A COMPUTATIONALLY EASY INDEXING OF A LANGUAGE OF WHILE PROGRAMS

Andrew Marshall

A Thesis
Submitted to the Graduate College of Bowling Green
State University in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

May 2008

Committee:
Larry Dunning, Advisor
Ron Lancaster
Ray Kresman
ABSTRACT

Larry Dunning, Advisor

The concept of an effective enumeration of all possible computer programs is a foundation of computability theory. To achieve an effective enumeration an indexing is applied to order the elements, or programs, of the model language. This thesis demonstrates a practical and computationally easy indexing of a model of computation often used in introductory computability courses, the language of while programs.
ACKNOWLEDGMENTS

I would like to thank Dr. Larry Dunning for the topic idea and for the many hours of advice and assistance. This would not have been possible without his help. A special thanks to Jennifer for all her support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION AND THE LANGUAGE OF WHILE PROGRAMS</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>The Language of While Programs</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>WHILE LANGUAGE MODIFICATIONS</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>New Program Structure</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Variable Ordering</td>
<td>3</td>
</tr>
<tr>
<td>2.3</td>
<td>Branching Structures</td>
<td>4</td>
</tr>
<tr>
<td>2.4</td>
<td>Program Structure</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>PROGRAM INDEXING</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Program Index to Program</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Program to Program Index</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>RESULTS</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Results</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Concluding Remarks</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>APPENDIX A. PROGRAMS</td>
<td>18</td>
</tr>
<tr>
<td>Figure/Graph</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>An example while program</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Two possible while loops.</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Branching structure representation.</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>Central Binomial Coefficients in Pascal's triangle.</td>
<td>4</td>
</tr>
<tr>
<td>2.4</td>
<td>Two possible interpretations of the binary string 1010.</td>
<td>5</td>
</tr>
<tr>
<td>2.5</td>
<td>New While Language statements</td>
<td>5</td>
</tr>
<tr>
<td>2.6</td>
<td>Macro statement for $X_i := X_i + 1$.</td>
<td>6</td>
</tr>
<tr>
<td>2.7</td>
<td>Method of assigning binary representations to the While programming statements</td>
<td>6</td>
</tr>
<tr>
<td>2.8</td>
<td>Example enumeration for programs with 2 variable references</td>
<td>7</td>
</tr>
<tr>
<td>2.9</td>
<td>First 4 templates using the balanced binary string enumeration</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of finding the number of variable references if the initial program index is 12.</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of finding the template index with a program index of 12.</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>A portion of Pascal's triangle and some possible moves that can be taken.</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>Example of computing the template from Pascal's triangle for a program with index 12.</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>Example of computing the variable ordering for a variable index of 5.</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>Example of computing the template index from the binary string of the template</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Growth of Program index to Program</td>
<td>14</td>
</tr>
<tr>
<td>4.2</td>
<td>Growth of Program to Program index</td>
<td>15</td>
</tr>
</tbody>
</table>
Chapter 1
INTRODUCTION AND THE LANGUAGE OF WHILE PROGRAMS

1.1 Introduction

Frequently in computability theory, an indexing is applied to all objects in a model language, for example all possible programs written in some programming language. The objects can then be effectively enumerated, and statements can be made about the properties of these objects. Rarely is an examination made on the complexity of computing these indexes. The demonstration that an indexing is possible is usually all that is required.

The present work supplements these other indexing by demonstrating that an index can be applied and used with linear complexity in practice and no more than quadratic complexity in theory. The method presented uses a model of computation that is often used in an introductory course on computability, the language of while programs. Beginning with the While Language of Kfoury et al [2] as described in Dunning and Lancaster [1], the method first transforms that language, and demonstrates how the changes enable a near linear algorithm for computing a while program index and for computing the while program from an index. The method can then be used, in an introductory course to computability to demonstrate effective enumerations. In addition, actual implementations of the algorithm are given.

1.2 The Language of While Programs

The language of while programs as given by Kfoury et al [2] is equivalent in computational power to the Turing machine. It allows any possible program to be written using a finite number of variables and four statement types. Each variable can hold only a single nonnegative integer of arbitrary size. The syntax of the statements as given in Dunning and Lancaster [1] is

1. Assignment statements:
   
   (a) $X_i := 0$
   
   (b) $X_i := X_j + 1$
   
   (c) $X_i := X_j - 1$

2. While Statement:

   (a) $\text{While}(X_i <> X_j)do \{ \text{some statement(s)} \}$
where $X_i$ and $X_j$ are variables.

