Radboud University Nijmegen

Faculty of Science

Gray codes and other paths through hypercubes

Finding efficient algorithms to determine the next step

Thesis BSc Mathematics

Author:
Sebastiaan van Krieken
s4496892

Supervisor:
dr. Wieb Bosma

Second reader:
dr. Henk Don

July 2021

Contents

1 Introduction **3**
1.1 Notation . . . 3

2 Gray codes **5**
2.1 Binary Gray codes . . . 5
2.1.1 Reflected Gray codes . . . 5
2.1.2 Converting binary reflected Gray codes to binary numbers and vice versa . . . 5
2.2 Generalized Gray codes . . . 6
2.2.1 Reflected Gray codes over an alphabet . . . 6
2.2.2 Recursive definition for reflected Gray codes for an even sized alphabet . . . 6
2.2.3 A memoryless iterative algorithm for reflected Gray codes with an even sized alphabet . . . 7
2.2.4 Recursive definition for reflected Gray codes for an odd sized alphabet 10
2.2.5 A memoryless iterative algorithm for reflected Gray codes with an odd sized alphabet . . . 10
2.2.6 Knuth's loopless algorithm . . . 11

3 Half growing cube **13**
3.1 The problem . . . 13
3.2 Modeled as a Hamiltonian path through a growing hypercube . . . 13
3.2.1 One dimension . . . 13
3.2.2 Two dimensions . . . 13
3.2.3 Three dimensions . . . 14
3.2.4 Higher dimensions . . . 15
3.3 Recursive definition half growing cube algorithm . . . 16
3.3.1 Alternative half growing algorithm . . . 23
3.4 Recursive half growing cube algorithm implementation . . . 23
3.5 Constant time half growing cube algorithm . . . 25

4 Fully growing cube **29**
4.1 The problem . . . 29
4.2 Modeled as a Hamiltonian path through a growing hypercube . . . 29
4.2.1 One dimension . . . 29
4.2.2 Two dimensions . . . 29
4.2.3 Three dimensions . . . 30
4.2.4 Chessboard argument . . . 30
4.2.5 Four dimensions . . . 31
4.2.6 Even dimensions . . . 35
4.2.7 Some alternatives for odd dimensions . . . 35
4.3 Recursive definition fully growing cube algorithm for even dimensions . 35
4.4 Recursive fully growing Cube algorithm for even dimensions implementation 37
4.5 Constant time fully growing cube algorithm . . . 38

1 Introduction

When taking the 2-bit binary numbers in their normal order: 00, 01, 10, 11, we see that when we move from 01 to 10, we have to change both bits. If we, however, order them 00, 10, 11, 10, we only have to change one bit per step. Such an ordering for n-bit binary numbers is called a Gray code, named after the inventor Frank Gray, who used it in a patent in 1947. However due to its useful applications, it was not the first occurrence of the code.

For instance, if you have a bunch of levers and manually want to try every combination, using a Gray code will save you a lot of work, since you only have to pull one lever each time. Also, in some situations it is essential that we only change one bit, since when changing multiple bits, there might be a small moment of time where only a subset of these bits have changed, which might cause problems. We will go into the most common Gray code, the so called *reflected Gray codes*.

Now if we want to try every combination of a combination lock in an efficient way, where we pass every combination exactly once and we only turn one wheel at each step and this wheel we turn a minimal distance, we come to what is called the *Generalized Gray codes*. These are Gray codes, where instead of an alphabet of just $\{0, 1\}$, we use a more generalized alphabet $\{0, 1, .., m\}$. For one of these Generalized Gray codes, the reflected Generalized Gray codes, we will implement an algorithm which will give the tuple that follows the input tuple. This algorithm has linear complexity, is iterative and does not require any extra memory. We will also show an algorithm that works in constant time, but does require extra memory.

Now if you imagine a combination lock where every wheel goes from zero to infinity, with a set number of wheels, and we want to try every combination with minimal effort and an extra requirement: the combinations with highest number m should be tried before the combinations with highest number $> m$, you get the "half growing" Gray code. It is called this way, because if you visualize the path, you get a Hamiltonian path through a hypercube, where half the sides keep getting an extra layer. For this problem, we will define a working path for which we will implement both a linear time algorithm that does not require extra memory and a constant time algorithm that requires extra memory the same size as the input tuple. The same problem, but where the "wheels" go from minus infinity to infinity is called the "fully growing" Gray code. In this case the code gives a path through a hypercube that adds an extra layer on every side. For this problem we will give a linear algorithm for the cases with even dimensions. Sadly it is not possible for odd dimensions. The "half growing" and "fully growing" Gray code have useful applications, for instance they could be used to find polynomials with specific characteristics, especially if we want to minimize the maximum value of the coordinates.

For further reading and general information on Gray codes not covered in this thesis, I recommend Knuth's The Art of Computer Programming 4a [2](pages 281-319) and as a matter of fact the Wikipedia page[3] on Gray codes is an excellent introductory source as well.

1.1 Notation

In this section we will highlight a few notations that are used throughout the thesis:

- Often tuples will be written as numbers, for example we will sometimes write *abcd* for the tuple (a, b, c, d).
- When we have a tuple $x = (x_1, .., x_{d-1}, x_d)$, we will write x_i^j when we want the tuple that only contains the coordinates between i and j, $(x_i, .., x_j)$. Not to be confused with x_i to the power j.

- Generally, in computer science indices start at 0 and in mathematics indices start at 1. This thesis is a bit inconsistent in this regard, since in Chapter 2 the indices start at 0 and in the other chapters the indices start at 1. This is the result of a discrepancy between a desire to resemble the actual code and a personal preference.

- Some functions are defined in such way that it may seem that the same input tuple will be mapped to two different outputs. In this case the function actually maps the input to the first output. For example, we might have a function defined as follows:

$$\begin{array}{lcll} (0,..,0,m,0) & \mapsto & (0,..,0,m+1,0) & \\ (0,..,0,m,a) & \mapsto & (0,..,0,m,a-1) & \text{with } 2|a \\ (0,..,0,m,0,a) & \mapsto & (0,..,0,m,0,a-1) & \text{with } 2 \nmid a \\ x_1^{d-1}a & \mapsto & \mathrm{HG}_{d-1}^{-1}(x_1^{d-1})a & \text{with } 2|a \\ x_1^{d-1}a & \mapsto & \mathrm{HG}_{d-1}(x_1^{d-1})a & \text{with } 2 \nmid a \end{array}$$

  Then the input tuple $(0,..,0,m,0)$ coincides with both the first, second and fourth line. In that case the output is the one given by the first line.

2 Gray codes

2.1 Binary Gray codes

Definition 2.1. A **binary Gray code of n-bit numbers** is an ordering of all n-bit binary numbers, such that two successive numbers differ in exactly one bit.

Analogously, a binary Gray code can be defined for n-tuples:

Definition 2.2. A **binary Gray code of n-tuples** is a bijection $g : \{0,1\}^n \to \{0,1,...,2^n-1\}$, such that $g(x) - g(y) = 1 \implies \#\{i | 0 \leq i < n, x_i \neq y_i\} = 1$.

2.1.1 Reflected Gray codes

One simple way of recursively constructing an n-bit binary Gray code, is by taking the elements of a binary Gray code of $n-1$ bits and prefixing them with a 0 and then adding to this list the elements of the binary Gray code of $n-1$ bits in the reverse order, prefixed with a 1. If we start with the 1 bit Gray code 0,1 and repeatedly apply the aforementioned algorithm, we will create the **reflected Gray codes for n-bit numbers**.

If we start with the order 0 1, the resulting order of the 4-bit binary reflected Gray code will become:

0:	0000	8:	1100
1:	0001	9:	1101
2:	0011	10:	1111
3:	0010	11:	1110
4:	0110	12:	1010
5:	0111	13:	1011
6:	0101	14:	1001
7:	0100	15:	1000

By convention, just like normal binary numbers, the reflected Gray code for binary numbers "grows" to the left. That is to say that the head of the reflected Gray code for n-bits is simply equal to the reflected Gray code for $(n-1)$-bits where the elements are prefixed with a 0. However, if we want to use tuples or arrays it makes more sense to have the code "grow" to the right, since arrays themselves generally "grow" to the right. In this case, we suffix the reflected Gray code of one dimension less with a 0 and we suffix its mirror with 1. This will give us the **binary reflected Gray codes for n-tuples:**

0:	(0,0,0,0)	8:	(0,0,1,1)
1:	(1,0,0,0)	9:	(1,0,1,1)
2:	(1,1,0,0)	10:	(1,1,1,1)
3:	(0,1,0,0)	11:	(0,1,1,1)
4:	(0,1,1,0)	12:	(0,1,0,1)
5:	(1,1,1,0)	13:	(1,1,0,1)
6:	(1,0,1,0)	14:	(1,0,0,1)
7:	(0,0,1,0)	15:	(0,0,0,1)

2.1.2 Converting binary reflected Gray codes to binary numbers and vice versa

There is a simple function to convert the mth binary number to the mth element of the binary reflected Gray Code. If we model the number as a tuple, the function is $a : \{0,1\}^n \to \{0,1\}^n$ with $a((x_0,..,x_{n-1}))_i = x_i \mathbf{xor} x_{i+1}$, with $x_n = 0$. In other words, the value at index i of the output is the **xor** function applied to the values at indices i

and $i+1$ of the input. The reason this works, is because of the reflective nature: Take an arbitrary $m \in \mathbb{N}$ and let $d \in \mathbb{N}$ be the smallest natural number such that $2^d > m$. Then the first $d-1$ bits of the mth element of the Gray code coincide with the first $d-1$ bits of the (2^d-1-m)th element of the Gray code. You get the (2^d-1-m)th binary number by complementing all the bits of the mth binary number, and when complementing all the bits, the **xor** of two adjacent bits remains the same. A simple formal proof can then be given by using induction on the size of the binary number, but has been left out in this thesis.

