Shift Gray Codes

by

Aaron Michael Williams

B.Math., Combinatorics & Optimization, University of Waterloo, 2001
B.Math., Computer Science, University of Waterloo, 2001
M.Math., Combinatorics & Optimization, University of Waterloo, 2004

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

© Aaron Michael Williams, 2009, University of Victoria.
All rights reserved. This dissertation may not be reproduced in whole or in part,
by photocopying or other means, without the permission of the author.
Chapter 1 is reprinted for your enjoyment.

Aaron (22/12/2009)
Chapter 1

Combinatorial Generation

“It is really quite simple. We have been compiling a list which shall contain all the possible names of God.”

“I beg your pardon?”

“We have reason to believe,” continued the Lama imperturbably, “that all such names can be written with not more than nine letters in an alphabet we have devised.”

“And you have been doing this for three centuries?”

“Yes. We expected it would take us about 15,000 years to complete the task.”

“Oh.” Dr. Wagner looked a little dazed. “Now I see why you wanted to hire one of our machines.”

- The Lama and Dr. Wagner in *The Nine Billion Names of God*

Within Arthur C. Clarke’s classic 1953 short story [10], the Lama is confronted with a monumental task. In his belief system, the possible names of God can be written using at most nine letters from a special alphabet. His goal is to write out each of the nine billion possible names, thereby bringing a satisfactory end to existence. With this sole purpose, and three centuries of work, the inhabitants of his lamasery in Tibet have written only 2% of the possibilities. To fast-track the process, the Lama has traveled to New York to enlist the services of Dr. Wagner, and the *Mark V* automatic sequence computer. By turning over his faith to this machine — capable of thousands of calculations per second — the Lama estimates that his task can be completed within one hundred days.

Despite its fictional nature, Arthur C. Clarke’s short story accurately predicted the research area that provides the topic of this thesis. *Combinatorial generation* uses discrete mathematics and theoretical computer science to achieve its goal of “efficiently creating all possibilities”. To illustrate the type of result contained in this thesis, consider the following operation.

**Reordering a binary string using right-shifts**

Shift the first bit to the right until it passes over a 01 or the last bit.

For example, operation (i) transforms 100100100 into 001100100 since the first bit is shifted to the right until its passes over the first 01. This transformation is illustrated by 100100100 = 001100100. More generally, *right-shifts* are illustrated using right-arrows (i.e., $\overrightarrow{abcdef} = acdeb$). As another example, operation (i) produces 1100000 = 100001 since the first bit of 110000 is shifted past the last bit without every passing over a 01. Despite its simplicity, operation (i) has the property that it eventually reorders the bits of any binary string in all possible ways. For example, the binary strings with three 0s and three 1s are shown below, and each string is obtained by applying the
operation to the previous string

\[
\begin{align*}
110001, & 110010, 100011, 000111, 001011, 010011, 100101, 001101, 010101, 101001, \\
011001, & 110010, 100011, 001101, 010101, 101101, 110100, 101100, 011100.
\end{align*}
\tag{1.1}
\]

The order of strings in (1.1) is known as a right-shift Gray code since successive strings differ by a right-shift. Furthermore, the Gray code is circular since one additional application of operation (i) transforms the last string into the first (011100 = 111000). The strings in (1.1) are known as the (3, 3)-combinations, and in general the binary strings with \( s \) 0s and \( t \) 1s are known as the \((s,t)\)-combinations. The fact that \((s,t)\)-combinations are generated by operation (i) is the most basic corollary of a general theory on shift Gray codes that is developed in this thesis.

While operation (i) has mathematical intrigue, results in combinatorial generation must also be seen through the screen of a computer scientist. To continue our illustration, it is helpful to state the operation that is inverse to operation (i).

**Reordering a binary string using left-shifts**

Shift the bit following the first 01 (or the last bit) into the first position. 

In general, left-shifts are illustrated using left-arrows (i.e., \( \rightarrow abcd ef = aebcdf \)). Operations (i) and (ii) are also known as prefix-shifts. This is because operation (i) right-shifts the first symbol in a string, whereas (ii) left-shifts a symbol into the first position of a string. Notice that the prefix-shift performed by operation (ii) has the effect of simply “undoing” the prefix-shift performed by operation (i). One advantage of operation (ii) is that no scanning is required to perform successive applications. This is because the operation depends only on the position of the first 01, and because the operation changes this position in a predictable pattern. To see this pattern more clearly, consider the order of (3, 3)-combinations that is generated by operation (ii). For each string, the position of the first 01 is underlined. Notice that in each subsequent string, the underlined substring either moves one position to the right, or is reset to the beginning of the string if this string starts with 01.

\[
\begin{align*}
111000, & 101100, 110100, 011010, 101010, 010110, 001110, 100110, 110010, 101001, \\
101001, & 010101, 001101, 100101, 010011, 001011, 000111, 100011, 110001, 111000.
\end{align*}
\tag{1.2}
\]

For this reason, successive applications of operation (ii) can be applied extremely quickly. Another advantage of (ii) is its versatility. In particular, operation (ii) can be efficiently implemented using a wide variety of standard data types including linked lists, arrays, and computer words. Figure 1.1 illustrates this fact, and Chapter 4 includes implementations of the associated algorithms using these data types. Figure 1.1 also shows how operation (ii) can be implemented by hand. The benefit of this manual interpretation is discussed later in this chapter in the context of *The Nine Billion Names of God*.

Despite its fundamental nature, operation (ii) and its algorithmic consequences were only discovered recently (see Ruskey-Williams [70, 73]). The associated cool-lex order of \((s,t)\)-combinations turns out to be a subtle variation of lexicographic order, as seen by Figure 1.2. Furthermore, cool-lex order hides additional properties that may hold interest for discrete mathematicians and theoretical computer scientists. To conclude our initial introduction to the results contained in this thesis, consider the following string of length \( \binom{6}{3} = 20 \)

\[110010110110110000. \tag{1.3}\]
This string is known as a shorthand universal cycle for \((3, 3)\)-combinations. This is because it contains every \((3, 3)\)-combination exactly once as a circular substring, with the proviso that the last (redundant) symbol of is omitted. For example, the first five symbols of (1.3) are 11001, and this substring is shorthand for the \((3, 3)\)-combination 110010. Alternatively, (1.3) encodes the following ordering of \((3, 3)\)-combinations, where each string differs from the previous by shifting the first symbol into the last or second-last position

\[
110011001, 00101111, 01011011, 10110110, 11011000, 01101010, 10101101, 01001111, 01111000, 11000111, 10001111, 00111011, 01100111, \ldots
\]  

(1.4)

To interpret this pattern correctly, notice that (1.3) contains the first bit of each successive strings in (1.4). Similarly, the second, third, fourth, and fifth bits in (1.4) are simply rotations of (1.3). This fascinating pattern can be derived quite prestidigitally from the cool-lex order found in (1.1). The string in (1.3) is also known as a universal cycle for the middle levels. For any given value of \(k\), the middle levels are the binary strings of length \(2k+1\) containing \(k\) or \(k+1\) copies of 1. Prior to this thesis, there was no explicit construction known for these universal cycles.

For thorough coverage of combinatorial generation, the reader is directed to the upcoming volume of *The Art of Computer Programming* by Don Knuth. Named by *American Scientist* as one of the best twelve physical-science monographs of the 20th century, along with other notables such as Albert Einstein’s *The Meaning of Relativity* and Richard Feynman’s *QED*, *The Art of Computer Programming* is considered by many as the preeminent textbook in computer science. Several fascicles of this new volume have been printed [45, 46, 47] and include over 400 pages on the subject of combinatorial generation. In particular, the *Generating all Combinations* fascicle includes cool-lex order for \((4, 4)\)-combinations under the name “suffix-rotated” as well as Knuth’s own MMIX computer word implementation that is “incredibly efficient”. Another resource that will greatly expand the research area upon its release is the aptly named *Combinatorial Generation* textbook by Ruskey [62].
The remainder of this chapter is organized into three sections. Section 1.1 describes four historical foundations of modern combinatorial generation. Section 1.2 then discusses contemporary results in combinatorial generation. Section 1.3 completes the chapter by outlining the new results contained in this thesis. To help motivate the reader, each section relates its material to *The Nine Billion Names of God*.

### 1.1 Historical Foundations

Judging from the title of Clarke’s story, it is likely that the special alphabet used by the Lama contains 13 distinct symbols. However, one question that is left unanswered is the order in which the names are generated by the Mark V. Two engineers, George and Chuck, are sent to Tibet with the task of assembling and then programming the Mark V. The simplest and most obvious choice — but unlikely the most efficient — would have been to program the computer to output the names in lexicographic order. Three other possibilities are also outlined in the upcoming subsections, and collectively they form a historical foundation for combinatorial generation.

#### 1.1.1 Lexicographic Order

“I see. You’ve been starting at aaaaaaaa and working up to zzzzzzzz.”

“Exactly — though we use a special alphabet of our own.”

