Loopless Gray Code Enumeration and the Tower of Bucharest

Felix Herter\textsuperscript{1} and Günter Rote\textsuperscript{1}

\textsuperscript{1} Institut für Informatik, Freie Universität Berlin
Takustr. 9, 14195 Berlin, Germany
avealx@zedat.fu-berlin.de, rote@inf.fu-berlin.de

Abstract

We give new algorithms for generating all $n$-tuples over an alphabet of $m$ letters, changing only one letter at a time (Gray codes). These algorithms are based on the connection with variations of the Towers of Hanoi game. Our algorithms are loopless, in the sense that the next change can be determined in a constant number of steps, and they can be implemented in hardware. We also give another family of loopless algorithms that is based on the idea of working ahead and saving the work in a buffer.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases  Tower of Hanoi, Gray code, enumeration, loopless generation

Contents

1 Introduction: the binary reflected Gray code and the Towers of Hanoi .......................... 2
1.1 The Gray code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Loopless algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 The Tower of Hanoi . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.4 Connections between the Towers of Hanoi and Gray codes . . . . . . . . . . . . . . . . . . 4
1.5 Loopless Tower of Hanoi and binary Gray code . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.6 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2 Ternary Gray codes and the Towers of Bucharest ................................................. 5

3 Gray codes with general radixes ................................................................................. 7

4 Generating the $m$-ary Gray code with odd $m$ ......................................................... 7

5 Generating the $m$-ary Gray code with even $m$ ......................................................... 9

6 The Towers of Bucharest++ ......................................................................................... 9

7 Simulation .................................................................................................................. 11

8 Working ahead ............................................................................................................ 11
8.1 An alternative STEP procedure ............................................................................. 13
8.2 Correctness proofs for the work-ahead algorithms .................................................. 14

9 Concluding Remarks .................................................................................................. 16
1 Introduction: the binary reflected Gray code and the Towers of Hanoi

1.1 The Gray code

The Gray code, or more precisely, the reflected binary Gray code $G_n$, orders the $2^n$ binary strings of length $n$ in such a way that successive strings differ in a single bit. It is defined inductively as follows, see Figure 1 for an example. The Gray code $G_1 = 0, 1$, and if $G_n = C_1, C_2, \ldots, C_{2^n}$ is the Gray code for the bit strings of length $n$, then

$$G_{n+1} = 0C_1, 0C_2, \ldots, 0C_{2^n}, 1C_{2^n}, 1C_{2^n-1}, \ldots, 1C_2, 1C_1.$$  (1)

In other words, we prefix each word of $G_n$ with 0, and this is followed by the reverse of $G_n$ with 1 prefixed to each word.

\begin{verbatim}
000000 001011 010111 110100 111101 101101 000010 001001 010101
000011 001010 010111 111000 111011 100111 000100 001000 010001
000110 001101 010111 111100 111111 101110 001011 001110 011001
001011 001111 110100 111110 101000 101010 001111 010100 110101
010000 011000 110100 111100 101100 101110 010011 011011 111101
010010 011010 110111 111111 101001 101101 010101 011101 111110
011000 011100 110101 111101 101010 101110 011010 011110 111110
011010 011110 110101 111101 101011 101111 011111 011000 111111
000000 001011 010111 110100 111101 101101 000010 001001 010101
000011 001010 010111 111000 111011 100111 000100 001000 010001
000110 001101 010111 111100 111111 101110 001011 001110 011001
001011 001111 110100 111110 101000 101010 001111 010100 110101
000000 001011 010111 110100 111101 101101 000010 001001 010101
000011 001010 010111 111000 111011 100111 000100 001000 010001
000110 001101 010111 111100 111111 101110 001011 001110 011001
001011 001111 110100 111110 101000 101010 001111 010100 110101
000000 001011 010111 110100 111101 101101 000010 001001 010101
000011 001010 010111 111000 111011 100111 000100 001000 010001
000110 001101 010111 111100 111111 101110 001011 001110 011001
001011 001111 110100 111110 101000 101010 001111 010100 110101
000000 001011 010111 110100 111101 101101 000010 001001 010101
000011 001010 010111 111000 111011 100111 000100 001000 010001
000110 001101 010111 111100 111111 101110 001011 001110 011001
001011 001111 110100 111110 101000 101010 001111 010100 110101
\end{verbatim}

\[\text{Figure 1} \quad \text{The binary Gray code } G_6 \text{ for 6-tuples and the ternary Gray code for 4-tuples.}\]

1.2 Loopless algorithms

The Gray code has an advantage over alternative algorithms for enumerating the binary strings, for example in lexicographic order: one can change a binary string $a_0a_{n-1}\ldots a_1$ to the successor in the sequence by a single update of the form $a_i := 1 - a_i$ in constant time. However, we also have to compute the position $i$ of the bit which has to be updated. A straightforward implementation of the recursive definition (1) leads to an algorithm with an optimal overall runtime of $O(2^n)$, i.e., constant average time per enumerated bit string.
A stricter requirement is to compute each successor string in constant worst-case time. Such an algorithm is called a loopless generation algorithm. Loopless enumeration algorithms for various combinatorial structures were pioneered by Ehrlich [3], and different loopless algorithms for Gray codes are known, see Bitner, Ehrlich, and Reingold [1] and Knuth [6, Algorithms 7.2.1.1.L and 7.2.1.1.H]. These algorithms achieve constant running time by maintaining additional pointers.

### 1.3 The Tower of Hanoi

The Tower of Hanoi is the standard textbook example for illustrating the principle of recursive algorithms. It has $n$ disks $D_1, D_2, \ldots, D_n$ of increasing radii and three pegs $P_0, P_1, P_2$, see Fig. 2. The goal is to move all disks from the peg $P_0$, where they initially rest, to another peg, subject to the following rules:

1. Only one disk may be moved at a time: the topmost disk from one peg can be moved on top of the disks of another peg
2. A disk can never lie on top of a smaller disk.

![Figure 2 The Towers of Hanoi with $n = 6$ (square) disks. When running the algorithm HANOI from Section 1.3, the configuration in this picture occurs together with the bit string 110011. (There is no easy relation between the positions of the disks and this bit string.) The next disk to move is $D_1$; it moves clockwise to peg $P_0$, and the last bit is complemented. The successor in the Gray code is the string 110010. After that, $D_1$ pauses for one step, while disk $D_3$ moves, again clockwise, from $P_1$ to $P_2$, and the third bit from the right is complemented, leading to the string 110110. For moving a tower of height $n$, one has to move disk $D_n$ at some point. But before moving disk $D_n$ from peg $A$ to $B$, one has to move the disks $D_1, \ldots, D_{n-1}$, which lie on top of $D_n$, out of the way, onto the third peg. After moving $D_n$ to $B$, these disks have to be moved from the third peg to $B$. This reduces the problem for a tower of height $n$ to two towers of height $n - 1$, leading to the following recursive procedure.

```python
move_tower(k, A, B): (Move the k smallest disks $D_1 \ldots D_k$ from peg $A$ to peg $B$)
    if k <= 0: return
    auxiliary := 3 - A - B; (auxiliary is the third peg, different from $A$ and $B$.)
    move_tower(k - 1, A, auxiliary)
    move disk $D_k$ from $A$ to $B$
    move_tower(k - 1, auxiliary, B)
```
1.4 Connections between the Towers of Hanoi and Gray codes

The delta sequence of the Gray code is the sequence $1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, \ldots$ of bit positions that are updated. (In contrast to the usual convention, we number the bits starting from 1.) This sequence has an obvious recursive structure which results from (1). It also describes the number of changed bits when incrementing from $i$ to $i + 1$ in binary counting. Moreover, it is easy to observe that the same sequence also describes the disks that are moved by the recursive algorithm $\text{move$_t$ower}$ above. It has thus been noted that the Gray code $G_n$ can be used to solve the Tower of Hanoi puzzle, cf. Scorer, Grundy, and Smith [8] or Gardner [4]. In the other direction, the Tower of Hanoi puzzle can be used to generate the Gray code $G_n$, see Buneman and Levy [2].

