# Induction on Number Series

## Project Report

**Fakultät für Wirtschaftsinformatik und Angewandte Informatik**
**Otto-Friedrich-Universität Bamberg**

**Lehrstuhl für Kognitive Systeme**
**Betreuer: Prof. Dr. Ute Schmid**

**Sommersemester 2012**

Roland Grünwald
Elke Heidel
Alexander Strätz
Michael Sünkel
Robert Terbach

# Contents


**1. Introduction** **1**

**2. Human Inductive Reasoning** **2**

2.1. Intelligence and Intelligence Tests . . . . . 2
2.2. A Cognitive Model . . . . . 3
2.3. Cognitive Models for Solving Number Series . . . . . 4

**3. Creating Number Series** **5**

3.1. Complexity Classes . . . . . 5
3.1.1. Operational Complexity . . . . . 5
3.1.2. Numerical Complexity . . . . . 5
3.1.3. Structural Complexity . . . . . 5
3.1.4. Results . . . . . 6

**4. Inductive Programming** **9**

4.1. MagicHaskeller . . . . . 9
4.2. Solving Number Series in MagicHaskeller . . . . . 9
4.2.1. Number Series Generation with Haskell as Preliminary Work . . . . . 10
4.2.2. The Final MagicHaskeller Program . . . . . 12
4.3. Results and Observations . . . . . 12
4.3.1. General . . . . . 12
4.3.2. High Number Problem . . . . . 13
4.3.3. Optimization . . . . . 15
4.4. Conclusion . . . . . 15
4.4.1. RunAnalytical . . . . . 15
4.4.2. Generate-and-Test . . . . . 15

**5. Experiment** **17**

5.1. Method . . . . . 17
5.2. Results . . . . . 17
5.3. Time-Error-Rate . . . . . 22
5.4. Discussion . . . . . 25

**6. Conclusion** **26**

**A. Appendix** **28**

A.1. Calculation of the Mental Age . . . . . 28
A.2. A Taxonomy of Rules in the Raven Test . . . . . 28

A.3. About non-unique solvability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
A.4. E-mail from Susumu Katayama regarding RunAnalytical . . . . . . . . . . . . . . 29
A.5. Questionnaire . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
A.6. Source Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

# 1. Introduction

Considering the mathematical background of the topic, a general definition of a sequence of any number system(i.e. natural, integers, rational, real, complex) can be given as follows[Int7]: $\sum_{n=0}^{\infty} a_n = a_0 + a_1 + a_2 + \cdots$

According to this definition, every number series of any kind can be rewritten as a sum of infinite length, therefore containing infinitely many sequences of partial sums.
Considering the observation that any number system can also be formulated according to the defintion given above, one obvious approach to the computation of arbitrary series within one number system would be a continuous function $f$ manipulating any parameter $k$, the random seed of a number system through a randomly generated operation on $k$ according to the underlying algebraic laws of that system. As there are infinitely many number sequences, there is also an attempt for a classification according to its characteristics. The website http://oeis.org/ featuring an online search-engine is a good example for that. Theoretical use of certain number series can be found in almost all fields of science, applied mathematics, computer science, physics and engineering being the most committed.

When it comes to practical application, there are only very few examples. One of these lies in the field of practical psychology, where sequences of integers are used in measuring human intelligence, e.g. as part of the multi-dimensional test I-S-T 2000R. The task is to correctly predict the next or second-next number for given integer-sequences of certain length(see Chapter 2.1). One problem regarding these sequences is, that there is no definite answer to the question of how human intelligence processes the given numbers in terms of inductive reasoning, culminating in a generalized regular pattern, i.e. a solution. In the course of the project within the field of cognitive modeling, the aim of our work is to indirectly solve this problem through a combination of an empirical study providing insight of the cognitive process during the activity of solving a set of pre-defined integer-series through an online task, thereby measuring time, level of difficulty and collecting information about the reasoning itself and, on the other hand, by means of computational modeling, using the up-to-date inductive programming system MagicHaskeller to construct an algorithm simulating the hypothetical inductive reasoning process and testing it with different series of classified complexity.
Finally, after statistical analysis of the empirical part, we will relatively compare the results of the empirical and the computational part in terms of 'level of complexity' and 'performance-time' to provide a conclusive perspective.

# 2. Human Inductive Reasoning

Reasoning is a thought process involving logic. Instead of making arbitrary decisions humans support their choices on observation which is more or less plausible. Inductive reasoning consists of inferring general principles or rules from specific facts. A good inductive reasoning is likely to be true. [Int1] [Int2]

## 2.1. Intelligence and Intelligence Tests

One definition of intelligence is:

- The capacity to acquire and apply knowledge.
- The faculty of thought and reason.
- Superior powers of mind. [Int3]

Although this ability is not visible or tangible, instruments have been developed which allow to measure intelligence as defined before. At the beginning of the last century the French psychologist Alfred Binet was commissioned by his government to develop an objective test which would weed out backward children in state schools, and thus save public money and avoid holding back the work of the class by teaching children who were incapable of learning of a given standard. The first intelligence test was born! Relatively at the same time another psychologist, Charles E. Spearman, compared people's achievements with the help of various performance tests and found out that those who were good in one test section tend to be good in others. That's why he assumed the following:

> "Under certain conditions the score of a person at a mental test can be divided into two factors, one of which is always the same in all tests, whereas the other varies from one test to another; the former is called the general factor or G, while the other is called the specific factor."

[Int4]

John C. Raven, a student of Spearman, inspired by his teacher, *"set about developing tests of the two components of g identified by Spearman."* [Int5] These two components are (1) the ability to think clearly and make sense of complexity, which is known as eductive ability (from the Latin root 'educere', meaning 'to draw out') and (2) the ability to store and reproduce information, known as reproductive ability. [Int6] He wanted to design a set of overlapping, homogenous tasks whose solutions claimed different skills. In 1938 he found a method: the today well known Raven Progressive Matrices. It is a non-verbal multiple-choice test

> "with items within a set becoming increasingly difficult, requiring ever greater cognitive capacity to encode and analyze information.[ . . .] In each test item, the subject is asked to

identify the missing element that completes a pattern. Many patterns are presented in the form of a 4x4, 3x3, or 2x2 matrix."[Int6]

The Matrices are available in three different forms: The Standard, the Colored and the Advanced Progressive Matrices. For more information about the making of the matrices please read: [Int6]

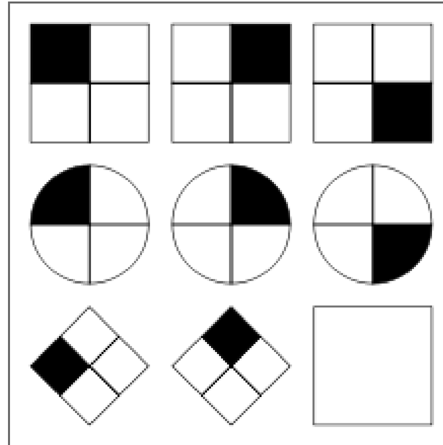

**Figure 2.1.** — This is an example of one kind of IQ test item, modeled after items in the Raven's Progressive Matrices test: [Int6]

Based on these observations psychologists developed global intelligence tests which are generally accepted and used. In 1953 the german psychologist Rudolf Amthauer developed the multidimensional 'Intelligence Structure Test'. Years later it was completed and improved by Liepmann, Beauducel, Brocke and Amthauer; the last change was made in 2000. The authors assume that intelligence is a complex construct characterized by a variety of sub-skills, and therefore cannot be a unit test to assess the intelligence; factors were mentioned from which one can conclude a certain degree of intelligence. As prime factors they distinguished the verbal, numerical and figural intelligence. These yield inductive reasoning. The second-order factor is the memory retention made up by (1) the fluid intelligence, which is reasoning without knowledge based on new information, and (2) the crystallized intelligence based on stored knowledge, which is dependent on culture and educational institutions.

## 2.2. A Cognitive Model

Due to empirical studies and their results we may say that Raven Tests are suited for measuring a certain kind of intelligence, as the way to reason about the solution, does not differ much from one person to another. That means the Raven-tests are built up similarly: the degree of difficulty increases monotonically from the simplest task to the very difficult ones. The question is: which psychological processes are responsible for the differences in people's ability to solve the task? At the Carnegie Mellon University the cognitive process of solving Raven Tests was analyzed and the experimental investigation led to the development of computer simulation models that tested the sufficiency of this analysis.

> "Two computer simulations, FAIRAVEN and BETTERAVEN express the differences between good and extremely good performance on the test. FAIRAVEN performs like the median college student; BETTERAVEN performs like one of the very best."[CarpEtAl90]

## 2.3. Cognitive Models for Solving Number Series

Impressed by the two programs developed by Carpenter, Just and Schell for Raven Matrices we aimed to develop one for number series. Known research done in this direction is by Burghard in 2005, and Ragni and Klein in 2011. Students of our university made it to their goal to analyze the human way(s) of solving number series in an empirical study and then implement these (in programs able to solve the given number series). Different approaches were taken: using Neuronal Networks, Generic Algorithms, or an inductive functional program system, the MagicHaskeller. To examine evaluable data people were asked to solve number series both in a time-constraint test gaining quantitative data and in a think-aloud test gaining qualitative data. The results are shortly mentioned here:

- "'Based on our data we propose the functionality of GAs as a cognitive model for the human strategy to solve number series because the procedure applied by humans resembles the steps a GA has."
- "...artificial neural networks are not yet comparable to the complexity of the human brain."
- "To sum up, solving number series with MagicHaskeller is possible. The program generation is powerful and can perform more efficiently by focusing on human constraints."