- The Lama in response to Dr. Wagner in *The Nine Billion Names of God*

Humanity’s most established order is lexicographic order. In lexicographic order, the letters in an alphabet are given a relative order. Then, the relative order of any two words is determined by the relative order of their leftmost differing letters. For example, *aardvark* comes before *aardwolf* in the English dictionary because when comparing the leftmost differing symbols *v* comes before *w* in the English alphabet. Likewise, *sloth* comes before *sloths* because the absence of a letter is assumed to have the lowest order. In a similar manner, lexicographic order is used to compare the relative value of numbers in the Hindu-Arabic number system, except that in this case leading zeros need to be prefixed to ensure that the numbers have the same length. For example, 13 has a lower value than 14. Likewise, 0999 has lower value than 1000. (Alternatively, one can consider numbers to be right-justified with the absence of a symbol having the lowest order, and so 999 is less than 1000.) Furthermore, the same principles have been applied to other objects for thousands of years. Historical examples include Shao Yung’s ordering of binary hexagrams in *I Ching*:

\[
\begin{align*}
\text{IIIIIIII} & \quad \cdots \quad \text{VIII}
\end{align*}
\]

and Yang Hsuing’s ordering of ternary tetragrams:

\[
\begin{align*}
\text{E} & \quad \cdots \quad \text{W}
\end{align*}
\]

Notice that the two philosophers had differing opinions on whether the flat “yang” should have the highest or lowest order, and whether the symbols should be read from top-to-bottom or bottom-to-top. These historical examples were brought to the author’s attention by [62] and [47].

Although lexicographic order is a natural choice for man, it is not necessarily a natural choice for machine. The primary difficulty comes from the problem of roll-over. For example, driving

---

1There are 10.6-billion tridenary (base-13) strings of length nine, but not all require consideration according to the Lama’s belief system.

Online Version (December 23, 2009)
Figure 1.2: An artistic representation of co-lexicographic (above) and cool-lex (below) order for (5,5)-combinations (see [6]). White and black regions represent 0 and 1 respectively. Individual strings are read along a line segment originating from the center, and the first and last strings are at either side of 12 o’clock. Cool-lex proceeds leftwards (counterclockwise) and involves left-shifts, while reverse cool-lex proceeds rightwards (clockwise) and involves right-shifts. Co-lexicographic order proceeds counterclockwise, while reverse co-lexicographic order proceeds clockwise.
one additional kilometer in a vehicle will cause its odometer to roll-over from 299999 to 300000, thereby changing the value of every digit in the process. Similarly, using nine letter words over the Lama’s 13-letter language there would be roll-over from aaaaaaaaaa to baaaaaaa. This worst-case behavior can be the limiting factor in many situations. For example, the Mark series of computers were known for their peculiar timing idiosyncracies, where certain operations could only be performed during certain clock cycles (see Wikipedia [98]). Accordingly, if programmed in lexicographic order, then the Mark V may have been forced to wait for the longest possible update time before outputting each successive name. Depending on its parallelization capabilities, updating each letter could have taken nine times as long as updating a single letter. Furthermore, the frequency of roll-overs could also lead to undue electrical consumption and mechanical wear on the Mark V’s electromatic typewriter (both of these considerations are significant when running a multi-month generator-powered project on a remote Tibetan mountaintop). For this reason, the Manhattan-based engineers may have asked their colleagues at Bell Laboratories for suggestions before embarking on their long voyage to the lamasery.

1.1.2 The Binary Reflected Gray Code

“Once it has been programmed properly it will permute each letter in turn and print the result. What would have taken us fifteen thousand years it will be able to do in a hundred days.”

- The Lama in The Nine Billion Names of God

While working with Bell Laboratories, Frank Gray filed U.S. patent 2,632,058 in 1947 based on the binary reflected code [29]. Gray’s result shows how binary strings of length \( n \) can be ordered so that successive strings differ in exactly one bit. For example, Gray’s ordered list for the binary strings of length \( n = 2 \) is denoted \( G_2 \) and appears below. Notice that it contains every possible binary string of length two exactly once, and that a single overlined bit is changed to obtain each successive string. For example, \( \overline{00} \) is succeeded by \( 01 \).

\[
G_2 = \overline{00}, \overline{01}, \overline{11}, \overline{10} \quad \text{and} \quad \text{reverse}(G_2) = 1\overline{0}, 1\overline{1}, \overline{01}, 0\overline{1}
\]

\[
G_3 = \overline{000}, \overline{001}, \overline{011}, \overline{010}, \overline{110}, \overline{111}, \overline{101}, 0\overline{11}, 0\overline{10}, 1\overline{01}, 1\overline{00}
\]

\[
\begin{align*}
\text{reverse}(G_2) & = 1\overline{0}, 1\overline{1}, \overline{01}, \overline{00} \\
G_2 & = 0\overline{0}, \overline{01}, \overline{11}, \overline{10}
\end{align*}
\]

The term reflected in binary reflected code comes from the operation of reversing the order that each string appears in a list. For example, \( \text{reverse}(G_2) \) starts with the last string in \( G_2 \), namely \( 1\overline{0} \), and ends with the first string in \( G_2 \), namely \( \overline{00} \). Since each successive string in \( \text{reverse}(G_2) \) must also differ in the value of a single bit, this simple operation allows Gray to extend the pattern to binary strings with one more bit. In particular, the ordered list \( G_{n+1} \) is constructed by prefixing \( 0 \) to every string in \( G_n \), followed by prefixing \( 1 \) to every string in \( \text{reverse}(G_n) \). This construction is illustrated above for \( n = 2 \). In general, \( G_0 = \epsilon \) (the empty string) and then for \( n > 0 \),

\[
G_{n+1} = 0 \cdot G_n, 1 \cdot \text{reverse}(G_n).
\]

This type of construction is recursive since it describes the overall structure of the ordered list in terms of smaller ordered lists of the same type. Dr. Wagner may have also been aware that the binary reflected code can be described by a highly efficient iterative operation which describes
how to change any string in the list into the next string in the list (see Bitner-Ehrlich-Reingold [4]). Iteration is often more desirable than recursion due to its low overhead; this is especially true in this case since the available memory is limited to a small number of ferrite magnetic registers. Before discussing how the binary reflected code relates to the nine billion names of God, it is useful to point out that if the second sublist in the above expression is not reflected, then the result is a recursive construction for binary strings in lexicographic order. In other words, the binary reflected Gray code is a subtle variation of lexicographic order.

If George and Chuck were aware of the binary reflected code, then they may have wondered if the same principle could be extended to \( n \)-tuples in general. Given an alphabet, the term \( n \)-tuple refers to the set of all possible strings of length \( n \) over that alphabet. For this application, George and Chuck would have been primarily interested in 9-tuples over a 13-letter alphabet. With some back-of-the-envelope reckoning, they may have realized that reversing every second sublist, that is,

\[
G_{n+1} = A \cdot G_n, \ B \cdot \text{reverse}(G_n), \ C \cdot G_n, \ D \cdot \text{reverse}(G_n), \ldots, M \cdot G_n,
\]

ensures that successive strings differ by the increment, or decrement, of a single letter. For example, \text{AMMMmMMmM} is followed by \text{BMmMmMmM} in this \text{tridenary reflected Gray code}, and in general all roll-overs are avoided. Furthermore, a similarly efficient iterative operation also exists for this generalized notion of a \text{reflected Gray code} [46]. The two Americans could have used this iterative operation as a basis for programming the Mark V, and this approach would lead to significantly faster generation of each successive name (and a faster return home).

Despite its simplicity, or perhaps because of it, the binary reflected code, or \text{binary reflected Gray code} as it is now known, has become an extremely versatile piece of mathematics. Within information and communication technology its uses are wide and varied, with applications including analog-to-digital conversion, error correction, and decreased power consumption in hand-held devices (see Wikipedia [97]). Furthermore, the same order was used in telegraphy by Émile Baudot as early as 1878. It has also been used for other diverse purposes, including the CODACON spectrometer, and appears in research titles ranging from measurement and instrumentation to quantum chemistry (see Betta-Pietrosanto-Scaglione [3] and Sawae-Sakata-Tei-Takarabe-Manmoto [78]). To honor Gray’s popularization of this ubiquitous pattern, the term \text{Gray code} is now synonymous with the concept of \text{minimal-change order}. In other words, a Gray code is an ordering of objects such that successive objects differ in some small prescribed manner.

1.1.3 de Bruijn Cycles

“The second matter is so trivial that I hesitate to mention it - but it’s surprising how often the obvious gets overlooked.”

- Dr. Wagner in \text{The Nine Billion Names of God}

If Dr. Wagner’s PhD was in mathematics, then he may have been aware of \text{de Bruijn cycles}. Published just one year before Frank Gray’s patent, Nicolaas Govert de Bruijn proved that the binary strings of length \( n \) can be packed into one string of length \( 2^n \) (see de Bruijn [13]). For example,

\[
0000100110101111
\]

is a string of length \( 2^4 = 16 \), and it contains each of the 16 binary strings of length \( n = 4 \) exactly once as a substring. For the sake of clarification, the substrings appear in the following order

\[
0000, 0001, 0010, 0100, 1001, 0011, 0110, 1101, 1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000.
\]
Notice that each substring overlaps the previous substring in three bits, and the final three substrings wrap-around from the end of (1.5) to the beginning. Alternatively, each successive binary string in (1.6) is obtained by removing the leftmost bit and inserting a new rightmost bit. Therefore, de Bruijn cycles also produce a type of minimal-change order. Similar de Bruijn cycles also exist for $n$-tuples over any alphabet, including 9-tuples over a 13-letter alphabet. Depending upon the exact details of the Lama’s belief system, the overlapping of potential names may have been acceptable. For example, "AAAAAAAAABA..." may have been an acceptable encoding of the names "AAAAAAAAA", "AAAAAAAAAB", "AAAAAAAAABA", and so on. Allowing this overlap could have drastically reduced the time taken to complete the project. This is especially true considering that the bottleneck in creating lists is often the output of strings, as opposed to the creation of the strings within the computer’s memory. By using a de Bruijn cycle, the engineers would have reduced the amount of output by a factor of nine, not to mention the reduction in downtime required for replacing worn-out parts of the electromatic typewriter.

“All we need to do is to find something that wants replacing during one of the overhaul periods — something that will hold up the works for a couple of days. We’ll fix it, of course, but not too quickly.”

