Generating Some Restricted Classes of Permutations

by

Scott Arno Lausch
B Sc , Laurentian University, 1996

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr Frank Ruskey, Supervisor (Department of Computer Science)

[redacted]

Dr John Ellis, Departmental Member (Department of Computer Science)

[redacted]

Dr Dominique Roelants, Departmental Member (Department of Computer Science)

[redacted]

Dr Rick Brewster, External Examiner (Department of Mathematics, Capilano College)

Supervisor: Dr. Frank Ruskey

ABSTRACT

We consider the generation of two categories of permutations. The first category consists of stamp foldings and related objects. A stamp folding is a permutation representing the folding of a linear strip of stamps into a pile. The second category consists of Dumont permutations of both the first and second kind, which are defined by rules based on the parity of the indices and elements.

Algorithms which are asymptotically efficient were developed for all objects considered. Each algorithm used the technique of recursive backtracking. Also, a possible one-to-one mapping between the two types of Dumont permutations was conjectured.

Examiners:

[redacted]

Dr. Frank Ruskey, Supervisor (Department of Computer Science)

[redacted]

Dr. John Ellis, Departmental Member (Department of Computer Science)

[redacted]

Dr. Dominique Roelants, Departmental Member (Department of Computer Science)

[redacted]

Dr. Rick Brewster, External Examiner (Department of Mathematics, Capilano College)

Contents

List of Figures

1 Introduction

This thesis is concerned with the generation of certain types of restricted permutations. That is, all the objects being considered for generation are subsets of the permutations of the numbers from 1 through n inclusive, for some n.

We will be concerned with the efficiency of any algorithms developed. In general, algorithms to generate combinatorial objects such as permutations are considered to be intractable. This is due to the exponential increase in number of objects with increase in size of input.

Since the classical theory of computational complexity only deals with the difference between polynomial and nonpolynomial time, the class reductions from that theory will be of little use to us. All the generation algorithms considered here will necessarily involve nonpolynomial time due to their nature.

We will, however, be concerned that the algorithms involved are as efficient as possible (from a time-complexity standpoint). What we want for any given problem is to have an algorithm that runs in Constant Amortized Time, meaning that the amount of processing time is directly proportional to the number of objects that are listed. The term *Constant Amortized Time* (CAT for short) will be used throughout this thesis. We will state that an algorithm is CAT (see [17]) if it runs in Constant Amortized Time. Others refer to this behaviour as linear (see [16]). What is not counted in our consideration is the time to actually process the objects by outputting

them or otherwise. Instead, the important thing to count is the amount of data structure change needed to generate the objects. It should be noted that none of the algorithms developed in this thesis run in constant time.

Most of the algorithms in this thesis are generated using a *backtracking* approach, which is fairly common in combinatorial generation. Backtracking involves the building of an object by first considering the possible placements of one element. For each possible placement made, the algorithm recursively does the same with the remaining elements and positions.

For example, a backtracking algorithm to generate all permutations of the numbers 1 through n could work as follows. Choose which number will be placed in the first unused position. Recursively run this algorithm on the remaining list of numbers and empty positions. Choose the next number to put in this first unused position and run the algorithm recursively again. Choose the next number to put in this first unused position and run the algorithm recursively again. Make these recursive calls for each possible number that can be placed in the first unused position.

We will be concerned that such algorithms are CAT. This will be the case whenever there are no *dead ends* and no *outdegree one nodes* in the computation tree. A computation tree is a representation of the recursive subroutine calls made by an algorithm. A dead end occurs when a recursive call is made and no valid data structure changes can be made (i.e. no new valid object is produced from that subroutine call). In that event, the algorithm must back up to the previous level of recursion.

and continue from there. A dead end can also be thought of as a node in the computation tree with outdegree zero. We call an algorithm BEST (meaning Backtracking Ensuring Success at Terminals, as given in [17]) when there are no dead ends. An outdegree one node appears in the computation tree when only one recursive call is made from a particular instance of the subroutine. This slows the running time of the algorithm significantly if the number of outdegree one nodes is larger than an amount proportional to the number of objects being generated. That is, if the number of outdegree one nodes is not O(number of objects). We can be sure that an algorithm is CAT if there are no dead ends or outdegree one nodes in the computation tree because this implies that each level of this tree (beginning at the leaves) is at least twice as large as the level above it. It is trivial to see that there can be at most 2ℓ nodes in such a tree if there are ℓ leaf nodes.

In this thesis, we loosen this restriction of the number of outdegree one nodes. We require the number of these nodes in the computation tree to be bounded by $c\ell$, where c is a constant. This change gives us 3 sufficiency criteria for declaring an algorithm CAT: constant time is used per recursive function call, there are no dead ends in the computation tree, and the number of outdegree one nodes in the computation tree is bounded by $c\ell$, where c is a constant and ℓ is the number of leaf nodes in the computation tree.

2 The Generation of Permutations

A great number of algorithms have been developed for generating unrestricted permutations (see [18]) Many of these algorithms generate permutations in a lexicographic (dictionary) order Some start with the first element and use a Next element function to move to the next element, while others use a recursive backtracking algorithm

Since most of the algorithms developed in this thesis are of a recursive nature, it will be useful to consider a common recursive permutation generation algorithm By considering the Steinhaus-Johnson-Trotter (SJT) algorithm, we will illustrate a backtracking algorithm which is CAT This will be easily seen by considering that there are no dead ends or paths in its computation tree

The idea of the SJT algorithm is to consider each permutation of size $n-1$ ($n-1$-*permutation*) in turn One places the element n in each possible position within a given $n-1$-permutation starting from the beginning and moving to the end, or vice versa When one deals with the next $n-1$-permutation, one places the element n on the same extreme (beginning or end) as the previous n-permutation left at If the $n-1$-permutations one is using have been constructed in this manner as well, all n-permutations will be generated in such a way that there will only be two changed positions from one permutation to the next - a swap

The Steinhaus-Johnson-Trotter (SJT) algorithm is shown in Figure 1 The Move subroutine referred to in the SJT algorithm is given in Figure 2

```
Perm (n)
    local i
    if n > N then
        PrintIt
    else
        Perm(n + 1)
        for i from 1 to n − 1
            Move(n, dir[n])
            Perm(n + 1)
        end for
        dir[n] ← −dir[n]
    end if
```

Figure 1 Pseudocode for SJT algorithm

```
Move (x, d)
    local j
    j ← π^{-1}[x]
    π[j] ← π[j + d]
    π[j + d] ← x
    π^{-1}[x] ← j + d
    π^{-1}[π[j]] ← j
```

Figure 2 Move subroutine for SJT algorithm

12 21

123 132 312 321 231 213

1234 1243 1423 4123 4132 1432 1342 1324 3124 3142 3412 4312
4321 3421 3241 3214 2314 2341 2431 4231 4213 2413 2143 2134

Figure 3 Results of SJT algorithm for $N = 2, N = 3$, and $N = 4$

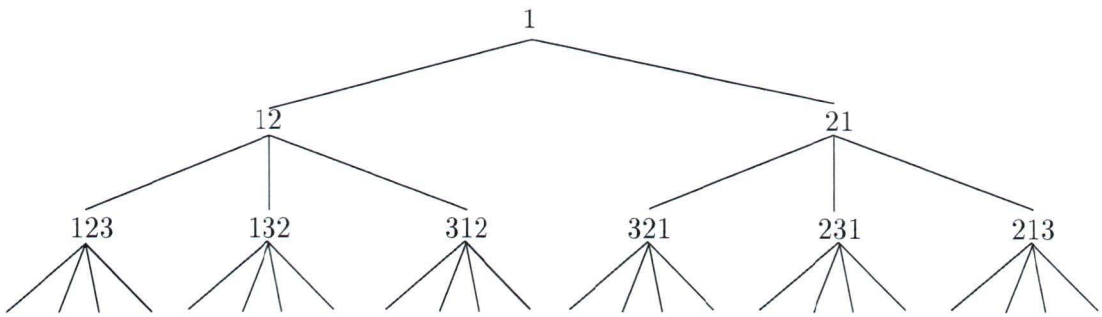

Figure 4 Computation tree for the SJT algorithm where $N = 4$ The leaf nodes are omitted due to space

Before calling Perm, the permutation is initialized by setting $\pi = \pi^{-1} = \iota$, where π represents the permutation and ι represents the identity function Also, the direction is set (-1 for left and 1 for right) for each element in the permutation by setting $dir[i] = -1$ for $i \in 1, 2, \ldots, N$ To use the algorithm, one calls Perm(1) To illustrate the SJT algorithm, Figure 3 shows the results for $N = 2, 3$, and 4 respectively

It is easily observed how the pattern of the previous size result is contained within the result of a given size For example, the results of size 4 contain each of the results (123, 132, 312, etc) in the same order as in size 3 with the element 4 taking on all the possible positions with each ordering of [3]

A computation tree for this algorithm for $N = 4$ is given in Figure 4. This will assist us in determining that the algorithm is CAT. Three facts will suffice to show this. First, we must determine that only a constant amount of computation (other than subroutine calls) is done at each node. This is seen to be true by looking at the algorithm in Figures 1 and 2. Secondly, we must know that there are no dead ends. Thirdly, we must know that the number of outdegree one nodes is bounded by $O(\ell)$, where ℓ is the number of leaves in the tree. These latter two items hold by the following argument. Each level of the computation tree lists all the permutations of a particular length, say n. Each of these n-permutations has $n + 1$ permuations of length $n + 1$ which are built from it. Thus, there is a branching factor greater than 1 at each level of the computation tree, making this a BEST algorithm. We may also conclude that the SJT algorithm is CAT. These 3 criteria will be shown to be satisfied by each of the generation algorithms developed in this thesis, proving that each is CAT.

3 Stamp Foldings and Related Objects

There are a number of variations of stamp foldings as well as some objects related to them such as meanders and closed meanders. We will be concerned with the development of CAT algorithms for many of these objects.

3.1 Definitions and Previous Results for Stamp Foldings

In mathematical terms, a *stamp folding* of length n, also known as an *n-fold*, is a permutation of the numbers from 1 through n with the added restriction related to the following physical model.

Consider a linear strip of n stamps numbered 1 through n, with perforations between stamp k and $k+1$. We distinguish between the top and bottom faces and also between the right and left sides of each stamp. Such a strip of stamps may be folded in a variety of ways along the perforations to create a pile of stamps the size of one stamp. We assume that the perforations are completely elastic so the distance of stretching is not limited.

A stamp folding is represented by a permutation in the following way. Take the above physical stamp folding and orient it so that the stamp labelled 1 is facing upwards in the pile and is oriented correctly with respect to right and left sides. (Whether this stamp labelled 1 is actually on the top of the pile is irrelevant.) Read the labels of the stamps from the top of the pile through to the bottom. This permu-

tation represents a stamp folding mathematically. We will refer to such permutations of length n as $S(n)$.

In the diagrams shown, the pile of stamps is drawn rotated 90 degrees, so that the top stamp is the leftmost vertical line and the bottom stamp is the rightmost vertical line. The perforations between stamps are represented by horizontal lines. Figure 5 shows all of $S(4)$, all of the 4-folds. In our visual representation, we also use the convention that the perforation from the stamp labelled 1 to the stamp labelled 2 is drawn towards the top of the diagram.

Given a valid permutation, it is easy to reconstruct the folding. However, given a folding, extra information might be needed to reconstruct the permutation. If n is odd, no extra information will be needed. On the other hand, if n is even, we would need to determine which end is which (i.e. which vertical line having only one connection to a horizontal line is the stamp labelled 1 and which is the stamp labelled n.) We can see this problem in the first two stamp foldings on the top row of Figure 5. These diagrams are identical, yet the permutations which correspond with them are different.

There are several simple and interesting properties of stamp foldings, two of which are:

(A) The line cannot cross itself. For this would mean that the top and bottom edges of the strip touch at the point of intersection. In other words, the strip would

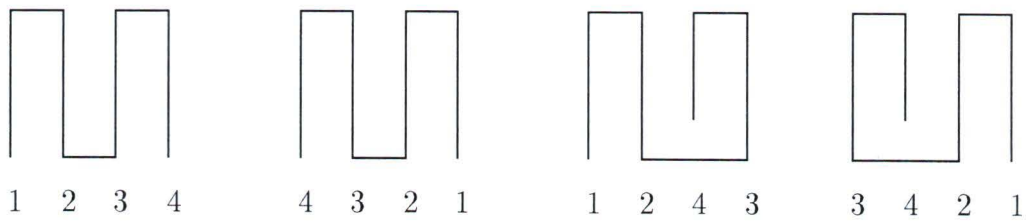

Figure 5 The 16 stamp foldings of $S(4)$

cross itself, which is physically impossible without the strip breaking

(B) The horizontal segments representing the perforations between the odd numbered stamps and their immediate successors lie above the vertical segments, whereas the segments representing the perforations between the even numbered stamps and their successors lie below the vertical segments. This property follows by induction on n.

For example, consider the permutation 1324. No matter which way one attempts to draw the corresponding stamp-folding diagram for this permutation, the line will cross itself or property (B) above will be violated. This permutation fits the characterization given in the theorem following, since in this case $k = 1$ and $r = 3$.

Koehler [8] describes stamp foldings and gives an algorithm for counting how many there are of each size. A forbidden subsequence characterization of stamp foldings is given by the following Theorem of Koehler.

Theorem 1 (Koehler) *An ordering of the numbers* $1, 2, \ldots, n$ *represents a stamp folding of length* n *if and only if*

No numbers appear in the order $\ldots k \ldots r \ldots k+1 \ldots r+1$, *or any of the circular permutations of these numbers, where* $1 \leq k, r \leq n-1, k \neq r, k \equiv r (\mathrm{mod} 2)$

Proof. Necessity is shown as follows. Property (B) above implies that k and r are both even or both odd. There are eight such situations for a given k and r.

$$\begin{array}{cccc} k, & r, & k+1, & r+1 \\ r, & k+1, & r+1, & k \\ k+1, & r+1, & k, & r \\ r+1, & k, & r, & k+1 \end{array}$$

and four more where the roles of k and r are interchanged. These cover exactly the conditions given. Given this fact from Property (B), Property (A) above is violated if and only if between some stamp k and its successor $k+1$ there is either a stamp r whose successor $r+1$ lies outside the pair k $k+1$ in the ordering, or a stamp $r+1$ whose predecessor r lies outside the pair.

Sufficiency will be shown by induction on n. As a basis, the condition is obviously true for $n = 1$. For the inductive step, Every n-fold may be considered as a superfold of the stamps $1, 2, \ldots, n-1$, with stamp n so placed as to make a fold. By the induction hypothesis, we may assume that all the $(n-1)$-folds are described by all the possible orderings of $n-1$ numberings satisfying the condition.

If the vertical line representing stamp $n-1$, when extended, would hit a horizontal line representing the perforations between some stamp k and $k+1$, where $k \equiv n-1 (\mathrm{mod} 2)$, then, to make a fold, the horizontal line of perforations between $n-1$ and n must lie completely within the other horizontal line, and cannot cross the vertical lines of stamps k or $k+1$.

If the vertical line of stamp $n-1$, when extended, misses the horizontal line of perforations between k and $k+1, k \equiv n-1 (\mathrm{mod} 2)$, then the vertical line of stamp n may be placed so that stamps $k, k+1$ lie either entirely within or entirely outside

the lines $n, n-1$

But these possibilities for placing stamp n are precisely all the ways of satisfying the given condition. □

The terms *within* and *outside* will be used again later in proving the correctness of an algorithm to generate stamp foldings.

