, since now more nodes contain the same numbers. Looking at the example in figure 10.7 for instance, we see several nodes containing the set {3,2} and some containing the set {3,2,1}. If 1 and 2 are added to each set however, these nodes will contain the same set and can therefore be merged. The resulting figure is as follows:

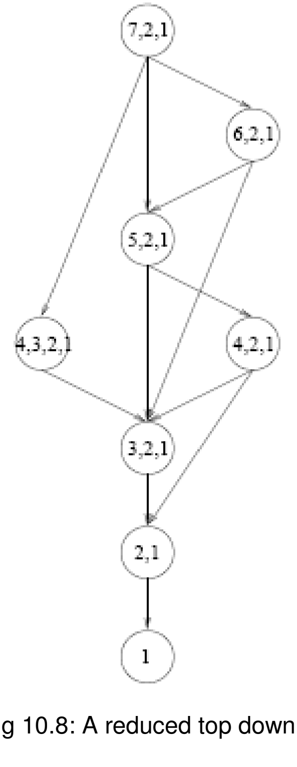


Fig 10.8: A reduced top down search tree for *n = (7,2,1).* Figure taken from L(bleichenbacher)

The top down method can be applied both in a depth first and breadth first oriented way.

10.4.2 Adaptations to breadth first and depth first search:

Both the breadth first and depth first method as described before add nodes in an increasing order: starting at the smallest value to add, they work their way upwards. An alternative is to add them in decreasing order, just as described in chapter 8 for the power tree method. In chapter 12, we will have a look at the results of using the reversed version of the depth first search method.
Also, the top down method can de altered in such a way that largest elements are considered first instead of smallest.

11. The program

In the program accompanying this paper, a range of different methods can be used to calculate addition chains. All these methods have been presented in this paper. The algorithms used are:

1. Binary method
2. Doubling method
3. Factor method
4. Window method
5. Power tree method
6. Breadth first method
7. Combined method
8. Depth first method

In the next chapter, the results of the different methods are displayed. The graphs displayed in that chapter are created using MATLAB.

11.1 About the program

The computer program has been programmed in java. The reason for this is that I know this language best and learning another programming language would fall outside the scope of this project. However, as will become apparent in the next chapter, this choice comes with some major downside. Some of the algorithm presented in this paper, mainly the minimal algorithms, require quite an amount of memory. And, as I found out when working on this project, the memory capacities of java were not always enough to cope with this. The end result is that the program is not capable of computing chains for all *n* for all methods. However, this downside only became apparent to me after working on the program for quite a while. Reprogramming the code into another language was hardly an option, since this would have required too much time. Apart from that, the focus in this project has been mainly on the correct working and understanding of the different algorithms, not as much on the performance per se. By using a familiar programming language, it was easiest to generate algorithms which worked correctly.

During the coding, yet other design decisions have been made.
The first of these decisions is on which methods to include in the program. I have chosen to include the five ‘approach-algorithms’ (the binary, doubling, factor, window and power tree method), since these are the methods mentioned most often. At first, I planned to add yet another method, using a new method, to be designed by myself. However, I was unable to find a new method which would produce results that are comparable to any of the others on the one hand, but would also be different enough on the other. Therefore, I chose to construct three minimal methods, each using a modification of a basic tree based method.
One other method that is often mentioned in other papers is the m-ary method, also discussed in chapter 8.6. I have chosen not to include this method, first of all out of timing constraints. Besides that, the method uses an approach quite similar to the window method, with the window allocation as the main difference between these two algorithms.

11.2 About the power Tree method

When working with the power tree method, one can either choose to calculate the power tree up to a layer *l* or up to a specific goal *n*. The benefit of calculating for layers is that you get the result for each number present in the tree up to the specified layer. This saves the time of having to run the algorithm for each of these numbers separately. On the other hand, if you choose to calculate the tree up to a specific number, additional parts of the search tree can be pruned (using similar methods as the minimal algorithms discussed in chapter 10). On the downside, we do have to compute the tree from scratch for each number, increasing the time needed by the algorithm. This decision between the layer-based and goal based power tree method therefore comes down to choosing between time and memory. In this program, I have chosen to go for the layer-based approach. As a result, the program does run out of memory for larger layers. However, considering the fact that this method is an approach algorithm, noting the timing constraints and for the purpose of this paper, it deemed to be more important to generate the results within a reasonable time frame than to generate data using less memory.

11.3 On the minimal algorithms:

When calculating the upper bound used for the minimal algorithms, both theoretical and practical bounds are used. First, the upper bound is calculated using the binary method. If it is known that this bound is minimal (see chapter 10), this value is used as an upper bound. Otherwise, the factor and window method (for window values of 3, 5 and 7) are used to calculate the upper bound, as well as the wattel bound. The minimum value of each of these results is then used as an upper bound. As we saw in the previous chapter, the wattel bound is performs best for relatively low values of k. However, when using the program to generate the results presented in chapter 12, I used the maximum possible value for k (so 2^{2k} is as close to *n* as possible). At the time of programming, I mistakenly assumed this to be the optimal value for k.

A practical upper bound is calculated using the factor and window methods. Of course, using the binary method to calculate an upper bound would be useless, since this would yield the exact same result as the binary bound algorithm. The fact that the power tree algorithm was not used to calculate the upper bound is a direct consequence of the fact that this algorithm has been implemented in a layer oriented manner. This prohibits us from generating a maximum chain length by simply applying the power tree algorithm for single *n*.

12 Results

In this chapter, the results of using the computer program created for this paper are given. This chapter is divided in two parts; one in which the results of the approach algorithms are given and compared and one in which the minimal algorithms are compared.
Note, that in order to generate these results, the program has been adapted in a minor way: instead of giving the actual chain as output, now the chain length is given.
The code for the program itself can be found in appendix III.

To generate the results, each of the ‘approach algorithms’ was run for the numbers 1 to 1.000.000.
The power tree method has been run for the first 25 layers, to make sure that it would generate data for most of the numbers in the range of 1 to 1.000.000. As will be shown later-on in this chapter however, the power tree method failed to generate a chain for 166 numbers in this range, due to memory constraints.
The other algorithms, giving minimal chains, have been used to generate data for the numbers 1 through 750. This relatively small number has been chosen considering the amount of time consumed to generate results using these methods, as well as the amount of memory needed. For these methods though, data has not been generated for all numbers in the requested range as well, due to the memory limitations of the Java Virtual Machine.

For these results we have created several graphs, using MATLAB. These graphs are included in this chapter.
Please note that while the results gathered for these specific parameters do give a hunch of the general properties of each of the methods, the exact results are specific for the test data that is used. The results are intended to provide insight in the general effectiveness of the different methods, and not to present and discuss the absolute facts and numbers.

12.1 Approach algorithms

Let’s start with giving a table for the total number of steps needed for each of the approach methods for 1 ≤ n ≤ 1.000.000.

Method	**Total number of steps** for 1 ≤ n ≤ 1.000.000
Binary method	26836444
Factor method	25313202
Window method (w = 2)	24667443
Window method (w = 3)	24237567
Window method (w = 4)	24222477
Window method (w = 5)	24238943
Window method (w = 7)	24324853
Window method (w = 11)	24667443

Table 12.1: Total number of steps for the different approach algorithms for 1 ≤ n ≤ 1.000.000

12.1.1 Comparing the binary and doubling method

As could be expected (see chapter 8.2), the binary and doubling method yield chains of exact equal length. The actual chains are different, but since here, we are interested in the length of the chains, that is irrelevant. Therefore we will not look at the doubling method in the rest of this chapter. But note that any result generated for the binary method will hold as well for the doubling method.

12.1.2 Comparing the binary and the factor method

Let us next compare the binary and the factor method, when looking at all the chain lengths for 1 ≤ *n* ≤ 1.000.000:

The binary and the factor method perform equally well 149614 times.
The binary method beats the factor method 161989 times.
The factor method beats the binary method 688397 times.

Please note that one method 'beats' another if it needs less steps to reach the goal *n*.

The first 10 *n* for which:	
the binary method beats the factor method:	*the factor method beats the binary method:*
33	15
49	27
65	30
66	31
67	39
69	45
98	51
129	54
130	55
131	60

Table 12.2: Comparing the binary and factor method

For the first million *n,* the binary method uses 26836444 steps.
The average chain length for the binary method is therefore 26.84

The total chain length for the factor method is 25313202,
giving an average chain length of 25.31

This means that for this test data, on average the factor method creates chains that are 1.06 times shorter than the binary chains.

12.1.3 Finding the optimal window values

Now, let's have a look at the window method. Before we can compare this method to any of the other algorithms used, we first need to figure out for which window value this algorithm performs best.
In order to do so, we've run the program, using a range of different window values, namely *w* = 1, 2, 3, 4, 5, 7 and 11.

Note that, as was mentioned before in chapter 8.5, running the window algorithm with a window value of 1 yields the exact same results as the binary method.
When comparing the results for the different window values, we use the results of *w = 1* as a basis for our comparisons.
As a shorthand, we will used the term 'window*x*-method' for 'the window method with a window value of *x'*
-For window w = 2:

The window1 and the window2 method perform equally well 111555 times.
The window2 method beats the window1 (binary) method 888445 times.
The window1 method beats the window2 method 0 times.

The first 10 n for which the window2 method beats the window1 method:
15
27
30
31
47
51
54
55
59
60

Table 12.3: Comparing the window1 and window2 method

- *For window w = 3*

  The window1 and the window3 method perform equally well 75958 times.
  The window3 method beats the window1 method 916965 times.
  The window1 method beats the window3 method 7077 times.

The first 10 n for which the window3 method beats the window1 method:
23
27
31
43
45
46
47
51
54
59

Table 12.4: Comparing the window1 and window3 method

- *For window w = 4*

  The window1 and the window4 method perform equally well 72230 times.
  The window4 method beats the window1 method 905495 times.
  The window1 method beats the window4 method 22275 times.

The first 10 n for which the window4 method beats the window1 method:
39
43
47
51
55
59
63
75
77
78

Table 12.5: Comparing the window1 and window4 method

- *For window w = 5*

  The window1 and the window5 method perform equally well 69870 times.
  The window5 method beats the window1 method 883364 times.
  The window1 method beats the window5 method 46766 times.

The first 10 n for which the window5 method beats the window1 method:
75
79
83
87
91
95
99
103
107
111

Table 12.6: Comparing the window1 and window5 method

- *For window w = 7*

The window1 and the window7 method perform equally well 71022 times.
The window7 method beats the window1 method 856405 times.
The window1 method beats the window7 method 72573 times.

The first 10 n for which the window7 method beats the window1 method:
291
295
299
303
307
311
315
319
323
327

Table 12.7: Comparing the window1 and window7 method

- *For window w = 11*

The window1 and the window11 method perform equally well 122137 times.
The window11 method beats the window1 method 853189 times.
The window1 method beats the window11 method 24638 times.

The first 10 n for which the window11 method beats the window1 method:
4611
4615
4619
4623
4627
4631
4635
4639
4643
4647

Table 12.8: Comparing the window1 and window11 method

The total chain length of the window1 method is: 26836444.
The average chain length for the window1 method is: 26.83.
Note that this corresponds to the results for the binary method, as expected.

The total chain length of the window2 method is: 24667443
The average chain length for the window2 method is: 24.667443

The total chain length of the window3 method is: 24237567
The average chain length for the window3 method is: 24.237567

The total chain length of the window4 method is: 24222477
The average chain length for the window4 method is: 24.222477

The total chain length of the window5 method is: 24238943
The average chain length for the window5 method is: 24.238943

The total chain length of the window7 method is: 24324853
The average chain length for the window7 method is: 24.324853

The total chain length of the window11 method is: 24667443
The average chain length for the window11 method is: 24.501371

These results can be visualized in graphs. Let’s first look at the graphs for 1 ≤ n ≤ 750 (figure 12.1)
Note that in this figure, also a line is added with the optimal chain length. More information on this can be found later on in this chapter. As a note: the graph has been cut of at layer 6, so that those chains *c* with l(c) < 6 is not shown. Only a few nodes for the first couple of *n* can be reached in fewer steps. By decreasing the range of steps in the graph, the results for the steps that are plotted can be seen better.

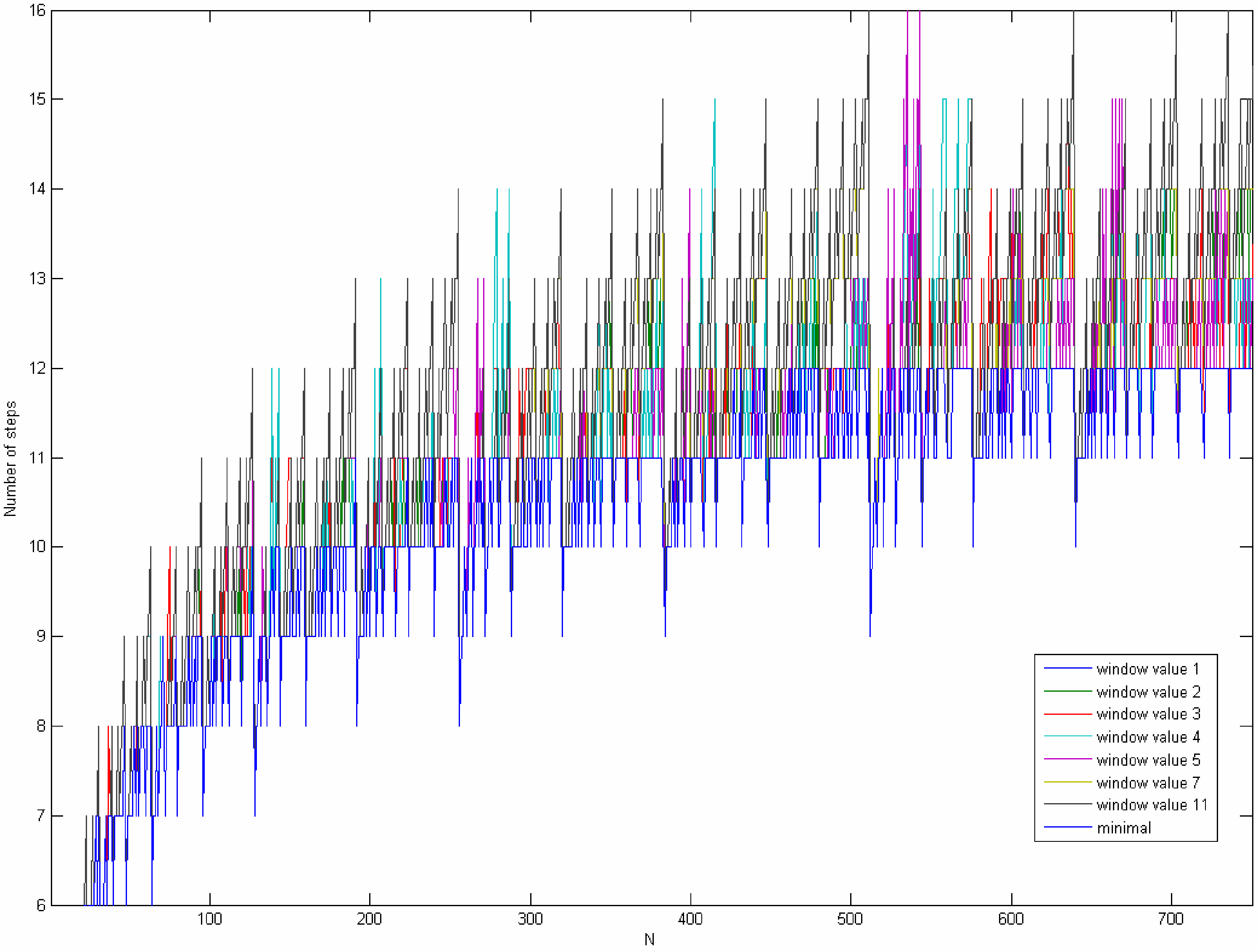


Fig 12.1: The varying window methods and minimal tree length for 1 ≤ n ≤ 750

In the following picture, the values for the chain length are displayed for the varying window lengths, starting from *n = 750* to *50.000*. Here, we can see once again that the window values of 3, 4 and 5 perform best. It also becomes apparent that the total window size grows very slowly with increasing *n*. Perhaps now Knuth's remark that the function *l(n)* is surprisingly smooth is even more obvious (even though the data represented in this graph is not for *l(n)* itself).

Results for window methods: 750 ≤ n ≤ 50000

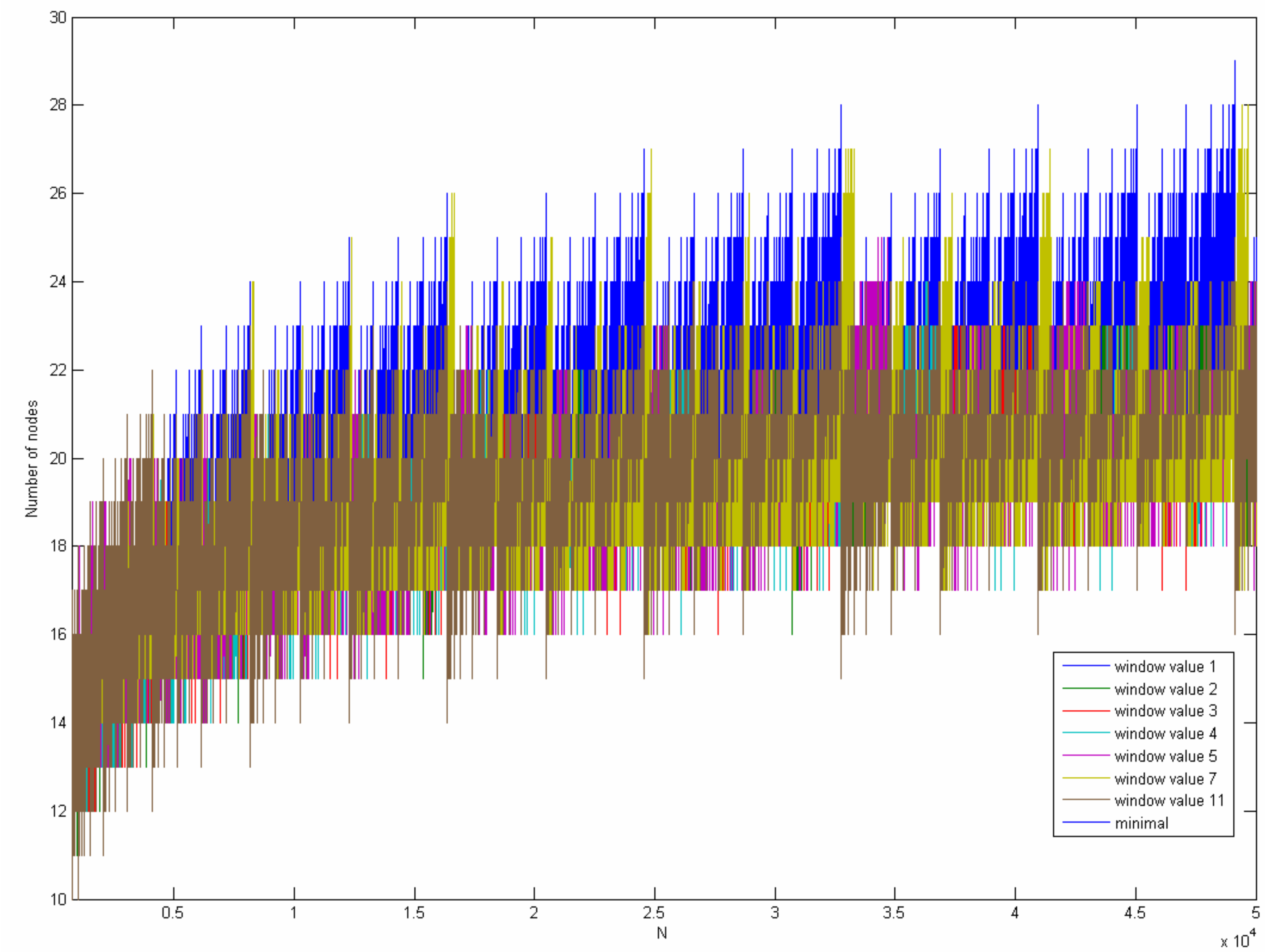


Fig 12.2 The varying window methods and minimal tree length for 750 ≤ n ≤ 50000

12.1.4 Comparing the window method to the factor method.

We just saw that the window method performs best with a window value between 3 and 5. Now, let's have a look at how these methods perform when compared to the factor method.