- Chuck in The Nine Billion Names of God

If George and Chuck wanted to use this approach, then they would have needed to be able to efficiently create a suitable de Bruijn cycle. Although it was not part of de Bruijn’s original research, it is possible to induce a de Bruijn cycle for the $n$-tuples over an alphabet by a reduction from lexicographic order of the same words. For the sake of illustration, consider the lexicographic order of binary strings of length four

\[0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111.\]  
(1.7)

The de Bruijn cycle in (1.5) is obtained by concatenating the underlined portions of the strings in (1.7). The underlined portions represent the non-repeating or aperiodic prefix of the corresponding string. For example, the 01 is underlined in 0101 since 0101 can be formed by repeating 01 twice. Likewise, 0000 since 0000 can be formed by repeating 0 four times. Furthermore, a string only has an underlined portion if it is the lexicographically smallest in its rotation set. The rotation set of a string $s$ is denoted by $\bigcirc(s)$ and is a set containing every rotation of the string. For example, 

$\bigcirc(0010) = \{0010, 0100, 1000, 0001\}.$

Within this set, 0001 is the lexicographically smallest. Thus, 0010 does not have an underlined portion in (1.7), but 0001 does. Similarly, 0101 and 1110 also have underlined portions. A very interesting theoretical result discussed in [46] that dates back to the 1930s (see Martin [55]) is the fact that this technique always creates the lexicographically smallest possible de Bruijn cycle. More recent research has shown that this de Bruijn cycle can be generated efficiently (see Fredericksen-Maiorana [21], Fredericksen-Kessler [20], and Ruskey-Savage-Wang [66]). This construction is known as the FKM algorithm.

“A rather more interesting problem is that of devising suitable circuits to eliminate ridiculous combinations. For example, no letter must occur more than three times in succession.”

- The Lama in The Nine Billion Names of God
Roughly speaking, backtracking involves the avoidance of unnecessary work. As mentioned by the Lama in his initial meeting with Dr. Wagner, certain tuples need not be generated, including those that contain three identical letters in succession. Although fewer than a half-billion possibilities have three identical letters in succession, George and Chuck surely would have preferred to spend an extra half-week back in Manhattan than waiting for the Mark V to output these “ridiculous combinations”. While backtracking to avoid these possibilities would be relatively easy using lexicographic order or the tridenary reflected Gray code, it would be significantly more challenging using de Bruijn cycles.

Similar to the binary reflected code, de Bruijn cycles have also proven to have a wide variety of applications, and have been the subject of a number of generalizations referred to as universal cycles as initiated by Chung-Diaconis-Graham [9].

1.1.4 Johnson-Trotter-Steinhaus Order

This, thought George, was the craziest thing that had ever happened to him. Project Shangri-La, some wit back at the labs had christened it.

- in The Nine Billion Names of God

The Shangri-La reference is to a 1937 movie entitled Lost Horizon in which a group of westerners find themselves “trapped” by a High Lama (played by Sam Jaffe) at an idyllic Himalayan Shangri-La. As is the case in the movie, the westerners in this story have concerns about staying too long in the mountaintop paradise. With the Mark V’s job quickly coming to an end, and their transport not arriving for another week, Chuck is worried about the monks’ reaction when the last name is printed and the world does not end. On the other hand, George fears that the monks will re-examine their centuries-old calculations and conclude that the search for His names is not complete.

Just what obscure calculations had convinced the monks that they needn’t bother to go on to words of ten, twenty, or a hundred letters, George didn’t know. One of his recurring nightmares was that there would be some change of plan and that the High Lama (whom they’d naturally called Sam Jaffe, though he didn’t look a bit like him) would suddenly announce that the project would be extended to approximately A.D. 2060.

- George in The Nine Billion Names of God

Although the prospect of generating longer words was daunting to George, the good news was that the Mark V was already programmed to output n-tuples. Thus, assuming he could teach the junior monks to run the scheduled maintenance and repairs, he could then, in theory, make a small change to the program and return home. However, this would not be true if he needed to reprogram the automatic sequence computer to generate a different type of list. Judging from the Lama’s instructions on avoiding consecutive identical letters, George could have wondered if the next task would involve writing out the six billion permutations of the special 13-letter alphabet. The permutations of a set, or alphabet, include every rearrangement of its symbols. For example, the permutations of \{1, 2, 3, 4\} are

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

Of course, the Mark V could be programmed to output the permutations in lexicographic order, as illustrated above. However, similar timing issues would still exist, with AMLKJHGFEDCB rolling-over to BACDEFGHIJKLM in this alternate list of names. Adding to his concern would be the
realization that the minimal-change operation used in the reflected Gray code simply cannot work for permutations. For instance, given the permutation ABCDEFGHIJKLMNOP, consider what happens when a single letter, say B, is changed to another letter, say L. The resulting string, ALCDEFGHIJKLMNOP is not a permutation since it is missing the letter B and contains two copies of L. However, if the original copy of L is changed to B, then the resulting string, ALCDEFGHIJKLMNOP is again a permutation. The net result is that the B and L have been transposed. In general, transpositions are illustrated using line-segments (i.e., abcd = acedbf). Ten years after Clarke’s short story was published, the Johnson-Trotter-Steinhaus minimal-change order for permutation was published [42, 90, 87]. Within this order, each successive permutation differs by an adjacent-transposition, meaning that the transposed symbols are next to each other. For example, the order for \( n = 4 \) appears below

\[
\begin{align*}
1234, & 1243, 1423, 4123, 4132, 1342, 1324, 1324, 3142, 3141, 3412, 3412, \\
& 3421, 3421, 3241, 3214, 2341, 2314, 2341, 4231, 4213, 4213, 2143, 2134.
\end{align*}
\]

Within the above list\(^3\), notice that the largest symbol, 4, sweeps back and forth, with a single pause when it reaches the extreme left and right positions. This idea drives the entire order since there are \( n \) permutations of \( \{1,2,\ldots,n\} \) that can be obtained by inserting the symbol \( n \) into a fixed permutation of \( \{1,2,\ldots,n-1\} \). The sweeping motion exhausts all \( n \) of these possibilities for a permutation of \( \{1,2,\ldots,n-1\} \), and is followed by the single transposition that creates the next permutation of \( \{1,2,\ldots,n-1\} \). Historically, the Johnson-Trotter-Steinhaus order was influential to the development of combinatorial generation in academia, and by the 1970s there were several survey papers discussing the merits of various methods for generating permutations (see Ord-Smith [56, 57], Sedgewick [79], and Roy [61]). Generating permutations by transpositions was also important to the private sector by this time, as evidenced by an internal memorandum by Goldstein-Graham [27] from Bell Laboratories in 1964.

Although George could not look into the future for help, he could ask for divine intervention. Such a request may have reminded George of Sunday morning, and the wonderfully intricate patterns of music played in many churches. The art, and science, of method ringing was developed in English church towers during the 17th century (see Wikipedia [100]). The oversized tuned bells in these towers were not particularly adept at creating melodies. For this reason, the ringers would focus on sounding all of the bells in some order, one after another. Such orders are called rounds, and are essentially permutations of \( \{1,2,\ldots,n\} \), where \( n \) is the total number of bells and ringers. If the ringers wished to play two rounds in succession, then the second round would have to be played in a similar order to the first. For example, the succession of the following two rounds would not be feasible

\[
\begin{align*}
\begin{array}{ccccccccccc}
\text{Round 1:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{Round 2:} & 1 & 6 & 3 & 4 & 5 & 2 & 7 & 8
\end{array}
\end{align*}
\]

This is due to the fact that in the second round, \( \text{\textbullet}_6 \) would still be completing its ringing cycle from the first round. For this reason, the study of change ringing focuses on playing successive rounds such that a small number of adjacent-transpositions occur. (The adjacent-transposition operation also ensures that each ringer maintains a relatively consistent cadence.) For example, the succession of the following two rounds would be possible in change ringing.

\[
\begin{align*}
\begin{array}{ccccccccccc}
\text{Round 1:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{Round 2:} & 1 & 3 & 2 & 4 & 5 & 7 & 6 & 8
\end{array}
\end{align*}
\]

\(^3\)This list was obtained from Frank Ruskey’s The Combinatorial Object Server [81] where one can find many orders including others from this thesis.
One of the biggest goals in change ringing is to avoid repeated rounds, and method ringing is focused on using mathematical patterns for this purpose. If every possible round is played exactly once, then the performance is called an *extent*. Extents with \( n = 7 \) are called *peals*, and the \( 7! = 5040 \) distinct rounds are played thousands of times per year. For this reason, George’s prayers could have been answered one Sunday morning in New York at a holy “bell lab”. In particular, the Johnson-Trotter-Steinhaus order may have been played as a send-off for George’s Tibetan adventure. A verbose description of *plain change* order and its variations appear in the 1668 text *Tintinnalogia* by Duckworth-Stedman [15] subtitled *Wherein is laid down plain and easie Rules for Ringing all sorts of Plain Changes*.

### 1.2 Contemporary Results

The historical foundations of combinatorial generation were surveyed in the previous section. To motivate these foundations, *The Nine Billion Names of God* was used to demonstrate the utility of the reflected Gray code and de Bruijn cycles for \( n \)-tuples, as well as the Johnson-Trotter-Steinhaus order for permutations. Implicit in the discussions of these minimal-change orders was a particular sequence of three steps designed to achieve the goal of “efficiently creating all possibilities”. This sequence of steps includes:

1. Model the possibilities by a suitable combinatorial object.
2. Find a minimal-change order for the instances of the combinatorial object.
3. Implement an efficient algorithm that generates the minimal-change order.