> "The series completion task requires correspondence finding, pairwise comparison of adjacent corresponding elements, and the induction of rules based on patterns of pairwise similarities and differences." [SK63]

In Klaus Korossy's opinion:

> "Models of sequence completion solution include four component processes: detection of relation, discovery of periodicity, completion of pattern description, and extrapolation, where the first three processes combine to generate a pattern description."[KK98]

(see more in the Appendix)

As stated by Korossy we have to be careful in chosing the series, as their solvability is not always unique. It means we can't say that more complex pattern descriptions make greater demands on working memory because subjects may detect easier ways to solve the series as it was built.

> "Thus, what one intelligence test measures, according to the current theory, is the common ability to decompose problems into manageable segments and iterate through them, the differential ability to manage the hierarchy if goals and subgoals generated by this problem decomposition, and the differential ability to form higher-level abstractions. " [CarpEtAl90]

# 3. Creating Number Series

## 3.1. Complexity Classes

### 3.1.1. Operational Complexity

When we created the number series, we ran into many problems. We got the idea that number series have different types of complexity. Quickly we figured out, we could vary in operational complexity, meaning addition and subtraction as simple operations and multiplication or division as more complex operations. We also have square as complex operations, but omitted higher potencies as of their fast growing results. We also came up with ideas for completely different operations like *digit sum*, but we had problems with generated series and therefore omitted this as well. We also restricted our number series to be monotonously increasing, so subtraction and division were not an option anymore.

### 3.1.2. Numerical Complexity

Orthogonal to the operational complexity described before, we could also vary numerical complexity. This means, quite simple, to have number series with *low* values, contrasting *high* values. The border cannot be given explicitly, because, when creating number series, we learned that it is easily possible to generate high numbers within few steps, even when beginning with one-digit numbers.
The idea behind numerical complexity is that humans cannot operate with numbers, let's say in the thousands and above, as easy as with the *times table*. So we decided *by instinct* if a series is numerically complex or not. This should not be a hurdle for the computer, of course.

### 3.1.3. Structural Complexity

Our third dimension of classification is *structural* complexity. We tried to find different series with the same generational structure. A reasonable number of classes would be five, we decided, so that in total we had 20 different series.
In the beginning we tried to increase the number of operators used on numbers and created several number series, but we had two problems with these: On the one hand, it is only possible to combine two different operators, e.g. multiplication and addition, both only once, otherwise they are multiplied out. The number series we created this way all reduced to a simple pattern, and therefore *fell* in complexity. The pattern reaches its maximum complexity with $a_(n-1)$ $op_1$ $var_1$ $op_2$ $var_2$. This does not allow for a sufficient variation of complexity to make up five

classes. On the other hand it is not possible to vary operational complexity anymore, as the pattern only allows for unique operators. If one defines more operators, this variation could be an option, but not in our arithmetic case.
In the next approach we tried to get rid of this issue. The first step was only allowing *one* operation at once (the above pattern could be an option for one complexity class, though). In the first class we simply use one, and always the same, operator on the previous number. In the second class we allowed the index (the position in the series) to have an influence on the operator. Class three is built with two alternating operators, or operands. Class four creates *fibonacci*-like series, where each number is based upon two predecessors. In the fifth class number series from class four are used as operands for the generation. One interesting observation we made was that class one is a subset of class three.
Of course other classes are possible, for example the rules of class two and three could be combined, other series could be fed into class five, or more different operations can be done in class three (we only give very few numbers to the testants so that this would get really hard to recognize). When trying to find more, it is crucial to be sure that they to not reduce to an already existing class arithmetically.

- class 1: $n_{i-1}$ — fix operator — fix value
- class 2: $n_{i-1}$ — fix operator — series from class 1
- class 3: $n_{i-1}$ — varying operators — varying values
- class 4: two predecessors — fix operator
- class 5: $n_{i-1}$ — fix operator — series from class 4

### 3.1.4. Results

As a result we have a set of rules to create number series, like Carpenter et al.[CarpEtAl90] have created for the raven matrices. Notice that all our series are created recursively exept the two with squares. Later we found out that these series can be written recursively, as well. But when doing so, their structure puts them into class 2 — the list of odd numbers is added to the predecessor. Most of them can be rewritten index-based as well. List 3.1.4 shows each number series we created with their functions.

- Numeric simple // Operational simple
  - Class 1 — NS1
    `3, 5, 7, 9, 11, (13)`
    $[a_n = a_{n-1} + 2]$
  - Class 2 — NS2
    `2, 7, 13, 20, 28, (37)`
    $[a_n = a_{n-1} + (n + 4)]$

  - Class 3 — NS3
    `5, 7, 10, 12, 15, (17)`
    $[a_{n=2k-1} = a_{n-1} + 2 || a_{n=2k} = a_{n-1} + 3]$
  - Class 4 — NS4
    `2, 2, 4, 6, 10, (16)`
    $[a_n = a_{n-1} + a_{n-2}]$
  - Class 5 — NS5
    `1, 2, 4, 7, 12, (20)`
    $[a_n = a_{n-1} + fib(n, 1, 2)]$

- Numeric complex // Operational simple
  - Class 1 — NS6
    `107, 291, 475, 659, 843, (1027)`
    $[a_n = a_{n-1} + 184]$
  - Class 2 — NS7
    `237, 311, 386, 462, 539, (617)`
    $[a_n = a_{n-1} + (n + 73)]$
  - Class 3 — NS8
    `128, 254, 381, 507, 634, (760)`
    $[a_{n=2k-1} = a_{n-1} + 126 || a_{n=2k} = a_{n-1} + 127]$
  - Class 4 — NS9
    `103, 103, 206, 309, 515, (824)`
    $[a_n = a_{n-1} + a_{n-2}]$
  - Class 5 — NS10
    `1, 24, 47, 93, 162, (277)`
    $[a_n = a_{n-1} + fib(n, 23, 23)]$

- Numeric simple // Operational complex
  - Class 1 — NS11
    `1, 4, 9, 16, 25, (36)`
    $[a_n = n^2]$
  - Class 2 — NS12
    `1, 1, 2, 6, 24, (120)`
    $[a_n = a_{n-1} * n]$
  - Class 3 — NS13
    `4, 6, 12, 14, 28, (30)`
    $[a_{n=2k-1} = a_{n-1} + 2 || a_{n=2k} = a_{n-1} * 2]$
  - Class 4 — NS14
    `2, 2, 4, 8, 32, (256)`
    $[a_n = a_{n-1} * a_{n-2}]$

- Class 5 — NS15
  `1, 1, 3, 12, 84, (924)`
  $[a_n = a_{n-1} * fib(n, 1, 3)]$

- Numeric complex // Operational complex
  - Class 1 — NS16
    `121, 144, 169, 196, 225, (256)`
    $[a_n = (n + 10)^2]$
  - Class 2 — NS17
    `7, 7, 14, 42, 168, (840)`
    $[a_n = a_{n-1} * n]$
  - Class 3 — NS18
    `16, 48, 51, 153, 156, (468)`
    $[a_{n=2k-1} = a_{n-1} * 3 || a_{n=2k} = a_{n-1} + 3]$
  - Class 4 — NS19
    `2, 3, 6, 18, 108, (1944)`
    $[a_n = a_{n-1} * a_{n-2}]$
  - Class 5 — NS20
    `3, 3, 9, 36, 252, (2772)`
    $[a_n = a_{n-1} * fib(n, 1, 3)]$

# 4. Inductive Programming

## 4.1. MagicHaskeller

MagicHaskeller is a system for inducing functions. The goal is to synthesize programs from a set of existing functions and input-output specifications. To achieve this, two different approaches are given by the MagicHaskeller: The *analytical approach* synthesizes programs by conducting inductive inference[KS11], while the *generate-and-test approach* creates many programs and tries to find one that matches the specified input-output pairs. In an e-mail Susumu Katayama pointed out that the analytical approach of MagicHaskeller is not able to solve higher-order functions (see A.4), which is crucial to our series. It is also not able to solve fibonacci series, which we use in two of our complexity classes[KS11]. Therefore, we cannot use the analytical approach in our experiment. The generate-and-test approach cannot make any inferences on the examples given. It just creates a stream of functions and tests them against the given examples. This is done with an exhaustive breadth-first search. This means that at first MagicHaskeller tries the most simple possible operator combinations, increasing complexity step by step. In the case of our number series this can be done arbitrarily long, considering the identity property of 1 at multiplication and 0 at addition, respectively.
Program generation is done from a set of given functions, so called *primitives*, and a function to put these primitives into. MagicHaskeller has sets of these functions built-in so that the basic example given by Katayama can be derived very easily. For our specific use, and to have control of the operators used we will make up the primitives set by our own.
To see the results of *generate-and-test*, and as a little preview to our findings see listing 4.1. In the listing we can see how the MagicHaskeller tries out different operator combinations systematically. Of course we only see the working results here, but the schema gets clear.

```
\a -> seriesCL1to4 a (a + 2) [\_ c _ -> c + 2]
\a -> seriesCL1to4 a (a + 2) [\_ c _ -> 2 + c]
\a -> seriesCL1to4 a (2 + a) [\_ c _ -> c + 2]
\a -> seriesCL1to4 a (2 + a) [\_ c _ -> 2 + c]
```

