Scan 7877

Liu

7877

Master's Essay

Efficient Recognition of Integer Sequences

Peter Liu
pwlliu@neumann.uwaterloo.ca

December 1, 1994

Abstract

Researchers have long sought efficient methods for recognizing integer sequences. There are many ways to accomplish this recognition. One approach is to search for relations between the given integer sequence and a given database of well-known integer sequences. This paper presents a strategy to achieve the exploration by performing pattern queries on a database of well-known sequences. Efficiency is obtained through preprocessing the sequence database, use of hashing techniques, and use of carefully designed algorithms. Query evaluation algorithms are presented and performance results are given.

Key Words: *Searching, Preprocessing, Hashing, Optimization.*

Contents

1 Introduction

Integer sequences have been studied for many years, motivated by needs that arise in fields such as discrete mathematics, combinatorics, image processing, speech recognition and others. It was a milestone when Neil Sloane published his book *A Handbook of Integer Sequences* [12] in 1973. In his book, he listed 2372 sequences of nonnegative integers, arranged lexicographically, with an index, references, and notes for users. Since then, it is possible for an user to find out information about an *unknown* sequence easily, simply by looking it up the handbook. However, many sequences do not appear as any one of the listed sequences; rather, they are simply related with those listed sequences. For example, an integer sequence can be a linear combination of two sequences in the handbook. To check this, it is necessary to build an electronic edition of the handbook. This would permit queries that are infeasible or impossible using a printed book. Furthermore, such an electronic handbook would make it easier to make corrections and additions to cooperate with the progress in the field. In order to achieve the goal, we must provide efficient solutions to do the queries. What kind of queries should we allow to be performed? Having designed the query set, what kind of strategies should we use to evaluate those queries efficiently? This paper is an attempt to solve some of these particular problems. We have implemented some of these query algorithms on Sun-Sparc machine by using the C++ language. The program code is listed in Appendix A. Performance tests have been conducted and experimental data is reported in section 3.

In section 2, we will discuss some similar work that has been done previously. In section 3, we will outline some queries which we have implemented and show their algorithms and their performance results. In section 4, we will study a similar system, which is called *superseeker* created by Neil Sloane, AT&T Bell Lab and others. We will analyze the similarities and differences between our work and *superseeker*. Finally, in section 5, we will provide the possible directions of future work.

2 Previous Work

2.1 Dana Angluin's Work

One approach to recognize an integer sequence is by checking whether or not the sequence can be generated from certain rules, for example some standard recurrence relations. Dana Angluin, a researcher in the Electronics Research Lab of UC Berkeley, proposed some techniques [2] for constructing easily inferred sequences in 1974. Her method was to enumerate those techniques to show the inferrability of some sequences. Basically, her attempt was to describe what sequences of numbers have an obvious, or easily inferred, pattern. The test of such a sequence is that from a few successive terms of it a person very easily guesses a correct rule of generation for it. Characterizing the sequences was chosen as a step towards understanding the process of learning-from-examples; That was an informal first approximation to that characterization.
Following is some typical types of sequences of her enumeration:

1. Simple arithmetic sequences.

   - constant.
     3, 3, 3, 3, 3, ...
     56, 56, 56, 56, 56, ...
   - $x_{i+1} = f(x_i)$, where $f(x) = x + c, cx, x^c, cx + d$
     1, 2, 3, 4, 5, 6, 7, 8, 9, ...
     7, 10, 13, 16, 19, 22, 25, ...
     2, 9, 23, 51, 107, 219, ...
   - squares and other quadratic polynomials
     1, 4, 9, 16, 25, 36, 49, 64, ...
     2, 5, 10, 17, 26, 37, 50, 65, ...
   - cubes
     1, 8, 27, 64, 125, 216, 343, ...

2. Interleaving.

   - two 'independent' sequences interleaved x_i, y_i
     1, 1, 1, 2, 1, 3, 1, 4, 1, 5, 1, 6, 1, 7, 1, 8, ...
     0, 19, 1, 20, 2, 21, 3, 22, 4, 23, 5, 24, 6, 25, ...

11, 12, 111, 123, 1111, 1234, 11111, 12345, ...
1, 2, 3, 11, 22, 33, 111, 222, 333, 1111, 2222, ...

- two 'dependent' sequences interleaved $x_i, f(x_i)$
  4, 3, 9, 8, 16, 15, 25, 24, 36, 35, 49, 48, ...
  2, 4, 3, 8, 4, 16, 5, 32, 6, 64, 7, 128, ...
  32, 3, 2, 64, 6, 4, 128, 1, 2, 8, 256, 2, 5, 6, 512, ...
  6, 69, 12, 1236, 24, 2481, 48, 48144, 96, ...
  11, 9, 27, 25, 51, 49, 83, 81, 123, 121, ...
- interleaved three sequences, with binary operator, $x_i, y_i, f(x_i, y_i)$ or $x_i, x_{i+1}, f(x_i, x_{i+1})$
  2, 4, 6, 3, 8, 11, 4, 16, 20, 5, 32, 37, 6, ...
  2, 3, 6, 4, 5, 20, 6, 7, 42, 8, 9, 72, 10, ...
  6, 12, 612, 24, 48, 2448, 96, 192, 96192, ...

3. Sequences of groups.

- constructing sequences of groups
  1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ...
  1, 2, 1, 3, 2, 1, 4, 3, 2, 1, ...
  2, 44, 888, 16161616, 3232323232, ...
- interleaving with groups
  10, 11, 22, 100, 111, 222, 333, 1000, 1111, 2222, ...
  2, 4,2, 8, 16, 32, 16, 8, 64, 128, 256, 512, 256, ...
  41234, 512345, 6123456, 71234567, ...

4. Concatenation.

- constant groups concatenated
  1, 122, 122333, 1223334444, ...
  90009, 1600061, 250000052, 3600000063, ...
- concatenation of pairs
  12, 34, 56, 78, 910, 1112, 1314, ...
  21, 411, 8111, 161111, 3211111, ...
  2628, 6365, 124126, 215217, 342344, ...

- concatenation, other patterns of matching
  838, 16416, 32532, 64664, 1287128, ...
  38, 815, 1524, 2435, 3548, ...
  235, 5711, 111317, 17923, 232931, ...

5. Digits.

   - digits intertwined with constants
     2, 4, 8, 106, 302, 604, 10208, 20506, ...
     2126, 2225, 2326, 2429, 2624, 2821, 212020, ...
   - digits repeated as a function of position or value
     1166, 3366, 6644, 110000, 114444, 119966, ...
     8, 166, 322, 644, 122888, 255666, 511222, ...

2.2 A. K. Dewdney's Work

Similar to Angluin's work, in 1986, A. K. Dewdney provided strategies to infer additional terms for a given finite sequence [8]. He used a *"multilevel inferring"* method to obtain possible additional terms to produce different candidates for the given sequence. Recognition can be achieved by checking all the candidates.
To demonstrate Dewdney's strategies, we are presenting two sequence examples:

- sequence 1: 2, 4, 8, 14, ...
- sequence 2: 1, 2, 6, 24, ...

One common question one may ask is that what are the subsequent numbers which do not appear here ? In particular, what is the next number in each of the two sequences ?