(Within the last two steps, a universal cycle is included in the possible minima-change orders.) The primary purpose of this section is to survey contemporary results in combinatorial generation, and it does so by following the three steps listed above. Section 1.2.1 introduces several additional combinatorial objects including necklaces and trees, as well as the concept of a fixed-content language. Sections 1.2.2 and 1.2.3 present contemporary transposition Gray codes and shift Gray codes for fixed-content languages, as well as the new concept of a shorthand universal cycles for fixed-content languages. Section 1.2.4 then completes the sequence of steps by discussing known algorithms, and the measures of efficiency that are used in this thesis.

The secondary purpose of this section is to illustrate that optimization problems can also be solved by following the same sequence of three steps. In an *optimization problem*, each instance of a combinatorial object has an associated *value*, and the goal is to find an instance with the greatest value. One way to solve an optimization problem is to generate each possibility and *evaluate* it. This approach is often (derisively) referred to as a *brute force solution* within computer science. Despite this fact, there are many real-world situations when this approach is either necessary or desirable. For example, brute force is often employed when the underlying problem is extremely difficult. Section 1.2.5 returns to the lamasery to introduce a notoriously difficult problem known as the stacker-crane problem. Discussions from Sections 1.2.1-1.2.4 contribute to an understanding of how to implement a highly-optimized brute force solution to the stacker-crane problem. In particular, this optimized brute force solution provides a concrete real-world application of the shift Gray codes and efficient algorithms developed in this thesis.

### 1.2.1 Combinatorial Objects

“What source of electrical energy have you?”
“A diesel generator providing 50 kilowatts at 110 volts. It was installed about five years ago and is quite reliable. It’s made life at the lamasery much more comfortable, but of course it was really installed to provide power for the motors driving the prayer wheels.”

- Dr. Wagner and the Lama in The Nine Billion Names of God

To model the possible solutions of a given problem, it is helpful to first have an understanding of several basic combinatorial objects. Thus far, the reader has been introduced to \(n\)-tuples and permutations. This section expands this list of combinatorial objects by introducing necklaces and trees. This section also discusses the value of a combinatorial object based on its pairs of adjacent symbols. Before beginning this discussion, it is useful to note that the term combinatorial object is often used to refer to a specific instance of a combinatorial object (i.e., a specific permutation of \(\{1,2,\ldots,n\}\)) or to the set of all such instances (i.e., the set of all permutations of \(\{1,2,\ldots,n\}\)). In other words, “combinatorial object” is a figure of speech known as a synecdoche. More precisely, a synecdoche occurs whenever a specific term is used to refer to a more general term, or a general term is used to refer to a more specific term (see dictionary.com [36]).

In his initial meeting with Dr. Wagner, the Lama mentions that a diesel generator was installed at the lamasery to provide power for the motors that automatically turn their prayer wheels. Although prayer wheels are most often adorned with the mantra Om Mani Padme Hum, it is also common to inscribe their outer surface with the Eight Auspicious Signs (see Wikipedia [101]). Although the precise order can differ around the world, in Tibet the Ashtamangala typically refers to the following order of these symbols: endless knot, lotus flower, victory banner, wheel of Dharma, treasure vase, golden fish pair, parasol, and conch shell. To provide additional karma in advance of the Lama’s trip to New York, it would not have been surprising if the lamasery dedicated a prayer wheel to each of the possible circular orderings of the Eight Auspicious Signs. More abstractly, this section describes these possible inscriptions on the lamasery’s prayer wheels as the necklaces containing eight beads of different colours.

To formalize the definition, recall the notion of string rotations from Section 1.1.3. A necklace is an equivalence class of strings under rotation. By convention, the symbols in a necklace are known as coloured beads. For example, the necklaces containing two black beads, two grey beads, and two white beads are illustrated in Figure 1.3. Each necklace in Figure 1.3 can be represented by a string of symbols by mapping \(\bullet \leftrightarrow 1\), \(\circ \leftrightarrow 2\), \(\bigcirc \leftrightarrow 3\), and then by recording one of its clockwise rotations. In particular, it is customary to use the lexicographically largest or smallest clockwise rotation. For example, the lexicographically largest representation of the necklace in the top-left corner of Figure 1.3 is 332211.

![Figure 1.3: Necklaces containing two black, two grey, and two white beads.](image)

4There are 5040 such possibilities since the endless knot can be followed by any of the 7! permutations of the remaining symbols.
While necklaces can be used to encapsulate circular orders, it can also be desirable to encapsulate hierarchical orders. For example, the High Lama would have tutored several students at the lamasery over the years, and then these students would have themselves tutored additional students, and so on. The resulting student-teacher relationship would form a type of family tree. This thesis will focus on ordered trees with a fixed-branching sequence in Chapter 2. For the sake of illustration, Figure 1.4 depicts the ordered trees containing a single node with three children, two nodes with a single child, and three leaves (in black). Each tree in Figure 1.4 can be represented by a string of symbols by mapping each node to its number of children, and then by recording its pre-order traversal. For example, the leftmost tree in Figure 1.4 can be represented by 311000.

Although the combinatorial objects in these two figures are quite different, they are similar in at least one respect. To formalize this connection, say that a multiset is a set that allows repetition, and a multiset permutation is a permutation of a multiset. Notice that each necklace in Figure 1.3 can be encoded as a permutation of the multiset \{1,1,2,2,3,3\}, and each tree in Figure 1.4 can be encoded as a permutation of the multiset \{0,0,1,1,1,3\}. As a third example, the set of all \( \frac{4!}{1!2!1!} = 12 \) multiset permutations over \{1,2,2,3\} appears below

\[
\{1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221\}. \tag{1.8}
\]

Multiset permutations generalize both permutations and \((s,t)\)-combinations. In particular, permutations are the multiset permutations of the set \{1,2,\ldots,n\}, and \((s,t)\)-combinations are the permutations of the multiset containing \(s\) copies of 0 and \(t\) copies of 1. More generally, a fixed-content language is any set of strings that contain the same multiset of symbols. In particular, fixed-content languages can be used to represent the necklaces in Figure 1.3, the trees in Figure 1.4, and the set of multiset permutations found in (1.8). On the other hand, \{122,112\} is a simple example if a language that does not have fixed-content.

A fixed-content language whose strings contain only 0s and 1s is known as a fixed-density language with density referring to the number of 1s. In other words, fixed-density languages are subsets of \((s,t)\)-combinations.

As previously mentioned, each instance of a combinatorial object can have an associated value. (This is not the case in The Nine Billion Names of God since each \(n\)-tuple has equal importance as a possible name of God.) For example, if a combinatorial object is represented by a string, then its associated value could depend on its pairs of adjacent symbols. Given a string \(s_1s_2\cdots s_n\), the ordered pairs of adjacent symbols and unordered pairs of adjacent symbols of \(s\) are respectively

\[
\begin{align*}
\text{ordered pairs of adjacent symbols} & = \{s_1s_2, s_2s_3, \ldots, s_{n-1}s_n\} \\
\text{unordered pairs of adjacent symbols} & = \{s_1, s_2\}, \{s_2, s_3\}, \ldots, \{s_{n-1}, s_n\}.
\end{align*}
\tag{1.9}
\]

For simplicity, these terms are henceforth abbreviated to ordered pairs and unordered pairs.

Another important consideration is the representations of a combinatorial object. A representation of a combinatorial object is a data type that can be used to store the object, together with a convention for how to store the object in this data type. For example, this section section discussed...
how certain necklaces and trees can be represented by fixed-content languages using the convention of a lexicographically largest rotation and a pre-order traversal, respectively. As another example, Figure 1.1 showed that there are simple conventions for representing \((s,t)\)-combinations in an array, linked list, or computer word. In some situations, the representation is fixed by the application at hand.

Before concluding this section, an important point involving fixed-content languages must be raised. Many combinatorial objects are naturally represented by sets of strings that do not have fixed-content. On the other hand, many of these combinatorial objects have subsets that are naturally represented by fixed-content languages. At first it may seem to be more useful to have a minimal-change order for the unrestricted language that does not have fixed-content. However, minimal-change orders for the restricted fixed-content languages are often more useful. This is due to the fact that minimal-change orders for the fixed-content subsets can often be combined into minimal-change orders for the superset. Conversely, it is much less likely that a minimal-change order for the superset can be divided into minimal-change orders for the fixed-content subsets. This point is not discussed again until Chapter 5, but it should be considered a primary motivation for exploring the existence of minimal-change orders for fixed-content languages.

1.2.2 Minimal-Change Orders

“Your Mark V computer can carry out any routine mathematical operation involving up to ten digits. However, for our work we are interested in letters, not numbers. As we wish you to modify the output circuits, the machine will be printing words, not columns of figures.”

- The Lama speaking to Dr. Wagner in *The Nine Billion Names of God*

No discussion of minimal-change orders is complete without mention of a general result by Sekanina [80]. The result states that the cube of any connected graph has a Hamiltonian path. (A Hamiltonian path is a path that travels through each vertex of a graph exactly once, and the cube of a graph is obtained by adding an arc from node \(u\) to node \(v\) so long as the shortest path from \(u\) to \(v\) is at most three.) Therefore, if \(\sigma\) is some operation on strings, and if \(L\) is a set of strings in which any string can be transformed into any other string by repeated applications of \(\sigma\), then there exists an ordering of the strings in \(L\) such that at most three applications of \(\sigma\) are necessary to transform any string into the next string. In general, a minimal-change order using at most \(k\) applications of \(\sigma\) between successive objects is known as a \(k-\sigma\) Gray code for \(L\). (When \(k = 1\) the term is shortened to a \(\sigma\) Gray code.) The aforementioned \(3-\sigma\) Gray code due to Sekanina can be obtained by a prepostorder traversal of any spanning tree in the initial graph (see [47] for further details).

