Loopless Gray Code Enumeration and the Tower of Bucharest

Felix Herter$^1$ and Günter Rote$^2$

1 Institut für Informatik, Freie Universität Berlin, Takustr. 9, 14195 Berlin, Germany
avealx@zedat.fu-berlin.de

2 Institut für Informatik, Freie Universität Berlin, Takustr. 9, 14195 Berlin, Germany
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 Tower 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

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.19

1 Introduction: Binary reflected Gray code and the Tower 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

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

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.

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 $a_n a_{n-1} \ldots a_1$ 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 [10, 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.

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 it.
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.

$$move\_tower(k, A, B): \text{ (Move the } k\text{ smallest disks } D_1 \ldots D_k \text{ from peg } A \text{ to peg } B)$$

\begin{Verbatim}
if $k \leq 0$: return
auxiliary := 3 - A - B; \text{ (auxiliary is the third peg, different from } A \text{ and } B.)
move\_tower($k - 1$, $A$, auxiliary)
move disk $D_k$ from $A$ to $B$
move\_tower($k - 1$, auxiliary, $B$)
\end{Verbatim}

### 1.4 Connections between the Tower 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 $move\_tower$ above. It has thus been noted that the Gray code $G_n$ can be used to solve the Tower of Hanoi puzzle, cf. 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 Tower 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 $move\_tower$, 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.

\begin{algorithm}[H]
\caption{HANOI. Loopless algorithm for the binary Gray code.}
\begin{algorithmic}
\State Initialize: Put all disks on $P_0$.
\State loop: Move $D_1$ clockwise.
\State Let $D_k$ be the smaller of the topmost disks on the two pegs that don’t carry $D_1$.
\State If there is no such disk, terminate.
\State Move $D_k$ clockwise if $k$ is odd; otherwise, move it counterclockwise.
\end{algorithmic}
\end{algorithm}

To obtain the Gray code, we simply set $a_k := 1 - a_k$ whenever we move the disk $D_k$ [2]. 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 [2]. We have chosen the above formulation since it is better suited for generalization.
19:4  Loopless Gray Code Enumeration and the Tower of Bucharest

1.6 Overview

In the remainder of this paper, we will generalize these connections to Gray codes for larger radixes (alphabet sizes). Section 2 is devoted to ternary Gray codes and their connections to the so-called Towers of Bucharest. After defining Gray codes with general radixes in Section 3, we extend the ternary algorithm from Section 2 to arbitrary odd radixes \( m \) in Section 4, and even to mixed (odd) radixes (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 our main algorithms (Sections 4, 5, and 8) in Python. The preprint [8] contains simulations of all our algorithms.

2 Ternary Gray codes and the Towers of Bucharest

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

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.\(^1\) This version of the game was already considered in 1944 (not under this name) by Scorer, Grundy, and Smith [13, Section 4(iii)] and has been thoroughly investigated, see Chapter 8 in the extensive monograph about the Tower of Hanoi by Hinz, Klavžar, Milutinović, and Petr [9].

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:

\[\begin{align*}
&\text{Proposition 2.} \\
1. &\text{In the Towers of Hanoi, there are three possible moves from any position, except when all disks are on one peg: In these cases, there are only two possible moves.} \\
2. &\text{In the Towers of Bucharest, there are two possible moves from any position, except when all disks are on peg } P_0 \text{ or } P_2: \text{In those cases, there is only one possible move.}
\end{align*}\]

\(^1\) 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” [12], 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
Figure 3 The state graphs of (a) the Tower of Hanoi and (b) the Tower of Bucharest with \( n = 5 \) disks.

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 3a 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 the disks, since they always had to be carried all the way from the South wing to the North wing and back. As shown in this article, the consequences of the new rule in delaying the game are even more spectacular. These measures were definitely overcautious, in particular since nobody was there to move the discs any more, and moreover, the pegs with the golden discs were carefully guarded. Nevertheless, he was worried that his wife and children would wander around in the palace and play with the disks, thereby setting the prophecy into motion again, like in that movie, “Jumanji” with Robin Williams. He was not sure how the guarding officers would behave in a conflict between the loyalty to their orders and the authority of his family members. You may draw your own conclusions, but in my opinion, this count, or king if you wish, was pretty paranoid. In the end, it served him nothing. He was swept away by the revolution.