The window3 method and factor method perform equally well 214937 times.
The window3 method beats the factor method 652548 times.
The factor method beats the window3 method 132515 times.

The first 10 times that	
the window3 method beats the factor method:	*window3 method loses to the factor method:*
23	15
33	30
43	37
46	39
47	55
49	60
59	74
65	75
66	78
67	79

Table 12.9: Comparing the window3 and factor method

The window4 method and factor method perform equally well 198777 times.
The window4 method beats the factor method 653100 times.
The factor method beats the window4 method 148123 times.

The first 10 times that	
the window4 method beats the factor method:	the window4 method loses to the factor method:
33	15
43	27
47	30
49	31
59	45
65	54
66	60
67	61
77	62
83	63

Table 12.10: Comparing the window4 and factor method

The window5 method and factor method perform equally well 191813 times.
The window5 method beats the factor method 652538 times.
The factor method beats the window5 method 155649 times.

The first 10 times that	
the window5 method beats the factor method:	*the window5 method loses to the factor method:*
33	15
49	27
65	30
66	31
67	39
69	45
83	51
98	54
99	55
107	60

Table 12.11: Comparing the window5 and factor method

Of course, the total and average chain length for the factor method and window methods are the same as before:
The total chain length for the factor method is: 25313202
The average chain length for the factor method is: 25.313202

The total chain length for the windows3 method is: 24237567
The average chain length for the windows3 method is: 24.237567

The total chain length for the windows4 method is: 24222477
The average chain length for the windows4 method is: 24.222477

The total chain length for the windows5 method is: 24238943
The average chain length for the windows5 method is: 24.238943

This data shows, that when compared to the factor method, a window length of 3 or 4 seems like the best choice. When choosing a window length of 3, the number of times that the window method loses is smallest, when choosing a window length of 4, the number of times that the window method wins is largest.

12.1.5 Comparing the power tree method to the other approach methods

Now, it is time to take a look at how the power tree method performs when compared to the other three ‘approach-algorithms’.

First of all though, we need to note that when computing the power tree method up to level 25, this algorithm will not have any results for 166 numbers in total. However, the memory constrains of Java inhibit us from increase the level any further

The power tree method equals the binary method 15273 times.
The power tree method beats the binary method 984561 times.
The binary method beats the power tree method 0 times.

The first 10 times the power tree method beats the binary method:
15
23
27
30
31
39
43
45
46
47

Table 12.12: Comparing the power tree and binary method

The power tree method equals the factor method 40298 times.
The power tree method beats the factor method 959493 times.
The factor method beats the power tree total method 43 times.

The first 10 times	
the power tree method beats the factor method:	*the power tree method loses to the factor method:*
23	19879
33	39758
43	43277
46	60749
47	79516
49	86554
59	121498
65	136199
66	159032
67	173069

Table 12.13: Comparing the power tree and factor method

The power tree method equals the window3 method 185264 times.
The power tree method beats the window3 method 813814 times.
The window3 beats the power tree method 756 times.

The first 10 times	
the power tree method beats the window3 method:	*the power tree method loses to the window3 method:*
15	1051
30	2099
37	2102
39	4123
55	4147
60	4198
69	4204
74	6229
75	8219
78	8243

Table 12.14: Comparing the power tree and window3 method

The total chain length for the power tree method is: 22831963
The average chain length for the power tree method is: 22.835753

For the other methods, the chain lengths are now slightly different from those values presented in table 12.1, since here we will not consider those 166 nodes for which the power tree method does not yield any results:
The total chain length for the binary method is: 26831203
The average chain length for the binary method is: 26.835657

The total chain length for the factor method is: 25308562
The average chain length for the factor method is: 25.312763

The total chain length for the window3 method is: 24233184
The average chain length for the window3 method is: 24.237207

It shows, that the power tree method performs much better than the others. On the down side however, this method does need more time and memory to finish. These constraints increase when *n* becomes larger, becoming too large a burden for large *n* at some point for the java-program.

The computer program further shows that using the power tree method with the 'biggest number first' adaptation indeed yields the same result as the doubling method, as was already explained in chapter 8.

Now that we have given the results for all the above algorithms, let's try to visualize these results once more. First, for 1 ≤ n ≤ 750, again with the minimal chain length l(n) added in this figure:

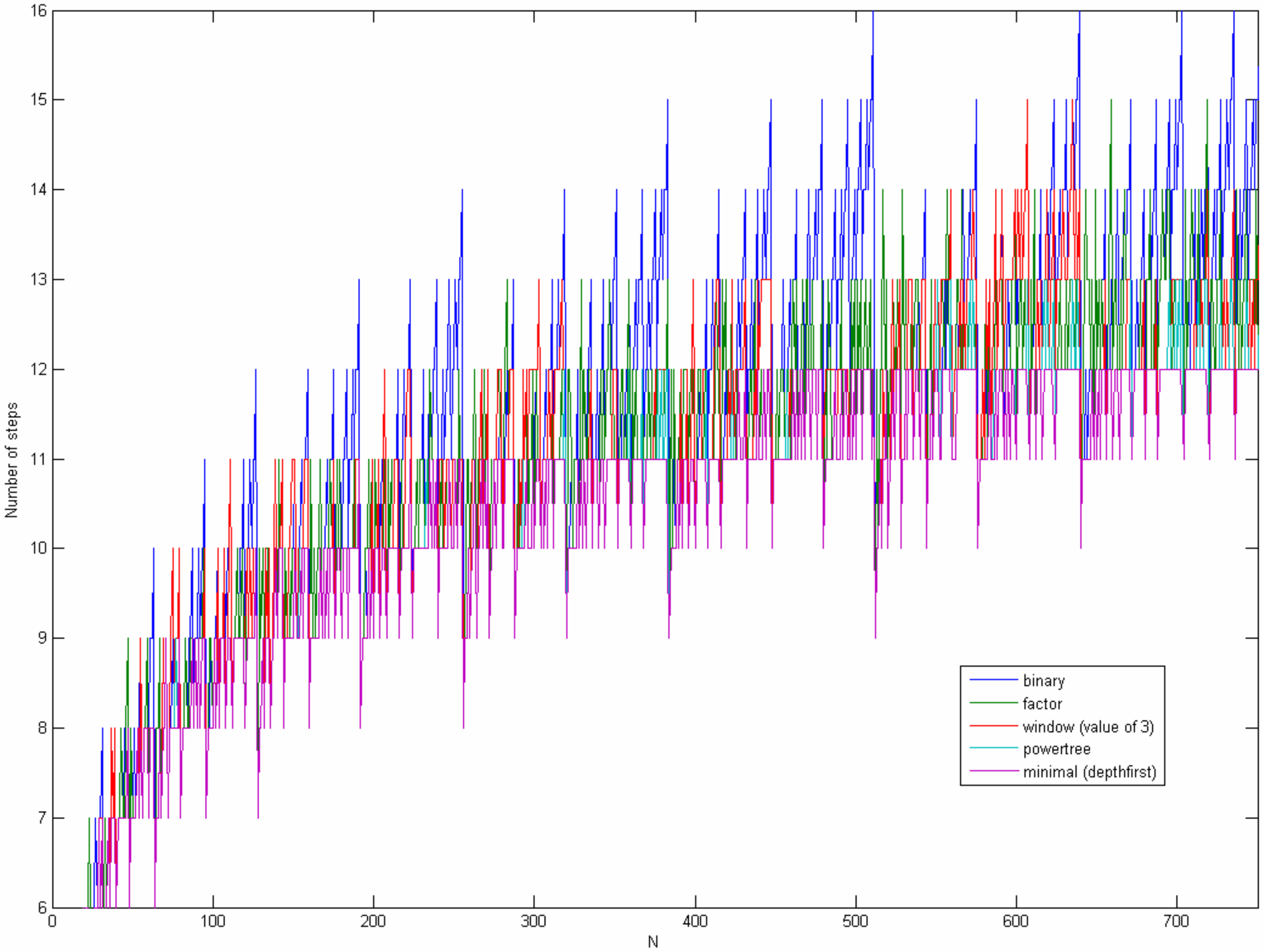


Fig 12.3 The varying approach algorithms and minimal method combined for 1 ≤ n ≤ 750

And the results for all the methods for the 750 ≤ n ≤ 50000:

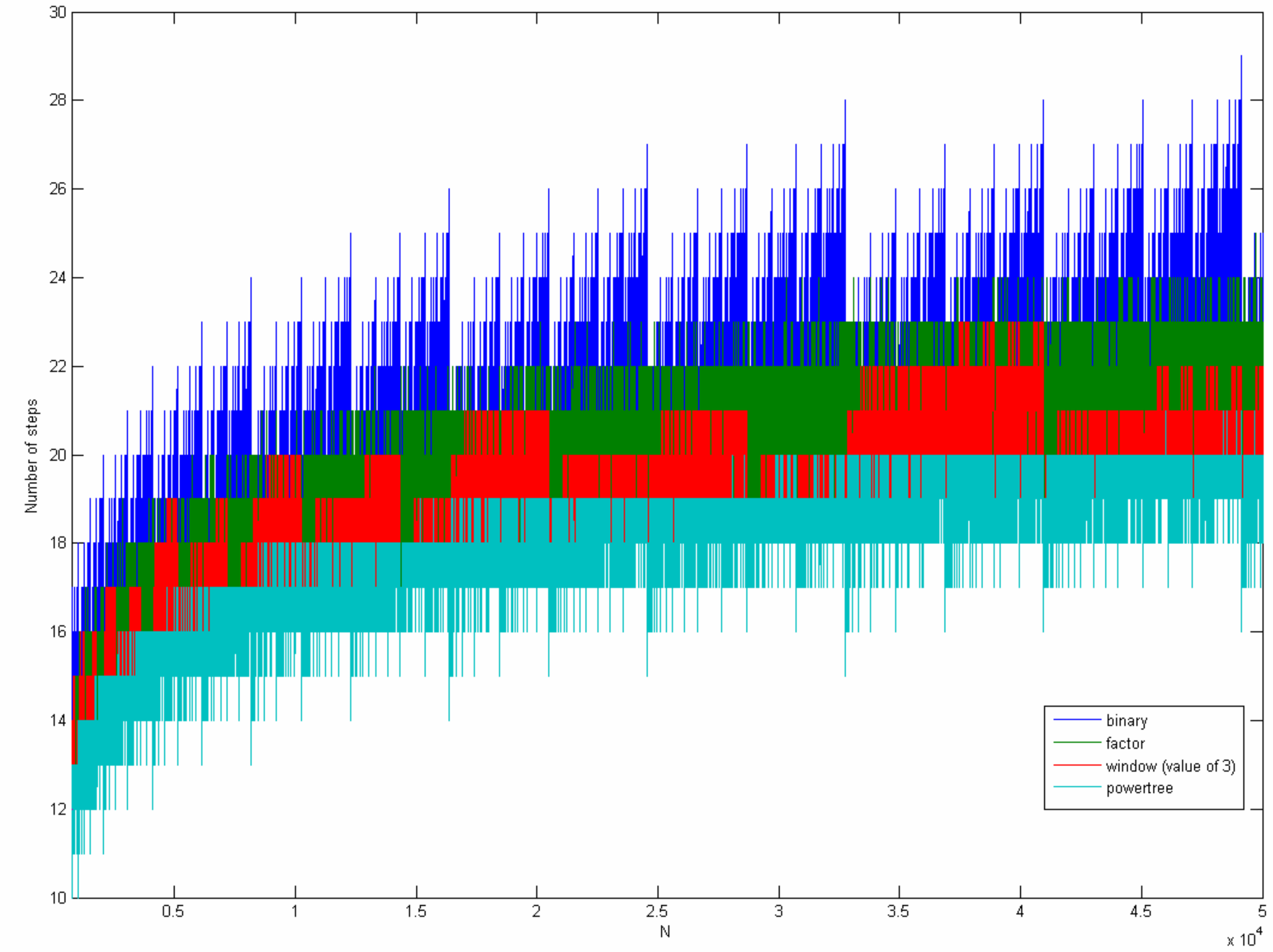


Fig 12.4 The varying approach algorithms for 750 ≤ n ≤ 50.000

And for 50000 ≤ n ≤ 100000:

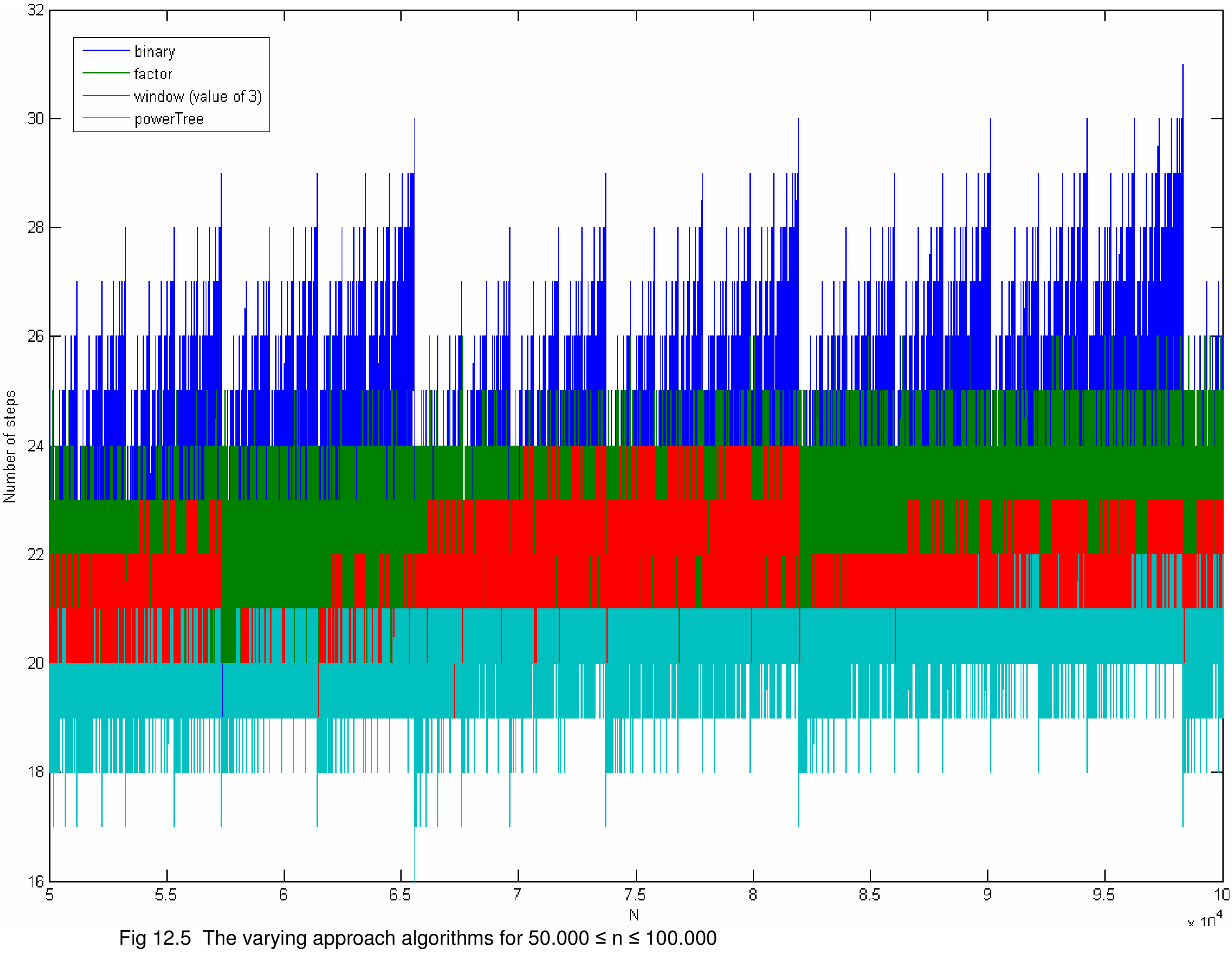


Fig 12.5 The varying approach algorithms for 50.000 ≤ n ≤ 100.000

While we do have results ranging up to 1.000.000, we have refrained from creating such graphs as above for all the test-data, since creating a graph for this amount of numbers would rather cloud the vision than increase insight in and understanding of the result.

When looking at this last picture, we can see once more that the total number of addition steps needed grows increasingly slow for larger numbers of *n*.
In the following table (12.15), one can find the first *n* for each number of steps, obtained using the power tree method. The last number for each level *l* can be calculated simply by the function 2^l: since a doubling step is the biggest possible step and the calculation of 2^l in *l* layers uses doubling steps only, no chains of length *l* can be found for a number larger than 2^l.

Length l	First number n for this layer
1	2
2	3
3	5
4	7
5	11
6	19
7	29
8	47
9	71
10	127
11	191
12	319
13	551
14	1007
15	1711
16	2687
17	4703
18	8447
19	15179
20	28079
21	45997
22	89599
23	138959
24	257513
25	485657

Table 12.15: Values of $c_{powerTree}(r)$

When comparing this table to table 4.1 in chapter 4 (for c(r)), we can see that the results are equal for the first 11 layers. For r ≥ 12, however, c(r) is smaller than $c_{power\ tree}(r)$.

12.2 Total methods:

Let us now look at the total methods.
In this section we will compare the breadth first search and depth first search. No further attention will be given to the combined method: the lower bound that was used is too weak to produce any gain. The result of this is that the combined method will first produce a tree up to the minimal length using depth first search and continue searching afterwards using the breadth first search. This yields the exact same results as the breadth first search itself, and therefore, including the combined method in our comparisons would not produce any new insights.

12.2.1 Results for the breadth first search method

Breadth first search for the numbers 1 – 750 failed to generate a chain for the following numbers (for each of these numbers, the java virtual machine ran out of memory):
319, 379, 383, 407, 427, 431, 535, 559, 571, 573, 575, 599, 605, 607, 623, 631, 633, 635, 637, 638, 639, 669, 671, 695, 699, 701, 703, 717, 719, 727, 731, 733, 734, 735, 743, 747

In total, it failed for 36 numbers under 750 (4.8 %)

12.2.2 Results for the depth first search method

Depth first search for the numbers 1 – 750 failed to generate a chain for these numbers:
379, 383, 407, 559, 573, 599, 605, 607, 623, 631, 635, 638, 671, 695, 699, 701, 703, 717, 719, 727, 733, 743

In total it failed for 22 numbers under 750 (2.93%)

For the following numbers, the depth first search was able to give a chain, where the breadth first search failed: 319, 427, 431, 535, 571, 575, 633, 637, 639, 669, 731, 734, 735, 747 (14 in total)

Looking at this the other way around; there was no number for which the breadth first search did generate a result and the depth first search failed.

One could say that the numbers for which the depth first search method is unable to provide a chain are particularly difficult.

12.2.3 Comparing the depth first search method and the breadth first search method

The following 3 results are generated for those numbers only where both the depth first and breadth first search algorithm both generated a result:

The number of times that depth and breadth first search needed the same amount of nodes: 11
The number of times that breadth first search needed less nodes: 684
The number of times that depth first search needed less nodes: 19

Total number of nodes needed by the **breadth** first search: 170676273
Total number of steps needed by the **breadth** first search: 7278

Total number of nodes needed by the **depth** first search: 186029577
Total number of steps needed by the **depth** first search: 7445

Total number of nodes needed by the **depth** first search, where both breadth and depth first search generated result: 161161100
Total number of steps needed by the **depth** first search, where both breadth and depth first search generated result: 7278
Logically, the depth first and **breadth** first method both need the same amount of steps, since they both generate minimal chains.

The average number of nodes needed per *n* by **breadth** first search: 239042.399
The average number of steps needed per *n* by **breadth** first search: 10.1932773

Average number of nodes needed per *n* by **depth** first search: 255535.133
Average number of steps needed per *n* by **depth** first search: 10.2266484
Average number of nodes needed per *n* by **depth** first search, where both breadth and depth first search generated result: 225715.826
Average number of steps needed per *n* by **depth** first search, where both breadth and depth first search generated result: 10.1932773

Looking at these results, the depth first search algorithm seems to perform best. On average, it needs fewer nodes to compute its result, even though it might happen that the depth first search will add the goal number to its tree more than once (which never happens using the breadth first search). Therefore, the depth first search beats the breadth first search only a handful of times; on the other hand for 684 of the numbers under 750 where both breadth first and depth first search generated a result the breadth first search method needed less nodes. This apparent contradiction can also be explained by the fact that the depth first search method sometimes adds the goal multiple times before finishing. In these cases, the depth first method needs just a couple of extra nodes. However, in other cases, by updating the maximum chain length during the calculations, the depth first method can be way more efficient than the breadth first method for certain *n*. This causes the fact that the breadth first search method will win from the depth first search by a small number of nodes for most *n*, while on the other hand the average chain length for the depth first search method will be shorter.

