Counting the number of Skolem sequences using inclusion-exclusion

Jeppe Winther Larsen
14-10-1985
jwl@itu.dk

Advisor: Thore Husfeldt
thore@itu.dk

August 2009
IT-university of Copenhagen
Contents

1 Introduction .......................... 1
   1.1 Historical background ............... 1
   1.2 Definitions and notation .......... 3
   1.3 Previous work ..................... 4
   1.4 Motivation and perspectives ...... 5

2 Theory ............................. 6
   2.1 Connection to perfect matchings .... 6
   2.2 Godfrey’s algorithm ............... 6
   2.3 The inclusion-exclusion principle .. 7
   2.4 New algorithm .................... 8
   2.5 Variants .......................... 9

3 Implementation ..................... 11
   3.1 Bit operations .................... 11
   3.2 Basics ............................ 11
   3.3 Optimising using binary Gray codes ...... 12
   3.4 Exploiting the reflections ........ 13
   3.5 Parallelization ................... 14
   3.6 Skipping update of the sum ........ 15
   3.7 Variants .......................... 15

4 Results ............................ 17
   4.1 The effect of the optimisations .... 17
   4.2 Godfrey’s versus ours ............. 17
   4.3 New values ........................ 19
   4.4 GMP performance penalty .......... 20

5 Concluding remarks ................. 21
   5.1 Further work ..................... 21
   5.2 Reflections on the project ...... 21

References .......................... 23

A Appendix ........................... 25
   A.1 Original project description ..... 25
   A.2 basic.c ............................ 26
   A.3 graycode.c ......................... 28
   A.4 reflections.c ...................... 31
   A.5 triplet.c .......................... 35
   A.6 generalsets.c ...................... 37
Abstract

We consider a problem from combinatorics known as Skolem sequences. In general a Skolem sequence over some set $A$ is a sequence $(a_1 \ldots a_{2n})$ where each $a_i$ occur exactly twice and is placed $a_i$ positions apart in the sequence. We want to count exactly the number of such sequences for a given $A$. We use the inclusion-exclusion principle to develop an algorithm that runs in $O(2^{2n})$ time and compare it to the very similar algorithm for the same problem by Mike Godfrey. We extend the algorithm to some variations of the problem. We implement the algorithm in C, analyse its performance in practice and give some new previously unknown number of solutions to some of the variations.
1 Introduction

In this thesis we consider a problem from combinatorics known as the Langford pairing problem or Skolem sequences, named after two mathematicians who have formulated this problem independently. The concept is to distribute a set of positive integers into a sequence of pairs where each pair is separated by a given number of positions. This problem has evolved both into a recreational mathematical puzzle and inspired research in other areas of combinatorics. Finding one solution to this kind of problem is computational easy, when the sets are of the form \([1, \ldots, n]\), whereas counting the number of solutions is hard. We seek to formulate an approach to count the number of solutions in exponential time.

1.1 Historical background

The Langford pairing problem was first introduced by the Scottish mathematician C. Dudley Langford in 1958:

"Years ago, my son, then a little boy, was playing with some coloured blocks. There were two of each colour, and one day I noticed that he had placed them in a single pile so that between the red pair there was one block, two between the blue pair, and three between the yellow. I then found that by a complete rearrangement I could add a green pair with four between them." [12]

Langford simplified the idea to numbers where each pair of a number \(n\) was separated by \(n\) other numbers. He gave examples of solutions for \(n = 3, 4, 7, 9, 11, 12\) and 15, and asked for a theoretical treatment. A year later the problem was revisited by C. J. Priday and Roy O. Davies [17] who proved for which \(n\)'s such solutions exist and introduced the idea of hooked sequences, which puts a single 0 or gap in the sequence, and looped sequences, satisfying the conditions set by Langford. Davies also proposed the counting aspect:

"It would be interesting to know roughly how many different solutions of Langford’s problem exist for large \(n\). They are surprisingly numerous even for \(n = 7\): namely, 25 distinct perfect sequences, not counting as distinct a sequence and the same one in reverse order. For \(n = 3\) and \(n = 4\) there is only one solution." [17]

While working on Steiner triple systems, an object studied in combinatorial design, Norwegian mathematician T. H. Skolem actually proposed a very similar problem a year before in 1957, which was noted by Davies:

"A study of the structure of some triple systems of Steiner led me to consider the following problem: Is it possible to distribute the number 1, 2, \ldots, 2n in \(n\) pairs \((a_r, b_r)\) such that we have \(b_r - a_r = r\) for \(r = 1, 2, \ldots, n\)?" [19]

This can be viewed the same way as Langford’s problem, with the slight variation that the pairs are \(n\) positions apart. This variation was actually later rediscovered by R. S. Nickerson in 1967 [14].

It seems that the Langford problem has been more widespread as a mathematical recreational puzzle such as in Martin Gardner’s popular puzzle books,
whereas Skolem sequences have inspired several research papers and the sequences have been applied to many other kinds of problems. This is not all that surprising since Langford formulated the problem just as a recreational puzzle, whereas Skolem used these kind of sequences to create Steiner triple systems in his original paper.

The counting aspect of the Langford problem has led to a computational challenge of finding values for large values of $n$, and John E. Miller has dedicated a webpage to the problem where he documents these findings [13]. Table 1 is taken from that webpage, and sums up all the findings of the numbers of Langford and Skolem sequences. We have slightly modified it to make it consistent with our notation (see section 1.2).

<table>
<thead>
<tr>
<th>Problem</th>
<th>Date</th>
<th>Person</th>
<th>Computer</th>
<th>Time</th>
<th>Language</th>
<th>Where</th>
</tr>
</thead>
<tbody>
<tr>
<td>L(3-4)</td>
<td>?</td>
<td>C. Dudley Langford</td>
<td>Hand</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>L(7)</td>
<td>1959</td>
<td>Roy O. Davies</td>
<td>Hand</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>L(7-8)</td>
<td>May-67</td>
<td>Dave Moore</td>
<td>TRW-130</td>
<td>5m,40m</td>
<td>FORTRAN</td>
<td>Rolls Royce</td>
</tr>
<tr>
<td>L(7-8)</td>
<td>Nov-67</td>
<td>Glen F. Stahly</td>
<td>IBM 1130</td>
<td>?</td>
<td>FORTRAN</td>
<td>Gonzaga</td>
</tr>
<tr>
<td>L(7-8)</td>
<td>Nov-67</td>
<td>John Miller</td>
<td>IBM 1130</td>
<td>?</td>
<td>FORTRAN</td>
<td>Gonzaga</td>
</tr>
<tr>
<td>L(7-12)</td>
<td>Nov-67</td>
<td>E. J. Groth</td>
<td>SDS 930</td>
<td>&lt;d</td>
<td>FORTRAN</td>
<td>Motorola</td>
</tr>
<tr>
<td>L(11-12)</td>
<td>1968?</td>
<td>John Miller</td>
<td>IBM 1130</td>
<td>?</td>
<td>Asm</td>
<td>Gonzaga</td>
</tr>
<tr>
<td>L(15)</td>
<td>Sep-80</td>
<td>John Miller</td>
<td>VAX 11/780</td>
<td>?</td>
<td>Pascal</td>
<td>L&amp;C</td>
</tr>
<tr>
<td>L(15)</td>
<td>Feb-87</td>
<td>Frederick Groth</td>
<td>Commodore 64</td>
<td>15.5 d</td>
<td>Asm</td>
<td>Home</td>
</tr>
<tr>
<td>L(16)</td>
<td>Feb-87</td>
<td>Frederick Groth</td>
<td>Commodore 64</td>
<td>122.4 d</td>
<td>Asm</td>
<td>Home</td>
</tr>
<tr>
<td>L(15)</td>
<td>Jul-89</td>
<td>Andrew Burke</td>
<td>Cogent XTM</td>
<td>?</td>
<td>C</td>
<td>OGI</td>
</tr>
<tr>
<td>L(16)</td>
<td>Jul-89</td>
<td>Andrew Burke</td>
<td>Cogent XTM</td>
<td>120h</td>
<td>C</td>
<td>OGI</td>
</tr>
<tr>
<td>L(19)</td>
<td>May-99</td>
<td>Rick Groth Team</td>
<td>Mac/Pentium</td>
<td>2 mo</td>
<td>C</td>
<td>Distributed</td>
</tr>
<tr>
<td>L(19)</td>
<td>Jul-99</td>
<td>John Miller</td>
<td>DEC Alpha</td>
<td>2.5 years!</td>
<td>C</td>
<td>L&amp;C,...</td>
</tr>
<tr>
<td>L(19)</td>
<td>Mar-02</td>
<td>Ron van Bruchem</td>
<td>Pentium</td>
<td>~6H</td>
<td>FORTRAN</td>
<td>Godfrey’s</td>
</tr>
<tr>
<td>L(20)</td>
<td>Feb-02</td>
<td>Godfrey/van Bruchem</td>
<td>AMD/Pentium</td>
<td>1 Week</td>
<td>FORTRAN</td>
<td>UMIST/home</td>
</tr>
<tr>
<td>L(23)</td>
<td>Apr-04</td>
<td>Krajek Team</td>
<td>Sun/Intel</td>
<td>4 days</td>
<td>Java/CONFIT</td>
<td>REIMS</td>
</tr>
<tr>
<td>L(24)</td>
<td>Apr-05</td>
<td>Krajek Team</td>
<td>12-15 processors</td>
<td>3 months</td>
<td>Java/CONFIT?</td>
<td>REIMS?</td>
</tr>
</tbody>
</table>

Table 1: Table from Miller’s webpage [13] with names and information about each finding of the known numbers of Langford and Skolem sequences.