What became of the golden disks? Nobody knows. It is sometimes claimed that they were hidden in a subterranean cave, and hobby archaeologists are still looking for them occasionally. But probably they have found their way to the black market. Today, tourists that visit the palace are led to a stump on the floor in the North wing, which is supposedly the remainder of one of the pegs. The South wing is closed for restoration.

Another story, even more unbelievable but no less bloody than the first one, puts the Towers of Bucharest in the context of the legendary caliph Harun-al-Rashid. One night, the caliph was again wandering through the streets of Baghdad, as usual dressed like an ordinary businessman, in order to assure himself that the people were still loving his reign, admiring his wisdom, and praising his justice. He noticed a crowd of lookers-on who were gathered around a man and a woman sitting on the ground side by side, silently and solemnly executing the moves of the Towers of Bucharest. One of them would pick up a disk and set it on an adjacent peg. By the rules of the game, there was always one of them for whom the two involved pegs were easy to reach, and this was the one who carried out the move. Only on the infrequent occasions when one of the larger and heavier disks had to be moved, they helped each other. The man wore a
Loopless Gray Code Enumeration and the Tower of Bucharest

its position on the drawing is not straightforward.) By contrast, we see that the graph of the Towers of Bucharest in Figure 3b 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 $\pm 1$. This connection has already been noted earlier; it is explicitly mentioned in Graham, Knuth, and Patashnik [5, Exercises 1.2–1.3, p. 17, with answers on p. 483], or Guan [6, Theorem 4]. 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.

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.

May that as it be, something unexpected happened. As the khaliph 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 and handed him a black gown. “Hurry up, you are late. We were waiting only for you!” The gown covered his whole stature and hid his face, and he entered a room that was barely lit by an open fire. Seven other men in black gowns were already sitting on small stools in a circle around the fire. One stool was free, and he sat down. Beside the fire, there was a small ivory model of the Towers of Bucharest, with the four largest of the $n = 6$ disks on the final peg. Disc 1 was on the middle peg, and disk 2 was on the first peg. The khaliph, having watched the game just before, understood immediately what that meant. Nobody said a word. The tension rose. After six minutes, a lady entered and addressed them. She was the singer from the fountain. “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 disk will bring me the head of the detestable caliph Harun-al-Rashid (ca. 763–890). 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 disk. The khaliph 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
3 Gray codes with general radixes

An $m$-ary Gray code enumerates the $n$-tuples $(a_n, \ldots, a_1)$ 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 $C_1, C_2, \ldots, C_m^n$ be the Gray code for the strings of length $n$. Then the strings of length $n + 1$ are generated in the order $C_1 0, C_1 1, C_1 2, \ldots, C_1 (m - 2), C_1 (m - 1)$, $C_2 0, C_2 1, C_2 2, \ldots, C_2 (m - 2), C_2 (m - 1)$, $C_3 0, C_3 1, C_3 2, \ldots, C_3 (m - 2), C_3 (m - 1)$, $C_4 0, C_4 1, C_4 2, \ldots, C_4 (m - 2), C_4 (m - 1)$, $C_5 0, C_5 1, \ldots$ (2)

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

The Python program in Appendix A.4 implements this recursive definition directly.

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.

Figure 4 shows an example with $m = 5$. This game with 5 pegs is called the Tower of Klagenfurt, after the birthplace of the senior author.²

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 violate the rules by putting 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

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.
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. By induction, 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 is never moved on top of \( D_1 \) either.