The comparison between the number of nodes needed by the breadth first search and depth first search can be visualized as well:

For 1 ≤ n ≤ 750

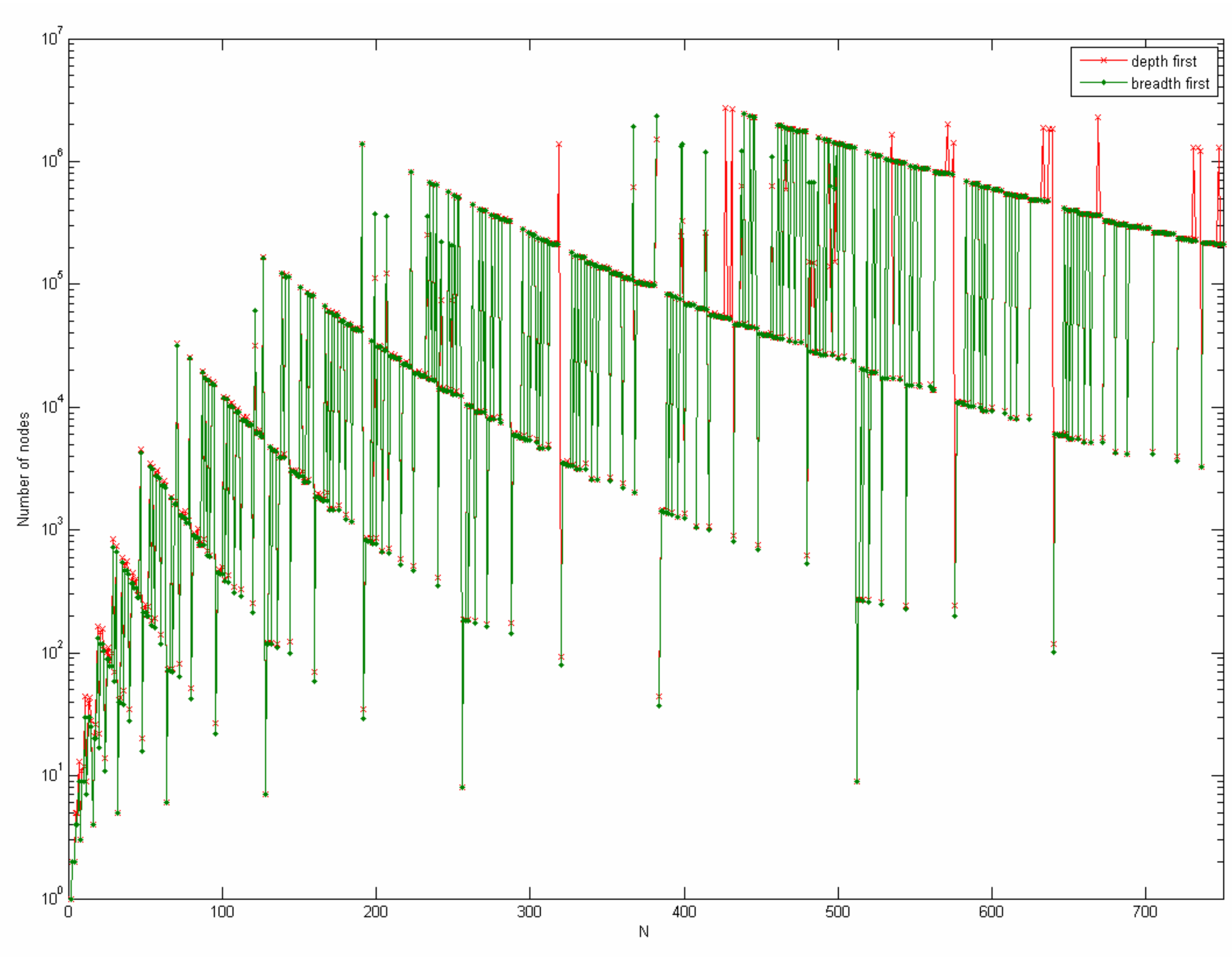


Fig 12.6: Number of nodes used by the breadth first and depth first method for 1 ≤ n ≤ 750

And, for a more detailed look, this figure has been split in three parts: for 1 ≤ *n* ≤ 250

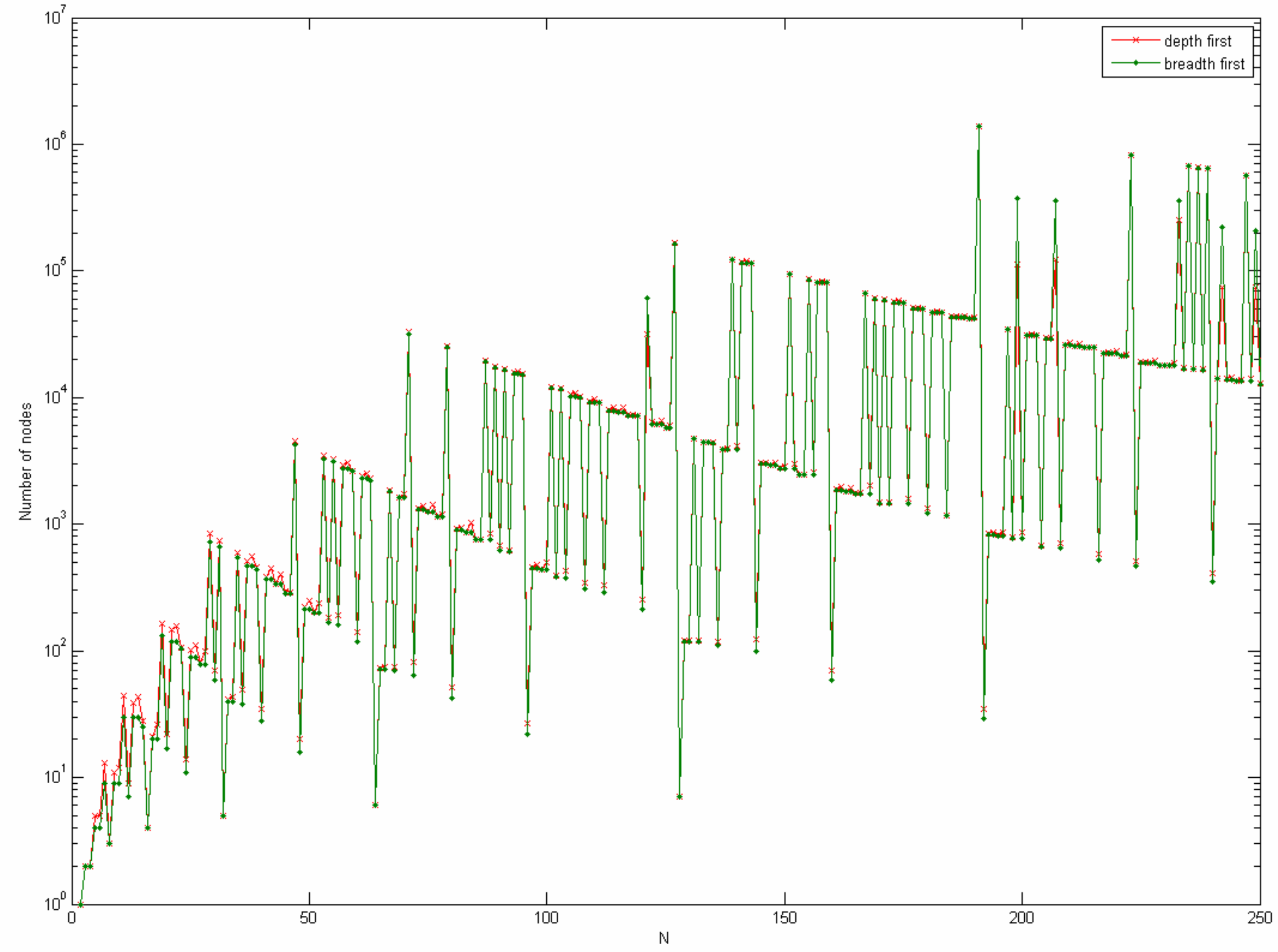


Fig 12.7: Number of nodes used by the breadth first and depth first method for 1 ≤ n ≤ 250

250 ≤ n ≤ 500

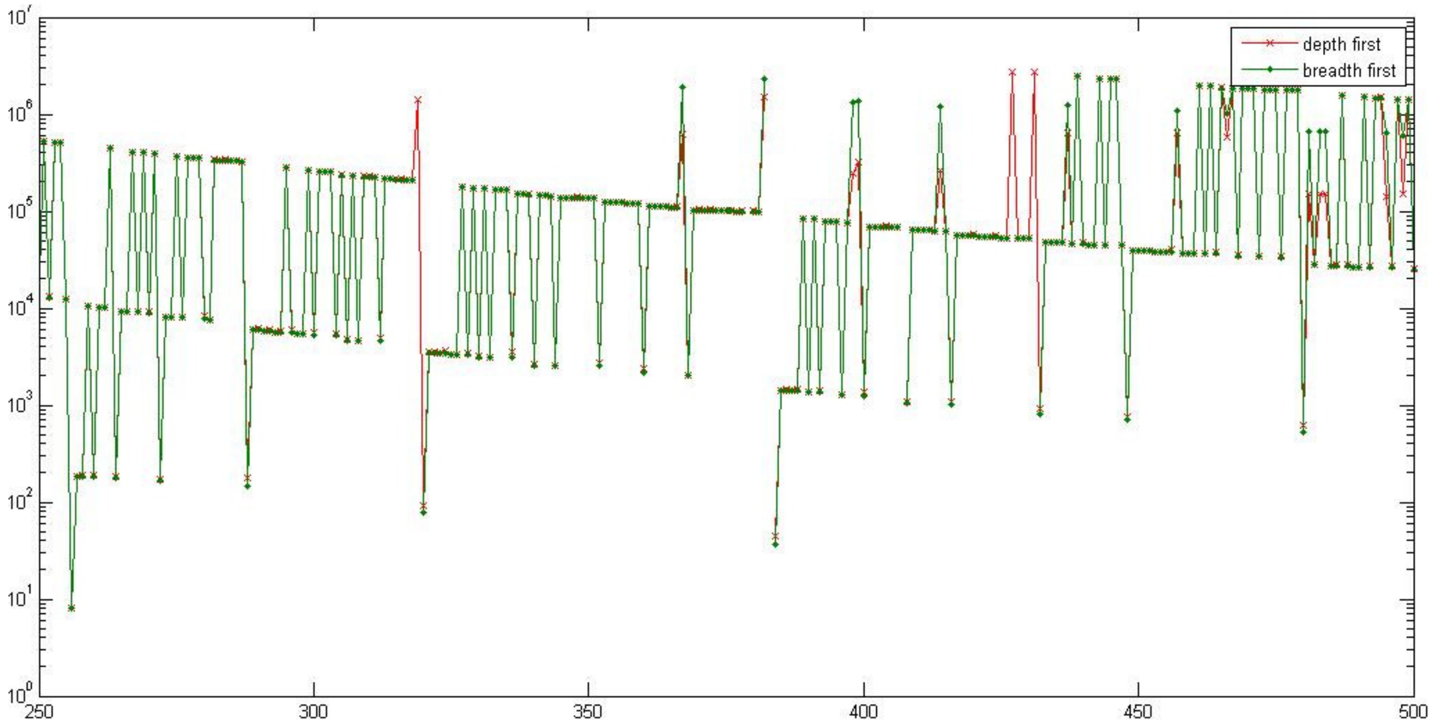


Fig 12.8: Number of nodes used by the breadth first and depth first method for 250 ≤ n ≤ 500

and finally 500 ≤ n ≤ 750

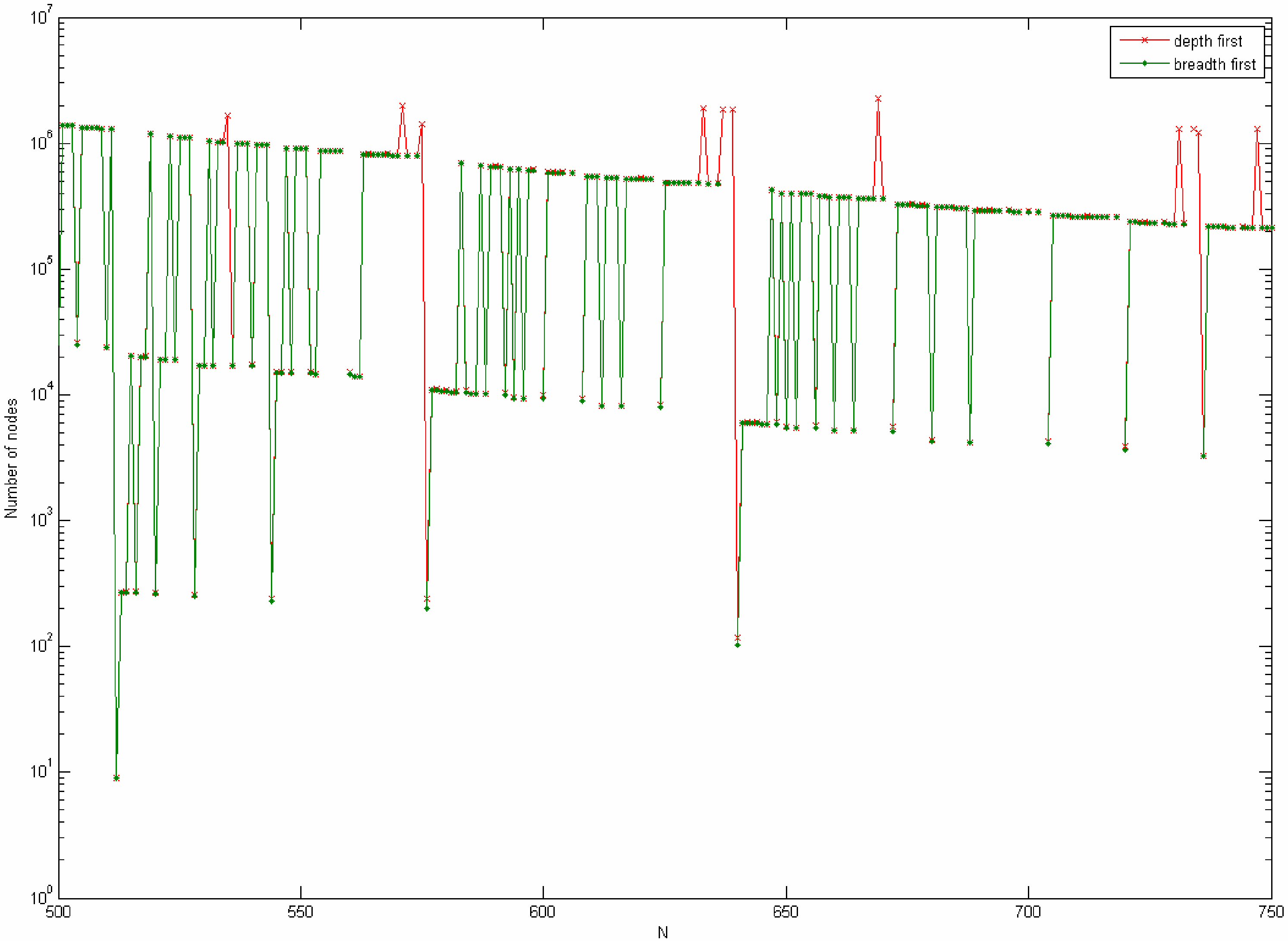


Fig 12.9: Number of nodes used by the breadth first and depth first method for 500 ≤ n ≤ 750

Note that in the last two pictures, some gaps in the graph lines can be seen. These gaps occur where the corresponding method was unable to find a chain without running out of memory.

12.2.4 Comparing the depth first method and the adapted depth first method

As a last check, we can compare the depth first search and the adapted depth first search method (where the largest number is added first)
When computing the results up to n = 500, the adapted algorithm fails to find a chain twice where the regular algorithm does find a result (for the numbers 427 and 431)
In total, the regular algorithm fails 3 times (for n = 379, 383, 407), where the adapted algorithm fails 5 times (for n = 379, 383, 407 427, 431)

Number of times that depth and adapted depth first search use the same amount of nodes: 475
Number of times the **adapted** method needs less nodes: 16
Number of times the **regular** depth first method needs less nodes: 4

Total number of nodes needed by the **adapted** algorithm: 75231924
Total number of steps needed by the **adapted** algorithm: 4725

Total number of nodes needed by the **regular** depth first algorithm: 82638226
Total number of steps needed by the **regular** depth first algorithm: 4749

Total number of nodes needed by the **regular** algorithm, where both the adapted as the regular algorithm compute a chain: 77214766
Total number of steps needed by the **regular** algorithm, where both the adapted as the regular algorithm compute a chain: 4725

Average number of nodes needed by the **adapted** algorithm: 151983.685
Average number of steps needed by the **adapted** algorithm: 9.54545

Average number of nodes needed by the **regular** algorithm: 166274.097
Average number of steps needed by the **regular** algorithm: 9.55533
Average number of nodes needed by the **regular** algorithm, where both the adapted as the regular algorithm compute a chain: 155989.426
Average number of steps needed by the **regular** algorithm, where both the adapted as the regular algorithm compute a chain: 9.54545

Note, that while regular depth first search has been able to find more chains, for those n where both the regular and adapted algorithm generated result, the regular algorithm needs more nodes.

12.3 Comparing the minimal results to the other different approach methods

Finally, we can compare the minimal results to the other methods. This yields the following picture (note that this is the same picture as 12.3):

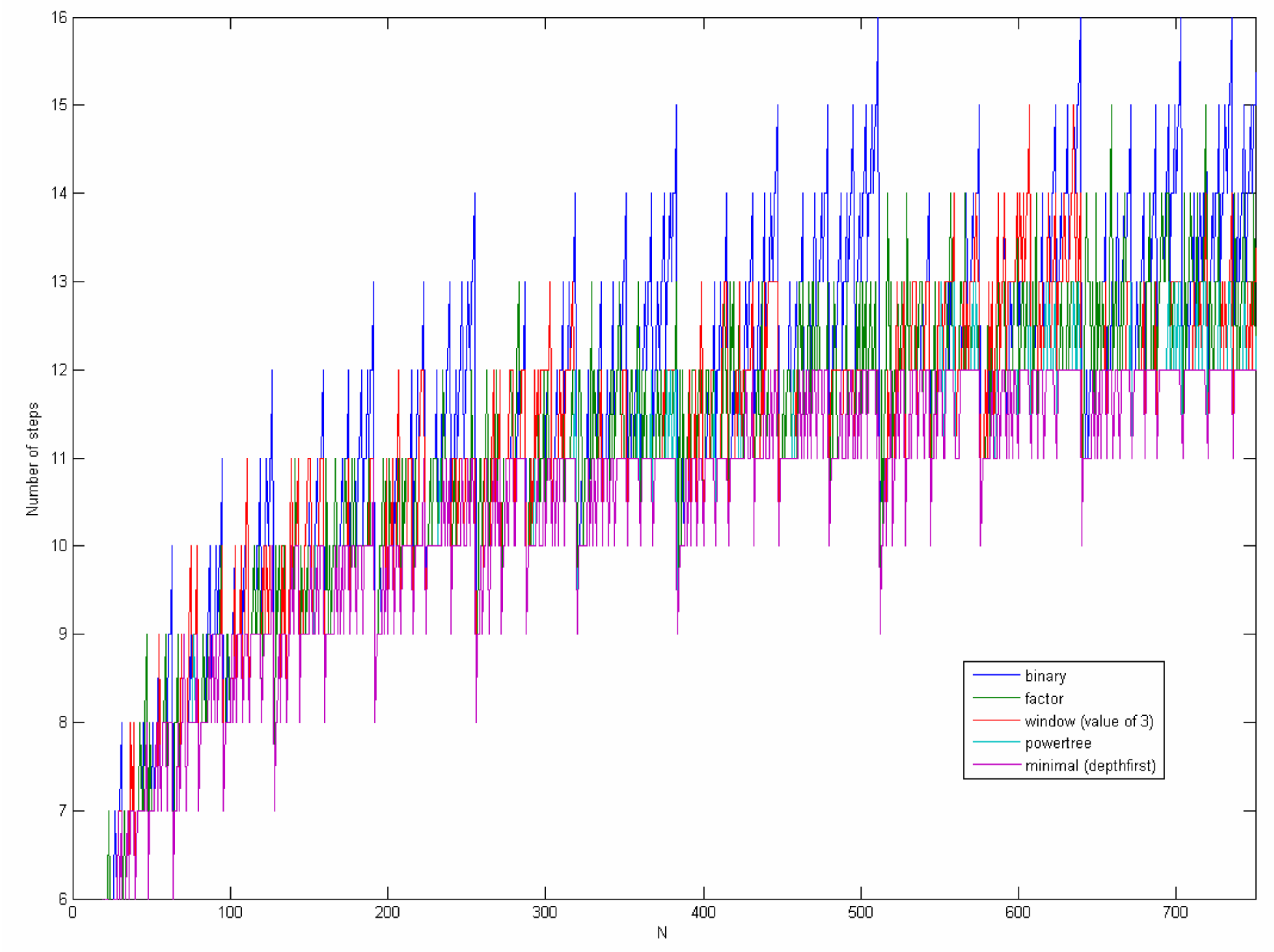


Fig 12.10: The varying approach methods and minimal tree length for 1 ≤ n ≤ 750

Here, we used the depth first search method, since this is capable of finding the largest amount of minimal chains.