The function to convert the mth binary reflected Gray Code to the mth binary number is slightly more difficult, but still quite simple too. Again, we model the number as a tuple, and the function is $b : \{0,1\}^n \to \{0,1\}^n$ with $b((x_0,..,x_{n-1}))_i = x_i \mathbf{xor} x_{i+1} \mathbf{xor} .. \mathbf{xor} x_{n-1}$. In other words, the value at index i of the output is the **xor** function applied to the values at indices i to $n-1$ of the input. The validity of this function can be seen by using the previous function and induction (once again a formal proof has been left out): By the previous function, we know $x_i = b((x_0,..,x_{n-1}))_i \mathbf{xor} b((x_0,..,x_{n-1}))_{i+1}$, but then also $b((x_0,..,x_{n-1}))_i = x_i \mathbf{xor} b((x_0,..,x_{n-1}))_{i+1}$, by the nature of the **xor** operator. Then we can use induction to prove $b((x_0,..,x_{n-1}))_i = x_i \mathbf{xor} x_{i+1} \mathbf{xor} .. \mathbf{xor} x_{n-1}$, with induction hypothesis $b((x_0,..,x_{n-1}))_{i+1} = x_{i+1} \mathbf{xor} .. \mathbf{xor} x_{n-1}$.

2.2 Generalized Gray codes

In the previous section, we restricted ourselves to Gray codes of an alphabet of size two. In this section we want to expand this definition to larger alphabets.

We can interpret a Gray code as a strict total order, here modeled as a bijection with a subset of the natural numbers.

Definition 2.3. A **Gray code of n-tuples over an alphabet** $\mathbb{A} = \{0,..,m\}$ is a bijection $g : \mathbb{A}^n \to \{0,1,...,(m+1)^n - 1\}$, such that $g(x) - g(y) = 1 \implies K(x,y) = 1$, where $K((x_0,..x_{n-1}),(y_0,..,y_{n-1})) = \sum_{i=0}^{n-1} |x_i - y_i|$.

We can also interpret a Gray code as a bijection from a set of tuples to itself, where we get the next tuple in the code by applying this function.

Definition 2.4. A **Gray code of length n over alphabet** $\mathbb{A} = \{0,..,m\}$ is a bijective function $h : \mathbb{A}^n \setminus \{x_{\text{last}}\} \to \mathbb{A}^n \setminus \{0..0\}$, where $x_{\text{last}} \in \mathbb{A}^n \setminus \{0..0\}$, such that for all $x \in \mathbb{A}^n \setminus \{x_{\text{last}}\}$, $K(x, h(x)) = 1$, where $K((x_0,..x_{n-1}),(y_0,..,y_{n-1})) = \sum_{i=0}^{n-1} |x_i - y_i|$.

In other words, we want to order all tuples of length n with digits from alphabet $\mathbb{A} = \{0,..,m\}$ in such a way, that every tuple differs from the previous one in exactly 1 digit, and the difference between these two differing digits must be minimal.

2.2.1 Reflected Gray codes over an alphabet

Just like with the reflected binary Gray codes, we can use the Gray code of dimension $(n-1)$ to construct a Gray code of dimension n. The result does slightly depend on whether the size of the alphabet is even or odd, therefore we have split into two cases.

2.2.2 Recursive definition for reflected Gray codes for an even sized alphabet

Let $\mathbb{A} = \{0,1,..,m\}$ be an alphabet of even size. Assume that the reflected Gray codes for sizes $k < n$ are known. Let $f_k : \mathbb{A}^k \setminus \{0..0m\} \to \mathbb{A}^k \setminus \{0..00\}$ be the function that takes a tuple of size k and returns the tuple that comes next in the reflected Gray code. These functions f_k are then inductively defined by the following rules:

For tuples of length 1:

$$\begin{aligned} f_1 : \mathbb{A} \setminus \{m\} &\to \mathbb{A} \setminus \{0\} \\ f_1(a) &= a + 1 \end{aligned}$$

And for f_{n+1}, given f_n:

$$\begin{aligned} f_{n+1}(0..0mx_n) &= 0..0m(x_n + 1) && \text{if } 2|x_n \\ f_{n+1}(0..00x_n) &= 0..00(x_n + 1) && \text{if } 2 \nmid x_n \\ f_{n+1}(x) &= f_n(x_0^{n-1})x_n && \text{if } 2|x_n \\ f_{n+1}(x) &= f_n^{-1}(x_0^{n-1})x_n && \text{if } 2 \nmid x_n \end{aligned}$$

where $x_i^j := x_i x_{i+1} ... x_{j-1} x_j$ for a tuple $x = x_0 x_1 .. x_n$.

2.2.3 A memoryless iterative algorithm for reflected Gray codes with an even sized alphabet

In this section we will construct an iterative algorithm whose input and output acts the same as the function defined in section 2.2.2. The intuition behind the algorithm is that we want to find out in which direction the element at index 0 is moving. Let us say we have the tuple $x = x_0 x_1 .. x_n$ of length $n + 1$. When x_n is even, the direction of x_0 is the same as when we would have just looked at the tuple x_0^{n-1}. On the other hand, when the number at the last index is odd, the direction of x_0 is switched compared to the direction it would have had if we were just looking at the tuple x_0^{n-1}. The principle of the algorithm is that we move from index n to index 1 and we count how many times the direction switches and then change the value at index 0 accordingly. Since the value at index 0 can not go below 0 or above m, there is a couple of cases where the algorithm is not quite as simple. A comparison can be made with ordinary numbers: If we add 1 to an ordinary number, usually only the rightmost digit increases, unless the rightmost digit equals 9, in which case we have a carry and change some digits on other indices too. The difference is that with Gray codes we still only change one digit, just not the one at index 0.

We define the auxiliary function $P(x) = (\sum_{i=0}^{n} x_i) \mod 2$, which checks if the sum of the digits of a tuple is even or odd, also known as the parity. P is used to decide if the direction of the remaining tuple is "forwards" or "backwards".

$A_{n+1} : \mathbb{A}^{n+1} \setminus \{0..0m\} \to \mathbb{A}^{n+1} \setminus \{0..0\}$ is the function given by the algorithm. This is what the algorithm does for an input tuple of length $n + 1$, $x = x_0 x_1 ... x_n$:

- If $P(x_1^n) = 0$ and $x_0 \neq m$: return $(x_0 + 1)x_1^n$
- If $P(x_1^n) = 1$ and $x_0 \neq 0$: return $(x_0 - 1)x_1^n$
- If $P(x_1^n) = 0$ and $x_0 = m$:
  - if $P(x_1) = 0$: return $x_0(x_1 + 1)x_2^n$
  - if $P(x_1) = 1$: return $x_0(x_1 - 1)x_2^n$
- If $P(x_1^n) = 1$ and $x_0 = 0$: let $j = \min \{0 < j \leq n | x_j \neq 0\}$
  - if $P(x_j) = 0$: return $0..0(x_j - 1)x_{j+1}^n$
  - if $P(x_j) = 1$ and $x_j \neq m$: return $0..0(x_j + 1)x_{j+1}^n$

- if $x_j = m$: in this case we know that $j < n$, since the function is not defined for $0..0m$
  - if $P(x_{j+1}) = 0$: return $0..0x_j(x_{j+1}+1)x_{j+2}^n$
  - if $P(x_{j+1}) = 1$: return $0..0x_j(x_{j+1}-1)x_{j+2}^n$

Correctness of algorithm: In order to prove the correctness of the algorithm, we have to show that for all $n \in \mathbb{Z}_{>0}$ the function A_n, given by the algorithm, coincides with the function f_n, given by the recurrence relations in Section 2.2.2.