Several loopless ways to compute the next move for the Towers of Hanoi are known, and they lead directly to loopless algorithms for the Gray code. We describe one such algorithm.

1.5 Loopless Tower of Hanoi and binary Gray code

From the recursive algorithm $\text{move$_t$ower}$, it is not hard to derive the following fact.

\begin{itemize}
  \item \textbf{Proposition 1.} If the tower should be moved from $P_0$ to $P_1$ and $n$ is odd, or if the tower should be moved from $P_0$ to $P_2$ and $n$ is even, the moves of the odd-numbered disks always proceed in forward ("clockwise") circular direction: $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_0$, and the even-numbered disks always proceed in the opposite circular direction: $P_0 \rightarrow P_2 \rightarrow P_1 \rightarrow P_0$.
\end{itemize}

In the other case, when the assumption does not hold, the directions are simply swapped. Since we are interested not in moving the tower to a specific target peg, but in generating the Gray code, we stick with the proposition as stated.

\textbf{Algorithm HANOI.} Loopless algorithm for the binary Gray code.

Initialize: Put all disks on $P_0$.

\begin{algorithm}
  \textbf{loop}:
  \begin{itemize}
    \item Move $D_1$ clockwise.
    \item Let $D_k$ be the smaller of the topmost disks on the two pegs that don’t carry $D_1$.
    \item If there is no such disk, terminate.
    \item Move $D_k$ clockwise if $k$ is odd; otherwise, move it counterclockwise.
  \end{itemize}
\end{algorithm}

To obtain the Gray code, we simply set $a_k := 1 - a_k$ whenever we move the disk $D_k$. See Fig. 2 for a snapshot of the procedure.

We would not need the clockwise/counterclockwise rule for $D_k$: Since we must not put $D_k$ on top of $D_1$, there is anyway no choice of where to move it. We have chosen the above formulation since it is better suited for generalization.

1.6 Overview

In the remainder of this paper, we will generalize these connections to Gray codes for larger radices (alphabet sizes). Section 2 is devoted to ternary Gray codes and their connections to the so-called Tower of Bucharest. After defining Gray codes with general radices in Section 3, we extend the ternary algorithm from Section 2 to arbitrary odd radices $m$ in Section 4 and even to mixed (odd) radices (Section 6). In Section 5, we generalize the binary Gray code algorithm of Section 1.5 to arbitrary even $m$. Finally, in Section 8, we develop loopless algorithms based on an entirely different idea of “working ahead” that is related to converting amortized running-time bounds to worst-case bounds. In the appendix, we give prototype code for simulating all our algorithms in Python.
2 Ternary Gray codes and the Towers of Bucharest

A ternary Gray code enumerates the $3^n$ $n$-tuples $(a_n, \ldots, a_1)$ with $a_i \in \{0, 1, 2\}$. Successive tuples differ in one entry, and in this entry they differ by ±1.

The following simple variation of the Towers of Hanoi will yield a ternary Gray code ($m = 3$): We disallow the direct movement of a disk between pegs $P_0$ and $P_2$: a disk can only be moved to an adjacent peg. We call this the Towers of Bucharest.

Figure 3 shows the state space of the Towers of Bucharest in comparison with the Towers of Hanoi. In accordance with this figure, we can make the following easy observations:

- **Proposition 2.**
  1. In the towers of Hanoi, there are three possible moves from any position, except when the whole tower is on one peg: In these cases, there are only two possible moves.
  2. In the towers of Bucharest, there are two possible moves from any position, except when the whole tower is on peg $P_0$ or $P_2$: In those cases, there is only one possible move.

---

1 The custom of naming variations of the Tower of Hanoi game after different cities, instead of using ordinary names such as “three-in-a-row” [7], has a long tradition. The name “Towers of Bucharest” has been suggested by Günter M. Ziegler. Several legends rank themselves around these towers. A little count from Transylvania had conquered the whole country and had become a powerful Lord. In order to celebrate his glory, he built a magnificent palace in the capital city Bucharest, and he suppressed his people as best he could. He also had a dog named Heisenberg. In a nearby monastery, the monks had golden disks of different sizes on three pegs, and they had played the Towers of Hanoi game for centuries. It was already foreseeable that the game was drawing towards its conclusion. According to an ancient prophecy, the palace of the ruler of the country would crumble and his rule would come to an end when the game would be finished. When the count, who called himself king by this time, heard this story, he did not like it. First of all, he had the monks beheaded and told them to do some useful job instead. Secondly, he removed the pegs with the discs and took them to his palace. He made sure that they were placed very far away from each other: The first peg was put in the South wing, the second peg in the North wing, and the third peg again in the South wing. One may wonder why he did not place them in some more logical arrangement like South-South-North or South-North-North, or perhaps North-middle-South or South-middle-North. The reader will soon understand that this placement was a clever decision when she or he learns what else he did. The North wing could only be reached from the South wing through the middle wing, or by going out on the street and reentering on the other side, but I don’t think it is very wise to go into the street carrying a heavy golden disk. Anyway, his third action was his most wicked and smartest move: It occurred to him that he was powerful and he was the ruler, and he therefore had the power to change the rules. He decreed that the discs can only be moved between the first and the second peg or between the second and the third peg. Direct moves between the first and third peg were henceforth forbidden. This would delay the moves of
Proof. 1. The disk $D_1$ can be moved to any of the other pegs (two possible moves). In addition, the smaller of the topmost disks on the other pegs (if those pegs aren’t both empty) can be moved to the other peg which is not occupied by $D_1$.

2. If the disk $D_1$ is in the middle, it can be moved to any of the other pegs, but no other move is possible. If the disk $D_1$ is on $P_0$ or $P_2$, it has only one possible move, and the smaller of the topmost disks on the other pegs (if those pegs aren’t empty) also has one possible move, similarly as above.

Both games have the same set of $3^n$ states, corresponding to the possible ways of assigning each disk to one of the pegs $P_0$, $P_1$, $P_2$. The nodes in the corners marked $P_0$, $P_1$, $P_2$ represent the states where all disks are on one peg. The graph of the Towers of Hanoi in Figure 3 approaches the Sierpiński gasket. The optimal path of length $2^n - 1$ is the straight path from $P_0$ to $P_2$. (The directions of the edges in this drawing of the state graph are not directly related to the pegs that are involved in the exchange, and the relation between a state and its position on the drawing is not straightforward.) By contrast, we see that the graph of the Towers of Bucharest in Figure 3 is a single path through all nodes.