A variation on these stamp foldings we consider are those with one symmetry removed, which we will refer to as $R(n)$. We accept in $R(n)$ only those members of $S(n)$ which begin with a lower number than they end with. We define an equivalence relation R on permutations such that aRb iff a is the reverse of b. The relation R partitions $S(n)$ into pairs of permutations. This corresponds to ignoring the distinction between the stamp labelled 1 and the stamp labelled n, meaning that we are not concerned with which end is which. We notice that $|R(n)| = |S(n)|/2$ and that no two diagrams representing different permutations will be identical. Figure 6 shows the permutations of $R(4)$.

In addition, we can remove a second type of symmetry if we ignore which end of the strip of stamps is labelled 1 (as opposed to n). By doing this we can remove foldings that are symmetric to each other by reversing the labelling. Removing both this symmetry and the one mentioned above gives us a representation for unlabelled stamp foldings, which we will refer to as $U(n)$ in general. For practical purposes, we must keep the labels. For example, the permutations 1342, 3124, 2431, and 4213

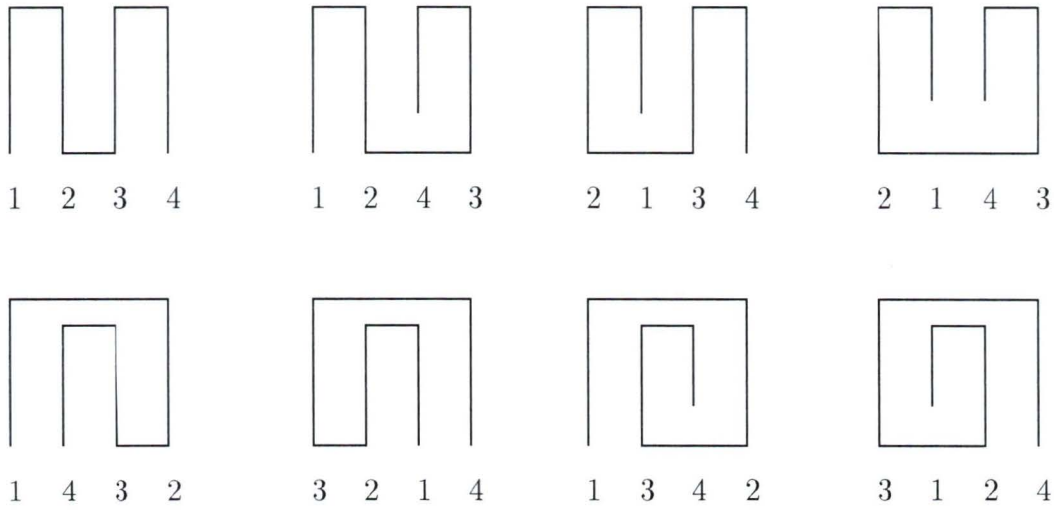

Figure 6 The 8 stamp foldings of $R(4)$

are all labellings of the same unlabelled stamp folding Likewise, 1234 and 4321 are both labellings of the same unlabelled stamp folding Out of the 4 maximum possible labellings of an unlabelled stamp folding, we choose the lexicographically smallest Figure 7 shows all the stamp foldings of $U(4)$

3.2 Definitions for Meanders

A *meander* is a permutation of $1, \ldots, n$ described by the following physical model

Consider a river flowing from the South-West to the East that crosses an East-West road n times Our convention is to label the crossings of the road from 1 through n starting with the entry point from the South-West as 1 and ending with the exit point to the East as n Figure 8 shows all the meanders where $n = 5$ The thin horizontal lines represent roads We refer to all the meanders of length n as $M(n)$ The difference between a meander and an n-fold is that an n-fold has no restrictions

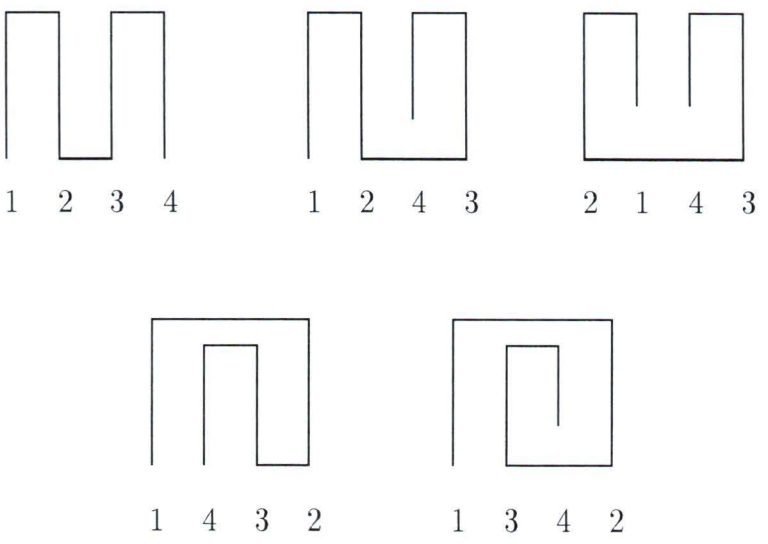

Figure 7 The 5 stamp foldings of $U(4)$

regarding the 1-stamp and the n-stamp, whereas a meander has restrictions regarding the ends of the strip

A *closed meander* is a variation of a meander in which there are $2n$ crossings of the river They are obtained by taking all the odd-length meanders of size $2n-1$ and adding a connection (and extra crossing of the road) between the entry and exit points As a convention, we always construct this connection to the east of the others This addition of one crossing gives exactly all the closed meanders because any closed meander of length $2n$ can be broken apart at the last crossing, leaving a meander of length $2n-1$ Figure 9 shows the 8 closed meanders where $2n=6$ We refer to the closed meanders of length n as $C(n)$

Meanders are of interest as a topological problem to nuclear physicists as a model for the compact folding of polymers (see [2], [3], and [4])

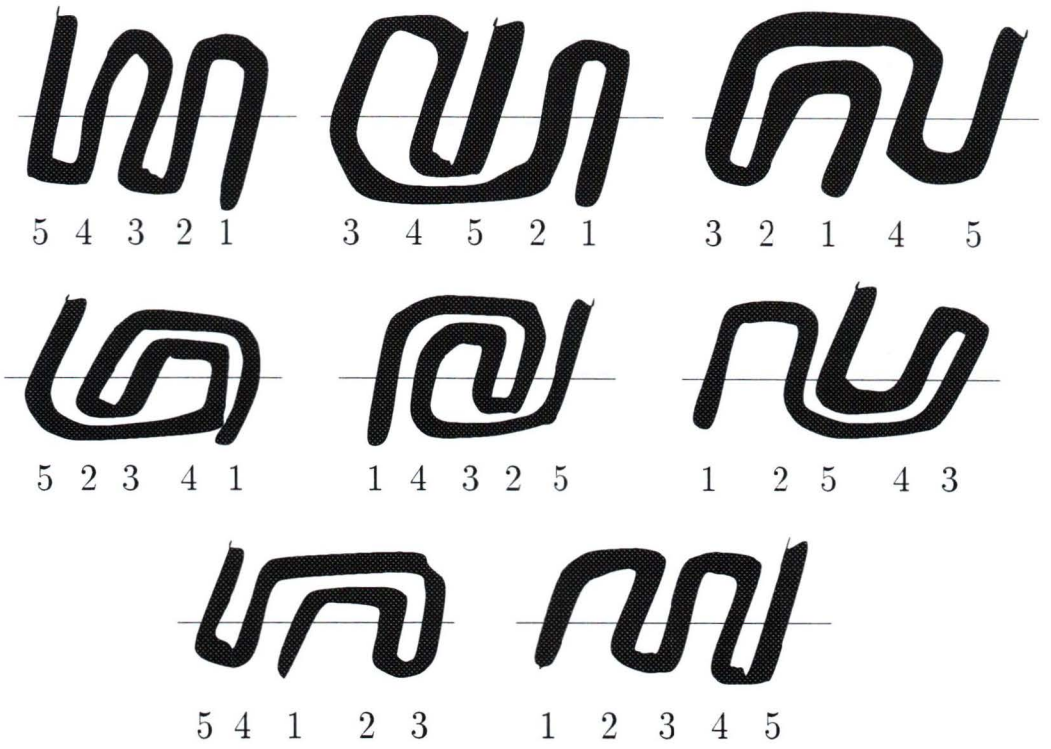

Figure 8 The 8 meanders where $n = 5$, $M(5)$

There is a one-to-one correspondence between meanders and *simple alternating transit (S A T) mazes*, which is given in [14].

We do not consider issues of symmetry with meanders and closed meanders in this thesis. It appears to be a non-trivial problem to determine which closed meanders are symmetric using a data structure.

3.3 A Data Structure and Algorithm for Stamp Foldings

The proof of Koehler's Theorem inspires a recursive algorithm for generating stamp foldings. If we start with a valid stamp folding of size n, we can generate stamp foldings of size $n + 1$ by attaching the stamp labelled $n + 1$ in all possible positions

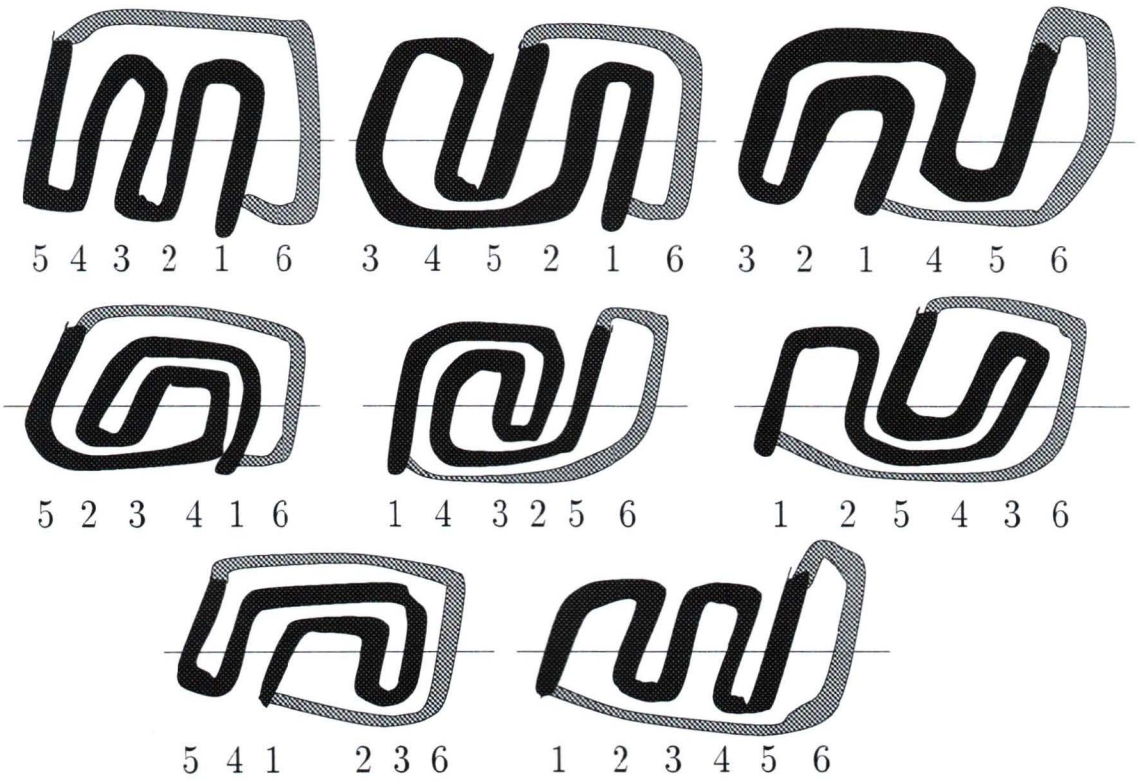

Figure 9: The 8 closed meanders where $2n = 6$, $C(3)$

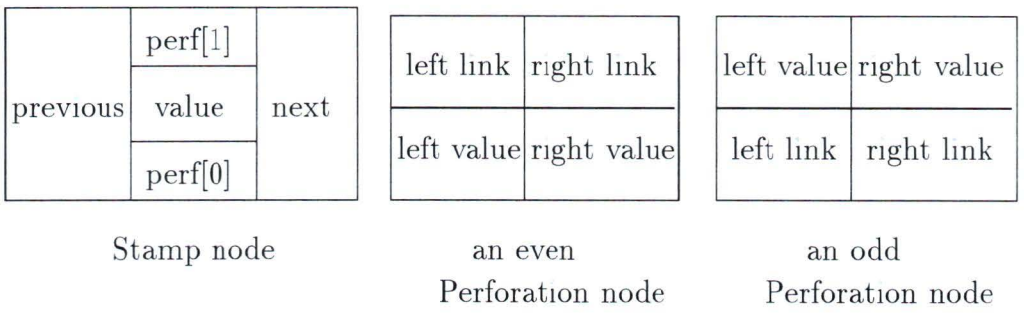

Figure 10 Nodes for the stamp folding Data Structure

which do not violate the condition of the theorem

The data structure we use involves a doubly-linked list, where each element in the list represents a labelled stamp There are also pointer connections between stamps corresponding to the perforations between them Each stamp except for the first and last will be connected by means of perforations to exactly two other stamps

These perforations are also represented by nodes Each perforation node has two values and two pointers to point to the stamps it connects This makes it possible to traverse from a stamp labelled k to a stamp labelled $k+1$ or vice versa in constant time

To illustrate this data structure, a diagram for the stamp folding 4, 3, 2, 6, 5, 1 is shown in Figure 11 Figure 10 shows the fields for the stamp and perforation data structures We will refer to the left value and left link of a perforation as simply the left side (and similarly for the right) for the sake of brevity Note that in Figure 11, the stamp labelled 4 and the stamp labelled 0 are connected, resulting in a circular list

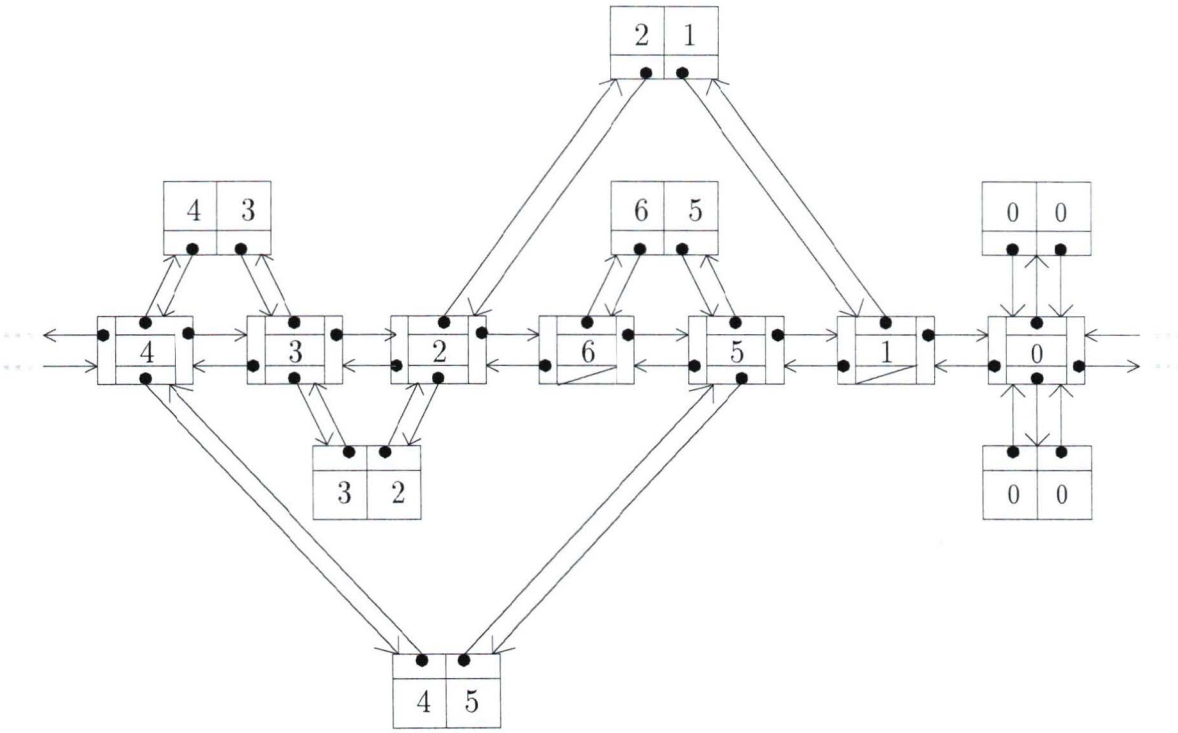

Figure 11 The Data Structure for stamp folding 4, 3, 2, 6, 5, 1. The perforation for 6 is omitted for clarity.

