Loopless Gray Code Enumeration and the Tower of Bucharest

Felix Herter[a], Günter Rote[a]

[a]*Institut für Informatik, Freie Universität Berlin Takustr. 9, 14195 Berlin, Germany*

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.

Keywords: Tower of Hanoi, Gray code, enumeration, loopless generation

---

Contents

Email addresses: `avealx@zedat.fu-berlin.de` (Felix Herter), `rote@inf.fu-berlin.de` (Günter Rote)

1. Introduction: The 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 1a for an example. The Gray code $G_1 = 0, 1$, and if $G_n = C_1, C_2, \dots, C_{2^n}$ is the Gray code for the bit strings of length n, then

$$G_{n+1} = 0C_1, 0C_2, \dots, 0C_{2^n},\ 1C_{2^n}, 1C_{2^n-1}, \dots, 1C_2, 1C_1. \tag{1}$$

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.

(a)

000000	001011	010111	110100	101110
000001	001001	010110	111100	101111
000011	001000	010010	111101	101101
000010	011000	010011	111111	101100
000110	011001	010001	111110	100100
000111	011011	010000	111010	100101
000101	011010	110000	111011	100111
000100	011110	110001	111001	100110
001100	011111	110011	111000	100010
001101	011101	110010	101000	100011
001111	011100	110110	101001	100001
001110	010100	110111	101011	100000
001010	010101	110101	101010	

(b)

0000	0111	0222	1112	1001	2120	2220
0001	0112	1222	1111	1000	2110	2221
0002	0102	1221	1110	2000	2111	2222
0012	0101	1220	1120	2001	2112	
0011	0100	1210	1121	2002	2102	
0010	0200	1211	1122	2012	2101	
0020	0201	1212	1022	2011	2100	
0021	0202	1202	1021	2010	2200	
0022	0212	1201	1020	2020	2201	
0122	0211	1200	1010	2021	2202	
0121	0210	1100	1011	2022	2212	
0120	0220	1101	1012	2122	2211	
0110	0221	1102	1002	2121	2210	

Figure 1: (a) The binary Gray code G_6 for 6-tuples. (b) The ternary Gray code for 4-tuples, as considered in Section 4 and defined in Section 5.

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 $a_n a_{n-1} \dots a_1$ 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, which is optimal.

A stricter requirement is that the *worst-case* time between two successive strings is constant. Such an algorithm is called a *loopless* generation algorithm. We will discuss this concept more thoroughly in Section 2. Different loopless algorithms for Gray codes are known, see Bitner, Ehrlich, and Reingold [1] and Knuth [2, Algorithms 7.2.1.1.L and 7.2.1.1.H]. These algorithms achieve constant time by maintaining additional pointers in a smart way.

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 $D_1, D_2, \dots, D_n$ 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.

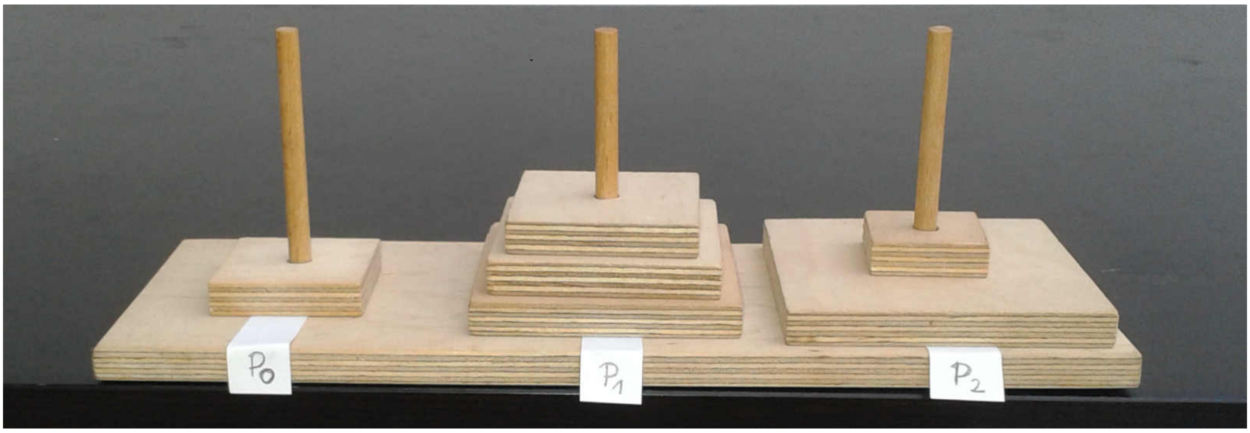


Figure 2: The Tower of Hanoi with $n = 6$ (square) disks. When running the algorithm HANOI from Section 1.5, the configuration in this picture occurs together with the bit string 110011. (The relation between the positions of the disks and this bit string is not straightforward, cf. [3, Section 3].) The next disk to move is D_1; it moves clockwise from peg P_2 to 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 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.

procedure MOVE-TOWER(k, A, B). Moves the k smallest disks $D_1 \ldots D_k$ from peg A to peg B
  if $k \leq 0$: **return**
  auxiliary := $3 - A - B$; Comment: *auxiliary* is the third peg, different from A and B.
  MOVE-TOWER$(k - 1, A, \mathit{auxiliary})$
  move disk D_k from A to B
  MOVE-TOWER$(k - 1, \mathit{auxiliary}, B)$

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 of numbering the bits starting from 0, we start at 1.) This sequence has an obvious recursive structure which results from (1). It also describes the number of changed bits when incrementing a number from j to $j + 1$ 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 well-known Tower of Hanoi puzzle, cf. Scorer, Grundy, and Smith [3, Section 5] or Gardner [4]. The delta sequence does not specify the direction of movement, but this can be easily recovered, see Proposition 1 below. Conversely, the Tower of Hanoi puzzle can be used to generate the Gray code G_n, see Buneman and Levy [5].

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.

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: $P_0 \to P_1 \to P_2 \to P_0$, and the even-numbered disks always proceed in the opposite circular direction: $P_0 \to P_2 \to P_1 \to P_0$.* □

In the other case, when the assumption does not hold, the directions are simply swapped. Since our goal is not to move the tower to a specific target peg, but to generate the Gray code, we stick with the proposition as stated.

Algorithm HANOI. Loopless algorithm the Tower of Hanoi and for the binary Gray code.
Initialize: Put all disks on P_0.
loop:
Move D_1 clockwise.
Let D_k be the smaller of the topmost disks on the two pegs that don't carry D_1.
If there is no such disk, TERMINATE.
Move D_k clockwise if k is odd; otherwise, move it counterclockwise.