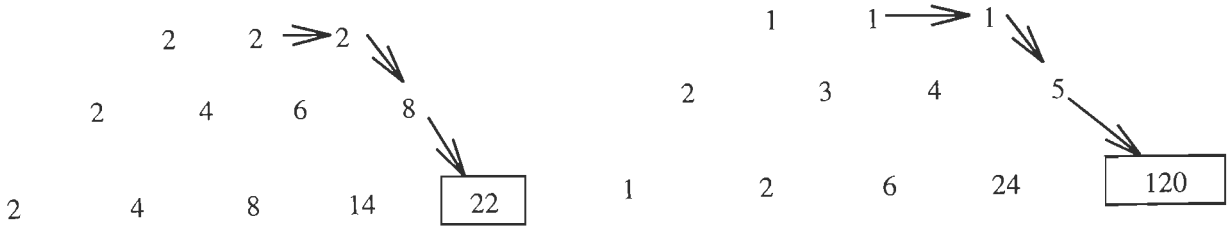
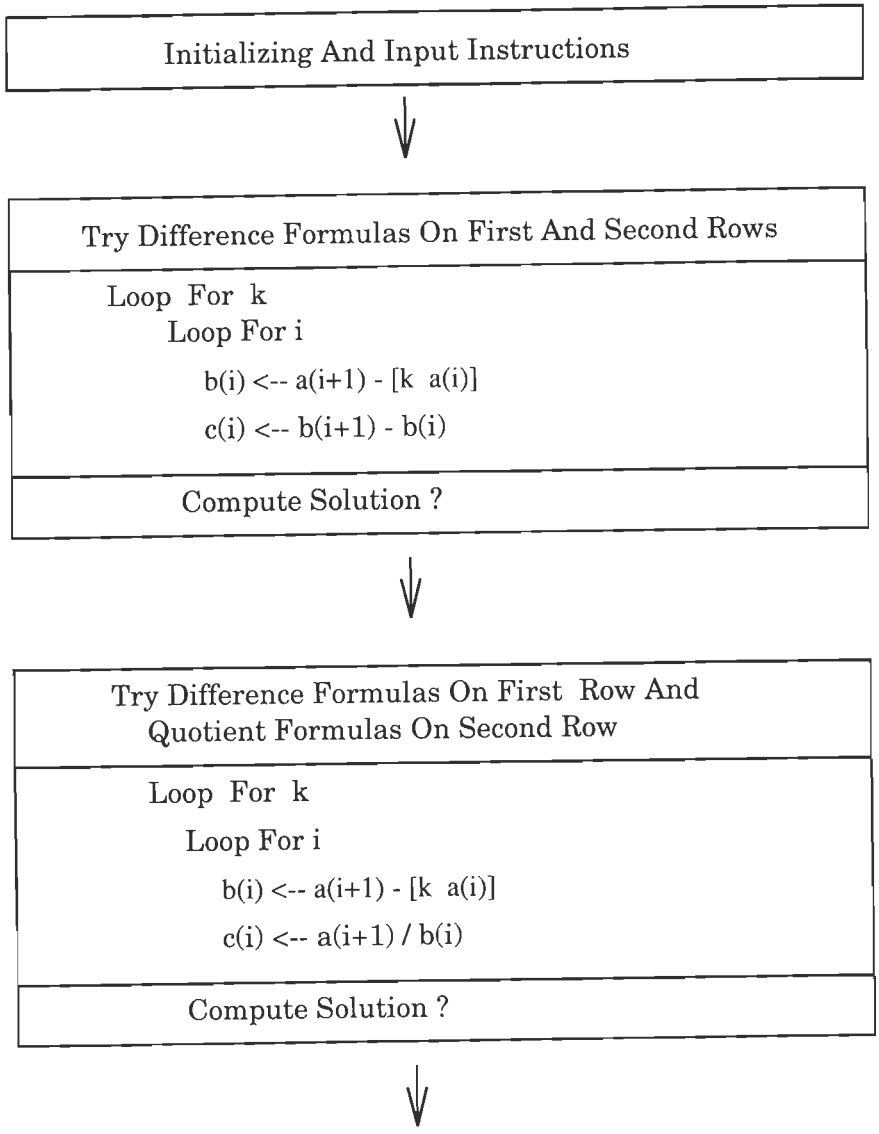


figure 2.1

By using Dewdney's strategies: a so-called SEQ program, one candidate of the answers for sequence 1 is 22; and sequence 2 is 120. Following is the flow chart diagram of the SEQ program:



(continue next page)

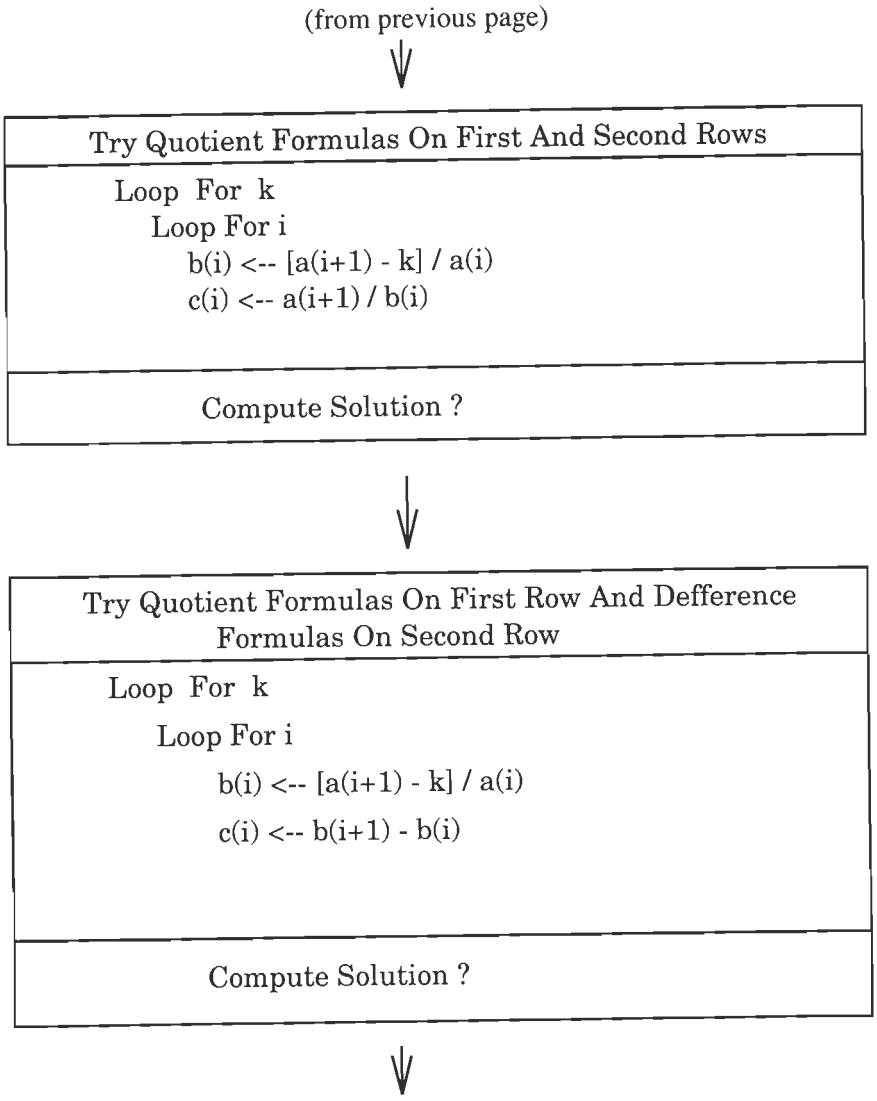


figure 2.2

There are two kinds of rules considered by SEQ: *additive* and *multiplicative*. One way to find the rules is to construct a so-called difference pyramid (see figure 2.1). At the bottom of the pyramid is the given sequence, and the pyramid is built up from bottom to top by finding the differences between successive numbers in the preceding level or row of numbers. Thus in se-

quence 1, the first number in the second row of the pyramid is obtained from the first two numbers in the first row, namely 2 and 4. Their difference is 2, and so 2 is the first number in the second row. Similarly, the other numbers in the second row are $8 - 4$, or 4, and $14 - 8$, or 6; the second row is the sequence 2, 4, 6. Continuing the same process to a third row of the pyramid gives a sequence with only two numbers, and they are both 2's . The equality of all the numbers in some row signals to stop building the pyramid upward and to start building it sideways. It is reasonable to assume the next number in the second row is obtained from the preceding number 6, by adding the number 2: the sum is 8. So 22 is a possible candidate for the next number. Actually, the sequence 2, 4, 8, 14, 22, ... is generated by successive values of the quadratic, or second-degree, polynomial $x^2 - x + 2$. The second way is to construct a set of quotients from the sequence instead of a set of differences (see figure 2.1). By taking quotients of successive pairs in the sequence 2: 1, 2: 6, 24, ... we obtain the second row in a pyramid, the sequence 2, 3, 4, ... The second sequence hints that the third row in the pyramid must be obtained by taking differences, not quotients. In due course, the solution itself is thus 120.
The main strategy in SEQ program is to build pyramids by considering both the consecutive differences and the consecutive quotients of successive pairs of numbers in a given row. Even more, it examines successive pairs of numbers in a sequence for more general additive and multiplicative rules. In the additive rule the first member of each pair may be multiplied by a constant k before the usual addition is done, and in the multiplicative rule the constant k may be added just after the usual multiplication (see figure 2.2).