**Listing 4.1** — Results from generate-and-test

## 4.2. Solving Number Series in MagicHaskeller

Regarding previous work of [DWZ12] results regarding Application of MH via generate-and-test were:
7 of the proposed 20 series could be solved by the program-feed using a set of basic functions(series, checksum, square,.. - for details, see [DWZ12]) in combination with the definition

of natural-number series 'smallNats' and 'nats'; furthermore the 'filterSeries' definition. The complexity of the types of solved series hereby varies, $a_(n) = a_(n-1) + (n * 2)$ being the most complex.
When it comes to the Quantitative Analysis of the survey-data, a one-sample t-test was applied, thereby falsifying the hypothesis(which was: previous knowledge about number-series of the test-persons leads to a better test-result.). Moreover, by evaluation of the think-aloud-protocols, 3 solution strategies could be observed(see [DWZ12] pp. 10 for detailed information).

In our experiment, we want the MagicHaskeller to solve number series. We have made restrictions on these number series regarding the arithmetical operations possible. Since we only have monotonically increasing numbers we only need addition, multiplication, and for two of our series we need the square operator. These are the basic primitives given to the MagicHaskeller.

### 4.2.1. Number Series Generation with Haskell as Preliminary Work

Before we explain our final MagicHaskeller configuration, we first introduce the way we developed it. To give the MagicHaskeller a function to operate on, we first needed to investigate about generating number series. Therefore, we created functions to calculate the series ourselves. We found functions for each complexity class we defined before and were able to create the series just by changing the function parameters. According to our classes we began with the most simple form. In the first class we are constrained to constantly applying the same operator on the previous number. See our generation function in the first two lines of listing 4.2. Notice that we only pass unary operators, meaning partially applied binary operators, to the function (e.g. *plus 2*). We do this so we only have one numeric parameter. The other parameter is what we want MagicHaskeller to find. The example also shows that we create our number series recursively, the same way we did when "inventing" the series *off-line*. In the last line of listing 4.2 we see the creation of the series containing all odd numbers, starting at three. More concrete we give the number 3 as start point for the series as first parameter and *add 2* as operator.

```
seriesCL1 :: Int -> (Int -> Int) -> [Int]
seriesCL1 n f = n : seriesCL1 (f n) f

series_CL1  = seriesCL1 3 (+ 2)
```

**Listing 4.2** — Complexity Class 1 series generator

To explain the other techniques we used, the generator functions of our other classes will be explained now. The second complexity class contains number series with constantly monotonically increasing arguments for addition and multiplcation. In order to achieve this, an index was integrated counting the recursive calls. Also a second input parameter was added, to afford addressing this index by the use of higher order functions as input where the first argument $n$ is the current series element and the second argument $i$ is the counting index (4.3).

```
seriesCL2 :: Int -> (Int -> Int -> Int) -> [Int]
seriesCL2 n f = rec n 0 f
where rec n i f = n : rec (f n i) (i + 1) f
```

```
series_CL2  = seriesCL2 2 (\n i -> n + i + 1)
```

**Listing 4.3 —** Complexity Class 2 series generator. A higher order function is passed as second argument

In listing 4.4 we changed the operator parameter to be a list. This is because we want to have alternating operations on the previous number. See again the last line for function application. We pass *plus 2* and *plus 3* as operators. The function takes the first element, applies it, and appends it to the list for the next use. We can see in this example that class 1 is a subset of class 3, namely a list with only one operator in it (of course it is possible to give the same operator many times, as well, MagicHaskeller will want to try this at some place). To make MagicHaskeller capable of putting operators into a list, we also need to define list operations as primitives, but we will come to this point later on.

```
seriesCL3 :: Int -> [(Int -> Int)] -> [Int]
seriesCL3 n (h:t) = n : seriesCL3 (h n) (t ++ [h])

series_CL3  = seriesCL3 5 [(+ 2),(+ 3)]
```

**Listing 4.4 —** Complexity Class 3 series generator. Now a list of operators is passed as second argument.

In the last example we must part from our idea of applying unary operators. In class 4 we create *Fibonacci*-like series, and, therefore, must pass an operator capable of taking the previous two numbers. The resulting function is shown in listing 4.5. Not only that we have a binary operator, as explained, we also have two numbers to start the series from, instead of one. We could also have started with using the same number twice, but in the series we want to test we have to cope with two distinct starting numbers.

```
seriesCL4 :: Int -> Int -> (Int -> Int -> Int) -> [Int]
seriesCL4 n0 n1 f = n0 : n1 : rec n0 n1 f
where rec n0 n1 f = (f n0 n1) : rec n1 (f n0 n1) f

series_CL4  = seriesCL4 2 2 (+)
```

**Listing 4.5 —** Complexity Class 4 series generator. To create *Fibonacci*-like series we need two starting numbers and a binary operator.

By now we have a generation function for each of our complexity classes. The next step is combining them to one overall function, which can produce the entirety of our number series. Therefore we need two arguments for two possible starting values and a list of functions taking three input parameters. ”Possible starting values” means, that we need to create series with one and series with two starting values out of one generation function. If the first argument is less or equal to zero, the second one is recognized as the only starting value. The list of three-digit higher order functions enables the combination of the starting value(s) and the index. In listing 4.6 the exhaustive function can be found showing the generation of all complexity classes.

```
seriesCL1to4 :: Int -> Int -> [(Int -> Int -> Int -> Int)] -> [Int]
seriesCL1to4 n0 n1 l | n0 <= 0   = n1 : rec 0 n1 0 l
                     | otherwise = n0 : n1 : rec n0 n1 0 l
where rec n0 n1 i (h:t) = (h n0 n1 i) : rec n1 (h n0 n1 i) (i + 1) (t
   ++ [h])
```

```
series_CL1 = seriesCL1to4 0 3 [(\_ n1 _ -> n1 + 2)]
series_CL2 = seriesCL1to4 0 2 [(\_ n1 i -> n1 + i + 1)]
series_CL3 = seriesCL1to4 0 5 [(\_ n1 _ -> n1 + 2), (\_ n1 _ -> n1 + 3)
    ]
series_CL4 = seriesCL1to4 2 2 [(\n0 n1 _ -> n0 + n1)]
```

**Listing 4.6** — Complexity Classes 1 - 4 series generator.

### 4.2.2. The Final MagicHaskeller Program

MagicHaskeller, in the generate-and-test mode that we use, generates output by combining known primitives. In the previous section we developed a function that can be used to generate any of our defined number series. This function is one of the primitives we will provide MagicHaskeller. The program can be seen in 4.8. The listing shows the other primitives in the `seriesLib` set, which are list operations and arithmetic operations. When preparing MagicHaskeller we also pass the set `nats` from the MagicHaskeller library for the program should operate on numbers. The set represents the natural numbers. In fact it does not, this led to issues that will be discussed in 4.3.2. The call for the program looks as follows:

```
filterSeries (\f -> take 5 (f (3 :: Int)) == [3, 5, 7, 9, 11 :: Int])
    >>= MagicHaskeller.pprs
```

**Listing 4.7** — Exemplary Call.

MagicHaskeller is now starting to generate all possible programs matching the type `[Int]` by using the given primitives. If a generated program fulfils the boolean check seen in the call, it is printed as one possible result. Since we did not add a filter removing semantically equivalent results, many similar programs will be printed.