3.3.1 Description of Algorithm

The pseudocode given in Figure 12 describes the algorithm to generate unrestricted stamp foldings of length $n+1$ given a stamp folding of size n. Parentheses are used in a functional way to indicate reference to fields in the different parts of the data structure. This style will be used throughout the thesis when describing algorithms.

The stamp folding algorithm accepts a valid member of $S(n)$, say v and generates all possible members of $S(n+1)$ which have v as a subsequence. For example, if the algorithm is given the stamp folding depicted in Figure 11, it would choose the following positions as valid for placing the stamp labelled 7: between 2 and 6, between 4 and 3, between 6 and 5. It would also use this order of choices, since it is designed to start from the current stamp (6 in this case) and move leftwards in its search for valid positions, wrapping around to the far right when the far left has been reached. This is why line 16 in the pseudocode of Figure 12 indicates that the variable *current* is reset to the stamp previous to it (note that previous items are leftwards in the list). By performing these stamp placements and generations recursively, the algorithm can generate all valid stamp foldings.

Initialization includes creating the stamps labelled 0 and 1 and the perforations attached to those stamps. Then GenStamp can be called, passing in the stamp labelled 1.

The parameter *latest* represents the stamp most recently added to the data struc-

```
GenStamp(latest)
(1)  local parity, newPerf, temp, current
(2)  parity ← value(latest) mod2
(3)  if value(latest) = n then
(4)       display the folding
(5)       exit
(6)  end if
(7)  value(temp) ← value(latest) +1
(8)  right side(newPerf) ← temp
(9)  perf[(parity +1) mod 2](temp) ← newPerf
(10) left side(perf[parity](latest)) ← temp
(11) current ← latest
(12) repeat
(13)      Insert (current, temp)
(14)      GenStamp (temp)
(15)      Remove (temp)
(16)      current ← previous(current)
(17)      if value(current) ≠ value(latest) then
(18)           Move(parity, current)
(19)      end if
(20) until current = latest
```

Figure 12: Pseudocode for Stamp generation algorithm

```
Move(parity, current)
    if right_side(Perf[parity](current)) = current then
        current ← left_side(Perf[parity](current))
        if value(current) ≠ 0 then
            return
        end if
        swap(newPerf)
    else
        current ← right_side(Perf[parity](current))
        swap(newPerf)
    end if
```

Figure 13 Pseudocode for Move subroutine

ture. The variable *temp* is the new stamp being added to the permutation. The variable *newPerf* is the newest perforation being added to the data structure. It is connected to *temp*. The variable *current* points to a stamp in the existing data structure to indicate the next valid position in which to place the new stamp *temp*. Insert is a procedure to insert the *temp* stamp before *current* in the linked list. When a new stamp is added, it is placed to the immediate left of *current*. Delete is a procedure to delete the *temp* stamp from the linked list.

Move is a subroutine which moves *current* to the next available position to the left or moves *current* to the rightmost available position if there are no more positions available on the left. This decision is made by checking whether the connection to a perforation is on the left or right side of that perforation. The Move subroutine is given in Figure 13.

The function swap switches the right and left sides of a perforation. This is necessary whenever *current* moves from a position leftwards from *latest* to a position

rightwards from *latest*.

3.3.2 Correctness and Running Time

The correctness proof for GenStamp requires a few definitions. We will state that a perforation $k, k+1$ in a stamp folding is *within* another perforation $r, r+1$ when k and $k+1$ both occur between the stamps r and $r+1$. We will say that a perforation $k, k+1$ in a stamp folding is *outside* a particular perforation if it is not within that perforation. The same terminology can be used of an individual stamp to say that it is either *within* or *outside* a particular perforation.

Theorem 2 *GenStamp generates all the stamp foldings of length n.*

Proof. Correctness of the algorithm is shown by referring to the argument in Koehler's theorem and using mathematical induction. The following serves as an invariant for the induction. In the GenStamp algorithm, given an r-fold, the Move subroutine always chooses exactly those positions that are required to build all the $r+1$-stamps that start from that r-fold.

Basis. Consider the stamp labelled 1. It is trivial to see that Genstamp will generate both 21 and 12 from this, and so the invariant holds for the basis.

Inductive Step. Assume that we are given an r-fold which has been generated by GenStamp and that the invariant was true during each step of that generation.

When deciding where all the valid positions for the $r+1$-stamp are, the algorithm moves the *current* pointer using information about all the perforations between k and $k+1$ where $k \equiv r(\mathrm{mod}2)$

In creating the perforation $r, r+1$, the algorithm begins knowing the position of r relative to the other stamps. It suffices to show that the Move subroutine maintains these conditions while allowing every valid position. When Move traverses leftwards in an attempt to place the stamp $r+1$ in each valid position, it makes one of three decisions at each stamp it encounters. (1) If *current* is at the left side of a perforation of the parity in question, $r, r+1$ is entirely within that perforation, so the algorithm resets *current* to the rightmost side of that perforation. (2) If *current* is at the right side of a perforation of the parity in question, $r, r+1$ is entirely outside that perforation, so the algorithm resets *current* to just beyond the leftmost side of that perforation. (3) If *current* is the stamp labelled 1 and we are considering even parity, *current* is moved one place leftward, since there is no perforation at this point. It should be noticed that when the stamp labelled 0 is reached, Move does not cause a change to the *current* pointer. Afterwards *current* is reset to the stamp on the far right of the linked list (to the left of 0).

Thus, the algorithm ensures that the new perforation being added is either entirely within or entirely outside all other such perforations $k, k+1$. In doing this, the perforations of a given parity never cross each other. Also, all possible valid positions are used for *current* because the only positions missed are those which are outside a

perforation that r is within and those which are inside a perforation that r is outside.

□

The stamp folding algorithm works in a recursive backtracking manner. It is obvious from the algorithm that the amount of computation done for each call of the algorithm is proportional to the number of recursive calls being made. It will therefore suffice to show that there are no dead ends and no paths in the computation tree. (Recall the definitions of dead end and path given in the Introduction.) There are no paths in the tree simply because at every recursive level there are at least two possible values for *current* - the positions immediately to the left and right of latest. There are also no dead ends because new stamps can always be built from existing smaller ones in this way. Thus, GenStamp is CAT.

It should be noted that an algorithm similar to GenStamp was developed by W. F. Lunnon (see [12]). Lunnon used a different data structure and did not adapt his result to the question of generating meanders.

3.4 An Algorithm to Draw Stamp Foldings

It is not difficult to design an algorithm which, given a stamp folding of length n, will draw a diagram similar to those of Figures 5 through 7.

The first step is to draw n parallel lines to represent the stamps. What then remains is to draw the perforations. We will consider only the perforations which are

drawn above the stamps since the other perforations can be drawn in an analogous way.

If we associate the left side of a perforation with a left parenthesis and similarly for the right side, we notice a very simple property - the perforations drawn above the stamps correspond to a well-formed parenthesis string. This is true because each perforation drawn has two matching parts and no perforations may cross each other.

We would like to know what horizontal position each perforation should have. Using the well-formed parenthesis concept, we can make use of a general tree to represent this. The algorithm shown in Figure 14 constructs a general tree where each node represents information about a perforation. Each node contains the vertical position of each of the perforation's ends (called *leftorder* and *rightorder*), a pointer to the perforation (called *perforation*), and the level at which that node occurs in the tree (called *level*). The level is to be filled in after constructing the entire tree. Figure 15 depicts the structure of a node in this tree.

The initialization required before using this algorithm is simple. It involves setting *treecurrent* to a blank tree node which will be the root of the tree.

In TreeMake, *current* is a pointer to a stamp, *node* is used to make a new node in the *tree*, and *order* is the order in which *current* appears in the stamp folding. Also, *treecurrent* is the current node in the *tree*. We call TreeMake(8) in the example given.

TreeMake will build a general tree representing the parenthesis structure of the

```
TreeMake(first)
    local current, order, node, treecurrent
    current ← next(first)
    order ← 1
    while current is not the stamp labelled 0
        if current is the stamp labelled n then
            order ← order +1
            current ← next(current)
            continue at the top of the while loop
        end if
        if current is a left parenthesis then
            node ← new, empty tree node
            leftorder(node) ← order
            place node as next child of treecurrent
            treecurrent ← node
        else
            rightorder(node) ← order for current node in tree
            treecurrent ← parent(treecurrent)
        end if
        order ← order +1
        current ← next(current)
    end while
```

Figure 14 Pseudocode for the TreeMake algorithm

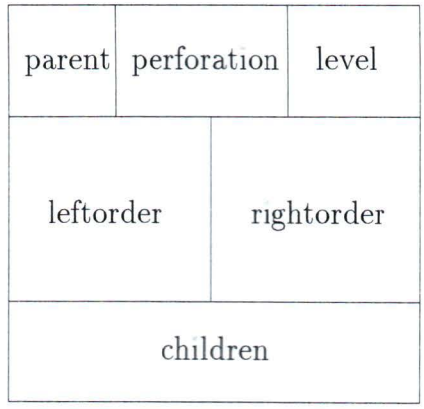

Figure 15 A node of the tree used in TreeMake

perforations. After using this algorithm, a simple recursive procedure can be used to assign the proper values to *level* in the tree by assigning 1 to all leaves and assigning 1+ max(*level*s of children) to all other nodes. We will then have a data structure which contains the relative horizontal level of each perforation in addition to the vertical positions where they begin and end. For example, Figure 16 depicts the tree this algorithm would construct for the stamp folding 8, 7, 6, 5, 10, 11, 4, 3, 12, 2, 1, 9 after the levels have also been assigned.

The TreeMake algorithm and its data structure are original work by the author.

3.5 Removing Symmetric Foldings

We can easily remove symmetric foldings of $R(n)$ by choosing only those stamp foldings whose starting number is smaller than its ending number. We know this because the symmetry we are considering simply involves the reversing of the permutation. To decide whether a particular stamp folding is to be kept or not simply requires a comparison of the first and last stamps - a constant time operation. Thus, this modification to GenStamp is also CAT.

Removing the second type of symmetry also involves a relatively simple modification to GenStamp. This is equivalent to generating unlabelled stamp foldings and then imposing a labelling on them. We know that for any given unlabelled stamp folding there are 4 different labellings: an initial labelling, its relabelled permutation, and the reverse of each of these, where a relabelled permutation is created by replac-

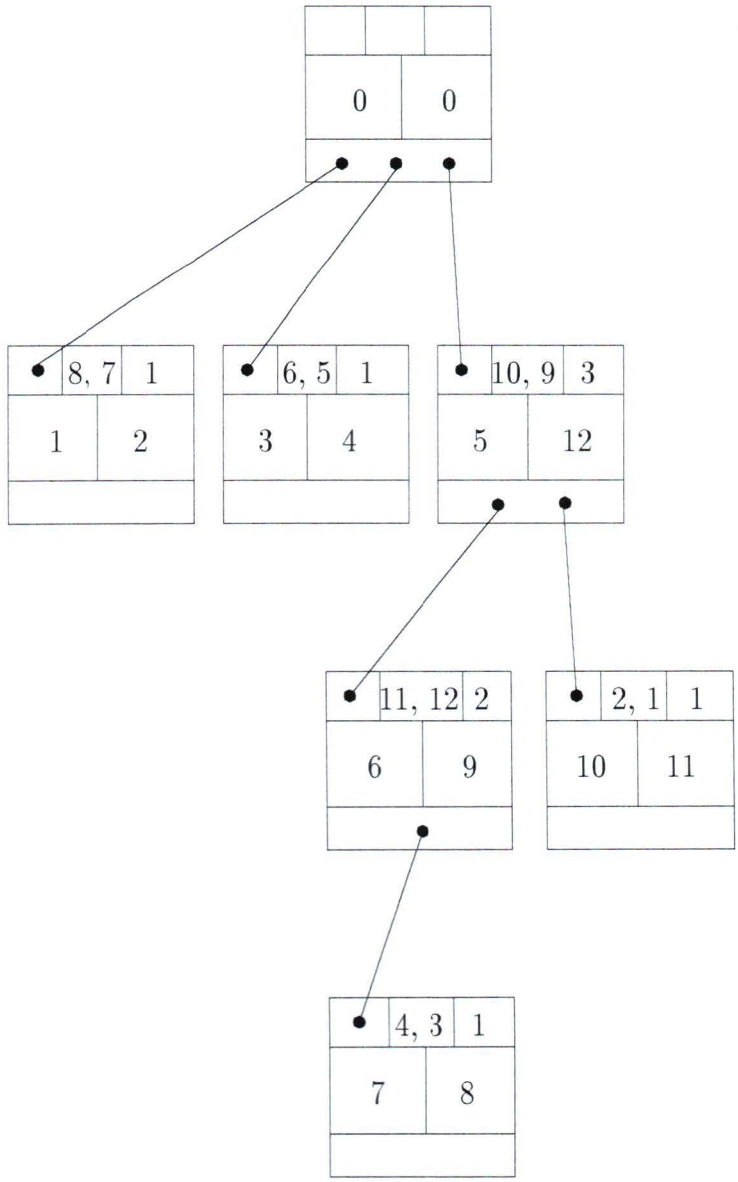

Figure 16: Example of the tree for $8, 7, 6, 5, 10, 11, 4, 3, 12, 2, 1, 9$

ing a value x with $N+1-x$. We can think of these 4 labellings of a permutation as defining an equivalence relation on all stamp foldings.

Suppose we consider only the first two and last two elements of a permutation that is generated, say $a, b, \ldots, c, d$. Then, it suffices to only accept this permutation if $a, b < d, c$ and $a, b < N+1-a, N+1-b$ and $a, b < N+1-d, N+1-c$. For example, if we allow π to represent a particular $(n-4)$ - permutation which forms part of a stamp folding, the stamp folding $3, 2, \pi, 1, 4$, has 3 other equivalent labellings: $4, 1, \pi, 2, 3$ and $4, 5, \pi, 6, 3$, and $3, 6, \pi, 5, 4$. Thus, we accept $3, 2, \pi, 1, 4$ since it is lexicographically least. This decision to accept or reject the given stamp folding can always be made in constant time, independent of the length of π. Thus if we modify GenStamp to only accept those stamp foldings that match the above criteria, it will require a constant amount of time to make the decision, and at most 3 in 4 stamp foldings will be rejected. Thus, the modified algorithm is also CAT.