2.3 J. Calmet and I. Cohen's Work

Almost at the same time as Dewdney's work, J. Calmet and I. Cohen [5] reported strategies to recognize sequences with recurrence relations (RR). Their strategies were part of the efforts to build a new system for manipulating integer sequences effectively.
One of the contributions of their work was providing strategies to recognize recurrence relations in sequences of numbers.

1. They first presented the standard approach method:*differences.*
   Let a_n be a sequence, where $n = 1, 2, ...$

The first differences are:

$$\triangle a_n = a_{n+1} - a_n,$$

The second ones are:

$$\triangle^2 a_n = \triangle a_{n+1} - \triangle a_n,$$

and in general:

$$\triangle^m a_n = \triangle^{m-1} a_{n+1} - \triangle^{m-1} a_n,$$

This leads to the RR:

$$\triangle^m a_n = \sum_{i=0}^{m} (-i)^i \binom{i}{m} a_{m+n-1}$$

The disadvantage of this method is that its efficiency depends on the mth differences values.

2. Then they mentioned some classical methods such as factorization of a_i, comparison of a known series and study of the ratio a_{n+1}/a_n.

3. After that, they provided a general treatment for the following RR:

$$a_n = C(n)a_{n-1} + B(n)a_{n-2} \tag{1}$$

where

$$C(n) = c_2 n^2 + c_1 n + c_0, \tag{2}$$

$$B(n) = b_2 n^2 + b_1 n + b_0, \tag{3}$$

The treatment was that solving the system of linear equations given by (1) first. This gives the possible values for $B(n)$ and $C(n)$. Then the values for the c's and b's are obtained by solving the systems of equations (2,3) when $C(n)$ and $B(n)$ are not constant.

4. Treatments for following RRs:

$$a_n = b a_{n-1}^3 + c a_{n-1}^2 + d a_{n-1} + e \tag{4}$$

where

$$a_n = b' + \sum_{j=1}^{l} c_i a_{n-j}$$

$$a_n = ba_{n-1}a_{n-2} + ca_{n-1} + da_{n-2} + e$$

b, b', c, d, e are constant and l is either given in the input or set to 5 by default.

$$a_n = na_{n-1} + (-1)^n \tag{5}$$

$$a_n = na_n + b \tag{6}$$

are straightforward. They simply proceeded by checking the given sequences of numbers. Note that for some RRs, a minimum number of terms are required.

Generally speaking their methods were a purely algebraic approach to the RR problems.

2.4 Eilam-Tzoreff and Vishkin's Work

One alternative approach is that, to search a set of given integer sequences, to explore certain relations between the given sequence and the set. In particular, a so called *minimum distance* relation has been studied extensively by many researchers. This is motivated by the needs arising from image recognition, speech recognition and molecular biology. Much progress has been made and many solutions have been produced. Results were summarized by the book *Time Warps, String Edits, and Macromolecules : the Theory and Practice of Sequence Comparison* [11]. Recently, Eilam-Tzoreff and Vishkin fertiled this field by providing a strategy [9] to explore relations between a given integer sequence and the linear combination of a given sequences set. Their idea was to preprocess the set of given sequences so that it will gain efficiency in later exploration.
Let P denote the pattern and T denote the text, and let $m = |P|, n = |T|$, and we assume each element in the P or T is a real number and $m \ll n$.
In their paper [9], Eilam-Tzoreff and Vishkin first introduced following five problems:

1. The *adding transformation problem*: For each $j, 1 \leq j \leq n - m + 1$, find whether there exists a constant c_0 such that $\forall, 1 \leq i \leq m, T_{j-1+i} = P_i + c_0$. A match is called an *adding transformation* occurrence.

2. The *multiplying transformation problem*: For each $j, 1 \leq j \leq n-m+1$, find whether there exists a constant c_1 such that $\forall, 1 \leq i \leq m, T_{j-1+i} = c_1 P_i$. A match is called an *multiplying transformation* occurrence.

3. The *linear transformation problem*: For each $j, 1 \leq j \leq n-m+1$, find whether there exists a constant c_0, c_1 such that $\forall, 1 \leq i \leq m, T_{j-1+i} = c_1 P_i + c_0$. A match is called an *linear transformation* occurrence.

4. The *k-degree polynomial transformation problem*: For each $j, 1 \leq j \leq n-m+1$, find whether there exist $k+1$ constants $c_0, c_1, ..., c_k$ such that $\forall, 1 \leq i \leq m, T_{j-1+i} = \sum_{l=0}^{k} c_l P_i^l$ (where P_i^l is the ith element of the pattern P raised to the lth power). A match is called a *k-degree polynomial transformation* occurrence.

5. The *k-linear transformation problem*: We are given k patterns $B_1, ...B_k$. For each $j, 1 \leq j \leq n-m+1$, find whether there exist k constants $c_1, c_2, ..., c_k$ such that $\forall, 1 \leq i \leq m, T_{j-1+i} = \sum_{l=1}^{k} c_l B_{lj}$ (where B_{lj} denotes the ith element of the pattern B_l). A match is called a *k-linear transformation* occurrence of the k patterns $B_1, B_2, ..., B_k$ in the tex.

Then the authors outlined the relations among these five problems, they are:

1. Reduction of the adding transformation problem into the exact string matching problem.
   The algorithm for the adding transformation problem is that for the given text T we look at $\overline{T}$, *the sequence of differences* of T, which is defined as: $\overline{T}_j = \overline{T}_{j+1} - \overline{T}_j$, $1 \leq j \leq n-1$. Similarly, define $\overline{P}_j = \overline{P}_{j+1} - \overline{P}_j$, $1 \leq j \leq m-1$. Thus, we can apply a known exact string matching algorithm for $\overline{T}$ and $\overline{P}$. The constant c_0 for each position j is determined by solving $T_j = P_1 + c_0$.

2. Reduction of the multiplying transformation problem into the exact string matching problem.
   Let $\overline{T}$ and $\overline{P}$ be the *sequences of quotients* of T and P, respectively, which are defined as follows: $\overline{T}_j = \frac{\overline{T}_{j+1}}{\overline{T}_j}$ for $1 \leq j \leq n-1$, and $\overline{P}_i = \frac{\overline{P}_{i+1}}{\overline{P}_i}$ for $1 \leq i \leq m-1$. Again, we can apply a known exact string matching algorithm for $\overline{T}$ and $\overline{P}$. The constant c_1 for each position j is obtained by solving $T_j = c_1 P_1$.

3. Reduction of the linear transformation problem into the multiplying transformation problem.
   Let $\overline{T}$ and $\overline{P}$ be the *sequences of differences* of the text and the pattern, respectively. We apply the multiplying transformation algorithm for $\overline{T}$ and $\overline{P}$. The constant c_0 for each position j is determined by solving the equation $T_j = c_1 P_1 + c_0$.

4. The k-degree polynomial transformation problem is included in the $(k+1)$-linear transformation problem.
   To see this simply choose the pattern $P_1^l P_2^l ... P_m^l$ (denote P^l) as B_l,$1 \leq j \leq k$ and the pattern consisting of m ones (denoted P^0) as B_{k+1}.

After that, the authors provided the algorithm for the k-linear transformation Problem. The algorithm has two stages: 1. Analysis of the pattern. 2. Analysis of the text. The main contribution of the algorithm is to use a table called a *witness* in the pattern analysis stage, which is defined as follows:
Figure 2.3 describes the k input patterns $B_1, ..., B_k$, all starting at the same vertical location and again these same k patterns starting at location j of the former k patterns.

For each $j, 2 \leq j < m$, $witness(j)$ is a $2k$-tuple $(l_1, l_2, ...l_{2k})$ And each location l_s suggests the equation with $2k$ unknowns $c_1, c_2, ..., c_{2k}$,

$$\sum_{i=1}^{k} c_i B_{i,l_s} = \sum_{i=1}^{k} c_{k+i} B_{i,j+l_s-1}.$$

The determination of l_s locations is that, l_1 is 1. Once the $i-1$ independent equations of locations $l_1, l_2, ..., l_{i-1}$ are given, l_i is the least location which gives the i'th independent equation. If an i'th independent equation does not exist, $l_i, ..., l_{2k}$ are all zero.

