A Dynamic Programming Approach to Generating a Binary Reflected Gray Code Sequence

Andrew T. Phillips and Michael R. Wick
Department of Computer Science
University of Wisconsin – Eau Claire
Eau Claire, WI 54701
phillipa@uwec.edu
wickmr@uwec.edu

Abstract

A binary reflected Gray code (Gray code for short) is a one-to-one function *G(i)* of the integers $0 \bullet i \bullet 2^n - 1$ (for some pre-selected value of n) to binary numbers such that the binary number for $G(i)$ and $G(i+1)$ differ in exactly one bit. For example, with N = 3, the Gray codes (written in binary) are 000, 001, 011, 010, 110, 111, 101, 100. In this paper, we describe a dynamic programming algorithm and associated storage efficient data structure that is capable of generating a complete n-bit binary reflected Gray code sequence in provably optimal $\Theta(2^n)$ time and space. The algorithm uses the inherent redundancy in the elements of a binary reflected Gray code sequence to avoid recalculation of repeated subsequences. The resulting method is an optimal, convenient, and elegant solution to working with binary reflected Gray codes.

1 Introduction

A binary reflected Gray code (Gray code for short) is a one-to-one function $G(i)$ of the integers $0 \bullet i \bullet 2^n - 1$ (for some pre-selected value of n) to binary numbers such that the binary number for $G(i)$ and $G(i+1)$ differ in exactly one bit. For example, with $n = 3$, the Gray codes (written in binary) are 000, 001, 011, 010, 110, 111, 101, 100. The inverse Gray code $G^{-1}(g)$ is the one-to-one function that computes the integer i for which $G(i) = g$. Gray codes are named for Frank Gray who patented the use of them in shaft encoders in 1953 [1].

Gray codes originated when digital logic circuits were built from vacuum tubes and electromechanical relays, at which time counters generated enormous power demands and noise spikes when many bits changed at once. Using Gray code counters, any increment or decrement in the value changes only one bit, regardless of the size of the number, thereby minimizing the effect of noise. Mechanical position sensors use Gray codes to convert the angular position of a disk to a digital form [1]. For example, some parts of a track on a disk have metal, corresponding to a “1”, and other parts have insulator, corresponding to a “0”. Each sensor has a row of metal fingers, radiating out from the center of the disk, with one finger riding on each track. As the disk rotates, the metal and insulator regions of the track move under each finger, and the combination of 1’s and 0’s read by the fingers indicate the angular position of the disk. Gray codes are used in place of other possible binary sequences because the fingers can’t be lined up perfectly, and if two bits were to change at the same, one finger would register its bit change before the other finger did, causing an undesirable glitch. Other applications of Gray codes involve computing Hamiltonian circuits in hypercubes [2], the classification of Venn diagrams [3], the design of communication codes [4], and the solution of the famous Towers of Hanoi problem [5]. For example, Table 1 (adapted from [6]) illustrates how the Towers of Hanoi can be solved using Gray codes.

In Table 1, each bit in the three-bit binary number in the Bitstring column corresponds to one of the three discs to be moved, with bit i representing disc i. The convention here is that the smallest disc is disc 1 (the least significant bit), and the largest is disc 3 (the most significant bit). The sequence is initialized to 000, and a change in the i^{th} bit from one binary code to the next represents a move of disc i. For example, the flipping of bit 1 from 000 to 001 implies that disc 1 is to be moved, whereas the flipping of bit 2 from 111 to 101 implies that disc 2 is to be moved. Hence, the binary reflected Gray code sequence represents a sequence of single disc moves that can be used to solve the Towers of Hanoi problem.

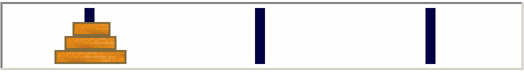
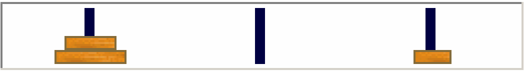
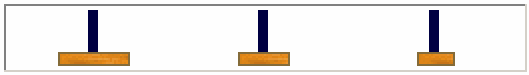
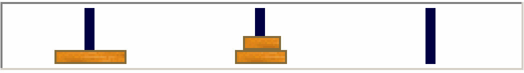
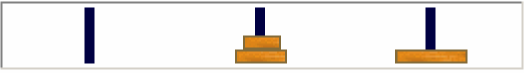
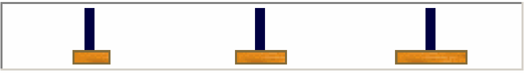
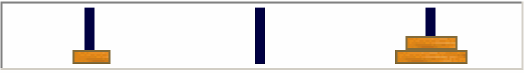
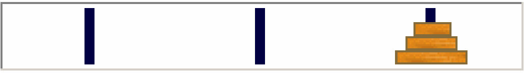
Bitstring $n \cdots 321$	**Towers of Hanoi**
000	
001	
011	
010	
110	
111	
101	
100	