While programs are then defined as one or more statements. It was for this language of while programs that indexing functions were initially sought, see Kfoury et al [2]. A method for enumerating this language is given in Dunning and Lancaster [1] where the authors develop the idea of enumeration based on program templates. While program templates are compound statements but with no variable references. A single variable placeholder, $X$, is used to indicate the location of a variable reference. The template’s statements can then be assigned variables by some variable ordering rule, (see figure 1.1). A variable ordering rule is a method for assigning variables to the variable placeholders in a program. The variable ordering rule restricts the assignment of variables in such a way that only a finite number of orderings are possible, see Dunning and Lancaster [1]. The variable ordering rule along with a method for enumerating the templates, then yields a method for effectively enumerating the while programs.

Figure: 1.1 An example while program (1), its corresponding template (2), and its variable ordering (3).
Chapter 2
WHILE LANGUAGE MODIFICATIONS

2.1 New Program Structure

Three changes were made to the language of while programs as presented by Kfoury et al [2]:

1. The assignment of variable names is restricted according to a “factorial variable ordering principle” which replaces the variable ordering principle given in Dunning and Lancaster [1].

2. The branching structure of the language of while programs was expanded to allow a “do ... while” as well as the “while ... do” construction.

3. The assign and increment statement was replaced by a simple increment statement.

Several other changes to the variable ordering principle and the branching structure of the While Language were investigated. The changes presented below were found to be the best choices computationally, all others resulting in $O(n^2)$ or greater complexity.

2.2 Variable Ordering

One of the difficulties in reducing the complexity of the index is that the variable orderings of Dunning and Lancaster [1] rely on the Bell [3] numbers, and in doing so, at best must compute a two-dimensional table. Other naive approaches require computation of $k^k$ for $k = 1, \ldots, n$ and are therefore still not efficient enough computationally. To overcome this difficulty a variable ordering principle based on the factorials is used. The factorial variable ordering allows variables to range from 1 to the number of the variable’s position in the variable sequence. That is:

**Definition 1.** Factorial Variable Ordering Principle: Given a variable sequence $x_{n1}x_{n2}...x_{nj}...x_{ni}$ variable $x_{nj}$ may be assigned from the range $1 \leq n_j \leq j$.

Using this variable ordering, computing the number of possible variable orderings for $n+1$ variable references, given the number for $n$ references, is equivalent to computing the $n+1$ factorial from the $n$th Factorial: $(n+1)! = n! \times (n+1)$.

2.3 Branching Structures

The While Language defined in Dunning and Lancaster [1] required that a while statement precede
any closing brace, and not be empty. We have removed these two requirements. This allows braces to
precede while statements, and both branching structures to be empty (see figure 2.1). There is also no
distinction made between closing and opening braces, until they are matched with their corresponding while
statement. This method allows for the representation of the branching structures in the following way: Let

1. \( X_2 := 0 \)
   \[
   \begin{cases}
   X_1 := 0 \\
   \end{cases}
   \]
   \( \text{while} (X_1 \neq X_2) \}

2. \( X_2 := 0 \)
   \[
   \begin{cases}
   X_1 := 0 \\
   \end{cases}
   \]
   \( \text{while} (X_1 \neq X_2) \}

Figure 2.1: Two possible while loops. Loop (1) is a Do-While, Loop (2) a While-Do.

0 represent a brace either \“\{\” or \“\}\” and let 1 represent a while statement either \“while\{\” or \“\}while.\” The
branching structures can then be represented as binary strings with a balanced number of ones and zeros,
as shown in figure 2.2. This allows for the counting of the number of possible loops by using the Central
Binomial Coefficients, computed by \( \binom{2n}{n} \). The Central Binomial Coefficients can also be computed using the
central column of Pascal’s triangle, as shown in figure 2.3.

1. 1100 Denotes \( while\{while\{} \)

2. 1010 Denotes \( while\{\}while\{} \)

3. 1001 Denotes \( while\{\}{}while\)

4. 0110 Denotes \( \{while\}while\{} \)

5. 0101 Denotes \( \{while\}{}while\)

6. 0011 Denotes \( \{\}{}while\) \}

Figure 2.2: Branching structure representation.
Figure 2.3: Central Binomial Coefficients in Pascal’s triangle.

Using \( \binom{2n}{n} \), computing the \( n+1 \) coefficient from the \( n \)th coefficient can be accomplished via the factorials, or Pascal’s triangle as above.