1. Text Analysis.
   The test analysis consists of three steps.S_j will represent a set whose elements are k-tuples of real numbers. The first step initializes for each position j in the tex a set S_j such that if a k-linear transformation occurrence starts at j then the k multiplyers must be a k-tuple $c_{j,1}, ..., c_{j,k}$ in S_j. The the second step, the candidacy of some positions in the tex is invalidated. The third step applies a kind of character by character

check to finally find in which of the remaining candidates a k-linear transformation occurrence of the pattern really starts.

2. Pattern Analysis.
   The *witness* table is computed using the naive algorithm. First set $l_1 = 1$. Suppose $l_1, l_2, ..., l_{i-1}$ were determined. The $i-1$ independent equations are kept as a triangulated system. In order to determine the value of l_i we search locations $j \geq l_{i-1} + 1$, in the upper patterns. Each location j suggests an equation. We add the equation to the system of the $i-1$ independent equations and triangulate the system so that it has i independent equations and l_i is set to be j. Otherwise, we continue to the next location. If all locations $j \geq l_{i-1} + 1$ do not suggest an ith independent equation, then each of $l_i, l_{i+1}, ...l_{2k}$ is set to zero.

Finally, the authors presented the algorithm for the minimum distance k-linear transformation problem. They defined the minimum distance at position j, as follows. For each position j, find k numbers $c_{j,1}, c_{j,2}, ..., c_{j,k}$ which provide the minimum in

$$L_j = \sum_{i=1}^{m} (T_{j-1+i} - \sum_{l=1}^{k} c_{j,l} B_{l,i})^2$$

The minimum distance k-linear transformation problem is to find a position j and numbers $c_{j,1}, ..., c_{j,k}$ for which L_j is a global minimum.
The algorithm is:

1. Compute $\sum_{i=1}^{m} B_{l,i} B_{r,i}, \forall l, r, 1 \leq l, r \leq k$.
2. Compute $\sum_{i=1}^{m} T_{j-1+i}^2, \forall j, 1 \leq j \leq n - m + 1$.
3. Compute the k convolutions $\sum_{i=1}^{m} T_{j-1+i} B_{l,i}, \forall j, 1 \leq j \leq n - m + 1$ for each $l, 1 \leq l \leq k$.
4. Compute for each j the minimum value of L_j.
5. Find a position j for which a global minimum L_j is achieved.

2.5 F. Bergeron and S. Plouffe's Work

In 1991, S. Plouffe and Francois Bergeron presented a strategy: *Computing the Generating Function of a Series Given its First Terms* [4]. In their technical report, they outlined the strategy and gave the Maple program which produced generating functions for about 600 sequences out of the 4568 sequences appearing in the projected second version of Sloane's handbook. The main idea of their method was to use operations on a given series that might transform it into a series admitting a rational generating function. Upon success, one needs only compute the inverse of these operations on the rational function obtained. The critical part of the program was a test for the existence of a *good* rational generating function for a series. They used **convert/ratpoly** function of Maple with a test on the sum of degree of the numerator and denominator. And the most important part of the program was in the rationality test with some operations like derivatives, logarithmic derivative, reversion of series.

The strategy is as follows:

1. Perform transformation $\mathcal{T}$ on a given series

$$\alpha = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n + O(x^{n+1}).$$

2. Then test the resulting series $\mathcal{T}(\alpha)$ for rationality

$$\mathcal{T}(\alpha) = b_0 + b_1 x + b_2 x^2 + ... + b_k x^k + O(x^{k+1}).$$

3. If $\mathcal{T}(\alpha)$ admits a rational generating function $f(x)$, then the program computes $\mathcal{T}^{-1}(f(x))$, where $\mathcal{T}^{-1}$ is the inverse transformation for $\mathcal{T}$.

Following are three examples they presented in their paper, which were generated by using their program:

1. Example 1

$$sequence : 2, 5, \frac{11}{2}, \frac{19}{3}, \frac{29}{4}, \frac{41}{5}, \frac{55}{6}, \frac{71}{7}, \frac{89}{8}, \frac{109}{9}.$$

$$generating function : \frac{2 - x^2}{(1-x)^2} + \ln(\frac{1}{1-x}).$$

This result illustrates the use of the derivative in the program.

2. Example 2

$$sequence: 1, 1, 1, \frac{5}{6}, \frac{17}{24}, \frac{73}{120}, \frac{97}{180}, \frac{2461}{5040}, \frac{3631}{8064}, \frac{152531}{362880}.$$

$$generatingfunction: \frac{e^{\frac{x^2}{4}+\frac{x}{2}}}{\sqrt{1-x}}.$$

This result illustrates the use of the logarithmic derivative in the program.

3. Example 3

$$sequence: 1, 3, 12, 55, 273, 1428, 7752, 43263, 246675.$$

$$generatingfunction: f(x) = \frac{(12\sqrt{81x-12} - 108\sqrt{x})^{\frac{1}{3}} - (12\sqrt{81x-12} + 108\sqrt{x})^{\frac{1}{3}}}{6\sqrt{x}} - 1.$$

This result illustrates the use of a rationality test on the reverse (substitutional inverse) of a series. The reverse of this generating function is

$$\frac{x}{(1+x)^3}$$

Hence the generating function $f(x)$ is obtained as the real solution of the cubic equation

$$(1 + f(x))^3 x - f(x) = 0.$$

Later in 1992, as part of his Masters' degree fulfillment [10], S. Plouffe presented his enhancement for their results by adding three so-called *P-recurrences, Euler and Recoupment* methods in the Maple program, and detailed his discoveries by listing 1031 sequences, which could be produced their generating functions by using the program mentioned above, which were out of 4568 sequences in the projected second version of Sloane's handbook. In his article, he illustrated all 1031 such sequences, the related generating functions, the sequence number for both sequences in *The Handbook* [12] and in the new version one. Moreover, he showed the kind of methods: methods in [4], or methods in [10] , from which the generating function was produced.

2.6 A. Bhansali and S. S. Skiena's Attempt

Recently, in 1994, A. Bhansali and S. S. Skiena introduced a concept: $(RD)^*$ sequences, a class of sequences defined by a generalization of difference and ratio table algorithms in their paper [1].

1. Difference table is defined as follows:
   Let $\{a_n\}$ be a sequence, where $n = 1, 2, \ldots$
   relation:
   $$\triangle^m a_n = \triangle^{m-1} a_{n+1} - \triangle^{m-1} a_n,$$
   $$\triangle^0 a_n = a_n$$
   this will lead to kth-degree polynomial:
   $$P(n,m) = \sum_{i=0}^{m} \binom{n}{i} \triangle^i a_0$$

2. Ratio table is defined as follows:
   relation:
   $$\delta^m a_n = \frac{\delta^{m-1} a_{n+1}}{\delta^{m-1} a_n},$$
   $$\delta^0 a_n = a_n$$
   will lead to kth-degree exponential function:
   $$P(n,m) = \prod_{i=0}^{m} (\delta^i S_0)^{\binom{n}{i}}$$

3. $(RD)^*$ table is defined as follows:
   $(RD)^*$ table is the combination of *Difference* and *Ratio* tables

4. A $(RD)^*$ sequence is a sequence S for which there exists a finite heig ht integer (RD^*) table which generates S.

5. Let f be a function defining the sequence S at level $d+1$ of a $(RD)^*$ table. Then, if the operations are all difference, the closed form g representing S is:
   $$g(n) = \sum_{i=0}^{d} \binom{n}{i} \triangle^i g_0 + \sum_{i=0}^{n-d-1} \binom{n-i}{d} f_i$$

If the operations are all ratio, the closed form g representing S is:

$$g(n) = (\prod_{i=0}^{d} (\delta^i g_0)^{\binom{n}{i}})(\prod_{i=0}^{n-d-1} f_i^{\binom{n-i}{d}})$$

6. Results:
   The integer $(RD)^*$ table of minimum height k associated with a sequence S of length n can be found in $O(kn)$ time.

The paper [1] also presented the implementation of SEQ, a program for sequence analysis by using above ideas.