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 [5]. We have chosen the above formulation since it is better suited for generalization (Section 7).

1.6. Overview

In this paper, we will generalize the connections between Gray codes and the Tower of Hanoi to Gray codes for larger radixes (alphabet sizes). Section 4 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 5, we extend the ternary algorithm from Section 4 to arbitrary odd radixes m in Section 6, and even to mixed (odd) radixes (Section 8). In Section 7, we generalize the binary Gray code algorithm HANOI from above to arbitrary even m. Finally, in Section 10, 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. The introductory Section 2 discusses the concept of loopless algorithms in greater depth, and should dispel any hopes that the reader might have of finding something that would be of great practical value. Section 3 mentions fast computer hardware operations as an alternative option for generating Gray codes and sets our topic apart from such practices. In the brief remainder of the introduction, Section 1.7, we prepare the readers' minds for the primary "model of computation" that we will use. In the concluding section, Section 11, we will reflect our results and how they were achieved, and we will indicate some open problems.

These results were presented at the 8th International Conference on Fun with Algorithms (FUN 2016) in La Maddalena island off Sardinia in June 2016 [6]. The preprint [7] contains prototype simulations of all our algorithms in the programming language Python.

1.7. Algorithms without computers

The algorithm HANOI does not run on a conventional computer but on a different piece of hardware (Fig. 2). We will show more such examples. Of course, it is easy to translate these algorithms into "simulations" on the electronic computers to which we are so accustomed. However, we encourage the readers to join us in thinking directly about algorithms for this restricted world, namely, looking at stacks of disks on different pegs.

This relates to the *CS-Unplugged*[1] project (Computer Science without a computer) in the context of educating children about Computer Science, and it underlines the point that Computer Science, or *Informatics*, as it is more appropriately called in other languages, is not the science of computers. "Computer Science is no more about computers than astronomy is about telescopes" is a saying which often attributed to E. W. Dijkstra, but which apparently goes back to Mike Fellows. In the case of astronomy, it must of course be conceded that telescopes, and more generally, devices and procedures for physical measurements, are eminently relevant. Similarly, there is an important part of Computer Science that deals with the design, the organization, and the use of computers. However, a core part of Computer Science, in particular in theoretical computer science and the analysis of algorithms, is concerned with ideas that are separate from the physical embodiment in electronic computers. One can even argue that a major effort of Computer Science (programming languages, operating systems) consists in providing layers of abstraction that help to avoid direct contact with computers.

[1] `csunplugged.org`

2. Loopless generation algorithms

The efficiency of enumeration algorithms can be judged by different criteria. Besides the overall runtime for generating all solutions of a combinatorial problem, we may be interested in a finer analysis of the runtime. The performance measures for combinatorial enumeration algorithms include [8]

a) the *delay* between successive solutions,
b) the *setup time* for generating the first solution,
c) the *finishing time* for determining that the last solution has been generated and no further solution exists,
d) and the *memory* requirement.

The best conceivable algorithms have $O(1)$ delay, $O(n)$ setup time, and $O(1)$ finishing time, where n is the size of the generated solutions. For such algorithms, Ehrlich [9] coined the term *loopless* in 1973, and he pioneered loopless enumeration algorithms for various combinatorial structures. All algorithms that we consider have the additional property that they use only $O(n)$ memory.

It is not necessary that a loopless algorithm should contain no loops in the program besides an outer loop that iterates over the solutions. Since it is guaranteed that the number of operations between successive visits is bounded in advance, any inner loops can be eliminated by unrolling them sufficiently often, hence making the algorithm loopless in the literal sense of the word.

In order to go from one solution to the next in constant time, the difference between successive solutions must necessarily be small. Therefore, loopless algorithms go hand in hand with Gray codes, where the difference between successive elements is just a single entry.

The primary purpose for enumerating combinatorial objects is usually not to print or store a complete list, but to investigate the objects one by one, to "visit" them by some procedure, which depends on the application. In our case, the bit strings might represent all subsets of an n-element set, and we want to evaluate some objective function on each set in order to find the best one. If the objective function can be easily updated when a single element is inserted or removed, a Gray code is the sequence of choice.

Since the number of enumerated solutions is usually huge, the dominating algorithmic factor for the total running time is the delay. However, a small *worst-case* delay, as required for a loopless algorithm, is unnecessary for such an application. A bound on the *average delay* is good enough. There are of course areas where a worst-case bound for individual steps is essential, for example when processing queries in interactive systems, in parallel computing, or in real-time applications. These are areas where predictability is more important than overall speed. However, to put our results into the proper perspective, we emphasize that we do not envision such scenarios for our algorithms. Moreover, the effort for generating all bit strings is often negligible compared to the time that it takes to process each bit string, and hence the speed of generation is of minor importance.

We conclude this discussion with a quote from Don Knuth, from the documentation of a loopless generation program SPIDERS that he wrote[2], which summarizes the point nicely.

> The extra contortions that we need to go through in order to achieve looplessness are usually ill-advised, because they actually cause the total execution time to be longer than it would be with a more straightforward algorithm. But hey, looplessness carries an academic cachet. So we might as well treat this task as a challenging exercise that might help us to sharpen our algorithmic wits.

We will come back to these remarks in the concluding section 11.

3. Bitwise operations as a fast alternative

The arithmetic and logical operations on full-word operands that are supported on conventional computers provide a fast alternative for computing the Gray code. For example, the j-th element of the Gray code can be computed directly with the help of the bitwise exclusive-or operation as "j XOR $\lfloor j/2 \rfloor$" for $j = 0, 1, \ldots, 2^n - 1$, cf. [2, Eq. 7.2.1.1–(9), p. 284]. In the notation of C or PYTHON, this can be written with the shift operator >> as `j^(j>>1)`. This technique can

[2] `www-cs-faculty.stanford.edu/~knuth/programs.html` (2001). He makes similar remark in [2, Answer to Ex. 7.2.1.2-19, p. 706].

even be extended to other radixes (Section 5), although we are not aware that this has been described anywhere. We will not further consider such algorithms here.

Loopless algorithms gain an advantage when one wants to identify the bit k that is changed. For example, when the Gray code models all subsets of an n-element set, the k-th element is inserted or removed, and one has to compute the effect of this operation on the set. In our combinatorial algorithms, the index k is directly available. When the Gray code is computed through bitwise operations, the XOR of two successive bit strings gives the binary representation of 2^k. From this, the index k can be recovered [2, p. 141–2]. However, the number of operations grows logarithmically with the word size unless the hardware provides special instructions such as counting the number of 1-bits in a word (sideways addition).