IB: The size of the tuple is 1: Let $a \in \mathbb{A} \setminus \{m\}$. Since the parity of the empty tuple is 0, $A_1(a) = a + 1 = f_1(a)$. So $A_1 = f_1$.

IH: We assume that $A_n = f_n$.

IS: In this part of the proof we have to check for all different cases that the algorithm A_{n+1} coincides with the recursive definition of f_{n+1}. We split into four main cases given by the parity of the tuple without the digit at index 0 and whether the digit at index 0 can be changed or not.

If $P(x_1^n) = 0$ and $x_0 \neq m$: Then $A_{n+1}(x) = (x_0 + 1)x_1^n$.

- if $P(x_n) = 0$, then $P(x_1^{n-1}) = 0$, so $A_n(x_0^{n-1}) = (x_0 + 1)x_1^{n-1}$. This means that $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$, by the recursive relation.
- if however $P(x_n) = 1$, then $P(x_1^{n-1}) = 1$, so $A_n((x_0 + 1)x_1^{n-1}) = x_0x_1^{n-1}$. This shows us that $A_{n+1}(x) = A_n^{-1}(x_0^{n-1})x_n = f_n^{-1}(x_0^{n-1})x_n = f_{n+1}(x)$.

If $P(x_1^n) = 1$ and $x_0 \neq 0$: Then $A_{n+1}(x) = (x_0 - 1)x_1^n$.

- if $P(x_n) = 0$, then $P(x_1^{n-1}) = 1, A_n(x_0^{n-1}) = (x_0 - 1)x_1^{n-1}$. This means that $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$, by the recursive relation.
- if however $P(x_n) = 1$, then $P(x_1^{n-1}) = 0$, so $A_n((x_0 - 1)x_1^{n-1}) = x_0x_1^{n-1}$. This means that $A_{n+1}(x) = (x_0-1)x_1^{n-1}x_n = A_n^{-1}(x_0^{n-1})x_n = f_{n-1}^{-1}(x_0^{n-1})x_n = f_{n+1}(x)$, by the recursive relation.

If $P(x_1^n) = 0$ and $x_0 = m$: we may assume that $n > 0$, since $f_1(m)$ is not defined. We will split into four cases given by the parity of x_1 and x_n and we may assume that $n > 1$ except in the case where $P(x_n) = 0$ and $P(x_1) = 0$. We first split into the two situations given by the parity of x_1:

- $P(x_1) = 0$: Then $A_{n+1}(x) = x_0(x_1 + 1)x_2^n$. We again split into two cases:
  - $P(x_n) = 0$: It is possible that $n = 1$, in which case x is of the form $x = ma$, with $a \in \mathbb{A}$ and $2|a$. The algorithm A_1 gives $A_1(ma) = m(a + 1)$ which coincides with $f_1(ma) = m(a + 1)$, where we use that ma is of the form $0...0ma$ with no zeroes. If however $n > 1$, then $A_n(x_0^{n-1}) = x_0(x_1 + 1)x_2^{n-1}$, because $x_0 = m$, $P(x_1^{n-1}) = 0$ and $P(x_1) = 0$. This means that $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$.
  - $P(x_n) = 1$: Then $A_n(x_0(x_1 + 1)x_2^{n-1}) = x_0^{n-1}$, because $P((x_1 + 1)x_2^n) = 0$ and $P(x_1 + 1) = 1$ and $x_0 = m$. So $x_0(x_1 + 1)x_2^n = A_n^{-1}(x_0^{n-1})$, and $A_{n+1}(x) = A_n^{-1}(x_0^{n-1})x_n = f_n^{-1}(x_0^{n-1})x_n = f_n(x)$.
- $P(x_1) = 1$: Then $A_{n+1}(x) = x_0(x_1 - 1)x_2^n$ (Note that this is well-defined, since $P(x_1) = 1$ implies $x_1 \neq 0$). Again, split in two cases:
  - $P(x_n) = 0$: Then $A_n(x_0^{n-1}) = x_0(x_1 - 1)x_2^n$, because $P(x_1^{n-1}) = 0$, $x_0 = m$ and $P(x_1) = 1$. So $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$.

– $P(x_n) = 1$: Then $A_n(x_0(x_1 - 1)x_2^{n-1}) = x_0x_1x_2^{n-1} = x_0^{n-1}$, because $P((x_1 - 1)x_2^{n-1}) = 0$, since $P(x_n) = 1$, $P((x_1 - 1)) = 1 - P(x_1)$ and $P(x_1^n) = 0$, and $x_0 = m$. This means that $A_{n+1}(x) = x_0(x_1 - 1)x_2^n = A_n^{-1}(x_0^{n-1})x_n = f_n^{-1}(x_0^{n-1})x_n = f_{n+1}(x)$.

If $P(x_1^n) = 1$ and $x_0 = 0$: We define j as $j = \min\{0 < j \leq n | x_j \neq 0\}$.

- $P(x_j) = 0$: Then $A_{n+1}(0..0x_jx_{j+1}^n) = 0..0(x_j - 1)x_{j+1}^n$. We know $j \neq n$, because then $P(x_1^n) = 0$. We split into two cases:
  - $P(x_n) = 0$: In this case $A_n(x_0^{n-1}) = 0..0(x_j - 1)x_{j+1}^n$, because $P(x_1^{n-1}) = 1$ and $P(x_j) = 0$. So, $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$.
  - $P(x_n) = 1$: In this case $A_n(0..0(x_j - 1)x_{j+1}^{n-1}) = 0..0x_jx_{j+1}^{n-1} = x_0^{n-1}$, because $P(x_1^{j-1}(x_j - 1)x_{j+1}^{n-1}) = 1$, $x_0 = 0$, $P(x_j - 1) = 1$ and $P(x_j - 1) \neq m$. This means that $A_{n+1}(x) = A_n^{-1}(x_0^{n-1})x_n = f_n^{-1}(x_0^{n-1})x_n = f_{n+1}(x)$.
- $P(x_j) = 1$ and $x_j \neq m$: Then $A_{n+1}(0..0x_jx_{j+1}^n) = 0..0(x_j + 1)x_{j+1}^n$.
  - $j = n$: In this case x is of the form $0..0a$, with $P(a) = 1$. $f_{n+1}(0..0a) = 0..0(a + 1) = A_{n+1}(x)$.
  - $j < n$ and $P(x_n) = 0$: $A_n(x_0^{n-1}) = 0..0(x_j + 1)x_{j+1}^{n-1}$, because $P(x_0^{n-1}) = 1$, $x_0 = 0$, $P(x_j) = 1$ and $x_j \neq m$. This means that $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$.
  - $j < n$ and $P(x_n) = 1$: $A_n(0..0(x_j + 1)x_{j+1}^{n-1}) = x_0^{n-1}$, because $P(x_1^{j-1}(x_j - 1)x_{j+1}^{n-1} = 1$, $x_0 = 0$ and $P(x_j + 1) = 0$. This means that $A_{n+1}(x) = A_n^{-1}(x_0^{n-1})x_n = f_n^{-1}(x_0^{n-1})x_n = f_{n+1}(x)$
- $x_j = m$ and $P(x_{j+1}) = 0$: We know $j < n$, because the algorithm is not defined on $0..0m$. Then $A_n(x) = 0..0x_j(x_{j+1} + 1)x_{j+2}^n$. We have a couple of cases:
  - $j + 1 = n$: x is of the form: $x = 0..0m(x_{j+1})$. $f_{n+1}(x) = 0..0m(x_{j+1} + 1)$, because $2|x_{j+1}$, so $A_{n+1}(x) = f_{n+1}(x)$.
  - $j + 1 < n$ and $P(x_n) = 0$: $A_n(0..0x_jx_{j+1}x_{j+2}^{n-1}) = 0..0x_j(x_{j+1} + 1)x_{j+2}^{n-1}$, because $P(x_1^{n-1}) = 1$, $x_0 = 0$, $x_j = m$ and $P(x_{j+1}) = 0$. This means that $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$.
  - $j + 1 < n$ and $P(x_n) = 1$: $A_n(0..0x_j(x_{j+1} + 1)x_{j+2}^{n-1}) = 0..0x_jx_{j+1}x_{j+2}^{n-1}$, because $P(x_1^j(x_{j+1} + 1)x_{j+2}^{n-1}) = 1$, $x_0 = 0$, $x_j = m$ and $P(x_{j+1} + 1) = 1$. This means that $A_{n+1}(x) = A_n^{-1}(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$.
- $x_j = m$ and $P(x_{j+1}) = 1$: Again, we know that $j < n$, because the algorithm is not defined on $0..0m$. Then $A_{n+1}(0..0x_jx_{j+1}x_{j+2}^n) = 0..0x_j(x_{j+1} - 1)x_{j+2}^n$. We have have a couple of different cases:
  - $j + 1 = n$: This case is actually impossible, since we know $P(x_{j+1}) = 1$, but we also know $P(x_1^n) = 1$, but the only two non-zero digits of x are m and x_{j+1}, which implies that $P(x_1^n) = 0$.
  - $j + 1 < n$ and $P(x_n) = 0$: $A_n(x_0^{n-1}) = 0..0x_j(x_{j+1} - 1)x_{j+2}^{n-1}$, because $P(x_1^{n-1}) = 1$, $x_0 = 0$, $x_j = m$ and $P(x_{j+1}) = 1$. This means that $A_{n+1}(x) = A_n(x_0^{n-1})x_n = f_n(x_0^{n-1})x_n = f_{n+1}(x)$.
  - $j + 1 < n$ and $P(x_n) = 1$: $A_n(0..0x_j(x_{j+1} - 1)x_{j+1}^{n-1}) = x_0^{n-1}$, because $P(x_1^{j-1}x_j(x_{j+1} - 1)x_{j+2}^{n-1}) = 1$, $x_0 = 0$, $x_j = m$ and $P(x_{j+1} - 1) = 0$. This means that $A_{n+1}(x) = A_n^{-1}(x_0^{n-1})x_n = f_n^{-1}(x_0^{n-1})x_n = f_{n+1}(x)$.