2.7 Our Approach

Our approach is closely reflect our goal: *building an electronic handbook of integer sequences.* such an electronic handbook has the capabilities: to query the known sequences in the Handbook [4, 10] for a given sequence according to certain relationship by a *query language*, to *deduce* the sequence to see if it fits certain standard forms, or to*compute* additional terms of the sequence. Our approach is a *searching* strategy rather than a *computing* strategy. To *recognize* a given sequence, our method is to search the integer sequence database to see if the sequence has a or more relations with the sequence(s) in the database. Let S be a given sequence, D be the sequence database. These relations are as follows:

1. Is there an exact match between S and $T_i \in D$?
2. Is S simply some sequence in the D shifted ?
3. Are all members of S contained in some sequence $T_i \in D$?
4. Is S an affine transformation of some sequence $T_i \in D$?
5. Can S be written as a polynomial, of degree bounded by a constant ?
6. Can S be written as the linear combination of two T_i and $T_j \in D$?
7. Is the S close to a T_i?
8. Does S leave a constant remainder when divided by a T_i ?

9. Does affine transformation of S give a subsequence of some $T_i \in D$?

Querying thousands of integer sequences in nine (or more later) forms makes our system more complicated and challenging to be implemented. Due to the increasing volume of integer sequences, efficient algorithms must be created to evaluate those queries, and to improve the system's responsiveness. In this paper, we are presenting some of these nine query evaluation algorithms. The efficiency is obtained through preprocessing the database, use of hashing techniques and use of number theory. The detail algorithms is given in section 3.

3 Queries

There are 9 pattern queries we present here. They are the direct consequences of our goal, implementing an *electronic handbook* of Sloane's handbook. Most of the queries gain efficiencies by means of preprocessing the integer sequence database $\mathcal{D}$.

Definition 1: Let $n, m > 0$ be integers, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$ and $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ be two integer sequences, we say two sequences $S1$ and $S2$ are identical, denoted as

$$S1 = S2$$

if and only if

$$s1[k] = s2[k], k = 0, 1, ..., m, \quad and \quad n = m.$$

Definition 2: Let $n, m > 0$ be integers, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$ and $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ be two integer sequences, we say two sequences $S1$ and $S2$ are an exact match, denoted as

$$S1 \equiv S2$$

if and only if

$$s1[k] = s2[k], k = 0, 1, ..., h; h = \min\{m, n\}.$$

Definition 3: Let $n, m > 0$ be integers, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$ and $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ be two integer sequences, we say $S1$ is a shifted sequence of $S2$, denoted as

$$S1 \Longrightarrow S2$$

if and only if $\exists g \in [0, m-1]$ such that,

$$s1[k] = s2[k+g], k = 0, 1, ..., h; h = \min\{m-g, n\}.$$

Definition 4: Let $n, m > 0$ be integers, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$ and $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ be two integer sequences, we say $S1$ is a subsequence of $S2$, denoted as

$$S1 \in S2$$

if and only if $\exists h_k \in [0, m]$ such that,

$$s1[k] = s2[h_k], k = 0, 1, ..., n; h_k < h_{k+1}.$$

Definition 5: Let $n, a > 0$ be integers, and let $S = \{s[0], s[1], s[2], ...s[n]\}$ be an integer sequence, we denote the integer sequence $T = \{a \cdot s[0], a \cdot s[1], a \cdot s[2], ...a \cdot s[n]\}$ as

$$T = a \cdot S.$$

Definition 6: Let $n, b > 0$ be integers, and let $S = \{s[0], s[1], s[2], ...s[n]\}$ be an integer sequence, we denote the integer sequence $T = \{s[0] + b, s[1] + b, s[2] + b, ...s[n] + b\}$ as

$$T = S + b.$$

Definition 7: Let $n, q > 0$ be integers, and let $S = \{s[0], s[1], s[2], ...s[n]\}$ be an integer sequence, we say sequence $T = \{s[0]^q, s[1]^q, s[2]^q, ...s[n]^q\}$ is the qth-power sequence of S, denote as

$$T = S^q.$$

Definition 8: Let $n > 0$ be an integer, and let $S = \{s[0], s[1], s[2], ...s[n]\}$ be an integer sequence, we denote $|S|$ as the length of sequence S, thus, we have

$$|S| = n + 1.$$

Definition 9: Let $n, m > 0$ be integers, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$ and $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ be two integer sequences, we denote $|S1 - S2|_2$ be the distance of two sequences $S1$ and $S2$, we define:
1. when $m \leq n$,

$$|S1 - S2|_2 = \min_{h \in [0, n-m]} \sqrt{\sum_{k=0}^{m} (s[k] - T_i[k+h])^2}.$$

2. when $m > n$

$$|S1 - S2|_2 = \min_{h \in [0, m-n]} \sqrt{\sum_{k=1}^{m} (s[k+h] - T_i[k])^2}.$$

Definition 10: Let $n, m, w > 0$ be integers, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$, $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ and $T = \{t[0], t[1], t[2], ...t[w]\}$ be three integer sequences, we say

$$S1 \equiv T \pmod{S2},$$

if and only if, for $\forall k \in [0, w]$, where $w \leq \min\{m, n\}$, such that,

$$s1[k] \equiv t[k] \pmod{s2[k]}.$$

Definition 11: Let $n, m > 0$ be integers, let $c > 0$ be constant, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$ and $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ be two integer sequences, we say

$$S1 \equiv c \pmod{S2},$$

if and only if, for $\forall k \in [0, h]$, where $h = \min\{m, n\}$, such that,

$$s1[k] \equiv c \pmod{s2[k]}.$$

Definition 12: Let $n, m > 0$ be integers, and let $S1 = \{s1[0], s1[1], s1[2], ...s1[n]\}$ and $S2 = \{s2[0], s2[1], s2[2], ...s2[m]\}$ be two integer sequences, we denote sequence $\{s1[0] + s2[0], s1[1] + s2[1], s1[2] + s2[2], ..., s1[h] + s2[h]\}$, where $h = \min\{m, n\}$ as

$$S1 + S2.$$

3.1 Query 1:

Is there an exact match between S and $T_i \in \mathcal{D}$? That is,

$$\exists i,\ T_i \equiv S$$

Evaluation strategy:

- *Preprocessing:* Every time when new sequence(s) are added to the database, we need to sort the new database to make an updated one with sequences in lexicographical order.

- *Search Algorithm:* Using binary search:
  Let N be the number of integer sequences in the database, $h_1 = 1$ and $h_2 = N$, then
  Step 1: binary search, such that

  $$\exists i \in [1, N], \qquad T_i \equiv S$$

  that is, compute $h = \lceil \frac{h_1+h_2}{2} \rceil$, check if $T_h \equiv S$ then exit *step 1.*
  if $T_h > S$ then $h = \lceil \frac{h+h_2}{2} \rceil$
  otherwise, if $T_h < S$ then $h = \lceil \frac{h_1+h}{2} \rceil$
  if $h = h_1$ or $h = h_2$ exit *step 1.*
  go back to do checking.
  Step 2: backward linear search, such that

  $$\forall j < i, T_j \equiv S$$

  that is, in case of a match $T_i \equiv S$
  let $j = i - 1$ check if $T_j \equiv S$
  if it is true repeat until a false.
  Step 3: forward linear search, such that

  $$\forall s > i, T_s \equiv S$$

  that is, in case of a match $T_s \equiv S$
  let $s = i + 1$ check if $T_s \equiv S$
  if it is true repeat until a false.
  Step 4: Retrieve those sequence T_r, where

  $$\forall r \in [j, s]$$

Because the chance of an exact match is rare, on average the algorithm runs in $O(M\log(N))$,
where $M = \min(|S|$, average of $|T_i|)$, $N =$ number of T_i in the database.