The minimal method and binary method tie 254 times.
The minimal method beats the binary method 474 times.
Of course, the binary method never beats the minimal method.

The first 10 times the minimal method beats the binary method
15
23
27
30
31
39
43
45
46
47

Table 12.16: Comparing the minimal and binary method

The minimal method and factor method tie 400 times.
The minimal method beats the factor method 328 times.

The factor method never beats the minimal method.

The first 10 times the minimal method beats the factor method:
23
33
43
46
47
49
59
65
66
67

Table 12.17: Comparing the minimal and factor method

The minimal method and window3 method tie 475 times.
The minimal method beats the window3 method 253 times.
The minimal method never loses to the window3 method.

The first 10 times the minimal method beats the window3 method:
15
30
37
39
55
60
69
74
75
77

Table 12.18: Comparing the minimal and window3 method

The minimal method and power method tie 705 times.
The minimal method beats the power method 23 times.
The power method beats the minimal method 0 times.

The first 10 times the minimal method beats the power tree method:
77
154
233
293
308
319
359
367
377
382

Table 12.19: Comparing the minimal and power tree method

The following results are only for those numbers in the range of 1-750, and only those for which the depth first search algorithm has been able to provide a chain:
The total chain length for the binary method is: 8451
The average chain length for the binary method is: 11.608516

The total chain length for the factor method is: 8110
The average chain length for the factor method is: 11.140110

```
The total chain length for the window3 method is: 8031
The average chain length for the window3 method is: 11.031593

The total chain length for the power tree method is: 7748
The average chain length for the power tree method is: 10.642857

The total chain length for the minimal method is: 7445
The average chain length for the minimal method is: 10.226648
```

Of course, the minimal method will produce the shortest chains for all *n*. No algorithm is capable of producing shorter chains than the minimal method.

13 Conclusion

After looking at the results of the program in the previous chapter, we can conclude the following:

- Based on the average chain length, the binary method performs worst of all analyzed methods.

When looking at the window method, we see that:
- based on the average chain length, a window value of 4 seems optimal for 1 ≤ *n* ≤ 1.000.000
- looking at the fact that the window3 method beats the binary method more often than the window4 method (and the fact that the binary method beats window3 method in fewer cases than it beats the window4 method), a window value of 3 seems optimal for 1 ≤ *n* ≤ 1.000.000)
- looking at figure 12.1 shows, that no single window value performs best for all *n*; different window values perform best for different *n.*
- the window length uses addition sequences. Improvements on sequence-algorithms might therefore lead to improvements to the window method.

Looking at these first two points, the choice for our window value *w* depends on our goal. If the goal is to produce a good result for as many *n* as possible, a window value of 3 is the way to go. If on the other hand the goal is to produce chains that are quite a bit shorter for some *n,* at the cost of producing chains that are slightly longer for other *n*, a window value of 4 seems to be a better choice.

When comparing the approach algorithms, the power tree method performs best for 1 ≤ *n* ≤ 1.000.000. However, as a down-side, it does require more time and memory to complete than the other algorithms. This causes our application to fail for 166 out of the million cases tried. If memory and time constraints are more strict, the window algorithm seems to be the algorithm of choice: with a proper window, it performs better than the factor method and binary method.

When looking at the minimal chain algorithms analyzed in the previous chapter, the depth first method performs best. On average, it needs fewer nodes to calculate a minimal chain. Added to that (and also because of that), our Java program runs out of memory for fewer *n* when using the depth first method when compared to the breadth first routine. The probable reason for this is that the depth first algorithm updates the maximum bound during its calculations. By updating this bound, the total number of nodes that need to be added to calculate a minimal chain can be diminished. A downside of using the depth first algorithm is its stopping condition: The breadth first algorithm can simply halt whenever a chain for *n* has been found. The depth first search method however can only quit when a chain of length equal to the lower bound is found. Otherwise, the algorithm will need to continue until the entire tree has been finished. Therefore, the sharper the lower bound, the better the depth first algorithm performs.

Note that when discussing these results, only the chain length and numbers of nodes visited are taken into account. No tests were done to calculate the actual amount of memory or time needed by any of the algorithms. Apart from the fact that java might not be the optimal language to perform such tests, this also falls outside the scope of this paper. If such issues would be considered to be of bigger importance than the chain length actually produced, one might come to different conclusions.

In the next chapter, a couple of improvements are suggested for the minimal chain methods. If these changes would be applied, the program might perform better.

14. Future work

So far, we have seen the performance of the different algorithms implemented in the program. While we have seen that, when using a minimal algorithm, the depth first method can be used best, we also saw its shortcomings. In this chapter, we will look at some possible adaptations which could improve the results found so far.

14.1 Programming language

First of all, it would be best if the methods would be reprogrammed in another language. While the net result would be equal, the performance of the program could improve on the following points:

- the amount of time needed by the different algorithms
- the total amount of memory needed
- the total amount of memory available.

The combination of these points can result in a program that runs faster and/or will not run out of memory or will do so less often. Especially due to this last point, reprogramming the code into another language will allow the program to calculate chains for a larger amount of *n* when using the minimal algorithms or for larger layers when using the power tree method.

14.2 Bounds

The sharper the bounds are, the better the algorithm will perform.
There are two clear improvements that can be made to the upper bound. First of all, by adapting the wattel-bound in such a way that an optimal value of *k* is used. As we saw in chapter 9, the optimal bound value for *k* is 3, for $n \leq 35.000$. In our implementation however, we used a maximum value for *k*. As shown in figure 9.2 in chapter 9 though, this is a rather bad choice. A second improvement to the upper bound calculation could be to use the power tree method to calculate the practical upper bound. This would require some adaptations to the power tree method, so that it can be calculated for a goal *n* instead of a layer *l*. However, since the power tree method performs better than the other approach algorithms, the result might well be that the upper bound used for the depth first search algorithm is lower.

As seen in chapter 12 when looking at the combined method, the lower bound used in our program is too weak to produce any benefits. If a sharper lower bound can be found, both the combined algorithm and the depth first algorithm might be able to produce better results.

14.3 Pruning algorithm

In the program, the following pruning algorithm was used:

- Calculate the upper bound of the chain.
- Set the minimum value mv for the upper bound to *n*.
- For each layer l, starting at l = (upper bound – 1), the minimum value is $\lceil mv(l+1)/2 \rceil$

We have seen using this pruning algorithm enables us to prune large parts of the tree. However, this algorithm could be further improved. Thurber suggests some possible improvements to this algorithm. In his paper [o(Thurber)], he starts with the same pruning algorithm, and adds some extra cases.
For instance, consider the case where *n* is not a power of 2. In each minimal chain, the last step is a star step, since otherwise the element a_{r-1} could be removed from the chain and this chain would not be minimal. For any odd number, it is also known that the last step is not a doubling step. Therefore, for any minimal chain for an odd *n*,
$n \leq a_{r-1} + a_{r-2}$
So,

$n \leq 3\ a_{r-2}$
As a result, $a_{r-2} \geq \lceil n/3 \rceil$
Now, of course, $a_{r-2} \geq \lceil n/3 \rceil$ can only hold if $a_{r-3} \geq \lceil n/(3 \cdot 2) \rceil$
This process can be repeated until a_0 is reached
When we use this new pruning rule for *n = 55* and an upper bound of 8, we would get the following set of minimal values for each position in the chain:
1 1 2 3 5 10 19 28 55
This clearly gives a better result than the original algorithm, which would produce:
1 1 1 2 4 7 14 28 55

Thurber also adds yet another class of pruning bounds. For more information on this, see [o(Thurber)]

Apart from this, some of the conditions for addition chains given in chapter 4 could also be used for pruning algorithms.

14.4 Adapted algorithm

Last but not least, an adaptation to the search algorithm could prove to be worthwhile. In chapter 12, it became clear that the combined method produced the same results as the breadth first method. A reason for this is that the lower bound used was not sharp enough. Now, there is a possible way to adapt the depth first search algorithm in such a way that the lower bound might be used in a better way. Currently, when sketching the depth first search method in very broad lines, it basically works as follows:

- Create the search tree in a depth first search method, up to the maximum depth (the upper bound).
- Only add those numbers which need not be pruned.
- If a chain for the goal *n* has been found which is shorter than the upper bound, then set the maximum depth to be the length of this chain.
- As soon as a chain for the goal *n* has been found or the entire tree has been searched, we can stop the algorithm

Now, consider the following algorithm

1. Set the ***lower*** bound to be the maximum length *m*
2. Determine the pruning bounds using the pruning algorithm and *m*
3. Build a depth first tree up to level m, using the pruning bounds.
    a. As soon as a chain for *n* has been found, we know this chain is minimal and the algorithm can terminate
4. If no chain has been found, increase *m* by one and repeat the algorithm starting from the second step, building the entire tree from scratch again (since the pruning bounds are updated as well)

This algorithm uses an approach that is in some ways similar to the depth first algorithm described in chapter 10; since it also works its way through the tree in a depth first manner. In other ways however, this adapted algorithm is entirely different. Instead of constructing a depth first tree up to the upper bound, this algorithm starts at the lower bound. When no chain for *n* has been found, the algorithm increases its maximum depth and starts all over again, building a new depth first tree up to the new maximum depth. One of the downsides of this approach is that, with a poor lower bound, the algorithm will need to construct an entirely new tree over and over again; a time consuming process. With a sharp lower bound however, the algorithm performs quite well: it will need one or only a few iterations before a minimal chain has been found. And there is more positive news: the algorithm will require less memory, since it will never have to keep track of any nodes in the tree that are at a depth greater than the actual minimal length. Besides that, due to the fact that also the pruning algorithm is updated in each loop of the algorithm, less useless nodes are added to the tree at lower

levels. As soon as the algorithm starts with a new loop, the old tree can be purged from memory completely, thus freeing this space.
So, in short, using this adapted method will lessen the amount of memory needed, at the cost of increasing the running time of the program.

15 References

[a] "1 + 1 = 2 en hoe nu verder?", Wiskunde B-Dag 2002, 29 november 2002

[b] Evert Wattel and G.A. Jensen, "Efficient Calculation of Powers in a semigroup", Stichting Mathematisch Centrum, februari 1968

[c] Jörg Sauerbrey and Andreas Dietel, "Resource Requirements for the application of addition chains in modulo exponentiation" , Lecture Notes in Computer Science Volume 658/1993, Lehrstuhl für Datenverarbeiting, Technische Universität München. 1993

[d] Jurjen Bos and Matthijs Coster, "Addition Chain Heuristics" , Proceedings on advances in cryptology, Centrum voor Wiskunde en Informatica, 1989

[e] Y. Yacobi, "Exponentiating faster with addition chains", Lecture notes in computer science, Volume 473/1991, p 222 – 229, Bellcore, 1991

[f] Donald Knuth, The art of Computer programming, Volume 2, Seminumerical algorithms, p 441-462, Addison-Wesley, 1980

[g] F. Bergeron, J. Berstel and S. Brlek, "Efficient computation of addition chains", Journal de théorie des nombres de Bordeaux 6, p 21-38, 1994

[h] Zhong Li and Ashish Tiwari, "RSA Public Key Cryptosystem", 2000

[I] Dan Gordon, "Fast exponentiation methods", Center for communications research, 1999

[j] François Morain and Jorge Olivos, "Speeding up the computations on an elliptic curve using addition-substraction chains", 1990

[k] Wieb Bosma, "Signed bits and fast exponentiation", Journal de Theorié des Nombres de Bordeaux 13, p 27-41, 2001

[l] Daniel Bleichenbacher, "Efficiency and Security of Cryptosystems based on number theory, dissertation for the degree of doctor of technical sciences", 1996

[m] Achim Flammenkamp, "Drie Beiträge zur diskreten Mathematik, Addiontionsketten, No-three-in-line-problem, Sociable Numbers, Diplomarbeit, Fakultät für Mathematik der Universität Bielefeld", 1991

[n] Edward G. Thurber, "Addition chains – an erratic sequence", department of Mathematics and Computer science, Biola University, La Mirada, 1992

[o] Edward G. Thurber, "Efficient Generation of minimal length addition chains", SIAM J. Comput. Vol 28, No.4, p. 1247-1263, 1999

[p] Daniel Bleichenbacher and Achim Flammenkamp, "An efficient algorithm for computing shortest addition chains"

[q] Noboru Kunihiro and Hirosuke Yamamoto, "Window and extended window methods for addition chain and addition-substraction chain", special section on Cryptography and information security, IEICE Trans. Fundamentals, vol E81-A, 1998

[r] F. Bergeron, J. Berstel, S. Brlek and C. Duboc, "Addition chains using continued fractions", journal of algorithms 10, p 403-412, 1989

[s] Joachim von zur Gathen and Michael Nöcker, "Computing special powers in finite fields", Mathematics of computation, 2003

[t] Daniel M. Gordon, "A survey of fast exponentiation methods", Center for communication research, 1997

[u] http://en.wikipedia.org/wiki/lucas_chain, retrieved 2007

[v] Moses Charikar, Eric Lehman, April Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai and Abhi Shelat, "The smallest grammar problem," IEEE transactions on information theory, 2002

[w] http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html, retrieved 2007

[x] http://en.wikipedia.org/wiki/Fermat_number#Primality_of_Fermat_numbers, retrieved 2007.

[y] http://wwwhomes.uni-bielefeld.de/achim/addition_chain.html, retrieved 2006, Achim Flammenkamp

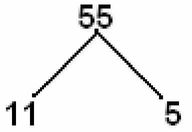
Appendices

I The build-tree algorithm for *n = 55*
II Shortest addition chains for n ≤ 750
III AdditionChains.java

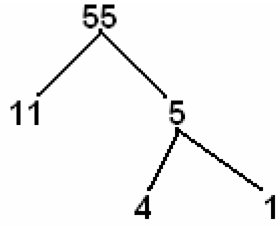
Appendix I: The build-tree algorithm for n = 55

1. Add 55 to the tree:

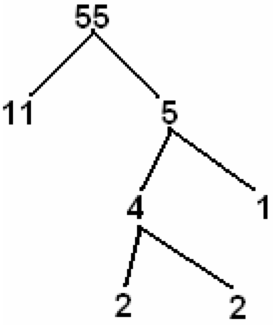
```
BuildTree(55)
2.  55 is not equal to 1
    a.  p = 5, q =11.
    b.  N/A
    c.
```



```
    d.  call BuildTree(5)
          BuildTree(5)
            2. 5 is not equal to 1
              b.  p= 5, q = 1
              c.  p = 1, q = 4
              d.
```

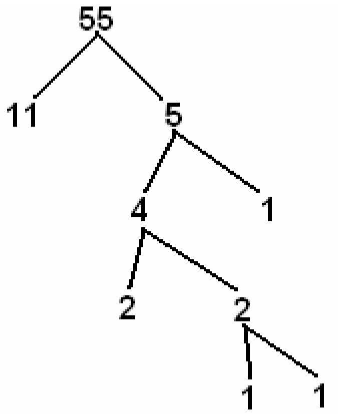


```
              e.  p = 1
              f.  q is not equal to 1, so call BuildTree(4)
                    BuildTree(4)
                       2. 4 is not equal to 1
                         c. p = 2. q = 2
                         d. N/A
                         e.
```



```
                         f.  call BuildTree(2)
                               BuildTree(2)
                                  2. 2 is not equal to 1
                                    a.   p = 1, q = 1
                                    b.   p = 1, q = 1
```

c.



d. p = 1

e. q = 1

e. call BuildTree(2)

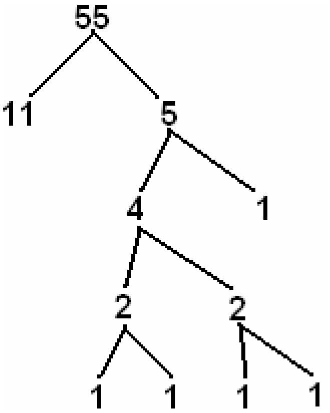
BuildTree(2)

2. 2 is not equal to 1

a. p = 1, q = 1

b. p = 1, q = 1

c.



d. p = 1

e. q = 1

e. call BuildTree(11)

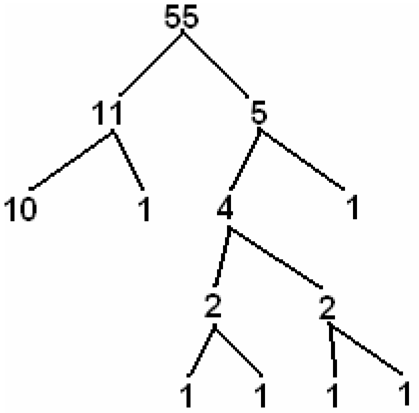
BuildTree(11)

2. 11 is not equal to 1

a. p= 11, q = 1

b. p = 1, q = 10

c.

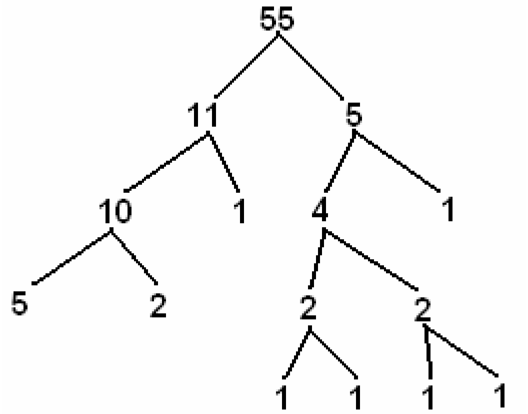


d. p = 1

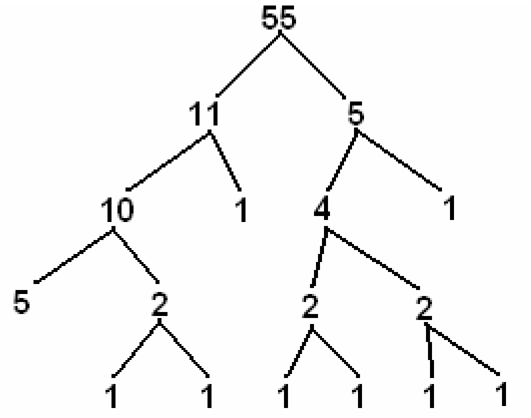
e. call BuildTree(10)

BuildTree(10)

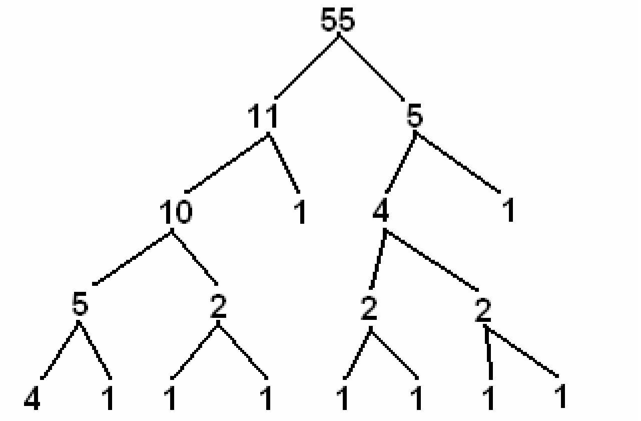
2. 10 is not equal to 1
 a. p = 2. q = 5
 b. N/A
 c.



 d. call BuildTree(2)
   BuildTree(2)
     2. 2 is not equal to 1
      a. p = 1 q = 1
      b. p = 1, q = 1
      c.