## 4.3. Results and Observations

### 4.3.1. General

Twelve of our twenty series could be solved by the MagicHaskeller. In the cases where the MagicHaskeller can find a solution, it always puts out a stream of solutions, semantically identical in most cases, because of the identity property of 1 and commutativity. We found different solutions, both wrong, for NS15, though. This means that it was not unique regarding the given first five numbers. The oeis[1] only finds the one sequence, that we intended to have, with the next number 924, as expected. Interestingly the MagicHaskeller finds a different solution to NS3 than expected. We wanted it to find a solution with a list of two alternating functions ( `[\b _ _ -> b + 2, \b _ _ -> b + 3 ]` ), instead it builds up its solution with the first two values (it calculated the second with `( a + 2 )` itself) as starting values, with the pattern $n_i = n_{i-2} + 5$. This, of course, leads to a correct solution, but we did not think of this way. The pattern also leads to another kind of class, that we did not define. Notice that the 5 in the pattern comes

[1] On-Line Encyclopedia of Integer Sequences: `http:/www.oeis.org` Sequence ID: A070825

```
module Series where

import MagicHaskeller
import MagicHaskeller.LibTH

seriesLib :: [Primitive]
seriesLib = $(p [|(seriesCL1to4 :: Int -> Int -> [(Int -> Int -> Int -> Int
    )] -> [Int], [] :: [(Int -> Int -> Int -> Int)], (:) :: (Int -> Int ->
    Int -> Int) -> [(Int -> Int -> Int -> Int)] -> [(Int -> Int -> Int ->
    Int)], (+) :: Int -> Int -> Int, (*) :: Int -> Int -> Int|])



mseries :: ProgramGenerator pg => pg
mseries = mkPGOpt (options{constrL=True}) (seriesLib ++ nats)

seriesCL1to4 :: Int -> Int -> [(Int -> Int -> Int -> Int)] -> [Int]
seriesCL1to4 n0 n1 l | n0 <= 0   = n1 : rec 0 n1 0 l
                     | otherwise = n0 : n1 : rec n0 n1 0 l
        where rec n0 n1 i (h:t) = (h n0 n1 i) : rec n1 (h n0 n1 i) (i + 1)
            (t ++ [h])

filterSeries pred = filterThen pred (everything (mseries :: ProgGen))
```

**Listing 4.8** — The final MagicHaskeller Program

from the first parameter of the function, as it is the first number in the series. MagicHaskeller does not give it explicitly, the reason for this is explained in section 4.3.2. MagicHaskeller is also not able to find solutions to the other class 3 sequences, why so will be clear after section 4.3.2,NS18 should have had been solved, though.

### 4.3.2. High Number Problem

The results show problems with high numbers, the numerical hard number series 6-8, 10, 16 and 18 could not be solved by the MagicHaskeller. This is quite irritating, because we expected that high numbers would not be any kind of problem for a computer. It is not the case that the function `seriesCL1to4` is not able to solve the series, for example NS6 is solved by the application of `\a -> seriesCL1to4 0 a [_ c _ -> c + 184]`.

The reason for this problem was not easy to find, but explains the situation well. When inspecting the programs MagicHaskeller produced it was suspicious that it always finds solutions using the numbers two and three. Looking at the source code of MagicHaskeller the reason became obvious. The set *nats* in `LibTH`, that we use, does not contain the natural numbers. It only contains 1, 2 and 3! Therefore MagicHaskeller simply does not know about other numbers and cannot use them. A solution could be to let the MagicHaskeller use the set *naturals*, that defines the natural numbers recursively. When trying this, no solution could be found within several minutes; the computational complexity is too high and calculation time grows too fast. There

| Series | Resulting Program | Result | Correct |
|---|---|---|---|
| 1 | `\a -> seriesCL1to4 a (a + 2) [_ c _ -> c + 2]` | 13 | yes |
| 2 | not solved | — | — |
| 3 | `\a -> seriesCL1to4 a (a + 2) [\b _ _ -> b + a]` | 17 | yes |
| 4 | `\a -> seriesCL1to4 a a [\b c _ -> c + b]` | 16 | yes |
| 5 | `\a -> seriesCL1to4 a 2 [\b c _ -> c + (b + a)]` | 20 | yes |
| 6 | not solved | — | — |
| 7 | not solved | — | — |
| 8 | not solved | — | — |
| 9 | `\a -> seriesCL1to4 a a [\b c _ -> c + b]` | 824 | yes |
| 10 | not solved | — | — |
| 11 | not solved | — | — |
| 12 | `\a -> seriesCL1to4 a a [_ c d -> c * (d + 2)]` | 120 | yes |
| 13 | not solved | — | — |
| 14 | `\a -> seriesCL1to4 a a [\b c _ -> c * b]` | 256 | yes |
| 15 | `\a -> seriesCL1to4 a a [\b c d -> c * (d + (b + 2 ))]` | 1428 | no |
| | `\a -> seriesCL1to4 a a [_ c d -> c * (3 + (d * d))]` | 1008 | no |
| 16 | not solved | — | — |
| 17 | `\a -> seriesCL1to4 a a [_ c d -> (d + 2) * c]` | 840 | yes |
| 18 | not solved | — | — |
| 19 | `_ -> seriesCL1to4 2 3 [\b c _ -> c * b]` | 1944 | yes |
| 20 | `\a -> seriesCL1to4 a a [_ c d -> c * (3 + (d * d))]` | 3024 | no |

**Figure 4.1.** — Results of the MagicHaskeller. Only one example result is given, MagicHaskeller produces a stream of semantically identical programs. The *Result*-column shows the number *predicted* by the program.

are many parameters, that the numbers could be put into, but the breadth-first search will take too long to find a first solution.
When feeding the needed numbers for a specific series into the primitives set directly, MagicHaskeller can find the solution very quickly. What really would be needed is a function, or another way, to extract information, in form of integer numbers, from the given series. With this the needed numbers can be found within the given data, without the need of an external definition of natural numbers.
Also notice that, without having 0 in `nats` the case differentiation in our function does not work. When passing 0 as *primitive* it should work, though.

### 4.3.3. Optimization

After figuring out these problems we improved our source code to get more number series solved. The numbers 0,1,2,3,4,5 and some selected higher numbers were added to the primitive set used for generating programs. Now the series 2, 6, 7, 8 and 9 could be solved, too. Another aspect we were wondering about was, that the square number series 11 and 16 could not be solved, although the needed numbers were now available. Maybe we gave the MH not enough time finding the solution(e. g. (d + 2) * (d + 2) - series 11). We simply added a square function for natural numbers and the MH generated a solution very fast.
The MH does not generate more than one list item in the function list, although we provided the list constructors. That's why it's not able to solve the alternating series 13 and 18. The easier ones 3 and 8 could be solved by setting two starting values and adding the sum of the alternating addition values to the pre-predecessor. The desired solutions of the alternating series can be seen in listing 4.9.

```
seriesCL1to4 0 5 [(\_ c _ -> c + 2), (\_ c _ -> c + 3)]
seriesCL1to4 0 128 [(\_ c _ -> c + 126), (\_ c _ -> c + 127)]
seriesCL1to4 0 4 [(\_ c _ -> c + 2), (\_ c _ -> c * 2)]
seriesCL1to4 0 16 [(\_ c _ -> c * 3), (\_ c _ -> c + 3)]
```

**Listing 4.9** — Alternating number series

Finally 18 out of 20 number series could be solved. The updated results after optimizing can be found in Figure 4.2.

## 4.4. Conclusion

### 4.4.1. RunAnalytical

Regarding the analytical synthesis via the module .RunAnalytical, it can be concluded that due to missing support of higher-order functions(MH v 0.8.6.3) a feasible approach of solving the set of series does not exist. Therefore, the only way of solving integer-series including higher-order constructs with the program MagicHaskeller is via generate-and-test(as long as there is no additional implementation in the near future).

### 4.4.2. Generate-and-Test

Concluding the sections 4.3.1–4.3.3 we can say that the generate-and-test method is really powerful in finding solutions for number series. It only suffers from not knowing numbers. The recursive definition of numbers leads to too high complexity to calculate a solution, but given the needed numbers the approach is effective.

| Series | Resulting Program | Result | Correct |
|---|---|---|---|
| 1 | `\a -> seriesCL1to4 0 a [_ c _ -> c + 2]` | 13 | yes |
| 2 | `\a -> seriesCL1to4 0 a [_ c d -> c + (d + 5)]` | 37 | yes |
| 3 | `\a -> seriesCL1to4 a (a + 2) [\b _ _ -> b + a]` | 17 | yes |
| 4 | `\a -> seriesCL1to4 a a [\b c _ -> b + c]` | 16 | yes |
| 5 | `\a -> seriesCL1to4 a 2 [\b c _ -> (b + c) + 1]` | 20 | yes |
| 6 | `\a -> seriesCL1to4 0 a [_ c _ -> c + 184]` | 1027 | yes |
| 7 | `\a -> seriesCL1to4 0 a [_ c d -> c + (d + 74)]` | 617 | yes |
| 8 | `\a -> seriesCL1to4 a (a + 126) [\b _ _ -> b + 253]` | 760 | yes |
| 9 | `\a -> seriesCL1to4 a a [\b c _ -> b + c]` | 824 | yes |
| 10 | `\a -> seriesCL1to4 a 24 [\b c _ -> (b + c) + 22]` | 277 | yes |
| 11 | `\a -> seriesCL1to4 0 a [_ _ d -> square (d + 2)]` | — | yes |
| 12 | `\a -> seriesCL1to4 0 a [_ c d -> c * (d + 1)]` | 120 | yes |
| 13 | not solved | — | — |
| 14 | `\a -> seriesCL1to4 a a [\b c _ -> b * c]` | 256 | yes |
| 15 | `\a -> seriesCL1to4 a a [\b c d -> c * (d + (b + 2 ))]` | 1428 | no |
|  | `\a -> seriesCL1to4 a a [_ c d -> c * (3 + (d * d))]` | 1008 | no |
| 16 | `\a -> seriesCL1to4 0 a [_ _ d -> square (d + 12)]` | — | yes |
| 17 | `\a -> seriesCL1to4 0 a [_ c d -> c * (d + 1)]` | 840 | yes |
| 18 | not solved | — | — |
| 19 | `\a -> seriesCL1to4 a 3 [\b c _ -> b * c]` | 1944 | yes |
| 20 | `\a -> seriesCL1to4 a a [_ c d -> c * (3 + (d * d))]` | 3024 | no |