- *Performance tests:*

Queries	Response time
10(times)	0.03(seconds)
20	0.06
30	0.10
40	0.12
50	0.19
100	0.40
150	0.61
200	0.81
300	1.22
400	1.60
500	2.03
600	2.41
700	2.79
800	3.15
900	3.53
1000	3.91

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

3.2 Query 2:

Is S simply some sequence $T_i \in \mathcal{D}$ shifted ? That is,

$$\exists i \; S \Longrightarrow T_i$$

Evaluation strategy:

- *Search Algorithm:* Using $Knuth - Morris - Pratt(KMP)$ [13] matching algorithm: first we compute the prefix function, then we check

whether or not there is a full number integer match. (Details of the KMP algorithm can be found in [13])

- *Complexity of Algorithm:* On average, the algorithm runs in O(MN), where $M = \min(|S|$, average of $|T_i|)$, $N =$ number of T_i in the database.
- *Performance tests:*

Queries	Response time
10(times)	1.01(seconds)
20	2.01
30	3.00
40	4.02
50	5.01
100	10.1
150	15.2
200	20.2
300	30.3
400	40.3
500	50.4
600	63.5
700	73.7
800	83.5
900	93.2
1000	103.3

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

3.3 Query 3:

Are all members of S contained in some sequence $T_i \in \mathcal{D}$? That is,

$$\exists i,\ S \in T_i$$

Evaluation strategy:

- *Search Algorithm:* This is a naive algorithm: For a given sequence $S = (s[1], s[2], ...s[m])$ and a sequence T_i in the database, First we check if $\exists h_1$ such that $s[1] = T_i[h_1]$, then we check if $\exists h_2 > h_1$ such that $s[2] = T_i[h_2]$, and so on. Finally, we check if $\exists h_m > h_{m-1}$ such that $s[m] = T_i[h_m]$.

- *Complexity of Algorithm:*
  An upper bound on the running time of the algorithm is $O(NM)$, where $M = \max|T_i|$.

- *Performance tests:*

Queries	Response time
10(times)	0.73(seconds)
20	1.42
30	2.17
40	3.00
50	3.76
100	7.84
150	12.8
200	17.0
300	24.8
400	34.3
500	41.4
600	49.0
700	56.7
800	64.9
900	73.2
1000	81.0

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

3.4 Query 4:

Is the sequence S an affine transformation of some sequence $T_i \in \mathcal{D}$? That is,

$$\exists i, a, b,\ S \equiv aT_i + b$$

Evaluation strategy:

- *Preprocessing:* Our idea comes from the paper [9]. First, we compute the differences of all the terms of T_i, then divide each term by its next term in the *difference sequence* to obtain a new sequence. Namely,

$$\mathcal{T}_i[k] = T_i[k+1] - T_i[k]; \Gamma_i[k+1] = \frac{\mathcal{T}_i[k+1]}{\mathcal{T}_i[k]}, \forall k \geq 0$$

  Then, we sort all the Γ_i in lexicographical order to form a table $\mathcal{J}$.

- *Search Algorithm:* Given a sequence S, we simply use binary search to search the table $\mathcal{J}$ to do exact match search. Namely, search to see if

$$S \equiv \Gamma_i, \exists \Gamma_i \in \mathcal{J}$$

  We fetch all the indices of matched sequences Γ_i, which are candidates using the ideas in [9], Finally, we confirm those sequences for the affine transformation matches by exact checking: $S \equiv aT_i + b$.

- *Complexity of Algorithm:* On average, algorithm runs on $O(\ln(NM))$ $\approx O(\ln(N))$, where $M \ll N$.

- *Performance tests:*

Queries	Response time
10(times)	0.05(seconds)
20	0.11
30	0.16
40	0.21
50	0.27
100	0.58
150	0.85
200	1.18
300	1.76
400	2.34
500	3.04
600	3.61
700	4.15
800	4.71
900	5.37
1000	5.95

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

3.5 Query 5:

Can the sequence S be written as a polynomial, of degree bounded by c, in some $T_i \in \mathcal{D}$? That is,

$$\exists i, p < c,\ S \equiv \mathcal{P}(T_i) = a_0 + a_1 T_i + a_2 T_i^2 + ... + a_p T_i^p$$

Evaluation strategy:

- *Search Algorithm:* This is a naive algorithm. Assuming $p < |S|$, we solve the equation

$$\begin{pmatrix} s_[1] \\ s_[2] \\ s_[3] \\ . \\ . \\ . \\ s_[p+1] \end{pmatrix} = \begin{pmatrix} 1 & T_i[1] & T_i[1]^2 & T_i[1]^3 & . & . & . & T_i[1]^p \\ 1 & T_i[2] & T_i[2]^2 & T_i[2]^3 & . & . & . & T_i[2]^p \\ 1 & T_i[3] & T_i[3]^2 & T_i[3]^3 & . & . & . & T_i[3]^p \\ 1 & T_i[4] & T_i[4]^2 & T_i[4]^3 & . & . & . & T_i[4]^p \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ 1 & T_i[p+1] & T_i[p+1]^2 & T_i[p+1]^3 & . & . & . & T_i[p+1]^p \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \\ . \\ a_p \end{pmatrix}.$$

and then, *confirm* the match by exact term by term checking

$$s_[k] = a_0 + a_1 T_i[k] + a_2 T_i[k]^2 + ... + a_p T_i[k]^p, p < k \leq |S|$$

However, we use a better algorithm as follows:
because

$$\mathcal{P}(T_i[g]) = \sum_{j=0}^{p} \prod_{h \neq j}^{0<h\leq p} \left(\frac{(s[i][g] - s[i][h])}{(s[i][j] - s[i][h])} s[i][j]\right), g \leq p$$

our confirmation checking is simply done by

$$\mathcal{P}(T_i[k]) = \sum_{j=0}^{p} \prod_{h \neq j}^{0<h\leq p} \left(\frac{(s[i][k] - s[i][h])}{(s[i][j] - s[i][h])} s[i][j]\right), p < k \leq |S|$$

- *Complexity of Algorithm:* Compute $\mathcal{P}(T_i)$ needs $O(M^2)$, so an upper bound for the algorithm is $O(M^2 N)$, where $M = \min\{c, |S|, \max|T_i|\}$.

- *Performance tests:*

Queries	degree=4	degree=6
10(times)	2(seconds)	3(seconds)
20	3	5
30	5	8
40	7	11
50	8	14
100	15	29
150	24	42
200	32	56
300	47	84
400	62	113
500	76	141
600	92	169
700	107	197
800	124	224
900	143	251
1000	163	278

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

3.6 Query 6:

Can S be written as the linear combination of two T_i and $T_j \in \mathcal{D}$? That is,

$$\exists i, j, a, b,\ S \equiv aT_i + bT_j$$

Evaluation strategy:

- *Preprocessing:*
  First pick a small set of primes, say P = {2, 3, 5, 7, 11, 13, 17}, and then for each prime $p \in P$, we do the following:
  for each b $\in$ {0, 1, 2, ... , p-1}, we compute $bT_j[i] \pmod p$ for $i = 1$, 2, ... e_p, where $Np^{-e_p} \approx 1$, or $e_p = \lceil \log_p(N) \rceil$.
  Then, we build the sorted lists in such a way that each item in the list has the mapping into the same point $\mathcal{H}$, which is a vector of $(a_1, a_2, ...a_{e_p})$.

Thus, we produced an hashing table, which will facilitate our search later for this query.

- *Preprocessing Cost:*
  Runing time: The computation $bT_j[i] \pmod p$ need a total

$$N \sum_{p \in P} (p * e_p) = N \sum_{p \in P} (p \lceil \log_p(N) \rceil)$$

multiplications and mod operations. So it costs $O(N\ln(N))$ running time. For example, if $N = 2372$, then it will be

$$(2*11+3*7+5*5+7*4+11*3+13*3+17*3)N = 219*2372 = 519468$$

multiplications and modulus.
Consumption Space: For each prime p, there are p mappings into the table for each sequence T_i, plus N headers hashing to the lists, therefore there are

$$N * \sum_{p \in P} p + N = 59N$$

nodes. And each node has 8 bytes, thus total space needed is $445N$, which is $O(N)$. For example, if N=2372, the table will be approximately 1.4 Mbytes.