Operations

Before he could finish the sentence, the Lama had produced a small slip of paper.

“This is my certified credit balance at the Asiatic Bank.”
“Thank you. It appears to be–ah–adequate.”
- The Lama and Dr. Wagner in *The Nine Billion Names of God*

The most important measures of a minimal-change order are the type of operation it uses, and its number of applications required to create successive instances of the underlying combinatorial object within the minimal-change order. In general, no single operation is more useful than another. The reason for this fact has to do with the different representations of a combinatorial object. For example, if an application forces the combinatorial object to be stored in an array, then a minimal-change order involving transpositions will likely result in a faster algorithm than a minimal-change order involving shifts. On the other hand, an algorithm based on linked-lists may be more efficient if it is based on a minimal-change order involving shifts. In general, the relative expense of an operation is dependent on the representation of the underlying combinatorial object. On the other hand, it is almost always desirable to reduce the number of application required to create successive instances.

A second expense for an operation arises when the instances of a combinatorial object have an associated value. To illustrate this point, let us compare the expense of transpositions and shifts with respect to the ordered and unordered pairs described in (1.9). To aid in the comparison, another natural operation on strings is briefly considered. A substring-reversal, or simply reversal, replaces a substring with its reversal. This operation is illustrated using bi-directional arrows (i.e., $abcdef \rightarrow acedbf$). Notice that an adjacent-transposition is a special case of a left-shift, right-shift, and substring-reversal, as illustrated below

$$123456 = 123456 = 123456 = 123456 = 124356.$$ (Substring-reversal Gray codes are not covered in detail in this thesis, although it is mentioned that combinatorial generation using this operation has applications to computational biology [83], and an elegant result for generating permutations by substring-reversals was discovered by Zaks [104].)

When applying transpositions and shifts to a string, only a small number of ordered and unordered pairs can change. For example, an adjacent-transposition $\ldots i-1i+1\ldots$ will change at most three ordered pairs, and at most two unordered pairs. (In particular, the ordered pairs $s_{i-1}s_i$, $s_is_{i+1}$, and $s_{i+1}s_{i+2}$ are replaced by the ordered pairs $s_{i-1}s_{i+1}$, $s_{i+1}s_i$, and $s_is_{i+2}$.) Similarly, a prefix-shift $\ldots i-1i\ldots$ will change at most two pairs, regardless of whether the pairs are ordered or unordered. (In particular, the ordered pairs $s_{i-1}s_i$ and $s_is_{i+1}$ are replaced by the ordered pairs $s_is_1$ and $s_{i-1}s_{i+1}$.) A substring-reversal $\ldots i-1i\ldots$ can change at most two unordered pairs. (In particular, the unordered pairs $\{s_{i-1}, s_i\}$ and $\{s_j, s_{j+1}\}$ are replaced by the unordered pairs $\{s_{i-1}, s_1\}$ and $\{s_i, s_{j+1}\}$.) On the other hand, the number of ordered pairs that are changed by a substring-reversal is bounded only by the length of the string. (In particular, $\ldots i-1i\ldots$ changes all $n-1$ ordered pairs.) These bounds are presented in Table 1.1, along with similar bounds for general shifts and transpositions. Table 1.1 is discussed again in Section 1.2.4. In general, this type of analysis provides an important motivation for developing different types of Gray codes for the same combinatorial object.

<table>
<thead>
<tr>
<th></th>
<th>adjacent-transposition</th>
<th>transposition</th>
<th>prefix-shift</th>
<th>shift</th>
<th>reversal</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ordered</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>n-1</td>
</tr>
</tbody>
</table>

Table 1.1: The maximum number of ordered and unordered pairs that are changed after applying various operations to a string of length $n$. 

Online Version (December 23, 2009)
Before concluding this section it is mentioned that many of the shift Gray codes presented in this thesis can also be viewed as transposition Gray codes. For example, the left-shift Gray code of \((3,3)\)-combinations given in (1.2) is expressed below using transpositions

\[
\begin{align*}
011100, & \quad 101100, 110100, 011010, 101010, 010110, 001110, 100110, 110010, 011001, \\
101001, & \quad 010101, 001101, 100101, 010011, 001011, 100011, 110001, 111000.
\end{align*}
\]

Notice that the above strings are in a 2-transposition Gray code. For this reason, (1.2) is both a left-shift Gray code and a 2-transposition Gray code. More generally, operations (i) and (ii) can always be described by transposing one or two pairs of bits. Furthermore, this is true for the general result on shift Gray codes for fixed-density languages developed in this thesis.

### Transposition Gray Codes

Given the Johnson-Trotter-Steinhaus order, it becomes natural to ask which other fixed-content languages can be generated by single adjacent-transpositions. There are some very basic obstructions to this goal, so the question was later expanded to single transpositions, and then to a constant number of transpositions. For example, it is not possible to generate \((2,2)\)-combinations using single adjacent-transpositions. As with many results in combinatorial generation, it is helpful to frame the argument in terms of graph theory. Consider the adjacent-transposition graph in Figure 1.5 which contains a vertex for every \((2,2)\)-combination and an edge for every pair of vertices whose strings differ by an adjacent-transposition. Since there is no Hamilton path in the graph there is no adjacent-transposition Gray code for \((2,2)\)-combinations.

![Figure 1.5: Adjacent-transposition graph for \((2,2)\)-combinations.](image)

More generally, a parity argument by Ruskey [64] shows that \((s,t)\)-combinations have an adjacent-transposition Gray code if and only if \(s\) and \(t\) are both odd. On the other hand, it is possible to relax the adjacent-transposition operation and create a transposition Gray code for combinations. For example, the relative order of the \((s,t)\)-combinations within the binary reflected code on \(n = s + t\) bits is a transposition Gray code (see Liu-Tang [54]) that can be generated by an efficient algorithm (see Ehrlich [4]). This order is illustrated below for \(s = 2\) and \(t = 2\), where strings are crossed out if they are not \((2,2)\)-combinations

\[
\mathcal{G}_4 = 0000, 0001, 1000, 0100, 1100, 0010, 1010, 0110, 1110, 1001, 1101, 0001, 0011, 0111, 1111, 0101, 1101, 1001, 0001.
\]

In general, the transposed symbols in this Gray code are arbitrarily far apart. However, it is possible to create a two-close transposition Gray code for \((s,t)\)-combinations, in which each transposition involves symbols that are separated by at most one other symbol (see Ruskey [63] and Chase [8]). More generally, efficient algorithms and two-close transposition Gray codes exist for \(k\)-ary Dyck words (see Vajnovszki-Walsh [94]). Further results along this line are contained in the Generating All Trees fascicle of The Art of Computer Programming [47].
Transposition Gray codes also exist for multiset permutations. These Gray codes are the basis of efficient algorithms (see Ko-Ruskey [48], Takaoka [89], Vajnovszki [92], and Korsh-LaFollette [52]). Multiset permutations and balanced parentheses can be simultaneously generalized to linear-extensions of partially ordered sets, which are discussed in Section 2.3. In this case transposition Gray codes exist in some cases (see Pruesse-Ruskey [58], [64], and Stachowiak [85]), but not all cases (see Pruesse-Ruskey [60]). In all cases, Canfield-Williamson [7] showed that a constant number of transpositions can be used to create efficient algorithms for generating them. Further generalizations have been considered including basic words of anti-matroids in which 2-transposition Gray codes (using one or two transpositions between successive objects) are known to exist by Pruesse-Ruskey [59].

Although there are no known Gray codes for multiset necklaces, an efficient lexicographic algorithm is known (see Sawada [75]). Furthermore, when the multiset is restricted to contain only 0s and 1s then transposition Gray codes (see Wang-Savage [96] and Ueda [91]) and efficient generating algorithms (see Ruskey-Sawada [67, 68]) are known to exist, although the representatives used in this case are not always lexicographically smallest or largest. (Multiset necklace languages are also known as fixed-content necklace languages, and fixed-density necklace languages refer to the binary case.) Efficient algorithms for generating unlabeled necklaces (which allow the symbols to be permuted) exist (see Ruskey-Sawada [69]), binary necklaces (which have no density restriction) do not have a Gray code changing a single bit [96] but do have a Gray code changing at most two bits (see Vajnovszki [93]), and Gray codes changing at most three symbols also exist for unrestricted necklaces over arbitrary bases (see Weston-Vajnovszki [95]).

### Shift Gray Codes

In general, shift Gray codes have received much less attention from the academic community. Excluding the aforementioned results involving adjacent-transpositions, shift Gray codes were previously known for multiset permutations (see Korsh-Lipschutz [53]) and linear extensions of partially ordered sets (see Korsh-LaFollette [51]). These Gray codes have efficient algorithms, although the implementations span several pages with multiple instructions on each line. Prefix-shift Gray codes for permutations were also known to exist (see Langdon [25, 26] and Corbett [12] and [40]).

#### 1.2.3 Universal Cycles

To understand the relationship between fixed-content languages and universal cycles, consider the problem of constructing a universal cycle for the six permutations of \{1, 2, 3\}. Since the universal cycle must contain 321, then it is safe to assume that the alleged universal cycle is of the form 321xyz where \(x, y, z \in \{1, 2, 3\}\). Within this alleged universal cycle, \(21x\) is a substring, and therefore \(21x\) must be a permutation of \{1, 2, 3\}. Thus, \(x = 3\). Similarly, \(1xy = 13y\) is a substring of the universal cycle and so \(y = 2\). Likewise, \(z = 1\). However, 321321 is certainly not a universal cycle for the permutations of \{1, 2, 3\} since it contains two copies of 321 and no copies of 123. In general, universal cycles for fixed-content languages rarely exist. (More precisely, they exist if and only if the language is comprised of every rotation of a single string.)