**Figure 4.2.** — Optimized results of the MagicHaskeller.

# 5. Experiment

## 5.1. Method

After having the sequence of our number series we structured the set of number series like a questionnaire embedded in a website. For this purpose we used www.soscisurvey.de, which is a professional software-package for online questionnaires. To prevent sequence effects as much as possible, we ordered the sequences in such a way, that each number series was once within the first five number series, once within the last five number series and two times between them:

1. questionnaire : 2,1,15,6,8,12,4,17,20,16,11,18,5,19,14,9,10,13,3,7
2. questionnaire: 4,11,10,3,14,19,7,9,13,8,15,12,1,20,17,5,2,6,18,16
3. questionnaire : 12,9,20,16,17,4,3,10,8,13,7,18,6,5,2,1,19,15,14,11
4. questionnaire : 5,7,19,13,18,6,3,9,1,2,10,11,15,14,16,8,12,4,20,17

Which sequence of number series the subjects got was completely random. So there was a likelihood of 25% per different questionnaire. The time measurement was executed separately for each page in the questionnaire, measured in seconds. First the subjects had to answer questions about gender, age and occupation. Then they were given two examples of how number series can be built and were asked to pace themselves through the following pages, of course trying to solve the tasks, but always having in mind that the time they spend on each page is also recorded for further analysis. Another request was not to use any help such as a notepad, a calculator, the internet or other means. On each page the subjects were asked to complete the series and to estimate the difficulty of it on a rating scale from one to five. So the time measured was not exactly the time needed to solve the number series, but the measurement included the time to rate the difficulty and to click on to the next page. At the end of the sequence, we asked to indicate which means they used, if at all.
The questionnaire used can be seen in A.5.

## 5.2. Results

In four weeks time we got 99 data. There were 53 data we decided to throw out because there were subjects who just opened the questionnarie, read a few pages and left without even clicking through the number series; there were a few who clicked through the most pages without completing anything; there were a few who in our interpretation did not take it serious and wrote always the same number; all these data were dropped. At last we had 46 valid data. As mentioned before, we prepared four questionnaries. They were completed by 16 female, 29

| Questionarrie Number | How Often Filled out | Percentage |
|---|---|---|
| 1 | 11 | 23,9 |
| 2 | 8 | 17,4 |
| 3 | 9 | 19,6 |
| 4 | 18 | 39,1 |

**Figure 5.1.** — Statistics about our questionnaire

male subjects and one was not specified. Most of the participants were students (13f and 22m) between 19 and 31 years, and 11 participants had other occupations. These were between 18 and 42 years old. The first questionnarie missed the page about aids and from the other 35 16 indicated the calculator, 12 the notepad and 2 foreign help. 5 didn't give any indication, which could mean they didn't use anything but their memory.

A few things were striking as we compared results. At some of the number series the participants gave solutions near to the correct result. Here we concluded that they had difficulties with the workload. This applies for NS7, NS8, (NS9) and NS18. Another factor is the uniqueness: NS4, NS5 and NS20 were the ones where another suggestion occured more than 5 times out of 46. (that means more than 10%)

In the following table we wish to give a summary of all the number series, the correct result and the solution process. NS5, NS10, NS15 and NS20 were by far solved less often than the other series. It is not surprising that the estimated difficulty of these series lies at 4 and 5. We assume a correlation between time and estimated difficulty.

The meantime for solving the number series is calculated out of all times even if the result wasn't correct. But the min - Max time is only from those subjects who gave the correct answer.

When grouping the average times (see figure 5.2) the hardness of class 5 stands out directly. In all numerical and operational complexity variations we observed a dramatic increase in mean working time. Another obvious effect is the long time needed for numerically complex and operationally simple number series. Figure 5.3 confirms the hardness of class 5 number series. A minor effect can be observed in class 2, which takes more time and, as can be seen in 5.4 is less often solved correctly in comparison to classes 3 and 4, which we intended to be harder. The same effect can be seen in figure 5.5—the test persons also had the *feeling* that class 2 is slightly harder.