With that, we have checked for all possible cases that the algorithm coincides with the recurrence relations. □

Complexity analysis: We want to look at the time it takes to calculate for a given tuple of length n the tuple that follows. Determining the parity of the input tuple takes linear time, since we loop through the input. Furthermore, we loop through the tuple to determine the index of the first non-zero digit. This is also linear if implemented in the most naive way. The remaining operations do not depend on the size of the input. Therefore the complexity of the algorithm is $O(n)$.

2.2.4 Recursive definition for reflected Gray codes for an odd sized alphabet

The reflected Gray codes for an odd sized alphabet are very similar to the codes for an even sized alphabet. One major difference though, is that when the most significant digit is maximal, m, the remaining $(n-1)$-sized tuple is moving "forwards", because in this case m is even. In the previous case, the last tuple of size n was $0..0m$. In the odd-sized case, the last tuple of size n will be $m..mm$. This will have some implications for the recursive definition and the algorithm.

Let $\mathbb{A} = \{0, 1, .., m\}$ be an alphabet of odd size. Assume that the reflected Gray codes for sizes $k < n$ are known. Let $f_k : \mathbb{A}^k \setminus \{m..mm\} \to \mathbb{A}^k \setminus \{0..00\}$ be the function that takes a tuple of size k and returns the tuple that comes next in the reflected Gray code. These functions f_k are then inductively defined by the following rules:

For tuples of length 1:

$$\begin{aligned} f_1 : \mathbb{A} \setminus \{m\} &\to \mathbb{A} \setminus \{0\} \\ f_1(a) &= a + 1 \end{aligned}$$

And for f_{n+1}, given f_n:

$$\begin{aligned} f_{n+1}(m..mmx_n) &= m..mm(x_n + 1) && \text{if } 2|x_n \\ f_{n+1}(0..00x_n) &= 0..00(x_n + 1) && \text{if } 2 \nmid x_n \\ f_{n+1}(x) &= f_n(x_0^{n-1})x_n && \text{if } 2|x_n \\ f_{n+1}(x) &= f_n^{-1}(x_0^{n-1})x_n && \text{if } 2 \nmid x_n \end{aligned}$$

where $x_i^j := x_i x_{i+1} ... x_{j-1} x_j$ for a tuple $x = x_0 x_1 .. x_n$.

2.2.5 A memoryless iterative algorithm for reflected Gray codes with an odd sized alphabet

Just like the recursive definition, this algorithm is similar to the algorithm for an even sized alphabet. The difference, once again, stems from the fact that the last tuple is $m..mm$, instead of $0..0m$. Another difference is that, here the maximal digit, m, does not change the parity of the tuple, because m is even. This makes the algorithm simpler. Overall, the principle is still that we want to find out in which direction the digit at index 0 is moving.

This is what the algorithm does for an input tuple of length $n+1$, $x = x_0 x_1 ... x_n$:

- If $P(x_1^n) = 0$ and $x_0 \neq m$: return $(x_0 + 1)x_1^n$
- If $P(x_1^n) = 1$ and $x_0 \neq 0$: return $(x_0 - 1)x_1^n$
- If $P(x_1^n) = 0$ and $x_0 = m$:
  let $j = \min \{0 < j \leq n | x_j \neq m\}$

- if $P(x_j) = 0$: return $x_0^{j-1}(x_j + 1)x_{j+1}^n$
- if $P(x_j) = 1$: return $x_0^{j-1}(x_j - 1)x_{j+1}^n$

- If $P(x_1^n) = 1$ and $x_0 = 0$:
  let $j = \min\{0 < j \leq n | x_j \neq 0\}$
  - if $P(x_j) = 0$: return $0..0(x_j - 1)x_{j+1}^n$
  - if $P(x_j) = 1$: return $0..0(x_j + 1)x_{j+1}^n$

Correctness of algorithm: The proof of this algorithm is analogous to the proof of the algorithm in Section 2.2.3.

Complexity analysis: Determining the index of the first non-maximal digit can be done in linear time. For the remaining part , the algorithm acts the same in terms of complexity as the algorithm in Section 2.2.3, therefore the complexity of the entire algorithm is also $O(n)$.

2.2.6 Knuth's loopless algorithm

In this section we will look at an algorithm found in Knuth's The Art of Computer Programming 4a [2] (Algorithm H on page 300) using a concept originally used by Ehrlich [1] (Bitner, Ehrlich and Reingold CACM 19 (1976) 517-521). It is a very efficient algorithm for generating the general reflected Gray codes, whose speed, apart from initialization, does not depend on the length of the tuples, plus it does not need to memorize all the previous tuples. It does however require an extra piece of memory the same size as the input tuple. This piece of memory is there to keep track of which indices should be changed next. This means that if we have just the input tuple, the algorithm does not work in constant time, because we first have to calculate the correct values for this auxiliary memory.

The implementation in Knuth [2] is actually a more general version of the algorithm, since the alphabet can depend on the index: Let n be the size of the reflected Gray codes we are generating. For all $0 \leq j < n$, the digit on index j ranges from 0 inclusive to m_j exclusive. But that is not the point of this algorithm, since the algorithm in the previous sections can also be tweaked to work for tuples where the alphabet depends on the index. The point of this algorithm is to show that there is a technique to calculate the next element of the Gray code in constant time.

The algorithm is based on two principles: First of all, if we track the value of a single digit at index j, it is either moving from 0 to $m_j - 1$ or from $m_j - 1$ to 0. That is, its direction does not change until it reaches one of the extremes 0 or $m_j - 1$. Second of all, starting at $(0, .., 0)$, at the kth step, we change the digit at index j, where $j = \max\{0 \leq j < n : \prod_{i=0}^{j-1} m_i | k\}$. So every time the iteration is a multiple of $\prod_{i=0}^{j-1} m_i$, the digit at index j changes, except after the digit at index j just reached an extreme value 0 or $m_j - 1$. In Knuth's terminology, after reaching an extreme value, the coordinate at j is called *passive*. This means that the next time the iteration is a multiple of $\prod_{i=0}^{j-1} m_i$, a higher index digit gets changed instead of j, after which the coordinate j is active again.

The following section contains Knuth's algorithm as it is written in "The Art of Computer Programming 4a" ad verbum. The "Algorithm L" that is referred to is the same algorithm but for binary Gray codes, i.e., $\forall j : m_j = 2$. Note that in his notation the coordinate at index 0 is on the right.

Knuth's section literally: *(Loopless reflected mixed radix Gray generation).* This algorithm visits all n tuples $(a_{n-1}, .., a_0)$ such that $0 \leq a_j < m_j$ for $0 \leq j < n$, changing only one component by ± 1 at each step. It maintains an array of focus pointers $(f_n, .., f_0)$

to control the actions as in algorithm L, together with an array of directions $(o_{n-1}, .., o_0)$. We assume that each radix m_j is ≥ 2.

H1. [Initialize.] Set $a_j \leftarrow 0$, $f_j \leftarrow j$, and $o_j \leftarrow 1$, for $0 \leq j < n$; also set $f_n \leftarrow n$.

H2. [Visit.] Visit the n tuple $(a_{n-1}, .., a_1, a_0)$.

H3. [Choose j.] Set $j \leftarrow f_0$ and $f_0 \leftarrow 0$. (As in Algorithm L, j was the rightmost active coordinate; all elements to its right have now been reactivated.)

H4. [Change coordinate j] Terminate if $j = n$; otherwise set $a_j \leftarrow a_j + o_j$.