Before 2002 the method used was simply to generate all $2n!$ permutations of the sequence and count the valid ones. Seemingly no work has been done to apply a more structured theoretical approach to the problem, rather than
simple bruteforce. But in early 2002 physicist Mike Godfrey found an algebraic
method to count the number of Langford sequences in exponential time:

"Because an exact calculation seemed difficult, I began to look for an
analytical method to estimate the pre-exponential factor in my crude
formula, so that at least the behaviour of \( L(2, n) \) would be known
for large \( n \). That was the motivation for investigating algebraic
expressions from which \( L(2, n) \) might be extracted, the hope being
that for large \( n \) the extraction could be done relatively easily, by
asymptotic evaluation of an integral, for example." [5]

Godfrey used his algorithm to compute the number of Langford sequences for
\( n = 19 \) in only six hours, where the previous calculation three years earlier had
taken two and a half years. Godfrey was then able to compute the values for
\( n = 20 \) Langford sequences, and \( n = 20, 21 \) for the Skolem variant using about a
week worth of computation time. His approach has later let to the computation
of the number of Langford sequences for \( n = 23, 24 \) using multiple processors
[13].

In his latest volume of the popular *The Art of Computer Programming* series
on combinatorial algorithms Donald E. Knuth has chosen the Langford pairing
problem as an introduction to the subject [7], which he uses as a foundation
for introducing five typical interesting aspects of combinatorics: existence, con-
struction, enumeration, generation and optimization. He also uses Godfrey’s
method in some of his exercises for the enumerative aspect, and this served as
a starting point for this thesis. Also, the topic in general as well as the enum-
eration perspective, has a whole chapter written by N. Shalaby in the *Handbook
of Combinatorial Designs* [18].

1.2 Definitions and notation

A Skolem sequence is a sequence \((a_1, \ldots, a_{2n})\) of positive integers over the set
\(\{1, \ldots, n\}\) where each \(a_i\) occur exactly twice and satisfying the condition that
these two is \(a_i\) positions apart. Another definition defines it as a sequence of \(2n\)
integers satisfying the two conditions: (1) for every \(k \in \{1, \ldots, n\}\) there exist
exactly two elements \(a_i\) and \(a_j\) from the sequence such that \(a_i = a_j = k\), and
(2) that \(|i - j| = k\). We call it a sequence of order \(n\) [18]. An example of a
Skolem sequence of order 4 is \((2, 3, 2, 4, 3, 1, 4)\).

A variant called *extended* Skolem sequences allow a single 0 in the sequence.
If this position in the sequence of length \(2n+1\) is fixed it is called a *hooked* Skolem
sequence. The definition from [18] places the hook at position \(2n\). When the
hook is placed in the middle of the sequence (that is at position \(n+1\)), it is
known as a *Rosa* or *split Skolem* sequence. An example of a hooked Skolem
sequence of order 3 is \((3, 1, 1, 3, 2, 0, 2)\).

These variants can also be generalised to the Langford sequences, where the
condition is instead that each \(a_i\) has \(a_i\) numbers between them. It can also
be viewed as a Skolem sequence over the set \(\{2, \ldots, n+1\}\). An example of a
Langford sequence of order 4 is \((4, 1, 3, 1, 2, 4, 3, 2)\).

Normally these sequences are considered using pairs, but the prob-
lem can also be formalised with triplets, quads and so forth. For
triplets we just have a sequence of \(3n\) numbers, but with the same re-
requirements. An example of a Langford triplet sequence of order 9 is (1, 9, 1, 6, 1, 8, 2, 5, 7, 2, 6, 9, 2, 5, 8, 4, 7, 6, 3, 5, 4, 9, 3, 8, 7, 4, 3).

The general case is to consider these sequences over the set \( \{1, \ldots, n\} \), but it also works with arbitrarily sets such as \( \{2, 3, 5, 6\} \) and even for multisets \( \{2, 2, 3, 5\} \). For example a Skolem sequence over \( \{2, 3, 5, 6\} \) is (5, 6, 2, 3, 2, 5, 3, 6).

We want to count the number of such sequences for a given \( n \) and we will use the following notation, along with their id in the *The On-Line Encyclopedia of Integer Sequences* [20]:

\[
S(n) \quad \text{Skolem sequences, A4075} \\
S_k(n) \quad \text{hooked Skolem sequences, A4076 when } k = 2n \\
S_*(n) \quad \text{extended Skolem sequences, A4077} \\
L(n) \quad \text{Langford sequences, A014552} \\
L_k(n) \quad \text{hooked Langford sequences} \\
L_*(n) \quad \text{extended Langford sequences} \\
L^3(n) \quad \text{Langford triplets, A59107} \\
S^3(n) \quad \text{Skolem triplets, A59108}
\]

**Existence**

A Skolem sequence \( S(n) \) exists if and only if \( n \equiv 0, 1 \pmod{4} \), and \( n \equiv 0, 3 \pmod{4} \) for \( L(n) \). A hooked Skolem sequence \( S_{2n}(n) \) only exists if \( n \equiv 2, 3 \pmod{4} \), and \( n \equiv 1, 2 \pmod{4} \) for \( L_{2n}(n) \), whereas the extended sequences \( S_*(n) \) and \( L_*(n) \) exists for all \( n \). The Rosa sequence \( S_{n+1}(n) \) only exists if \( n \equiv 0, 3 \pmod{4} \) and \( n \neq 1 \) [18].

**Reflected sequences**

We need to make a strict definition on how we count the number of such sequences. We define reflected sequences as distinct, so for example (2, 3, 2, 4, 3, 1, 1, 4) is not the same as (4, 1, 1, 3, 4, 2, 3, 2). This is the same approach used by Shalaby [18], but the previous work on \( L(n) \) did not count the reflected solutions thus halving the number of solutions and Miller’s webpage [13] also uses this approach.

### 1.3 Previous work

Besides Godfrey’s work there has not been much research on counting the numbers of Langford and Skolem sequences, and Godfrey has not even published anything about his algorithm, besides his letter of explanation on Miller’s webpage [5]. However, his method and some of the optimisations he proposes has been, in our opinion better, described by Knuth [7]. The latest work on the calculations of \( L(23) \) and \( L(24) \) was done by French computer scientists led by Michaël Krajecki as part of research in parallelization, where the Langford problem was modelled as CSP [9] [10] [11].
In other aspects the Skolem sequences have led to more theoretical research in combinatorics and been applied to many kinds of structures. Nevena Francetić and Eric Mendelsohn has published a very thorough survey of Skolem sequences, many of its variants and the various areas it has been applied to, such as graph labelling and triple systems [4]. A connection to Hefter’s first and second difference problem, a problem that asks to partition \(\{1, \ldots, 3n\}\) into \(n\) triples \((a_i, b_i, c_i)\) with some conditions, has been showed by G. K. Bennett, Mike J. Grannell and Terry S. Griggs, who also give exponentially lower bounds on the number of Skolem sequences, which in turn gives a lower bound on number of solutions to Hefter’s difference problem [1]. Gustav Nordh has studied the generalization of the problem where he considers other sets than those of the basic form of \(\{1, \ldots, n\}\) and defines a set that can be partitioned into a valid Skolem sequence as a perfect Skolem-set [16]. He has also shown that in general, the problem of deciding whether a set is a perfect Skolem-set is \(NP\)-complete [15].

We shall in particular build on the work done by Godfrey, using Knuth’s descriptions, and we will take inspiration from Thore Husfeldt’s and Andreas Björklund work on applying the inclusion-exclusion principle to enumerative combinatorics [2].

1.4 Motivation and perspectives

The motivation for this thesis is driven by the lack of theoretical treatment of the enumeration aspect of Skolem and Langford sequences. While in other areas the sequences have been well studied, the enumeration aspect appears mostly as a sort of hobby or merely as a benchmark for technical programming ideas. Mike Godfrey made a substantial improvement with his algorithm for counting the sequences using exponential time, but without proper documentation, his algorithm will probably remain as it is without further study. We hope to open the problem further by using well known principles from combinatorics to model and prove the correctness of our algorithm, and in doing so extending both our and Godfrey’s algorithm to the variants of the problem.
2 Theory

In the following sections we explain the necessary preliminaries and theoretical aspects behind the Skolem sequences and the algorithm we develop to enumerate the number of such sequences.