4. Ternary Gray codes and the Towers of Bucharest

A ternary Gray code enumerates the 3^n n-tuples $(a_n, \dots, a_1)$ 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.[3] This version of the game was already considered in 1944 (not under this name) by Scorer, Grundy, and Smith [3, 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 [11].

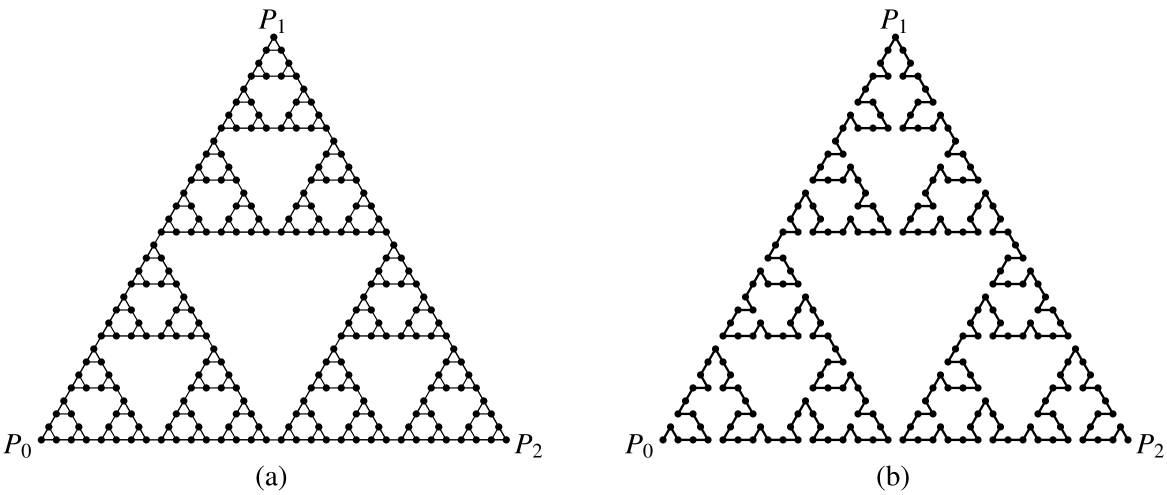


Figure 3: The state graphs of (a) the Tower of Hanoi and (b) the Tower of Bucharest with $n = 5$ disks

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 all disks are 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 all disks are on peg P_0 or P_2: In those cases, there is only one possible move.*

[3]It is an established custom to name variations of the Tower of Hanoi game after different cities, instead of using ordinary names such as "three-in-a-row" [10]. The name "Towers of Bucharest" has been suggested by Günter M. Ziegler. Several legends rank themselves around the towers of Bucharest, see [6, 7].

The original name of the "Tower of Hanoi" game has the word tower in singular. The plural "towers of Hanoi" have become popular in the Computer Science literature [11, p. 46], probably because it is tempting to associate the three disk-carrying pegs with towers. The original name continues to be prevalent in the mathematics literature. We honor both traditions by not sticking to a fixed usage.

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 the target point, P_1 or 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 complicated.) 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 [3, 11]. 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 already been noted earlier; it is explicitly mentioned in Graham, Knuth, and Patashnik [12, Exercises 1.2–1.3, p. 17, with answers on p. 483], or Guan [13, Theorem 4]. In fact, the algorithm produces *the* ternary reflected Gray code, which we are about to define below in Section 5; 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. When the algorithm arrives at this configuration, 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, and so on.

5. Gray codes with general radixes and with mixed radixes

An m-ary Gray code enumerates the n-tuples $(a_n, \dots, a_1)$ with $0 \le a_i < m$, changing a single digit at a time by ± 1. The reflected Gray code can be recursively described as follows: Let $C_1, C_2, \dots, C_{m^n}$ be the Gray code for the strings of length n. Then the strings of length $n + 1$ are generated in the order

$$\begin{array}{ll} C_1 0, C_1 1, C_1 2, \dots, C_1(m-2), C_1(m-1), & C_2(m-1), C_2(m-2), \dots, C_2 2, C_2 1, C_2 0, \\ C_3 0, C_3 1, C_3 2, \dots, C_3(m-2), C_3(m-1), & C_4(m-1), C_4(m-2), \dots, C_4 2, C_4 1, C_4 0, \\ C_5 0, C_5 1, C_5 2, \dots, C_5(m-2), C_5(m-1), & \dots \end{array} \tag{2}$$

This recursive definition differs from our first definition (1) for the special case of the binary Gray code, where we have added the new digit at the front, but the two definitions are equivalent. The definition (2) with the appended digit is more suited for deriving the algorithms that are to follow. We see that each digit alternates between an upward sweep from 0 to $m - 1$ and a return sweep from $m - 1$ to 0.

The more general Gray code for *mixed radixes* $(m_n, \dots, m_1)$, where each digit has its own range $0 \le a_i < m_i$, is defined in an analogous way.

6. Generating the *m*-ary Gray code with odd *m*

For odd m, the ternary algorithm from Section 4 can be generalized. We need m pegs $P_0, \dots, P_{m-1}$. The leftmost peg P_0 and the rightmost peg P_{m-1} play a special role.

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.

In this algorithm and the algorithms that follow, it is understood that we visit a string of the Gray code at the start and after each move of a disk. (Writing this explicitly would clutter the description of the algorithms.) As for the towers of Bucharest, we can directly translate the position of a disk into a digit of the string. Figure 4 shows an example with $m = 5$. This game with 5 pegs is called the Towers of Klagenfurt, after the birthplace of the senior author.[4]

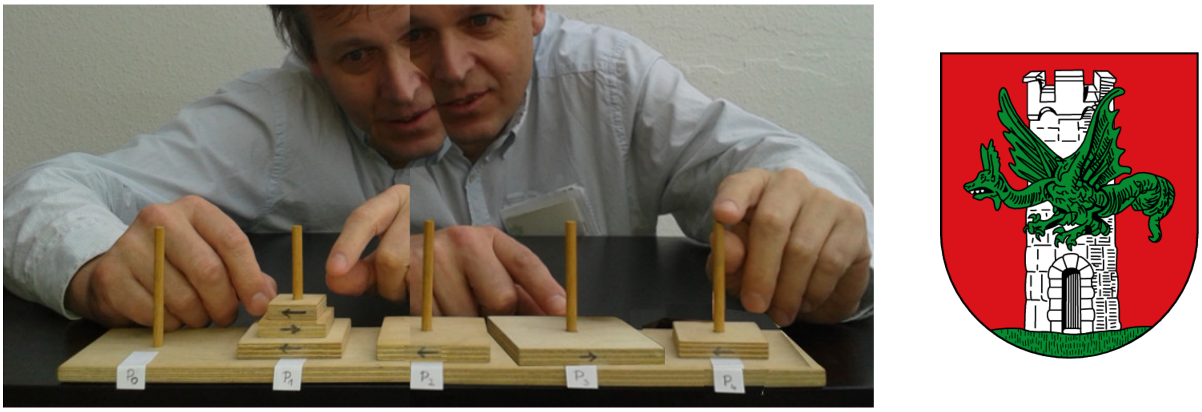


Figure 4: The Towers of Klagenfurt. This configuration represents the string 321411 over the radix $m = 5$. The arrows of the disks indicate the current direction of movement for Algorithm ODD. The next step moves the smallest disk D_1 onto peg P_0, changing the string to 321410. After that, disk D_2 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 explicitly specified, whereas the movement of the other disks, whenever D_1 is at rest, is "figured out" by the algorithm. It is not immediately 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)*, and all moves that it performs are valid.*