H5. [Reflect?] If $a_j = 0$ or $a_j = m_j - 1$, set $o_j \leftarrow -o_j$, $f_j \leftarrow f_{j+1}$, and $f_{j+1} \leftarrow j + 1$. (Coordinate j has thus become passive.) Return to H2.

3 Half growing cube

3.1 The problem

With the generalized Gray codes we were given an alphabet, $\mathbb{A} = \{0, .., m\}$, and our task was to order all tuples of size n in such a way that the difference between two consecutive tuples is minimal. When you have done this for size n, you can easily suffix the tuples with a 0 and add a new dimension and just continue generating Gray codes. This process of adding dimensions can be continued ad finitum, so the only restriction is actually the alphabet.

We now want to look at the situation where we restrict the dimensions and allow the alphabet to grow indefinitely: When we list all the tuples that can be made with alphabet $\mathbb{A}_m = \{0, .., m\}$, we want to extend this list elegantly so it contains all the tuples that can be made with alphabet $\mathbb{A}_{m+1} = \{0, .., m + 1\}$. Of course, here we also want two consecutive tuples to differ in only one position and we want the difference between the two digits on this position to be only 1.

Definition 3.1. A **Half growing cube code of dimension** d is a bijection $g : \mathbb{N}^d \to \mathbb{N}$, such that

$$g(x) - g(y) = 1 \implies K(x, y) = 1$$

and

$$g(x) \leq g(y) \implies M(x) \leq M(y),$$

where $K((x_1, ..x_d), (y_1, .., y_d)) = \sum_{i=1}^{d} |x_i - y_i|$ and $M((x_1, ..x_d)) = \max\{x_1, .., x_d\}$.

An application of this code, is to traverse polynomials with positive coefficients of a certain size, where we want to try the ones with minimal coefficients first and want to minimize the changes at each step.

3.2 Modeled as a Hamiltonian path through a growing hypercube

3.2.1 One dimension

This case is extremely straightforward. We simply start with 0 and at each step we add 1.

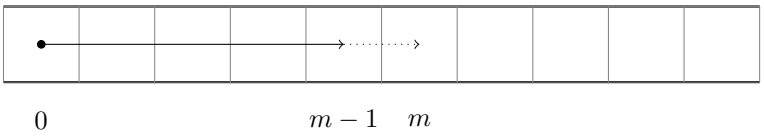

Figure 1: The one-dimensional route.

3.2.2 Two dimensions

This case is also very straightforward. We start at 00, and at this location we have two options, 10 and 01. If we go to 10, from here we could go to either 20 or 11. But we are currently just using alphabet $\mathbb{A}_1$, so we first want to visit all the tuples we can make with this alphabet. Thus we move to 11 and our route onwards is already decided: We constantly move along the edge given by: $E_m = \{ab \in \mathbb{A}_m^2 | a = m \text{ or } b = m\}$. Each time we finish such an edge, there is only direction we can move in. Therefore, there are exactly two different routes, either our first move is from 00 to 01 or from 00 to 10. Then for each new edge, we start this edge at $0m$ and end in $m0$ or vice versa.

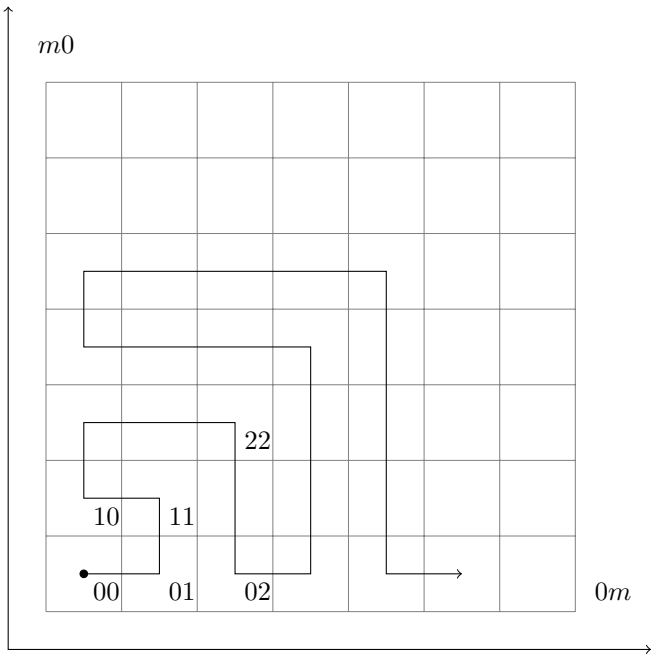

Figure 2: The two-dimensional route.

3.2.3 Three dimensions

This is where it gets interesting. In three dimensions, we have much more freedom of movement. After visiting all the tuples given by alphabet $\mathbb{A}_{m-1} = \{0, .., m-1\}$, we want to visit all the tuples in the shell $S_m = \mathbb{A}_m^3 \setminus \mathbb{A}_{m-1}^3 = \{abc \in \mathbb{A}_m^3 | a = m \text{ or } b = m \text{ or } c = m\}$. There are many different ways to traverse this shell, so our goal is simplicity and elegance.

Similarly to the reflected Gray codes, we want to use the solutions of a lower dimension to solve the problem. Let us look at the cube given by $\mathbb{A}_m$ and in particular its outer shell S_m. When we look at the slices we get by fixing the last coordinate, c, we see that whenever $0 \leq c < m$, this slice is equivalent to the 2-dimensional edge E_m, and when $c = m$, this slice is equivalent to the square $\mathbb{A}_m^2$. Of these two shapes, we already know how to traverse them. The idea of the algorithm is, when m is odd, the way we move through a new shell, is that we start at $00m$ and move through $S_m \cap (\mathbb{A}_m^2 \times \{m\})$, by using the path used for two dimensions and then we end up in $m0m$. From here we step to $m0(m-1)$. Next, for each $c = m-1, m-2, .., 0$, we move through $S_m \cap (\mathbb{A}_m^2 \times \{c\})$ by using the path we used for E_m, where we move from $0mc$ to $m0c$ or vice versa. For all $c \in \{1, .., m-1\}$, after traversing $E_m \times \{c\}$, we subtract 1 from the last coordinate and enter $E_m \times \{c-1\}$ This way we visit all the coordinates of the shell S_m and end up in $0m0$. When m is even, we use the same path, but we go backwards, so we start from $0m0$ and end up in $00m$.

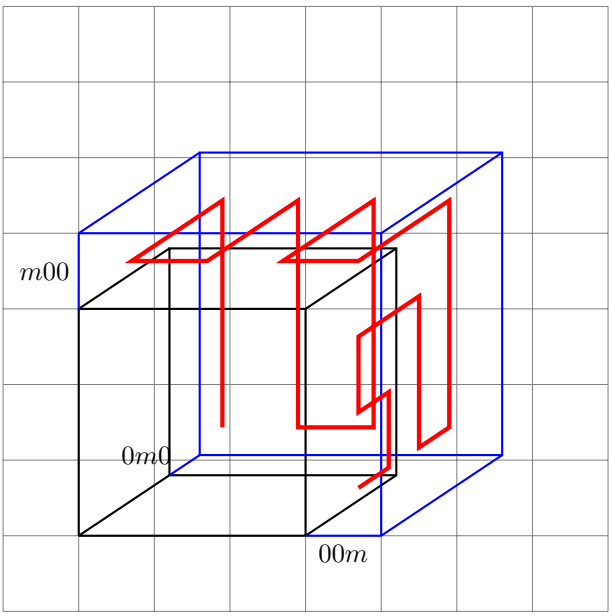

Figure 3: The red line depicts the route through S_m in the three-dimensional case

3.2.4 Higher dimensions

For d dimensions, we generalize the idea that we used for three dimensions: We assume that we have a path through $\mathbb{A}_m^{d-1}$ that starts at $0..00$ and ends at $0..0m0$ if m is odd and ends at $0..0m$ if m is even. We also assume that we have a path through the $d-1$-dimensional edge $E_m = \{x_1..x_{d-1} \in \mathbb{A}_m^{d-1} | \max\{x_1, .., x_{d-1}\} = m\}$ that moves from $0..0m$ to $0..m0$ if m is odd and from $0..m0$ to $0..0m$ if m is even.