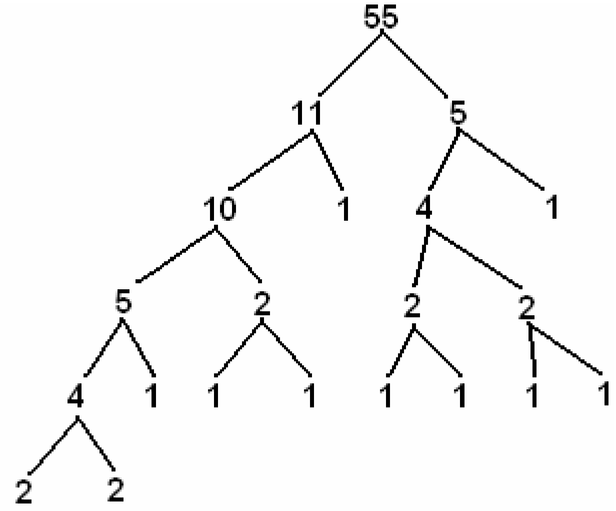


      d. p = 1
      e. q = 1
 e. call BuildTree(5)
   BuildTree(5)
     2. 5 is not equal to 1
      a. p = 5, q = 1
      b. p = 1, q = 4
      c.

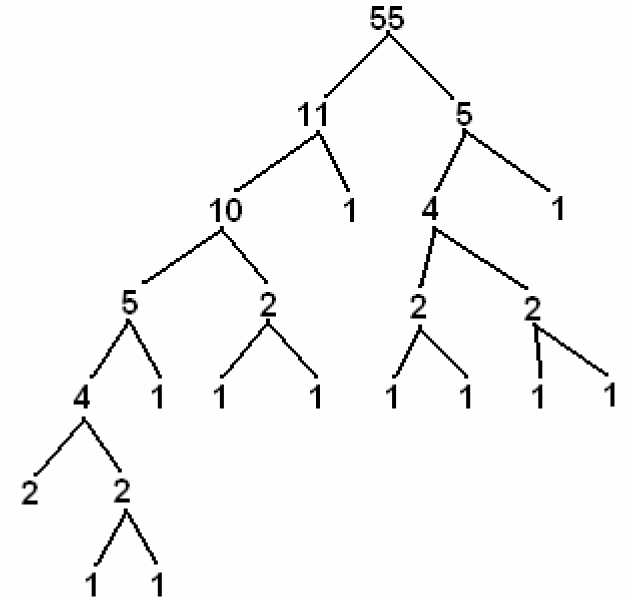


      d. p = 1

e. call BuildTree(4)
   2. 4 is not equal to 1
      a. p = 2. q = 2
      b. N/A
      c.

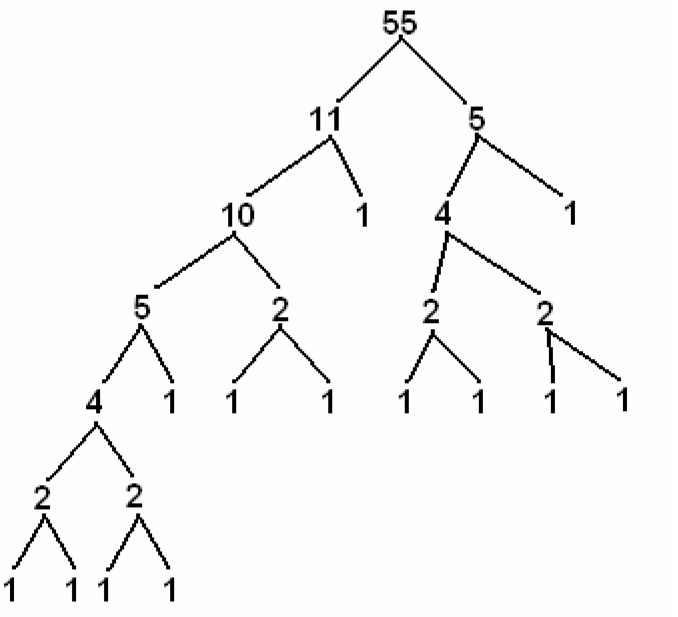


      d. call BuildTree(2)
         BuildTree(2)
           2. 2 is not equal to 1
              a. p = 1, q = 1
              b. p = 1, q = 1
              c.

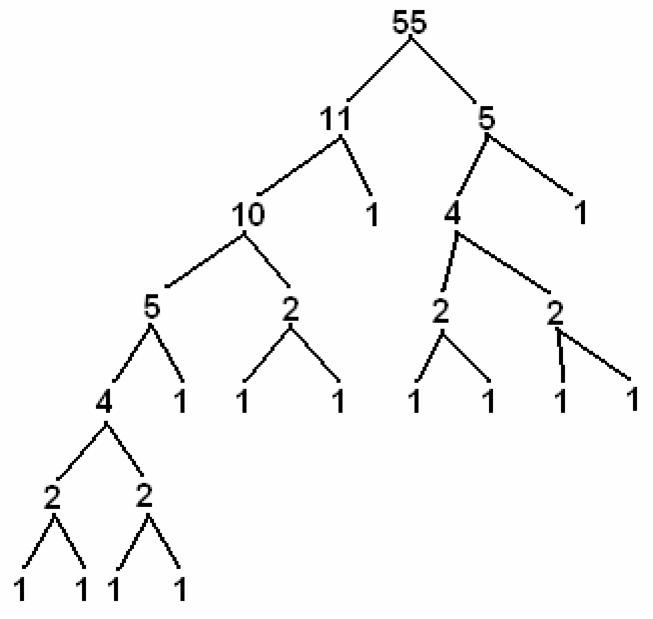


              d. p = 1
              e. q = 1
      e. call BuildTree(2)
         2. 2 is not equal to 1
            a. p = 1, q = 1
            b. p = 1, q = 1

c.



d. p = 1
e. q = 1



Now that we have the factor tree, we still need to convert this to a proper addition chain, by following the rest of the algorithm.

Appendix II: Shortest addition chains for n ≤ 750

In the following table, the shortest addition chains are given, as found by the depth first search algorithm (see Appendix III for exact code of this algorithm). As mentioned in this paper, this algorithm was unable to found a chain for 22 numbers in this range, indicated by '—' in the table below.

Goal	Minimal chain	# Nodes needed by
n	length l(n)	depth-first search
1	0	0
2	1	1
3	2	2
4	2	2
5	3	5
6	3	5
7	4	13
8	3	3
9	4	11
10	4	12
11	5	44
12	4	9
13	5	39
14	5	43
15	5	28
16	4	4
17	5	21
18	5	26
19	6	163
20	5	22
21	6	147
22	6	158
23	6	107
24	5	14
25	6	101
26	6	111
27	6	82
28	6	100
29	7	850
30	6	70
31	7	741
32	5	5
33	6	41
34	6	43
35	7	589
36	6	49
37	7	510
38	7	563
39	7	458
40	6	35
41	7	386
42	7	447
43	7	338
44	7	403
45	7	293
46	7	287
47	8	4496
48	6	20
49	7	225
50	7	249
51	7	203
52	7	237
53	8	3495
54	7	181
55	8	3273
56	7	192
57	8	2914
58	8	3093
59	8	2646
60	7	142
61	8	2387
62	8	2529
63	8	2306
64	6	6
65	7	73
66	7	75
67	8	1835
68	7	75
69	8	1641
70	8	1746
71	9	32801
72	7	81
73	8	1347
74	8	1402

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
75	8	1291
76	8	1420
77	8	1138
78	8	1202
79	9	25441
80	7	51
81	8	926
82	8	944
83	8	871
84	8	1028
85	8	762
86	8	762
87	9	19684
88	8	848
89	9	17693
90	8	674
91	9	16977
92	8	624
93	9	15749
94	9	16086
95	9	15403
96	7	27
97	8	459
98	8	473
99	8	437
100	8	495
101	9	12219
102	8	395
103	9	11737
104	8	431
105	9	10592
106	9	10793
107	9	10085
108	8	341
109	9	9337
110	9	9743
111	9	9144
112	8	327
113	9	8027
114	9	8404
115	9	7760
116	9	8338
117	9	7321
118	9	7331
119	9	7218
120	8	252
121	9	31430
122	9	6439
123	9	6153
124	9	6508
125	9	5827
126	9	6089
127	10	165419
128	7	7
129	8	120
130	8	122
131	9	4776
132	8	122
133	9	4413
134	9	4450
135	9	4401
136	8	118
137	9	3899
138	9	3952
139	10	122855
140	9	4118
141	10	117337
142	10	119454
143	10	115154
144	8	123
145	9	3030
146	9	3091
147	9	2938
148	9	3093
149	9	2775
150	9	2887
151	10	94611
152	9	2973
153	9	2467
154	9	2461
155	10	85896
156	9	2593
157	10	81589
158	10	82811
159	10	80853
160	8	70
161	9	1891
162	9	1961

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
163	9	1830
164	9	1915
165	9	1757
166	9	1757
167	10	66783
168	9	2001
169	10	60732
170	9	1495
171	10	59403
172	9	1475
173	10	56507
174	10	57988
175	10	56061
176	9	1578
177	10	50877
178	10	51625
179	10	50027
180	9	1337
181	10	47359
182	10	48209
183	10	46815
184	9	1183
185	10	43684
186	10	44432
187	10	43187
188	10	44268
189	10	42519
190	10	42860
191	11	1392250
192	8	35
193	9	844
194	9	860
195	9	818
196	9	858
197	10	34885
198	9	786
199	10	112530
200	9	862
201	10	31285
202	10	31702
203	10	30760
204	9	682
205	10	29479
206	10	29607
207	10	122609
208	9	708
209	10	26054
210	10	26949
211	10	25572
212	10	26479
213	10	24834
214	10	24892
215	10	24800
216	9	578
217	10	22516
218	10	22699
219	10	22297
220	10	23295
221	10	21570
222	10	21822
223	11	821655
224	9	513
225	10	19007
226	10	19076
227	10	18646
228	10	19633
229	10	17952
230	10	18052
231	10	17923
232	10	18789
233	10	250577
234	10	17096
235	11	670574
236	10	16865
237	11	653954
238	10	16649
239	11	649252
240	9	408
241	10	14177
242	10	74630
243	10	13985
244	10	14349
245	10	13590
246	10	13736
247	11	564951
248	10	14087
249	10	73991
250	10	12917

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
251	11	528754
252	10	13401
253	11	511603
254	11	517103
255	10	12328
256	8	8
257	9	185
258	9	187
259	10	10506
260	9	187
261	10	10111
262	10	10148
263	11	443977
264	9	183
265	10	9176
266	10	9205
267	11	410180
268	10	9235
269	11	399303
270	10	9189
271	11	395854
272	9	173
273	10	8082
274	10	8107
275	11	366631
276	10	8171
277	11	355311
278	11	357252
279	11	354641
280	10	8381
281	10	7491
282	11	339626
283	11	332733
284	11	339788
285	11	328279
286	11	329860
287	11	326488
288	9	176
289	10	6029
290	10	6110
291	10	5903
292	10	6092
293	10	5702
294	10	5778
295	11	279882
296	10	5932
297	10	5366
298	10	5360
299	11	262320
300	10	5612
301	11	254285
302	11	255605
303	11	253232
304	10	5507
305	11	235299
306	10	4728
307	11	232429
308	10	4678
309	11	227720
310	11	229529
311	11	226827
312	10	4902
313	11	214548
314	11	215714
315	11	214120
316	11	216485
317	11	210568
318	11	212402
319	11	1390759
320	9	92
321	10	3511
322	10	3544
323	10	3453
324	10	3643
325	10	3322
326	10	3322
327	11	180260
328	10	3434
329	11	171002
330	10	3168
331	11	169445
332	10	3148
333	11	167410
334	11	167495
335	11	166922
336	10	3493
337	11	150258
338	11	152095

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
339	11	148572
340	10	2642
341	11	144223
342	11	146376
343	11	143378
344	10	2578
345	11	137514
346	11	137800
347	11	137078
348	11	140586
349	11	134907
350	11	135914
351	11	134784
352	10	2694
353	11	123596
354	11	124354
355	11	122756
356	11	124802
357	11	121220
358	11	121223
359	11	120943
360	10	2389
361	11	112333
362	11	113042
363	11	111625
364	11	113954
365	11	109833
366	11	110464
367	11	617179
368	10	2039
369	11	103115
370	11	103698
371	11	102615
372	11	104727
373	11	101306
374	11	101532
375	11	101372
376	11	102899
377	11	99208
378	11	100272
379	—	—
380	11	100249
381	11	98149
382	11	1511884
383	—	—
384	9	44
385	10	1432
386	10	1450
387	10	1402
388	10	1448
389	11	83477
390	10	1364
391	11	82889
392	10	1408
393	11	78621
394	11	78945
395	11	78219
396	10	1288
397	11	76439
398	11	246846
399	11	325642
400	10	1374
401	11	69076
402	11	69579
403	11	68485
404	11	69741
405	11	67603
406	11	67571
407	—	—
408	10	1084
409	11	63733
410	11	64138
411	11	63482
412	11	64015
413	11	63060
414	11	266017
415	11	62992
416	10	1084
417	11	56543
418	11	56740
419	11	56165
420	11	58471
421	11	54744
422	11	54816
423	11	54708
424	11	56089
425	11	53103
426	11	53091

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
427	12	2729844
428	11	53161
429	11	52915
430	11	52947
431	12	2693616
432	10	909
433	11	47400
434	11	47678
435	11	47073
436	11	47626
437	11	633570
438	11	46793
439	12	2450573
440	11	48152
441	11	44642
442	11	44734
443	12	2343883
444	11	45182
445	12	2307463
446	12	2314155
447	11	44110
448	10	758
449	11	39125
450	11	39770
451	11	38750
452	11	39253
453	11	38286
454	11	38338
455	11	38265
456	11	39907
457	11	632506
458	11	36584
459	11	36564
460	11	36743
461	12	1964494
462	11	36475
463	12	1958198
464	11	37516
465	12	1869158
466	11	590064
467	12	1859268
468	11	35038
469	12	1827104
470	12	1835578
471	12	1822461
472	11	34178
473	12	1779091
474	12	1787865
475	12	1776000
476	11	33832
477	12	1763114
478	12	1764460
479	12	1759278
480	10	618
481	11	151300
482	11	28593
483	11	150617
484	11	149929
485	11	27600
486	11	27852
487	12	1562803
488	11	28110
489	11	26652
490	11	26804
491	12	1502807
492	11	26990
493	12	1480376
494	12	1487215
495	11	139462
496	11	27137
497	12	1407338
498	11	152088
499	12	1401231
500	11	25360
501	12	1382833
502	12	1385910
503	12	1379798
504	11	26060
505	12	1323025
506	12	1328747
507	12	1317048
508	12	1332106
509	12	1303435
510	11	23775
511	12	1299266
512	9	9
513	10	271
514	10	273

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
515	11	20446
516	10	273
517	11	20145
518	11	20182
519	12	1182879
520	10	269
521	11	19139
522	11	19168
523	12	1128730
524	11	19198
525	12	1120267
526	12	1120698
527	12	1117041
528	10	259
529	11	17195
530	11	17220
531	12	1035920
532	11	17236
533	12	1022069
534	12	1026537
535	12	1652510
536	11	17230
537	12	994135
538	12	995990
539	12	992740
540	11	17256
541	12	976564
542	12	978069
543	12	975558
544	10	241
545	11	15090
546	11	15117
547	12	910950
548	11	15121
549	12	902845
550	12	906875
551	12	900180
552	11	15175
553	11	14729
554	12	873432
555	12	871506
556	12	875670
557	12	867252
558	12	870942
559	—	—
560	11	15392
561	11	13839
562	11	13839
563	12	818056
564	12	828771
565	12	810905
566	12	811409
567	12	810497
568	12	821269
569	12	797115
570	12	801266
571	12	2001597
572	12	801345
573	—	—
574	12	793083
575	12	1428479
576	10	241
577	11	10949
578	11	11062
579	11	10791
580	11	11038
581	11	10528
582	11	10616
583	12	691000
584	11	10806
585	11	10116
586	11	10110
587	12	665775
588	11	10266
589	12	656222
590	12	657850
591	12	655317
592	11	10331
593	12	621216
594	11	9462
595	12	618484
596	11	9412
597	12	613775
598	12	614660
599	—	—
600	11	9876
601	12	587661
602	12	589556

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
603	12	585906
604	12	589763
605	—	—
606	12	584442
607	—	—
608	11	9356
609	12	544668
610	12	547356
611	12	542352
612	11	8257
613	12	534090
614	12	534720
615	12	533769
616	11	8103
617	12	521601
618	12	522503
619	12	521025
620	12	525778
621	12	518569
622	12	518403
623	—	—
624	11	8443
625	12	488703
626	12	489879
627	12	487044
628	12	490240
629	12	483889
630	12	488329
631	—	—
632	12	487814
633	12	1879809
634	12	476771
635	—	—
636	12	480604
637	12	1846550
638	—	—
639	12	1858795
640	10	117
641	11	6011
642	11	6048
643	11	5945
644	11	6048
645	11	5854
646	11	5854
647	12	420565
648	11	6169
649	12	401262
650	11	5537
651	12	399260
652	11	5517
653	12	396053
654	12	397246
655	12	395578
656	11	5652
657	12	376639
658	12	377279
659	12	375323
660	11	5251
661	12	370293
662	12	370479
663	12	370131
664	11	5187
665	12	364421
666	12	365811
667	12	363871
668	12	364801
669	12	2277018
670	12	363534
671	—	—
672	11	5647
673	12	326788
674	12	328224
675	12	325068
676	12	329708
677	12	320561
678	12	321739
679	12	320082
680	11	4328
681	12	309274
682	12	309792
683	12	308686
684	12	314156
685	12	305579
686	12	306043
687	12	305364
688	11	4188
689	12	293855
690	12	294580

Goal n	Minimal chain length l(n)	# Nodes needed by depth-first search
691	12	293484
692	12	294442
693	12	292893
694	12	292791
695	—	—
696	12	298707
697	12	287068
698	12	287080
699	—	—
700	12	289459
701	—	—
702	12	286435
703	—	—
704	11	4313
705	12	264976
706	12	265528
707	12	264122
708	12	266218
709	12	262467
710	12	262556
711	12	262351
712	12	265405
713	12	258581
714	12	259284
715	12	258398
716	12	258687
717	—	—
718	12	258075
719	—	—
720	11	3953
721	12	236468
722	12	237446
723	12	235304
724	12	237443
725	12	233370
726	12	234373
727	—	—
728	12	237160
729	12	228572
730	12	229120
731	12	1308373
732	12	229963
733	—	—
734	12	1296348
735	12	1213867
736	11	3277
737	12	215651
738	12	216435
739	12	215356
740	12	217061
741	12	214271
742	12	214273
743	—	—
744	12	217864
745	12	210828
746	12	210848
747	12	1290912
748	12	211318
749	12	210474
750	12	211170

Appendix III: Code for AdditionChains.java

Please note, that the code for both input.java and output.java has not been included in this file.