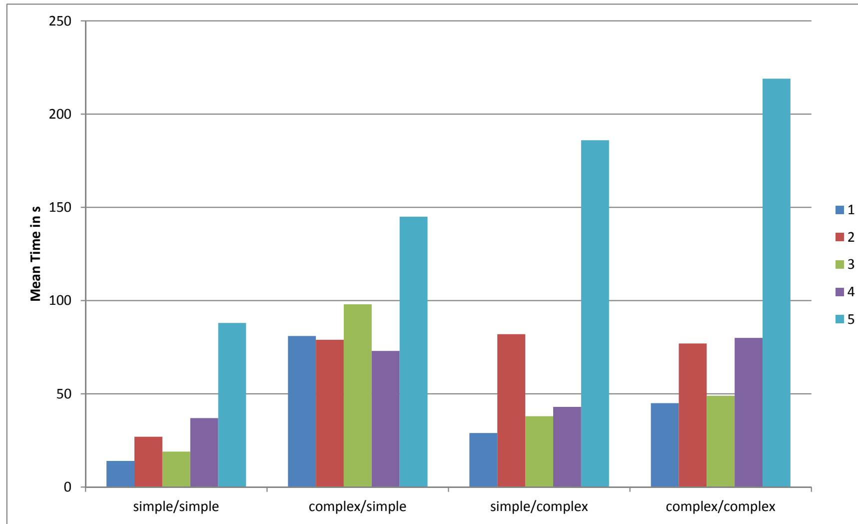


**Figure 5.2.** — Average time, grouped by numerical/operational complexity

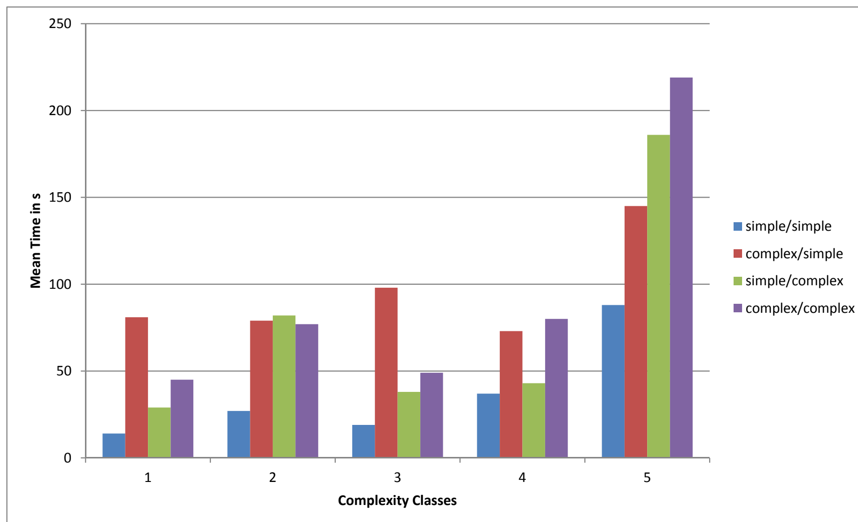


**Figure 5.3.** — Average time, grouped by class complexity

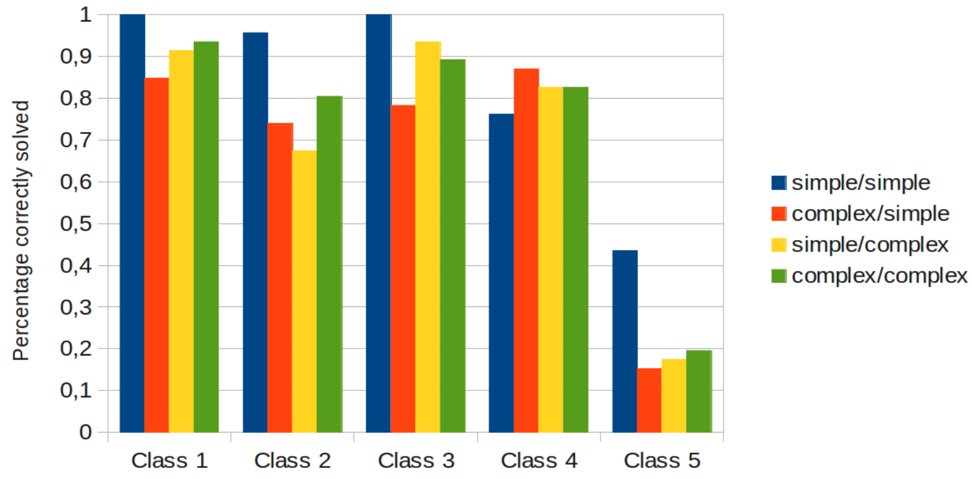


**Figure 5.4.** — Percentage of correctly solved number series, grouped by class complexity

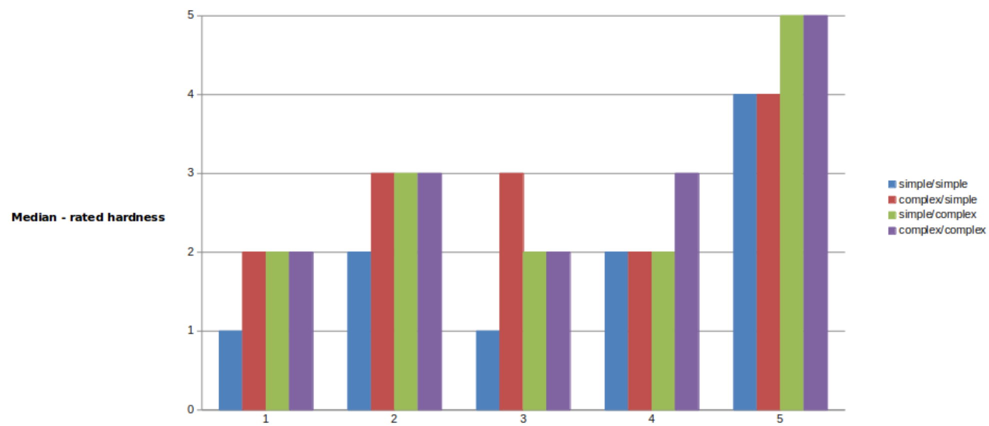


**Figure 5.5.** — Median of user ratings, grouped by class complexity

| Nr. | Number Series | How to Solve | Correct Results | Nr. of Correct Results | Median Estimated Difficulty | Meantime for solving Number Series | Min - Max Time for Corr. Result |
|---|---|---|---|---|---|---|---|
| 1. | 3, 5, 7, 9, 11, | $a_n = a_{n-1} + 2$ | 13 | 46 | 1 | 13,78 | 5 - 48 |
| 2. | 2, 7, 13, 20, 28, | $a_n = a_{n-1} + (n+4)$ | 37 | 44 | 2 | 27,21 | 8 - 84 |
| 3. | 5, 7, 10, 12, 15, | $a_{n=2k-1} = a_{n-1} + 2$<br>$a_{n=2k} = a_{n-1} + 3$ | 17 | 46 | 1 | 19,15 | 8 - 61 |
| 4. | 2, 2, 4, 6, 10, | $a_n = a_{n-1} + a_{n-2}$ | 16 | 35 | 2 | 37,68 | 12 - 138 |
| 5. | 1, 2, 4, 7, 12, | $a_n = a_{n-1} + fib(n,1,2)$ | 20 | 20 | 4 | 88 | 23 - 341 |
| 6. | 107, 291, 475, 659, 843, | $a_n = a_{n-1} + 184$ | 1027 | 39 | 2 | 81,04 | 29 - 210 |
| 7. | 237, 311, 386, 462, 539, | $a_n = a_{n-1} + (n+73)$ | 617 | 34 | 3 | 78,65 | 33 - 327 |
| 8. | 128, 254, 381, 507, 634, | $a_{n=2k-1} = a_{n-1} + 126$<br>$a_{n=2k} = a_{n-1} + 127$ | 760 | 36 | 3 | 97,76 | 29 - 337 |
| 9. | 103, 103, 206, 309, 515, | $a_n = a_{n-1} + a_{n-2}$ | 824 | 40 | 2 | 73,02 | 18 - 505 |
| 10. | 1, 24, 47, 93, 162, | $a_n = a_{n-1} + fib(n,23,23)$ | 277 | 18 | 4 | 144,84 | 211 - 218 |
| 11. | 1, 4, 9, 16, 25, | $a_n = \quad n^2$ | 36 | 42 | 1 | 29,41 | 8 - 95 |
| 12. | 1, 1, 2, 6, 24, | $a_n = a_{n-1} * n$ | 120 | 31 | 3 | 82,28 | 17 - 317 |
| 13. | 4, 6, 12, 14, 28, | $a_{n=2k-1} = a_{n-1} + 2$<br>$a_{n=2k} = a_{n-1} * 2$ | 30 | 43 | 2 | 37,76 | 9 - 176 |
| 14. | 2, 2, 4, 8, 32, | $a_n = a_{n-1} * a_{n-2}$ | 256 | 38 | 2 | 43,08 | 12 - 217 |
| 15. | 1, 1, 3, 12, 84,, | $a_n = a_{n-1} * fib(n,1,3)$ | 924 | 8 | 5 | 185,60 | 35 - 1237 |
| 16. | 121, 144, 169, 196, 225, | $a_n = \quad (n+10)^2$ | 256 | 43 | 2 | 44,65 | 19 - 97 |
| 17. | 7, 7, 14, 42, 168, | $a_n = a_{n-1} * n$ | 840 | 37 | 3 | 77,28 | 14 - 266 |
| 18. | 16, 48, 51, 153, 156, | $a_{n=2k-1} = a_{n-1} * 3$<br>$a_{n=2k} = a_{n-1} + 3$ | 468 | 41 | 2 | 48,89 | 16 - 127 |
| 19. | 2, 3, 6, 18, 108, | $a_n = a_{n-1} * a_{n-2}$ | 1944 | 37 | 3 | 80,43 | 21 - 275 |
| 20. | 3, 3, 9, 36, 252, | $a_n = a_{n-1} * fib(n,1,3)$ | 2772 | 9 | 5 | 218,65 | 54 - 2380 |

| Number Series | Failed to Solve | Error Percentage |
|---|---|---|
| 1 | 0 | 0,0 |
| 2 | 2 | 4,36 |
| 3 | 0 | 0,0 |
| 4 | 11 | 23,91 |
| 5 | 26 | 56,52 |
| 6 | 7 | 15,22 |
| 7 | 12 | 26,09 |
| 8 | 10 | 21,74 |
| 9 | 16 | 13,04 |
| 10 | 28 | 60,87 |
| 11 | 4 | 8,70 |
| 12 | 15 | 32,61 |
| 13 | 3 | 6,52 |
| 14 | 8 | 17,35 |
| 15 | 38 | 82,61 |
| 16 | 3 | 6,52 |
| 17 | 9 | 19,57 |
| 18 | 5 | 10,87 |
| 19 | 9 | 19,57 |
| 20 | 37 | 80,43 |

**Figure 5.6.** — The diagramm shows what we expected: the harder the task, the higher the error percentage is and as we saw in the previous table the difficult number series 5, 10, 15 and 20 took more time and the rate of correct results is much lower than at the other series.

## 5.3. Time-Error-Rate

Figure 5.9 shows the average time for each number series divided into completed correctly (blue) and completed incorrectly or not at all (red). Number series one and three were solved by all subjects and therefore have no red bar. Surprisingly in the most cases there was not much difference between the time of correct and incorrect/no results. About the easy number series 1-5 with low number and low operator complexity you can't say much, because they were solved by almost all subjects and the sample size for the "not-solved-time" is too small or nonexistent. Besides number series five was ambiguous and should not be considered. For the number series with high number complexity the "solved-time" is a bit higher than the "not-solved-time". This might be because even if you knew the compositional rules of the number series, you had to calculate the result, which takes some time. In contrast the number series with high operator complexity were handled faster by the test persons, who came up with the

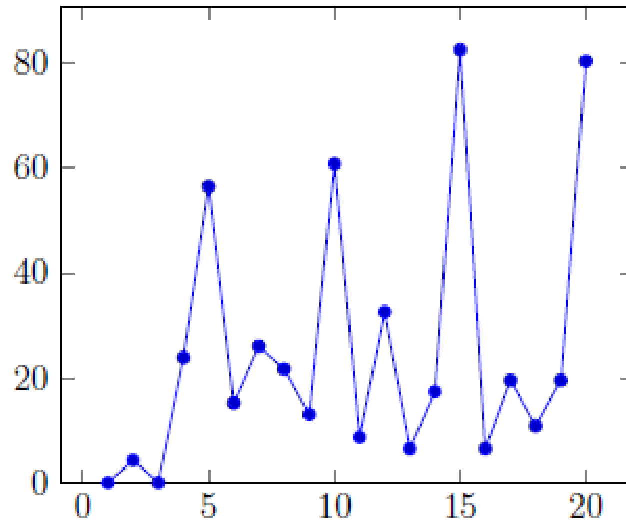


**Figure 5.7.** — Error Percentage

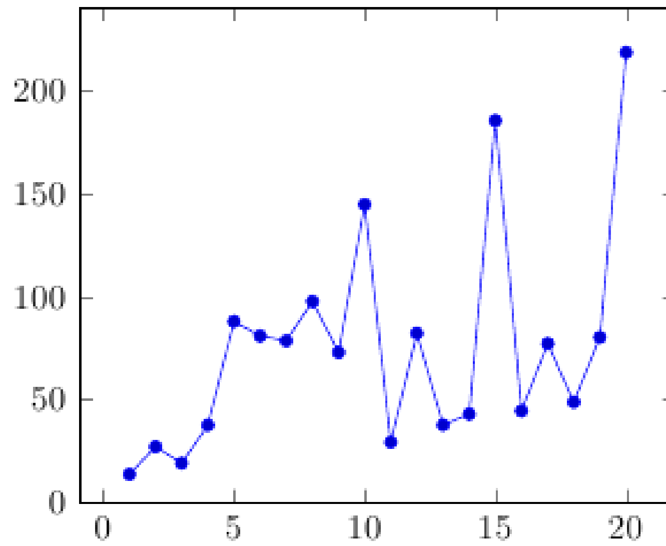


**Figure 5.8.** — Mean Time

correct result. Exceptions were number series 15 and 20. These two sequences were by far the most difficult. They were only solved by 8/9 subjects. The average time for both number series,