The d-dimensional shell is equal to the union of a couple of disjoint sets: $S_m = \{x_1..x_d \in \mathbb{A}_m^d | \max\{x_1, .., x_d\} = m\} = E_m \times \{0\} \cup E_m \times \{1\} \cup ... \cup E_m \times \{m-1\} \cup \mathbb{A}_m^{d-1} \times \{m\}$. When m is odd, we start at $0..00m$. We use the $(d-1)$-dimensional route for traversing $\mathbb{A}_m^{d-1} \times \{m\}$, where we move from $0..00m$ to $0..0m0m$ (Since the $(d-1)$-dimensional route goes from $0..00$ to $0..0m0$ when m is odd). At this point we move backwards in the most significant dimension: from $0..0m0m$ to $0..m0(m-1)$. Next, we use the $(d-1)$-dimensional edge route backwards for traversing $E_m \times \{m-1\}$, where we move from $0..0m0(m-1)$ to $0..0m(m-1)$. Again we move from $0..0m(m-1)$ to $0..0m(m-2)$. Here we use the $(d-1)$-dimensional edge route forwards to get from $0..0m(m-2)$ to $0..m0(m-2)$ and then move to $0..m0(m-3)$. From here on out, we repeat these last steps until we reach $0..0m0$ and have visited each location of the shell S_m.

When m is even, we move from $0..0m0$ to $0..00m$ using a route that is pretty much the opposite of the route when m is odd.

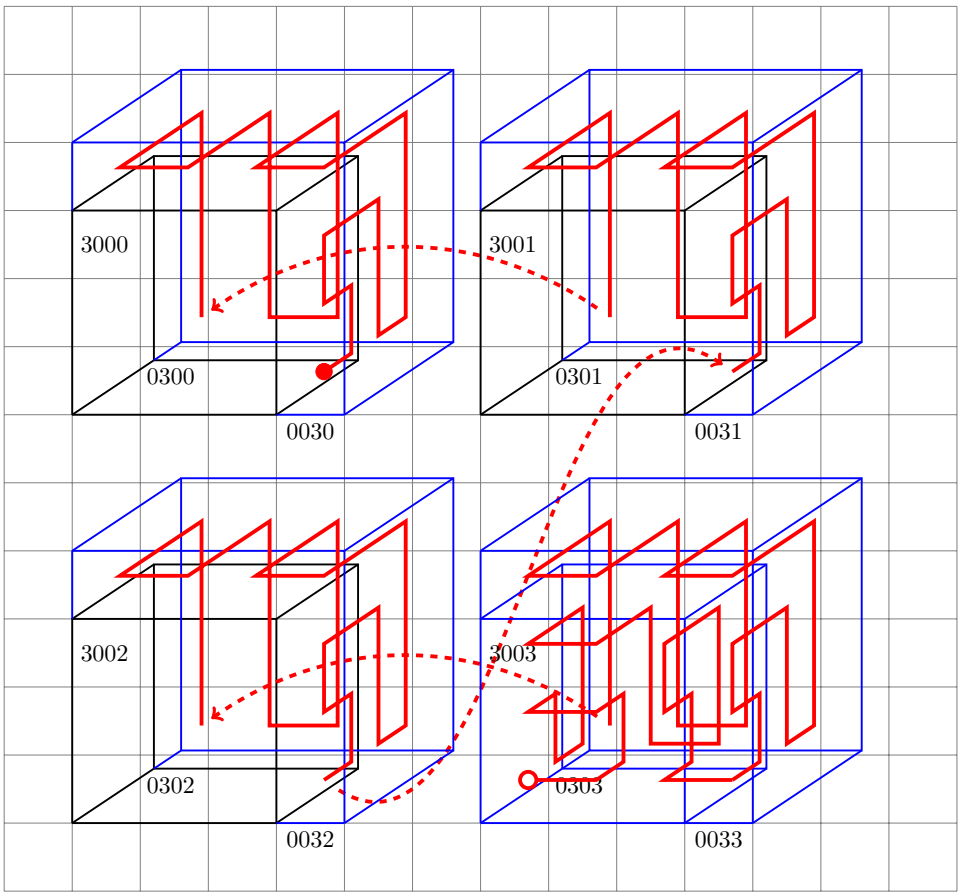

Figure 4: The red line depicts the route through S_3 in the four-dimensional case.

3.3 Recursive definition half growing cube algorithm

As previously stated, the goal is to arrange, for a given dimension d, all elements of $\mathbb{N}^d$, in such a way that for two following elements $x = x_1..x_d$ and $y = y_1..y_d$, it holds that $\sum_{i=1}^{d} |x_i - y_i| = 1$. Furthermore, for every m, the first $(m+1)^d$ elements should only contain digits from $\{0, 1, .., m-1, m\}$. One way of modeling this, is as a bijective function HG from $\mathbb{N}^d$ to $\mathbb{N}^d \setminus \{(0, .., 0)\}$, where the arrangement we get is given by starting at element $(0, .., 0)$ and repeatedly applying this function HG.

This function HG, corresponding to the Hamiltonian paths from Section 3.2, and its inverse HG^{-1} are defined as follows:

When the dimension is $d = 1$:

$$\begin{aligned} \mathrm{HG}_1 : \mathbb{N} &\to \mathbb{N} \setminus \{0\} \\ a &\mapsto a + 1 \end{aligned} \qquad\qquad \begin{aligned} \mathrm{HG}_1^{-1} : \mathbb{N} \setminus \{0\} &\to \mathbb{N} \\ a &\mapsto a - 1 \end{aligned}$$

When the dimension is $d = 2$:

Let m be the maximal digit of the input.

$\mathrm{HG}_2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N} \setminus \{(0,0)\}$
When m is odd:

$$\begin{aligned}
&(m,0) \mapsto (m+1,0) \\
&(m,a) \mapsto (m,a-1) \quad \text{with } a \neq 0 \\
&(a,m) \mapsto (a+1,m) \quad \text{with } a \neq m
\end{aligned}$$

When m is even:

$$\begin{aligned}
&(0,m) \mapsto (0,m+1) \\
&(m,a) \mapsto (m,a+1) \quad \text{with } a \neq m \\
&(a,m) \mapsto (a-1,m) \quad \text{with } a \neq 0
\end{aligned}$$

$\mathrm{HG}_2^{-1} : \mathbb{N} \times \mathbb{N} \setminus \{(0,0)\} \to \mathbb{N} \times \mathbb{N}$
When m is odd:

$$\begin{aligned}
&(0,m) \mapsto (0,m-1) \\
&(m,a) \mapsto (m,a+1) \quad \text{with } a \neq m \\
&(a,m) \mapsto (a-1,m) \quad \text{with } a \neq 0
\end{aligned}$$

When m is even:

$$\begin{aligned}
&(m,0) \mapsto (m-1,0) \\
&(m,a) \mapsto (m,a-1) \quad \text{with } a \neq 0 \\
&(a,m) \mapsto (a+1,m) \quad \text{with } a \neq m
\end{aligned}$$

When the dimension is $d > 2$:
As defined before, if $x = (x_1, x_2, .., x_{d-1}, x_d)$ then $x_1^{d-1} = (x_1, x_2, .., x_{d-2}, x_{d-1})$. And m is again defined as the maximal digit of the input, i.e., if our input is $(x_1, x_2, .., x_d)$, $m = \max\{x_1, x_2, .., x_d\}$.
$\mathrm{HG}_d : \mathbb{N}^d \to \mathbb{N}^d \setminus \{(0, .., 0)\}$
We now split into two cases:
m is odd:

$$\begin{aligned}
&(0,..,0,m,0) &&\mapsto && (0,..,0,m+1,0) && \\
&(0,..,0,m,a) &&\mapsto && (0,..,0,m,a-1) && \text{with } 2|a \\
&(0,..,0,m,0,a) &&\mapsto && (0,..,0,m,0,a-1) && \text{with } 2 \nmid a \\
&x_1^{d-1}a &&\mapsto && \mathrm{HG}_{d-1}^{-1}(x_1^{d-1})a && \text{with } 2|a \\
&x_1^{d-1}a &&\mapsto && \mathrm{HG}_{d-1}(x_1^{d-1})a && \text{with } 2 \nmid a
\end{aligned}$$

m is even:

$$\begin{aligned}
&(0,..,0,m) &&\mapsto && (0,..,0,m+1) && \\
&(0,..,0,m,a) &&\mapsto && (0,..,0,m,a+1) && \text{with } 2 \nmid a \\
&(0,..,0,m,0,a) &&\mapsto && (0,..,0,m,0,a+1) && \text{with } 2|a \text{ and } a \neq m \\
&x_1^{d-1}a &&\mapsto && \mathrm{HG}_{d-1}^{-1}(x_1^{d-1})a && \text{with } 2|a \\
&x_1^{d-1}a &&\mapsto && \mathrm{HG}_{d-1}(x_1^{d-1})a && \text{with } 2 \nmid a
\end{aligned}$$