2.1 Connection to perfect matchings

Skolem sequences can be viewed as a case of perfect matchings for specific kinds of graphs. A perfect matching in a graph $G = (V, E)$ is a subset $M \subseteq E$ of disjoint edges that cover $V$.

First we need to model a Skolem sequence the same way as Skolem did it in his original paper, by having a set of ordered pairs $(a_i, b_i)$ such that $b_i - a_i = i$ with all $i$’s giving the set $\{1, \ldots, n\}$. Thus $a_i$ and $b_i$ corresponds to two positions of a number in the sequence and $i$ is the value of this number. So the Skolem sequence $(2, 3, 2, 4, 3, 1, 1, 4)$ gives the following set of pairs $\{(1, 3), (2, 5), (4, 8), (6, 7)\}$. Then given a graph $G = (V, E)$ that has at least one perfect matching $M$, we can view the set of edges $(u, v) \in E$ as the set of pairs $(u, v)$ in a Skolem sequence, where the set of vertices $V$ is the set of numbers in the pairs. Then the matching $M$ is a set of pairs giving a Skolem sequence. We are mostly interested in the special case where the graph has $2n$ numbered vertices of $\{1, \ldots, 2n\}$, where each vertex $v_i$ is connected to the vertices between $i - n \leq i \leq i + n$ due to the rules of the Skolem sequences, and satisfying the condition that for all $(u, v) \in M$ that $\bigcup |u - v| = \{1, \ldots, n\}$. Figure 1 (a) shows such a graph for $n = 4$ and (b) one of its perfect matching giving the Skolem solution $\{(1, 3), (2, 5), (4, 8), (6, 7)\}$.

![Figure 1: (a) a graph for Skolem sequences of order 8 (b) one of its solutions - a perfect matching](image)

2.2 Godfrey’s algorithm

There is no official paper on Godfrey’s algorithm, but is has been described by Knuth [7] and Krajecki et al. [11]. Godfrey’s own description of his method can be found on Miller’s webpage [5]. Here is the algorithm as formulated by...
Krajecki et al.:

\[ 2^{2n+1}L(n) = \sum_{(x_1, \ldots, x_{2n}) \in \{-1,1\}^{2n}} 2^n \prod_{i=1}^{2n} x_i \prod_{k=1}^{2n-n-1} x_k x_{k+i+1} \]  (1)

This gives an \( O(2^{2n}n^2) \) algorithm which is much better than the previous used naive approach of \( 2n! \). However, this approach only counts the number of solutions, it is not able to give the actual solutions. Knuth claims in his exercise this works because:

"When the products are expanded, we obtain a polynomial of \( (2n - 2)!/(n - 2)! \) terms, each of degree \( 4n \). There’s a term \( x_1^2 \cdots x_{2n}^2 \) for each Langford pairing; every other term has at least one variable of degree 1. Summing over \( x_1, \ldots, x_{2n} \in \{-1,1\} \) therefore cancels out all the bad terms, but gives \( 2^{2n} \) for the good terms." [7]

However, it is somewhat unclear what is going on and exactly why this works. We will instead apply the inclusion-exclusion principle to the problem and as it turns out, it gives us a very similar algorithm. In some sense Godfrey’s algorithm works by the inclusion-exclusion principle, though it is not formulated as such. Doing so also allows us to extend the algorithm to triplets, which does not seem possible using Godfrey’s algorithm, and the other variants that turns out to be possible for both, but have not been considered with Godfrey’s method before.

2.3 The inclusion-exclusion principle

Husfeldt and Björklund has proposed to use the inclusion-exclusion principle to various counting problems [2]. We introduce the principle with inspiration from [3] and how we can apply it to Skolem sequences.

The principle of inclusion-exclusion deals with set theory. We consider the number of elements the two sets \( A \) and \( B \) have together, that is \( |A \cup B| \). We can do that by adding the size of the two sets then subtracting the size of the intersection:

\[ |A \cup B| = |A| + |B| - |A \cap B| \]

This is what is known as the principle of inclusion-exclusion. It can of course be generalised to more sets:

\[ |A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C| \]

The right-hand side contains all possible intersections of the three sets, subtracting when the number of sets is even and adding when uneven. To generalise this further, we define our universe (all elements in every set) as \( X \) and \((A_1, A_2, \ldots, A_n)\) is some subsets of \( X \), then \( A_I = \bigcap_{i \in I} A_i \) where \( I \) is the index set \( \{1, \ldots, n\} \), and \( A_\emptyset = X \) because intersecting with no sets should give the largest set. We want to count the number of elements that lie in none of the subsets \( A_I \), this gives us:

\[ |A_1 \cup \ldots \cup A_n| = \sum_{I \subseteq \{1, \ldots, n\}} (-1)^{|I|} |\bigcap_{i \in I} A_i| \]  (2)

Suppose that an element \( x \in X \) lie in none of the sets in \( A_I \) then its contribution is 1 to the left hand side of the equation. On the right hand side it only
contributes 1 to the sum when $I = \emptyset$ and with positive sign since $(-1)^0 = 1$. Otherwise, if an element $x \in A_I$ then we want its contribution to be 0, so if $x \in A_i$ for all $i$ in the subset $J \subseteq I$, then the contribution of $x$ is:

$$\sum_{J \subseteq I} (-1)^{|J|} = \sum_{i=0}^{|J|} \binom{|J|}{i} (-1)^i$$

which we want to be 0. The quantity $\binom{|J|}{i}$ is the combinations of $|J|$ objects taking $i$ at a time. If we expand (3) we get $\binom{|J|}{0}(-1)^0 + \binom{|J|}{1}(-1)^1 + \binom{|J|}{2}(-1)^2 \ldots \binom{|J|}{i}(-1)^i$. To show that this actually gives 0, we need the binomial theorem:

$$\sum_{i=0}^r \binom{r}{i} a^i b^{r-i} = (a + b)^r$$

This look similar to (3), but in our case of we have $a = -1$ and $b = 1$, so we can continue the expression as:

$$\sum_{i=0}^{|J|} \binom{|J|}{i} (-1)^i 1^{|J| - i} = (-1 + 1)^{|J|} = 0$$

when $|J| \neq 0$. Hereby elements that belongs to no set $A_i$ contributes 1 to the sum and elements in one or more $A_i$ contributes 0. The total sum is therefore the number of elements in our universe $X$ that lie in none of the subsets $A_I$.

To connect this to the Skolem sequences, view $A_i$ as all arrangements of the numbers that avoid the position $i$. Then $A_1 \cup \ldots \cup A_n$ is the number of ways to arrange the numbers that avoid no positions, hereby solutions to our problem.

### 2.4 New algorithm

In the previous section we established how we can model the enumeration of Skolem sequences with inclusion-exclusion. Now we need to use that knowledge to develop an actual algorithm for the counting problem. Husfeldt and Björklund applied the inclusion-principle to problems such as counting the number of disjoint set covers and number of perfect matching [2], which is very similar to our problem, and expressed these kind of problems on the following form:

$$S(n) = \sum_{X \subseteq \{1, \ldots, 2n\}} (-1)^{|X|} a(X)$$

where $X$ denotes the positions to be avoided and $a(X)$ is a function that counts the solutions $S(n)$ avoiding the positions in $X$. So with inspiration from Godfrey’s algorithm (1) we can let $X$ be a binary sequence $(x_1, \ldots, x_{2n}) \in \{0, 1\}$ where every 0 in the sequence is a disallowed position. This lets us express the positions the following way: if for example our subset $X = \{2, 3, 7, 8\}$ we encode it as 10011100. Then we can write $a(X)$ as follows:

$$a(X) = \prod_{k=1}^n \sum_{j=1}^{2n-k} x_j x_{j+k}$$

where $X$ denotes the positions to be avoided and $a(X)$ is a function that counts the solutions $S(n)$ avoiding the positions in $X$.
a(X) can be seen as the function that checks the given subset according to the rules for these kind of sequences, where for each k every possible way of pairing k in the sequence is checked and added to the sum. To illustrate this further, see figure 2 which shows what actually happens in the summation of a(X) - here illustrated with n = 3 for one X. The binary sequence has length 2n and for each k all the ways to place the numbers k positions apart is added up.

<table>
<thead>
<tr>
<th>k Positions in X compared in the summation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

Figure 2: A sample run of a(X) with n = 3 that shows how every possible way of placing the numbers in the sequence is checked and summed up for each k.