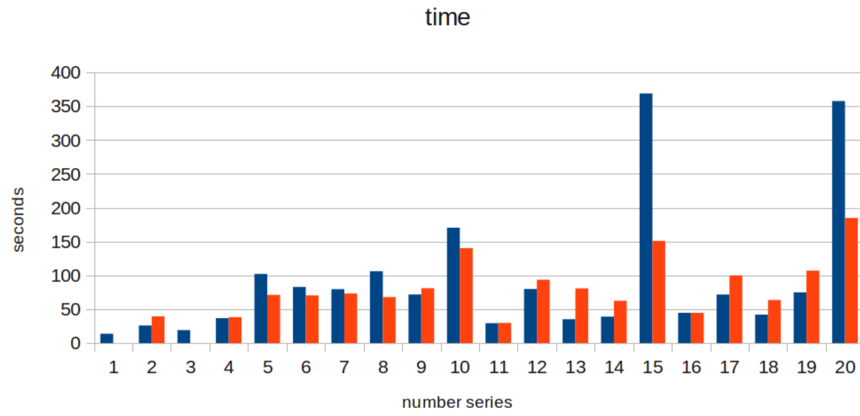


**Figure 5.9.** — Comparison of error-rates and solution time

when solved, is above 350 seconds. The reason why there is so much difference between the solved and not solved time is probably that they took a lot of time to finish correctly and most of the subjects gave up much earlier or came up with incorrect, but easier solutions.

## 5.4. Discussion

The structural complexity had the most influence on the error percentage (see figure 5.3). The number series 5,10,15 and 20 had by far the highest error percentage and meantime. The correlation of the complexity classes and the number of correct results was proven by a chi-squared test. The high number complexity caused several calculation errors. For example in NS7, where the correct solution was 617, some subjects answered the question with 607, 614, 616 and 618. As expected the number sequences with the lowest error percentage were also the lowest in the difficulty rating. Comparing the results to number series in the `oeis` we can see, not only for NS5, but it is a good example, that there are various solutions to our given series. 19 is a valid solution, as well as 20 or 21—but the definitions get rather complex quickly, e.g. for a series with 21 as next number: $a(0) = 0, a(1) = a(2) = a(3) = 1; thereafter, a(n) = a(n-1) + a(n-2) + a(n-4)$. For 19: Numbers n such that $binomial(2n, n) + 1$ is prime—these are most probably not the solutions our participants thought of.
Number Series 9 is also an interesting case. The number series was: $103, 103, 206, 309, 515$ and the correct solution 824. Some of our participants, about 10%, made the same error, their solution was 721. Instead of interpreting the first two numbers, $103, 103$ as a start and then always adding the two previous numbers ($a_n = a_(n-1) + a_(n-2)$) they saw that 103 was added to the previous number twice and then 206 was added once, they concluded that 206 had to be used again. This clearly has some logic in it, but then, what are the two 103s for? The first interval—0—violates their schema. By simply continuing the sequences a bit longer lower these difficulties as the participants could determine that their result function was wrong and the number series would not be falsly ambiguous.
Regarding the solving times, where series with high numerical complexity and low operational complexity seem harder than number series with both dimensions high complexity (see figure 5.2) an explanation may be that the test persons *expected* multiplication on high numbers, where low operational complexity uses addition. Furthermore, we could not choose high operands in the complex/complex scenario because multiplication with e.g. 32 leads to tremendously high numbers in few steps. To the contrary, the multiplications we used ($*2$, $*3$) are quite easy to handle by humans and do not lead to high difficulties due to high operational complexity. The user ratings support that thesis, *high* operational complexity is only rated slightly harder than *low*.

# 6. Conclusion

In the end we did not use the analytical approach of the Magic Haskeller, due to the fact it cannot handle higher order functions. But using the generate and test processing we got the most of our number series solved. We constructed a function (seriesCL1to4) that was able to express all of our NS. This function plus some selected natural numbers in addition with some natural number operators were given to the Magic Haskeller as primitive set. Generating all type consistent permutations of our primitives, the MH solved 18 out of 20 number series using the brute force generate and test approach. The Solving will take much more time when the primitive set is expanded. Adding all natural numbers less than an upper bound, would be a proper example. The performance time would raise exponentially in trade-off to a higher solving rate. The potential of solving number series with the MH's generate and test approach is mostly exhausted with our realization. Maybe there could be another approach in the future based on the .RunAnalytical module.

Our experiment was not fully developed and only later we discovered errors that can be avoided in a second trial. We have not examined whether the order of the series had an impact, we assume, however, that it did not play a role. It was important that the NS is unique and that the time for finding a solution is as short as possible. In the data analysis we noticed that some NS allowed several solutions, so they were not unique. Because aids were used we can't say that the measured time was "real mind computation" time. So it is not possible to compare humans and our program. We would propose that new number series should first be tested on uniqueness. When so far no multiple solutions come out it should be better to build up the NS with at least 6 or 7 numbers because we realized that the risk of multiple solutions is greater at 5 preset numbers. The time for each set of numbers is to be determined by the complexity of the NS and the solution should not only be the next number, or the next two numbers, but also the way of solving the NS should be given. This means, however, that the test takes more time. Additionally we would propose to test the NS first with the program and afterwards with people.

Another approach could be to test allegedly not unique number series, like our NS 9 or 10, where the same, though wrong, answer was given multiple times. The formula the testpersons found would fit for all except one number of the original series. This could be interesting for further investigation. Do the people not recognize the error, or do they accept it, because they have found a *possible* solution?

# Bibliography


[Int1] http://www.englisharticles.info/2011/02/22/inductive-reasoning/

[Int2] http://en.wikipedia.org/wiki/Inductive-reasoning/

[Int3] http://www.thefreedictionary.com/intelligence

[Int4] http://en.wikipedia.org/wiki/Charles-Spearman

[Int5] http://en.wikipedia.org/wiki/John-C.-Raven

[Int6] http://en.wikipedia.org/wiki/Raven's-Progressive-Matrices

[Int7] http://en.wikipedia.org/wiki/Series_%28mathematics%29

[CarpEtAl90] Carpenter, P. A., Just, M. A. and Shell, P. (1990). What one intelligence test measures: A theoretical account of the reasoning in the Raven Progressive Matrices test. Psychological Review, 97:404431

[Amt90] Amthauer, R., Brocke, B., Liepmann, D. and Beauducel, A. (1990). Intelligenz-Struktur-Test 2000 (I-S-T 2000) Hogrefe, Göttingen.

[SK63] Kenneth Kotovski and Herbert A. Simon (1963). Empirical tests of a theory of human acquisition of concepts for sequential patterns.

[KK98] http://www.dgps.de/fachgruppen/methoden/mpr-online/issue4/art5/node1.html

[KS11] S. Katayama: MagicHaskeller: System Demonstration, 2011

[DWZ12] M. Düsel, A. Werner, T. Zeißner: Solving Number Series with the MagicHaskeller, Course-report(KogSys-KogMod-M), Otto-Friedrich-Universität Bamberg 2012(non-published)

# A. Appendix

## A.1. Calculation of the Mental Age

For example if a child is 7 years and 1 month old, we may begin by giving him tests at the 6-year level. If he passes all these plus four for age 7, three for age 8, and one for age 9, his mental age is calculating by crediting him with 72 months for passing all tests at age 6 and two months mental credit for each additional test passed, i.e., 8 months at age 7 (4 out of 8 tests), 6 months at age 8, and two months at age 9. His total mental age (M.A.) is therefore 88 months, the I.Q. is 88/85. To remove the decimal point, the result is multiplied by 100, giving an I.Q. of 104.

## A.2. A Taxonomy of Rules in the Raven Test

- Constant in a row - the same value occurs throughout a row. but changes down a column.
- Quantitative pairwise progression - a quantitative increment or decrement between adjacent entries in an attribute such as size. position, or number.
- Figure addition or subtraction - a figure from one column is added to (juxtaposed or superimposed) or subtracted from another figure to produce the third.
- Distribution-of-three-values - three values from a categorical attribute (such as figure type) are distributed through a row.
- Distribution-of-two-values - two values from a categorical attribute are distributed through a row: the third value is null.