$\mathrm{HG}_d^{-1} : \mathbb{N}^d \setminus \{(0, .., 0)\} \to \mathbb{N}^d \setminus \{(0, .., 0)\}$
We again split into two cases:
m is odd:

$$\begin{aligned}
&(0,..,0,m) &&\mapsto && (0,..,0,m-1) && \\
&(0,..,0,m,a) &&\mapsto && (0,..,0,m,a+1) && \text{with } 2 \nmid a \text{ and } a \neq m \\
&(0,..,0,m,0,a) &&\mapsto && (0,..,0,m,0,a+1) && \text{with } 2|a \\
&x_1^{d-1}a &&\mapsto && \mathrm{HG}_{d-1}^{-1}(x_1^{d-1})a && \text{with } 2 \nmid a \\
&x_1^{d-1}a &&\mapsto && \mathrm{HG}_{d-1}(x_1^{d-1})a && \text{with } 2|a
\end{aligned}$$

m is even:

$$\begin{array}{lcll}
(0,..,0,m,0) & \mapsto & (0,..,0,m-1,0) & \\
(0,..,0,m,a) & \mapsto & (0,..,0,m,a-1) & \text{with } 2|a \\
(0,..,0,m,0,a) & \mapsto & (0,..,0,m,0,a-1) & \text{with } 2 \nmid a \\
x_1^{d-1}a & \mapsto & \mathrm{HG}_{d-1}^{-1}(x_1^{d-1})a & \text{with } 2 \nmid a \\
x_1^{d-1}a & \mapsto & \mathrm{HG}_{d-1}(x_1^{d-1})a & \text{with } 2|a
\end{array}$$

Proof correctness of algorithm: The function HG_d has to have a couple of properties in order to be adequate. Firstly, we do not want to visit any element of $\mathbb{N}^d$ more than once. We can show that no element will be visited more than once by proving that the function HG_d is bijective and the bijectivity we will prove by showing that the HG_d^{-1} is indeed the inverse of HG_d. Secondly, we want to show that at every step, only one digit of the input is changed, and this digit is changed by plus or minus 1. Thirdly, we want to show that all elements of $\{0,1,..,m\}^d$ are visited before the elements of $\{0,1,..,m,m+1\}^d \setminus \{0,1,..,m\}^d$. This will be done by showing that for every m, the maximal digit will only be raised to $m+1$ once and never lowered. And lastly, we want to show that we do indeed visit each element by repeatedly applying HG_d to the starting element $(0,..,0)$. This will be proven after the first three properties are proven.

We repeat some auxiliary functions: $M((x_1,x_2,..,x_d)) = \max\{x_1,x_2,..,x_d\}$ and $K_d((x_1,..x_d),(y_1,..,y_d)) = \sum_{i=1}^{d} |x_i - y_i|$. And we will call the HG_d^{-1} function as defined at the start of Section 3.3, HG_d^*, since we have yet to prove that it is indeed the inverse.

Explicitly put, we will prove that $\forall d \in \mathbb{Z}_{>0}$:

(1) $\mathrm{HG}_d^* \circ \mathrm{HG}_d = \mathrm{id}_{\mathbb{N}^d}$.

(2) $\mathrm{HG}_d \circ \mathrm{HG}_d^* = \mathrm{id}_{\mathbb{N}^d \setminus \{(0,..,0)\}}$.

(3) $\forall x \in \mathbb{N}^d \ K_d(x, \mathrm{HG}_d(x)) = 1$.

(4) $\forall x \in \mathbb{N}^d \setminus V_d$: $M(\mathrm{HG}_d(x)) = M(x)$ and $\forall x \in V_d \ M(\mathrm{HG}_d(x)) = M(x)+1$, where, for $d > 1$, $V_d = \{(0,..,0,m,0) \in \mathbb{N}^d : 2 \nmid m\} \cup \{(0,..,0,m) \in \mathbb{N}^d : 2|m\}$ and $V_1 = \mathbb{N}$.

Induction basis:

HG_1:

(1): Take $a \in \mathbb{N}$. $\mathrm{HG}_1^*(\mathrm{HG}_1(a)) = \mathrm{HG}_1^*(a+1) = a+1-1 = a$.

(2): Take $a \in \mathbb{N} \setminus 0$. $\mathrm{HG}_1(\mathrm{HG}_1^*(a)) = \mathrm{HG}_1(a-1) = a-1+1 = a$.

(3): Take $a \in \mathbb{N}$. $K_1(a, \mathrm{HG}_1(a)) = K_1(a, a+1) = |a+1-a| = 1$.

(4): $V_1 = \mathbb{N}$. Take $a \in V_1$. $M(\mathrm{HG}_1(a)) = M(a+1) = a+1 = M(a)+1$.

HG_2:

(1): $\mathrm{HG}_2^* \circ \mathrm{HG}_2$:

Let m be the maximal digit of the input.

When m is odd:

$$\begin{cases}
(m,0) \mapsto \mathrm{HG}_2^*((m+1,0)) = (m,0) & \\
(m,a) \mapsto \mathrm{HG}_2^*((m,a-1)) = (m,a) & \text{with } a \neq 0 \\
(a,m) \mapsto \mathrm{HG}_2^*((a+1,m)) = (a,m) & \text{with } a \neq m
\end{cases}$$

When m is even:

$$\begin{cases}
(0,m) \mapsto \mathrm{HG}_2^*((0,m+1)) = (0,m) & \\
(m,a) \mapsto \mathrm{HG}_2^*((m,a+1)) = (m,a) & \text{with } a \neq m \\
(a,m) \mapsto \mathrm{HG}_2^*((a-1,m)) = (a,m) & \text{with } a \neq 0
\end{cases}$$

(2): $\mathrm{HG}_2 \circ \mathrm{HG}_2^*$:

Let m be the maximal digit of the input.

When m is odd:

$$\begin{cases} (0,m) \mapsto \mathrm{HG}_2((0,m-1)) = (0,m) \\ (m,a) \mapsto \mathrm{HG}_2((m,a+1)) = (m,a) & \text{with } a \neq m \\ (a,m) \mapsto \mathrm{HG}_2((a-1,m)) = (a,m) & \text{with } a \neq 0 \end{cases}$$

When m is even:

$$\begin{cases} (m,0) \mapsto \mathrm{HG}_2((m-1,0)) = (m,0) \\ (m,a) \mapsto \mathrm{HG}_2((m,a-1)) = (m,a) & \text{with } a \neq 0 \\ (a,m) \mapsto \mathrm{HG}_2((a+1,m)) = (a,m) & \text{with } a \neq m \end{cases}$$

(3): This is clear from the definition.

(4): From the definition we can see that the only times that the maximal digit of the input changes is when m is odd and the input is $(m,0)$ or when m is even and the input is $(0,m)$. This coincides exactly with V_2.

Induction hypothesis: Take $d > 2$, we assume that HG_{d-1} adheres to the four conditions (1), (2), (3), and (4). Conditions (1) and (2) imply bijectivity. From bijectivity and condition (3), it is also true that $\forall x \in \mathbb{N}^{d-1} \setminus \{(0,..,0)\}$: $K_{d-1}(x, \mathrm{HG}^*_{d-1}(x)) = 1$. Also, from bijectivity and condition (4) we get that $\forall x \in \mathbb{N}^{d-1} \setminus V^*_{d-1}$: $M(\mathrm{HG}^*_{d-1}(x)) = M(x)$ and $\forall x \in V^*_{d-1} \setminus \{(0,..,0)\}$: $M(\mathrm{HG}^*_{d-1}(x)) = M(x)-1$, where $V^*_{d-1} = \{(0,..0,m,0) \in \mathbb{N}^{d-1} : 2|m\} \cup \{(0,..,0,m) \in \mathbb{N}^{d-1} : 2 \nmid m\}$.

Induction step: We will look at the functions $\mathrm{HG}_d : \mathbb{N}^d \to \mathbb{N}^d \setminus \{(0,..,0)\}$ and $\mathrm{HG}^*_d : \mathbb{N}^d \setminus \{(0,..,0)\} \to \mathbb{N}^d$.

We will begin by proving conditions (1), (3) and (4), and we will prove condition (2) afterwards.

First we look at the elements for which the maximal digit is odd: Let $x \in \mathbb{N}^d$ with $2 \nmid M(x)$ and $m := M(x)$. There are 5 cases:

- x is of the form $(0,..,0,m,0)$: Then $\mathrm{HG}_d(x) = (0,..,0,m+1,0)$. We see that $K_d(x, \mathrm{HG}_d(x)) = 1$ and $M(\mathrm{HG}_d(x)) = M(x)+1$, which is correct, since $x \in V_d$. $\mathrm{HG}^*_d(\mathrm{HG}_d(x)) = \mathrm{HG}^*_d((0,..,0,m+1,0)) = (0,..,0,m,0) = x$, since $2|M(\mathrm{HG}_d(x))$.

- x is of the form $(0,..,0,m,a)$ with $2|a$ and $a \neq 0$: Then $\mathrm{HG}_d(x) = (0,..,0,m,a-1)$. We see that $K_d(x, \mathrm{HG}_d(x)) = 1$ and $M(\mathrm{HG}_d(x)) = M(x)$. $\mathrm{HG}^*_d(\mathrm{HG}_d(x)) = \mathrm{HG}^*_d((0,..,0,m,a-1)) = (0,..,0,m,a) = x$, since $2 \nmid (a-1)$ and $a-1 \neq m$.