Each a(X) takes O(n^2) time and we need to sum over 2^{2n} subsets, giving the same running time as Godfrey’s algorithm (1). This gives an algorithm for S(n) which also can be easily applied to L(n), just replace the rightmost summation with \( \sum_{j=1}^{2n-k} x_j x_{j+k+1} \), and it looks surprisingly similar to Godfrey’s algorithm. And we can also see now that Godfrey’s approach works due to the same principles, but using the inclusion-exclusion principle has given us a better foundation for understanding and extending the algorithm, and our way of counting the exact number of sequences for a given n, rather than 2^{2n} times the number of sequences, is much less confusing.

2.5 Variants

Applying to hooked sequences

Hooked sequences can be seen as a sequence with one position that always must be avoided. So to find for example \( S_{2n}(n) \) we let 2n ∈ X, and for the Rosa variant simply n + 1 ∈ X. The same can be done with Godfrey’s algorithm by avoiding the use of \( x_{2n} \).

Applying to extended sequences

For the extended sequences the same reasoning as for the hooked sequences apply. We need to sum over all \( X \subseteq \{1, \ldots, 2n + 1\} \). The naive way would be to run the algorithm for \( S_1(n) + S_2(n) + \ldots + S_{2n}(n) \) (though only computation up to \( S_{n+1}(n) \) is needed since \( S_i(n) = S_{2n-i+1}(n) \) due to the reflections), but we can do it with just a single run of the algorithm only doubling the running time (since we need to sum over \( 2^{2n+1} \) subsets instead of \( 2^{2n} \)). The summation in a(X) considers all the ways to place the n numbers, but for the extended sequences we also need to take into account a single 0. Therefore we sum over \( 2^{2n+1} \) subsets of X. The single 0 can be placed on every allowed position in the subset, so for each considered X we need to also multiply with the numbers of
1’s in $X$. Therefore we simply need to include the factor $|X|_1$ in (6):

$$S_+(n) = \sum_{X \subseteq \{1, \ldots, 2n+1\}} (-1)^{|X|} |X|_1 a(X) \quad (8)$$

It is not clear if and why this also applies to Godfrey’s algorithm, but it turns out it does, when we implement this in section 3.7.

**Applying to triplets**

Counting the triplet variant $S^3(n)$ is very straightforward. We just need to sum over all $X \subseteq \{1, \ldots, 3n\}$ and add and extra term to $a(X)$ accordingly. This gives us:

$$S^3(n) = \sum_{X \subseteq \{1, \ldots, 3n\}} (-1)^{|X|} n \prod_{k=1}^{3n-2k} x_j x_{j+k} x_{j+2k} \quad (9)$$

It might be possible to apply Godfrey’s algorithm to triplets as well, but it is not as straightforward. The fact that it uses $+1$ and $-1$ in the summation and produces the number of solutions times $2^{2n}$ makes it unclear if the cancelling of the bad terms still works, and as will be shown in section 3.7, we were unsuccessful in making the implementation work with Godfrey’s method.

**Support for general sets**

So far we have only looked at Skolem sequences over sets of the form $\{1, \ldots, n\}$, but these kind of sequences can be made for any sets and even multisets. $a(X)$ needs to be modified slightly to check for the positions given by the set. We define our set $T$ and $t_k$ as the $k$’th element from $T$. Then we can write:

$$S(T) = \sum_{X \subseteq \{1, \ldots, 2n\}} (-1)^{|X|} \prod_{k=1}^{2n-2t_k} x_j x_{j+t_k} \quad (10)$$

where $n = |T|$. This also solves the $NP$-complete problem of deciding if a set $T$ is also a perfect Skolem-set in exponential time. If $S(T)$ gives a value over 0, then $T$ is a perfect Skolem-set. Now we see the payoff of using the principle inclusion-exclusion as a model for our algorithm. Instead of just having a way to count the normal Skolem and Langford sequences, we can now take any set or multiset and (10) will count the number of permutations over that set into an allowed sequence.
3 Implementation

We have implemented the algorithms with various optimization techniques including parallelisation and with support for the extended and hooked sequences. All implementations are written in standard C and the GMP library has been used to handle the very large integer values needed to compute the sum. The source code can be located at http://itu.dk/people/jwl/skolem/

3.1 Bit operations

Since we use some bitwise operations in our implementation of the algorithm, we will briefly introduce the concept here.

Bitwise operations operate on two bit patterns, compare the individual bits producing a new bit pattern. The interesting operations for us is AND and XOR. Bitwise AND does a logical conjunction, that is a comparison of two boolean values that yields true only if both values are true, on each bit in the pattern. In most programming languages, including C, this is denoted by \\&. For example 101011 & 001011 = 000011. We will use the AND operation for comparing the terms in \(X\).

Another operation we will use is the exclusive-or operation called XOR, which is a logical exclusive disjunction that also compares two boolean values and yields true if one of the values is true, but not both. This operation is denoted by \(\oplus\) and in C by \(^{\wedge}\). We shall use this to implement the binary Gray code.

In our algorithm when we need to iterate over all positions in \(X\), bit-shifting is very helpful. It is not a bitwise operation as the two others, since it only operates on a single bit pattern. A bit-shift moves the whole bit pattern to the right or left, discarding bits that are shifted out and zeroes are added in the other end. A left shift, denoted by \(<\ll\>, multiplies the value by \(2^n\) and a right shift, denoted by \(<\gg\>, is the same as dividing by \(2^n\), where \(n\) is the number of positions the bits are shifted. For more on bit operations we refer to [8] and [21].

3.2 Basics

The basics of our algorithm (7) is to compute \(S(n)\) in one big loop where we iterate over all \(2^{2n}\) binary permutations, compute the value for each by iterating through all positions in the binary sequence and add its sum to the total summation. It goes through the following repeating steps: (1) take next integer \(x\) from the loop in the range 0 to \(2^{2n}\) (2) for each \(k\) in 1 to \(n\) do (3) set the sum \(s_k\) to 0, for each \(j\) in 1 to \(2n - k\) increment \(s_k\) with \(x_jx_{j+k}\) (4) calculate the product of all \(s_k\) (5) count the number of 1’s in \(i\), if odd subtract to the total sum else add. Each part of the algorithm and its corresponding lines of code is shown in figure 3.

To add a few explanations to the code, the CHECK BIT macro simply does a binary AND on the two variables, and return 1 on true and 0 on false (when using Godfrey’s method it is simply 1 and −1 instead), and the count_bits() function is a constant time function that counts the number of bits in an integer, taken from [21]. The variables tmp and bigsum are of type mpz_t from the GMP library and their values are updated using the appropriate mpz function calls.
\(X \subseteq \{1, \ldots, 2n\}\) for \((x = 0; x < (1 \text{ LL } < (2n)); x++)\)

\[
\sum_{j=1}^{2n-k} x_jx_{j+k}
\]

\[
\prod_{k=1}^{n} \text{mpz} \_\text{set} \_\text{si}(\text{tmp}, \text{prod}[0]);
\]

\[
\text{mpz} \_\text{mul} \_\text{si}(\text{tmp}, \text{tmp}, \text{prod}[j]);
\]

\[
\sum_{i=1}^{(-1)^{|\text{X}|}} \text{if(\text{count} \_\text{bits}(x)) } \text{mpz} \_\text{sub}(\text{bigsum}, \text{bigsum}, \text{tmp});
\]

\[
\text{else mpz} \_\text{add}(\text{bigsum}, \text{bigsum}, \text{tmp});
\]

Figure 3: Basic implementation

Also note how we use bitshifting operations on the \(swp\) variable in order to check the bits according to the \(a(X)\) function. So \(swp\) checks the \(x_j\) part in the summation and \(swp<<k\) represents \(x_{j+k}\). Complete source is located in appendix A.2.

### 3.3 Optimising using binary Gray codes

We can greatly reduce the running time of our algorithm by iterating over all the subsets \(X\) in Gray code order, allowing us to do each \(a(X)\) in \(O(n)\) time instead of \(O(n^2)\). In the normal way to count in binary several bits might be changed at the same time, but if we could ensure that only a single bit changed in each iteration, we would not need to check every position in the sequence for every \(k\), only update the previous sum accordingly for the position of the bit that was changed. Generating bit patterns in such a way is called reflected binary code or Gray codes, named after Frank Gray who filed a patent for this method in 1947. For more information on Gray codes we refer the reader to [6] and [21].