3.6 Meanders and Closed Meanders

Meanders can also be generated by making some additions to GenStamp. We can easily ensure that the stamp labelled 1 is not within any even perforations by restricting the addition of odd-numbered stamps. A new odd-numbered stamp will violate the needed condition for the stamp labelled 1 if and only if *current* changes from some position on the right of the stamp labelled 1 to the stamp labelled 1. To prevent this, we make two changes to the original algorithm. Firstly, if *current* ever gets reset to

the stamp labelled 1, we reset it to the last stamp. Secondly, if *current* ever gets reset to the stamp labelled 0, we reset it to the stamp labelled 1. All that is required to implement this is an array which, for each stamp label in the folding, indicates which side of 1 it is on. This can be maintained in constant time per addition or removal. These new conditions require changes at lines (11) and (16) of GenStamp on page 21.

What remains is to ensure that the stamp labelled n can be extended infinitely in the vertical direction without crossing any other stamps or perforations. Any stamp which may be extended in this way, we will call *free*. Similarly, if the highest-numbered stamp in a folding is free, we will call the folding *free*. Any stamp or folding which is not free, we call *blocked*. We devise a data structure for this which will keep track of how many steps of new stamp additions it takes to make the highest-numbered end free. If this number is greater than the number of stamps left to add, the stamp labelled n has no chance to be free.

Keeping track of how many stamp additions are required to make the highest-numbered end free involves keeping a count of that number. A stack of data regarding this is constructed so that the number of items on this stack is the number of stamp additions required to make the highest-numbered end free. We also generate an extra stamp for each stack item so that when a path of length 2 or more would normally be made in the computation tree, the new stamps can simply be used. These extra stamps will be part of the folding and will actually be the addition required to make the highest-numbered end free. The extra stamps and perforations

that are added must be designed so that when the algorithm decides to make a path in the computation tree, it will be able to simply connect the extra stamps to the others in constant time. In general, GenStamp is enhanced mostly by using a separate stack and adding in extra stamps that will function as what would have been created by a path in the computation tree.

The AddStamp routine may push or pop stack items based upon the effect of the stamp being added with respect to the free-ness of the highest numbered end. RemoveStamp restores the changes made by AddStamp (as before).

The new version of GenStamp, called GenMeander is found as Figure 17.

Initialization includes creating the stamps labelled 0 and 1 and the perforations attached to those stamps as with GenStamp. Then GenMeander can be called, passing in the stamp labelled 1.

In GenMeander, the variable *latest* represents the stamp most recently added to the data structure. The field *temp* is the new stamp being added to the permutation. The field *newPerf* is the newest perforation being added to the data structure. It is connected to *temp*. The field *current* points to a stamp in the existing data structure to indicate the next valid position in which to place the new stamp *temp*. Prune is a boolean function which determines whether the last stamp or two need to be generated specially to keep the highest-numbered end free. AddStamp is a procedure to insert the *temp* stamp after *current* in the linked list. Note that *current* is to the left of

```
GenMeander(latest)
(1)   local: parity, newPerf, temp, current
(2)   parity ← value(latest) mod2
(3)   if value(latest) = n then
(4)         display the folding
(5)         exit
(6)   end if
(7)   if Prune(latest)
(8)         use the clue and generate up to two extra stamps
(9)   else
(10)        value(temp) ← value(latest) +1
(11)        right side(newPerf) ← temp
(12)        perf[(parity +1) mod 2](temp) ← newPerf
(13)        left side(perf[parity](latest)) ← temp
(14)        current ← latest
(15)        repeat
(16)              AddStamp (current, temp)
(17)              GenStamp (temp)
(18)              RemoveStamp (temp)
(19)              current ← previous(current)
(20)              if value(current) ≠ latest then
(21)                    Move(parity, current)
(22)              end if
(23)        until current = latest
(24)  end if
```

Figure 17 Pseudocode for the GenMeander algorithm

where the next stamp should be inserted, so new stamps are inserted to the right of *current*. RemoveStamp is a procedure to delete the *temp* stamp from the linked list. Move is a subroutine which moves *current* to the next available position to the left or moves *current* to the rightmost available position if there are no more positions available on the left. This decision is made by checking whether the connection to a perforation from the *current* stamp is on the left or right side of that perforation, as in GenStamp.

The stack data structure which is used to keep a record of how many stamps need to be added to make the highest-numbered end free is called a *Clue*. The left side of Figure 18 depicts this data structure by showing that there are 3 fields of data in the main record, and 7 fields of data in each stack item (referred to as a step node). In the clue record, *first* is a link to the first step, which is the top of the stack, *last* is a link to the last step, which is the bottom of the stack, and *depth* is an integer representing the number of items on the stack. Each step record represents where a stamp would be added to help make the highest-numbered end free. In effect, each step represents a position between two stamps. Within the step record, *left_wall* represents the label of the stamp to the left of this position, while *right_wall* represents the label of the stamp to the right of this position. The fields *direction_needed*, *direction_taken*, and *type* are all used in the AddStamp and RemoveStamp routines to help maintain this data structure. The field *direction_needed* indicates the direction of the *corresponding* stamp of the present node in relation to the *corresponding* stamp of the first item that was pushed on the clue stack. The field *direction_taken* gives the direction of stamp

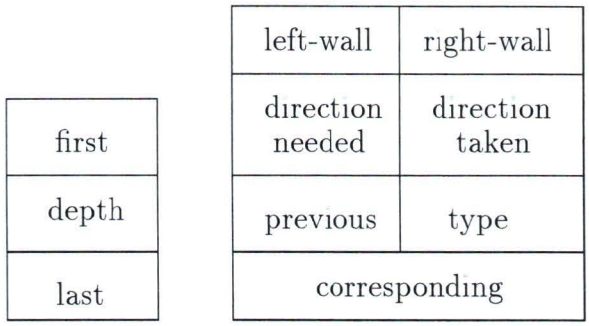

Figure 18. Clue data structure used for meanders

addition which caused this step in the clue to be constructed. The field *previous* is part of the stack structure, pointing to the next item in the stack. Finally, *corresponding* is a link to a stamp which is constructed and connected into this position in case the clue needs to be used in the stamp folding. Two example clues are sketched in Figure 33 on page 46.

It is worth noting that GenMeander is simply a refinement of GenStamp. Thus, it is also related to the proof of Koehler's theorem. In GenMeander, all the meanders of length $n+1$ which are obtained by a particular meander of length n are generated recursively by a single subroutine call to GenMeander. The only differences from the generation of stamps are the following. (1) An additional data structure is used and maintained in AddStamp and RemoveStamp to determine how many stamp additions are needed to make the highest-numbered end free. The data structure also holds those specific additions that are needed. (2) The Move subroutine now ensures that the infinite extension of the 1-stamp is not violated. (3) Either the last stamp or two

```
Prune(value)
    if N mod 2 = 0 and depth(clue) > 0 then
        if ending(clue) is to left of the 1 stamp or is the 1 stamp then
            return depth(clue) +2 ≥ N− value
        end if
    if N mod 2 = 0 and value is to the left of 1 and depth(clue) = 0 then
        return depth(clue) +2 ≥ N− value
    end if
    return depth(clue) +1 ≥ N− value
```

Figure 19 Pseudocode for the Prune function

are generated in a way that guarantees the end will be free, or the additional stamps from the clue are connected to the partially generated meander, or both

Prune is a boolean function which tells GenMeander whether or not it should be attaching the clue or generating the last few stamps in a careful way The section of the GenMeander algorithm which is used if Prune returns true restricts the additions of stamps to particular additions necessary to complete a path in the computation tree A description of the Prune function is given in Figure 19

Line 8 of GenMeander states "use the clue and generate up to two extra stamps" This involves the decisions and actions given in Pseudocode form in Figure 20

Two examples of stamp foldings which would cause the execution of line 5 of the pseudocode in Figure 20 are given as Figure 21 The solid lines indicate what has already been generated The dotted lines indicate what remains to be added, either by connecting the stamps in the clue, or by generating new stamps, or both The stamps which are to extend infinitely either up or down are indicated with longer

```
(1) old_depth ← depth(clue)
(2) if clue is of length 0 then
(3)       if value(latest) = n − 1 then
(4)             if latest is to right of 1 stamp or value(latest) is odd then
(5)                   add a final stamp on the far right
(6)             else
(7)                   add a final stamp on the far left
(8)             end if
(9)       else
(10)            add a stamp to the far left, then to the far right
(11)      end if
(12) else
(13)      if first(clue) = n then
(14)            display the folding
(15)      else if first(clue) = n − 1 then
(16)            if N is even then
(17)                  add a final stamp on the far right
(18)            else
(19)                  if corresponding(first(clue)) is to the right of 1 then
(20)                        add a final stamp on the far right
(21)                  else
(22)                        add a final stamp on the far left
(23)                  end if
(24)            end if
(25)      else if first(clue) = n − 2 then
(26)            add a final stamp on the far left, then one on the far right
(27)      end if
(28) end if
```

Figure 20 Pseudocode for generating the last stamp or two in GenMeander

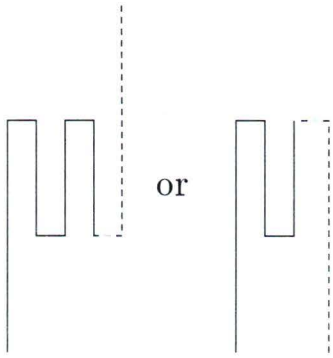

Figure 21 Adding a final stamp on the right when the folding is free

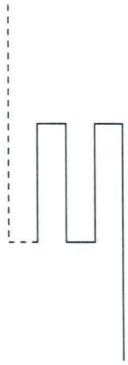

Figure 22 Adding a final stamp on the left when the folding is free

lines. This kind of diagram will be used several times following.

Similarly, an example of a stamp folding which would cause the execution of line 7 of the pseudocode in Figure 20 is given in Figure 22.

Figure 23 depicts an example of a stamp folding which would cause the execution of line 10 of the pseudocode in Figure 20.

Line 17 of the pseudocode in Figure 20 is also clarified with an example. Figure 24 gives a stamp folding which would cause the execution of this line. In this case, the part of the dotted line which extends the solid line upwards would already be

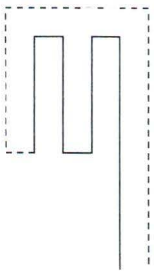

Figure 23 Adding final stamps on the left and then the right when the folding is free

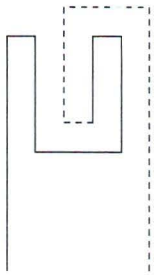

Figure 24 Adding a final stamp when the parity is even and the folding is blocked

represented with the clue data structure, while the remaining part of the dotted line, which extends to the right and then downwards infinitely, would be this final addition

Similar to the previous diagram, Figure 25 gives a stamp folding which would cause the execution of line 20 in the pseudocode of Figure 20. In Figure 25, the section of the dotted line which extends the solid line downwards would already be represented with the clue data structure, while the remaining part of the dotted line, which extends to the right and then upward infinitely, would be this final addition

Figure 26 gives an example of a stamp folding which would cause the execution of line 22 in the pseudocode of Figure 20. As in the last two examples, the first section of dotted line connected to the solid line represents a stamp which is already part of

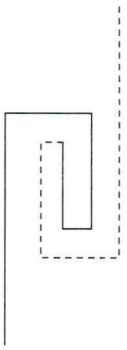

Figure 25 Adding a final stamp on the right when the folding is blocked

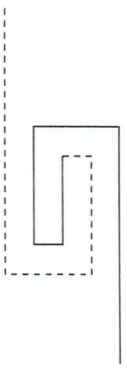

Figure 26 Adding a final stamp on the left when the folding is blocked

the clue. The remainder of the dotted line is actually added as a result of the rest of the algorithm.

An example of a stamp folding that would cause the execution of line 26 in Figure 20 is given in Figure 27. Again, the first section of dotted line representing a stamp would already be part of the clue, while the sections on the far left and far right would be added as a result of this part of the pseudocode.

We return to describing further details of the GenMeander algorithm of Figure 17 at this point.

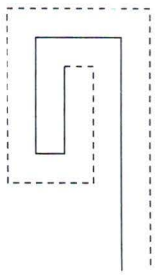

Figure 27. Adding final stamps on the right and then the left when the folding is blocked

Given in Figure 28 is the pseudocode for the new version of the Move routine for GenMeander. The function swap found in this pseudocode switches the right and left sides of a perforation. This is necessary whenever *current* moves to a position rightwards from *latest*.

We now turn our attention to the AddStamp routine referred to in GenMeander. There are several details of that routine which are now discussed.

Pseudocode for the AddStamp routine is given in Figure 29.

Line 5 of the pseudocode for AddStamp is further described by the pseudocode of Figure 30.

Four stamp folding examples related to a free folding becoming blocked are given in Figure 31. These are examples of stamp foldings which are related to line 5 of AddStamp. These four stamp foldings represent the four assignments to *direction_needed(temp_step)* in Figure 30.

Line 14 of AddStamp involves the following assignment.

```
Move(parity, current)
    if current = first and parity is even then
        set current to the 1 stamp and return from Move
    end if
    if current is not the 1 stamp then
        if right_side(Perf[parity](current)) = current then
            current ← left_side(Perf[parity](current))
            if value(current) ≠ 0 then
                return
            end if
            swap(newPerf)
        else
            current ← right_side(Perf[parity](current))
            swap(newPerf)
        end if
    else
        current ← first
    end if
```

Figure 28 Pseudocode for the Move routine in GenMeander

```
AddStamp(current, new)
1     local temp_step
2     code for adding stamp into linked list
3     if depth(clue) = 0 then
4           if new addition is not at beginning or end then
5                 create temp_step appropriately
6                 Push(temp_step)
7                 temp_step ← 0
8           end if
9     else
10          if stamp to right of new addition has value of right_wall(last(clue)) then
11                temp_step ← Pop()
12                if clue stack is not empty then
13                      last(clue) ← prev(temp_step)
14                      set type(last(clue))
15                else
16                      temp_step(prev) ← 0
17                end if
18          else
19                if the addition of last stamp shortened the clue then
20                      set direction_needed based upon type(last(clue))
21                      set rest of temp_step appropriately
22                      Push(temp_step)
23                      temp_step ← 0
24                else
25                      set temp_step and adjust last(clue) appropriately
26                      Push(temp_step)
27                      temp_step ← 0
28                end if
29          end if
30    end if
```

Figure 29 Pseudocode for the AddStamp routine in GenMeander

```
if new is even then
    if new was added to the right of new −1 then
        direction_needed(temp_step) ← left
    else
        direction_needed(temp_step) ← right
    end if
else
    if new was added to the right of new −1 then
        direction_needed(temp_step) ← right
    else
        direction_needed(temp_step) ← left
    end if
end if
type(temp_step) ← 0
if new was added to the right of new −1 then
    right_wall(temp_step) ← next(new)
    left_wall(temp_step) ← new
else
    right_wall(temp_step) ← new
    left_wall(temp_step) ← prev(new)
end if
direction_taken(temp_step) ← unknown
```