Table 1: Application of Gray Codes to the Towers of Hanoi Problem

In this paper, we present the derivation of a time and space optimal algorithm for generating the entire Gray code sequence $G(i)$, $0 \bullet i \bullet 2^n - 1$, for a given number of bits n. In addition, we also give the corresponding time and space optimal algorithm for generating the indices $G^{-1}(g)$ for the sequence of Gray codes encoded by g.

2 The Recurrence Relations

A sequence of binary reflected Gray codes follow a well-defined pattern. In an n bit system of 2^n Gray code values, the kth binary reflected Gray code, denoted by $G(k,n)$, can be defined by the recurrence relation:

$$G(k,1) = k$$

$$G(k,n) = \begin{cases} 0G(k,n-1) & k < 2^{n-1} \\ 1G(2^n - 1 - k, n-1) & k \geq 2^{n-1} \end{cases}$$

for $0 \bullet k < 2^n$. In reverse, to compute the index k corresponding to the binary reflected Gray code $g = g_{n-1}g_{n-2}\ldots g_0$, where $g_j \in \{0,1\}$ for all j, one can compute $G^{-1}(g_{n-1}g_{n-2}\ldots g_0,n)$ as follows (note that $n = \lfloor \log_2(g) \rfloor$):

$$G^{-1}(g_0, 1) = g_0$$

$$G^{-1}(g_{n-1}g_{n-2}\cdots g_0, n) = \begin{cases} G^{-1}(g_{n-2}g_{n-3}\cdots g_0, n-1) & g_{n-1} = 0 \\ (2^n - 1) - G^{-1}(g_{n-2}g_{n-3}\cdots g_0, n-1) & g_{n-1} = 1 \end{cases}$$

Both functions, $G(k,n)$ and $G^{-1}(g,n)$, are computed in •(n) time for any fixed k or g. Clearly, if one wanted only one Gray code or the index corresponding to just one Gray code value, then the functions above are very efficient (linear time in the number of bits is optimal). But more typically, the <u>entire</u> Gray code sequence, or its entire inverse sequence, is desired. To achieve this, we can use a straightforward recursive divide-and-conquer style implementation based on the recurrence relations above. Figure 1 contains the recursive implementation that produces the complete binary reflected Gray code sequence $G(k,n)$, k=0...2^n-1, for n-bits.

```
public class Recursive {
  public static String G(int k , int n){
     if (n == 1){
        return new Integer(k).toString();
     }
     else if (k < (int)Math.pow(2,n-1)){
        return "0" + G(k, n - 1);
     }
     else {
        return "1" + G((int)Math.pow(2,n) - 1 - k, n - 1);
     }
  }

  public static String[] graycode(int n){
     String[] result = new String[(int)Math.pow(2,n)];
     for (int k = 0; k < (int)Math.pow(2,n); k++){
        result[k] = G(k,n);
     }
     return result;
  }

  public static void main(String[] args) {
     String[] codes = graycode(4);
     for (int i = 0; i < codes.length; i++){
        System.out.println(codes[i]);
     }
  }
}
```

Figure 1: Recursive Divide-and-Conquer Construction to Compute $G(k,n)$ for all k.

Is this recursive implementation an efficient way to compute the entire Gray code sequence? Since each computation of $G(k,n)$ requires •(n) steps, and the implementation of *graycode(n)* calls $G(k,n)$ for k=0 to 2^n-1, the total running time is on the order of $\bullet(n \cdot 2^n)$. The same analysis is true for computing the entire inverse sequence for $G^{-1}(g,\mathrm{n})$.

Can we do better? Since any algorithm that generates and stores the entire n-bit Gray code sequence must produce and store 2^n Gray codes, the generation of a complete Gray code sequence is an intractable problem (as it must produce a non-polynomial amount of output). Therefore, the best possible time and space complexity is $\bullet(2^n)$. Our two simple recursive divide-and-conquer algorithms based on the direct implementation of the definitions of $G(k,n)$ and $G^{-1}(k,n)$ are therefore suboptimal.