We need to generate the same \(2^{2n}\) binary codes as before, but in such an order that only one bit changes in each iteration and so we can identify which bit was changed. We do that by extracting the least significant bit, or right most bit, from the counter \(x\) in the big loop by doing a binary AND on its negative value and assigning this position to \(\text{min}\), which we use to update our Gray code integer \(\text{gray}\) by doing a binary XOR assignment \(gray = gray \oplus \text{min}\). The summation loop is changed to just make the changes on the sum for each \(k\) updating the sum from the previous summation, rather than starting over each time. So we maintain the sums in an array of size \(k\), the changed bit at position \(\text{min}\) is the starting point in the Gray code integer \(\text{gray}\) and depending on whether the bit was changed from 0 to 1 or the reverse, we add or subtract to the previous sum. Using the Gray code also allows us to determine the size of \(X\), the number of bits setted, more efficiently than before. Since we just change one bit for each iteration we shift between odd and even sized subsets, so when
the iteration number in our loop is odd we subtract from the total sum and add when even. Figure 4 shows how the Gray code works and the new code for the summation.

\[
\begin{array}{c}
101100 \\
\text{AND} \ 010100 \\
000100 \\
\end{array}
\min = x \& -x;
\]

\[
\begin{array}{c}
111110 \\
\text{XOR} \ 000100 \\
111010 \\
\end{array}
\gray \^= \gray \& \min;
\]

\[
\begin{array}{c}
111010 \\
swp=\min\gg1;
\end{array}
\text{for (k=0; k < n \&\& swp; k++, swp>>=1)}
\{
\prod[k] -= \text{CHECK_BIT}(\gray, \swp);
\}
\]

\[
\begin{array}{c}
111010 \\
swp=\min<<1;
\end{array}
\text{for (k=0; k < n \&\& swp < \text{bigN} ; k++, swp<<=1)}
\{
\prod[k] -= \text{CHECK_BIT}(\gray, \swp);
\}
\]

Figure 4: Summation using Gray code

Notice how we update the sum for each \(k\) directly instead of computing the sum from scratch. The \(\min\) value now corresponds to the term \(x_j\) and \(\swp\) iterates for each \(k\) first to the right of the changed bit and then for each \(k\) to the left. In the above example the bit changed from 1 to 0, therefore we subtract from the sum. If the change was from 0 to 1 we would have added to the sum. Note that \(\text{bigN}\) is the value for \(2^n\). Since using the Gray code makes it possible to predict the number of setted bits, we can instead of calling the \text{count_bits()}\ function, simply check if the iteration in the loop is odd or even by \(x\&1\).

We came over a strange discovery when implementing this with Godfrey’s method. In the basic implementation it worked as expected using 1 and \(-1\), but not with the Gray code. By chance we found out that we needed to use 2 and \(-2\) instead in the \text{CHECK_BIT} macro to make it work. This is probably needed because going from 1 to \(-1\) is a difference of 2, but this confusing oddity is one of those things that confirms us in the rightness of our approach for our algorithm. Complete source is located in appendix A.3.

### 3.4 Exploiting the reflections

There is a great deal of symmetry in the binary Gray codes we compute the sum of, so we could exploit this to reduce the number of Gray codes. Specifically the sum of a reflected code is the same - that is 01010011 will give the same sum as its reflection 11001010. To exploit this we take inspiration from Knuth [7] who explains how to do this in phases.

The idea is to fix a bit at position \(p\) where \(p\) denotes the phase:

```c
gray = UULL;
if(p!=n)
    gray |= 1UL<<p;
```
for each phase and $2^p$ number of outer loops $q$ that fixes the bits at position $x_q$ and $x_{2n-q}$:

```c
for (j=0; j < p; j++)
if(outer & 1LL<<j)
    gray |= 1LL<<j | 1LL<<(NPOS-j-1);
```

and an inner loop that chooses $2^{2n-2p-2}$ terms in Gray code order:

```c
len = 1LL<<(2*(n-p-1));
```

where `

| len denotes the number of iterations in the inner loop which is the same as explained in section 3.3. The last phase covers the palindromic cases such as 10011001 and 10000001. Figure 5 list a part of the Gray codes generated using this method for $n = 4$.

<table>
<thead>
<tr>
<th>Outer</th>
<th>Phase 0</th>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3</th>
<th>Phase 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000000</td>
<td>01000000</td>
<td>00100000</td>
<td>00010000</td>
<td>00000000</td>
</tr>
<tr>
<td></td>
<td>11000000</td>
<td>01100000</td>
<td>00110000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11100000</td>
<td>01110000</td>
<td>00111000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>10100000</td>
<td>01010000</td>
<td>00101000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>10000010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11000001</td>
<td>10100001</td>
<td>10010001</td>
<td>10000001</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11100001</td>
<td>10110001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>11000101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>01100010</td>
<td>01010010</td>
<td>01000010</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>01110010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11100011</td>
<td>11010011</td>
<td>11000011</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>11110011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>11101011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11111111</td>
</tr>
</tbody>
</table>

Figure 5: Reflected binary Gray codes in phases for $n = 4$

This method almost halves the number of subset $X$ that needs to be considered. Only almost because of the palindromic cases. Complete source code is in appendix A.4.

### 3.5 Parallelization

The algorithm is easily parallized due to the fact that each sum can be computed independently, so we can easily split the range of the total $2^{2n}$ binary codes into several independent processes and just put the result together at the end. To avoid rounding errors we only allow for splitting into $\log_2$ divisible number of processes - 2, 4, 8 and so forth. When we know the total number of terms and how many processes we split into, we can get the number of terms each process
needs to consider as well as its starting point. When using the Gray code the sum just have to be computed once using the basic naive method for the starting point in the range. Figure 6 shows an example where \( n = 8 \) and the number of processes is 4.

<table>
<thead>
<tr>
<th>Process</th>
<th>Start</th>
<th>Stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>16384</td>
</tr>
<tr>
<td>1</td>
<td>16384</td>
<td>32768</td>
</tr>
<tr>
<td>2</td>
<td>32768</td>
<td>49152</td>
</tr>
<tr>
<td>3</td>
<td>49152</td>
<td>65536</td>
</tr>
</tbody>
</table>

length = bigN >> num_proc;  
start = proc_id*length;  
stop = start + length;

Figure 6: The ranges for 4 processes on \( S(8) \)

Letting \( \text{num\_proc} \) be in \( \log_2 \) allows us to get the length for each process by a binary right shift on the total number of terms in \( \text{bigN} \). In the above example with 4 processes, \( \text{num\_proc} \) is the \( \log_2 \) of 4 which is 2, and remember that a right shift is the same as a division by \( 2^n \), we get the range split appropriately into 4 parts. And since \( \text{bigN} \) is a power of 2, we avoid annoying rounding errors.

3.6 Skipping update of the sum

For each iteration in our algorithm we update the total sum, which is quite an expensive operation since it uses the GMP functions. If one of the \( k \) partly sums is 0, then the result of \( \prod_{k=1}^{n} \) will also be 0 and there is no need to update the sum. So we change our updating function to the following:

```c
mpz_set_si (tmp,1);
for (k=0; k < n; k++)
{
  if (prod[k] == 0LL)
  {
    return;
  }
  mpz_mul_si (tmp, tmp, prod[k]);
}
```

So if any of the \( \text{prod}[k] \) is 0 we just abort and continue with the next iteration. As we shall in section 4.2 this has some surprising effects.

3.7 Variants

As we established in section 2.5 extending the algorithm to the variants does not require many changes, and that is also the case with the implementation.

Hooksed

When computing the sum of a hooked sequence, we want to keep the fixed position in the binary code setted to 0. This is best done when using the Gray code, which allow us to let our \( \text{min} \) "jump over" the hooked position and do the rest of the algorithm without further changes. While this also should be possible when exploiting the reflections and with parallesation, we have not
implemented this because it will only make the code more complex and it will not show anything interesting. So the only extra line needed in our loop right before the Gray code is updated is if (min>hook) min<<=1; where hook denotes the position of the hook.

**Extended**

Using the Gray code makes it easy to keep track of the number of 1’s in the integer. We maintain a variable which we increment by one when the Gray code changed a bit from 0 to 1, and vice versa decrement when going from 1 to 0. This variable is then multiplied with the sum for each iteration. This is implemented also exploiting the reflections, and source code is in appendix A.4.

**Triplets**

Besides setting bigN to $2^{3^n}$, we simply need to add a third term to loop that updates the sum using the Gray code. Making it as follows:

```c
for (k = 0; k < n && swp < <(k + 1) < bigN; k++, swp <<= 1)
{
    prod[k] -= CHECK_BIT(gray, swp) * CHECK_BIT(gray, swp < <(k + 1));
}
```

This only works with our way of counting using 1 and 0 in the Gray code sequence. With everything else we have implemented, it also worked with Godfrey’s method simply letting the CHECK_BIT macro use 1 and −1 instead (or 2 and −2 in Gray code) as the only change needed in the program, but as we have expected it does not work for the triplets. Source code in appendix A.5.