To get around this limitation, one can use an alternate representation for each permutation. For example, the six-digit string below on the left is an order-isomorphic universal cycle for the permutations of \{1, 2, 3\}. A string is order-isomorphic to a permutation of \{1, 2, \ldots, n\} if the string
contains \( n \) distinct integers whose relative orders are the same as in the permutation.

\[
\begin{array}{ccc}
321341 & 321, 213, 134, 341, 413, 132 & 321, 213, 123, 231, 312, 132 \\
\text{order-isomorphic universal cycle} & \text{substrings} & \text{permutations}
\end{array}
\]

In particular, the substrings of the order-isomorphic universal cycle appear above in the middle, and the corresponding permutations appear above on the right. In the above example, one additional symbol was required since the symbols in \( \{1, 2, 3, 4\} \) were used for the permutations of \( \{1, 2, 3\} \). Johnson [41] recently confirmed a long-standing and difficult conjecture [9] by showing that one additional symbol is sufficient for making order-isomorphic universal cycles for the permutations of \( \{1, 2, \ldots, n\} \). One drawback of using order-isomorphism is that the resulting permutations can vary significantly from one to the next.

This thesis introduces the idea of using shorthand isomorphism. The shorthand representation of a permutation is simply the permutation with its last symbol removed. For example, the shorthand representation of 12345 is 1234. An example of a shorthand universal cycle for the permutations of \( \{1, 2, 3\} \) appears below, along with its substrings of length two and the permutations that result from suffixing the last “missing” symbol

\[
\begin{array}{ccc}
321312 & 32, 21, 13, 31, 12, 23 & 321, 213, 132, 312, 123, 231 \\
\text{shorthand universal cycle} & \text{substrings} & \text{permutations}
\end{array}
\]

Notice that the resulting permutations differ from one another in a very predictable fashion. In particular, each successive permutation differs by shifting the first symbol to the right past all of the other symbols, or past all of the other symbols except one. For example, if 12345 is a substring of a shorthand universal cycle for the permutations of \( \{1, 2, 3, 4, 6\} \) then the next symbol must either be 1 or 6. These two possibilities, along with their substrings of length five and the result permutations appear below

\[
\begin{array}{ccc}
12345\ldots & 12345, 23456, \ldots & 123456, 234561, \ldots \\
\text{shorthand universal cycle} & \text{substrings} & \text{permutations}
\end{array}
\]

\[
\begin{array}{ccc}
123451\ldots & 12345, 23451, \ldots & 123456, 234516, \ldots \\
\text{shorthand universal cycle} & \text{substrings} & \text{permutations}
\end{array}
\]

Notice that 123456 is followed either by 234561 or 234516. In general, shorthand universal cycles always provide prefix-shift Gray codes that shift the first symbol into the last or second-last position. The former case provides a slight improvement over general prefix-shifts with respect to the number of ordered and unordered pairs that change. In particular, the prefix-shift \( s_1 \ldots s_n \) replaces the ordered pair \( s_1s_2 \) by the ordered pair \( s_ns_1 \). For this reason, shorthand universal cycles provide a slight advantage, on average, to prefix-shift Gray codes with respect to the total number of ordered and unordered pairs that change.

Shorthand isomorphism can be applied to any fixed-content language, including multiset permutations. Furthermore, the same observations involving prefix-shifts still hold. Despite these advantages, the idea of shorthand isomorphism is somewhat new to the academic community. Under a different name, Jackson [39] proved that shorthand universal cycles exist for the permutations of \( \{1, 2, \ldots, n\} \). (In particular, he proved that a universal cycle for the \( k \)-permutations of \( \{1, 2, \ldots, n\} \) exist for all \( n \) and \( 1 \leq k \leq n − 1 \), and the \( k = n − 1 \) case is equivalent to a shorthand universal cycle for permutations.) However, like de Bruijn’s pioneering work, the proof relies on
Chapter 1: Combinatorial Generation

Section 1.2: Contemporary Results

graph theoretic arguments and does not provide a reasonable construction method for large values of \( n \). Knuth asked for an efficient construction within the *Generating all Permutations* fascicle [46] of the new volume of *The Art of Computer Programming*. An answer to this question was provided by Ruskey-Williams [71] and is due to appear in the final printing of the volume. An alternative answer was known to the bell-ringing community and is discussed in Section 4.2.1.

1.2.4 Efficient Algorithms

Patiently, inexorably, the computer had been rearranging letters in all their possible combinations, exhausting each class before going on to the next. As the sheets had emerged from the electromagnetic typewriters, the monks had carefully cut them up and pasted them into enormous books.

- in *The Nine Billion Names of God*

Section 1.1 used *The Nine Billion Names of God* to discuss the advantages of minimal-change orders over lexicographic order. In particular, minimal-change orders were of interest to George and Chuck since they could reduce the amount of time and effort expended by the Mark V printer. On the other hand, the engineers may have tried to convince the Lama that no amount of printing was necessary. To justify this possibility, suppose the Lama’s belief system did not explicitly state that the possible names of God had to be written, but instead could be spoken or thought. If this were the case, then the engineers may have suggested that the Mark V simply generate each possible name within its internal memory. In this scenario, the bottleneck in completing the project would no longer be printing each name once. Instead the bottleneck would be representing each name once within the contents of the Mark V’s memory.

In fact, the aforementioned scenario arises quite frequently in modern-day computer science. More specifically, it is often desirable to create each combinatorial object once in computer memory and to bypass storing or printing each object. Knuth makes note of this fact on the first page of his *Generating all Combinations* fascicle [45] when stating that the goal “is to study methods for running through all possibilities”. In particular, this situation arises when solving optimization problems by brute force. To measure the efficiency of these types of algorithms it is necessary to introduce several technical terms. When “running through all possibilities” the possibilities are best generated *in-place* using a *single shared object*. This means that a single combinatorial object is stored in the computer memory, and then it is repeatedly modified to create every possibility. The speed of a combinatorial generation algorithm can then be measured in terms of how quickly it can make each successive modification. In particular, the time required for each modification can be lower than the size of the possibility. This is especially true if the combinatorial objects are generated in a minimal-change order. For example, the reflected Gray code for \( n \)-tuples and the Johnson-Trotter-Steinhaus order for the permutations of \( \{1,2,\ldots,n\} \) can be generated by algorithms that require a constant amount of time to create each successive possibility, irrespective of the value of \( n \) [46]. More generally, the term *loopless* is due to Ehrlich [18] and refers to an algorithm that creates each successive possibility in worst-case \( O(1) \)-time, where the hidden constant does not depend on the size of the possibilities being created. A number of the efficient algorithms discussed in Section 1.2.2 are loopless algorithms including [4], [94], [89], [92], [52], [7], [53], and [51].

In practice, many of the fastest algorithms are not loopless, but run in *constant amortized time (CAT)*. This means that individual modifications may take more than \( O(1) \)-time, but the modifications take a constant amount of time on average. Lexicographic orders are frequently used to create CAT algorithms, although clever optimization and analysis is often required to prove the...
result. See [62] for the best resource on CAT algorithms using lexicographic order. CAT algorithms mentioned in Section 1.2.2 include [48] and [75]. Also see [66] for analysis on the construction of the lexicographically smallest de Bruijn cycle.

In terms of memory consumption, an in-place combinatorial generation algorithm can be measured by how much additional memory it uses. Additional memory does not include the storage used for the single shared object, and in some cases may be lower than this quantity. In particular, algorithms with this property may use a constant number of additional variables, where each additional variable is a single integer or pointer requiring $O(\log n)$ bits. In the case of optimization problems, additional memory also does not include memory necessary for specifying the associated values. (The table in Figure 1.6 provides an example of this uncounted expense.)

Before concluding this discussion on efficient algorithms, it is important to recall that many problems associate a value with each instance of a combinatorial object. In these cases, the bottleneck of solving the problem may involve computing the value of each object as opposed to generating each object. On the other hand, in some situations it is possible to update the associated value of successive objects in the same way that the single shared object is modified. For example, recall that Table 1.1 summarizes the number of ordered and unordered pairs that are changed by applying various operations. In particular, at most two pairs are changed when applying the prefix-shift operation. Therefore, if a prefix-shift Gray code is used in an application whose value depends solely on these pairs, then it will be possible to update the associated value of each successive object in worst-case $O(1)$-time. In other words, the evaluation loopless. For optimization problems, simultaneous worst-case $O(1)$-time generation and worst-case $O(1)$-time evaluation is the ultimate goal with respect to combinatorial generation.

1.2.5 Stacker-Crane Problem

The High Lama and his assistants would be sitting in their silk robes, inspecting the sheets as the junior monks carried them away from the typewriters and pasted them into the great volumes.

- in *The Nine Billion Names of God*

For three centuries, the inhabitants of the Tibetan lamasery had been filling massive volumes with the potential names of God. To ensure that none of the names were overlooked, the Lamas must have had rigorous checks and balances built into their daily routine. Once a volume was completed, it would have to be inspected by several other Lamas before being deposited into the library. Empty books would then make their way to the Lamas who were ready to start writing anew, and from time-to-time it would have been necessary to remove books from the library to do additional “bookkeeping” on the progress of the overall list.