So, what is the source of the inefficiencies in our algorithms, and can those inefficiencies be removed? To explain the inefficiency, we focus on the algorithm for producing the Gray codes, since the analysis is similar for the inverse problem. Notice that our algorithm suffers from the fact that the recursive sub-problems solved during the algorithm's execution are not independent of one another. Table 2 illustrates this point.

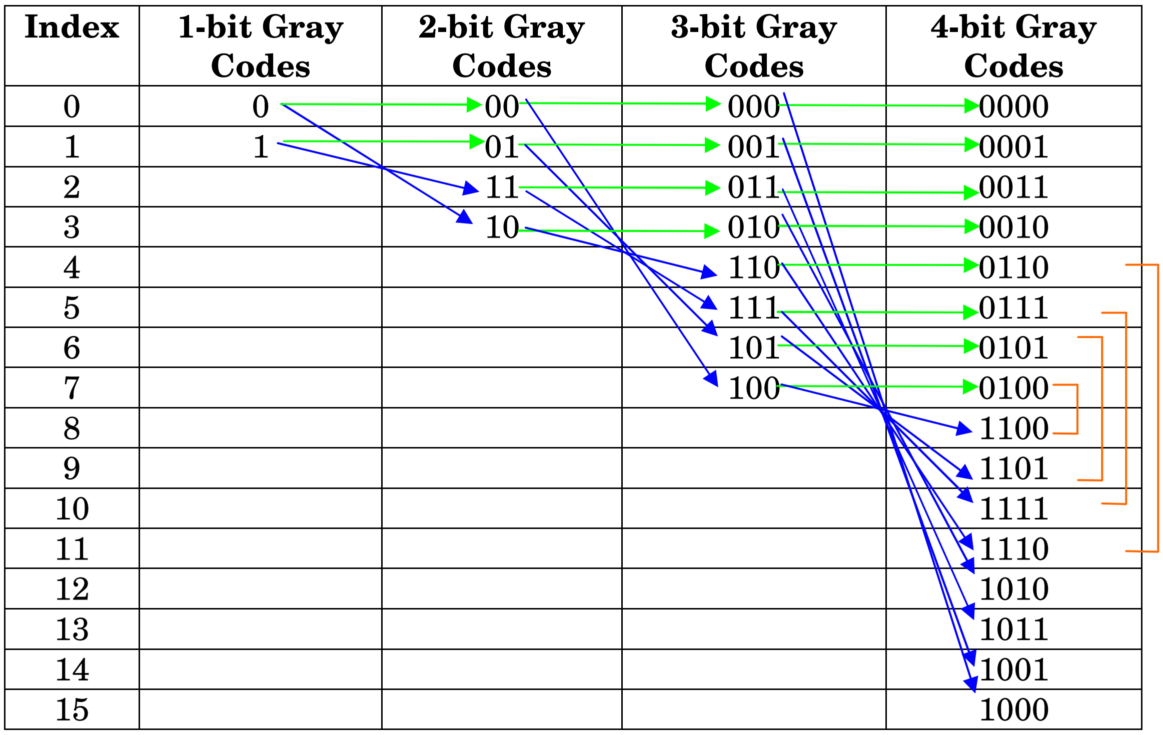
Index	**1-bit Gray Codes**	**2-bit Gray Codes**	**3-bit Gray Codes**	**4-bit Gray Codes**
0	0	00	000	0000
1	1	01	001	0001
2		11	011	0011
3		10	010	0010
4			110	0110
5			111	0111
6			101	0101
7			100	0100
8				1100
9				1101
10				1111
11				1110
12				1010
13				1011
14				1001
15				1000

Table 2: Illustration of Repeated Work during Generation of 4-bit Gray Code Sequence

For example, to generate the 6th (index 5) 4-bit Gray code 0111 requires generation of the 6th 3-bit Gray code 111, which requires generation of the 3rd 2-bit Gray code 11, which requires generation of the 2nd 1-bit Gray code 1. In addition, generation of the 11th (index 10) 4-bit Gray code 1111 also requires this exact same set of prerequisite Gray codes to be generated. This repeated work is the source of the inefficiency in our algorithms.