Figure 30: Pseudocode for line 5 of AddStamp

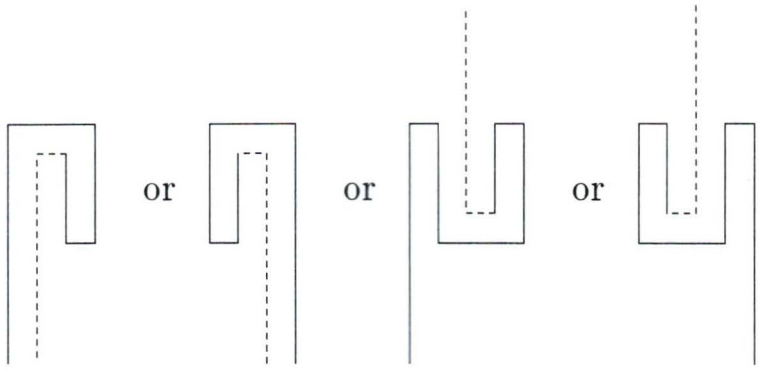

Figure 31: Adding a new stamp to create a new clue

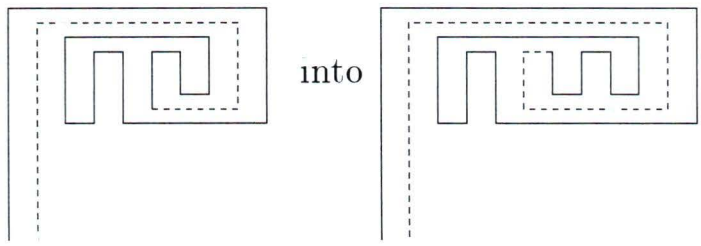

Figure 32: Adding a new stamp which lengthens the clue when the previous addition had shortened the clue

type(last(clue)) ← 4∗ (*new* is even) +2∗ (*new* added right) + *direction_taken* (*prev*(*temp_step*() is left +1

This is simply a binary encoding based on the three variables *parity(new)*, *direction_added(new)*, and *direction_taken*(*prev*(*temp_step*))

An example to illustrate the effects of the section of pseudocode in AddStamp from lines 19 through 23 is shown in Figure 32

Two example clues which correspond to the two parts of Figure 32 are illustrated in Figure 33

Line 20 of AddStamp sets *direction_needed(temp_step)* according to the table in Figure 34. A dash in the table indicates that an item is irrelevant in determining the result column in that particular case

Line 21 of AddStamp is further explained through the pseudocode of Figure 35

The section of pseudocode in AddStamp from lines 25 through 27 is shown in Figure 36

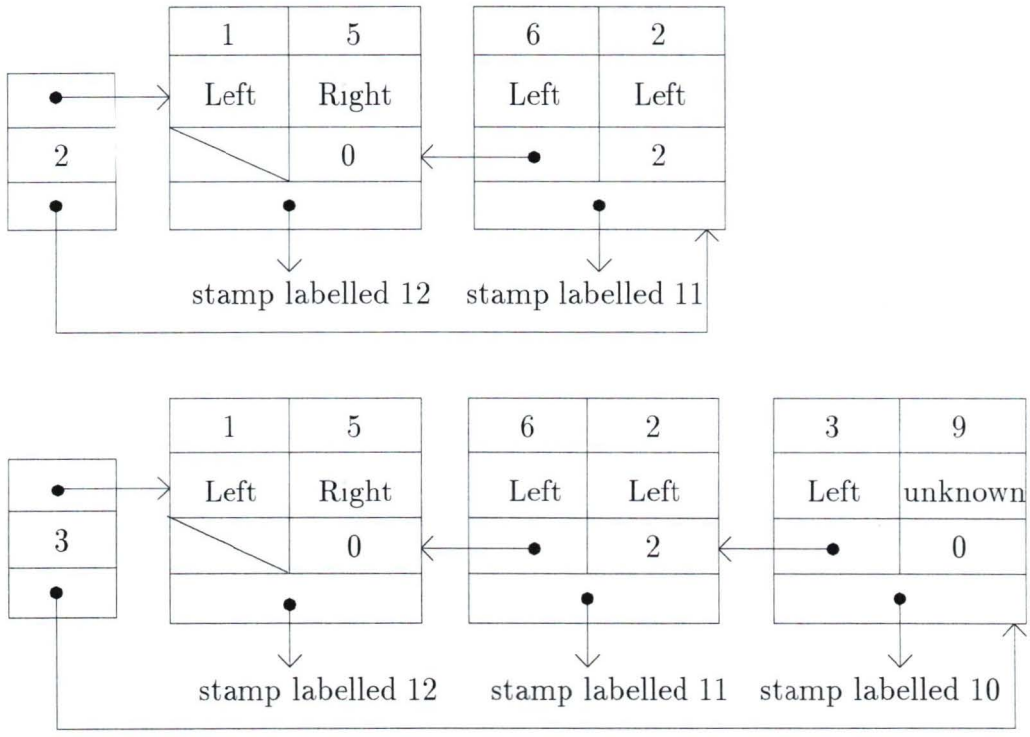

Figure 33 The two Clue data structures from Figure 32

Line 25 of AddStamp is further explained through the pseudocode of Figure 37.

We now turn our attention back to one remaining routine which is referred to in the GenMeander algorithm. Figure 38 gives the pseudocode for the RemoveStamp routine. RemoveStamp restores the clue to the way it was before the corresponding call to AddStamp.

Closed meanders of length $2n$ can be generated from regular meanders by doing the following. Generate all meanders of a certain odd length $2n-1$, place a stamp labelled $2n$ at the end of this permutation, connect $perf[0]$ of this new stamp to the stamp labelled 1, and connect $perf[1]$ of this new stamp to the stamp labelled $2n-1$. This addition to the meander generation algorithm does not affect its running time

type(*last*(*clue*))	*new* added to right?	*direction_taken* (*last*(*clue*))	*direction_needed* (*temp_step*)
1	No	-	Right
1	Yes	Left	Left
1	Yes	Right	Right
2	Yes	-	Left
2	No	Right	Right
2	No	Left	Left
3	No	-	Right
3	Yes	Left	Left
3	Yes	Right	Right
4	Yes	-	Left
4	No	Right	Right
4	No	Left	Left
5	No	-	Left
5	Yes	Left	Right
5	Yes	Right	Left
6	Yes	-	Right
6	No	Right	Left
6	No	Left	Right
7	No	-	Left
7	Yes	Left	Right
7	Yes	Right	Left
8	Yes	-	Right
8	No	Right	Left
8	No	Left	Right

Figure 34 A chart explaining line 20 of the AddStamp algorithm

```
type(temp_step) ← 0
if (new is odd and direction_needed(temp_step) is right)
      or (new is even and direction_needed(temp_step) is left) then
      right_wall(temp_step) ← next(new)
      left_wall(temp_step) ← new
else
      right_wall(temp_step) ← new
      left_wall(temp_step) ← prev(new)
end if
direction_taken(temp_step) ← unknown
```

Figure 35 Pseudocode for line 21 of AddStamp

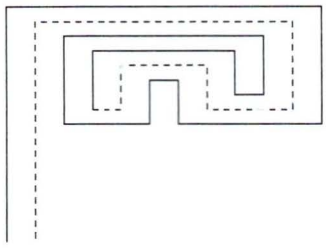

Figure 36 Adding a new stamp which lengthens the clue when the previous addition had also lengthened the clue

```
if new is added to the right of new −1 then
    direction_taken(last(clue)) ← right
else
    direction_taken(last(clue)) ← left
end if
direction_needed(temp_step) ← direction_needed(last(clue))
if new is even then
    if direction_needed(last(clue)) is left then
        right_wall(temp_step) ← next(new)
        left_wall(temp_step) ← new
    else
        right_wall(temp_step) ← new
        left_wall(temp_step) ← prev(new)
    end if
else
    if direction_needed(last(clue)) is left then
        right_wall(temp_step) ← new
        left_wall(temp_step) ← prev(new)
    else
        right_wall(temp_step) ← next(new)
        left_wall(temp_step) ← new
    end if
end if
direction_taken(temp_step) ← unknown
type(temp_step) ← 0
```

Figure 37: Pseudocode for line 25 of AddStamp

```
RemoveStamp(new, temp_step)
    code to remove stamp from linked list
    if temp_step is empty and depth(clue) ≠ 0 then
        temp_step = Pop()
        delete temp_step
    else
        if only one item on stack then
            Push(temp_step)
        else
            temp = Pop()
            delete temp
            Push(prev(temp_step))
            Push(temp_step)
        end if
    end if
```

Figure 38: Pseudocode for RemoveStamp routine in GenMeander

since adding the extra stamp takes constant time.

3.7 Open Problems and Extensions

This section briefly mentions extensions to the above research and problems related to it for which the author does not know a solution.

As seen in the algorithm for drawing stamp foldings, the perforations of each side of the folding correspond to well-formed parenthesis strings, and thus also correspond to trees. An algorithm to generate stamp foldings which directly uses tree generation might possibly be more elegant than the one given. Unfortunately, the problem of determining when two well-formed parentheses form the perforations to a valid stamp folding that is connected and does not cross itself appears to be quite complicated.

There is a connection between stamp foldings and simple alternating transit (S.A.T.) mazes (see [14]). It would be interesting to explore this connection to determine how to generate these mazes.

A variation of the stamp folding generation algorithm that would be of use in counting the number of stamp foldings of different sizes is the following. Generate only those foldings which are *oriented*. An oriented folding is one which begins with the stamp labelled 1. All foldings can be obtained from these by performing cyclic shifts of the permutation. Thus, we could generate all oriented stamp foldings of length n, and then multiply the number of these by n to get the total number of stamp foldings.

The problem of generating multidimensional foldings such as map foldings (2 dimensional) may be a matter of simply extending the existing data structure and algorithm. One algorithm that treats this subject is found in [13]. Similarly, it might be interesting to develop an algorithm to generate multidimensional meanders.

Another consideration would be to generate partial stamp foldings, where the stamps have not been completely piled. Flexagons (see [6]) are partial stamp foldings where the strip is still linear but the stamps are all triangular. Generation of Flexagons may be interesting as well.

4 Dumont (Genocchi) Permutations

This begins the second half of the research in this thesis, which can be read separately from the first. There are two distinct classes of permutations which are called Dumont Permutations. These permutations were introduced in [5] and elaborated in [15], [10], and [11]. The two classes of permutations have the same enumeration but no obvious bijection had been discovered. The highlight of this section of the thesis is the conjecture of such a bijection in section 4.5.

4.1 Definitions and Previous Results

The Dumont permutations (also called the Genocchi permutations) are of two kinds.

The *Dumont permutations of the first kind* (DP1(n)) are the set of permutations of $[2n] = \{1, 2, \ldots, 2n\}$ with the following two rules as restrictions.

1) every even number must be followed by a smaller number.

2) every odd number must be followed by a larger number or be the last value in the permutation.

For example, DP1(2) contains the following three permutations.

2 1 4 3 3 4 2 1 4 2 1 3.

Those for which $2n = 6$ are

2 1 4 3 6 5 3 4 2 1 6 5 4 2 1 3 6 5 4 3 5 6 2 1 5 6 3 4 2 1 6 3 4 2 1 5
2 1 5 6 4 3 3 5 6 4 2 1 4 2 1 5 6 3 4 3 6 2 1 5 5 6 4 2 1 3 6 4 2 1 3 5
2 1 6 4 3 5 3 6 4 2 1 5 4 2 1 6 3 5 5 6 2 1 4 3 6 2 1 4 3 5

These permutations are defined to be of even length for the sake of convenience. If we were to consider odd-length sequences, our highest value would be an odd number. According to rule 2 above, it would necessarily be last. Obviously, the permutations of length $2n + 1$ are simply the sequences of length $2n$ with the number $2n + 1$ appended at the end. For example, DP1(n) where $2n = 5$ contains the following three permutations.

2 1 4 3 5 3 4 2 1 5 4 2 1 3 5

The *Dumont permutations of the second kind* (DP2(n)) are a set of permutations of $[2n]$ with the following two rules as restrictions.

1) every odd position must have a value greater than or equal to its index.

2) every even position must have a value less than its index.

For example, DP2(2) includes only the permutations

2 1 4 3 3 1 4 2 4 1 3 2,

and DP2(3) includes only the permutations

2 1 4 3 6 5 3 1 4 2 6 5 4 1 5 2 6 3 4 1 6 2 5 3 5 1 4 2 6 3 6 1 4 2 5 3
2 1 5 3 6 4 3 1 5 2 6 4 4 1 5 3 6 2 4 1 6 3 5 2 5 1 3 2 6 4 6 1 3 2 5 4
2 1 6 3 5 4 3 1 6 2 5 4 4 1 3 2 6 5 5 1 4 3 6 2 6 1 4 3 5 2

These permutations are defined to be of even length for the sake of convenience,

as with DP1(n). If we were to consider odd-length permutations, our highest value would be an odd number. According to rule 1 above, it would necessarily be last. Obviously, the permutations of length $2n+1$ are simply the permutations of length $2n$ with the number $2n+1$ appended at the end. For example, DP2(n) where $2n = 5$ includes only the permutations

2 1 4 3 5 3 1 4 2 5 4 1 3 2 5

With the main definitions now given, the remainder of this part of the thesis deals with the following items: a brief discussion of the Genocchi numbers, an algorithm to generate the set DP2(n) and a proof of its correctness, an algorithm to generate the set DP1(n) and a proof of its correctness, an algorithm which maps an element from DP2(n) to an element of DP1(n), and an algorithm which maps an element from DP1(n) to an element of DP2(n). The last algorithm is simply a modification of the one preceding it. All these algorithms are new results.

4.2 Genocchi Numbers

The enumeration of both kinds of Dumont permutations give the Genocchi numbers, denoted by g_{2n}, and found in Sloane's database of integer sequences ([19]) as number A001469 or M3041. The first several values are given in the following table.

n	1	2	3	4	5	6	7	8	9	10
g_{2n}	1	1	3	17	155	2073	38227	929569	28820619	1109652905

One can obtain these using the generating function

$$\frac{2t}{e^t+1} = t(1-\tanh(\frac{t}{2})) = t + \sum_{n\geq 1}(-1)^n \frac{t^{2n}}{(2n)!}g_{2n},$$

which is given in [15] and [1]. Another way is to use the recurrence formula

$$\sum_{i\geq 0}(-1)^i \binom{n}{2i} g_{2(n-i)} = 0$$

given in [11]

The Genocchi numbers are related to the Bernoulli numbers in the following way (see [7]). The tangent function

$$\tan z = \frac{\sin z}{\cos z} = \sum_{n\geq 0}(-1)^{n-1}4^n(4^n-1)B_{2n}\frac{z^{2n-1}}{(2n)!}$$

includes the Bernoulli numbers B_n as coefficients. A section of this formula is referred to as the tangent numbers T_n

$$T_{2n-1} = (-1)^{n-1}\frac{4^n(4^n-1)}{2n}B_{2n}$$

The odd positive integers $(n+1)T_{2n+1}/2^{2n}$ are the Genocchi numbers g_{2n}

D. Dumont ([5]) showed that g_{2n+1} is the number of permutations of $[2n]$ which are Dumont permutation of the first kind and also of the second kind. In other words,

$$|DP1(n)| = |DP2(n)| = g_{n+1}$$

These results give no indication of how to generate the permutations.

4.3 A Backtracking Algorithm for Generating DP2(n)

In this section, we develop an algorithm for generating Dumont permutations of the second kind (DP2's) and prove that it is CAT.

The first approach we might take is to create a backtracking algorithm that determines the value of each successive position starting at the first. Unfortunately, this naive approach to backtracking will not give us a CAT algorithm, as can be seen by the lexicographic computation tree shown in Figure 39. For example, the leftmost bottom leaf of this tree represents the permutation 214365. The ratio of dead ends and paths in the computation tree to g_{n+1} is not bounded by a constant. Therefore, the algorithm cannot be CAT.