**General sets**

With the Skolem and Langford sequences we could just hardcode the positions into our loop, but with general sets we need to handle the positions more dynamically. So we keep our set in array of $n$ elements, where $n$ now is the size of set, and use its values in our summation loop. That makes the Gray code loop look as follows:

```c
swp = min >> delta[0];
for (k = 0; k < n && swp < <(k + 1) < bigN; k++, swp = min >> delta[k])
{
    prod[k] -= CHECK_BIT(gray, swp);
}

swp = min << delta[0];
for (k = 0; k < n && swp < bigN; k++, swp = min << delta[k])
{
    prod[k] -= CHECK_BIT(gray, swp);
}
```

Note that $n$ is now the size of the set or the length of the array delta which contains our set. Instead of moving the position in swp either one to the left or right each time, we set the position directly from each number in the set with the starting point in min. This works both with our and Godfrey’s method. Source code in appendix A.6.
4 Results

This section documents the running of the algorithms, comparisons of their performance and new found number of solutions to some variations of the Langford and Skolem sequences. Runs have been made on a 2.53GHz machine running Linux 2.6.28 and the programs were compiled using GCC version 4.3.3 with -O3 optimization flag and dynamically linked to the GNU Multiple Precision Arithmetic Library (GMP) version 4.2.4, which was needed to deal with the very large integer values. Unless otherwise stated all runs are made with our way of counting in the algorithm. All runs have successfully confirmed the known values of Skolem and Langford sequences, including giving 0 for the n’s with no solutions, so we are pretty confident that our implementations work correctly.

4.1 The effect of the optimisations

Figure 7 shows on a logarithmic scale the time used for calculating \( S(n) \) for \( 9 \leq n \leq 20 \) without any optimisations, using the Gray code and exploiting the reflections. Without any optimisations the algorithm runs in \( O(2^{2n}n^2) \) time, whereas the Gray code reduces the complexity to \( O(2^{4n}) \) whereas exploiting the reflections is just a constant factor of about halving the running time. Using

![Figure 7: Actual running times of our algorithm with optimisations](image)

the Gray code has as expected the biggest impact, but the exponential factor is clearly still the dominating factor since all lines goes straightly exponential.

4.2 Godfrey’s versus ours

The only difference in the implementation of our and Godfrey’s algorithm is in the way we calculate the sum, either with 1 and 0 or 1 and -1, so the comparison is completely fair. The runs with Godfrey’s algorithm and ours using the reflections show that Godfrey’s runs considerably faster. This seems
confusing since we have just stated that the implementation and complexity is identical. It turns out that the little trick with skipping the expensive update of the total sum when one of the sums was 0, see section 3.6, actually does all the difference. Figure 8 shows (a) out of how many considered terms in the calculation we were able to skip the expensive update and (b) the actual running times for ours and Godfrey’s algorithm. These runs was done with the version that exploits the reflections.

<table>
<thead>
<tr>
<th>n</th>
<th>Our</th>
<th>%</th>
<th>Godfrey’s</th>
<th>%</th>
<th>Total terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>50%</td>
<td>0</td>
<td>0%</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>43.75%</td>
<td>4</td>
<td>25%</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>32.81%</td>
<td>14</td>
<td>21.75%</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>70</td>
<td>27.34%</td>
<td>76</td>
<td>29.69%</td>
<td>256</td>
</tr>
<tr>
<td>5</td>
<td>230</td>
<td>22.46%</td>
<td>258</td>
<td>25.20%</td>
<td>1024</td>
</tr>
<tr>
<td>6</td>
<td>767</td>
<td>18.72%</td>
<td>1268</td>
<td>30.96%</td>
<td>4096</td>
</tr>
<tr>
<td>7</td>
<td>2551</td>
<td>15.57%</td>
<td>4560</td>
<td>27.83%</td>
<td>16384</td>
</tr>
<tr>
<td>8</td>
<td>8516</td>
<td>12.99%</td>
<td>21680</td>
<td>33.08%</td>
<td>65536</td>
</tr>
<tr>
<td>9</td>
<td>28195</td>
<td>10.75%</td>
<td>81522</td>
<td>31.10%</td>
<td>262144</td>
</tr>
<tr>
<td>10</td>
<td>93093</td>
<td>8.87%</td>
<td>361004</td>
<td>34.42%</td>
<td>1048576</td>
</tr>
<tr>
<td>11</td>
<td>304528</td>
<td>7.26%</td>
<td>1382784</td>
<td>32.97%</td>
<td>4194304</td>
</tr>
<tr>
<td>12</td>
<td>995898</td>
<td>5.93%</td>
<td>6026880</td>
<td>35.92%</td>
<td>16777216</td>
</tr>
<tr>
<td>13</td>
<td>3222212</td>
<td>4.80%</td>
<td>23294234</td>
<td>34.71%</td>
<td>67108864</td>
</tr>
<tr>
<td>14</td>
<td>10408281</td>
<td>3.87%</td>
<td>99462384</td>
<td>37.05%</td>
<td>268435456</td>
</tr>
<tr>
<td>15</td>
<td>33450276</td>
<td>3.11%</td>
<td>387714604</td>
<td>36.10%</td>
<td>1073741824</td>
</tr>
<tr>
<td>16</td>
<td>106865225</td>
<td>2.48%</td>
<td>1635333408</td>
<td>38.07%</td>
<td>4294967296</td>
</tr>
</tbody>
</table>

As can be seen in the table, using Godfrey’s algorithm can abort the expensive update of the sum over 30% of the times in most cases, whereas our
method can do that less and less as \( n \) increases. This gives Godfrey’s method a better running time in practice of about 15%. If we do not skip the update of the sum, the running times are identical. This is one of those things that is really impossible to predict beforehand and was probably the most interesting surprise when running these tests.

4.3 New values

Previously only results for the hooked and extended sequences for \( n \leq 14 \) where known [18]. In table 2 we give new results for both the Skolem and Langford variant for \( n \)'s from 15 to 18. The computation time for the hooked sequences of order 18 was over 730 minutes using the simple Gray code version, and 440 minutes for the extended sequences of order 18 using the version that exploits the reflections.

\[
\begin{array}{cccc}
  n & S_n(n) & S_{2n} & L_n(n) \\
  15 & 1875064880 & 168870048 & 959931648 \\
  16 & 17851780784 & 0 & 7711661232 \\
  17 & 165367171136 & 0 & 71927185248 \\
  18 & 150656453568 & 113071735648 & 702865301632 \\
\end{array}
\]

Table 2: New values for the extended and hooked Skolem Langford sequences

Table 3: The number of distinct Rosa sequences in the Langford and Skolem variant
4.4 GMP performance penalty

Since we deal with very large integer values the standard data types are not sufficient, but the use of the GMP library to handle these values has a considerable performance penalty. Our runs showed that using the GMP data types the total calculation runs four times slower than using native C data types. The expensive part is the multiplication, which we do up to \( n \) times in each of the \( 2^{2n} \) iterations. An unsigned long long int has a max value of 18,446,744,073,709,551,615, which should seem to be sufficient for even for calculations of larger \( n \)'s, but while it would be sufficient for the result, the algorithm works in such a way that the sum during the calculation greatly exceeds that max value. So even for \( n > 17 \) we would run into integer overflows using the native data types. With Godfrey’s method it is even worse, since the sums are \( 2^{2n} \) times bigger.
5 Concluding remarks

We have successfully applied the inclusion-exclusion principle to count the number of Skolem sequences and used it to develop an algorithm for this. Using the inclusion-exclusion principle gave us a much clearer and understandable algorithm than Godfrey’s, even though they were very similar. It did not perform better than Godfrey’s algorithm in practice due to a simple constant time optimisation in the implementation where we skip an expensive operation if we encounter a 0 in the partly sums, but using our approach made it easier to extend both our and Godfrey’s algorithm to other variants of the problem and making them much more generalised to enumerative sequences over any kind of set, rather than just the normal ones of \{1, \ldots, n\}. We have after some computation time found some new values for the hooked and extended sequences.

5.1 Further work

Establishing the working principles of our algorithm for enumerating Skolem sequences using well known ideas from combinatorics, should allow for more work on applying the algorithm for some other variants of the sequences we have not considered. While we cannot hope for improvements in the computational complexity of the algorithm, more constant time optimization should be possible due to the great symmetry in the calculation.

One of the big bottlenecks is the use of the GMP library to handle the very large sums, so it could be interesting to avoid that even for larger values of \(n\), either with using another computer architecture or finding some other way to compute the sum. Knuth actually claims in his exercise that it is not necessary to compute the sum exactly [7], but we have not looked into this.

Since there is a connection between the number of Skolem sequences and the number of solutions to Hefter’s first difference problem and the Rosa sequences have a connection to Hefter’s second difference problem [1], maybe the exact number of Skolem and Rosa sequences will give new bounds to Hefter’s difference problems.

Our algorithm does not generate Skolem sequences, so this does not help in the areas where Skolem sequences are used to construct other kind of systems. To actually generate all the possible Skolem sequences of a given order, we cannot see another way than checking all \(2n!\) permutations.

The next results in this area in line to be found in the near future will probably be \(S(24)\) and \(S(25)\), and it is really just a matter of throwing a lot of processing power into it, preferably using multiple processors.