3 A Dynamic Programming Algorithm

The dependency between the sub-problems of the divide-and-conquer approach immediately suggests the use of a dynamic programming approach. In a dynamic programming approach, the sub-problems are ordered, solved, and stored so that the solution of a problem solves each sub-problem *only once*. In our Gray code application, this can be achieved by producing all 1-bit Gray codes before any 2-bit Gray codes, all 2-bit Gray codes before any 3-bit Gray codes, and so on. The Table 3 illustrates this relationship.



Index	**1-bit Gray Codes**	**2-bit Gray Codes**	**3-bit Gray Codes**	**4-bit Gray Codes**
0	0	00	000	0000
1	1	01	001	0001
2		11	011	0011
3		10	010	0010
4			110	0110
5			111	0111
6			101	0101
7			100	0100
8				1100
9				1101
10				1111
11				1110
12				1010
13				1011
14				1001
15				1000

Table 3: Illustration of the Reuse of Cached Sub-problem Solutions

This approach requires us to generate and store the entire (n-1)-bit Gray code sequence prior to generating any of the codes in the n-bit Gray code sequence, and hence, is no longer a recursive approach (although a recursive approach using memoization is also possible). The obvious approach to implementing this algorithm is to use a two-dimensional array to store the previously generated Gray code sequences so that they can be easily located to produce the next "longer" sequence of Gray codes. Figure 2 contains a Java implementation for this approach.

```
public class DynamicProgramming {
   public static String[] graycode(int n){
      int k = (int) Math.pow(2,n);
      String[][] _G = new String[k][n+1];

      _G[0][1] = "0";
      _G[1][1] = "1";

      for (int j = 2; j <= n; j++) {
         int half = (int) Math.pow(2,j-1);
         int c = (int) Math.pow(2,j) - 1;

         // row index < 2^(n-1)
         for (int i = 0; i < half; i++) {
            _G[i][j] = "0" + _G[i][j-1];
         }

         // row index >= 2^(n-1)
         for (int i = half; i < 2*half; i++) {
            _G[i][j] = "1" + _G[c-i][j-1];
         }
      }

      /* the Gray code sequence for n bits is now found
         in entries _G[i][n], for i = 0 to k-1 */

      String[] result = new String[k];
      for (int i = 0; i < k; i++) {
         result[i] = _G[i][n];
      }

      return result;
   }

   public static void main(String[] args) {
      String[] codes = graycode(4);
      for (int i = 0; i < codes.length; i++){
         System.out.println(codes[i]);
      }
   }
}
```

Figure 2: Dynamic Programming Construction of all *n*-bit Gray Codes.

Notice that this algorithm first produces and stores a Gray code sequence of length 2 (n=1 bit), then of length 4 (2 bits), then length 8 (n=3 bits), and so on until the complete sequence of length 2^n for n bits is produced and stored. Both the time and space complexity of this algorithm is therefore given by

$$\sum_{j=1}^{n} 2^j = \Theta(2^{n+1}) .$$

This is within a constant multiple of the optimal time and space of •(2^n).

4 A Space and Time Optimal Algorithm

A close examination of Table 3 reveals some important information. First, rather than storing each Gray code as a string, we could simply store each Gray code as an integer between 0 and $2^n - 1$ thereby allowing each Gray code to be stored as an integer in an n-bit word. Second, the generation of the n-bit Gray code sequence only depends on the storage of the (n-1)-bit Gray code sequence and not any of the preceding Gray code sequences. This relieves the algorithm from needing to store anything but the immediately previous Gray code sequence. Third, notice that the first half of each n bit Gray code sequence is generated by directly copying the values of the previous (n-1)-bit Gray code sequence and prepending a leading “0”. However, given our new representation of each Gray code as an n-bit word, this simply equates to making explicit the implicit zero in the most-significant bit! This means that the entire (n-1)-bit Gray code sequence can be retained without modification (given a fixed word size) when generating the n-bit Gray code sequence; hence, there is *no work required* by the algorithm to generate the first half of the n-bit sequence. Finally, notice that the generation of the second half of the n-bit Gray code sequence only involves prepending a "1" to the most significant bit of each (n-1)-bit Gray code, but where the order in which the (n-1)-bit Gray codes are used is the reverse order in which they appear in the (n-1)-bit sequence (as illustrated by the orange groupings in Table 3). This allows the algorithm to generate the second half of the n-bit Gray code sequence by setting each successive Gray code to the corresponding Gray code in the first half of the sequence added to the constant value 2^{n-1} (to "set" the nth bit).