Before this new looping structure can be accepted, the issue of ambiguity must be removed. The binary representation 1010 is a valid representation yet it has two possible interpretations, as shown in figure 2.4.

\[
\begin{align*}
1. \quad \text{while}(X)\{ \\
\quad \text{while}(X)\{ \\
\quad \}
\}
\end{align*}
\]

\[
\begin{align*}
2. \quad \text{while}(X)\{ \\
\quad \}
\end{align*}
\]

Figure 2.4: Two possible interpretations of the binary string 1010.

To remove ambiguity, the convention that no nesting of differing types of loops are allowed, is adopted. That is, nesting do-while loops inside while loops is not allowed and vice-versa. This may appear rather restrictive but it is equivalent to scanning from the top of the program and matching while and brace statements to the first possible match. This ensures there is only one way of interpreting nested loops, removing the ambiguity. This is similar to, “If-Then-Else” matching in programs. Note that in adding do-while loops, the computational power of the language was not decreased (or increased).

### 2.4 Program Structure

One additional change was made to the While Language from Dunning and Lancaster [1]. This change, as will be demonstrated later, allows for the use of the Central Binomial Coefficient to calculate all possible
templates, rather than just the number of looping constructs as in section 2.3. To use the Central Binomial Coefficients in this way, the resulting program templates need to be represented by binary strings of balanced ones and zeros. This is accomplished by changing the statements in the following manner, so that each statement references a single variable:

<table>
<thead>
<tr>
<th>Old Statements</th>
<th>New Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_i := 0 )</td>
<td>( X_i := 0 )</td>
</tr>
<tr>
<td>( X_i := X_j + 1 )</td>
<td>( X_i := X_i + 1 )</td>
</tr>
<tr>
<td>( \text{while}(X_i \neq X_j) { )</td>
<td>( \text{while}(X_i) { )</td>
</tr>
<tr>
<td></td>
<td>( (X_j) }</td>
</tr>
</tbody>
</table>

Figure 2.5: New While Language statements.

The “set to zero” statement remains the same. The successor statement is restricted and is now equivalent to an incremental operator, like the “++” in the C++ language. The following macro can be used in place of the old increment statement. [ Note that the case \( i = j \) is special ]

\[
\begin{align*}
X_i & := 0 \\
\text{while}(X_i \neq X_j) \{ \\
X_i & := X_i + 1 \\
\} \\
X_i & := X_i + 1
\end{align*}
\]

Figure 2.6: Macro statement for \( X_i := X_j + 1 \).

In the case of the branching statements, one of the variable references is moved to the single brace with which no variable references were previously associated. The resulting new statements now each have a single variable reference and much like the loops in the previous section, can be represented using binary strings. Two bits are required to represent each statement and can be assigned using a binary tree.

Note that although the statements have changed, the computational power of the language has not, and the language is still easily understood. The fact that the language was not drastically changed to obtain the desired result is important. Because this model of computation is primarily used in introductory computability courses, the fact that it remains easily understandable is of value.

The order of enumeration of the binary strings is important and is now considered. The following enumeration method, differing from the method in Dunning and Lancaster [1], is used because it allows
the utilization of Pascal’s triangle. The importance of Pascal’s triangle will be demonstrated later. The enumeration is in reverse lexicographical order, always maintaining a balance of ones and zeros, figure 2.8 gives an example.

\[\begin{align*}
&1100 \\
&1010 \\
&1001 \\
&0110 \\
&0101 \\
&0011
\end{align*}\]

Figure 2.8: Example enumeration for programs with 2 variable references.

Although this changes the order of enumeration from Dunning and Lancaster [1], all templates are still included in the enumeration.

<table>
<thead>
<tr>
<th>Templates</th>
<th>Binary Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(X := X + 1)</td>
<td>10</td>
</tr>
<tr>
<td>(X := 0)</td>
<td>01</td>
</tr>
<tr>
<td>(while(X \neq){}(X))</td>
<td>1100</td>
</tr>
<tr>
<td>(X := X + 1)</td>
<td>1010</td>
</tr>
<tr>
<td>(X := X + 1)</td>
<td>1010</td>
</tr>
</tbody>
</table>

Figure 2.9: First 4 templates using the balanced binary string enumeration
Chapter 3

PROGRAM INDEXING

3.1 Program Index to Program

The indexing is now demonstrated, starting with a program index and producing the actual program. Later the reverse, program to index, will be demonstrated. Given a program index, several items must be extracted from that index to produce the actual program:

1. The number of variable references.

2. The program template.

3. The variable ordering.

Once these three items are obtained from the index, they can be combined to produce the actual program. The first step is to find the number of variable references. This is accomplished by computing the Central Binomial Coefficients, by the following formula \( \binom{2n}{n} = \frac{(2n)!}{n!} \). Start at \( n = 1 \), \( n \) is the number of possible variable references and increase \( n \) incrementally. The result is multiplied by the factorial of \( n \), where the factorial of \( n \) is the number of possible variable orderings for \( n \) variables. This continues until the index of the program is contained within the resulting product. The resulting \( n \) is the number of variable references, (see figure 3.1). This can also be viewed as moving down the center of Pascal’s triangle multiplying each element by the factorial of its position. The given number of templates for \( n \) variable references is \( \binom{2n}{n} \) as stated above. The number of programs for \( n \) variable references is \( \binom{2n}{n} \cdot n! = \frac{(2n)!}{n!} \), the number of templates multiplied by the number of variable orderings. At each \( n \), we will need a table of values from 1 to \( (2n)! \) to compute the number of programs. The method of memoization, storing previously computed values for later reuse, can be employed so that for each \( n \) only two additional factorials are computed. This can be done because the algorithm begins at 1 and increases \( n \) at each step. The results of the previous steps are stored and can be recalled for use in the next iteration without recomputing them. In general:

\[
\text{while} \left\{ \text{max} < \text{pnum} \right\} \\
\text{memoize} \rightarrow \text{fact}[i + 1], \text{fact}[i + 2] \\
\text{max} \leftarrow \frac{\text{fact}[2i]}{\text{fact}[i]} \\
i = i + 1 \\
\text{endwhile}
\]

This calculation also allows for the memoization [storage of results in a table to avoid recomputation] of
\[
I = 1
\]
\[
\text{fact}[2]/\text{fact}[1] = 2 \leq 12
\]
\[
\text{Memoize} \rightarrow \text{factorial3, and4}
\]
\[
I = 2
\]
\[
\text{fact}[4]/\text{fact}[2] = 12 = 12
\]
Number of variable ref = 2

Figure 3.1: Example of finding the number of variable references if the initial program index is 12.

the factorials, from 1 to \(2n\). This ensures that no remaining factorial computations are required because the remaining computations only require the factorials up to \(2n\). Note that we do not compute Pascal’s entire triangle, just the central column, thus giving a computation linear in terms of the number of arithmetic operations required. The above method also gives the number of programs that have one or fewer variable references; (2 in the above example). Using this as a “base,” subtract the base from the program index, divide by the factorial of \(n\), and the template index will be produced. The template index is the index of the program template in the enumeration of all templates with \(n\) variable references (see figure 3.2). In general \(T_{\text{index}} = \frac{\text{ProgNum} - \text{base}}{\text{NumVar!}}\). The second step is to produce the program template using the above information.

Program Index = 12
Base = 2
\[
12 - 2 = 10/\text{fact}[2] = 5
\]
Template Index = 5

Figure 3.2: Example of finding the template index with a program index of 12.

Constructing the template from the template index is done by moving back up Pascal’s triangle starting from the central column position last calculated in the previous section. Again the entire triangle is not needed; only two positions in the triangle are calculated at each step. The calculation begins at the current position \(\binom{n}{k}\), the stopping position in the previous section. The two positions directly above the current position are computed by selecting the factorial of \(n\) and the factorial of \(k\) and dividing. The starting \(n\) and \(k\) are calculated during the move down the triangle, in the previous section, and are adjusted at each step up the triangle. This is achieved using the rule:

Rule 1. Let \(\binom{n}{k}\) be the current position in Pascal’s triangle. If the next position selected moves closer to the central column, moving to balance the string, then \(n = n - 1\) and \(k = k - 1\). If the next position selected moves further away from the center, further unbalancing the string, then \(n = n - 1\) and \(k = k\).
To move back up the triangle, $2n$ moves are required, one for each binary position in the template representation. Each move also selects what binary digit will be placed in the current position. If a left move is selected, a one is placed in the binary string representing the template. If a right move is selected, a zero is placed in the string. Selection is based on the smallest value of the two computed positions containing the template index. The following figure, a portion of Pascal's triangle, demonstrates how $n$ and $k$ are computed by the Central Binomial Coefficients, and are reduced during the process of moving back up the triangle:

![Figure 3.3: A portion of Pascal’s triangle and some possible moves that can be taken moving back up the triangle.](image)

The previous change to the order of enumeration then enables the program template to be computed by the following method:

*Start with an empty string with $2n$ spaces ($n$ is the number of variable references), for a binary representation of the template. At each step up the triangle one position of the string is filled. At each step keep track of the balance of ones and zeros in the string. Movement proceeds by the following method:*

1. *Start at the center column. The string is balanced with no ones and zeros*
   
   (a) the positions directly above the current, denote these by left and right positions

   (b) *If index is $\leq$ the left position, move left, place a 1 in the binary string*

   (c) *If index is $> left$, move right, place a 0 in the binary string and set the index= index-left*

2. *Proceed this way until the top or a side of the triangle is reached.*

If a side of the triangle is reached before filling in the binary string, then there is only one way to fill in the remaining positions. The computation of the left and right positions can be stopped and the remaining ones
or zeros needed to balance the string can be inserted.

![Pascal's Triangle](image)

Figure 3.4: Example of computing the template from Pascal's triangle for a program with an index of 12.

Again because the entire triangle is not computed only $2n$ calculations are required. This process works because at each step it calculates the number of programs resulting from selecting a one or a zero, and then selecting the portion of these programs that contains the index, by selecting a one or zero. The enumeration ordering of templates provides the structure needed to decide, based on a selection of a one or a zero, where the index is contained.

The variable ordering is computed from its index, which is retrieved from the program index and the template index, in the following method:

**Variable Ordering**

1. $N$ to 2
2. Divide the index by $i$, the remainder +1 is the variable reference for position $i$.
3. The quotient is the new index.

Example: Index=5, Number Variables =3, XXX

- $i=3$, $5/3=1$, remainder=2, $2+1=3$, XXX
- $i=2$, $5/2=0$, remainder=1, $1+1=2$, X23
- $i=1$, always start with 1 123

Figure 3.5: Example of computing the variable ordering for a variable index of 5.
Once the variable ordering has been calculated, it can be combined with the template to produce the actual program.

### 3.2 Program to Program Index

Calculating the program index from the actual program uses the same methods as described in the previous section but in the reverse. Given a program the following items must be calculated to produce that program’s index.

1. The number of programs with one or fewer variable references than the given program
2. The template index, which is the position of the program index in the enumeration of all templates with $n$ variable references
3. The index of the variable ordering, which is the position of the variable ordering in the enumeration of all variable orderings with $n$ variable references

First a scan of the program is done and the binary string representing the template, the number of variable references $n$, and the variable ordering, are extracted. The program base (the number of programs that have from one to $n - 1$ variable references, with $n$ being the number of variable references of the given program) is then calculated. This is done in a method, similar to calculating the number of variable references in the previous section, by moving down the triangle. This again allows for the memoizing of the factorials. The difference between this method and the method described in the previous section is that now the number of variables (the number of steps down the triangle) is already known. Given the program’s template’s binary string representation, retrieved from the program, and starting at $(0, 0)$, the top of Pascal’s triangle, calculating the template index is done as follows:

*Calculate the two positions below the current position; denote these as the left and right positions. Start at the top of the triangle and move according to the following rules:*

1. *If there is a 1 in the current position of the template string move down and right.*
2. *If there is a 0 in the current position of the template string move down and left and add (left position – current position) to the program index.*

This is done for each position in the template string. Once the end of the template string is reached, the template index is computed, (see figure 3.6).
Computing the variable ordering index is also done in reverse of the previous method (of section 3.1). We reconstruct the quotient from the division done in that section and add it to the remainder to compute the variable index:

*The quotient is initially set to zero*

1. The remainder is retrieved from the given variable ordering.

2. The quotient is recomputed by multiplying the current quotient by the current position plus 1 and adding the remainder.

This is repeated starting at the first and proceeding to the last variable in the variable ordering, n times. The final quotient is the variable ordering index.

After the template and variable ordering indexes are computed, they are combined giving the final program index.
Chapter 4
RESULTS
4.1 Results

The following two graphs show the growth of the number of iterations for computing the program from its index and the index from its program, as the length of the input grows. The quantity “iterations” includes the number of times through each iterative routine in the algorithms as determined empirically from instrumented versions of the programs.