4.3.1 Description of Algorithm

The algorithm is of the backtracking type and fills in positions in the following order. We keep two pointers to the indices of the permutation. One points to even positions, the other to odd. They are initialized to 2 and $2n-1$ respectively. Position 2 is filled first, followed by position $2n-1$. Then each pointer is moved: the even pointer

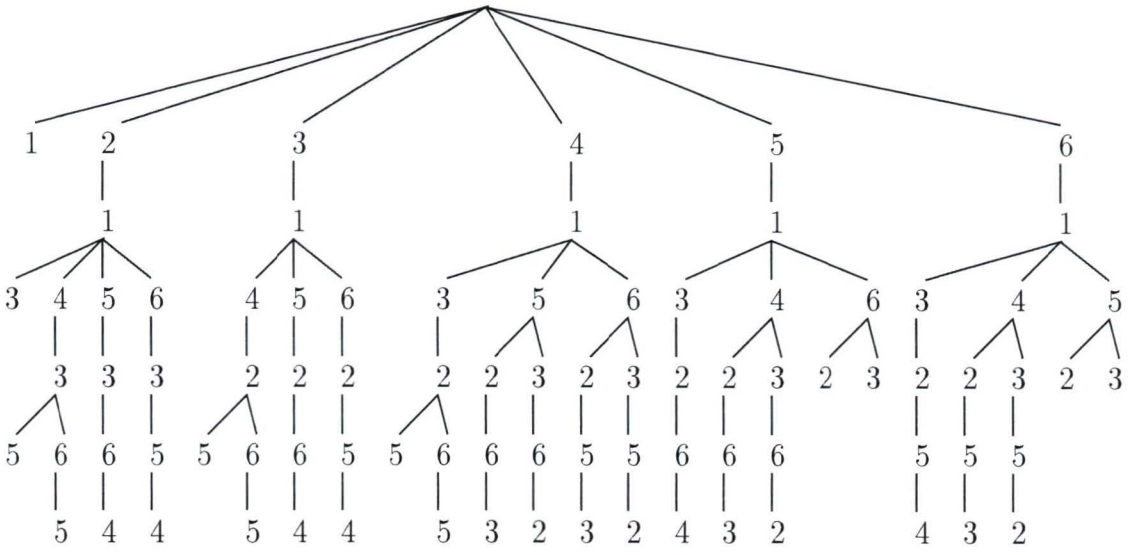

Figure 39 The computation tree for DP2(3) using the naive approach

moves up by two positions and the odd one moves down by two positions This continues until the even pointer reaches position $2n$ and the odd pointer reaches 1 This forward-backward scan is the main idea that allows us to develop a CAT generation algorithm and also the bijection algorithms

We initialize a linked list of the values 2 through $2n$ in increasing order so that the use of values can be remembered The array *perm* of $[1, 2, \quad , 2n]$ is where the permutation is stored

The algorithm, which is given as Figure 40, is used by making a call to Dumont2(1,2) after setting $perm[2]$ to 1 since the first step is always to place a 1 in the 2nd position

The **for** loops at lines (D8) and (D16) require some explanation We can determine which items have not been used by maintaining a doubly linked list initialized

```
Dumont2(position)
(D1)   local i, j
(D2)   if (position = 1) then
(D3)         display the sequence
(D4)         return
(D5)   end if
(D6)   if (position mod2 = 0) then
(D7)         oddPosition ← oddPosition −2
(D8)         for all i ∈ {j | oddPosition ≤ j ≤ n and j not used yet}
(D9)               perm[oddPosition] ← i
(D10)              Dumont2(i, oddPosition)
(D11)              perm[oddPosition] ← nothing
(D12)        end for
(D13)        oddPosition ← oddPosition +2
(D14)  else
(D15)        evenPosition ← evenPosition +2
(D16)        for all i ∈ {j | j < evenPosition and j not used yet}
(D17)              perm[evenPosition] ← i
(D18)              Dumont2(i, evenPosition)
(D19)              perm[evenPosition] ← nothing
(D20)        end for
(D21)        evenPosition ← evenPosition −2
(D22)  end if
```

Figure 40 Dumont2 Algorithm

with the values 1 through $2n$. As a value is used, it is removed from the list and a local variable in the procedure that removed it keeps a pointer to it and the value it should succeed upon reinsertion into the list. When the value must be returned to the list, this will be trivial to accomplish. Using such a list guarantees that we will use only a constant amount of computation per recursive call, since at any call of the procedure, we simply need to use either all unused values $< k$ or all unused values $\geq k$. In the former case, we use every item in the list from the beginning through $k-1$. In the latter, we use every item in the list from the end through k, traversing backwards.

Figure 41 shows the computation tree for the DP2(n) algorithm where $n = 3$. Beside the tree is an indication of which position in the permutation corresponds to that level in the tree. For example, the first leaf in the tree represents the permutation 613254. Note that outdegree 1 nodes occur only at the last two levels. Below, we will show that this is true in general.

4 3 2 Analysis - Proof of Efficiency

The algorithm described above for generating DP2(n) will be shown to be CAT in this subsection.

Theorem 3 *The DP2 algorithm is CAT.*

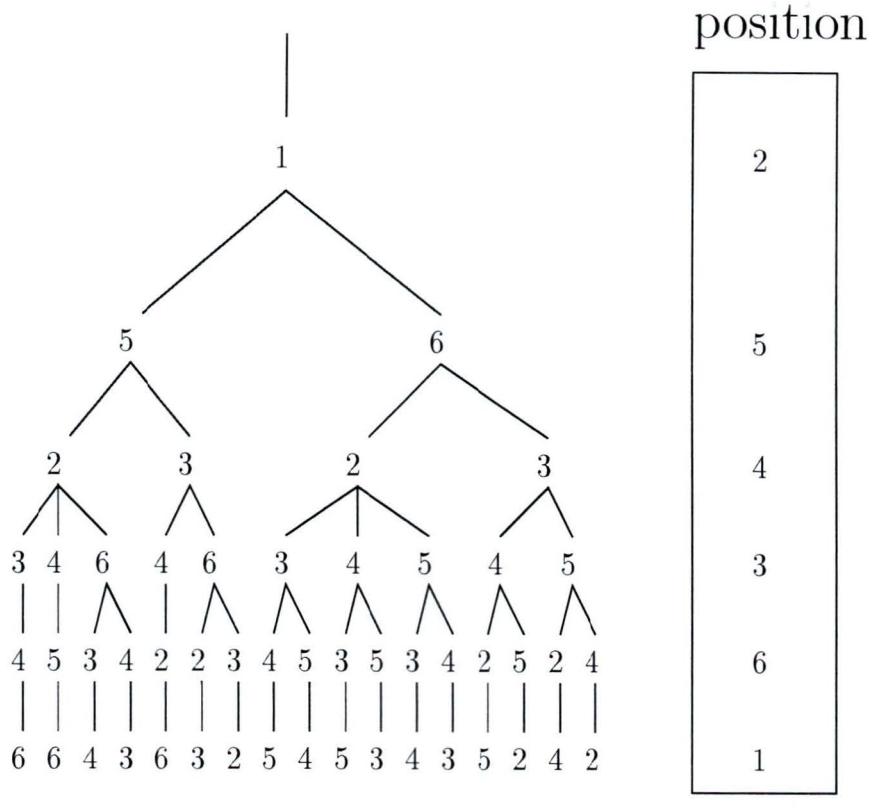

Figure 41 Computation Tree for DP2(3)

Proof Since the computation at each node only involves the placement of a value and a constant amount of time for each valid value for the next position to be filled, it suffices to show that the computation tree for the algorithm has no dead ends, and has at most $2g_{2n}$ nodes of outdegree one. We divide a latter section of the proof into two cases, based on the parity of n.

Define a function π to map the indices into the order in which the algorithm treats them. This function is a permutation. Specifically, $\pi : [2n] \to [2n]$, where if $x \in [2n]$ then

$$\pi(x) = \begin{cases} 2n - x + 1 & \text{if } x \text{ even} \\ x + 1 & \text{if } x \text{ odd} \end{cases}$$

Define α to be a function on $[2n]$ which, given an index value, returns the maximum number of values which can be placed there considering only the restrictions in the definition of DP2(n). Some initial values are $\alpha(1) = 2n$, $\alpha(2) = 1$, $\alpha(3) = 2n - 2$, $\alpha(4) = 3$. In general,

$$\alpha(x) = \begin{cases} x - 1 & \text{if } x \text{ even} \\ 2n - x + 1 & \text{if } x \text{ odd} \end{cases}$$

It is trivial to show that $\alpha = \pi^{-1}$. But π^{-1} is simply a mapping which gives, for each index $x \in [2n]$, in which order that position is filled. Thus we see that the respective order in which each position is filled is the same as the number of items that are eligible to go in each of those positions. In other words, the first position to be filled can only have one possible value, the second position filled has at most two

possible values, and so on.

Define a function β which gives, for each position, the maximum number of values which satisfy both the following conditions (assuming that we fill entries in this function in the order mapped to by π)

1) the value is eligible for that position, and

2) the value could be already taken.

Define one final function δ as follows. For $x \in [2n]$

$$\delta(x) = \alpha(x) - \beta(x)$$

Note that $\delta(x)$ enumerates the lowest possible branching factor in the computation tree when the value for position x is being chosen. This is a computation of the worst-case branching factor at each level of the computation tree. We can show that there are no dead ends and at most $c \cdot g_{2n}$ paths (where c is a constant) by showing that $\delta(x) \neq 0$ for all $x \in [2n]$ and $\delta(x) = 1$ for only a constant number of $x \in [2n]$.

We now consider how to compute $\delta(x)$ in the order given by π. That is, we compute $\delta \circ \pi$ in order.

We compare the sequences produced by $\alpha \circ \pi$ and $\beta \circ \pi$ to compute the sequence $\delta \circ \pi$. The first value filled is always at index $\pi(1) = 2$. $\beta(\pi(1)) = 0$ since none of the values have even been taken yet. Since $\alpha(\pi(1)) = 1$, then $\delta(\pi(1)) = 1$. The

second value gives $\pi(2) = 2n - 1$, $\beta(\pi(2)) = 0$ (since the two values $\geq 2n - 1$ could not yet have been used), $\alpha(\pi(2)) = 2$, and $\delta(\pi(2)) = 2$. Continuing in this fashion, we notice that the even-numbered positions require a value less than the index, and odd-numbered positions require a value greater than or equal to the index. Therefore the computation of β for the even-numbered positions can be computed regardless of what values have been chosen for odd-numbered positions and vice versa until the odd-numbered position is less than the even-numbered position. In other words, every second value of δ will be unaffected by every other second value of δ until the even and odd indexes start to count some of the same items due to overlapping.

There are $\lfloor 2n/4 \rfloor$ even-indexed values which are unaffected by choices for odd-indexed values, and similarly for odd-indexed values.

Consider the sequence produced by $\beta \circ \pi$ over $[2n]$. It starts with $0, 0, 1, 1, 2, 2, \ldots$. The sequence produced by $\alpha \circ \pi$ over $[2n]$is $1, 2, 3, 4, 5, \ldots$. Thus, the sequence produced by $\delta \circ \pi$ begins $1, 2, 2, 3, 3, 4, 4, \ldots$. This pattern continues until the even and odd indexes begin to allow some of the same used items. Exactly when this happens depends upon the value of $n \pmod 2$.

Case 1: $(n \pmod 2 = 0)$

In this case, the overlap begins at position $\pi(n+1)$. The odd and even indices of $\beta \circ \pi$ do not affect each other for $n/2$ or less. Thus, the initial subsequence of $\beta \circ \pi$ is $0, 0, 1, 1, 2, 2, \ldots, n/2 - 1, n/2 - 1$. The top half of Figure 42 depicts this situation,

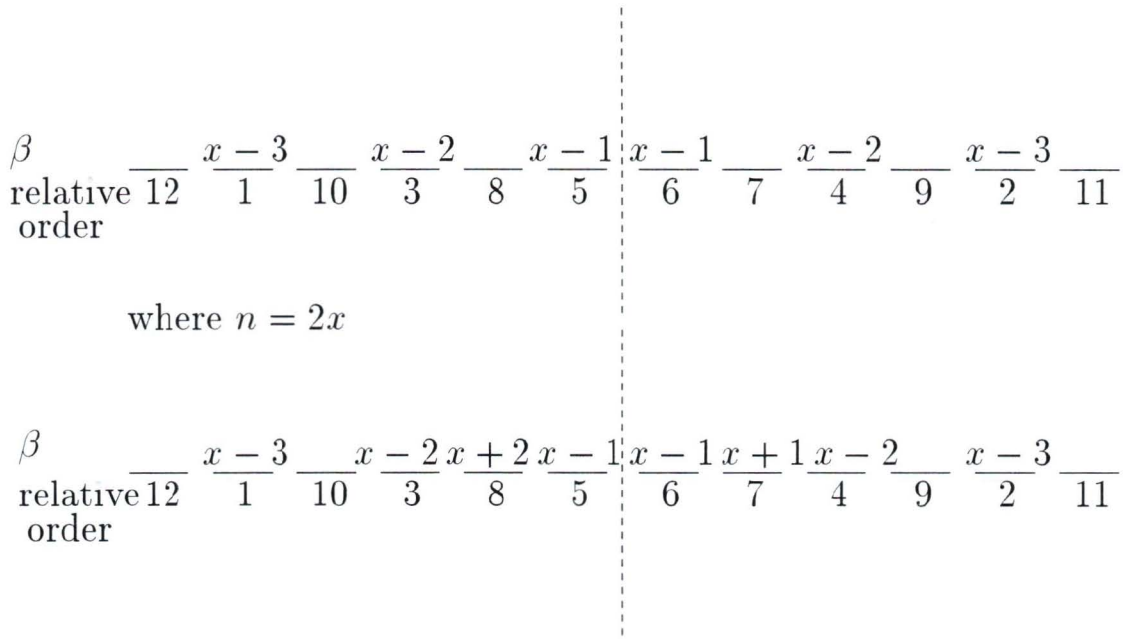

Figure 42 Computation of β for case 1 of DP2 proof

showing the β values for the last 6 indices from $\pi(n-5)$ to $\pi(n)$ inclusive The next β value to be computed is at index $\pi(n+1)$, which corresponds to relative index 7 in the diagram This will count the entry at relative index 5 and the entry at relative index 6 as well as all the entries to the left of that, already determined to be $n/2-1$ Thus, at the bottom half of the diagram, $n/2+1$ is at index 7 Similarly, index 8 has a value 3 greater than index 6 (which is $n/2+2$) because the values at indices 5, 6 and 7 are all new since computing the value for index 6 We can observe that the pattern of adding 3 to the value two indices lower will continue from this point onwards

Thus, we get a recurrence for this second half of the sequence of $\beta(\pi)$, say t_m

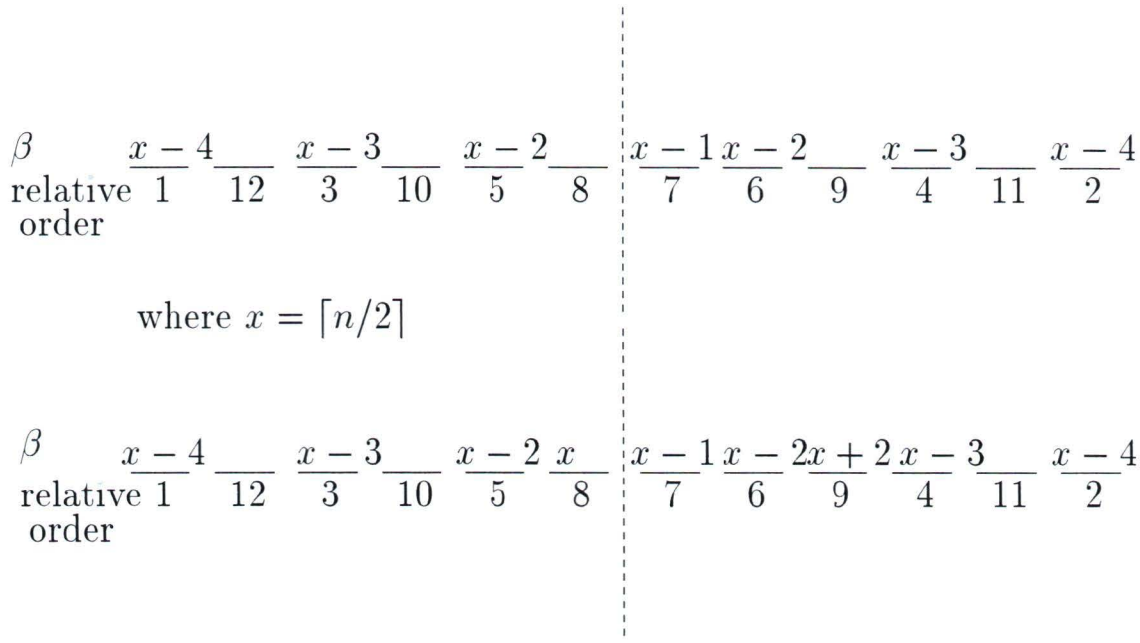

Figure 43 Computation of β for case 2 of DP2 proof

$$t_m = \begin{cases} n/2+1 & \text{if } m = 0 \\ n/2+2 & \text{if } m = 1 \\ t_{m-2}+3 & \text{otherwise} \end{cases}$$

Notice that $t_m = \beta \circ \pi(n+m)$.