- *Search Algorithm:* Given a sequence S we need to search, our algorithm proceeds in following steps,

  1. *Phase 1: finding candidates.*
     1. For each $i \in$ [1,N]
     2. For $p = 2$ do
     compute $(S - aS_i[j]) \pmod 2$ to obtain a vector H_a, where $j =$ 1, 2,...e_2; $a =$ 0,1,
     3. For $a = 0, 1$, lookup the hashing table and fetch a pointer pointing to a list L_a of indices by using H_a. Do the concatenation of two lists L_0 and L_1 to initiate a new sorted list $\mathcal{L}$.
     4. For each $p \in \{3, 5, 7, 11, 13, 17\}$ do
     If list $\mathcal{L}$ empty, exit.
     5. For each $a \in \{1, 2, \ldots$ p-1$\}$ do
     Compute the $S - aS_i[j] (\mathrm{mod} p)$ to obtain a vector H_a, where $j =$

1, 2, ... e_p. Lookup the hashing table and fetch a pointer pointing to a list L_a of indices by using H_a.
6. Do the intersection by *Merge Join* two sorted lists of L_a and $\mathcal{L}$ to create a new sorted list $\mathcal{T}$.
6.0. Assume $l1[i] \in L_a$ and $l2[j] \in \mathcal{L}$, the *Merge Join* algorithm is as follows,
6.1. Begin with $i = j = 1$, if $l1[i] > l2[j]$, $j = j + 1$;
6.2. Else if $l1[i] < l2[j]$, $i = i + 1$;
6.3. Else if $l1[i] = l2[j]$ remove $l2[j]$ from $\mathcal{L}$, then add it in a sorted list $\mathcal{T}$ and
6.4. $j = j + 1$ and $i = i + 1$.
7. Thus, building a temporary list $\mathcal{T}$ by adding the removed items. if list $\mathcal{L}$ empty, exit.
8. Finally, assign the sorted list $\mathcal{T}$ to $\mathcal{L}$.
As a result, we produce a list of indices $\mathcal{L}$ (maybe empty) for the i, say $(i_1, i_2, ...i_j)$. We conclude that it could be true that $S = aT_i + bT_{i_j}$, $i_j \in [1, N]$, which we call i_j as candidates for i.

2. *Phase 2: confirmation.*
Do the exact term by term checking for the query $S \equiv aT_i + bT_{j_i}$.

- *Analysis of Algorithm:*
Because $Np^{-e_p} \approx 1$, it means that on average, there is one item in each entry of the hashing table. There would be on average p hits of $S - aT_i$, for each prime p. Thus, the initial list $\mathcal{L}$ would most likely have 2 items for all $i \in [1, N]$. So now, when we proceed prime 3 and do the intersection, we are in the situation of using a list each with about 3 items to intersect the list at hand with 2 items. What will be the results ? Obviously, if we assume the integers are uniformly distributed in the database, the possibility for one item in *3 item list* to match one of the item in *2 item list* at hand will be $\frac{2}{N}$. And there are 3 items, the possibility for the intersected list to have one item would therefore be $3 * \frac{2}{N} = \frac{6}{N}$. For the same reason, the possibility of the intersected list with 2 items will be $\frac{9}{N^2}$, and so on. Because, there are total N lists in the table, and there would be $\frac{N*6}{N} = 6$ lists with one item, and $N * \frac{9}{N^2} = \frac{9}{N} \approx 0$ lists with 2 items or more, after the (2,3) intersection. So on average, there will be only 6 lists participating the prime 5 intersection, which is considered to be constant in running

time. After *prime 5* intersection, the possibility for a list to have one item is $\frac{5}{N} * 6 \approx 0$ (for $N \gg 30$). As a result, usually there would not be any list to be processed after *prime 5 intersection.* All the computation mainly is in *prime 2, prime 3, prime 5* phases.
Therefore, on average there will be

$$\sum_{(2,3)}^{i} (p_i * e_{p_i} * N) + 5 * 4 * 6 = (2 * 12 + 3 * 8)N + 120 \approx N\ln(N)$$

multiplications, subtractions and mod operations, and (2*3+2) N = $8N$ comparisons. The *confirmation phase* could be considered in constant running time. Thus, the algorithm runs in $O(N\ln(N))$.

- *Performance tests:*

Queries	Naive Algorithm	Our Algorithm
10(times)	595(seconds)	28(seconds)
20	1230	59
30	1961	90
40	2685	121
50	3259	150
100	5543	299
150	7876	436
200	10230	591
300	14858	891
400	19511	1187
500	24109	1488
600	28719	1783
700	33128	2086
800	37685	2385
900	43119	2683
1000	48881	2982

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.x / Sun-Sparc.

3.7 Query 7:

Is the S close to a $T_i \in \mathcal{D}$? That is,

$$\exists i, c, \ |S - T_i|_2 < c$$

Evaluation strategy:
We use L_2 Euclidean distance to compute the distance.

- *Search Algorithm:* For a given sequence $S = (s[1], s[2], ...s[m])$ and a sequence T_i in the database, we consider two cases:
  1. $m = |S| \leq |T_i| = w$
  Compute and check to see if $\exists h \in [0, w - m]$, such that

$$\sqrt{\sum_{k=1}^{m} (s[k] - T_i[k + h])^2} < c$$

  retrieve those T_i satisfying the equation.
  2. $m = |S| > |T_i| = w$
  Compute and check to see if $\exists h \in [0, m - w]$, such that

$$\sqrt{\sum_{k=1}^{m} (s[k + h] - T_i[k])^2} < c$$

  retrieve those $T_i s$ satisfying the equation.

- *Complexity of Algorithm:* The algorithm runs in $O(mwN)$.

- *Performance tests:*

Queries	Response time
10(times)	44(seconds)
20	64
30	85
40	108
50	129
100	242
150	352
200	462
300	726
400	974
500	1248
600	1496
700	1750
800	1995
900	2234
1000	2454

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

3.8 Query 8:

Does S leave a constant remainder when divided by a $T_i \in \mathcal{D}$? That is,

$$\exists i, S \equiv c \pmod{T_i}$$

Evaluation strategy:

- *Search Algorithm:* Given a sequence $S = (s[0], s[1], ...s[m])$, we simply check whether or not there is a constant for $\forall k \in [1, h], h = \min\{m, |T_i|\}, i \in [1, N]$, such that

$$s[k] \equiv c \pmod{T_i[k]}$$

More precisely, first compute $remainder = s[1] \bmod t_i[1]$, then check to see if $remainder = s[k] \bmod t_i[k], \forall k \in [1, h]$
Notice that we simply skip the items with $T_i[k] = 0$.

- *Complexity of Algorithm:* Obviously, an upper bound of the algorithm will be $O(MN)$, where $M = \min\{m, \max|T_i|\}$.

- *Performance tests:*

Queries	Response time
10(times)	1(seconds)
20	2
30	3
40	4
50	5
100	9
150	13
200	18
300	27
400	36
500	46
600	55
700	64
800	74
900	82
1000	91

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

3.9 Query 9:

Does some affine transformation of S give a subsequence of some $T_i \in \mathcal{D}$? That is,

$$\exists a \neq 0, b,\ aS + b \in T_i$$

Evaluation strategy:

- *Search Algorithm:* Given a sequence $S = (s[0], s[1], s[2], ...s[m])$, the search algorithm proceeds in following steps (assume $m < |T_i|$):

1. Pick the smallest $k > 0$, such that $s[k] \neq s[k+1]$.
2. Advance p, such that p is the smallest integer with $T_i[q_1] = T_i[q_2] = ... = T_i[q_k]$, and $q_r < p$, r = 1,2,...k. If not exists such $p \in [1, |T_i|]$, exit.
3. For $j = 1$.
4. Solve the equation.

$$\begin{pmatrix} s[k] & 1 \\ s[k+1] & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} T_i[p] \\ T_i[p+j] \end{pmatrix}.$$

5. Check the following items of sequence T_i by using unique solution a, b, namely to see if $\exists h_q$, such that

$$a * s[g] + b = T_i[q_k], g = 1, 2, ..., k-1;$$

$$a * s[k+1+q] + b = T_i[p+j+h_q], q = 1, 2, ..., m-k-1, h_q < h_{q+1} < |T_i|$$

6. If true, query is confirmed and retrieve T_i. Otherwise advance j by 1, and back to step 4.

- *Complexity of Algorithm:* Each checking of S needs $m = |S|$ steps, and the worse case is to go through the entire sequence T_i with $M = |T_i|$ steps. There are N sequences in total, consequently, the algorithm runs in $O(mMN)$.

- *Performance tests:*

Queries	Response time
10(times)	30(seconds)
20	58
30	87
40	119
50	147
100	311
150	491
200	642
300	983
400	1277
500	1576
600	1875
700	2177
800	2477
900	2805
1000	3105

1. Above 1000 queries were randomly generated during testing.
2. The sequence database is based on the *Handbook* [12].
3. Testing Environment: SunOS 4.1.2 / Sun-Sparc.