Graph 4.1: Growth of Program index to Program
As demonstrated by the above graphs, the Pascal triangle based solution, in respect to the number of operations, results in a linear complexity growth proportional to the input. It is important to note that the number of entries of the triangle actually computed is linear with respect to $n$. The algorithm itself performs $O(n)$ computations. These observations alone are not enough to conclude that the indexing is linear with respect to the length of the input. To be able to conclude mathematically that the entire indexing is linear, we must show how each part of the index grows in relation to the other. That is, we must compare the growth of the length of a program with the growth of its index. If we use the following to estimate the lengths of input, and Stirling approximation for the factorials we compute the relationship as follows:

1. Length of a program for $n$ variables $\approx n \log(n)$
2. Length of a program index $\approx \log\left(\binom{2n}{n} n!\right) = \log\left(\frac{(2n)!}{n!}\right) = \log((2n)! - \log(n!)$

Using the Stirling approximation for $n!$ in formula 2.

$log((2n)! - log(n!)) \approx log(\sqrt{2\pi n}(\frac{2n}{e})^{2n}) - log(\sqrt{2\pi n}(\frac{n}{e})^{n}) \approx$

$log(\sqrt{2\pi n} + 2n \log(\frac{2n}{e}) - log(\sqrt{2\pi n}) - n \log(\frac{n}{e}) \approx$

$O(\log(2) + 2n \log(2n) - log(\sqrt{2}) - n \log(n)) \approx$

$O(2n + n \log(n)) \approx$
\( O(n \log(n)) \)

From this approximation, the input to each algorithm grows at the same order, giving a linear relationship. However we still may not conclude this is in a linear algorithm, although according to the number of iterative steps taken the algorithm’s growth is linear with respect to the input. Because a product is computed at each step we must consider the complexity of multiplication. Division is also computed in an iterative statement of the algorithm. Knuth [5], proved that division is comparable to multiplication, except for a constant factor, so we only consider multiplication. The Furer [4] algorithm with complexity of \( n \log n \, 2^{O(\log^* n)} \), where \( \log^* n \) is the iterated logarithm, for two \( n \) bit numbers, is used as the upper bound for multiplication. We must consider the largest of these multiplications being performed. In other words, we want to determine the largest integer used in a multiplication, and this is \( n! \). The length or number of digits in \( n! \) is of order \( n \log(n) \). Given this upper bound on the data length and \( O(n) \), operations the resulting complexity is:

\[
(n \log(n)) \log(n \log(n)) 2^{O(\log^*(n \log(n)))}
\]

A stronger analysis: Let \( M(n) \) be the complexity of multiplying two \( n \)-bit numbers and \( M(m, n) \) be the complexity of multiplying an \( m \)-bit number by a \( n \)-bit number. We then have \( O(n) \) computations of order \( M(\log(i), i \log(i)) \). Regard the computation of multiplying a \( \log(i) \) bit number times an \( i \log(i) \) bit number as simply being a multiplication of a 1-digit number by an \( i \)-digit number in radix \( i = 2^{\log(i)} \). That is, perform the multiplication in groups of \( \log(i) \) bits. The computation now requires \( O(n) \) computations of order \( M(\log(i)) \). We can conclude from this, that in the best case \( M(n) = n \), but that it has an upper bound of \( M(n) = n \log(n) \, 2^{O(\log^* n)} \), using the Furer [4] results. Then with \( O(n) \) computations the algorithm has a \( n^{2 \log(n)} \) complexity in the best case and a \( n^{2 \log(n) \log(\log(n))} \, 2^{O(\log^*(\log(n)))} \) upper bound. Expressing the complexity in terms of the length of the data \( d \), with \( d = n \log(n) \), and using the above upper bound, the resulting complexity is \( O(d^2) \).

### 4.2 Concluding Remarks

This thesis demonstrates a method for applying an indexing to the While Language with linear complexity in practice and no more then quadratic complexity in theory. This method also removes the requirement to use “empty” functions to achieve an effective enumeration (see Kfoury et al [2]). Several approaches may be available to improve the complexity further. One approach could be to improve the algorithm presented here. This could possibly be done by investigating methods of fast factorial computation, in combination with methods of computing the binary template representation, i.e. traversing Pascal’s triangle, without multiplication. The second and probable the more fruitful approach would be to look for another encoding, with possible modifications to the While Language, which would lend itself to faster encoding-decoding.
REFERENCES