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 $D_2, \dots, D_n$, 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

[4]When the city of 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 some 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.

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 explicitly 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 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. We will apply this idea to the algorithm of Section 8 below, and this will result in a very simple algorithm.

7. 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_k whenever disk D_k is moved. The digit a_k runs through the cyclic sequence $0, 1, 2, \ldots, m-2, m-1, m-2, \ldots, 2, 1, 0, 1, 2, \ldots$. Thus it changes always by $d_k = \pm 1$, 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.

More precisely, we initialize all digits a_i to 0 and all directions d_i to $+1$ at the beginning. Every movement of a disk D_k in the above program is replaced by the following procedure:

procedure MOVE(k).
  if k is even:
    Move D_k one step in counterclockwise direction
  else:
    Move D_k one step in clockwise direction
  $a_k := a_k + d_k$
  if $a_k = 0$ **or** $a_k = m_k - 1$: $d_k := -d_k$
  visit the n-tuple $(a_n, \ldots, a_1)$

As in the binary case, it is far from straightforward to relate the disk configuration to the Gray code. For example, the configuration in Figure 4, interpreted in the context of algorithm EVEN for $m = 4$, appears when the string is 211030. The arrows shown on the disks play no role for this algorithm. Disk D_1 has just made three clockwise steps and is going to rest for one step. The next step moves D_3 clockwise (since 3 is odd) from P_4 to P_0, and the string is changed to 211130. After that, D_1 resumes its clockwise motion, and the string changes to 211131.

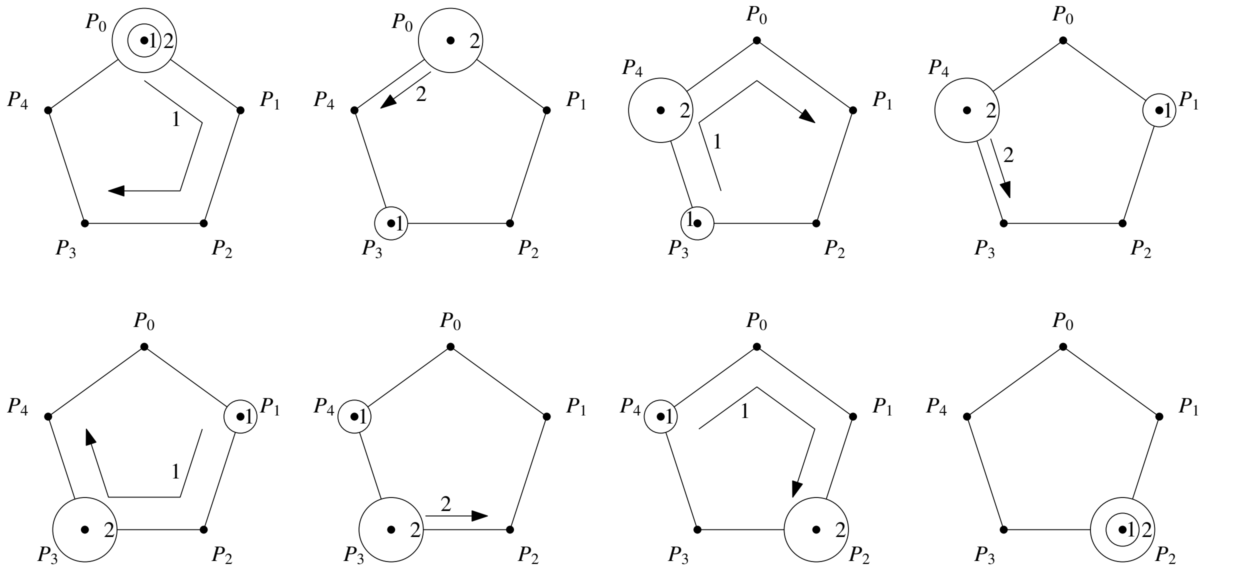


Figure 5: One period of movement of the two smallest disks D_1 and D_2 when Algorithm EVEN generates all tuples over an alphabet of size $m = 4$ using $m + 1 = 5$ pegs.

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. $\square$

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). For example, for $m = 4$ and $n = 2$, Fig. 5 shows the complete program of 15 moves that generate the $4^2 = 16$ codewords. However, it is easy to get from the first position to the last position in a total of 5 moves: 2 clockwise moves of D_1 interspersed with 3 counterclockwise moves of D_2. In fact, one can get from any position to any other position in at most 12 moves that respect the directions.

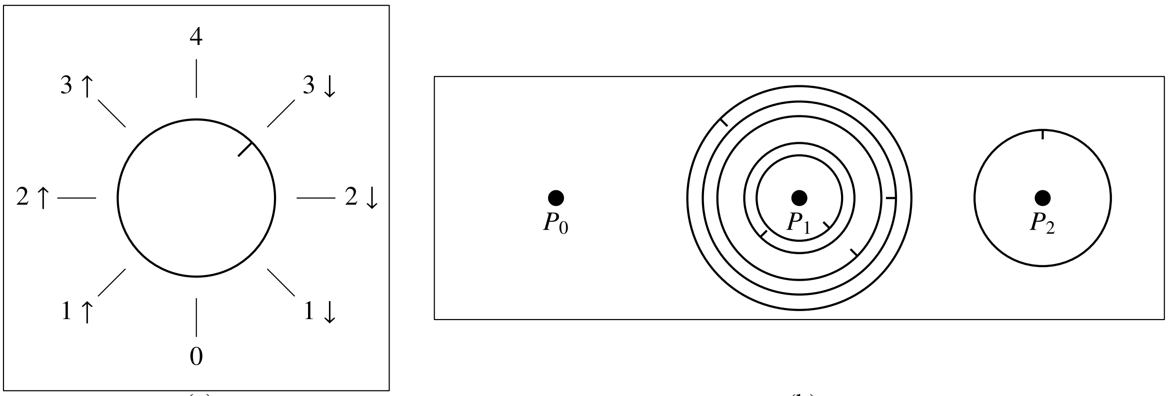
We could not come up with some natural constraints under which our algorithms give a shortest solution. (Of course, algorithm ODD always generates a *longest* sequence of moves without repetitions.)