## A.3. About non-unique solvability

"Cognitively oriented research on inductive reasoning has been focused on identifying the cognitive processes involved in psychometric tasks such as analogy problems (Holzman, Pellegrino and Glaser, 1982; Sternberg, 1977) and series-completion problems (Holzman, Pellegrino and Glaser, 1983; Kotovsky and Simon, 1973; Simon and Kotovsky, 1963). Prominent models of sequence completion solution, on which a large part of research has been based, include four component processes: detection of relations, discovery of periodicity, completion of pattern description, and extrapolation, where the first three processes combine to generate a pattern description. Conclusions about performance on series-completion problems are derived mainly from a presumed relationship between pattern descriptions and working memory: more complex pattern descrip-

tions make greater demands on working memory. Clearly, on this level of model building, the problem of non-unique solvability of sequential items may not be ignored. In any case, modeling the solution process for some type of task has to be founded on a careful task analysis where all characteristics of the specific type of task and particularly all possible alternative solution ways have to be taken into account." (Klaus Korossy)

## A.4. E-mail from Susumu Katayama regarding RunAnalytical

Some months ago Ute asked me a similar question of how to do that with MagicHaskeller, and I replied with a solution using the exhaustive search module. I am not sure if a similar solution exists for analytical stuff, because the solution using the exhaustive stuff requires dealing with higher-order functions, though the current analytical stuff cannot deal with them.
Extending the analytical stuff to higher-order can be a research topic, but I do not have time for that. (I have a long ToDo list now.) Maybe you could try that with either MagicHaskeller or IgorII+ because they are open source projects.
Best regards,
Susumu

## A.5. Questionnaire

Fragebogen

*1. Seite*

Liebe Testteilnehmer,

im Rahmen unseres diesjährigen Bachelor-Projekts im Bereich Kognitive Systeme möchten wir eine Untersuchung durchführen, die sich mit dem Lösen von Zahlenreihen beschäftigt. Dazu ist uns wichtig, dass Sie die Anweisungen verstehen. Denn nur dann ist es uns möglich, richtige Schlussfolgerungen zu ziehen.

Die ausgefüllten Fragebögen werden anonym übertragen. Falls Sie das Ergebnis der Evaluation erfahren möchten, können Sie uns am Ende der Befragung ihre Email-Adresse hinterlassen und Sie bekommen den Projektbericht zugeschickt.

Weiter

*2. Seite*

Zur Bearbeitung der Aufgabe bitten wie Sie keine Hilfsmittel zur Hand zu nehmen. Die Bearbeitungszeit wird berücksichtigt, deswegen bitten wir Sie den Fragebogen zügig zu bearbeiten. Es werden Ihnen 20 Zahlenreihen vorgegeben, die nach einer bestimmten Regel aufgebaut sind. Die Aufgaben sind entsprechend den aufgeführten Beispielen zu lösen:

1. Beispiel: 2, 5, 8, 11, 14, ?

In dieser Reihe ist jede folgende Zahl um 3 größer als die vorgehende. Die Lösung dieser Aufgabe lautet: 17

2. Beispiel: 9, 7, 10, 8, 11, ?

In dieser Reihe werden abwechselnd 2 abgezogen und 3 dazu gezählt (oder als erste zwei Zahlen gelten die 9 und die 7 und abwechselnd wird auf diese 1 dazu gezählt). Die Lösung dieser Aufgabe lautet: 9

Beim Bearbeiten der Fragen tragen Sie bitte das Ergebnis ins freie Feld ein. Falls Sie eine der Aufgaben nicht lösen können, können Sie das Feld frei lassen oder versuchen zu erraten. Bitte schätzen Sie auch anschließend auf der vorgesehenen Skala ein, wie schwer Sie diese Aufgabe empfunden haben.

Weiter

*3. Seite*

Welches Geschlecht haben Sie?
O weiblich
O männlich

Wie alt sind Sie?
Ich bin ___ Jahre alt

Was machen Sie beruflich?

____________________

Weiter

*4. Seite*

Geben Sie die nächste Zahl ein:
2, 7, 13, 20, 28, _(?)_ 37

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*5. Seite*

Geben Sie die nächste Zahl ein:
3, 5, 7, 9, 11, _(?)_ 13

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*6. Seite*

Geben Sie die nächste Zahl ein:
1, 1, 3, 12, 84, _(?)_ 924

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*7. Seite*

Geben Sie die nächste Zahl ein:
107, 291, 475, 659, 843, _(?)_ 1027

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*8. Seite*

Geben Sie die nächste Zahl ein:
128, 254, 381, 507, 634, _(?)_ 760

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*9. Seite*

Geben Sie die nächste Zahl ein:

1, 1, 2, 6, 24, _(?)_ 120

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*10. Seite*

Geben Sie die nächste Zahl ein:
2, 2, 4, 6, 10, _(?)_ 16

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*11. Seite*

Geben Sie die nächste Zahl ein:
7, 7, 14, 42, 168, _(?)_ 840

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*12. Seite*

Geben Sie die nächste Zahl ein:
3, 3, 9, 36 252, _(?)_ 2772

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*13. Seite*

Geben Sie die nächste Zahl ein:
121, 144, 169, 196, 225, _(?)_ 256

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*14. Seite*

Geben Sie die nächste Zahl ein:
1, 4, 9, 16, 25, _(?)_ 36

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*15. Seite*

Geben Sie die nächste Zahl ein:
16, 48, 51, 153, 156, _(?)_ 468

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*16. Seite*

Geben Sie die nächste Zahl ein:
1, 2, 4, 7, 12, _(?)_ 20

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*17. Seite*

Geben Sie die nächste Zahl ein:
2, 3, 6, 18, 108, _(?)_ 1944

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*18. Seite*

Geben Sie die nächste Zahl ein:
2, 2, 4, 8, 32, _(?)_ 256

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*19. Seite*

Geben Sie die nächste Zahl ein:
103, 103, 206, 309, 515, _(?)_ 824

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*20. Seite*

Geben Sie die nächste Zahl ein:
1, 24, 47, 93, 162, _(?)_ 277

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*21. Seite*

Geben Sie die nächste Zahl ein:
4, 6, 12, 14, 28, _(?)_ 30

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*22. Seite*

Geben Sie die nächste Zahl ein:
5, 7, 10, 12, 15, _(?)_ 17

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*23. Seite*

Geben Sie die nächste Zahl ein:
237, 311, 386, 462, 539, _(?)_ 617

Bewerten Sie die Schwierigkeit der Reihe.
O sehr leicht O leicht O mittel O schwer O sehr schwer

Weiter

*24. Seite*

Haben Sie Hilfsmittel verwendet?
O Taschenrechner O Notizzettel für Zwischenrechnungen O Internetsuche O Fremde Hilfe

Weiter

*25. Seite*

Falls Sie an den Ergebnissen unseres Projekts interessiert sind, können Sie uns hier Ihre Email-Adresse hinterlegen. Diese werden anonym und unabhängig von den Testergebnissen erhoben.
Email-Adresse: ___________________________

Weiter

*26. Seite*

Danke für Ihre Teilnahme!
Wir möchten uns ganz herzlich für Ihre Mithilfe bedanken.

Kognitive Systeme, Otto-Friedrich-Universität Bamberg

## A.6. Source Code

```
module Series where

import MagicHaskeller
import MagicHaskeller.LibTH

seriesLib :: [Primitive]
seriesLib = $(p [|(seriesCL1to4 :: Int -> Int -> [(Int -> Int -> Int
   -> Int)] -> [Int], [] :: [(Int -> Int -> Int -> Int)], (:) :: (Int
    -> Int -> Int -> Int) -> [(Int -> Int -> Int -> Int)] -> [(Int ->
    Int -> Int -> Int)], (+) :: Int -> Int -> Int, (*) :: Int -> Int
   -> Int, square :: Int -> Int)|])

numbers :: [Primitive]
numbers = $(p [|(0 :: Int, 1 :: Int, 2 :: Int, 3 :: Int, 4 :: Int, 5
   :: Int, 12 :: Int, 22 :: Int, 24 :: Int, 74 :: Int, 126 :: Int,
   127 :: Int, 184 :: Int, 253 :: Int)|])

mseries :: ProgramGenerator pg => pg
mseries = mkPGOpt (options{constrL=True}) (seriesLib ++ numbers)

square :: Int -> Int
square n = n * n

seriesCL1to4 :: Int -> Int -> [(Int -> Int -> Int -> Int)] -> [Int]
seriesCL1to4 n0 n1 l | n0 <= 0    = n1 : rec 0 n1 0 l
                     | otherwise = n0 : n1 : rec n0 n1 0 l
        where rec n0 n1 i (h:t) = (h n0 n1 i) : rec n1 (h n0 n1 i) (i
             + 1) (t ++ [h])

filterSeries pred = filterThen pred (everything (mseries :: ProgGen))
```

**Listing A.1** — MH