Let us see why this is true. By Proposition 2 this graph has maximum degree 2, and it follows that it must consist of a path between $P_0$ and $P_2$ (the only degree-1 nodes), plus a number of disjoint cycles. However, it is known that the path has length $3^n - 1$ and does therefore indeed go through all nodes. Since we will prove a more general statement later (Theorem 3), we only sketch the argument here: Solving the problem recursively in an analogous way to the procedure move_tower, we reduce the problem of moving a tower of $n$ disks from $P_0$ to $P_2$ (or vice versa) to three problem instances with $n - 1$ disks, plus two movements of disk $D_n$, and the resulting recursion establishes that $3^n - 1$ moves are required.

The states of the Towers of Bucharest correspond in a natural way to the ternary $n$-tuples: The digit $a_i \in \{0, 1, 2\}$ gives the position of disk $D_i$. It follows now easily that the solution of the Towers of Bucharest yields a ternary Gray code: Since we can move only one disk at a time, it means that we change only one digit at a time, and by the special rules of the Towers of Bucharest, we change it by ±1. (This connection has apparently not been made before.) In fact, the algorithm produces the ternary reflected Gray code, which we are about to define below in Section 3; see also Theorem 3.
Moreover, since there are only two possible moves, one just has to always choose the move which does not undo the previous move, and this leads to a very easy loopless Gray code enumeration algorithm.

It is remarkable that ternary Gray codes can be generated on the same hardware as binary Gray codes (Fig. 2). In the context of generating the ternary Gray code, the Gray code string can be directly read off the disks. For example, the configuration in Fig. 2 represents the string 211102. It is $D_1$’s turn to move, and the disk $D_1$ will make two steps to the left, generating the strings 211101 and 211100, and pauses there for one step, while disk $D_3$ moves to the right, leading to the string 211200, etc.

3 **Gray codes with general radixes**

An $m$-ary Gray code enumerates the $n$-tuples $(a_0, \ldots, a_{n-1})$ with $0 \leq a_i < m$, changing a single digit at a time by $\pm 1$. The $m$-ary reflected Gray code can be recursively described as follows: Let $C_1, C_2, \ldots, C_m$ be the Gray code for the strings of length $n$. Then the strings of length $n+1$ are generated in the order

\[
C_0, C_1, C_2, \ldots, C_m (m-2), C_1 (m-1), C_2 (m-1), C_3 (m-2), \ldots, C_2 (m-2), C_1 (m-1), C_0 (m-1), C_2 (m-2), \ldots, C_3 (m-2), C_4 (m-1), \ldots
\]

Each digit alternates between an upward sweep from 0 to $m-1$ and a return sweep from $m-1$ to 0.

4 **Generating the $m$-ary Gray code with odd $m$**

For odd $m$, the ternary algorithm can be generalized. We need $m$ pegs $P_0, \ldots, P_{m-1}$. The leftmost peg $P_0$ and the rightmost peg $P_{m-1}$ play a special role. The other pegs are called the *intermediate pegs*.

**Algorithm ODD.** Generation of the $m$-ary Gray code for odd $m$.

Initialize: Put all disks on $P_0$.

**loop:**
- Move $D_1$ for $m-1$ steps, from $P_0$ to $P_{m-1}$ or vice versa.
- Let $D_k$ be the smallest of the topmost disks on the $m-1$ pegs that don’t carry $D_1$.
- If there is no such disk, terminate.
- Move $D_k$ by one step:
  - If $D_k$ is on $P_0$ or $P_{m-1}$, there is only one possible direction where to go.
  - Otherwise, the disk $D_k$ continues in the same direction as in its last move.

---

cowboy hat, and the woman was in her bikini. After all, it might have been the Towers of Hanoi that they played. Some witnesses have later reported that they had seen a disc jumping between the first and the third peg, but this has never been conclusively confirmed.

Mary that as it be, something unexpected happened. As the khalip was engrossed in watching the spectacle and drew a bit closer, a small door in the wall beside him opened, which he had not noticed before. It gave onto a small garden. The moon had risen over the rooftops, and her light gave a sort of surreal atmosphere to the whole scene. In the middle of the garden, at the corner of a fountain, a woman sang, accompanying herself on the lute. She had a beautiful voice, a bit like Mariah Carey or Adele. The calif would have listened longer, but he was quickly escorted into a house, where a maid-servant took charge of him.

Rashid (ca. 763–809). Should he fail to fulfill this task, the other eight will kill him.

As the khalip was engrossed in watching the spectacle and drew a bit closer, a small door in the wall beside him opened, which he had not noticed before. It gave onto a small garden. The moon had risen over the rooftops, and her light gave a sort of surreal atmosphere to the whole scene. In the middle of the garden, at the corner of a fountain, a woman sang, accompanying herself on the lute. She had a beautiful voice, a bit like Mariah Carey or Adele. The calif would have listened longer, but he was quickly escorted into a house, where a maid-servant took charge of him.

May that as it be, something unexpected happened. As the khalip was engrossed in watching the spectacle and drew a bit closer, a small door in the wall beside him opened, which he had not noticed before. It gave onto a small garden. The moon had risen over the rooftops, and her light gave a sort of surreal atmosphere to the whole scene. In the middle of the garden, at the corner of a fountain, a woman sang, accompanying herself on the lute. She had a beautiful voice, a bit like Mariah Carey or Adele. The calif would have listened longer, but he was quickly escorted into a house, where a maid-servant took charge of him.

"Gentlemen. You have sworn to come to my rescue when I would be in need. Now the time has come to fulfill your oath. You see seven discs of different sizes. He who will draw the smallest disc will bring me the head of the detestable caliph Harun-al-Rashid (ca. 783–809). Should he fail to fulfill this task, the other eight will kill him, and we will come together and draw again." With these words, she dropped the discs into a chalice. In silence, each man picked a disc. The khalip was last to draw. As he opened his hand, sure enough, he found the smallest disc, disc number 1. He rose and said: "Fair lady, I will fulfill your order as I have promised. But pray tell me: by which deeds or words has the khalif enraged you so much that you wish him to
Figure 4 shows an example with $m = 5$. The game with 5 pegs is called the Tower of Klagenfurt, after the birthplace of the senior author.

**Figure 4** The Towers of Klagenfurt. This configuration represents the string 321411 over the radix $m = 5$. The next step of the Gray code moves the smallest disk $D_1$ onto peg $P_0$, changing the string to 321410. After that, disk $D_2$ moves from $P_1$ to $P_2$ and the next string is 321420. In the background, the two-headed Lindworm monster.

In this procedure, the movement of $D_1$ is “externally given”, whereas the movement of the other disks, whenever $D_1$ is at rest, is somehow “determined by the algorithm”. It is not obvious that the algorithm does not put a larger disk on top of $D_1$.

▶ **Theorem 3.** Algorithm ODD generates the $m$-ary reflected Gray code defined in (2).