8. 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 P_0 and P_{m-1} 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:

$$0,\ 1\!\uparrow,\ 2\!\uparrow, \ldots,\ (m-2)\!\uparrow,\ m-1,\ (m-2)\!\downarrow, \ldots 2\!\downarrow,\ 1\!\downarrow,\ 0,\ 1\!\uparrow,\ \ldots \tag{3}$$

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.



(a) (b)

Figure 6: (a) The upgraded disk of the Towers of Bucharest++ for $m = 5$, and the meaning of its positions. (b) The situation of Figure 4, compressed to 3 pegs. The smallest disk D_1 is about to turn and move from P_1 to P_0. After that, we will turn D_2 on P_1 without moving it.

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:

Proposition 5. *Algorithm ODD-COMPRESSED performs the same steps as Algorithm ODD, except that the contents of the intermediate pegs $P_1, \dots, P_{m-2}$ of Algorithm ODD are merged into the middle peg P_1 in Algorithm ODD-COMPRESSED.*

Proof. We can prove this by induction on the number of steps. The statement holds in the beginning. The motions of the explicitly controlled disk D_1 are in direct correspondence between the two algorithms. Let us now look at the choice of the moving disk D_k. This choice happens when D_1 is on one of the extreme pegs. In both algorithms, the chosen disk is the smallest disk not covered by D_1, and thus the two algorithms chose the same disk D_k. The dials have been designed in such a way that turning a disk and moving it according to the dial position precisely models the motion of the corresponding disk in Algorithm ODD. Thus, Algorithm ODD-COMPRESSED, like Algorithm ODD, will not move the disk D_k on top of D_1. It will also not move D_k on top of a different smaller disk, since D_k is the smallest disk not covered by D_1. Thus, the disks are in the proper order on each peg, after the move. It follows that the disks on the middle peg must be the merged disks from the intermediate pegs of Algorithm ODD. □

When this algorithm is combined with the imaginary control disk D_0 that was mentioned at the end of Section 6, we arrive at the following simple main loop of the algorithm:

while pegs P_0 and P_1 are not both empty:
  turn the smallest disk on P_0 and P_1
  turn the smallest disk on P_1 and P_2

The algorithms tests for termination in the **while**-condition after every even number of steps. This is sufficient, because we know that the total number of strings is odd, and hence that the total number of transition steps is even.

This is perhaps the easiest-to-describe of all our algorithms, but of course, some of the complexity is hidden in the mechanics of the turning operation. A program that implements this algorithm on a computer is given in [7, Appendix A.3]. The algorithm can even be generalized to mixed-radix Gray codes for some radix sequence $(m_n, \dots, m_1)$, provided that all m_i are odd.

9. Computer simulation

All our algorithms can be easily simulated in software on an electronic computer.[5] 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. Thus, if the radix m is regarded as a constant, then, since $k = m$ in Algorithm ODD and $k = m + 1$ in Algorithm EVEN, 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 even for large k, at the expense of an increased overhead, we can use the following easy fact, which follows directly from the algorithm statement.

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

To get a loopless implementation, the set of disks on a peg is maintained as a sequence of maximal intervals of successive integers, instead of storing them as a stack in the usual way. For example, instead of the list $[1, 2, 3, 6, 8, 9, 12, 16, 17, 18, 19]$, we store the list of pairs $[(1, 3), (6, 6), (8, 9), (12, 12), (16, 19)]$. 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. In the example, it would be disk D_4.

10. 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 general technique for converting amortized bounds into worst-case bounds (de-amortization). We will discuss the ideas behind this transformation in Section 10.1. In contrast to the previous sections, this approach has no connections to the Towers of Hanoi or similar motion-planning games. These algorithms are definitely not recommended when looplessness is not important, since the overall running time will be higher due to the overhead of an additional buffer.

We will introduce this method for the most general task: mixed-radix Gray code generation. 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 $(b_n, \dots, b_1)$ with $0 \le b_i < m_i$ in* lexicographic *order. If, between two successive tuples of the sequence, the j rightmost digits are changed, then, in 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 $(b_n, \dots, b_1)$ in a straightforward way:

Algorithm DELTA. Generation of the delta sequence for the Gray code.

```
Initialize: (b_n, ..., b_2, b_1) := (0, ..., 0, 0)
Q := an empty list
loop:
    i := 1
    while b_i = m_i − 1:
        b_i := 0
        i := i + 1
        if i = n + 1: TERMINATE
    b_i := b_i + 1
    Q.append(i)
```

[5] Nowadays, most households will more readily have access to a computer than to a tower of Hanoi.

The delta sequence is stored in the list Q. It is an easy exercise, at least in the binary case, to show that the total number of changed digits when counting from 0 to j is less than $2j$, see the bound (4) in the proof of Lemma 2 below. Correspondingly, the *average* or *amortized* number of loop iterations ("steps") 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. We first make a small cosmetic modification and move the reset operation "$i := 1$" to the end of the loop. The changes are marked by arrows:

Algorithm DELTA′. Generation of the delta sequence for the Gray code.

```
      Initialize: (b_n, ..., b_2, b_1) := (0, ..., 0, 0)
      Q := an empty list
→     i := 1
      loop:
          while b_i = m_i − 1:
              b_i := 0
              i := i + 1
              if i = n + 1: TERMINATE
          b_i := b_i + 1
          Q.append(i)
→         i := 1
```

After this transformation, it is easier to extract one iteration of the **loop/while** loop into a procedure STEP, as shown in the following loopless algorithm:

procedure STEP.

```
    if b_i = m_i − 1:
        b_i := 0
        i := i + 1
    else:
        if Q is not filled to capacity:
            b_i := b_i + 1
            Q.append(i)
            i := 1
```

Algorithm WORK-AHEAD. Generation of the Gray code.

```
(a_n, ..., a_2, a_1) := (0, ..., 0, 0)
(d_n, ..., d_2, d_1) := (1, ..., 1, 1)
(b_{n+1}, b_n, ..., b_2, b_1) := (0, 0, ..., 0, 0); m_{n+1} := 2
Q := a queue of capacity B := ⌈n/2⌉, initially empty
i := 1
loop:
    visit the n-tuple (a_n, ..., a_2, a_1)
    STEP
    STEP
    remove k from Q
    if k = n + 1: TERMINATE
    a_k := a_k + d_k
    if a_k = 0 or a_k = m_k − 1: d_k := −d_k
```

To produce one value of the delta sequence, between one and two STEPs are needed on average. Thus, the Gray code algorithm WORK-AHEAD on the right couples two production STEPs with one consumption step, which takes out an entry k of Q and carries out the update $a_k := a_k \pm 1$. Every digit a_k must cycle up and down through its values in the sequence (3), and thus, we have to remember the direction $d_k = \pm 1$ in which it moves, just like in Algorithm ODD.