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 alternately cover \( P_0 \) and \( P_{m-1} \) and exclude the covered peg for 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. The program in [8, Appendix A.3] applies this idea 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 \textbf{EVEN}. Generation of the \( m \)-ary Gray code for even \( m \).
Initialize: Put all disks on \( P_0 \).
loop:
\text{Move } D_1 \text{ for } m - 1 \text{ steps, in clockwise direction.}
\text{Let } D_k \text{ be the smallest of the topmost disks on the } m \text{ pegs that don’t carry } D_1.
\text{If there is no such disk, terminate.}
\text{Move } D_k \text{ by one step, in the direction determined by the parity of } k.
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 Gray code is determined by changing the digit $a_i$ whenever disk $D_i$ is moved. The digit $a_i$ runs through the sequence $0, 1, 2, \ldots, m - 2, m - 1, m - 2, \ldots, 2, 1, 0, 1, 2, \ldots$. 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. ◀

Except for $m = 3$ and $m = 2$, Algorithms ODD and EVEN do not generate a shortest sequence of moves to the target configuration, even if moves are allowed only between adjacent pegs (or cyclically adjacent pegs, in a direction depending on the disk parity). We could not come up with some 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
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)

We can 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 $P_1$. When
we say we turn a disk, this means that we turn it clockwise to the next dial position, and if
necessary, move it to the appropriate peg.

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 radixes $m_i \geq 2$, 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 $m$.

3 Nowadays, most households will more readily have access to a computer than to a tower of Hanoi.
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:

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

loop:
    $j := 1$
    while $b_j = m_j - 1$:
        $b_j := 0$
        $j := j + 1$
    if $j = n + 1$: TERMINATE
    $b_1 := b_1 + 1$
    $Q.append(j)$

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.

procedure STEP:
if $b_j = m_j - 1$:
    $b_j := 0$
    $j := j + 1$
else:
    if $Q$ is not filled to capacity:
        $b_j := b_j + 1$
        $Q.append(j)$
        $j := 1$

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 Gray code algorithm on the right couples two production STEPs with one consumption step, which takes out an entry $j$ of $Q$ and carries out the update $a_j := a_j \pm 1$. 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 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.

For 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 STEPs 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 [7] 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 [14, 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:

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

With this modified procedure \textbf{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 \textbf{STEP}' signals termination. Let us prove this:

The termination signal is sent instead of producing the value \(j = \bar{\rho}(k) = n + 1\) for \(k = m_0 m_1 \ldots m_n\). Generating this signal takes \(n + 1\) iterations of \textbf{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 \textbf{STEP} were carried out and contributed to the production of the termination signal. It follows that when \textbf{STEP} discovers that \(j = n + 1\), no unused entries are in the stack, and it is safe to terminate the program.

It is important not to “speed up” the program by moving the termination test into the \textbf{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:

\[
\rho(k) := \max\{i : 0 \leq i \leq n, \ m_1 m_2 \ldots m_i \text{ divides } k\}, \quad \bar{\rho}(k) := \rho(k) + 1
\]

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 \(\bar{\rho}(k)\) \textbf{STEP}s.

\textbf{Lemma 7.} In Algorithm WORK-AHEAD, the buffer \(Q\) never becomes empty.

\textbf{Proof.} We number the iterations of the main loop as 1, 2, \ldots, \(m_1 m_2 \ldots m_n\). In the last iteration, the algorithm terminates.

Let us show that the queue \(Q\) is not empty in iteration \(k\). We distinguish two cases.

(i) Up to and including iteration \(k\), two repetitions of \textbf{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

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

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

(4)

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 \( j = \rho(k_0) \), and it contained the values \( \rho(k_0), \rho(k_0 + 1), \ldots, \rho(k_0 + B - 1) \). Since then, STEP was called \( 2(k - k_0) \) times, and \( \rho(k) \) is ready when it is needed, provided that

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

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 \( \rho(k) \). Computing \( \rho(k_0 + B) \) 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

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

(5)

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

\[
S(k) = k + \left\lfloor \frac{k}{m_1} \right\rfloor + \left\lfloor \frac{k}{m_1m_2} \right\rfloor + \cdots + \left\lfloor \frac{k}{m_1m_2 \cdots m_n} \right\rfloor.
\]

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 \( |x| - |x'| < x - x' \) and 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 \( B := \left\lceil \frac{n}{2} \right\rceil \) satisfies \( 2B \geq n \).