**Proof.** It is clear from the algorithm that the last digit, which is controlled by the movement of $D_1$, changes in accordance with (2). We still have to show that when we discard the last digit and observe only the movement of the disks $D_2, \ldots, D_n$, the algorithm produces the Gray code for the strings of length $n - 1$. This is proved by induction.

By the rules of the algorithm, whenever $D_1$ rests, the disk that moves is $D_2$, unless $D_2$ is covered by $D_1$. Let us now observe the motion pattern of $D_1$ and $D_2$ that results from this rule. We start with $D_1$ on top of $D_2$, say, on peg $P_0$, with $D_1$ about to start its sweep. Whenever $D_1$ pauses for one step, $D_2$ will make a step towards $P_{m-1}$. After $D_2$ reaches $P_{m-1}$, it turns out that, because $m$ is odd, $D_1$ will make its next sweep from $P_0$ to $P_{m-1}$, resting on top of $D_2$. Now, since $D_2$ is covered, it will be one of the other disks $D_3, D_4, \ldots$ that will move. Then the same routine repeats in the other direction.

If we now ignore $D_1$ and look only at the motions of the other disks, the following pattern emerges: $D_2$ makes $m - 1$ steps from one end to the other, and then the smallest disk that is not covered by $D_2$ makes its move, according to the rules. This is precisely the same procedure as Algorithm ODD, with $D_2$ taking the role of the “externally controlled” disk $D_1$, and we have assumed by induction that this algorithm correctly produces the Gray code for the strings of length $n - 1$, and it does not put a larger disk on top of $D_2$. Since the larger disks are moved only when $D_2$ lies under $D_1$, it follows that a larger disk cannot be moved on top of $D_1$ either.

---

2 When Klagenfurt was founded, it was surrounded by a swamp. The swamp was inhabited by a dinosaur, the so-called Lindworm. The Lindworm would regularly come to the city and eat citizens. Occasionally, she would devour one of the towers of the city. The coat of arms of Klagenfurt shows the Lindworm dragon in front of the only remaining tower, see Figure 4. (Initially, there were five towers.) Over the centuries, the swamp has been drained, and the Lindworm is practically extinct.
One can actually apply one induction step of the proof in the opposite direction, introducing an additional “control disk” $D_0$ which does not have a digit associated with it. Its only role is to alternate cover $P_0$ and $P_{m-1}$ and exclude these pegs from the selection of the disk $D_k$ that should be moved. The algorithm becomes simpler because it does not have to treat $D_1$ separately from the other disks. (See Appendix A.3 where this idea is applied to the algorithm of Section 6 below).

### 5 Generating the $m$-ary Gray code with even $m$

For even $m$, we generalize Algorithm HANOI, which solves the case $m = 2$. We use $m + 1$ pegs $P_0, \ldots, P_m$, which we arrange in a cyclic clockwise order. We stipulate that disks $D_i$ with odd $i$ move only clockwise, and disks with even $i$ move only counterclockwise.

**Algorithm EVEN.** Generation of the $m$-ary Gray code for even $m$.

Initialize: Put all disks on $P_0$.

loop:
- Move $D_1$ for $m - 1$ steps, in clockwise direction.
- Let $D_k$ be the smallest of the topmost disks on the $m$ pegs that don’t carry $D_1$.
- If there is no such disk, terminate.
- Move $D_k$ by one step, in the direction determined by the parity of $k$.

The Gray code is determined by changing the digit $a_i$ whenever disk $D_i$ is moved. The digit $a_i$ runs through the sequence $0, 1, 2, \ldots, m - 2, m - 1, 1, 2, \ldots, 2, 1, 0, 1, 2, \ldots$. Thus it changes always by $\pm 1$, but we have to remember whether it is on the increasing or the decreasing part of the cycle. The position of disk $D_i$ is no longer directly correlated with the digit $a_i$; thus the digits $a_i$ have to be maintained separately, in addition to the disks on the pegs. It is far from straightforward to relate the disk configuration to the Gray code.

For example, when carrying out the algorithm for $m = 4$, the configuration in Figure 4 appears when the string is 211030. Disk $D_1$ has just made three steps and is going to rest for one step. The next step moves $D_3$ clockwise, since $3$ is odd, and the string is changed to 211130. After that, $D_1$ resumes its clockwise motion, and the string changes into 211131.

**Theorem 4.** Algorithm EVEN generates the $m$-ary reflected Gray code defined in (2).

**Proof.** This follows along the same lines as Theorem 3. When we look at the pattern of motion of $D_1$ and $D_2$, we observe again that $D_2$ makes $m - 1$ steps until it is covered by $D_1$, see Fig. 5. After the first move of $D_2$, the clockwise cyclic distance from $D_1$ to $D_2$ is 1, and with each move of $D_2$, this distance increases by 1. Thus, after $m - 1$ moves, the distance becomes $m - 1$, and $D_1$ will land on top of $D_2$ with its next sweep.

Algorithms ODD and EVEN do not generate a shortest sequence of moves to the target configuration except when $m = 3$ or $m = 2$. We could not come up with some set of natural constraints under which our algorithms give a shortest solution.

### 6 The Towers of Bucharest++

In Algorithm ODD, the intermediate pegs $P_1, \ldots, P_{m-2}$ will always be available for selecting the smallest disk $D_k$ to be moved. Thus, one can coalesce these pegs into one peg, keeping only the two extreme pegs separate. With three pegs, we can use the same hardware as the tower of Bucharest, but we have to record the value of the digits, since they are no longer expressed by the position. A simple method is to provide the disks with marks that indicate
Figure 5 One period of movement of the two smallest disks $D_1$ and $D_2$ when Algorithm EVEN generates all tuples over an alphabet of size $m = 4$ using $m + 1 = 5$ pegs.

the value as well as the direction of movement, which we have to remember anyway. Each disk cycles through $2m - 2$ values, potentially augmented with direction information:

$$0, 1↑, 2↑, \ldots, (m-2)↑, m-1, (m-2)↓, \ldots, 2↓, 1↓, 0, 1↑, \ldots$$

(3)

It makes sense to encode this information like a dial with $2m - 2$ equally spaced directions, as shown in Fig. 6a. A disk whose mark shows 0 is always on the left peg $P_0$. A disk whose mark shows $m - 1$ is always on the right peg $P_2$. Otherwise, it is on the middle peg $P_1$.

Algorithm ODD-COMPRESSED. Generation of the $m$-ary Gray code for odd $m$.

Initialize: Put all disks on $P_0$, and turn them to show 0.

loop:

- Turn disk $D_1$ $m - 1$ times until it arrives at one of the extreme pegs $P_0$ or $P_2$.
- Let $D_k$ be the smaller of the topmost disks on the two pegs not covered by $D_1$.
- If there is no such disk, terminate.
- Turn $D_k$ once.
The digits $a_i$ can be read off from the dial positions. Correctness follows by comparison with Algorithm ODD, checking that the transition between successive states is preserved when merging the intermediate pegs into one peg.

This algorithm can now even be generalized to mixed-radix Gray codes for the $n$-tuples $(a_n, \ldots, a_1)$ with $0 \leq a_i < m_i$, for some sequence of radices $m_i \geq 2$, provided that all $m_i$ are odd.