As an additional change, we have taken the termination test $i = n + 1$ out of the procedure STEP and moved it to the side of the consumer. This means that the value $i = n + 1$ will still be processed in procedure STEP, and accordingly, we had to extend the n-tuple b into an $(n + 1)$-tuple, setting m_{n+1} arbitrarily to 2.

The buffer Q has bounded size $B := \lceil \frac{n}{2} \rceil$. When Q would overflow, the procedure STEP does nothing, and repeated calls 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.

To show that the algorithm is correct, we have to ensure two things:

a) The queue Q is never empty when the algorithm retrieves an element from it. This is proved below in Lemma 2.

b) The clean way to terminate the algorithm would be to stop inserting elements into Q as soon as $i = n + 1$ is *produced* in STEP, as in Algorithm DELTA. Instead, termination is triggered when the value $k = n + 1$ is *removed* from Q. Due to this delayed termination test, it is possible that more iterations of STEP than needed are carried out. Lemma 3 will show that the number of these extra iterations is at most 1, and that they can therefore cause no harm.

For the *binary* Gray code ($m_i = 2$ for all $i = 1, \dots, n$), 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 4 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_k are of course also superfluous, in the binary case: The last two lines of the loop can be replaced by the statement $a_k := 1 - a_k$.

10.1. Working ahead, or delaying the output

The scheduling of operations is a recurring theme in the design of algorithms: Should I clean up immediately after making a mess, or should I wait until I look for something? One end of the spectrum are lazy data structures and, on a more fundamental level, lazy functional programming languages like Haskell: In contrast to the classical method of *strict evaluation*, which evaluates all arguments of a function before executing the body of the function, the evaluation of arguments is delayed until they are needed. Laziness allows to save unnecessary work in some cases. Laziness in data structures leads, in the case of the celebrated Fibonacci heaps, to the best known *amortized* performance for priority queue operations.

The other extreme is real-time (or looplessness), where special care is taken to spread the work evenly between the operations. The approach that we have taken in this section is to start with a straightforward algorithm with a good amortized runtime and *de-amortize* it: "Since amortized data structures are often simpler than worst-case data structures, it is sometimes easier to design an amortized data structure, and then convert it to a worst-case data structure, than to design a worst-case data structure from scratch" [14, Section 7, p. 84]. Kosaraju and Mihai [15] give a survey of de-amortization techniques. As an early example, they mention real-time simulations between different models of Turing machines.

A textbook example of amortized data structures are resizable arrays. The classical technique for implementing arrays whose size may grow is "doubling": When the array overflows its current size, we allocate a storage block that is twice as large. The array must be copied to the new location, and this takes linear time. But this burst of activity occurs sufficiently rarely so that the *amortized* complexity for extending an array by one element is constant. To convert this into a worst-case bound, one has to distribute the copying operation over the subsequent insertion operations. For a while, an old and a new copy of the array must be maintained simultaneously. In this case, when comparing the timing with the simple amortized algorithm, one would rather say that the real-time algorithm is working *behind*. This procedure is an instance of *global rebuilding* (see Overmars [16, Chapter V]), a de-amortization technique that applies to more general data structures under appropriate conditions.

In a similar vein, Guibas, McCreight, Plass, and Janet R. Roberts [17] 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.

Another example, which is less well known, are functional queues. In a purely functional language, one cannot perform assignments, and thus, it is not possible to join two linked lists together in constant time. The native list structure in such languages is a stack. A queue can be simulated by two stacks, reversing the "arrival stack" onto the "departure stack" whenever the latter becomes empty. This achieves constant *amortized* runtime for the queue operations. It is not straightforward to design real-time queues that achieve constant time in the worst case, see Hood and Melville [18] and Okasaki [14, 19].

For our task of combinatorial generation, the setting is much simpler, because we need not process requests of an unpredictable "user" in an on-line setting. We can plan everything in advance. We *work ahead* in the sense that the algorithm performs work that is not necessary for producing the current output. However, in this context it would be equally justified to say that we just *delay the output*.

Although the main idea of working ahead is straightforward, our loopless generation algorithms that are based on this idea require a nontrivial analysis of the buffer size. If we let the buffer grow without restrictions, we would need exponential space, except in the binary case (see Lemma 4). With a bounded buffer, we have to make sure that the opportunities for carrying out STEPs that are waisted due to a full buffer do not harm the success of the operation (Lemma 2).

We have recently applied the same technique to derive new loopless enumeration algorithms for permutations [20], using functional programming techniques. Since permutations of n elements can be related to mixed-radix Gray codes with radices $(m_{n-1}, \ldots, m_1) = (2, 3, 4, \ldots, n-1, n)$, our analysis can be applied. We are aware of only two previous instances where the idea of working ahead has been used in the area of combinatorial enumeration. The first is an algorithm of Wettstein for enumerating non-crossing perfect matchings of a planar point set. The idea is described in the preprint [21, Section 6], where it is credited to Emo Welzl; in the conference version [22], it is only mentioned. Wettstein combines the work-ahead idea with a rearrangement of the output sequence. In this way, he achieves *polynomial delay* between successive solutions, and in particular, before the first solution, despite having to build a network with exponential space in a preprocessing phase. Here we are at a different level of complexity, asking about polynomial time, whereas looplessness is about constant runtime.

The second instance is in a context similar to ours: generating a Gray code of all bitstrings of length $2n+1$ that contain n or $n+1$ ones. A recent algorithm of Mütze and Nummenpalo [23] can do this with $O(1)$ average runtime per bitstring. Even the existence of such a Gray code had been a long-standing open problem, and this algorithm is much more involved than our simple Gray code examples. The possibility to make the algorithm loopless by buffering the output is mentioned in the introduction of [23] in the remarks after Theorem 3. The algorithm strictly alternates between $\Theta(n)$ generation steps that take constant time and single steps that take $O(n)$ time. Thus, the organization of the buffer that is required for achieving looplessness would be straightforward.