5.2 Reflections on the project

We have not made any drastic changes to the original project agreement (for reference see appendix A.1), mostly regarding terminology and precisions. First of all, we have used the name Skolem sequences throughout the thesis rather than Langford sequences. This choice was made after discovering that Skolem sequences was much more well studied than the Langford variant. Secondly we wrote that we will use an algorithm described by Husfeldt and Björklund in [2], but as can be seen in the description of our inclusion-exclusion algorithm in 2.3.
it was the general principle of using the inclusion-exclusion principle on exact counting problems that was an inspiration, not a specific algorithm.
References


A Appendix

A.1 Original project description

Problem formulation As a basis for this thesis we consider the Langford pairing problem and some of its variants.

We will produce some implementations of the algorithm described by Husfeldt-Björkland in the paper Exact algorithms for exact satisfiability and number of perfect matchings and Godfrey’s algorithm for the Langford problem, in order to confirm previously found Langford-values (exact number of solutions to a Langford pairing problem of a given size) and hopefully find new ones. We will apply parallization techniques to the algorithms and analyze the improvement.

We will analyze each algorithm and measure their performance in practice.

Method Implement at least two different versions of the algorithm, a clean implementation of the described algorithm, and another one to be heavily low-leveled optimised C code, and let it run on a really fast machine.

What will be handed in A written report with descriptions of the problem, the algorithms and documentation of their output. Plus source code
A.2 basic.c

```c
#include <stdio.h>
#include <gmp.h>
#define GODFREY 0
#if GODFREY
#define CHECK_BIT(var, pos) (((var) & (pos)) ? 1LL : -1LL)
#else
#define CHECK_BIT(var, pos) (((var) & (pos)) ? 1LL : 0LL)
#endif
#define n 9

int count_bits(unsigned x) {
    x = (x & 0x55555555) + ((x >> 1) & 0x55555555);
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
    x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F);
    x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF);
    x = (x & 0x0000FFFF) + ((x >>16) & 0x0000FFFF);
    return x &1;
}

int main() {
    mpz_t bigsum;
    mpz_t tmp;
    unsigned long long int x, sum , swp;
    unsigned int j, k, odd;
    long long int prod[n];
    mpz_init(bigsum);
    mpz_init(tmp);
    for (j =0; j < n; j ++) prod[j] = 0LL;
    for (x =0; x < (1LL<<(2*n)); x ++)
    {
        for (k=1; k <= n; k++)
        {
            sum = 0LL;
            swp = 1LL;
            for(j=1; j <= (2*n-k); j++, swp<<=1)
            {
                sum += CHECK_BIT(x, swp)*CHECK_BIT(x,swp<<k);
            }
            prod[k-1] = sum;
        }
        mpz_set_si(tmp, prod[0]);
        for (j=1; j<n; j++)
            mpz_mul_si(tmp, tmp, prod[j]);
        if(count_bits(x)) mpz_sub(bigsum, bigsum, tmp);
        else mpz_add(bigsum, bigsum, tmp);
    }
    #if GODFREY
    mpz_div_ui(bigsum, bigsum, (1LL<<(2*n)));
```
53  #endif
54  gmp_printf("Skolem sequences of order %d: %Zd\n", n, bigsum);
55  }
A.3 graycode.c

```c
#include <stdio.h>
#include <gmp.h>
#define GODFREY 0
#if GODFREY
#define CHECK_BIT(var,pos) (((var) & (pos)) ? 2LL : -2LL)
#define CHECK_BIT_G(var,pos) (((var) & (pos)) ? 1LL : -1LL)
#else
#define CHECK_BIT(var,pos) (((var) & (pos)) ? 1LL : 0LL)
#endif
#define n 9
#define NPOS (2*n)

mpz_t bigsum;
mpz_t tmp;
long long prod[n];
unsigned int k, j;
unsigned long long sum, swp;

inline void updatesum(unsigned int odd)
{
  mpz_set_si(tmp,1);
  for(k=0; k < n; k++)
  {
    if(prod[k] == 0LL)
      return;
    mpz_mul_si(tmp,tmp,prod[k]);
  }
  if(odd) mpz_sub(bigsum,bigsum,tmp);
  else mpz_add(bigsum,bigsum,tmp);
}

inline void calc(unsigned long long gray)
{
  for(k=1; k <= n; k++)
  {
    sum = 0;
    swp = 1LL;
    for(j=1; j <= (NPOS-k); j++, swp<<=1) // n^2 calc of sum
    {
      if(GODFREY)
        sum += CHECK_BIT_G(gray,swp)*CHECK_BIT_G(gray,swp<<k);
      else
        if(gray & swp) sum += CHECK_BIT(gray,swp<<k);
    }
    prod[k-1] = sum;
  }
}

int main(int argc, char *argv[])
```

28
unsigned long long int x, min;
unsigned long long int gray = 0;
unsigned long long int bigN = (1LL<<(NPOS));
unsigned int multiproc = 0; // are we running as one of several processes?
unsigned int num_proc = 0; // log_2 of number of processes
unsigned int proc_id = 0; // id of this proces. Range 0 .. 2^num_proc - 1
mpz_init(bigsum);
mpz_init(tmp);
long long int hook = -1LL;
int arg = 1;
while(arg<argc && *argv[arg]=='-')
{
    switch(*(argv[arg]+1))
    {
        case 'p': // multiproc - does _not_ work with -h and -r
            ++arg;
multiproc = 1;
            if (sscanf(argv[arg], "%d/%d", &proc_id, &num_proc) != 2) return -1;
            break;
        case 'h': // hooked
            ++arg;
            hook = (1 LL << (((2* n)-1));
            break;
        case 'r': // Rosa
            ++arg;
            hook = (1 LL << (((n+1)-1));
            break;
        default:
            return -1;
    }
    ++arg;
}
unsigned long long int length = bigN >> num_proc;
unsigned long long int start = proc_id*length;
unsigned long long int stop = start + length;
gray = start ^ (start >> 1 LL);
for (k = 0; k < n; k++) prod[k] = 0LL;
calc(gray);
if (proc_id!=0)
    start++;
for (x = start; x < stop; ++x)
{
    min = x & (-x); // bit changed in graycode
    if (min>>hook) min<<=1; // fool the grey code to step over the hook
    gray ^= min; // graycode
    if (min&gray) // +1
swp = min >>= 1;
for (k = 0; k < n && swp; k++, swp >>= 1)
{
    prod[k] += CHECK_BIT(gray, swp);
}

swp = min <<= 1;
for (k = 0; k < n && swp < bigN; k++, swp <<= 1)
{
    prod[k] += CHECK_BIT(gray, swp);
}
else // 0
{
    swp = min >>= 1;
    for (k = 0; k < n && swp; k++, swp >>= 1)
    {
        prod[k] -= CHECK_BIT(gray, swp);
    }
    swp = min <<= 1;
    for (k = 0; k < n && swp < bigN; k++, swp <<= 1)
    {
        prod[k] -= CHECK_BIT(gray, swp);
    }
    updatesum(x & 1);
}
#if GODFREY
    gmp_printf("bigSum %Zd\n", bigsum);
    mpz_fdiv_q_2exp(bigsum, bigsum, NPOS);
#endif
if (hook > 1) printf("hooked ");
gmp_printf("skolem sequences of order %d: %Zd\n", n, bigsum);
A.4 reflections.c

```
#include <stdio.h>
#include <gmp.h>
#define GODFREY 0
#define EXT 1
#if GODFREY
#define CHECK_BIT(var, pos) (((var) & (pos)) ? 2LL : -2LL)
#define CHECK_BIT_G(var, pos) (((var) & (pos)) ? 1LL : -1LL)
#else
#define CHECK_BIT(var, pos) (((var) & (pos)) ? 1LL : 0LL)
#endif
#define n 9
#if EXT
#define NPOS (2*n+1)
#else
#define NPOS (2*n)
#endif

mpz_t bigsum;
mpz_t tmp;
unsigned int multiproc = 0; // are we running as one of several processes?
unsigned int num_proc = 0; // log_2 of number of processes
unsigned int proc_id = 0; // id of this process. Range 0 .. 2^num_proc - 1
unsigned int j, k, p, odd, ones;
unsigned long long int swp, min, inner, outer, sum, len, start, abortmul;
unsigned long long int bigN = 1LL << NPOS;
long long int prod[n];

inline void calc(unsigned long long int gray)
{
    for (k=1; k <= n; k++)
    {
        sum = 0;
        swp = 1LL;
        for(j=1; j <= (NPOS-k); j++, swp<<=1) // n^2 calc of sum
            #if GODFREY
            sum += CHECK_BIT_G(gray, swp)*CHECK_BIT_G(gray, swp<<k);
            #else
            if(gray & swp) sum += CHECK_BIT(gray, swp<<k);
            #endif
        prod[k-1] = sum;
    }
    #if EXT
    ones = 0;
    for (k=0; k<NPOS; k++) if (gray & (1LL<<k)) ones++;
    #endif
```
inline void updatesum(unsigned int odd, int times)
{
#if EXT
    mpz_set_si(tmp, -ones);
#else
    mpz_set_si(tmp, 1);
#endif
    for (k = 0; k < n; k++)
    {
        if (prod[k] == 0LL)
            return;
        mpz_mul_si(tmp, tmp, prod[k]);
    }
    for (k = 0; k < times; k++)
    {
        if (odd) mpz_sub(bigsum, bigsum, tmp);
        else mpz_add(bigsum, bigsum, tmp);
    }
}