III.a: AdditionChains.java

```
class AdditionChains {
  // Naam   : Sander van der Kruyssen
  // Programma: Addition Chains
  // Datum  : 08-10-'05 - 3-08-'07

  Input in;
  Output scrnOut, fileOut;

  String inputError = "wrong kind of input provided";
  int[] iArray;
  char[] cArray;
  BinaryTree tree;
  int pointer;
  int positionInWindows, maxPositionInLowerHalf;
  int totalNumberOfNodes=0;

  AdditionChains() {
    in = new Input();
    scrnOut = new Output();
    iArray = new int[50];
    cArray = new char[100];
  }
  int checkReadInt(){
    int number;
    String sInt = new String();
    sInt = "";

    while(Character.isDigit(in.nextChar())){
      sInt+=in.readChar();
    }
    try{
      number=(Integer.parseInt(sInt));
    }catch(Exception e) {
      scrnOut.println("\nError: Input is too high\n");
      return 0;
    }
    return number;
  }

  void nonIntegerInput() throws Exception {

      if(in.nextChar() == 'q'){
        System.exit(1);
      }
      in.readln();
      throw new Exception("Wrong kind of input provided");

  }

  void printTotalResult(int typeOfOutput, int number, PowerTreeNode node){
    boolean layerContainsChildren = true;

    for(int i =1; i <= node.layer; i++){
      scrnOut.println(node.data + " : ");
    }
  }

  void printLayer(int typeOfOutput, int layer, int number, PowerTree powerTree){
    int numberOfNodesPrinted=0;
    PowerTreeNode current = powerTree.returnRoot();
    int childNumber, currentLayer=1;
    if(typeOfOutput=='1'){
      if(layer==1){
        scrnOut.println(current.data + " 0");
        numberOfNodesPrinted++;
        return;
      }
    }else{
      if(layer==1){
        fileOut.println(current.data + " 0");
        numberOfNodesPrinted++;
        return;
      }
    }

    while(currentLayer<layer){
      // get the leftmost child of the current layer. NOTE: this need not simply be obtained by following
the leftmost branch of each node, since some nodes may contain 0 children
      while(current.numberOfChildren>0&&currentLayer<layer){
        current = current.getLeftMostChild();
        currentLayer++;
      }
      if(currentLayer!=layer){//we have branched down the leftmostbranche, but this does not get us at
the bottom layer... So, find the leftmostchild at current layer
        if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of current child
has other kids to currents right
          current = current.getNextChild();
        }else{//we need to go back up the tree to find the next node that has unexplored kids
          int nextChild=current.currentChildNumber;

while((current.currentChildNumber==(current.parent.numberOfChildren))&&current.parent!=null){ //as
long as this child is its parent rightmost kid (and there are still parents to explore...)
            current=current.parent; //go one level up the tree
            currentLayer--;
          }
          if(current.parent!=null){//there are yet more nodes to visit!
            current=current.getNextChild();
          }else{
            current=null;
          }
        }
      }
    }

    while(current!=null){//add children to current node, and find next node to add children to
      if(typeOfOutput=='1'){
        scrnOut.println(current.data + " " +(layer-1));
      }else{
        fileOut.println(current.data + " " +(layer-1));
      }
      /*for(int h = current.layer; h>=0; h--){
        PowerTreeNode temp=current;
        for(int i = h; i>0; i--){
          temp = temp.parent;
        }
        if(typeOfOutput=='1'){
          scrnOut.print(temp.data);
          if(h>0){
            scrnOut.print(" : ");
          }
          numberOfNodesPrinted++;
        }else{
          fileOut.print(temp.data);
          if(h>0){
            fileOut.print(" : ");
          }
          numberOfNodesPrinted++;
        }
      }*/
      /*if(typeOfOutput=='1'){
        scrnOut.println();
      }else{
        fileOut.println();
      }*/

      //get the next node in the current layer, for which we need to add children. Set current to null if no
such node can be found
      if(current.parent!=null){
        if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of current child
has other kids to currents right
          current = current.getNextChild();
        }else{//we need to go back up the tree to find the next node that has unexplored kids
          while(current.parent!=null&&
(current.currentChildNumber==(current.parent.numberOfChildren))){ //as long as this child is its parent
rightmost kid (and there are still parents to explore...)
            current=current.parent; //go one level up the tree
            currentLayer--;
          }
          if(current.parent!=null){//there are yet more nodes to visit!
            current=current.getNextChild();
            //now, we 'only' need to find the 'leftmostnode'
            while(currentLayer<layer&&current!=null){// get the leftmost child of the current layer. NOTE:
this need not simply be obtained by following the leftmost branch of each node, since some nodes may
contain 0 children
              while(current.numberOfChildren>0&&currentLayer<layer){
                current = current.getLeftMostChild();
                currentLayer++;
              }
              if(currentLayer!=layer){//we have branched down the leftmostbranche, but this does not
get us at the bottom layer... So, find the leftmostchild at current layer
                if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of
current child has other kids to currents right
                  current = current.getNextChild();
                }else{//we need to go back up the tree to find the next node that has unexplored kids

while(current.parent!=null&&(current.currentChildNumber==(current.parent.numberOfChildren))){ //as
long as this child is its parent rightmost kid (and there are still parents to explore...)
                    current=current.parent; //go one level up the tree
                    currentLayer--;
                  }
                  if(current.parent!=null){//there are yet more nodes to visit!
                    current=current.getNextChild();
                  }else{
                    current=null;
                  }
                }
              }
            }
          }else{
            current=null;
          }
        }
      }else{
        current=null;
      }
    }
  }

  void printPowerResult(int typeOfOutput, int number, PowerTree powerTree){
    boolean layerContainsChildren = true;

    for(int i =1; i <= number; i++){
      printLayer(typeOfOutput, i, number, powerTree);
    }

    scrnOut.println();
  }

  void printResult(int position,int number, int typeOfOutput) {
    if(typeOfOutput =='1'){
      for(int i = 0; i<position; i++){
             if(i!=(position-1)){
          //scrnOut.print(" : ");
        }
      }
      //scrnOut.println("");
      scrnOut.println(number + " " + (position-1));
     }else{
      if(typeOfOutput=='2'){
        for(int i = 0; i<position; i++){
          //fileOut.print(iArray[i]);
          if(i!=(position-1)){
          //fileOut.print(" : ");
          }
        }
        //fileOut.println("");
      }
      fileOut.println(number + " " + (position-1));
    }

    //pas deze methode aan om te wisselen tussen het printen van de keten zelf en het aantal stappen
in de keten, afhankelijk van wat als meest nuttig wordt beschouwd.
  }

  int maximumChainLength(int number){
    //Upper bound: l(n)< |_log n _| + v(n) -1
    int maximumLength, temp, windowSize;

    //theoretische bovengrenzen voor keten voor n = number
    int logn = (int)(Math.log(number)/Math.log(2));
    cArray = (Integer.toBinaryString(number)).toCharArray();
    int numberOfOnes=0;
    int A = 0, B = 0, C= 0 , D= 0;
    for(int i = 0; i<cArray.length; i++){
      if(cArray[i]=='1'){
        if(numberOfOnes==0){
          A=cArray.length-i;
        }else{
          if(numberOfOnes==1){
            B=cArray.length-i;
          }else{
            if(numberOfOnes==2){
              C=cArray.length-i;
            }else{
              if(numberOfOnes==3){
                D=cArray.length-i;
              }
            }
          }
        }
        numberOfOnes++;
      }
    }

    maximumLength = logn + numberOfOnes - 1;

    //as mentioned in knuth, the binary method is optimal when at most 3 1's are present in the binary
representation
    if(numberOfOnes<=3){
        return maximumLength;
    }
    if(numberOfOnes==4){
      // now, if v(n) = 4, then l(n) >= labda(n) + 2 in the following cases (else, it is equal to labda(n)+3
      //1. A-B=C-D
      //2. A-B=C-D+1
      //3. A-B=3, C-D=1
      //4. A-B=5, B-C = C-D= 1
      if(     ((A-B)==(C-D))  ||   ((A-B)==(C-D+1))     ||   ( (A-B)==3 && (C-D)==1)    ||   ( (A-B)==5 && (B-
C)==1 && (C-D) == 1) ){

         maximumLength = logn + 2;
         return maximumLength;
      }

    }

    // upper bound by wattel:
    //l(n)</ log(n)/log(2) + ((log(n)/log(2))/k) - k + 2^(k-1) + 1, where n >/ 2^ (2k)
    int k = 0;
    while(number>= Math.pow(2, (2*(1+k)))){
      k++;
    }

    temp = ((int)((Math.log(number)/Math.log(2))+((Math.log(number)/Math.log(2))/k)-k+Math.pow(2, (k-
1))+1))-1;
    if(temp<maximumLength){
      maximumLength = temp;
    }


    temp = buildFactorTree(number)-1;

    if(temp<maximumLength){
      maximumLength = temp;
    }
    windowSize = 3;
    temp = calcWindow(number, windowSize)-1;
    if(temp<maximumLength){
      maximumLength = temp;
    }
    windowSize = 5;
    temp = calcWindow(number, windowSize)-1;
    if(temp<maximumLength){
      maximumLength = temp;
    }
    windowSize = 7;
    temp = calcWindow(number, windowSize)-1;
    if(temp<maximumLength){
      maximumLength = temp;
    }
    return (maximumLength);

  }

  double minimumChainLength(int number){
    // Lower bound: log2n + log2v(n)  - 2.13 </ l(n)
    double minimumLength;

    double logn = (Math.log(number)/Math.log(2));
    cArray = (Integer.toBinaryString(number)).toCharArray();
    int numberOfOnes=0;
    for(int i = 0; i<cArray.length; i++){
      if(cArray[i]=='1'){
        numberOfOnes++;
      }
    }

    minimumLength = (logn + (Math.log(numberOfOnes)/Math.log(2)) - 2.13);
    return minimumLength;
  }

  int minimumValue(int layer, int number, int maximumLength){
    int currentValue= number;
    int currentLength=maximumLength;

    if(currentLength<layer){
      return 0;
    }

    while(currentLength>layer){
      currentValue=(int)(currentValue+1)/2; // Why the currentValue +1, you might think? Well, if current
value is X, then the node before X should have at least the value of X/2. Now, if X is even, the previous
node should simply have the value X/2. If X is odd however, it should have the value (X+1)/2. By
calculating (X+1)/2 and casting it to an int afterwards (effectively shopping off the value behind the dot),
we achieve this goal.
      currentLength--;
    }
    return currentValue;
  }

  boolean hasChild(PowerTreeNode node, int valueToAdd){
    for(int i = 0; i<node.numberOfChildren;i++){
      if(node.getChild(i).data==valueToAdd){
        return true;
      }
    }
    return false;
  }

  void buildDepthfirstTree(int number, int layer, PowerTree powerTree, boolean
numberHasBeenAdded, int maximumLength, int minimumLength, int typeOfOutput){
    if(number==1){
      iArray[0]=1;
      printResult(1, number, typeOfOutput);
      return;
    }
    int numberOfNodesAdded=0;
    int numberOfMinimalChains=0;
    PowerTreeNode current;
    current=powerTree.returnRoot();
    int currentLayer = 1;
    int minimumValue;
    PowerTreeNode temporary1=current, temporary2=current;
    PowerTreeNode finalNode = null;
    boolean properChainFound = false;
    boolean anyNodeWasAdded=false;
    while(!properChainFound&&current!=null){
      while(!numberHasBeenAdded&&current!=null){
        anyNodeWasAdded=false;
        temporary1=temporary2=null;
        for(int a = currentLayer; a>=0; a--){ //code for add lowest first!
          temporary1=current;
          for(int b = a; b>1; b--){
            temporary1 = temporary1.parent;
          }
          for(int d = a; d>=0;d--){
            temporary2 = current;
            for(int c = d; c>1;c--){
              temporary2 = temporary2.parent;
            }
            int valueToAdd = temporary2.data + temporary1.data;
            minimumValue = minimumValue(currentLayer,number, maximumLength);
            if(valueToAdd<= number && valueToAdd > current.data
&&!(hasChild(current,valueToAdd))){
              if(minimumValue<=valueToAdd){
                current.addChild(valueToAdd,current, currentLayer);
                numberOfNodesAdded++;

                if(valueToAdd == number){
                  numberHasBeenAdded=true;
                }
                anyNodeWasAdded=true;
              }
            }
          }
        }
        /*
        for(int a = 0; a<=currentLayer; a++){ //code for add highest first!
          temporary1=current;
          for(int b = a; b>1; b--){
            temporary1 = temporary1.parent;
          }
          for(int d = 0; d<=a;d++){
            temporary2 = current;
            for(int c = d; c>1;c--){
              temporary2 = temporary2.parent;
            }
            int valueToAdd = temporary2.data + temporary1.data;
            minimumValue = minimumValue(currentLayer,number, maximumLength);

            if(valueToAdd<= number && valueToAdd > current.data
&&!(hasChild(current,valueToAdd))){
              if(minimumValue<=valueToAdd){
                current.addChild(valueToAdd,current, currentLayer);
                numberOfNodesAdded++;
                if(valueToAdd == number){
                  numberHasBeenAdded=true;
                }
                anyNodeWasAdded=true;
              }
            }
          }
        }
          */
        if(anyNodeWasAdded){
          current=current.getLeftMostChild();
          currentLayer++;
          if(current.layer>(maximumLength)){
            current=current.parent;
            currentLayer--;

            while((current.parent!=null) &&
(current.currentChildNumber==current.parent.numberOfChildren)){
              current=current.parent;
              currentLayer--;
            }
            if(current.parent!=null){
              current=current.getNextChild();
            }else{
              current = null;
            }
          }
        }else{
          if((current.parent!=null) && (current.currentChildNumber==current.parent.numberOfChildren)){
            current=current.parent;
            currentLayer--;

            while((current.parent!=null) &&
(current.currentChildNumber==current.parent.numberOfChildren)){
              current=current.parent;
              currentLayer--;
            }

            if(current.parent!=null){
              current=current.getNextChild();
            }else{
              current = null;
            }
          }else{
            if((current.currentChildNumber<current.parent.numberOfChildren)){
              current=current.getNextChild();
            }
          }
        }
      }

      if(numberHasBeenAdded){
        if(current.layer<maximumLength){
          numberOfMinimalChains=0;
        }
        numberOfMinimalChains++;
        maximumLength=current.layer;
        finalNode = current;
        numberHasBeenAdded=false;
        if(maximumLength == minimumLength){
          properChainFound=true;
        }
        current=current.parent;
        currentLayer--;
        while((current.parent!=null) &&
(current.currentChildNumber==current.parent.numberOfChildren)){
          current=current.parent;
          currentLayer--;
        }
        if(current.parent!=null){
          current=current.getNextChild();
        }else{
          current = null;
        }

      }
    }
    current=finalNode;

    iArray = new int[current.layer+1];

    for(int i = (finalNode.layer); i>=0; i--){

      iArray[i]=current.data;
      if(current.parent!=null){
        current=current.parent;
      }
    }

    if(typeOfOutput=='1'){
      scrnOut.print(numberOfNodesAdded+ " ");
    }else{
      fileOut.print(numberOfNodesAdded+ " ");
    }
    printResult(finalNode.layer+1, number, typeOfOutput);
  }

  void calcDepthfirst(int number, int typeOfOutput) throws Exception{
    PowerTree powerTree = new PowerTree();
    PowerTreeNode temp;
    iArray= new int[125];
    int iposition=0;
    int i = 1;

    int layer;
    int minimumLength = (int)minimumChainLength(number)+1;
    int maximumLength = maximumChainLength(number);

    temp = powerTree.addRoot(i);
    layer=1;
    boolean numberHasBeenAdded = false;
    buildDepthfirstTree(number, layer,powerTree,numberHasBeenAdded,maximumLength,
minimumLength, typeOfOutput);
  }

  void depthfirstMethod(int typeOfOutput, String explanation) throws Exception {
    int beginNumber, endNumber;

    scrnOut.println("Enter a number between 0 and 2147483647 and press enter. \nThis program will
first build a total tree (using certain pruning rules), starting with a root of 1 and up to the provided
number.\n The result will be given as a final addition chain for the entered number.\n NOTE: for a large
number, the calculations might take a while and may run out of memory space.");
    while (!in.eof()) {
      if(!Character.isDigit(in.nextChar())){
          nonIntegerInput();
        }
        if(!in.eoln()){
          beginNumber=checkReadInt();
          if(beginNumber==0){
            return;
          }
          if(in.nextChar()!='-'){
            calcDepthfirst(beginNumber, typeOfOutput);
          }else{
            in.readChar();
            endNumber=checkReadInt();
            if(endNumber==0){
              return;
            }
            if(endNumber>=beginNumber){
              for(int i = beginNumber; i<=endNumber; i++){
              calcDepthfirst(i, typeOfOutput);
              }
            }else{
              throw new Exception("Error: The ending number has to be bigger than the starting
number");
            }
          }
        }
      in.readln();
      scrnOut.print('>');
    }
  }

  boolean buildCombinedTree(int number, int layer, PowerTree powerTree, boolean
numberHasBeenAdded, int maximumLength, int minimumLength){

    int numberOfNodesAdded=0;
    PowerTreeNode current;
    current=powerTree.returnRoot();
    int currentLayer = 1;
    int minimumValue;
    PowerTreeNode temporary1=current, temporary2=current;

    while(!numberHasBeenAdded&&current!=null){
      temporary1=temporary2=null;
      for(int a = currentLayer; a>=0; a--){ //code for add lowest first!
        temporary1=current;
        for(int b = a; b>1; b--){
          temporary1 = temporary1.parent;
        }
        for(int d = a; d>=0;d--){
          temporary2 = current;
          for(int c = d; c>1;c--){
            temporary2 = temporary2.parent;
          }
          int valueToAdd = temporary2.data + temporary1.data;
          minimumValue = minimumValue(currentLayer,number, maximumLength);
          if(valueToAdd<= number && valueToAdd > current.data &&!(hasChild(current,valueToAdd))){
            if(minimumValue<=valueToAdd){
              current.addChild(valueToAdd,current, currentLayer);
              numberOfNodesAdded++;
              if(valueToAdd == number){
                numberHasBeenAdded=true;
              }
            }
          }
        }
      }
      current=current.getLeftMostChild();
      currentLayer++;
      if(current.layer>=(minimumLength)){

        current=current.parent;
        currentLayer--;
        while((current.parent!=null) &&
(current.currentChildNumber==current.parent.numberOfChildren)){
          current=current.parent;
          currentLayer--;
        }
        if(current.parent!=null){
          current=current.getNextChild();
        }else{
          current = null;
        }
      }
    }
    if(numberHasBeenAdded){
      iArray = new int[layer+1];
      iArray[layer]=number;
      for(int i = (layer-1); i>=0; i--){
        iArray[i]=current.data;
        current=current.parent;
      }
    }
    totalNumberOfNodes += numberOfNodesAdded;
    return numberHasBeenAdded;
  }

  void calcCombined(int number, int typeOfOutput) throws Exception{
    PowerTree powerTree = new PowerTree();
    PowerTreeNode temp;
    iArray= new int[125];
    int iposition=0;
    int i = 1;
    int layer;
    totalNumberOfNodes=0;
    if(number==1){
        iArray[0]=1;
        layer=1;
    }else{
      if(number==2){
        iArray[0]=1;
        iArray[1]=2;
        layer=2;
      }else{
        double tempMinLength = minimumChainLength(number);
        int minimumLength = (int)tempMinLength;
        if(minimumLength<tempMinLength){
          minimumLength++;
        }
        int maximumLength = maximumChainLength(number);
        temp = powerTree.addRoot(i);
        layer=1;
        boolean numberHasBeenAdded = false;
        numberHasBeenAdded = buildCombinedTree(number,
layer,powerTree,numberHasBeenAdded,maximumLength, minimumLength);
        layer = (int)minimumLength;
        if(layer<minimumLength){
          layer++;
        }
        layer++;
        while(!numberHasBeenAdded){
          numberHasBeenAdded=addTotalLayer(number, layer, powerTree, numberHasBeenAdded,
maximumLength);
          layer++;
        }
      }
    }
    if(typeOfOutput=='1'){
      scrnOut.print(totalNumberOfNodes+ " ");
    }else{
      fileOut.print(totalNumberOfNodes+ " ");
    }
    printResult(layer, number, typeOfOutput);
  }

  void combinedMethod(int typeOfOutput, String explanation) throws Exception {
    int beginNumber, endNumber;
    scrnOut.println("Enter a number between 0 and 2147483647 and press enter. \nThis program will
first build a total tree (using certain pruning rules), starting with a root of 1 and up to the provided
number.\n The result will be given as a final addition chain for the entered number.\n NOTE: for a large
number, the calculations might take a while and may run out of memory space.");

    while (!in.eof()) {
      if(!Character.isDigit(in.nextChar())){
          nonIntegerInput();
        }
        if(!in.eoln()){
          beginNumber=checkReadInt();
          if(beginNumber==0){
            return;
          }
          if(in.nextChar()!='-'){
            calcCombined(beginNumber, typeOfOutput);
          }else{
            in.readChar();
            endNumber=checkReadInt();
            if(endNumber==0){
              return;
            }
            if(endNumber>=beginNumber){
              for(int i = beginNumber; i<=endNumber; i++){
              calcCombined(i, typeOfOutput);
              }
            }else{
              throw new Exception("Error: The ending number has to be bigger than the starting
number");
            }
          }
        }
      in.readln();
      scrnOut.print('>');
    }
  }

  boolean addTotalLayer(int number, int layer, PowerTree powerTree, boolean numberHasBeenAdded,
int maximumLength){
    if(number==2){
      iArray[0]=1;
      iArray[1]=2;
      return true;
    }else{
      if(number==1){
        iArray[0]=1;
        return true;
      }
    }

    int numberOfNodesAdded=0;
    int currentLayer=1;
    int minimumValue = minimumValue(layer,number, maximumLength);
    PowerTreeNode current;
    current=powerTree.returnRoot();
    while(currentLayer<layer){
      // get the leftmost child of the current layer. NOTE: this need not simply be obtained by following
the leftmost branch of each node, since some nodes may contain 0 children
      while(current.numberOfChildren>0){
        current = current.getLeftMostChild();
        currentLayer++;
      }
      if(currentLayer!=layer){//we have branched down the leftmostbranche, but this does not get us at
the bottom layer... So, find the leftmostchild at current layer
        if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of current child
has other kids to currents right
          current = current.getNextChild();
        }else{//we need to go back up the tree to find the next node that has unexplored kids

while((current.currentChildNumber==(current.parent.numberOfChildren))&&current.parent!=null){ //as
long as this child is its parent rightmost kid (and there are still parents to explore...)
            current=current.parent; //go one level up the tree
            currentLayer--;
          }
          if(current.parent!=null){//there are yet more nodes to visit!
            current=current.getNextChild();
          }else{
            current=null;
          }
        }
      }
    }
    while(current!=null && !numberHasBeenAdded){
      //add children to current node, and find next node to add children to
      PowerTreeNode temporary1=current, temporary2=current;
      for(int a = layer; a>=0; a--){ //code for add lowest first!
        temporary1=current;
        for(int b = a; b>1; b--){
          temporary1 = temporary1.parent;
        }
        for(int d = a; d>=0;d--){
          temporary2 = current;
          for(int c = d; c>1;c--){
            temporary2 = temporary2.parent;
          }
          int valueToAdd = temporary2.data + temporary1.data;

          if(valueToAdd<= number && valueToAdd > current.data &&!(hasChild(current,valueToAdd))){
            if(minimumValue<=valueToAdd){
              current.addChild(valueToAdd,current, layer);
              numberOfNodesAdded++;
              if(valueToAdd == number){
                numberHasBeenAdded=true;
              }
            }
          }
        }
      }//end of code for add lowest first
      /*for(int a = 0; a<=layer; a++){//code for add highest first!
        temporary1=current;

        for(int b = a; b>1; b--){
          temporary1 = temporary1.parent;
        }
        for(int d = 0; d<=a;d++){
          temporary2 = current;
          for(int c = d; c>1;c--){
            temporary2 = temporary2.parent;
          }
          int valueToAdd = temporary2.data + temporary1.data;

          if(valueToAdd<= number && valueToAdd > current.data &&!(hasChild(current,valueToAdd))){

            if(minimumValue<=valueToAdd){
              current.addChild(valueToAdd,current, layer);
              numberOfNodesAdded++;
              if(valueToAdd == number){
                numberHasBeenAdded=true;
              }
            }
          }
        }
      }//end of code for add highest first*/
      //get the next node in the current layer, for which we need to add children. Set current to null if no
such node can be found
      if(current.parent!=null){
        if(!numberHasBeenAdded){
          if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of current child
has other kids to currents right
            current = current.getNextChild();

          }else{//we need to go back up the tree to find the next node that has unexplored kids
            while(current.parent!=null&&
(current.currentChildNumber==(current.parent.numberOfChildren))){ //as long as this child is its parent
rightmost kid (and there are still parents to explore...)
              current=current.parent; //go one level up the tree
              currentLayer--;
            }
            if(current.parent!=null){//there are yet more nodes to visit!
              current=current.getNextChild();
              //now, we 'only' need to find the 'leftmostnode'
              while(currentLayer<layer&&current!=null){
                // get the leftmost child of the current layer. NOTE: this need not simply be obtained by
following the leftmost branch of each node, since some nodes may contain 0 children
                while(current.numberOfChildren>0){
                  current = current.getLeftMostChild();
                  currentLayer++;
                }
                if(currentLayer!=layer){//we have branched down the leftmostbranche, but this does not
get us at the bottom layer... So, find the leftmostchild at current layer
                  if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of
current child has other kids to currents right
                    current = current.getNextChild();
                  }else{//we need to go back up the tree to find the next node that has unexplored kids

while(current.parent!=null&&(current.currentChildNumber==(current.parent.numberOfChildren))){ //as
long as this child is its parent rightmost kid (and there are still parents to explore...)
                      current=current.parent; //go one level up the tree
                      currentLayer--;
                    }
                    if(current.parent!=null){//there are yet more nodes to visit!
                      current=current.getNextChild();
                    }else{
                      current=null;
                    }
                  }
                }
              }
            }else{
              current=null;
            }
          }
        }
      }else{
        current=null;
      }
    }

    totalNumberOfNodes += numberOfNodesAdded;
    if(numberHasBeenAdded){
      iArray = new int[layer+1];
      iArray[layer]=number;
      for(int i = (layer-1); i>=0; i--){
        iArray[i]=current.data;
        current=current.parent;
      }
    }
    return numberHasBeenAdded;
  }



  void calcTotal(int number, int typeOfOutput) throws Exception{
    PowerTree powerTree = new PowerTree();
    PowerTreeNode temp;
    iArray= new int[125];
    int iposition=0;
    int i = 1;

    int layer;
    int maximumLength = maximumChainLength(number);
    temp = powerTree.addRoot(i);
    layer=1;
    boolean numberHasBeenAdded = false;
    totalNumberOfNodes=0;

    while(!numberHasBeenAdded){
      numberHasBeenAdded=addTotalLayer(number, layer, powerTree, numberHasBeenAdded,
maximumLength);
      layer++;
    }
    if(typeOfOutput=='1'){
      scrnOut.print(totalNumberOfNodes+ " ");
    }else{
      fileOut.print(totalNumberOfNodes+ " ");
    }
    printResult(layer, number, typeOfOutput);
  }

  void totalMethod(int typeOfOutput, String explanation) throws Exception {
    int beginNumber, endNumber;

    scrnOut.println("Enter a number between 0 and 2147483647 and press enter. \nThis program will
first build a total tree (using certain pruning rules), starting with a root of 1 and up to the provided
number.\n The result will be given as a final addition chain for the entered number\n NOTE: for a large
number, the calculations might take a while and may run out of memory space.");

    while (!in.eof()) {
      if(!Character.isDigit(in.nextChar())){
          nonIntegerInput();
        }
        if(!in.eoln()){
          beginNumber=checkReadInt();
          if(beginNumber==0){
            return;
          }
          if(in.nextChar()!='-'){
            calcTotal(beginNumber, typeOfOutput);
          }else{
            in.readChar();
            endNumber=checkReadInt();
            if(endNumber==0){
              return;
            }
            if(endNumber>=beginNumber){
              for(int i = beginNumber; i<=endNumber; i++){
              calcTotal(i, typeOfOutput);
              }
            }else{
              throw new Exception("Error: The ending number has to be bigger than the starting
number");
            }
          }
        }
      in.readln();
      scrnOut.print('>');
    }
  }

  void addLayer(int number, int layer, PowerTree powerTree, boolean[] isAlreadyPresent){

    int numberOfNodesAdded=0;
    int currentLayer=1;
    PowerTreeNode current;
    current=powerTree.returnRoot();
    while(currentLayer<layer){
      // get the leftmost child of the current layer. NOTE: this need not simply be obtained by following
the leftmost branch of each node, since some nodes may contain 0 children
      while(current.numberOfChildren>0){
        current = current.getLeftMostChild();
        currentLayer++;
      }
      if(currentLayer!=layer){//we have branched down the leftmostbranche, but this does not get us at
the bottom layer... So, find the leftmostchild at current layer
        if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of current child
has other kids to currents right
          current = current.getNextChild();
        }else{//we need to go back up the tree to find the next node that has unexplored kids

while((current.currentChildNumber==(current.parent.numberOfChildren))&&current.parent!=null){ //as
long as this child is its parent rightmost kid (and there are still parents to explore...)
            current=current.parent; //go one level up the tree
            currentLayer--;
          }
          if(current.parent!=null){//there are yet more nodes to visit!
            current=current.getNextChild();
          }else{
            current=null;
          }
        }
      }
    }
    while(current!=null){
      //add children to current node, and find next node to add children to
      PowerTreeNode temporary=current;
      for(int j = 1; j<=layer;j++){
        temporary=current;
        for(int k = layer; k>j; k--){ //start at the root, and add kids going down the line towards the current
node.
          temporary=temporary.parent;
        }

        int valueToAdd = current.data + temporary.data;
        if(!isAlreadyPresent[valueToAdd]){
          current.addChild((current.data)+(temporary.data),current, layer);
          numberOfNodesAdded++;
          isAlreadyPresent[valueToAdd]=true;
        }
      }
      //get the next node in the current layer, for which we need to add children. Set current to null if no
such node can be found
      if(current.parent!=null){
        if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of current child
has other kids to currents right
          current = current.getNextChild();

        }else{//we need to go back up the tree to find the next node that has unexplored kids
          while(current.parent!=null&&
(current.currentChildNumber==(current.parent.numberOfChildren))){ //as long as this child is its parent
rightmost kid (and there are still parents to explore...)
            current=current.parent; //go one level up the tree
            currentLayer--;
          }
          if(current.parent!=null){//there are yet more nodes to visit!
            current=current.getNextChild();
            //now, we 'only' need to find the 'leftmostnode'
            while(currentLayer<layer&&current!=null){
              // get the leftmost child of the current layer. NOTE: this need not simply be obtained by
following the leftmost branch of each node, since some nodes may contain 0 children
              while(current.numberOfChildren>0){
                current = current.getLeftMostChild();
                currentLayer++;
              }
              if(currentLayer!=layer){//we have branched down the leftmostbranche, but this does not
get us at the bottom layer... So, find the leftmostchild at current layer
                if(current.currentChildNumber<(current.parent.numberOfChildren)){//the parent of
current child has other kids to currents right
                    current = current.getNextChild();
                }else{//we need to go back up the tree to find the next node that has unexplored kids

while(current.parent!=null&&(current.currentChildNumber==(current.parent.numberOfChildren))){ //as
long as this child is its parent rightmost kid (and there are still parents to explore...)
                      current=current.parent; //go one level up the tree
                      currentLayer--;
                    }
                    if(current.parent!=null){//there are yet more nodes to visit!
                      current=current.getNextChild();
                    }else{
                      current=null;
                    }
                }
              }
            }
          }else{
            current=null;
          }
        }

      }else{
        current=null;
      }
    }
  }



  void calcPower(int number, int typeOfOutput) throws Exception{
    PowerTree powerTree = new PowerTree();
    iArray= new int[125];
    int iposition=0;
    int i = 1;
    boolean[] isAlreadyPresent;
    isAlreadyPresent = new boolean[(int)(Math.pow(2,number-1))+1];
    int valueToBeAdded;
    int layer;
    powerTree.addRoot(i);

    layer=1;

    boolean needToContinue = true, goalHasBeenAdded=false;

    while(layer<number){
      addLayer(number, layer, powerTree,isAlreadyPresent);
      layer++;
    }
    printPowerResult(typeOfOutput, layer, powerTree);
  }

  void powerMethod(int typeOfOutput, String explanation) throws Exception {
    int endNumber;

    scrnOut.println("Enter a layer between 1 and 25 and press enter. \nThis program will construct a
power tree, starting with a root of 1 and up the layer provided.\n NOTE: for a large layer, the
calculations might take a while and may run out of memory space.");

    if(!Character.isDigit(in.nextChar())){
      nonIntegerInput();
    }


      endNumber=checkReadInt();
      if(endNumber==0){
        return;
      }
      if(endNumber>30){
        throw new Exception("Provided layer is too large.");
      }
      calcPower(endNumber,typeOfOutput);
  }

  int determineWindows(int number, int windowSize,int[][] windows, int[] lowerHalf){
    int positionInCArray=0;
    cArray = (Integer.toBinaryString(number)).toCharArray();
    int sizeCurrentWindow=0, valueCurrentWindow=0;
    int numberOfZeroes=0;
    int positionInLowerHalf=0;

    while(positionInCArray<cArray.length){//determine the windows
      sizeCurrentWindow=0;
      valueCurrentWindow=0;

      while(positionInCArray<cArray.length && sizeCurrentWindow<windowSize){

        while(positionInCArray<cArray.length && cArray[positionInCArray]=='0' &&
sizeCurrentWindow==0){ //skip nullen aan begin van window
          positionInCArray++;
          numberOfZeroes++;
        }

        if(positionInCArray<cArray.length){
          if(cArray[positionInCArray]=='0'){//for a zero, increase the size of the window by one

            sizeCurrentWindow++;
            positionInCArray++;
            numberOfZeroes++;
          }else{
            if(cArray[positionInCArray]=='1'){//for a one, increase the size of a window by one, and
increase the value of the window accordingly, corresponding to the number of skipped zeroes.
              while(numberOfZeroes>0){
                valueCurrentWindow= 2*valueCurrentWindow;
                numberOfZeroes--;
              }
              sizeCurrentWindow++;
              valueCurrentWindow = 2*valueCurrentWindow +1;
              positionInCArray++;
            }
          }
        }
      }
      if(valueCurrentWindow!=0){ //vangt af dat laatste window 0 is

        windows[0][positionInWindows]=valueCurrentWindow; //vult windows[][] met de waarde en
positie van de verschillende windows.
        windows[1][positionInWindows]=positionInCArray-numberOfZeroes;

        positionInWindows++;
        positionInLowerHalf=0;

        while(valueCurrentWindow>lowerHalf[positionInLowerHalf] &&
positionInLowerHalf<maxPositionInLowerHalf){
          positionInLowerHalf++;
        }

        for(int j = maxPositionInLowerHalf; j>positionInLowerHalf;j--){
          lowerHalf[j] = lowerHalf[j-1];
        }

        lowerHalf[positionInLowerHalf]=valueCurrentWindow;
        maxPositionInLowerHalf++;
      }else{
      }
    }
    for(int z=0; z<maxPositionInLowerHalf;z++){
      }
    return numberOfZeroes;
  }

  int calcWindowSequence(int positionInIArray, int[] lowerHalf){
    int i;

    boolean additivesFound=false;
    int a,b;
    int lastNumberNotChecked=0;
    positionInIArray=0;

    //1.   First, build a pseudo-chain, containing the numbers 1, 2 and all of the goals
    //2.   Start at the right end of this chain-under-construction
    //3.   Now, check whether adding two other numbers already present in the sequence add up to create this number
    //4.   If not, an extra element needs to be added to the chain, in such a way that now the current element can be created by addition of two other numbers. In order to do so, take the current number and subtract the integer to its left from it. Add the result to the addition sequence.
    //5.   Go one element to the left in the chain and repeat steps 3 and 4 until the leftmost number of the addition sequence (1) is reached.

    i=1;
    iArray = new int[125];
    iArray[positionInIArray]=i;//zet 1 op eerste position array
    positionInIArray++;

    i=2;
    iArray[positionInIArray]=i;//zet 2 op tweede position array
    positionInIArray++;

    //bouwt addition chain voor de windows zelf
    for(int j=0; j<maxPositionInLowerHalf;j++){ // step 1: Build a pseudo-chain, containing the numbers 1, 2 and all of the goals
      if(lowerHalf[j]!=iArray[positionInIArray-1] && lowerHalf[j]!=1){ //if the current number is not already present in the chain, then add it
        iArray[positionInIArray]=lowerHalf[j];
        positionInIArray++;
      }
    }
    lastNumberNotChecked=positionInIArray-1; //step 2 start at the right end of the chain under construction
    while(lastNumberNotChecked>0){//as long as there are still members of the current chain which we might not be able to add using two other members of the chain
      //Check whether adding two other numbers already present in the sequence add up to create this number!
      a=0;
      b=0;
      additivesFound=false;
      for(a = lastNumberNotChecked;(a>=0&&(!additivesFound));a--){
        for(b = lastNumberNotChecked;(b>=0&&!additivesFound);b--){
          if(iArray[lastNumberNotChecked]==(iArray[a]+iArray[b])){
            additivesFound=true;
          }
        }
      }
      if(additivesFound){//need not to decrease lastNumberNotChecked when no additive is found, since then a number is added before our current lastNumberNotChecked, so we still need to check the same amount of number
        lastNumberNotChecked--;
      }else{// now we need to add an element to the chain, in such a way that we can add up to iArray[lastNumberNotChecked]
        if(iArray[lastNumberNotChecked]>(3*iArray[lastNumberNotChecked-1])){//this extra check is needed, since after applying the backwards binary algorith, it might happen that additives will be found. Therefore, one cannot automatically continue with the code in the elseblock, but needs to check on additivesfound first
          while(iArray[lastNumberNotChecked]>(3*iArray[lastNumberNotChecked-1])){//as long as the current number is larger than 3*the previous element of the chain, apply the backwards binary algorithm
            //apply backwards binary algorithm
            for(int k = (positionInIArray-1);k>=lastNumberNotChecked;k--){ //shift all elements in the chain larger than the current one  one element to the right
              iArray[k+1] = iArray[k];
```

```
            }
            if(((int)iArray[lastNumberNotChecked]%2)!=0){//add the approriate next element
              iArray[lastNumberNotChecked]=(iArray[lastNumberNotChecked]-1);
            }else{
              iArray[lastNumberNotChecked]=(iArray[lastNumberNotChecked]/2);
            }
            positionInIArray++;
          }
        }else{

          int k =(positionInIArray-1);
          int numberToBeAdded = (iArray[lastNumberNotChecked]-iArray[lastNumberNotChecked-1]);
          while(iArray[k]>(numberToBeAdded)){
            iArray[k+1] = iArray[k];
            k--;
          }


          iArray[k+1]= (numberToBeAdded);
          positionInIArray++;

        }
      }

    }
    return positionInIArray;
  }
  int calcWindow(int number, int windowSize){
    int[] lowerHalf = new int[999];
    int[][] windows= new int[2][998];
    int positionInIArray=0;

    int numberOfZeroes;
    int realHighestNumber=0;
    int lastNumber=0;

    positionInWindows=0;
    maxPositionInLowerHalf=0;

    numberOfZeroes=determineWindows(number, windowSize, windows, lowerHalf);
    positionInIArray=calcWindowSequence(positionInIArray, lowerHalf);

    realHighestNumber = (positionInIArray-1);
    positionInIArray=0;

    while(windows[0][0]!=iArray[positionInIArray]){ // zoek de waarde van het eerste window op in de
keten tot nu toe
      positionInIArray++;
    }

    lastNumber=iArray[positionInIArray];
    for(int j = 0; j<positionInWindows;j++){ //voor elk window
      for(int l = 0; l<(windows[1][j+1]-windows[1][j]);l++){// voor zoveel stapjes als er tussen de twee
windows zit
        while(positionInIArray<= realHighestNumber && 2*lastNumber>iArray[positionInIArray]){
          positionInIArray++;
        }

        if(2*lastNumber < iArray[positionInIArray]){ // schuif alle ints in iarray die groter zijn dan de
huidige toe te voegen int 1 positie naar rechts in de array
          for(int k = realHighestNumber;k>=positionInIArray;k--){
            iArray[k+1] = iArray[k];
          }
          iArray[positionInIArray]=lastNumber=2*lastNumber;
          positionInIArray++;
          realHighestNumber++;
        }else{
          if(2*lastNumber==iArray[positionInIArray]){ //als het toe te voegen nummer al in de array staat,
voeg het dan niet nog een keer toe
            positionInIArray++;
            lastNumber=2*lastNumber;
          }
          else{
            if(2*lastNumber>iArray[positionInIArray]){ //anders: voeg het aan het einde toe
              iArray[positionInIArray]=lastNumber=2*lastNumber;
              positionInIArray++;
              realHighestNumber++;
            }
          }

        }
      }

      if(j<(positionInWindows-1)){ //als niet het laatste window: voeg de waarde van het window toe
        iArray[positionInIArray] = lastNumber=lastNumber+windows[0][j+1];
        positionInIArray++;
        realHighestNumber++;
      }
    }
    if((windows[1][1]==0)){ //if second window == 0, i.o.w. if there is only one window, increase the
position in IArray by one
      if(windows[0][0]==1){
        realHighestNumber--;
      }
      positionInIArray++;
    }
    while(numberOfZeroes>0){
      iArray[positionInIArray] = lastNumber=2*lastNumber;
      realHighestNumber++;
      positionInIArray++;
      numberOfZeroes--;
    }
    return realHighestNumber+1;

    }

  void windowMethod(int typeOfOutput, String explanation) throws Exception {
    int beginNumber, endNumber;
    int length;
    scrnOut.println("Please specifiy the windowsize. Typical window values are small primes. \nThis
program will calculate an estimate of the addition chain based on the window method.");

      if(!Character.isDigit(in.nextChar())){
        nonIntegerInput();
      }
      int windowSize = checkReadInt();
        if(windowSize==0){
          return;
        }
      in.readln();

      scrnOut.println(explanation);
      while(!in.eof()){
        if(!Character.isDigit(in.nextChar())){
          nonIntegerInput();
        }
        while(!in.eoln()){
          beginNumber=checkReadInt();
          if(beginNumber==0){
            return;
          }
          if(in.nextChar()!='-'){
            if(beginNumber==1){ //check whether goal number is 1, if so, add 1 to the chain and skip the
rest of the calculations
              iArray[0]=1;//zet 1 op eerste position array
              printResult(1, beginNumber, typeOfOutput);
            }else{
              length = calcWindow(beginNumber, windowSize);
              printResult(length, beginNumber, typeOfOutput);
            }
          }else{
            in.readChar();
            endNumber= checkReadInt();
            if(endNumber==0){
              return;
            }
            if(endNumber>=beginNumber){
              for(int i = beginNumber; i<=endNumber; i++){
                if(i==1){ //check whether goal number is 1, if so, add 1 to the chain and skip the rest of
the calculations
                  iArray[0]=1;//zet 1 op eerste position array
                  printResult(1, i, typeOfOutput);
                }else{
                  length=calcWindow(i, windowSize);
                  printResult(length, i, typeOfOutput);
                }
              }
            }else{
              throw new Exception("Error: The ending number has to be bigger than the starting
number");
            }

          }
        }
        in.readln();
        scrnOut.print('>');
          }
  }


  int buildFactorChain(TreeNode node, int current, boolean originalRight){
    // this somewhat complex method builds the chain corresponding to the tree built in calcfactor. It
roughly works as follows. The methods uses recursion to walk through the tree.
    //When the right child of current node is 2, a doubling step is used. when this right child is larger
then 2, a recursive call is used, with the right child as the new current node.
    // When the right child is 1, the value of the current node is replaced by the value of 'current': this
value is hereby put in memory, so it can be added lateron.
    //Now, the left child is looked at. Once again, if it is 2, a doubling step is taken, when bigger than two
a recursive call follows. Afterwards, as long as the root is not reached,
    // all the values that 'have been put in memory' are added.
    //             5
    //            /  \
    //       4    1
    //       /  \
    //      2    2
    //    / \  / \
    //   1  1  1  1
    if(node.right.data == 2){ //if the data of the node to the right equals two, then a doubling step should
be used in the addition chain
      current = 2*current;
      iArray[pointer]=current;
      pointer++;

    }else{
      if(node.right.data!=1){ //otherwise, use recursion when the data of the current node is larger then 2
        current=buildFactorChain(node.right, current, originalRight );
      }
    }

    node.data=current; //overwrite the value of the current node, thereby putting it in memory, so it can
be added lateron when it is needed
    if(node.left.data==2){ //if now the left child is two, use a doubling step again
      current = 2*current;
      iArray[pointer]=current;
      pointer++;
    } else {
      if(node.left.data!=1){ //if the left child is larger then 2, use a recursive step.
        current=buildFactorChain(node.left, current,originalRight);
      }
    }

    if(node.parent != null && node.right.data == 1){// as long as the root is not reached, check to see
whether the current node has 'remembered' data that needs to be added.
      // if so, add this value to the right of the chain
      current = current + node.data;
      iArray[pointer] = current;
      pointer++;
    } else {
      if(node.parent == null && originalRight){
        current = current + node.data;
        iArray[pointer] = current;
        pointer++;
      }
    }
    return current;
  }

  int buildFactorTree(int number){
    //factor method: first build a tree, by adding the smallest devider to the right and the resting value to
the left, halting at 1. For example:
    //              5
    //             / \
    //         4    1
    //        / \
    //       2    2
    //      / \  / \
    //     1  1  1  1

    int smallestDevider = 1;
    int trySmallestDevider=2;
    int valueResting=number;
    int currentValueResting=number;
    boolean smallestDeviderFound=false;

    tree = new BinaryTree();
    TreeNode temporary=tree.addRoot(number);
    if(number==1){
      int current = 1;
      iArray[pointer]=1;
      pointer++;
  }else{
   while(valueResting!=1&& !temporary.isFulfilled){
    currentValueResting=valueResting;

    while(currentValueResting!=1){ //While the remaining value is bigger than 1, the stopping
condition
     smallestDeviderFound=false;
     trySmallestDevider=2;
     while(!smallestDeviderFound&& valueResting/trySmallestDevider>=trySmallestDevider ){
//Look for a smallest devider
      if((valueResting % trySmallestDevider)==0){
       smallestDeviderFound=true;
      }else{//when it is not found, increment the 'trySmallestDevider' for the next run. Even
numbers apart from two are excluded, since they cannot be a smallest devider
       if(trySmallestDevider==2){
        trySmallestDevider++;
       }
       else{
        trySmallestDevider+=2;
       }
      }
     }

     if(!smallestDeviderFound){ //the current number is a prime
      smallestDevider=1;
     }else{
      smallestDevider=trySmallestDevider;
     }

     tree.addRight(smallestDevider,temporary); //add the smallest devider to the right
     currentValueResting=smallestDevider;
     if(smallestDevider==1){//and the remaining value to the left. Set a value of isFulfilled to true
when the number to be added equals 1
      valueResting=valueResting-smallestDevider;
      temporary.right.isFulfilled=true;
     }else{
      valueResting=valueResting/smallestDevider;
     }

     tree.addLeft(valueResting, temporary);

     if(valueResting==1){
      temporary.left.isFulfilled=true;
     }
     temporary=temporary.right; //preparations for the next run
     valueResting=temporary.data;
    }

    while(temporary.parent!=null &&
(temporary.data==1||((temporary.left.data==1||temporary.left.isFulfilled)&&(temporary.right.data==1||te
mporary.right.isFulfilled)))){//Check whether the current data is zero or all of the leafs that can be
reached from the current node are. When so, indicate that this node can be set to fulfilled
     temporary.isFulfilled=true;
     temporary=temporary.parent;
    }
    temporary=temporary.left;
    valueResting=temporary.data;
   }

   temporary = tree.returnRoot();
   pointer=0;

   int current = 1;
```

```
      iArray[pointer]=1;
      pointer++;
      boolean originalRight= (temporary.right.data==1);

      //after the tree has been set up, the corresponding addition chain needs to be calculated
      buildFactorChain(temporary, current, originalRight);
    }
    return pointer;
  }

  void factorMethod(int typeOfOutput, String explanation) throws Exception {
    int beginNumber, endNumber;
    int length;
    scrnOut.println(explanation + " This program will calculate an estimate of the addition chain based
on the factor method");

    while(!in.eof()){
        if(!Character.isDigit(in.nextChar())){
          nonIntegerInput();
        }
        while(!in.eoln()){
          beginNumber=checkReadInt();
          if(beginNumber==0){
            return;
          }
          if(in.nextChar()!='-'){
            length = buildFactorTree(beginNumber);
            printResult(length, beginNumber, typeOfOutput);

          }else{
            in.readChar();
            endNumber=checkReadInt();
            if(endNumber==0){
              return;
            }
            if(endNumber>=beginNumber){
              for(int i = beginNumber; i<=endNumber; i++){
                length = buildFactorTree(i);
                printResult(length, i, typeOfOutput);

              }
            }else{
              throw new Exception("Error: The ending number has to be bigger than the starting
number");
            }

          }
        }
        in.readln();
        scrnOut.print('>');
          }
  }

  int calcDoubling(int number) throws Exception {
    //this method is relatively easy: just add the largest number possible. This method gives chains of
equal length to the binary method.
    int i = 1;
    int iposition=0, localHighestNumber;

    iArray= new int[125];
    iArray[iposition]=i;
    iposition++;
    localHighestNumber=0;
```

```
      while(i!=number){
         if(i+iArray[localHighestNumber] <= number){
            i = i+ iArray[localHighestNumber];
            iArray[iposition]=i;
            iposition++;
            localHighestNumber++;
         }
         else{ localHighestNumber--;
         }
      }

      return iposition;
   }

   void doublingMethod(int typeOfOutput, String explanation) throws Exception {
      int beginNumber, endNumber;
      int length;
      scrnOut.println(explanation + " This program will calculate an estimate of the addition chain based
on the doubling method");

      while (!in.eof()) {
         if(!Character.isDigit(in.nextChar())){
               nonIntegerInput();
            }
            if(!in.eoln()){
               beginNumber=checkReadInt();
               if(beginNumber==0){
                  return;
               }
               if(in.nextChar()!='-'){
                  length = calcDoubling(beginNumber);
                  printResult(length, beginNumber, typeOfOutput);
               }else{
                  in.readChar();
                  endNumber=checkReadInt();
                  if(endNumber==0){
                     return;
                  }
                  if(endNumber>=beginNumber){
                     for(int i = beginNumber; i<=endNumber; i++){
                     length = calcDoubling(i);
                     printResult(length, i, typeOfOutput);
                     }
                  }else{
                     throw new Exception("Error: The ending number has to be bigger than the starting
number");
                  }
               }
            }
         in.readln();
         scrnOut.print('>');
      }
   }

   int calcBinary(int number) throws Exception{
      cArray = (Integer.toBinaryString(number)).toCharArray();

      iArray= new int[125];
      int i = 1;
      int whereInString = 1;
      int iposition=0;

      iArray[iposition] = i;
```

```
    iposition++;

    while(whereInString<cArray.length){
      //doubling for each bit
      i = 2*i;
      iArray[iposition] = i;
      iposition++;

      //add 1 for each bit set to '1'
      if (cArray[whereInString] == '1'){
        i++;
        iArray[iposition] = i;
        iposition++;
      }

      whereInString++;
    }
    return iposition;
  }



  void binaryMethod(int typeOfOutput, String explanation) throws Exception {
    int beginNumber, endNumber;
    int length;
    scrnOut.println(explanation + "\nThis program will calculate an estimate of the addition chain based
on the binary method");

    while(!in.eof()) {
        if(!Character.isDigit(in.nextChar())){
          nonIntegerInput();
        }

        if(!in.eoln()){

          beginNumber=checkReadInt();
          if(beginNumber==0){
            return;
          }
          if(in.nextChar()!='-'){
            length = calcBinary(beginNumber);
            printResult(length,beginNumber, typeOfOutput);
          }else{
            in.readChar();
            endNumber=checkReadInt();
            if(endNumber==0){
              return;
            }

            if(endNumber>=beginNumber){
              for(int i = beginNumber; i<=endNumber; i++){
                length = calcBinary(i);
                printResult(length,i, typeOfOutput);
              }
            }else{
              throw new Exception("Error: The ending number has to be bigger than the starting
number");
            }
          }
        }
      in.readln();
      scrnOut.print('>');
    }
  }

  void processInput(int typeOfOutput, int typeOfCalc) throws Exception{

    String explanation = "Enter a number between 0 and 2147483647 and press enter. \nWhen you
want to calculate the addition chain for multiple nummers at once, enter them with a '-' in between.
Example: 3-20.\n Since these calculations require quite some memory capacity, you can run 'java -
Xmx...M AdditionChains' to increase the memory capacity for the java program.";

    switch(typeOfCalc){

      case '1': //Binary method
      binaryMethod(typeOfOutput, explanation);
      break;


      case '2': //Doubling method
      doublingMethod(typeOfOutput, explanation);
      break;

      case '3': //factor method
      factorMethod(typeOfOutput, explanation);
      break;

      case '4'://window method
      windowMethod(typeOfOutput, explanation);
      break;

      case '5'://power method
      powerMethod(typeOfOutput, explanation);
      break;

      case '6'://total method
      totalMethod(typeOfOutput, explanation);
      break;

      case '7'://combined depth first/breadth first-method
      combinedMethod(typeOfOutput, explanation);
      break;

      case '8'://depthfirst method
      depthfirstMethod(typeOfOutput, explanation);
      break;
      case '9':
      return;

      case '0':
      System.exit(1);

      default:
      throw new Exception(inputError);

    }

    in.readln();
  }

  int readCalculationType() throws Exception {
    scrnOut.println("Please select your way of calculation");
    scrnOut.println("1. Binary method");
    scrnOut.println("2. Doubling method");
    scrnOut.println("3. Factor method");
    scrnOut.println("4. Window method");
    scrnOut.println("5. Power Tree method");
    scrnOut.println("6. Breadth first total method");
    scrnOut.println("7. Combined depth first/breadth first total method");
    scrnOut.println("8. Depth first total method");
    scrnOut.println();
    scrnOut.println("9. Return to start of program");
    scrnOut.println();
    scrnOut.println();
    scrnOut.println("q. Exit");

    int typeOfCalculation = in.readChar();
    in.readln();
    if(typeOfCalculation == 'q'){
      System.exit(1);
    }

    return typeOfCalculation;
  }

  int readOutputType() throws Exception {
    scrnOut.println("Please select the desired way of output");
    scrnOut.println("1. To screen");
    scrnOut.println("2. To file");
    scrnOut.println();
    scrnOut.println("q. Exit");
    int output = in.readChar();
    in.readln();
    if(output=='1'){
      scrnOut = new Output();
    } else {
      if(output=='2'){
        scrnOut.println("please enter filename");
        String fileName = new String();
        while(!in.eoln()){
          fileName += in.readChar();
        }
        fileOut = new Output(fileName);
        in.readln();
      }else{
        if(output=='q'){
          System.exit(0);
        }else{
          throw new Exception(inputError);
        }
      }
    }

    return output;
  }

  void readInput() throws Exception {
    int typeOfOutput = readOutputType();
     int typeOfCalculation = readCalculationType();

    if(typeOfCalculation == '9'){
      return;
    }
    processInput(typeOfOutput,typeOfCalculation);
  }

  void start() {
    try {
      while(true){ //loop, until exit has been chosen
```

```
            readInput();
         }

      }
      catch(Exception e) {
         scrnOut.println("\nError: " + e + "\n");
         start();
      }

   }


   public static void main( String[] argv ) {
      new AdditionChains().start();
   }
}
```