APPENDIX A:
Program Listings

Listing 1. Program index to Program:

```c
#define MAX 1000 //max number of statements allowed
//prints the var
#define wtvar() printf("%d", vars[islot++])
int build(int nvar, unsigned long long int tnum, int none, int nzero);
void printtemplate();
//variable odering, and template odering
int vars[MAX]; bool bslots[MAX];
int varcount; bool programs;
//factorials and central binomial coefficients
unsigned long long int fact[MAX]; int maxfact;

int main(int argc, char *argv[])
{ //Program Number and number of variables
    unsigned long long int pnum=0; int nvar;
    if(argc < 2) { cin>>pnum; } else { pnum=atoi(argv[1]); }
    //initialize Factorials
    maxfact=2; fact[0]=1; fact[1]=1; fact[2]=2;
    //find number of var ref and memoize factorials, binomial coefficients
    int tvar=1; int cb=2; bool end=false;
    unsigned long long int base=0; unsigned long long int z=0;
    unsigned long long int ftotal=2; unsigned long long int prevbase;
    while(!end)
    { z=(fact[tvar+tvar]/(fact[tvar])); //max # of programs for tvar # of vars
        if(pnum <= z){ end=true; break;}
        else { prevbase=base; base+=z;
            //memoize the factorials
            fact[cb+1]=ftotal*(cb+1); ftotal=fact[cb+1];
            fact[cb+2]=ftotal*(cb+2); ftotal=fact[cb+2];
            cb+=2; tvar++;
            //memoize the central binomial coefficients
        } }
    nvar=tvar; unsigned long long int pblock=pnum-base;
    if(pblock==0){ base=prevbase; nvar--;}
    pblock=pnum-base; varcount=nvar; int tempindex=0;
    //find the template number
    tempindex=pblock/fact[nvar]; if((pblock%(fact[nvar]))>0) tempindex++;
    //build template encoding
    build(nvar, tempindex, 0, 0);
    //find the index into the list of programs
    unsigned long long int iv=(pnum-base)-((tempindex-1)*fact[nvar]);
    //compute the variable ordering, use index into programs
    vars[0]=1; iv--;
    if(iv==0)
    for(int i=1; i<nvar; i++) { vars[i]=1; }
    else
    for(int i=nvar; i>1; i--)
    { int x=(int)iv/i; int y= x*i; int z=iv-y;
        vars[i-1]=z+1; iv=x; }programs=true; printtemplate(); return 0;
} //end of main

//Traverse up pascals triangle. Start at c_bin[nvar] Move up, and left or
//right, by computing the right and left side. if templ# < right move left
//templ# stays the same, else move right templ#--left.
//fillin templ string left=1 right=0 as each move is selected
int build(int nvar, unsigned long long int tnum, int none, int nzero)
{ }
```

int n=2*nvar; int b=nvar; bool fillin=false;
unsigned long long int ls=0; unsigned long long int rs=0;
//loop through each binary spot
for(int i=0; i<2*nvar; i++)
{ if(fillin)//Reached edge of triangle can fill in rest
    { if(none>nzero)
        { bslots[i]=0; nzero++; }
    else
        { bslots[i]=1; none++; } //end if
    } //end if
else{ //compute left side and right side based on #1s, #0s
    if(none >= nzero)
        { ls=(fact[n-1]/(fact[b]*fact[(n-1)-b]));
           rs=(fact[n-1]/(fact[b-1]*fact[(n-1)-(b-1)])); } 
    else
        { ls=(fact[n-1]/(fact[b-1]*fact[(n-1)-(b-1)]));
           rs=(fact[n-1]/(fact[b]*fact[(n-1)-b])); } 
    if(tnum > ls)//templ# > left side move right
        { if(nzero>none) {n--; } //Unbalancing string
           else {n--; b--;}//balancing string
           tnum=tnum-ls; bslots[i]=0; nzero++;
           if(rs==1) fillin=true; } 
    else
        { if(none>=nzero) {n--;} //Unbalancing string
           else {n--; b--;}//balancing string
           bslots[i]=1; none++;
           if(ls==1) fillin=true; } 
} /* end else */ }//end for loop
} //end build