Great care also would have been taken to ensure that no book was lost or misplaced during its transportation between the myriad of rooms and buildings in the lamasery. For this reason, the High Lama of a previous generation may have decreed that a single courier be in charge of all book deliveries on a given day. To avoid confusion (and limit back strain) the courier would be limited to carrying at most one giant volume at a time. Furthermore, the High Lama may have forbidden the courier from leaving books at an intermediate location. Finally, to provide a consistent environment free of disruptions, the courier may have been instructed to pick up and deliver all of the books while the remaining Lamas were joined in morning prayer. For this reason, the courier’s goal is to minimize the amount of elapsed time between the beginning of his first pick up and the end of his last delivery.
Such a system may have been in place by 1661 when Austrian explorer and professor of mathematics Johann Grueber traveled overland from Peking to Lhasa. Grueber’s sketch of the Potala palace is reproduced in Figure 1.6, and could have resembled the lamasery due to Clarke’s description of its adequate balance at the Asiatic Bank. Figure 1.6 also formulates a sample problem for the courier at the lamasery involving six books and five rooms.

Figure 1.6: An instance of the courier’s stacker-crane problem at the lamasery. There are five rooms, 1-5 and six books, A-F. The books need to be delivered between the rooms along their specified route. The table provides minimum room-to-room distances measured in walking minutes.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Now let us consider the possible solutions to the courier’s problem outlined in Figure 1.6. Due to the aforementioned constraints and goals, the courier must repeatedly pick up a book, deliver the book to its destination along the quickest route, and then move to the room where the next book is located. Therefore, the courier’s entire route can be modeled by the order of the books that he delivers. More formally, his delivery order in this case is simply a permutation of

\[
\{ A, B, C, D, E, F \}. \tag{1.10}
\]

Now consider the amount of time required for the courier to complete the following delivery order

\[
A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F. \tag{1.11}
\]

The total duration can be divided into two partial durations depending on whether the courier is carrying a book or not. More precisely, the first duration includes the times required to deliver each book once it has been picked up. Since each of the books is delivered once, then this first duration is

\[
\sum_{i=1}^{5} d_{i} = 1 + 2 + 3 + 1 + 5 = 13. \tag{1.12}
\]

The second duration includes the intermediate times consumed between deliveries, when the courier is walking from one room to the next without carrying a book. For example, given the delivery

---

5The complete sketch appears in Athanasius Kircher’s China Illustrata from 1667 [43].

6The mountainous terrain explains the lack of symmetry in room-to-room distances in Figure 1.6.
order in (1.11) then this second duration is

\[
\begin{align*}
&\text{\(\sum \binom{2}{3} + \binom{2}{3} + \binom{4}{5} + \binom{4}{4} + \binom{5}{5}\) = 7 + 7 + 1 + 0 + 5 = 20.}
\end{align*}
\] (1.13)

Therefore, the courier’s total duration for the delivery order in (1.11) is \(13 + 20 = 33\), by (1.12) and (1.13).

Given this description, it is clear that the first duration does not depend on the delivery order, and the second duration depends solely on the ordered pairs in the delivery order. For this reason, Table 1.1 implies that the duration of a delivery order can be updated in \(O(1)\)-time whenever the delivery order is modified by a shift or a transposition. In particular, only two additions and two subtractions from the table in Figure 1.6 need to be applied to update the duration of a delivery order once if it is modified by a prefix-shift. For example, given the following delivery order

\[
\text{\(\text{ABCDEF} = \text{GABDEF}\),}
\]

its duration can be computed from (1.12) and (1.13) as follows

\[
\begin{align*}
&33 - \sum \binom{2}{3} + \binom{4}{5} + \binom{4}{4} + \binom{5}{5} = 33 - 7 - 1 + 6 + 5 = 36.
\end{align*}
\] (1.14)

Now that the basic elements of the courier’s problem are understood, it is important to point out that there is a simplifying assumption made in the sample problem given in Figure 1.6. In particular, there is at most one book that needs to be transported between any ordered pair of rooms. For example, book \(\text{\textit{f}}\) is the single book that needs to be transported from room \(\text{\textit{a}}\) to room \(\text{\textit{b}}\). Now suppose that the courier’s problem is modified to include the transport of a second book from room \(\text{\textit{a}}\) to room \(\text{\textit{b}}\). Since the restrictions of the problem forbid the courier from carrying two books at once, the solutions can still be specified by the order of the books that are delivered. In particular, the solutions to this problem could be modeled by the permutations of

\[
\{\text{\textit{a}}, \text{\textit{b}}, \text{\textit{c}}, \text{\textit{d}}, \text{\textit{e}}, \text{\textit{f}}, \text{\textit{g}}\}
\]

where \(\text{\textit{g}}\) represents the additional book. However, this model has the disadvantage that it contains twice as many possibilities as distinct solutions. In particular, transposing \(\text{\textit{f}}\) and \(\text{\textit{g}}\) in any permutation will not actually change the courier’s route. The courier can avoid this redundancy by modeling the solutions on the permutations of the following multiset

\[
\{\text{\textit{a}}, \text{\textit{b}}, \text{\textit{c}}, \text{\textit{d}}, \text{\textit{e}}, \text{\textit{f}}, \text{\textit{g}}\}
\]

In general, the courier is faced with an optimization problem on multiset permutations, where the associated value depends solely on the ordered pairs. The courier’s problem is known in theoretical computer science and combinatorial optimization as a stacker-crane problem (SCP). In general, the stacker-crane problem arises when objects need to be quickly transported directly between locations by a single vehicle that can carry at most one object at a time. SCP is extremely difficult to solve in practice, and Frederickson-Hecht-Kim [23] have shown that the associated decision problem is NP-complete. (For an introduction to NP-completeness see Garey-Johnson [24].) In particular, SCP generalizes one of the most important NP-complete problems known as the traveling salesman problem (TSP). Informally, TSP is the special case of SCP where the objects need to be picked up but not delivered. For thorough coverage of TSP and its variations see Applegate-Bixby-Chvátal-Cook [1] and Gutin-Punnen [30].
The results of this thesis are applicable to SCP for several reasons. Chapter 3 provides the first prefix-shift Gray code for multiset permutations. (The operation that generates this Gray code is similar to operation (ii), and appears in a more formal setting in (4.3a) on page 107.) Prefix-shifts provide the best results with respect to changing ordered pairs in Table 1.1. Furthermore, this advantage is even more pronounced given the fact that adjacent-transposition Gray codes do not always exist for multiset permutations, as illustrated by Figure 1.5. Chapter 4 provides a loopless algorithm for generating this prefix-shift Gray code. In particular, Algorithm 7 is the first loopless algorithm for generating multiset permutations that uses a constant number of additional variables. (The algorithm has the surprising property that it does not store any information regarding the contents of the multiset.) Besides being more efficient, Algorithm 7 is also considerably simpler than any previously known algorithms for generating shift Gray codes. Collectively, these results provide a simple brute force solution to SCP featuring worst-case O(1)-time generation, worst-case O(1)-time evaluation, and the use of O(1) additional variables.\textsuperscript{7}

Interestingly, there is also a sense in which the courier could generate and evaluate his possible routes by a loopless manual algorithm. Recall the bottom-right illustration in Figure 1.1. By using marbles of various shades, a Lama could easily implement Algorithm 7 using one hand. (In particular, the Lama needs only to keep his eye on the leftmost pair of lighter-darker marbles in order to apply successive applications of (4.3a).) With his free hand, the same Lama could use an abacus to update the duration of each successive possibility using the type of calculations shown in (1.14). Alternatively, given the 300 year history of the project at the lamasery, it is also possible that the Lamas would have computed shorthand universal cycles for the permutations of all small multisets. By using the resulting prefix-shift Gray codes, a Lama could reduce his total amount of abacus work. (In particular, recall the observation from Section 1.2.3 that each prefix-shift of the form \( s_1 \rightarrow \ldots \rightarrow s_n \) changes only one ordered pair.)

Finally, the results of this thesis are also robust enough to handle changes in the underlying problem. Suppose the courier was falling behind on his name-writing assignments. For this reason, he wished to minimize not only the delivery time, but also the amount of time he spent walking from his room to pick up the first book, and the amount of time he spent returning to his room after completing the last delivery. With some additional provisions, the courier could model his route based on multiset necklaces instead of multiset permutations. Unfortunately, prior to this thesis there were no known Gray code for multiset necklaces. However, the same general theory that produced operations (i) and (ii) also produces similar rules for generating a shift Gray code for these objects.

Before concluding the discussion of this problem, it is mentioned that \[23\] provides a \( \frac{9}{5} \)-approximation algorithm for SCP and proves that SCP is NP-complete by reducing TSP to SCP. In their version of the problem there is a specified initial vertex \( v_0 \in V \). Their reduction also proves the NP-completeness of SCP without an initial vertex by replacing TSP by the minimum weight Hamiltonian path problem. Stacker-crane problems also remain NP-hard when the underlying mixed-graph is a tree, although stronger approximation algorithms exist by Frederickson-Guan \[22\] that almost always provide exact solutions as shown by Coja-Oghlan-Krumke-Nierhoff \[11\]. Generalizations including \( k \) cranes have also been considered \[23\] and also have real-world applications (see Burkard-Fruhwirth-Rote \[5\]). Stacker-crane problems can also be described as single vehicle pickup and delivery problems (or single vehicle dial-a-ride problems) that forbid multiple pickups between deliveries.

\textsuperscript{7}This brute force solution could be described as using cute force.
1.3 New Results

The purpose of the previous two sections was to survey historical and contemporary results in combinatorial generation, and to motivate the theoretical and practical value of studying shift Gray codes. The remainder of this thesis focuses on the abstract goals of constructing shift Gray codes and shorthand universal cycles, and for developing efficient algorithms for generating them. This section outlines the basic results.