For example, when $2n = 16$, $t_m = 5, 6, 8, 9, 11, 12, 14, 15$ for $m = 0, 1, \ldots, 7$. This recurrence relation has the closed form solution

$$t_m = \frac{n}{2} + 3\lfloor \frac{m}{2} \rfloor + 1 + (m \pmod 2)$$

Case 2: $(n \pmod 2 = 1)$

In this case, the overlap also begins at position $\pi(n+1)$. The odd and even indexes of $\beta \circ \pi$ do not affect each other for $n/2$ or less. Thus, the initial subsequence

of $\beta \circ \pi$ is $0, 0, 1, 1, 2, 2, \ldots, \lceil n/2 \rceil - 2, \lceil n/2 \rceil - 2, \lceil n/2 \rceil - 1$. The top half of Figure 43 depicts this situation, showing the β values for the last 7 indexes from $\pi(n-6)$ to $\pi(n)$ inclusive. The next β value to be computed is at index $\pi(n+1)$, which corresponds to index 8 in the diagram. This will count the entry at index 7 and the entry at index 6 as well as all the entries to the right of that, already determined to be $\lceil n/2 \rceil - 2$. Thus, at the bottom half of the diagram, $\lceil n/2 \rceil$ is at index 8. Similarly, index 9 has a value 3 greater than index 7 (which is $\lceil n/2 \rceil - 1$) because the entries at indices 6, 7 and 8 are all new since computing the entry for index 7. We can observe that the pattern of adding 3 to the value two entries lower will continue from this point onwards.

Thus, we get a recurrence for this second half of the sequence of $\beta(\pi)$, say s_m.

$$s_m = \begin{cases} \lceil n/2 \rceil & \text{if } m = 0 \\ \lceil n/2 \rceil + 2 & \text{if } m = 1 \\ s_{m-2} + 3 & \text{otherwise} \end{cases}$$

Notice that $s_m = \beta \circ \pi(n - 1 + m)$.

For example, when $2n = 14$, $s_m = 4, 6, 7, 9, 10, 12, 13$ for $m = 0, 1, \ldots, 6$. This recurrence relation is equivalent to the closed form

$$s_m = n - \lfloor \frac{n}{2} \rfloor + 3\lfloor \frac{m}{2} \rfloor + 2(m \pmod 2)$$

Now we may consider the computation of $\delta \circ \pi$ for both cases. This is just the computation of $\alpha \circ \pi - \beta \circ \pi$, which is $\iota - \beta \circ \pi$, where ι is the identity function. Two

observations can be made regarding this computation from the analysis presented. Firstly, there are only three values of ι for which $\iota - \beta \circ \pi(\iota) = 1$, namely $\iota = 1, 2n-1, 2n$. Secondly, it is evident that for other values $1 < \iota < 2n-1$, we have $\iota - \beta \circ \pi(\iota) > 1$.

We can see from the above argument that the worst-case branching factor is 1 in at most 3 levels in the computation tree - the first one and the last two. This means that in the worst case, there could be only 3 levels of recursive calls where only one choice is possible. The last two levels of the computation tree could each have at most g_{2n} paths. The first level is not counted because it is dealt with in the initialization of the algorithm by placing a 1 in the second position.

We have now established that the computation tree for the algorithm has no dead ends and at most $2g_{2n}$ outdegree one nodes. □

Hence, the algorithm is CAT. It is also BEST.

4.4 A Backtracking Algorithm for Generating DP1(n)

In this section, we develop an algorithm for generating Dumont permutations of the first kind (DP1(n)) and prove that it is CAT.

The first approach we might take is to create a backtracking algorithm that determines the value of each successive position starting at the first. As in the case of DP2(n), this naive approach to backtracking will not give us a CAT algorithm, as is

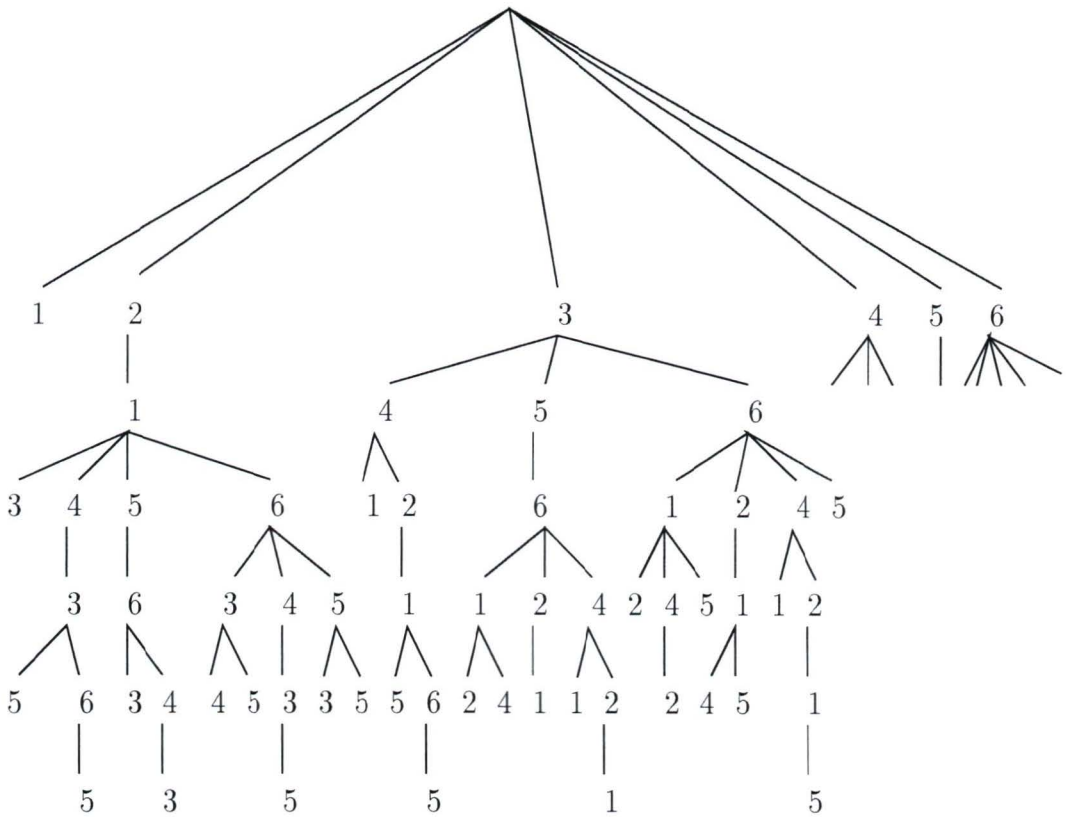

Figure 44 Part of the computation tree for DP1(3) using the naive approach

seen by the partial lexicographic computation tree shown in Figure 44 For example, the first bottom leaf of this tree represents the permutation 214365 The ratio of dead ends and paths in the computation tree to total nodes increases at a rate greater than a constant

4 4 1 Description of Algorithm

The algorithm is of the backtracking type and determines which values follow which other values in the order mapped to by π discussed in the section on DP2's For example, for $2n = 6$, the algorithm will determine what follows 2, then what follows

5, then what follows 4, then what follows 3, then what follows 6, then what follows 1

We initialize a set of $2n+1$ sublists, each containing a number from 1 through $2n$ or $\emptyset$ - an indication that no item follows. These sublists will be merged in pairs throughout the execution of the algorithm to form new sublists which reflect which element follows which. We also keep a list of elements which are available to follow other elements, and call these followers. Since exactly one element is to be last in the permutation, we need to be aware that $\emptyset$ is a follower as well. The initial list of followers should be $1, 2, 3, \ldots, 2n, \emptyset$.

The algorithm is used by making a call to Dumont1(1,2).

The **for** loops at lines (D10) and (D20) require some explanation. We can determine which items have not been used by maintaining a doubly linked list initialized with the values 1 through $2n$. As a value is used, it is removed from the list and a local variable in the procedure that removed it keeps a pointer to it and the value it should succeed. When the value must be returned to the list, this will be trivial to accomplish.

Also, the decision given at lines (D11) and (D21) require some explanation. It must be recognized that a value at the beginning of a sublist can still be in the list of available followers and yet not be available to follow a particular value. If there is such an item, it is at the beginning of a sublist. The particular value it cannot follow is the value at the end of the same sublist. Notice that this special situation only

```
Dumont1(value, position)
(D1)   local i, j
(D2)   adjust sublists so that value follows position
(D3)   if (position = 1) then
(D4)         display the one remaining sublist
(D5)         adjust sublists so that value no longer follows position
(D6)         return
(D7)   end if
(D8)   if (position mod2 = 0) then
(D9)         oddPosition ← oddPosition −2
(D10)        for all i ∈ {j | (oddPosition > j or j = ∅) and j not used yet}
(D11)              if i can follow oddPosition then
(D12)                    remove i from list of available followers
(D13)                    Dumont1(i, oddPosition)
(D14)                    add i back to list of available followers
(D15)              end if
(D16)        end for
(D17)        oddPosition ← oddPosition +2
(D18)  else
(D19)        evenPosition ← evenPosition +2
(D20)        for all i ∈ {j | j < evenPosition and j not used yet}
(D21)              if i can follow evenPosition then
(D22)                    remove i from list of available followers
(D23)                    Dumont1(i, evenPosition)
(D24)                    add i back to list of available followers
(D25)              end if
(D26)        end for
(D27)        evenPosition ← evenPosition −2
(D28)  end if
```

Figure 45 Dumont1 Algorithm

occurs for at most one available follower for any given value. Thus, these decision structures will not affect the time complexity of a single function call of Dumont1.

Having reasoned that the decision structures will not affect the time complexity of one call of the function, we can now consider the time complexity as we did with DP2(n). Using the list of followers in the way described guarantees that we will use only a constant amount of computation per recursive call, since at any call of the procedure, we simply need to use either all unused values $< k$ or all unused values $\geq k$ and $\emptyset$. In the former case, we use every item in the list from the beginning through $k-1$. In the latter, we use every item in the list from the end (which is $\emptyset$) through k, traversing backwards.

Figure 46 shows the computation tree for the DP1 algorithm where $2n = 6$. The notation xfy in this diagram denotes "x follows y". The first leaf in the tree represents the permutation 634215. Note that degree 1 nodes occur only at the last two levels. Below, we will show that this is true in general.

4.4.2 Analysis - Proof of Efficiency

The algorithm described above for generating DP1(n) will be shown to be CAT in this subsection.

Theorem 4 *The DP1(n) algorithm is CAT.*

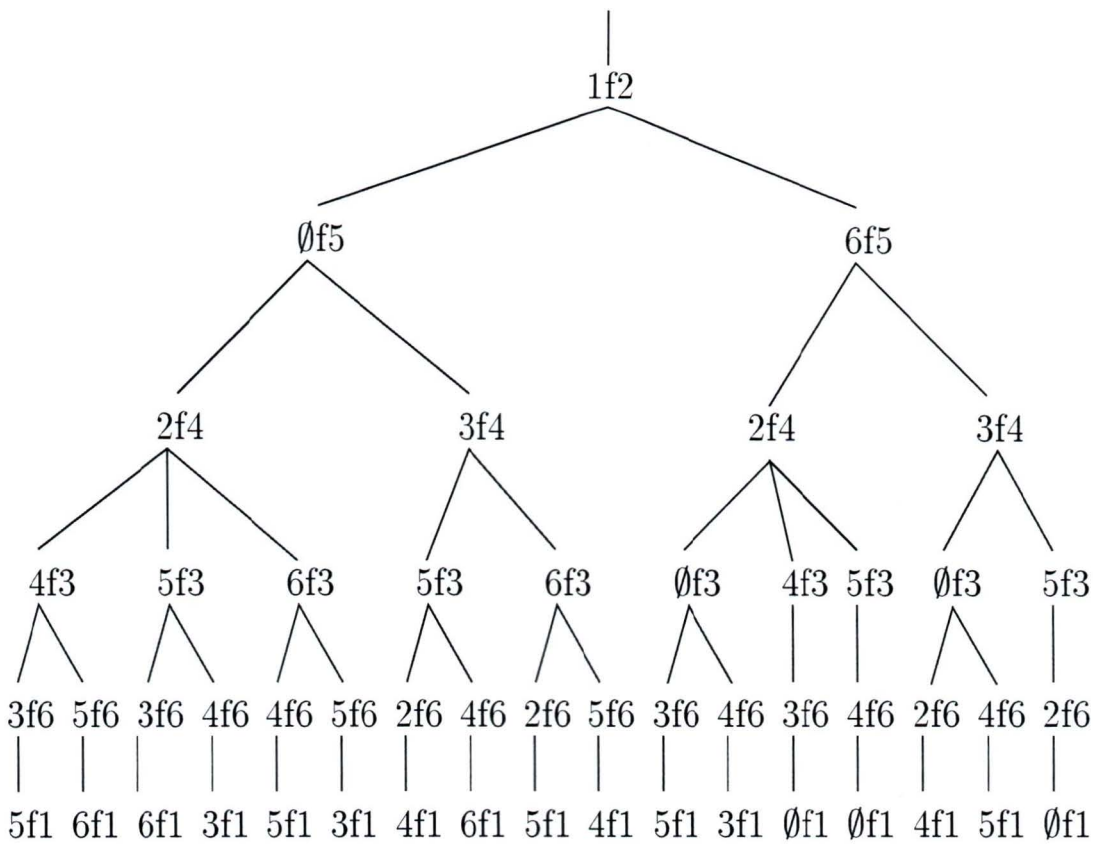

Figure 46 Computation Tree for DP1(3)

Proof Since the computation at each node only involves the placement of a value and a constant amount of time for each valid value for the next position to be filled, it suffices to show that the computation tree for the algorithm has no dead ends, and has at most cg_{2n} outdegree one nodes, where c is a constant. We divide a latter section of the proof into two cases, based on the value of $2n$ mod 4. Since $2n$ is even, this value will be either 0 or 2.