7 Simulation

All our algorithms can be easily simulated in software on a digital computer. A stack will do for each peg. If there are $k$ pegs, the algorithm takes $O(k)$ time to compute the next move and accordingly produce the next element of the Gray code sequence. If $m$ is constant, then $k = m$ or $k = m + 1$ in Algorithms ODD and EVEN, and these algorithms can pass as loopless algorithms. If $k$ is large, Algorithm ODD can be replaced by ODD-COMPRESSED, which has only 3 pegs, independent of $m$.

To make a truly loopless algorithm out of Algorithm EVEN, at the expense of an increased overhead, we can use the following easy fact, which follows directly from the algorithm statement.

Lemma 5. In the algorithms EVEN, ODD, and ODD-COMPRESSED, when a disk $D_k$ is moved, all smaller disks $D_1, \ldots, D_{k-1}$ are on the same peg.

To get a loopless implementation, the set of disks on a peg has to be maintained as a sequence of maximal intervals of successive integers, instead of storing them as a stack in the usual way. Then, whenever $D_1$ is at rest, the disk $D_k$ to be moved can be determined in constant time as the smallest missing disk on the peg containing $D_1$.

8 Working ahead

While we are at the topic of Gray codes, we might as well mention another approach for loopless generation of Gray codes, which results from a generally applicable technique for converting amortized bounds into worst-case bounds. We start from the observation that was already mentioned in connection with the delta-sequence in Section 1.4:

Proposition 6. Consider the enumeration of the $n$-tuples $(b_n, \ldots, b_1)$ with $0 \leq b_i < m_i$ in lexicographic order. If, between two successive tuples of the sequence, the $j$ rightmost digits are changed, then, at the corresponding transition in the Gray code, the $j$-th digit from the right is changed.

We can thus find the position $j$ that has to be changed in the Gray code by lexicographically “incrementing” $n$-tuples $(b_n, \ldots, b_1)$ in a straightforward way:

Nowadays, most households will more readily have access to a computer than to towers of Hanoi.
Algorithm DELTA. Generation of the delta-sequence for the Gray code.

Initialize: \((b_n, \ldots, b_2, b_1) := (0,0,\ldots,0,0)\)

\(Q := \) an empty list

\begin{algorithm}
\textbf{loop:}
\begin{algorithmic}
\State \(j := 1\)
\While {\(b_j = m_j - 1\)}
\State \(b_j := 0\)
\State \(j := j + 1\)
\EndWhile

\If {\(j = n + 1\)}
\State \text{TERMINATE}
\EndIf
\State \(b_j := b_j + 1\)
\State \(Q\).append\((j)\)
\Endalgorithmic
\end{algorithm}

The delta sequence is stored in \(Q\). It is known that the average number of loop iterations for producing an entry of \(Q\) is less than 2. We use this fact to coordinate the production of entries \(Q\) by Algorithm DELTA with their consumption in the Gray code generation, turning \(Q\) into a buffer of bounded capacity. This leads to the following loopless algorithm:

Algorithm WORK-AHEAD. Generation of the Gray code.

\begin{algorithm}
\textbf{procedure} STEP:
\begin{algorithmic}
\If {\(b_j = m_j - 1\)}
\State \(b_j := 0\)
\State \(j := j + 1\)
\Else:
\If {\(Q\) is not filled to capacity:}
\State \(b_j := b_j + 1\)
\State \(Q\).append\((j)\)
\State \(j := 1\)
\EndIf
\EndIf
\State \text{STEP}
\Endalgorithmic
\end{algorithm}

The procedure STEP on the left side encompasses one loop iteration of Algorithm DELTA. By programmer's license, we have moved the initialization \(j := 1\) of the loop variable to the end of the previous loop. We have also moved the termination test \(j = n + 1\) to the side of the consumer. Accordingly, we had to extend the \(n\)-tuple \(b\) into an \((n+1)\)-tuple, setting \(m_{n+1}\) arbitrarily to 2. When \(Q\) is full, nothing is done in the procedure STEP, and the repeated call of STEP will try to insert the same value into \(Q\). Thus, apart from the termination test, a repeated execution of STEP will faithfully carry out Algorithm DELTA.

The clean way to terminate the algorithm would be to stop inserting elements into \(Q\) as soon as \(j = n + 1\) is produced in STEP, as in Algorithm DELTA. Instead, termination is triggered when the value \(j = n + 1\) is removed from \(Q\). Due to this delayed termination test, a few more iterations of STEP can be carried out, but they cause no harm.

The Gray code algorithm on the right couples two production STEP's with one consumption step, which takes out an entry \(j\) of \(Q\) and carries out the update \(a_j := a_j + 1\). Every digit \(a_j\) must cycle up and down through its values in the sequence (3), and thus, we have to remember the direction \(d_j = \pm 1\) in which it moves, as in Algorithm ODD.

To show that the algorithm is correct, we have to ensure that the queue \(Q\) is never empty when the algorithm retrieves an element from it. This is proved below in Lemma 7.

The binary Gray code \((m_i = 2\) for all \(i = 1, \ldots, n)\), the algorithm can be simplified. With a slightly larger buffer \(Q\) of size \(B' := \max\{\lceil\frac{n+1}{2}\rceil, 2\}\), the test whether \(Q\) is filled to
capacity can be omitted, see Lemma 9 below. The reason is that the average number of production STEP's per item approaches 2 in the limit, and accordingly, the queue automatically does not grow beyond the minimum necessary size. The directions \( d_i \) are of course also superfluous in the binary case.

The idea of “working ahead” is opposite to the approach of delaying work as long as possible that underlies many “lazy” data structures and also lazy evaluation in some functional programming languages. In a similar vein, Guibas, McCreight, Plass, and Janet R. Roberts [5] have obtained worst-case bounds of \( O(\log k) \) for updating a sorted linear list at distance \( k \) from the beginning. Their algorithm works ahead to hedge against sudden bursts of activity. Our setting is much simpler, because we do not depend on the update requests of a “user” and we can plan everything in advance.

At a different level of complexity, the idea of working ahead occurs in an algorithm of Wettstein [9, Section 6]. This trick, credited to Emo Welzl, is used to achieve polynomial delay between successive solutions when enumerating non-crossing perfect matching of a planar point set, despite having to build up a network with exponential space in a preprocessing phase.

### 8.1 An alternative STEP procedure

As an alternative to the organization of Algorithm WORK-AHEAD, we can incorporate the termination test into the STEP procedure:

```plaintext
procedure \text{STEP}':\n  if \( j = n + 1 \): TERMINATE
  if \( b_j = m_j - 1 \):
    b_j := 0
    j := j + 1
  else:
    if \( Q \) is not filled to capacity:
      b_j := b_j + 1
      Q.append(j)
      j := 1
```

With this modified procedure \text{STEP}', the termination test in the main part of Algorithm WORK-AHEAD can of course be omitted. We also need not extend the arrays \( b \) and \( m \) to \( n + 1 \) elements.