III.b: BinaryTree.java

```
/**  @author
  Sander van der Kruyssen
**/
public class BinaryTree {
  private TreeNode root;
  private TreeNode temporaryNode;


  public BinaryTree() {
    init();
  }

  public BinaryTree init() {
    root = null;

    return this;
  }

  public TreeNode addRoot(int data) {
    if(root == null) {
      root = new TreeNode(data);
    }

    return root;
  }

  public TreeNode addLeft(int data, TreeNode parent) {
    temporaryNode = addLeft(parent, data);

    return temporaryNode;
  }

  public TreeNode addRight(int data, TreeNode parent) {
    temporaryNode = addRight(parent, data);

    return temporaryNode;
  }
  public TreeNode addRight(int data) {
    temporaryNode = addRight(root, data);

    return temporaryNode;
  }
  public TreeNode addLeft(int data) {
    temporaryNode = addLeft(root, data);

    return temporaryNode;
  }

  private TreeNode addLeft(TreeNode parent, int data) {
    parent.left = new TreeNode(data);
    parent.left.parent = parent;
    return parent.left;
  }

  private TreeNode addRight(TreeNode parent, int data) {
    parent.right = new TreeNode(data);
    parent.right.parent=parent;
    return parent.right;
  }
```

```
    public TreeNode returnRoot(){
      return root;
    }

    public int numberOfNodes() {
      return numberOfNodes(root);
    }

    private int numberOfNodes(TreeNode root) {
      if(root == null) {
        return 0;
      }

      return 1 + numberOfNodes(root.left) + numberOfNodes(root.right);
    }

    public void isFulfilled(TreeNode node){
      node.isFulfilled=true;
    }

    public int depth() {
      return depth(root);
    }

    private int depth(TreeNode root) {
      if(root == null) {
        return 1;
      }

      return 1 + Math.max(depth(root.left), depth(root.right));
    }

    private TreeNode clone(TreeNode root) {
      if(root == null) {
        return null;
      }
      return new TreeNode(root.data, clone(root.parent), clone(root.left), clone(root.right), root.isFulfilled);
    }

    public Object clone() {
      return clone(root);
    }
  }
```