10.2. An alternative STEP procedure

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

procedure STEP′:
    if $i = n + 1$: TERMINATE
    if $b_i = m_i - 1$:
        $b_i := 0$
        $i := i + 1$
    else:
        if Q is not filled to capacity:
            $b_i := b_i + 1$
            $Q.append(i)$
            $i := 1$

With this modified procedure 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 STEP′ signals termination. Let us prove this:

The termination signal is sent instead of producing the value $i = n + 1$. 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 $n + 1$ was started during iteration j_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 j_0, and all B_0 entries of the buffer have been used up at the beginning of iteration $j_0 + B_0$. 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 $i = n + 1$, 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 **if**-branch after the statement $i := i + 1$. 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 (p. 14).

10.3. Analysis and correctness proofs for the work-ahead algorithms

Let us first analyze the running time for each iteration of Algorithm DELTA. We can explicitly express the elements of the delta sequence in terms of the *ruler function* ρ. The ruler function ρ with respect to a sequence of radixes $m_1, \ldots, m_n$ is defined as follows:

$$\rho(j) := 1 + \max\{\, i : 0 \leq i \leq n,\ m_1 m_2 \ldots m_i \text{ divides } j \,\}$$

The delta sequence is nothing but the sequence $\rho(1), \rho(2), \ldots$, and the j-th value that is entered into Q is $\rho(j)$. For computing this value, Algorithm DELTA needs $\rho(j)$ iterations, and accordingly, Algorithm WORK-AHEAD needs $\rho(j)$ STEPs.

Lemma 2. *In Algorithm WORK-AHEAD, the buffer Q never becomes empty.*

Proof. We number the iterations of the main loop as $j = 1, 2, \ldots, m_1 m_2 \ldots m_n$. In the last iteration, the algorithm terminates.

Let us show that the queue Q is not empty in iteration j. We distinguish two cases.

a) Up to and including iteration j, two repetitions of STEP were always completed.

b) Some repetitions of STEP had no effect because the buffer Q was full.

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

$$S(j) := \sum_{i=1}^{j} \rho(i)$$

calls to STEP. To show that these calls are completed by the time when $\rho(j)$ is needed, we have to show

$$S(j) \le 2j. \tag{4}$$

In case (b), let j_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 $k = \rho(j_0)$, and it contained the values $\rho(j_0), \rho(j_0+1), \ldots, \rho(j_0+B-1)$. Since then, STEP was called $2(j - j_0)$ times, and $\rho(j)$ is ready when it is needed, provided that

$$1 + \sum_{i=j_0+B+1}^{j} \rho(i) \le 2(j - j_0)$$

whenever $j \ge j_0 + B$. The left-hand side of this inequality is the number of necessary STEPs for computing the values up to $\rho(j)$. Computing $\rho(j_0 + B)$ takes just one more STEP, since the STEP that would have stored this value in Q was abandoned in iteration j_0. Setting $j' = j_0 + B$, we can express the inequality equivalently as

$$S(j) - S(j') \le 2(j - j' + B) - 1 \text{ for } j' \le j \tag{5}$$

Now that we have worked out the inequalities (4–5) that we need, let us prove them. We can write an explicit formula for $S(j)$:

$$S(j) = j + \left\lfloor \frac{j}{m_1} \right\rfloor + \left\lfloor \frac{j}{m_1 m_2} \right\rfloor + \cdots + \left\lfloor \frac{j}{m_1 m_2 \ldots m_n} \right\rfloor$$

Since all $m_i \ge 2$, we get $S(j) \le j + j/2 + j/4 + j/8 + \cdots + j/2^n < 2j$, proving (4). For the other bound (5), we apply the relation $\lfloor x \rfloor - \lfloor x' \rfloor < x - x' + 1$ to the difference between corresponding terms of $S(j)$ and $S(j')$, and we get

$$S(j) - S(j') < (j - j') + (j - j') \cdot (\tfrac{1}{2} + \tfrac{1}{4} + \tfrac{1}{8} + \cdots + \tfrac{1}{2^n}) + n < 2(j - j') + n.$$

Since the left-hand side is an integer, we obtain $S(j) - S(j') \le 2(j - j') + n - 1$, and this implies (5) because the buffer size $B := \lceil \frac{n}{2} \rceil$ satisfies $2B \ge n$. □

In Algorithm WORK-AHEAD, the STEPs should generate entries $\rho(1), \rho(2), \ldots$ of Q up to $\rho(N)$, where $N := m_1 m_2 \ldots m_n$. The production of the STEPs may overshoot their target N, but the following lemma shows that is overshoots the 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 from size n to size $n + 1$, this one extra entry does not cause any harm. We could even tolerate the generation of delta-values up to $\rho(2N - 1)$.

Lemma 3. *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 $n+1 \geq 2B$ STEPs. 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. $\square$

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

Lemma 4. *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 j, $1 \leq j \leq 2^n$. This means that, before $k = \rho(j)$ is removed from Q, the $2j$ STEP operations have produced more than $j - 1 + B'$ values. But this is impossible, since, as we will show, the production of the first $j_1 = j + B'$ values takes strictly more than $2j$ STEPs. In terms of formulas, this is the following inequality:

$$S(j_1) = j_1 + \left\lfloor \frac{j_1}{2} \right\rfloor + \left\lfloor \frac{j_1}{2^2} \right\rfloor + \cdots + \left\lfloor \frac{j_1}{2^n} \right\rfloor > 2j$$

To show this inequality, we first consider the case $j_1 < 2^n$. We apply the inequality $\lfloor x \rfloor > x - 1$ to each term and obtain $S(j_1) > 2j_1 - j_1/2^n - n$, and since $j_1/2^n < 1$ and $S(j_1)$ is an integer, we get

$$S(j_1) \geq 2j_1 - n = 2j + 2B' - n > 2j.$$

Let us now look at the other case see at what time the first entries $\rho(j_1)$ with $j_1 \geq 2^n$ are entered into Q. When $j_1 = 2^n$, no round-off takes place in the formula for $S(j_1)$, and we have $S(2^n) = 2 \cdot 2^n - 1$. This shows that the production of $\rho(2^n)$ is completed in the first STEP of iteration 2^n. In the second STEP of this iteration, $\rho(2^n + 1) = 1$ is added to Q. Thus, when $\rho(2^n)$ is about to be retrieved, the buffer contains $2 \leq B'$ elements. Then the algorithm terminates, and no more elements are produced. $\square$

11. Conclusion

We have shown that the consideration of games can give inspiration for new loopless algorithms for electronic computers. Our approach of modeling the Gray code in terms of a motion-planning game has lead to loopless algorithms for Gray codes in a rather straightforward way. We did not have to go through "contortions" (cf. the quote in the end of Section 2, p. 5).