The algorithm still works correctly because there are no unused entries in the queue when \text{STEP}' signals termination. Let us prove this:

The termination signal is sent instead of producing the value \( j = \bar{\rho}(k) = n + 1 \) for \( k = m_0 m_1 \ldots m_{n-1} \). Generating this signal takes \( n + 1 \) iterations of STEP. In this time, no new values are added to the queue. Let us assume that the production of \( \bar{\rho}(k) \) was started during iteration \( k_0 \), and the buffer was filled with \( B_0 \leq B \) entries at that time. The first of these entries is consumed at the end of iteration \( k_0 \), and all \( B_0 \) entries of the buffer have been used up at the beginning of iteration \( k_0 + B_0 \). By this time, at most \( 2B_0 \leq 2B \leq n + 1 \) iterations of STEP were carried out and contributed to the production of the termination signal. It follows that when STEP discovers that \( j = n + 1 \), no unused entries are in the stack, and it is safe to terminate the program.

---

4 We thank Don Knuth for leading us to this reference.
It is important not to “speed up” the program by moving the termination test into the \texttt{if}-branch after the statement \(j := j + 1\). Also, we must use exactly the prescribed buffer size for \(Q\). Therefore, this variation is incompatible with the simplification for the binary case mentioned above.

### 8.2 Correctness proofs for the work-ahead algorithms

We define the ruler function \(\rho\) and the modified ruler function \(\bar{\rho}\) with respect to a sequence of radixes \(m_1, \ldots, m_n\) as follows:

\[
\rho(k) := \max\{ i : 0 \leq i \leq n, \ m_1m_2 \ldots m_i \text{ divides } k \}, \quad \bar{\rho}(k) := \rho(k) + 1
\]

Then the \(k\)-th value that is entered into \(Q\) is \(\bar{\rho}(k)\), and for computing this value, Algorithm DELTA needs \(\bar{\rho}(k)\) iterations, and accordingly, Algorithm WORK-AHEAD needs \(\bar{\rho}(k)\) STEPs.

\[\text{Lemma 7. In Algorithm WORK-AHEAD, the buffer } Q \text{ never becomes empty.}\]

\[\text{Proof. We number the iterations of the main loop as } 1, 2, \ldots, m_1m_2 \ldots m_n. \text{ In the last iteration, the algorithm terminates.}
\]

(i) Up to and including iteration \(k\), two repetitions of STEP were always completed.

(ii) Some repetitions of STEP had no effect because the buffer \(Q\) was full.

In case (i), production of all values \(\rho(i)\) for \(i = 1, \ldots, k\) requires

\[
S(k) := \sum_{i=1}^{k} \bar{\rho}(i)
\]
calls to STEP. To show that these calls are completed by the time when \(\bar{\rho}(k)\) is needed, we have to show

\[
S(k) \leq 2k.
\]

In case (ii), let \(k_0\) be the last iteration when an execution of STEP was “skipped”. This means that the queue \(Q\) was filled to capacity \(B\) just before removing the value \(j = \bar{\rho}(k_0)\), and it contained the values \(\bar{\rho}(k_0), \bar{\rho}(k_0 + 1), \ldots, \bar{\rho}(k_0 + B - 1)\). Since then, STEP was called \(2(k - k_0)\) times, and \(\bar{\rho}(k)\) is ready when it is needed, provided that

\[
1 + \sum_{i=k_0+B}^{k} \bar{\rho}(i) \leq 2(k - k_0)
\]

whenever \(k \geq k_0 + B\). The left-hand side of this inequality is the number of necessary STEPs for computing the values up to \(\bar{\rho}(k)\). Computing \(\bar{\rho}(k_0 + B)\) takes just one more STEP, since the STEP that would have stored this value in \(Q\) was abandoned in iteration \(k_0\). Setting \(k' = k_0 + B\), we can express the inequality equivalently as

\[
S(k) - S(k') \leq 2(k - k' + B) - 1 \text{ for } k' \leq k
\]

We can write an explicit formula for \(S(k)\):

\[
S(k) = k + \left\lfloor \frac{k}{m_1} \right\rfloor + \left\lfloor \frac{k}{m_1m_2} \right\rfloor + \cdots + \left\lfloor \frac{k}{m_1m_2 \ldots m_n} \right\rfloor
\]
Since all \( m_i \geq 2 \), we get \( S(k) \leq k + k/2 + k/4 + k/8 + \cdots + k/2^n < 2k \), proving (4). For the other bound (5), we use the relation
\[
\left\lfloor \frac{k}{x} \right\rfloor - \left\lfloor \frac{k'}{x} \right\rfloor < \frac{k - k'}{x} + 1
\]
to get
\[
S(k) - S(k') < (k - k') + (k - k') \cdot \left( \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} \right) + n < 2(k - k') + n
\]
Since the left-hand side is an integer, we obtain \( S(k) - S(k') \leq 2(k - k') + n - 1 \) and this implies (5) since the buffer size \( B := \lceil \frac{n}{2} \rceil \) satisfies \( 2B \geq n \).

In Algorithm WORK-AHEAD, the STEP should generate entries \( \bar{\rho}(1), \bar{\rho}(2), \ldots \) of \( Q \) up to \( \bar{\rho}(N) \), where \( N := m_1 m_2 \cdots m_n \).

The following lemma shows that production of STEP may overrun their target by at most one. Since the algorithm has already made provisions to generate \( \bar{\rho}(N) = n + 1 \) by extending the arrays \( b \) and \( m \) to size \( n + 1 \) instead of \( n \), this one extra entry does not cause any harm.

**Lemma 8.** In Algorithm WORK-AHEAD, the last entry that is added to \( Q \) is \( \bar{\rho}(N) \) or \( \bar{\rho}(N + 1) \).

**Proof.** The production of \( \bar{\rho}(N) = n + 1 \) takes \( n + 1 \geq 2B \) STEP. It follows that the buffer \( Q \) is empty when \( \bar{\rho}(N) = n + 1 \) is inserted, regardless of whether the production of \( \bar{\rho}(N) \) is started in the first or second STEP of an iteration.

If the production of \( \bar{\rho}(N) = n + 1 \) is completed in the second STEP of an iteration, it is thus immediately consumed, which leads to termination. If \( \bar{\rho}(N) \) is completed in the first STEP of an iteration, the second STEP will produce the value \( \bar{\rho}(N + 1) = 1 \), but then the algorithm will terminate as well.

Finally, we prove the simplification of the algorithm for the binary case.

**Lemma 9.** In the binary version of Algorithm WORK-AHEAD, i.e., when \( m_i = 2 \) for all \( i = 1, \ldots, n \), the buffer \( Q \) automatically never gets more than \( B' := \max\{\lceil \frac{n+1}{2} \rceil, 2 \} \) entries, even if the test in STEP whether the buffer is full is omitted.