III.c: TreeNode.java

```
/**  @author
  Sander van der Kruyssen
**/
public class TreeNode implements Cloneable {

  int data;
  boolean isFulfilled=false;
  TreeNode left,
      right,
      parent;

  public TreeNode(int data) {
    this(data, null, null, null, false);
  }

  public TreeNode(int data, TreeNode parent) {
    this(data, null, null, parent, false);
  }

  public TreeNode(int data, TreeNode parent, TreeNode left, TreeNode right, boolean isFulfilled) {
    this.data = data;
    this.left = left;
    this.right = right;
    this.parent = parent;
    this.isFulfilled = isFulfilled;
  }

  public TreeNode getParent(TreeNode current) {
    return parent;
  }

  public Object clone() {
    TreeNode clone;

    try {
      clone = (TreeNode) super.clone();
    } catch (CloneNotSupportedException e) {
      throw new Error("Error! TreeNode is not Cloneable");
    }



    return clone;
  }
}
```

III.d: PowerTree.java

```
/**  @author
  Sander van der Kruyssen
**/
public class PowerTree {
  private PowerTreeNode root;
  private PowerTreeNode temporaryNode;


  public PowerTree() {
    init();
  }

  public PowerTree init() {
    root = null;

    return this;
  }

  public PowerTreeNode addRoot(int data) {
    if(root == null) {
      root = new PowerTreeNode(data);
    }

    return root;
  }

  public PowerTreeNode addChild(int data, PowerTreeNode parent, int layer) {
    temporaryNode = addChild(parent, data, layer);

    return temporaryNode;
  }


  private PowerTreeNode addChild(PowerTreeNode parent, int data, int layer) {
    temporaryNode = parent.addChild(data, parent, layer);
    return temporaryNode;
  }

  public PowerTreeNode returnRoot(){
    return root;
  }

  public int numberOfNodes() {
    return numberOfNodes(root,0);
  }

  private int numberOfNodes(PowerTreeNode root, int numberOfNodes) {
    if(root == null) {
      return 0;
    }
    for(int i = 0; i<=root.getNumberOfChildren();i++){
      numberOfNodes+=numberOfNodes(root.getChild(i),numberOfNodes);
    }
    return numberOfNodes;
  }
  public int numberOfChildren(PowerTreeNode node){
    return node.getNumberOfChildren();
  }
}
```

III.e: PowerTreeNode.java

```
/**  @author
  Sander van der Kruyssen
**/

public class PowerTreeNode {

  int data;
  short numberOfChildren;
  short currentChildNumber;
  short maxNumberOfChildren = 255;//TODO: find proper values for arraysizes
  PowerTreeNode[] children;
  PowerTreeNode parent;
  byte layer;

  public PowerTreeNode(int data) {
    this(data, null, null, 0, 0, 0);
  }

  public PowerTreeNode(int data, PowerTreeNode parent) {
    this(data, parent, null, 0, 0, 0);
  }

  public PowerTreeNode(int data, PowerTreeNode parent, PowerTreeNode[] children2, int
numberOfChildren, int currentChildNumber, int layer) {
    this.data = data;
    this.parent = parent;
    this.children = children2;
    if(this.children==null){
      this.children=new PowerTreeNode[maxNumberOfChildren];
    }
    this.numberOfChildren =(short) numberOfChildren;
    this.currentChildNumber= (short)currentChildNumber;
    this.layer=(byte)layer;
  }

  public PowerTreeNode getParent(PowerTreeNode current) {
    return parent;
  }

  public PowerTreeNode addChild(int child, PowerTreeNode parent, int layer){
    PowerTreeNode newNode = new PowerTreeNode(child, parent);
    children[numberOfChildren]= newNode;
    newNode.parent=parent;

    numberOfChildren++;
    newNode.currentChildNumber=numberOfChildren;
    newNode.layer=(byte)layer;
    return newNode;
  }

  public PowerTreeNode getChild(int child){
    return children[child];
  }

  public PowerTreeNode getLeftMostChild(){
    return children[0];
  }
  public PowerTreeNode getNextChild(){
    return parent.children[currentChildNumber];
  }

  public int getNumberOfChildren(){
    return numberOfChildren;
  }


}
```