int main(int argc, char *argv[])
{
    mpz_init(bigsum);
    mpz_init(tmp);
    odd = 0;
    int arg = 1;
    while (arg < argc && *argv[arg] == '-')
    {
        switch (*(*(argv + arg) + 1))
        {
            case 'p':
                ++arg;
                multiproc = 1;
                if (sscanf(argv[arg], "%d/%d", &proc_id, &num_proc) != 2) return -1;
                break;
            default:
                return -1;
        }
        ++arg;
    }
    ++arg;
    for (p = 0; p < n+1; p++) // phases
    {
        printf("\nPhase %d\n", p);
        for (outer = 0; outer < 1ULL<<p; outer++)
        {
            gray = 0ULL;
            if (p != n)
                gray |= 1ULL<<p; // set bit at pos p fixed
        }
for (j = 0; j < p; j++)
    if (outer & 1LL << j)
        gray |= 1LL << j | 1LL << (NPOS - j - 1);

if (proc_id == 0)
    {
        calc(gray);
        odd = !odd;
        if (p < n)
            {
                if (p == n - 1) odd = 1;
                updatesum(odd, 2);
            }
            else
                updatesum(0, 1);
    }
else
    {
        if (p < n)
            {
                // find length to calculate for this process
                if (n - p > num_proc) len = 1LL << (2*(n - p - 1) - num_proc + EXT);
                else
                    {
                        if (proc_id > 0) continue;
                        len = 1LL << (2*(n - p - 1) + EXT);
                    }
        start = proc_id * len; // starting position used in multiproc
        if (proc_id != 0)
            {
                start --;
                len ++;
            }
        gray ^= (start ^ (start >> 1LL)) << (p + 1); // initialise gray code
        calc(gray);
        for (inner = start + 1; inner < len + start; inner++)
            {
                min = inner & (-inner); // bit changed in gray code
                min <<= p + 1; // shift according to phase
                gray ^= min;
                if (min & gray) // 1
                    {
                        ones ++;
                        swp = min >> 1;
                        for (k = 0; k < n && swp; k++, swp >>= 1) // updating bits to the left of min
                            {
                                prod[k] += CHECK_BIT(gray, swp);
                            }
                        }
swp = min << 1;
for (k = 0; k < n && swp < bigN; k++, swp <<= 1) //
    updating bits to the right of min
{
    prod[k] += CHECK_BIT(gray, swp);
}
else // 0
{
    ones--;
    swp = min >> 1;
    for (k = 0; k < n && swp; k++, swp >>= 1)
    {
        prod[k] -= CHECK_BIT(gray, swp);
    }
    swp = min << 1;
    for (k = 0; k < n && swp < bigN; k++, swp <<= 1)
    {
        prod[k] -= CHECK_BIT(gray, swp);
    }
    odd = ! odd;
    updatesum(odd, 2);
}
#ifdef EXT
    if (p == n && proc_id == 0)
    {
        gray |= 1LL << n;
        calc(gray);
        updatesum(1, 1);
    }
#endif
#ifdef GODFREY
    gmp_printf("bigSum %Zd\n", bigsum);
    mpz_fdiv_q_2exp(bigsum, bigsum, NPOS);
#endif
if (multiproc) printf("process (%d/%d) - ", proc_id, num_proc);
#ifdef EXT
    printf("enxtended ");
#endif
    gmp_printf("skolem sequences of order %d: %Zd\n", n, bigsum);
return 0;
A.5 triplet.c

1 #include <stdio.h>
2 #include <gmp.h>
3 #define CHECK_BIT(var, pos) (((var) & (pos)) ? 1LL : 0LL)
4 #define n 9
5 #if n%2==0
6 # define NODD 0
7 #else
8 # define NODD 1
9 #endif
10
11 mpz_t bigsum;
12 mpz_t tmp;
13 unsigned int k;
14 long long int prod[n];
15
16 inline void updatesum(unsigned int odd)
17 {
18   mpz_set_si(tmp,1);
19   for(k=0; k < n; k++)
20   {
21     if(prod[k] == 0LL)
22     {
23       return;
24     }
25     mpz_mul_si(tmp,tmp,prod[k]);
26   }
27   if(NODD)
28     mpz_mul_si(tmp,tmp,-1);
29   if(odd) mpz_sub(bigsum,bigsum,tmp);
30   else mpz_add(bigsum,bigsum,tmp);
31 }
32
33 int main()
34 {
35   unsigned long long int i,swp,min;
36   unsigned long long int gray = 0;
37   unsigned long long int bigN = (1LL<<(3*n));
38   mpz_init(bigsum);
39   mpz_init(tmp);
40   for (k=0; k < n; k++) prod[k] = 0LL;
41   for(i=0LL; i < bigN; i++)
42   {
43     min = i & (-i); // bit changed in graycode
44     gray ^= min; // graycode
45     if(min&gray) // +1
46     {
47       swp=min>>1;
48       for (k=0; k < n && swp>>(k+1); swp>>=1)
prod[k] += CHECK_BIT(gray, swp) * CHECK_BIT(gray, swp>>(k+1));
}

swp = min << 1;
for (k = 0; k < n && swp << (k+1) < bigN; k++, swp <<= 1)
{
    prod[k] += CHECK_BIT(gray, swp) * CHECK_BIT(gray, swp<<(k+1));
}
}
else // 0
{
    swp = min >> 1;
    for (k = 0; k < n && swp>>(k+1); swp >>= 1)
    {
        prod[k] -= CHECK_BIT(gray, swp) * CHECK_BIT(gray, swp>>(k+1));
    }
    swp = min << 1;
    for (k = 0; k < n && swp<<(k+1) < bigN; k++, swp <<= 1)
    {
        prod[k] -= CHECK_BIT(gray, swp) * CHECK_BIT(gray, swp<<(k+1));
    }
    updatesum(i&1);
}
gmp_printf("Triplet Skolem sequences of order %d: %Zd\n", n, bigsum);
A.6 generalsets.c

```c
#include <stdio.h>
#include <gmp.h>
#define GODFREY 0
#if GODFREY
#define CHECK_BIT(var,pos) (((var) & (pos)) ? 2LL : -2LL)
#else
#define CHECK_BIT(var,pos) (((var) & (pos)) ? 1LL : 0LL)
#endif
#define n 4
#define NPOS (2*n)

mpz_t bigsum;
mpz_t tmp;
long long int prod[n];
unsigned int k;

inline void updatesum(unsigned int odd)
{
  mpz_set_si(tmp,1);
  for(k=0; k < n; k++)
  {
    if(prod[k] == 0LL)
      return;
    mpz_mul_si(tmp,tmp,prod[k]);
  }
  if(odd) mpz_sub(bigsum,bigsum,tmp);
  else mpz_add(bigsum,bigsum,tmp);
}

int main(int argc, char *argv[])
{
  unsigned int delta[n] = {2,3,5,6};
  unsigned long long int i,swp,min;
  unsigned long long int gray = 0;
  unsigned long long int bigN = (1LL<<(NPOS));
  mpz_init(bigsum);
  mpz_init(tmp);
  for (k=0; k < n; k++) prod[k] = 0LL;
  for(i=0LL; i < bigN; i++)
  {
    min = i & (-i); // bit changed in graycode
    gray ^= min; // graycode
    if(min&gray) // +1
      {;
        swp=min>>delta[0];
        for (k=0; k < n && swp; k++, swp=min>>delta[k])
          {
            prod[k] += CHECK_BIT(gray,swp);
          }
      }
  }
}
```
swp = min << delta[0];
for (k = 0; k < n && swp < bigN; k++, swp = min << delta[k])
{
    prod[k] += CHECK_BIT(gray, swp);
}
}
else // 0
{
    swp = min >> delta[0];
    for (k = 0; k < n && swp && swp = min >> delta[k])
    {
        prod[k] -= CHECK_BIT(gray, swp);
    }
    swp = min << delta[0];
    for (k = 0; k < n && swp < bigN; k++, swp = min << delta[k])
    {
        prod[k] -= CHECK_BIT(gray, swp);
    }
    updatesum(i & 1);
}
#if GODFREY
    gmp_printf(" bigSum %Zd\n", bigsum);
    mpz_fdiv_q_2exp(bigsum, bigsum, NPOS);
#endif
gmp_printf("%Zd skolem sequenses over {", bigsum);
for (k = 0; k < n; k++)
    printf("%d", delta[k]);
printf("} \n");