4 Comparison with a similar system:

4.1 Superseeker of AT&T Bell Labs.

Recently, there is an integer sequence on-line lookup system appearing, which is called *superseeker*, built by Neil Sloane, Simon Plouffe, Bruno Salvy and others. It is an on-going system still under development, and the authors expect a new version will be launched in a few months. In the *superseeker* system, users can query an integer sequence by sending it via e-mail. When the *superseeker* received a lookup sequence, it will be processed in following steps:

1. Do an exact match checking to see if a match in the *On-Line Encyclopedia of Integer Sequences* (which is an extension of the sequences in *A Handbook of Integer Sequences* [12]).

2. Test if $S[n]$ is a polynomial in n.

3. Test if the differences of some order $\{S_d[n]\}$ are periodic.

4. Test if any row of the difference table of some depth is essentially constant.

5. Form some generating functions for the sequence for each of types:

   - ordinary generating function
   - exponential generating function
   - reversion of ordinary generating function
   - reversion of exponential generating function
   - logarithmic derivative of ordinary generating function
   - logarithmic derivative of exponential generating function

   Interested readers can find more details on these generating functions in paper [9].

6. Look for a linear recurrence with polynomial coefficients for the coefficients of the above 6 types of generating functions.

7. Apply the 98 transformations to the sequence and look up the result in the table. (Interested readers can have details of the 98 transformations by sending an e-mail to *superseeker@research.att.com* with a single line of integer sequence). For example , following are some of the 98 transformations:

   - Odd index in the sequence.
   - Even index in the sequence.
   - sequence $\frac{S[n]}{n}$.
   - sequence $S[n+1] - S[n]$.
   - sequence $S[n+2] - S[n]$.
   - sequence $S[n+1] + S[n]$.
   - sequence $S[n+2] + S[n]$.

- sequence $S[n] + n$.
- sequence $S[n] + 1$.
- sequence $\frac{V[n]=S[n]}{(n-1)!}$.
- sequence $V[n+1] - V[n]$.
- sequence $V[n+2] - V[n]$.
- sequence $V[n+1] + V[n]$.
- sequence $V[n+2] + V[n]$.
- sequence $V[n] + n$.
- sequence $V[n] + 1$.

8. If the original sequence is not in the table, find the 3 closest sequences in the table using the L1 metric. Only those with L1 ≤ 3 are reported.

4.2 Comparison

Basically, *superseeker's* lookup strategy is to search the sequence table, to test if there are exact matches between an given sequence or its transformations and the sequences in the table. Our approach is also search the similar sequence table, so we share the same methodology of querying a set of well-know sequences. Our difference is the way we have done the querying. *Superseeker* uses the given sequence or its transformation sequences to query the database. Its queries are mainly of exact match searches. While we use the given sequence itself to query the database, and our queries are not just exact matches. They could be the *linear combination of two sequences in the table*, or the *affine transformation of a sequence* or the *the polynomial of a sequence bounded by a constant.* These differences make our system more complicated and challenging to be implemented. Due to the increasing volume of integer sequences, efficient algorithms must be created to evaluate those queries, and to improve the system's responsiveness. The algorithms we have present in section 3 are just part of our efforts to solve the problem.

4.3 Our Discoveries

Because our different approach from *Superseeker*, we have *recognized* sequence patterns which *Superseeker* could not recognize. For example,

by using Query 7: linear combination of two sequences, we found that there are relations among sequences in *Handbook* [12] as follows,
$T_{1363} = T_{419} + T_{616}$
$T_{1625} = T_{659} + T_{1248}$
$T_{1746} = 2 * T_{1205}$
$T_{1923} = 2 * T_{825} + T_{1428}$
$T_{2008} = 2 * T_{789} + T_{1709}$
by using Query 4: affine transformation of a sequence, we found that,
$T_{697} = T_{1105} - 1$
$T_{727} = T_{1118} - 1$
$T_{833} = T_{1443} - 2 = T_{1239} - 1$
$T_{974} = 2 * T_{256} - 1$
$T_{1084} = T_{1544} - 2$
$T_{1141} = T_{1382} - 1$
$T_{1239} = T_{1443} - 1$
$T_{1331} = 3 * T_{188} - 2$
$T_{1459} = 2 * T_{1192} - 2$
$T_{1536} = 2 * T_{921} - 1$
$T_{1567} = 2 * T_{1049} - 1 = 2 * T_{616} + 1 = 4 * T_{391} - 3$
$T_{1587} = 2 * T_{1125} - 1$
$T_{1622} = 2 * T_{1191} - 1$
$T_{1746} = 2 * T_{1205}$
$T_{1748} = 2 * T_{1428} - 2$

Note that above T_i means the ith sequence in the *Handbook* [12].

5 Future Work

We are presenting a work still in progress. The research presented in this essay can certainly be extended in many ways.
One way is that, to design and implement the query language, to extend the query set, and provide efficient compilation strategies of such queries. Such extension will facilitate the sequence search and provide a more friendly user interface and improve the system's responsiveness.
Another way is to create new evaluation algorithms for some of the queries. For example, *Query 5: polynomial bounded by constant match* could be im-

proved by new algorithms. This would be the same case for *Query 10: some affine transformation is a subset of some sequences in the table.* Right now we are only using naive algorithms to handle this two queries. We have tried *Query: linear combination of three sequences*, and we found its processing time was extremely long, more than twenty hours for only one query ! It is not acceptable. Is there any techniques to solve the problem of this particular query ?

Extensions could also be made by integrate some transformation capability with the search mechanism. Unlike *superseeker* which does all the 98 transformations before looking up the table, we should build the transformation capability inside the query language. Providing methods for deducing sequences is another important extension. Researchers have provided efficient solutions to deduce that a sequence fits one of a number of standard forms. Although, attempts like building deducing sequence capabilities on top of an algebra system have been made (see [5]), or enumerating some easy *inferring rules* to deduce a sequence(see [5] or [8]). However, progress is too little in this field in comparing with the progress in other fields of computer science. Finally, once we recognize an integer sequence, an user may reasonably request to see additional terms, or a specific term not stored. So, it is valuable to provide methods to compute additional terms of a given sequence. D. Angluin [2] and A. Dewdney [8] have attempted to provide some tricks to handle the problem. More studies should be conducted to deal with this particular problem and efficient algorithms should be created in the near future.

6 Acknowledgement

I wish to acknowledge the debt of gratitude I owe to my supervisor, Dr. J. O. Shallit. This essay would not have been possible without his guidance and inspiration.

References

[1] A. Bhansali and S. S. Skiena's, Analyzing Integer Sequences, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 15, 1994.

[2] D. Angluin, Easily inferred sequences, Memorandum ERL-M499, Electronics Research Laboratory, University of California, Berkeley, 31 December 1974.

[3] J.-P. Allouche and J. Shallit. The ring of k-regular sequences. *Theoretical Computer Science* **98** 163–187, 1992.

[4] F. Bergerom and S. Plouffe, Computing the generating function of a series given its first terms, Technical Report No. 164, Université du Quebec à Montŕeal, Oct. 22, 1991.

[5] J. Calmet and I. Cohen, Symbolic manipulation of recurrence relations: an approach to the manipulation of special functions, in N. Inada and T. Soma, eds., *Symbolic and Algebraic Computation by Computers,* World Scientific Publishing, 55–65, 1985.

[6] J. Shallit. Integer sequence project. unpublished notes, Department of Computer Science, University of Waterloo, May 20, 1991.

[7] P. J. Cameron, Some sequences of integers, *Discrete Math.* **75** 89–102, 1989.

[8] A. K. Dewdney, Computer recreations: how a pair of dull-witted programs can look like geniuses on I.Q. test, *Sci. Amer.* **254(3)** 14–21, (March 1986).

[9] T. Eilam-Tzoreff and U. Vishkin, Matching patterns in strings subject to multi-linear transformations, in R.M. Capocelli, ed., *Sequences,* Springer-Verlag, 45–57, 1990.

[10] S. Plouffe, Approximations de series generatrices et quelques conjectures, Unité de Recherche no. 1304, Laboratoire Bordelais de Recherche en Informatique, Sept. 1992.

[11] D. Sankoff and J. B. Kruskal, Time Warps, String Edits, And Macromolecules: The Theory And Practice of Sequence Comparison. Addison-Wesley, 1983.

[12] N. J. A. Sloane, A Handbook of Integer Sequences, Academic Press, 1973.

[13] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. SIAM Journal on Computing, **6(2)**:323–350, 1977.