**Proof.** Let us assume for contradiction that the buffer becomes overfull in iteration \( k \), \( 1 \leq k \leq 2^n \). This means that, before \( j = \bar{\rho}(k) \) is removed from \( Q \), the \( 2k \) STEP operations have produced more than \( k - 1 + B' \) values. But this is impossible, since, as we will show, the production of the first \( k_1 = k + B' \) values would have taken
\[
S(k_1) = k_1 + \left\lfloor \frac{k_1}{2} \right\rfloor + \left\lfloor \frac{k_1}{2^2} \right\rfloor + \cdots + \left\lfloor \frac{k_1}{2^n} \right\rfloor > 2k
\]
STEPs. To show the last inequality, we first consider the case \( k_1 < 2^n \). We apply the inequality \( \lfloor x \rfloor > x - 1 \) and obtain \( S(k_1) > 2k_1 - k_1/2^n - n \), and since \( k_1/2^n < 1 \) and \( S(k_1) \) is an integer, we get
\[
S(k_1) \geq 2k_1 - n = 2k + 2B' - n > 2k.
\]
Let us now see at what time \( \bar{\rho}(k_1) \) for \( k_1 \geq 2^n \) is entered into \( Q \). When \( k_1 = 2^n \), no round-off takes place in the formula for \( S(k_1) \), and we have \( S(2^n) = 2 \cdot 2^n - 1 \). This shows that the production of \( \bar{\rho}(2^n) \) is completed in the first STEP of iteration \( 2^n \). In the second STEP of this iteration, \( \bar{\rho}(2^n + 1) = 1 \) is added to \( Q \). Thus, when \( \bar{\rho}(2^n) \) is about to be retrieved, the buffer contains \( 2 \leq B' \) elements. Then the algorithm terminates, and no more elements are produced.
9 Concluding Remarks

By our approach of modeling the Gray code in terms of a motion-planning game, we were able to get a mixed-radix Gray code only when all radices $m_i$ are odd. It remains to find a model that would work for different even radices or even for radices of mixed parity.

Another motion-planning game which is related to the binary Gray code is the Chinese rings puzzle, see Scorer, Grundy, and Smith [8], Gardner [4], or Knuth [5, pp. 285–6]. In this puzzle, there are at most two possible moves in every state, like in the towers of Bucharest, and by simulating the Chinese rings directly, one can obtain another loopless algorithm for the binary Gray code. However, this algorithm does not seem to extend to other radices.

References

1 James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected Gray code and its applications. Commun. ACM, 19(9):517–521, 1976.
2 Peter Buneman and Leon Levy. The towers of Hanoi problem. Information Processing Letters, 10(4–5):243–244, 1980.
3 Gideon Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. J. Assoc. Comput. Mach., 20(3):500–513, July 1973.
4 Martin Gardner. The curious properties of the Gray code and how it can be used to solve puzzles. Sci. American, 227:106–109, 1972.
5 Leo J. Guibas, Edward M. McCreight, Michael F. Plass, and Janet R. Roberts. A new representation for linear lists. In Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, STOC ’77, pages 49–60, New York, NY, USA, 1977. ACM.
6 Donald E. Knuth. Combinatorial Algorithms, Part 1, volume 4A of The Art of Computer Programming. Addison-Wesley, 2011.
7 Amir Sapir. The towers of Hanoi with forbidden moves. The Computer Journal, 47(1):20–24, 2004.
8 R. S. Scorer, P. M. Grundy, and C. A. B. Smith. Some binary games. The Mathematical Gazette, 28(280):96–103, 1944.
9 Manuel Wettstein. Counting and enumerating crossing-free geometric graphs. arXiv:1604.05350, April 2016.

A Appendix: PYTHON simulations of the algorithms

Here we list prototype implementations in PYTHON. They should run equally with Python 2.7 and Python 3. The pegs, the string $a$, and the array of directions are kept as global variables.
A.1 Basic procedures

The procedure visit prints the string and the contents of the pegs.

def visit():
    print ("".join(str(x) for x in reversed(a[1:])) + " "+".join("P{}:".format(k)+",".join(map(str,p))
           for k,p in enumerate(pegs)))

def find_smallest_disk(exclude=None):
    list_d_k = [(p[-1],k) for k,p in enumerate(pegs) if k!=exclude and p]
    if list_d_k:
        _,k = min(list_d_k) # smallest disk not covered by D1
        return k
    return None

A.2 Algorithm ODD, Section 4

def initialize_m_ary_odd(m,n):
    # n-tuple of entries from the set {0,1,...,m-1}
    global pegs, a, direction
    pegs = tuple([ ] for k in range(m))
    for i in range(n,0,-1):
        pegs[0].append(i)
    a = (n+1)*[0] # a[0] and direction[0] is wasted
    direction = (n+1)*[+1]

def Gray_code_m_ary_odd(m):
    visit()
    while True:
        for _ in range(m-1): # repeat m-1 times:
            move_disk(m,peg=a[1])
            visit()
        k = find_smallest_disk(exclude=a[1]) # smallest disk not covered by D1
        if k==None: return
        move_disk(m,peg=k)
        visit()

def move_disk(m,peg):
    # move topmost disk on peg
    disk = pegs[peg].pop()
    peg += direction[disk]
    a[disk] += direction[disk]
    if peg==m-1: direction[disk] = -1
    elif peg==0: direction[disk] = +1
    pegs[peg].append(disk)

    # run the program for a test:
    m=3
    initialize_m_ary_odd(m,6)
    Gray_code_m_ary_odd(m)
A.3 Algorithm ODD-COMPRESSED, Section 6

This implementation uses the idea of a "control disk" $D_0$ mentioned at the end of section 4. For uniformity, we attach a direction also to the dial positions 0 and $m - 1$, namely the direction in which the next move will proceed (in contrast to the convention used in Section 6).

```python
def initialize_m_ary_odd_compressed(m, n):
    # n-tuple of entries from the set {0,1,...,m-1}
    global pegs, a
    pegs = tuple([[] for k in range(3)])
    for i in range(n, 0, -1):
        pegs[0].append((i, 0, +1))
    a = (n+1)*[0]  # a[0] is wasted

def Gray_code_m_ary_odd_compressed(m):
    visit()
    while True:
        for control_disk in (2, 0):
            k = find_smallest_disk(exclude=control_disk)
            if k==None: return # All disks are on the same peg.
            move_disk_compressed(m, peg=k)
            visit()

def move_disk_compressed(m, peg):
    # retrieve topmost disk on peg:
    disk, value, direction = pegs[peg].pop()
    if value in (0, m-1) or value+direction in (0, m-1):
        peg += direction
        value += direction
        a[disk] += direction
        if value in (0, m-1):
            direction = -direction
        pegs[peg].append((disk, value, direction))

    # run the program for a test:
    m=5
    initialize_m_ary_odd_compressed(m, 5)
    Gray_code_m_ary_odd_compressed(m)
```
A.4 Algorithm EVEN, Section 5

def initialize_m_ary_even(m,n):
    initialize_m_ary_odd(m+1,n) # use m+1 pegs

def Gray_code_m_ary_even(m):
    peg_disk1 = 0 # position of disk D1
    visit()
    while True:
        for _ in range(m-1): # repeat m-1 times:
            turn_disk(m,peg=peg_disk1)
            peg_disk1 = (peg_disk1+1) % (m+1)
            visit()
        k = find_smallest_disk(exclude=peg_disk1)
            # smallest disk not covered by D1
        if k==None: return
        turn_disk(m,peg=k)
        visit()

def turn_disk(m,peg):
    # move the topmost disk on peg
    disk = pegs[peg].pop()
    if disk%2==1:
        peg = (peg + 1) % (m+1)
    else:
        peg = (peg - 1) % (m+1)
    pegs[peg].append(disk)
    a[disk] += direction[disk]
    if a[disk]==m-1: direction[disk] = -1
    elif a[disk]==0: direction[disk] = +1