In Algorithm WORK-AHEAD, the STEP\( \text{s} \) should generate entries \( \rho(1), \rho(2), \ldots \) of \( Q \) up to \( \rho(N) \), where \( N := m_1m_2 \cdots m_n \).

The following lemma shows that production of the STEP\( \text{\text{s}} \) may overrun their target by at most one. Since the algorithm has already made provisions to generate \( \rho(N) = n + 1 \) 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 \( \rho(N) \) or \( \rho(N + 1) \).

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

If the production of \( \rho(N) = n + 1 \) is completed in the second STEP of an iteration, it is thus immediately consumed, which leads to termination. If \( \rho(N) \) is completed in the first STEP of an iteration, the second STEP will produce the value \( \rho(N + 1) = 1 \), 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 \( j = \overline{\rho}(k) \) 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 \( \overline{\rho}(k_1) \) 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 \( \overline{\rho}(2^n) \) is completed in the first STEP of iteration \( 2^n \). In the second STEP of this iteration, \( \overline{\rho}(2^n + 1) = 1 \) is added to \( Q \). Thus, when \( \overline{\rho}(2^n) \) 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 radixes \( m_i \) are odd. It remains to find a model that would work for different even radixes or even for radixes of mixed parity.

Another motion-planning game which is related to the binary Gray code is the Chinese Rings puzzle, see Gardner [4], Knuth [10, pp. 285–286], or Scorer, Grundy, and Smith [13]. (Knuth [10, Solution to Ex. 7.2.1.1–(10), p. 679] gives a brief survey of the early literature, mentioning references that date back as far as the 16th century.) The goal is to detach a series of interlocked rings from a bar. Like the Towers of Bucharest, the Chinese rings allow at most two possible moves in every state. Each move removes or replaces a single ring. By simulating the Chinese rings directly, one can therefore obtain another loopless algorithm for the binary Gray code, see Misra [11], Knuth [10, Solution to Ex. 7.2.1.1–(12b) ]. However, this algorithm does not seem to extend to other radixes. (Scorer et al. [13, Section 5] analyzed a generalization of the Chinese Rings. We did not check whether it leads to interesting Gray codes.)

Acknowledgements. We thank Don Knuth for leading us to the reference [7] about the work-ahead approach, and we thank Sandi Klavžar for pointing us to the earlier references [5] and [6] that connect the Towers of Bucharest to ternary Gray codes.

References

19:16  Loopless Gray Code Enumeration and the Tower of Bucharest

5  Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

A  Appendix: PYTHON simulations of some algorithms

These programs run equally with Python 2.7 and Python 3. The pegs, the string a, and the array of directions are kept as global variables. See [8] for the complete set of programs.

A.1  Algorithm ODD, Section 4

def initialize_m_ary_odd(m,n):
    global pegs, a, direction
    pegs = tuple([0] 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):
    "generate n-tuple of entries from the set {0,1,...,m-1}"
    visit()
    while True:
        for _ in range(m-1): # repeat m-1 times:
            move_disk(m, peg=a[1])
            visit()
F. Herter and G. Rote

```
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]
pegs[peg].append(disk)
if peg==m-1: direction[disk] = -1
elif peg==0: direction[disk] = +1

def visit(): # print the current string and the contents of the pegs
    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

initialize_m_ary_odd(m=3,n=6) # run the program for a test
Gray_code_m_ary_odd(m=3)

A.2 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

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

A.3 Algorithm WORK-AHEAD, Section 8

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:
A.4 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 $(m_1, \ldots, m_n)$.

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

for g in mixed_Gray_code([2,4,5,2]): # run the program for a test
    print ("".join(str(x) for x in g))
```