Chapter 2 builds a framework that generalizes the fixed-content languages mentioned in this section together with fixed-content languages that represent balanced parentheses, $k$-ary Dyck words, unit interval graphs, Schröder and Motzkin paths, linear-extensions of posets, and additional variations of necklaces including Lyndon words (see Table 2.1 on page 28). The term bubble was chosen since the definition is related to bubble sort (see Knuth [44]). In particular, any string in a bubble language can be sorted into non-increasing order by a series of left-shifts that “bubble” larger symbols to the left.

Chapter 3 takes advantage of this new framework by proving that every bubble language has a simple left-shift Gray code. All of these Gray codes can be expressed using the cool-lex variation of lexicographic order. Co-lexicographic order is lexicographic order applied to strings read from right-to-left instead of left-to-right. This name is often abbreviated to co-lex, and this explains the cool-lex moniker. Table 1.2 compares the recursive structure of co-lex and cool-lex for the permutations of \{1, 2, 3, 4\}. Co-lex order is typically defined by the value of the rightmost symbol. This last symbol of each string is underlined and rewritten on the right side of the first column. Notice that each suffix in question appears contiguously and in increasing order: 1, 2, 3, 4. Furthermore, the same pattern holds recursively.

The second column again contains co-lex order on the left side, however in this case a different suffix is underlined and copied on the right side of the column. To describe the suffixes, notice that the first string, 4321, has its symbols in non-increasing order. Furthermore, it does not have an underlined suffix. This is because the underlined suffixes are the shortest suffixes that are not also suffixes of this special string 4321. These suffixes are known as scuts, which is an English word meaning “short stubby tail” (see dictionary.com [34]). For example, the scut of 4231 is 31 because it is not a suffix of 4321 and the next shortest suffix, 1, is a suffix of 4321. Notice that each scut appears contiguously and in the following order: 421, 31, 41, 2, 3, 4. Put another way, the scuts are ordered by decreasing length and then by increasing first symbol. Furthermore, the same pattern holds recursively.

With this second recursive interpretation of co-lex in mind, cool-lex order can be summarized succinctly as a reordering of the scuts and the string in non-increasing order. In particular, the non-increasing string 4321 appears last (instead of first) and the scuts are ordered by increasing first symbol and then by decreasing length (instead of by decreasing length and then by increasing first symbol). For example, the left side of the third column contains the cool-lex order for the permutations of \{1, 2, 3, 4\}. The scuts are again underlined and copied on the right side of the column. This time the scuts appear contiguously and in the following order: 2, 31, 41, 41, 421. Furthermore, the same pattern holds recursively. Figure 1.7 illustrates co-lex and cool-lex order for the permutations of \{1, 2, 3, 4, 5, 6, 7\}. When comparing pages 5 and 25, notice that Figure 1.2 emphasizes the reordering of the non-increasing string, while Figure 1.7 emphasizes the reordering of the scuts.

Cool-lex order for bubble languages has at least two important applications. The first application involves efficient algorithms. In general, any bubble language can be generated in cool-lex order by a simple algorithm found on page 91. Furthermore, this generic algorithm can be highly optimized for specific bubble languages as seen in the first half of Chapter 4.
Figure 1.7: An artistic representation of co-lexicographic (above) and cool-lex (below) order for permutations of \(\{1, 2, 3, 4, 5, 6, 7\}\). The lightest and darkest regions represent 1 and 7 respectively. Individual strings are read along a line segment originating from the center, and the first and last strings are at either side of 12 o’clock. Cool-lex proceeds leftwards (counterclockwise) and involves left-shifts, while reverse cool-lex proceeds rightwards (clockwise) and involves right-shifts. Co-lexicographic order proceeds counterclockwise, while reverse co-lexicographic order proceeds clockwise.
Table 1.2: Two recursive views of co-lex order, together with the recursive view of cool-lex order for permutations of \{1, 2, 3, 4\}.

<table>
<thead>
<tr>
<th>Co-lex</th>
<th>last</th>
<th>Co-lex</th>
<th>scut</th>
<th>Cool-lex</th>
<th>scut</th>
</tr>
</thead>
<tbody>
<tr>
<td>432_</td>
<td>1</td>
<td>432_</td>
<td>42_</td>
<td>143_</td>
<td>2</td>
</tr>
<tr>
<td>342_</td>
<td>1</td>
<td>342_</td>
<td>42_</td>
<td>413_</td>
<td>2</td>
</tr>
<tr>
<td>423_</td>
<td>1</td>
<td>423_</td>
<td>31_</td>
<td>341_</td>
<td>2</td>
</tr>
<tr>
<td>243_</td>
<td>1</td>
<td>243_</td>
<td>31_</td>
<td>134_</td>
<td>2</td>
</tr>
<tr>
<td>324_</td>
<td>1</td>
<td>324_</td>
<td>41_</td>
<td>314_</td>
<td>2</td>
</tr>
<tr>
<td>234_</td>
<td>1</td>
<td>234_</td>
<td>41_</td>
<td>431_</td>
<td>2</td>
</tr>
<tr>
<td>43_</td>
<td>2</td>
<td>43_</td>
<td>2</td>
<td>243_</td>
<td>31</td>
</tr>
<tr>
<td>34_</td>
<td>2</td>
<td>34_</td>
<td>2</td>
<td>423_</td>
<td>31</td>
</tr>
<tr>
<td>41_</td>
<td>2</td>
<td>41_</td>
<td>2</td>
<td>142_</td>
<td>3</td>
</tr>
<tr>
<td>14_</td>
<td>2</td>
<td>14_</td>
<td>2</td>
<td>412_</td>
<td>3</td>
</tr>
<tr>
<td>31_</td>
<td>2</td>
<td>31_</td>
<td>2</td>
<td>241_</td>
<td>3</td>
</tr>
<tr>
<td>13_</td>
<td>2</td>
<td>13_</td>
<td>2</td>
<td>124_</td>
<td>3</td>
</tr>
<tr>
<td>42_</td>
<td>3</td>
<td>42_</td>
<td>3</td>
<td>214_</td>
<td>3</td>
</tr>
<tr>
<td>24_</td>
<td>3</td>
<td>24_</td>
<td>3</td>
<td>421_</td>
<td>3</td>
</tr>
<tr>
<td>41_</td>
<td>3</td>
<td>41_</td>
<td>3</td>
<td>342_</td>
<td>421</td>
</tr>
<tr>
<td>14_</td>
<td>3</td>
<td>14_</td>
<td>3</td>
<td>234_</td>
<td>41</td>
</tr>
<tr>
<td>21_</td>
<td>3</td>
<td>21_</td>
<td>3</td>
<td>324_</td>
<td>41</td>
</tr>
<tr>
<td>12_</td>
<td>3</td>
<td>12_</td>
<td>3</td>
<td>132_</td>
<td>4</td>
</tr>
<tr>
<td>32_</td>
<td>4</td>
<td>32_</td>
<td>4</td>
<td>312_</td>
<td>4</td>
</tr>
<tr>
<td>23_</td>
<td>4</td>
<td>23_</td>
<td>4</td>
<td>231_</td>
<td>4</td>
</tr>
<tr>
<td>31_</td>
<td>4</td>
<td>31_</td>
<td>4</td>
<td>123_</td>
<td>4</td>
</tr>
<tr>
<td>13_</td>
<td>4</td>
<td>13_</td>
<td>4</td>
<td>213_</td>
<td>4</td>
</tr>
<tr>
<td>21_</td>
<td>4</td>
<td>21_</td>
<td>4</td>
<td>321_</td>
<td>4</td>
</tr>
<tr>
<td>12_</td>
<td>4</td>
<td>12_</td>
<td>4</td>
<td>432_</td>
<td></td>
</tr>
</tbody>
</table>

Theorem discussed in this thesis involves shorthand universal cycles. Interestingly, reverse cool-lex can be used to create shorthand universal cycles for multiset permutations in much the same way that de Bruijn cycles can be created from lexicographic order. In other words, cool-lex order provides a multiset permutation analogue to lexicographic order with respect to the FKM algorithm. For an illustration, consider again the right-shift Gray code for \((3,3)\)-combinations given in (1.1), except cross off the strings that are not lexicographically largest in their rotation set

\[
\begin{align*}
\text{110010, 110010, 110010, 110010, 110010, 111000, 111000, 111000, 111000, 111000, 111000, 111000,}
\end{align*}
\]

For example, 101100 is crossed off because 110010 is a lexicographically larger rotation. Appending the underlined aperiodic prefix of each remaining string yields

\[
\begin{align*}
1100101010100111000,
\end{align*}
\]

which is the shorthand universal cycle previously seen in (1.3). These results are investigated in the second half of Chapter 4 and rely on an interesting shorthand rotation property that is hidden in the cool-lex order of every bubble language.
1.3.1 Summary

This thesis introduces the concept of a bubble language in Chapter 2 and a new variation of lexicographic order called cool-lex order in Chapter 3. Whenever bubble languages are expressed in cool-lex order the result is a left-shift Gray code. Furthermore, these left-shift Gray codes can be used to create the efficient generation algorithms and shorthand universal cycles found in Chapter 4. Specific results include:

- the first prefix-shift Gray code for multiset permutations,
- the first shift Gray code for linear-extensions of B-posets, ordered trees with fixed branching sequence, restricted Schröder and Motzkin paths,
- the first minimal-change order for multiset necklaces, pre-necklaces, and Lyndon words,
- the first loopless algorithm using a constant number of additional variables for generating multiset permutations,
- the most efficient array-based algorithm for balanced parentheses in practice (see Arndt [2]),
- the first universal cycle for the middle levels (fixed-density de Bruijn cycle), and
- the first shorthand universal cycles of multiset permutations.

The interested reader will also find additional results and open problems in Chapter 5.