# run the program for a test:
m=4
initialize_m_ary_even(m,6)
Gray_code_m_ary_even(m)

A.5 Truly loopless implementation of Algorithm EVEN, Section 7

A peg is a list of pairs \((a, b)\) with \(a \leq b\), denoting a maximal subset \(a, a+1, \ldots, b\) of consecutive disks (an interval). The pairs are sorted, with the smallest disks at the end (at the “top”).

def initialize_m_ary_even_intervals(m,n):
    global pegs,position
    initialize_m_ary_even(m,n)
    pegs[0][:]=(1,n)
    position=[0]*(n+1)
def Gray_code_m_ary_even_intervals(m, n):
    visit()
    while True:
        for _ in range(m-1):  # repeat m-1 times:
            turn_disk_intervals(m, peg=position[1])
            visit()
        d = find_smallest_missing_disk(position[1])
        # smallest disk not covered by D1
        if d > n: return
        turn_disk_intervals(m, peg=position[d])
        visit()

def turn_disk_intervals(m, peg):
    disk, d2 = pegs[peg][-1]
    # remove disk from peg:
    if disk == d2:
        pegs[peg].pop()
    else:
        pegs[peg][-1] = (disk + 1, d2)
    if disk % 2 == 1:
        peg = (peg + 1) % (m + 1)  # turn the disk "clockwise"
    else:
        peg = (peg - 1) % (m + 1)  # turn the disk "counterclockwise"
    position[disk] = peg
    # add disk to peg:
    if pegs[peg]:
        d1, d2 = pegs[peg][-1]
        if disk < d1 - 1:
            pegs[peg].append((disk, disk))
        else:
            pegs[peg][-1] = (disk, d2)
    else:
        pegs[peg].append((disk, disk))
    a[disk] += direction[disk]
    if a[disk] == m - 1: direction[disk] = -1
    elif a[disk] == 0: direction[disk] = +1

def find_smallest_missing_disk(peg):
    (_, d2) = pegs[peg][-1]
    return d2 + 1

# run the program for a test:
m = 4
initialize_m_ary_even_intervals(m, 6)
Gray_code_m_ary_even_intervals(m, 6)
A.6 Algorithm WORK-AHEAD, Section 8

```python
def STEP():
    global j, b, m, B, Q
    if b[j] == m[j] - 1:
        b[j] = 0
        j += 1
    else:
        if len(Q) < B:
            b[j] += 1
            Q.append(j)
            j = 1

def initialize_work_ahead(n):
    global a, b, direction, B, Q, j
    a = (n+1)*[0]  # a[0], b[0], and direction[0] is wasted
    direction = (n+1)*[+1]
    b = (n+2)*[0]
    from collections import deque
    B = (n+1)//2
    Q = deque()
    j = 1

def Gray_code_work_ahead(n):
    while True:
        VISIT()
        STEP()
        STEP()
        j = Q.popleft()
        if j == n+1: break
        a[j] += direction[j]
        if a[j] in (0, m[j] - 1): direction[j] *= -1

def VISIT():
    print ("".join(str(x) for x in reversed(a[1:])))

# run the program for a test:

n = 4
m = [0] + [2, 4, 5, 2] + [2]  # initial 0 and final 2 are artificial
initialize_work_ahead(n)
Gray_code_work_ahead(n)
```
A.7 Algorithm WORK-AHEAD for the binary Gray code, Section 8

```python
def STEP_binary():
    global j, b, B, Q
    if b[j] == 1:
        b[j] = 0
        j += 1
    else:
        if len(Q) >= B:
            print("error")
            exit(1)
        b[j] = 1
        Q.append(j)
        j = 1

def initialize_work_ahead_binary(n):
    global a, b, B, Q, j
    a = (n + 1) * [0]  # a[0], b[0], and direction[0] is wasted
    b = (n + 2) * [0]
    from collections import deque
    B = max(2, (n + 2) // 2)
    Q = deque()
    j = 1

def Gray_code_work_ahead_binary(n):
    while True:
        VISIT()
        STEP_binary()
        STEP_binary()
        j = Q.popleft()
        if j == n + 1: break
        a[j] = 1 - a[j]

# run the program for a test:
n = 4
initialize_work_ahead_binary(n)
Gray_code_work_ahead_binary(n)
```
A.8 Algorithm WORK-AHEAD with the modification of Section 8.1

def STEP_x():
    global j, b, m, B, Q, n
    if j==n+1: raise StopIteration
    if b[j]==m[j]-1:
        b[j]=0
        j += 1
    else:
        if len(Q)<B:
            b[j] += 1
            Q.append(j)
            j = 1

def initialize_work_ahead_x():
    global n, a, b, direction, B, Q, j
    a = (n+1)*[0]  # a[0], b[0], and direction[0] is wasted
    direction = (n+1)*[+1]
    b = (n+1)*[0]
    from collections import deque
    B = (n+1)//2
    Q = deque()
    j = 1

def Gray_code_work_ahead_x():
    try:
        while True:
            VISIT()
            STEP_x()
            STEP_x()
            j = Q.popleft()
            a[j] += direction[j]
            if a[j] in (0, m[j]-1): direction[j] *= -1
    except StopIteration:
        return

# run the program for a test:
n=4
m=[0]+[2,4,5,2]  # initial entry 0 is redundant
initialize_work_ahead_x()
Gray_code_work_ahead_x()
A.9 General mixed-radix Gray code generation according to the recursive definition of Section 3

In order to have a reference implementation for comparing the results, we give a program straight from the definition 2 of Section 3 extended to arbitrary mixed radices \((m_1, \ldots, m_n)\). When this program is run with the data specified below, it should produce the same Gray codes as all the previous example programs combined (apart from the additional state of the pegs that is reported by these programs). The outputs coincide precisely after stripping everything after the first blank on each line. The source files of this arXiv preprint include scripts that will extract the program code from the \LaTeX file of this appendix (extract-code-from-appendix.py) and compare the outputs to check the results (check-output.sh) after running the examples.

```python
def mixed_Gray_code(ms):
    "a generator for the mixed-radix Gray code"
    if ms:
        m = ms[0]
        G1 = mixed_Gray_code(ms[1:])
        while True:
            g = next(G1)
            for lastdigit in range(0, m):
                yield g+(lastdigit,)
            g = next(G1)
            for lastdigit in reversed(range(0, m)):
                yield g+(lastdigit,)
    else:
        yield ()

for result in (mixed_Gray_code([3]*6), mixed_Gray_code([5]*5), mixed_Gray_code([4]*6), mixed_Gray_code([4]*6), mixed_Gray_code([2,4,5,2]), mixed_Gray_code([2]*4), mixed_Gray_code([2,4,5,2]),):
    print ("##########")
    for g in result:
        print ("".join(str(x) for x in g))
```