These observations lead to the very compact implementation shown in Figure 4 for generating the complete n-bit binary reflected Gray code sequence.

```
public class Optimal {
   public static int[] graycode(int n){
      int count = 0;
      int twoHatJ = 1;
      int[] g = new int[(int)Math.pow(2,n)];
      g[0] = 0;
      g[1] = 1;
      count = 2;
      for (int j = 1; j < n; j++){
         twoHatJ = twoHatJ << 1;
         for (int i = 1; i <= twoHatJ; i++){
            g[count++] = twoHatJ | g[twoHatJ - i]; // HERE
         }
      }

      return g;
   }

   public static void main(String[] args){
      int[] codes = graycode(4);
      for(int i = 0; i < codes.length; i++){
         System.out.println(codes[i]);
      }
   }
}
```

Figure 3: Time and Space Optimal Generation of all *n*-bit Gray Codes.

The sequence of bits generated by this algorithm is illustrated in the Table 4.

0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000

Table 4: Incremental Generation of the 4-bit Gray Code Sequence

The black bits represent the bits that are simply made explicit as the algorithm unfolds based on the underlying machine representation of each Gray code value as an unsigned integer. The blue bits represent bits constructed during the generation of the 1-bit Gray code sequence; the green bits represent bits constructed during the generation of the 2-bit Gray code sequence; the red bits represent bits constructed during the generation of the 3-bit Gray code sequence; the purple bits represent the bits constructed during the generation of the 4-bit Gray codes.

Notice that the algorithm first produces 2 values, then 2 additional values, then 4 additional values, then 8 additional values, and so on. Therefore the algorithm produces and stores (for an n-bit Gray code sequence) exactly

$$2 \quad + \quad \sum_{j=1}^{n-1} 2^{j} = 2^{n}$$

values. Thus, the time and space complexity for this method is •(2^n), which is optimal for the generation of a complete n-bit Gray code sequence. Further, the algorithm, as shown in Figure 3, is extremely concise and easy to implement.

While this analysis has focused on the generation of the Gray code sequence for any fixed value of n, the same approach can be applied to the inverse function, this is to generate the indices $G^{-1}(g)$ for the sequence of Gray codes encoded by g. Interestingly, the code for the inverse function is identical to the code for the normal function except for one line. In particular, the line marked by the "// HERE" comment in Figure 3 need only be replaced with

```
g[count++] = (2*twoHatJ-1) - g[i-1];
```

Given that single change, it is clear that that time and space complexity for this inverse method is also optimal at •(2^n).