Loopless algorithms for enumerating Gray codes were already known, cf. [2, 7.2.1.1.H], and thus we did not achieve new results in terms of improved asymptotic running time. In particular, we do not claim superiority of these algorithms over the existing algorithms. Such a comparison would depend on the hardware and on other factors. All we can say is that these algorithms enrich the arsenal of available algorithms for loopless generation. Still, it might be an interesting exercise to program these algorithms for Knuth's model computer `MMIX`[6] and analyze their performance.

The approach of Section 10 was very different. It used a de-amortizing technique for data structures, and applied it to loopless generation algorithms. The amortized analysis that goes with this technique was straightforward (inequality (4)), and the resulting algorithms are simple. The analysis of the required buffer size was, however, more intricate.

11.1. Open questions

With our approach, 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.

[6] `www-cs-faculty.stanford.edu/~knuth/mmix.html`

Another motion-planning game which is related to the binary Gray code is the *Chinese rings* puzzle, see Gardner [4], Knuth [2, pp. 285–286], or Scorer, Grundy, and Smith [3, Section 1]. Knuth [2, 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 [24], Knuth [2, Solution to Ex. 7.2.1.1–(12b), p. 680]. However, this algorithm does not seem to extend to other radixes. Scorer et al. [3, Section 5] analyzed a generalization of the Chinese rings. We did not investigate whether it leads to interesting Gray codes.

[1] J. R. Bitner, G. Ehrlich, E. M. Reingold, Efficient Generation of the Binary Reflected Gray Code and Its Applications, Commun. ACM 19 (9) (1976) 517–521, ISSN 0001-0782, doi:\let\@tempa\bibinfo@X@doi10.1145/360336.360343.

[2] D. E. Knuth, Combinatorial Algorithms, Part 1, vol. 4A of *The Art of Computer Programming*, Addison-Wesley, 2011.

[3] R. S. Scorer, P. M. Grundy, C. A. B. Smith, Some binary games, The Mathematical Gazette 28 (280) (1944) 96–103, ISSN 00255572, URL `http://www.jstor.org/stable/3606393`.

[4] M. Gardner, The curious properties of the Gray code and how it can be used to solve puzzles, Sci. American 227 (1972) 106–109.

[5] P. Buneman, L. Levy, The Towers of Hanoi problem, Information Processing Letters 10 (4–5) (1980) 243–244.

[6] F. Herter, G. Rote, Loopless Gray code enumeration and the Tower of Bucharest, in: E. D. Demaine, F. Grandoni (Eds.), Proceedings of the 8th International Conference on Fun with Algorithms (FUN 2016), vol. 49 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 19:1–19:19, doi:\let\@tempa\bibinfo@X@doi10.4230/LIPIcs.FUN.2016.19, 2016.

[7] F. Herter, G. Rote, Loopless Gray Code Enumeration and the Tower of Bucharest, preprint arXiv:1604.06707 [cs.DM], 2016.

[8] D. S. Johnson, M. Yannakakis, C. H. Papadimitriou, On generating all maximal independent sets, Information Processing Letters 27 (3) (1988) 119–123, ISSN 0020-0190, doi:\let\@tempa\bibinfo@X@doi10.1016/0020-0190(88)90065-8.

[9] G. Ehrlich, Loopless Algorithms for Generating Permutations, Combinations, and Other Combinatorial Configurations, J. Assoc. Comput. Mach. 20 (3) (1973) 500–513, ISSN 0004-5411, doi:\let\@tempa\bibinfo@X@doi10.1145/321765.321781.

[10] A. Sapir, The towers of Hanoi with forbidden moves, The Computer Journal 47 (1) (2004) 20–24.

[11] A. M. Hinz, S. Klavžar, U. Milutinović, C. Petr, The Tower of Hanoi — Myths and Maths, Birkhäuser, 2013.

[12] R. L. Graham, D. E. Knuth, O. Patashnik, Concrete Mathematics, Addison-Wesley, 1989.

[13] D.-J. Guan, Generalized Gray Codes with Applications, Proc. Natl. Sci. Council, Republic of China (A) 22 (6) (1998) 841–848.

[14] C. Okasaki, Purely Functional Data Structures, Cambridge University Press, 1998.

[15] S. R. Kosaraju, M. Pop, De-amortization of algorithms (preliminary version), in: W.-L. Hsu, M.-Y. Kao (Eds.), Computing and Combinatorics: 4th Annual International Conference, COCOON'98, Taipei, Taiwan, R.o.C., August 12–14, 1998, Proceedings, vol. 1449 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, Heidelberg, ISBN 978-3-540-68535-7, 4–14, doi:\let\@tempa\bibinfo@X@doi10.1007/3-540-68535-9_4, invited presentation, 1998.

[16] M. H. Overmars, The Design of Dynamic Data Structures, vol. 158 of *Lecture Notes in Computer Science*, Springer-Verlag, 1983.

[17] L. J. Guibas, E. M. McCreight, M. F. Plass, J. R. Roberts, A New Representation for Linear Lists, in: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, STOC '77, ACM, New York, NY, USA, 49–60, doi:\let\@tempa\bibinfo@X@doi10.1145/800105.803395, 1977.

[18] R. Hood, R. Melville, Real-time queue operations in pure LISP, Information Processing Letters 13 (2) (1981) 50–54, ISSN 0020-0190, doi:\let\@tempa\bibinfo@X@doi10.1016/0020-0190(81)90030-2.

[19] C. Okasaki, Simple and efficient purely functional queues and deques, J. Functional Programming 5 (4) (1995) 583–592.

[20] G. Rote, Loopless generation of permutations by adjacent transpositions, in preparation, 2017.

[21] M. Wettstein, Counting and enumerating crossing-free geometric graphs, preprint arXiv:1604.05350 [cs.CG], 2016.

[22] M. Wettstein, Counting and enumerating crossing-free geometric graphs, in: Proceedings of the Thirtieth Annual Symposium on Computational Geometry, SOCG'14, ACM, New York, NY, USA, ISBN 978-1-4503-2594-3, 1:1–1:10, doi:\let\@tempa\bibinfo@X@doi10.1145/2582112.2582145, 2014.

[23] T. Mütze, J. Nummenpalo, A constant-time algorithm for middle levels Gray codes, preprint arXiv:1606.06172 [cs.DM], 2016.

[24] J. Misra, Remark on Algorithm 246, ACM Trans. Math. Software 1 (3) (1975) 285.