We again use the definition for $\pi : [2n] \to [2n]$, where if $x \in [2n]$ then

$$\pi(x) = \begin{cases} 2n - x + 1 & \text{if } x \text{ even} \\ x + 1 & \text{if } x \text{ odd} \end{cases}$$

We also define α in an analogous way as in the proof for CATness for DP2's. Define α to be a function on $[2n]$ which, given a value, returns the maximum number of different possible values which can be placed after it (including the possibility of no followers) considering only the restrictions in the definition of DP1(n). Some initial values are $\alpha(1) = 2n$, $\alpha(2) = 1$, $\alpha(3) = 2n - 2$, $\alpha(4) = 3$. In general,

$$\alpha(x) = \begin{cases} x - 1 & \text{if } x \text{ even} \\ 2n - x + 1 & \text{if } x \text{ odd} \end{cases}$$

It is trivial to show that $\alpha = \pi^{-1}$. But π^{-1} is simply a mapping which gives, for each index $x \in [2n]$, in which order the followers of values should be chosen. Thus we see that the respective order in which followers are chosen is the same as the number of followers that are eligible to go after each of those values. In other words, the first value to have a follower chosen can only have one possible follower, the second value

for which a follower is chosen has at most two possible followers, and so on.

Define a function β which gives, for each value, the maximum number of values which satisfy both the following conditions (assuming that we assign followers to values in this function in the order mapped to by π).

1) the follower is eligible to go after the value, and

2) the follower could be already taken.

Define one final function δ as follows. For $x \in [2n]$

$$\delta(x) = \alpha(x) - \beta(x)$$

We can show that there are no dead ends and at most $c \cdot g_{2n}$ paths (where c is a constant) by showing that $\delta(x) \neq 0$ for all $x \in [2n]$ and $\delta(x) = 1$ for only a constant number of $x \in [2n]$.

It becomes evident that α, β, and δ behave exactly the same as in the proof of CATness for DP2(n). Thus, by the details of that proof, we have now established that the computation tree for the algorithm has no dead ends and at most $2g_{2n}$ outdegree one nodes. □

n	$usableDP2(n)$	n	$usableDP1(n)$
1	1, 2, 3, 4, 5, 6	1	2, 3, 4, 5, 6, $\emptyset$
2	1	2	1
3	3, 4, 5, 6	3	4, 5, 6, $\emptyset$
4	1, 2, 3	4	1, 2, 3
5	5, 6	5	6, $\emptyset$
6	1, 2, 3, 4, 5	6	1, 2, 3, 4, 5

Figure 47 Available values for each position in a DP2(3) and the corresponding DP1(3) followers

4.5 Conjecture: An explicit mapping between DP2(n) and DP1(n)

We present an algorithm which takes as input a Dumont permutation of the second kind and produces the corresponding Dumont permutation of the first kind. The algorithm uses a list of available values (or followers, as the case may be) for each of the values 1 through $2n$ for both DP1(n) and DP2(n) of size $2n$. We refer to these available choice lists through the use of the arrays *usableDP1* and *usableDP2*. For example, the lists of available values for a DP2(3) and DP1(3) are shown in Figure 47.

It should be noticed that the number of items in a given list of choices for DP2(n) is the same as the number of items in the corresponding list of choices for the DP1(n).

The main idea of the algorithm is to match each value choice of the DP2 that has been input with a choice of follower in building a DP1. When the correspondence of a single value and following value has been determined, all the *usable* lists for both DP2's and DP1's are changed to reflect these choices. This is accomplished by

removing all instances of the value from the *usableDP2* lists and all followers which reflect the choice of follower from the *usableDP1* lists. This last part involves first removing all instances of the follower chosen and then also involves possibly removing a particular follower from a particular $usableDP1$ list. Whenever a new follower is chosen, two sublists are joined. The joining of these two sublists gives a new constraint with respect to followers since the end of this new sublist cannot have the beginning value of the same sublist as its follower. Thus, such a follower is removed in the case where the ending of the sublist is not $\emptyset$.

The input of a DP2 is assumed to be accessed through an array called *dp2*. The setting of followers for the DP1 is assumed to be set by assigning values to the array *dp1* and by merging appropriate sublists as in the algorithm for generating DP1's.

dp1_pointer and *dp2_pointer* are used to search through corresponding lists of choices.

The Convert algorithm is given in Figure 48.

In the above, *beginning* and *ending* denote the first and last values in the latest sublist created.

Consider the input of 415263 into convert. The *usable* lists will initially be as given above. Then the 1 in this DP2 will correspond to choosing a 1 as a follower of 2, producing the modified version of the *usable* lists found in Figure 49.

Next, the 6 in position 5 of our input permutation corresponds to $\emptyset$ as a follower

```
Convert
    local: i, current_pos, dp1_pointer, dp2_pointer
    for all 1 ≤ i ≤ 2n
        current_pos ← π(i)
        dp1_pointer ← usableDP1[current_pos]
        dp2_pointer ← usableDP2[current_pos]
        while dp2[current_pos] ≠ value(dp2_pointer)
            move both dp pointers ahead one node in respective list
        end while
        dp1[current_pos] ← value(dp1_pointer)
        join two dp1 sublists so dp1[current_pos] follows current_pos
        remove all instances of dp2[current_pos] from usableDP2 lists
        remove all instances of dp1[current_pos] from usableDP1 lists
        remove any instance of beginning from dp1[ending]
    end for
```

Figure 48 **Convert** algorithm which takes a DP2(n) and maps it to a DP1(n)

n	$usableDP2(n)$	n	$usableDP1(n)$
1	2, 3, 4, 5, 6	1	3, 4, 5, 6, $\emptyset$
2		2	
3	3, 4, 5, 6	3	4, 5, 6, $\emptyset$
4	2, 3	4	2, 3
5	5, 6	5	6, $\emptyset$
6	2, 3, 4, 5	6	2, 3, 4, 5

Figure 49 First data structure change in mapping the DP2 415263 to a DP1

n	$usableDP2(n)$	n	$usableDP1(n)$
1	2, 3, 4, 5	1	3, 4, 5, 6
2		2	
3	3, 4, 5	3	4, 5, 6
4	2, 3	4	2, 3
5	5	5	6
6	2, 3, 4, 5	6	2, 3, 4, 5

Figure 50 Second data structure change in mapping the DP2 415263 to a DP1

n	$usableDP2(n)$	n	$usableDP1(n)$
1	3, 4, 5	1	3, 5, 6
2		2	
3	3, 4, 5	3	4, 5, 6
4	3	4	3
5	5	5	6
6	3, 4, 5	6	3, 4, 5

Figure 51 Third data structure change in mapping the DP2 415263 to a DP1

of 5. We then get the *usable* lists shown in Figure 50.

Next, the 2 in position 4 of our input permutation corresponds to 2 as a follower of 4. We then get the *usable* lists of Figure 51.

Next, the 5 in position 3 of our input permutation corresponds to 6 as a follower of 3. We then get the *usable* lists of Figure 52.

Next, the 3 in position 6 of our input permutation corresponds to 4 as a follower of 6. We then get the *usable* lists of Figure 53.

We are left with the 4 placed at position 1 of our input corresponding to 5 following 1 in the DP1 we build. By putting together all the sublists based on the

n	$usableDP2(n)$	n	$usableDP1(n)$
1	3, 4	1	3, 5
2		2	
3	3, 4	3	4, 5
4	3	4	3
5		5	
6	3, 4	6	4, 5

Figure 52 Fourth data structure change in mapping the DP2 415263 to a DP1

n	$usableDP2(n)$	n	$usableDP1(n)$
1	4	1	5
2		2	
3	4	3	5
4		4	
5		5	
6	4	6	5

Figure 53 Fifth data structure change in mapping the DP2 415263 to a DP1

followers we have determined, we get 364215.

The correctness of this algorithm is a conjecture and relies on the supposed property that at any given execution of the loop, the number of available choices in corresponding lists is always the same. This conjecture is supported by some experimentation, as follows. All DP2(n)'s were generated and redirected to a file, for some n. Each of these DP2(n)'s were converted using the Convert algorithm, and this output was redirected to a file, which was then sorted. Next, all DP1(n)'s were generated and redirected to a file, which was also subsequently sorted. The two sorted files were compared and found to be the same. This experiment was tried for $n = 2, 4, 6, \ldots, 12$. Recall there are 38227 DP1(12)'s and DP2(12)'s.

input to Convert	output from Convert	input to Convert	output from Convert
2 1 4 3 6 5	2 1 6 4 3 5	2 1 5 3 6 4	2 1 4 3 6 5
2 1 6 3 5 4	2 1 5 6 4 3	3 1 4 2 6 5	6 4 2 1 3 5
3 1 5 2 6 4	4 2 1 3 6 5	3 1 6 2 5 4	5 6 4 2 1 3
4 1 5 2 6 3	3 6 4 2 1 5	4 1 5 3 6 2	4 3 6 2 1 5
4 1 3 2 6 5	3 4 2 1 6 5	4 1 6 2 5 3	4 2 1 5 6 3
4 1 6 3 5 2	5 6 2 1 4 3	5 1 4 3 6 2	6 2 1 4 3 5
5 1 4 2 6 3	4 2 1 6 3 5	5 1 3 2 6 4	6 3 4 2 1 5
6 1 4 3 5 2	4 3 5 6 2 1	6 1 4 2 5 3	3 5 6 4 2 1
6 1 3 2 5 4	5 6 3 4 2 1		

Figure 54 The mapping of DP2(6) to DP1(6)

The mapping in Figure 54 was generated using the Convert algorithm to illustrate the correspondence between DP2's and DP1's of length 6.

The time complexity of the Convert algorithm is of interest. We now provide an analysis of that.

If we count the number of items placed in an array at initialization, assuming that n is the length of the permutations, we find the pattern $n, 1, n-2, 3, n-4, 5, \ldots, 2, n-1$ in attempting to count. If we count these in pairs, we find that each pair has $n+1$ items and there are $n/2$ pairs. Since this is done with two sets of arrays, we get a total of $n(n+1)$ operations for initialization of arrays.

We now consider the number of operations involved in searching through these $n(n+1)$ items for the corresponding values and searching through all the relevant arrays to remove those values and followers which are no longer necessary. Once a decision has been made that an entry in the *dp2* array corresponds to a follower for the

```
Convert
    local i, current_pos, dp1_pointer, dp2_pointer
    for all 1 ≤ i ≤ 2n
        current_pos ← π(i)
        dp1_pointer ← usableDP1[current_pos]
        dp2_pointer ← usableDP2[current_pos]
        while dp1[current_pos] ≠ value(dp1_pointer)
            move both dp pointers ahead one node in respective list
        end while
        dp2[current_pos] ← value(dp2_pointer)
        join two dp1 sublists so dp1[current_pos] follows current_pos
        remove all instances of dp2[current_pos] from usableDP2 lists
        remove all instances of dp1[current_pos] from usableDP1 lists
        remove any instance of beginning from dp1[ending]
    end for
```

Figure 55 Convert algorithm which takes a DP1(n) and converts it to a DP2(n)

DP1(n) being built, this value is removed from all the relevant arrays and the follower is removed likewise However, indices in the DP1 for which the corresponding follower has been determined need not be searched We notice that the order in which we consider indices of the DP1(n) input provides a certain pattern for counting (Recall that we consider indices in the order mapped to by π) The first index considered (2) has 1 element, the second has 2, the third has 3, and so on This continues to the $n-1$st element Thus, if we assume the algorithm ignores previously considered indices when searching through to remove values and followers that are no longer necessary, we ignore at least 1 element the first time, $1+2=3$ elements the second time, $1+2+3=6$ elements the third time, and $a(a+1)$ elements in general each time, where a is the number of times through the loop Since there are $n(n+1)$ items

in total, we want to consider the value of $n(n+1) - a(a+1)$ summed over a from 0 (to consider initialization) to $n-1$. This summation is $O(n^3)$. Thus, we conclude that the worst case running time of Convert is $O(n^3)$.

An algorithm which takes as input a DP1 of length $2n$ and converts it to the corresponding DP2 of length $2n$ can be constructed with a few minor changes to the previous algorithm. This new version of Convert is given in Figure 55.

The running time for this version of the Convert algorithm is the same as the previous one.

5 Conclusion

We have been successful at designing generation algorithms for a variety of permutations characterized by particular restrictions All of the generation algorithms are CAT, meaning that they have a time complexity that is asymptotically the best they can be with a sequential computing model

Bibliography

[1] Comtet, Louis, *Advanced Combinatorics*, D. Reidel Publishing Company, 1974.

[2] Di Francesco, P., Golinelli, O., Guitter, E., Meander, folding and arch statistics. *Special Issue: Combinatorics and Physics, Mathematical and Computer Modelling*, **144**(1996).

[3] Di Francesco, P., Golinelli, O., Guitter, E., Meanders and the Temperley-Lieb algebra. Saclay preprint (1996).

[4] Di Francesco, P., Golinelli, O., Guitter, E., Meanders: a direct enumeration approach. *Nuclear Physics B [FS]*, **482**(1996), 497-535.

[5] Dumont, D., Interprétation combinatoire des nombres de Genocchi. *Duke Mathematics*, **41**(1974), 305-318.

[6] Gardner, Martin, *The Scientific American Book of Mathematical Puzzles and Diversions*, Simon and Schuster, 1959, 1-14.

[7] Graham, R., Knuth, D., Patashnik, O., *Concrete Mathematics: a Foundation for Computer Science*, Addison-Wesley, 1994.

[8] Koehler, John E., Folding a strip of stamps. *Journal of Combinatorial Theory*, **5**(1968), 135-152.

[9] Kreweras, G. Personal communication, 1997.

[10] Kreweras, G. Sur les Permutations Comptées par les Nombres de Genocchi de 1-ière et 2-ième Espece. *European Journal of Combinatorics*, **18**(1997), 49-58.

[11] Kreweras, G. An additive generation for the Genocchi numbers and two of its enumerative meanings. *Bulletin of the ICA*, **20**(1997), 99-103.

[12] Lunnon, W. F., A Map-Folding Problem. *Mathematics of Computation*, **vol. 22**, (1968), 193-199.

[13] Lunnon, W. F., Multi-dimensional map-folding. *Computer Journal*, **vol. 14**, (1971), 75-80.

[14] Phillips, Tony, Math Department SUNY Stony Brook Web page,

`http://www.math.sunysb.edu/~tony/mazes/count.html`

[15] Randrianarivony, A , Zeng, J , Some Equidistributed Statistics on Genocchi Permutations *Electronic Journal of Combinatorics*, **vol 3**, No 2 (1996), 1-11

[16] Reingold, E , Nievergelt, J , and Deo, N , *Combinatorial Algorithms Theory and Practice* Prentice-Hall, 1977

[17] Ruskey, F , *Combinatorial Generation* Unpublished manuscript, 1996

[18] Sedgewick, R , Permutation generation methods *Computing Surveys 9* (1977), 137-164

[19] Sloane, N , The on-line encyclopedia of integer sequences,

`http //www research att com/~njas/sequences/`

VITA

Surname Lausch Given Names Scott Arno

Place of Birth Sudbury, Ontario, Canada

Educational Institutions Attended

Laurentian University 1992 to 1996

Degrees Awarded

B Sc (Honours) Laurentian University 1996

Honours and Awards

NSERC PGS A 1996 to 1998

ASI Graduate Recruitment Assistance Scholarship 1996

Canada Scholarship 1992 to 1996

Falconbridge Scholarship 1992 to 1996

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis

Generating Some Restricted Classes of Permutations

Author [redacted]

Scott Arno Lausch

February 18, 1999