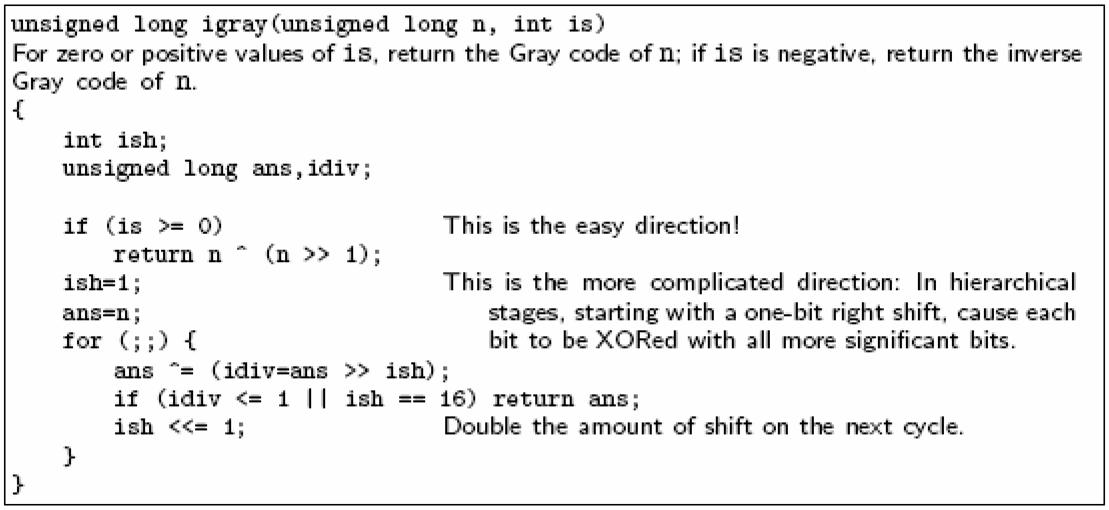
In object-oriented design, there is a technique of implementing such a pair of functions that differ only in certain predetermined steps. This technique is called the Template Method design pattern. In the Template Method design pattern, the invariant code between the two functions is defined only once, using a "hook" for the variant code. The variant code is then defined in two sub-classes so that when invoked by the invariant code, the appropriate variant behavior occurs. The Java implementation of this idea that now implements both the Gray code and inverse Gray code functions is shown in Figure 4.

```
public abstract class TemplateMethod {
   protected abstract int mapping(int[] g, int twoHatJ, int i);

   public int[] graycode(int n){
      int count = 0;
      int twoHatJ = 1;
      int[] g = new int[(int)Math.pow(2,n)];
      g[0] = 0;
      g[1] = 1;
      count = 2;
      for (int j = 1; j < n; j++){
         twoHatJ = twoHatJ << 1;
         for (int i = 1; i <= twoHatJ; i++){
            g[count++] = mapping(g, twoHatJ, i);
         }
      }

      return g;
   }
}

public class ForwardVariant extends TemplateMethod {
   protected int mapping(int[] g, int twoHatJ, int i){
      return twoHatJ | g[twoHatJ - i];
   }
}

public class ReverseVariant extends TemplateMethod {
   protected int mapping(int[] g, int twoHatJ, int i){
      return (2*twoHatJ-1) - g[i-1];
   }
}

public class Main {
   public static void main(String[] args) {

      // list the Gray codes in order
      ForwardVariant fv = new ForwardVariant();
      int[] codes = fv.graycode(4);
      for (int i = 0; i < codes.length; i++){
         System.out.println(codes[i]);
      }

      // list the indices corresponding to the Gray codes
      ReverseVariant rv = new ReverseVariant();
      int[] indices = rv.graycode(4);
      for (int i = 0; i < indices.length; i++){
         System.out.println(indices[i]);
      }
   }
}
```

Figure 4: Template Method Construction of Gray Codes

5 Summary and Significance

In this paper, we have presented the derivation and implementation of a time and space optimal algorithm for the generation of an n-bit binary reflected Gray Code sequence and its inverse (or reverse) function. The resulting algorithm is concise and almost trivial to implement. Furthermore, the reverse algorithm for generating the index of each Gray code is virtually identical to the forward algorithm (a single line change), and the resulting pair of algorithms can be effectively combined into a single generic function via the use of the Template Method design pattern. The end result is an optimal, convenient, and elegant solution to working with binary reflected Gray codes. By way of comparison, one of the most common implementations of these methods is also concise, but quite cryptic as shown in Figure 5.



```
unsigned long igray(unsigned long n, int is)
For zero or positive values of is, return the Gray code of n; if is is negative, return the inverse
Gray code of n.
{
    int ish;
    unsigned long ans,idiv;

    if (is >= 0)                  This is the easy direction!
        return n ^ (n >> 1);
    ish=1;                        This is the more complicated direction: In hierarchical
    ans=n;                            stages, starting with a one-bit right shift, cause each
    for (;;) {                        bit to be XORed with all more significant bits.
        ans ^= (idiv=ans >> ish);
        if (idiv <= 1 || ish == 16) return ans;
        ish <<= 1;                Double the amount of shift on the next cycle.
    }
}
```

Figure 5: Numerical Recipes [7] Version of Gray Code Methods

References

[1] Gray, F. Pulse Code Communication. United States Patent Number 2,632,058. March 17, 1953.

[2] Gilbert, E.N. *Gray Codes and Paths on the n-Cube*. Bell System Tech. J. 37 (1958), pp. 815-826.

[3] F. Ruskey, *A Survey of Venn Diagrams,* Elec. J. of Comb. (1997).

[4] Goddyn, L., Gvozdjak, P. *Binary gray codes with long bit runs*, Electron. J. Combin. 10 (2003), pp. 1-10.

[5] Gardner, M. *The Binary Gray Code*. Ch. 2 in Knotted Doughnuts and Other Mathematical Entertainments. New York: W.H. Freeman, 1986.

[6] The Combinatorial Object Server. *Information on Subsets of a Set*. http://www.theory.csc.uvic.ca/~cos/inf/comb/SubsetInfo.html.

[7] Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T. Gray Codes. §20.2 in *Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd ed*. Cambridge, England: Cambridge University Press, 1992, pp. 886-888.