//prints the template
void printtemplate()
{ int mar=0; int islot=0; int bincount=0;
for(int i=0; i<varcount; i++)
{ int pone=bslots[bincount]; int ptwo=bslots[bincount+1]; bincount+=2;
    if(pone==0 && ptwo==1){cout<<"z ";wtvar();cout<<"\n"; } 
    else
        if(pone==1 && ptwo==0){cout<<"s ";wtvar();cout<<"\n";}
    else
        if(pone==1 && ptwo==1){cout<<"w ";wtvar();cout<<"\n"; }
    else{cout<<"b ";wtvar();cout<<"\n"; }
} //end for loop
cout<<"#"; cout<<endl; }
Listing 2. Program to Program Index:

```
#define MAX 1000 //max number of variable references
//variable references, template references and factorials
int vars[MAX]; bool templ[MAX]; unsigned long long int fact[MAX];

int main(int argc, char *argv[])
{ int vc=0; int tc=0; int i=0;
  int ic=0; int cc=0; bool inputdone=false;
  //parse input
  string input; int argnum=1;
  if (argc >= 2){input.assign(argv[argnum]); argnum++;}
  else {getline(cin, input, '\n');}
  //Assign input to template and variables
  while(!inputdone)
  { int lc=0;
    for(int i=0; i<input.size(); i++)
    { switch(input[i]){ case ' ': break;
    case '#': inputdone=true; break;
    case 'z': templ[tc]=0; templ[tc+1]=1; tc+=2; cc++; break; //x:=0
    case 's': templ[tc]=1; templ[tc+1]=0; tc+=2; cc++; break; //x:=x+1
    case 'w': templ[tc]=1; templ[tc+1]=1; tc+=2; cc++; break; //while()do{
    case 'b': templ[tc]=0; templ[tc+1]=0; tc+=2; cc++; break; //}(x)
    } //end switch
    if((input[i] >= '0') && (input[i] <= '9')){
      char cp; ostringstream s1; cp=input[i];
      while((cp >= '0')&&(cp <= '9')){s1<<cp; i++; cp=input[i];}//endwhile
      string s2 = s1.str();
      vars[vc]=atoi(s2.c_str()); vc++; ic++; i--;
      if(input[i]=='#'){inputdone=true; i+=input.size(); break;}}/*end if*/
    } //end for
    if(inputdone==false){
      if (argc >= 2){ input.assign(argv[argnum]); argnum++; }
      else { getline(cin, input, '\n'); } //end if
  } //end while

  //Initialize the factorials
  fact[0]=1; fact[1]=1;
  //for finding the program base
  unsigned long long int pbase=0; int fc=1;
  unsigned long long int oldpbase=0;
  //find program base and memoize the factorials
  for(int i=1; i<=vc; i++)
  { fact[fc+1]=(fc+1)*t; t=fact[fc+1];
    fact[fc+2]=(fc+2)*t; t=fact[fc+2];
    fc+=2; //compute 2 fact at a time for fact[2*i] in base calc
    oldpbase=pbase;
    //calc the base= #templates*#variable ref
    pbase+=(fact[2*i]/fact[i]);
  } //end for loop
  pbase=oldpbase; //get prev template from the current template

  //for computing template number
  int top=0; int bot=0; int numone=0;
  int numero=0; unsigned long long int current=1;
  unsigned long long int max=1; unsigned long long int LS=0;
  unsigned long long int RS=0; unsigned long long int tindex=1;
  //build the template number using pascals triangle, go down and left
  //or down and right based on a 1 or 0 in the current pos of the string.
  //if 0 add the left side to the template index.
  for(int i=tc-1; i>=0; i--)
  } //end while
```
{ if(numone==numzero){ //current string is balanced
  LS=fact[top+1]/(fact[top+1-(bot+1)]*fact[bot+1]);
  RS=LS; int temp=LS-current;
  if(temp[i]==0){if(temp>0)tindex+=temp; //add to templ index
    top++; bot++; current=LS; numzero++;}
  else{top++; bot++; numone++; current=RS;}
} //end numone==numzero
else if(numone>numzero){ //more 1s than 0s in current string
  LS=fact[top+1]/(fact[top+1-(bot)]*fact[bot]);
  RS=fact[top+1]/(fact[top+1-(bot+1)]*fact[bot+1]);
  int temp=LS-current;
  if(temp[i]==0){if(temp>0)tindex+=temp; //add to templ index
    top++; bot++; current=LS; numzero++;}
  else{top++; bot++; numone++; current=RS;}
} //end numone>numzero
else if(numone<numzero){ //more 0s than 1s in current string
  LS=fact[top+1]/(fact[top+1-(bot+1)]*fact[bot+1]);
  RS=fact[top+1]/(fact[top+1-(bot)]*fact[bot]);
  int temp=LS-current;
  if(temp[i]==0){if(temp>0)tindex+=temp; //add to templ index
    top++; bot++; current=LS; numzero++;}
  else{top++; numone++; current=RS;}
} //end if numone<numzero
} //end for loop

//for the variable ordering
int q=0; int tempindexin=0; int rem=0;
//compute the variable ordering number
for(int i=0; i<vc; i++)
  { rem=vars[i]-1; q=tempindexin; tempindexin=(q*(i+1))+rem; }//end for

//final program number
unsigned long long int pnum=pbase+tempindexin+1;
cout<<pnum<<endl; return 0; }