- x is of the form $(0,..,0,m,0,a)$ with $2 \nmid a$: Then $\mathrm{HG}_d(x) = (0,..,0,m,0,a-1)$. We see that $K_d(x, \mathrm{HG}_d(x)) = 1$ and $M(\mathrm{HG}_d(x)) = M(x)$. $\mathrm{HG}^*_d(\mathrm{HG}_d(x)) = \mathrm{HG}^*_d((0,..,0,m,0,a-1)) = (0,..,0,m,0,a) = x$, since $M(\mathrm{HG}_d(x))$ is odd and $2|(a-1)$.

- x is of the form $x_1^{d-1}a$ with $2|a$ and $x_1^{d-1} \neq (0,..,0,m)$: In that case $\mathrm{HG}_d(x) = \mathrm{HG}^*_{d-1}(x_1^{d-1})a$. $x_1^{d-1} \neq (0,..,0,m)$, so by condition (4) of the IH, $M(\mathrm{HG}^*_{d-1}(x_1^{d-1})) = M(x_1^{d-1})$. Since $a < m$, we know $M(x) = M(x_1^{d-1}) = M(\mathrm{HG}^*_{d-1}(x_1^{d-1})) = M(\mathrm{HG}^*_{d-1}(x_1^{d-1})a) = M(\mathrm{HG}_d(x))$. Also, $K(x, \mathrm{HG}_d(x)) = K(x_1^{d-1}, \mathrm{HG}^*_{d-1}(x_1^{d-1})) = 1$ by (3) of the IH. Furthermore, we get $\mathrm{HG}^*_d(\mathrm{HG}_d(x)) = \mathrm{HG}^*_d(\mathrm{HG}^*_{d-1}(x_1^{d-1})a) = \mathrm{HG}_{d-1}(\mathrm{HG}^*_{d-1}(x_1^{d-1}))a = x_1^{d-1}a = x$, where we use the fact that $\mathrm{HG}^*_{d-1}(x_1^{d-1}) \neq (0,..,0,m,0)$ and (2) of the IH. We know $\mathrm{HG}^*_{d-1}(x_1^{d-1}) \neq (0,..,0,m,0)$, because if it were the case then $x_1^{d-1} = \mathrm{HG}_{d-1}((0,..,0,m,0)) = (0,..,0,m+1,0)$ and then $M(x) \neq m = M(x)$.

- x is of the form $x_1^{d-1}a$ with $2 \nmid a$ and $x_1^{d-1} \neq (0,..,0,m,0)$ and $a \neq m$: Then $\mathrm{HG}_d(x) = \mathrm{HG}_{d-1}(x_1^{d-1})a$. $a < m$, so $M(x_1^{d-1}) = m$. $x_1^{d-1} \neq (0,..,0,m,0)$ and

$M(x_1^{d-1}) = m$, so $M(\mathrm{HG}_{d-1}(x_1^{d-1})) = M(x_1^{d-1}) = m$, (4) of IH. Then $M(\mathrm{HG}_d(x))$ $= \max\{M(\mathrm{HG}_{d-1}(x_1^{d-1})), a\} = \max\{M(x_1^{d-1}), a\} = M(x)$. Also, $K(x, \mathrm{HG}_d(x)) = K(x_1^{d-1}, \mathrm{HG}_{d-1}(x_1^{d-1})) = 1$ by (3) of the IH. $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*(\mathrm{HG}_{d-1}(x_1^{d-1})a)$ $= \mathrm{HG}_{d-1}^*(\mathrm{HG}_{d-1}(x_1^{d-1}))a = x_1^{d-1}a = x$, where we use that $\mathrm{HG}_{d-1}(x_1^{d-1}) \neq (0, .., 0, m)$ and (1) of the IH. $\mathrm{HG}_{d-1}(x_1^{d-1})$ can not be $(0, .., 0, m)$, because if it were, then $x_1^{d-1} = \mathrm{HG}_{d-1}^*(0, .., 0, m) = (0, .., 0, m-1)$ and then $M(x) = M(x_1^{d-1}a) = m - 1$, but it is defined as m.

- x is of the form $x_1^{d-1}m$ with $x_1^{d-1} \neq (0, .., 0, m, 0)$: Then $\mathrm{HG}_d(x) = \mathrm{HG}_{d-1}(x_1^{d-1})m$. $x_1^{d-1} \neq (0, .., 0, m, 0)$ and $M(x_1^{d-1}) \leq m$, so $M(HG_{d-1}(x_1^{d-1})) \leq m$, by (4) of IH. Then $M(\mathrm{HG}_d(x)) = \max\{M(\mathrm{HG}_{d-1}(x_1^{d-1})), m\} = m = M(x)$. Also, $K(x, \mathrm{HG}_d(x)) = K(x_1^{d-1}, \mathrm{HG}_{d-1}(x_1^{d-1})) = 1$ by (3) of the IH. $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*(\mathrm{HG}_{d-1}(x_1^{d-1})m) = \mathrm{HG}_{d-1}^*(\mathrm{HG}_{d-1}(x_1^{d-1}))m = x_1^{d-1}m = x$, where we use that $\mathrm{HG}_{d-1}(x_1^{d-1}) \neq (0, .., 0)$ and (1) of the IH. $\mathrm{HG}_{d-1}(x_1^{d-1})$ can not be $(0, .., 0)$, because $\mathrm{Im}(\mathrm{HG}_{d-1}) = \mathbb{N}^{d-1} \setminus \{(0, .., 0)\}$.

Now we look at the elements for which the max digit is even: Let $x \in \mathbb{N}^d$ with $2|M(x)$ and $m = M(x)$. We split into 6 cases:

- x is of the form $(0, .., 0, m)$: Then $\mathrm{HG}_d(x) = (0, .., 0, m+1)$. We see that $K_d(x, \mathrm{HG}_d(x)) = 1$ and $M(\mathrm{HG}_d(x)) = M(x) + 1$, which is correct, since $x \in V_d$. $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*((0, .., 0, m+1)) = (0, .., 0, m) = x$, since $2|M(\mathrm{HG}_d(x))$.

- x is of the form $(0, .., 0, m, a)$ with $2 \nmid a$: Then $\mathrm{HG}_d(x) = (0, .., 0, m, a+1)$. We see that $K_d(x, \mathrm{HG}_d(x)) = 1$ and $M(\mathrm{HG}_d(x)) = M(x)$, because $a < m$ since a is odd and $a \leq m$. $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*((0, .., 0, m, a+1)) = (0, .., 0, m, a) = x$, since $2|(a+1)$.

- x is of the form $(0, .., 0, m, 0, a)$ with $2|a$ and $a \neq m$: $\mathrm{HG}_d(x) = (0, .., 0, m, 0, a+1)$. We see that $K_d(x, \mathrm{HG}_d(x)) = 1$ and $M(\mathrm{HG}_d(x)) = M(x)$. $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*((0, .., 0, m, 0, a+1)) = (0, .., 0, m, 0, a) = x$, since $M(\mathrm{HG}_d(x))$ is even and $2 \nmid (a-1)$.

- x is of the form $x_1^{d-1}a$ with $2|a$ and $x_1^{d-1} \neq (0, .., 0, m, 0)$ and $a \neq m$: Then $\mathrm{HG}_d(x) = \mathrm{HG}_{d-1}^*(x_1^{d-1})a$. $x_1^{d-1} \neq (0, .., 0, m, 0)$, so by condition (4) of the IH, $M(\mathrm{HG}_{d-1}^*(x_1^{d-1})) = M(x_1^{d-1})$, so $M(x) = M(\mathrm{HG}_d(x))$. Also, $K(x, \mathrm{HG}_d(x)) = K(x_1^{d-1}, \mathrm{HG}_{d-1}^*(x_1^{d-1})) = 1$ by (3) of the IH. Furthermore, $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*(\mathrm{HG}_{d-1}^*(x_1^{d-1})a) = \mathrm{HG}_{d-1}(\mathrm{HG}_{d-1}^*(x_1^{d-1}))a = x_1^{d-1}a = x$, where we use that $\mathrm{HG}_{d-1}^*(x_1^{d-1}) \neq (0, .., 0, m)$ and (2) of the IH.

- x is of the form $x_1^{d-1}m$ with $x_1^{d-1} \neq (0, .., 0)$: Then $\mathrm{HG}_d(x) = \mathrm{HG}_{d-1}^*(x_1^{d-1})m$. $M(x_1^{d-1}) \leq m$, so $M(\mathrm{HG}_{d-1}^*(x_1^{d-1})) \leq m$, so $M(\mathrm{HG}_d(x)) = m$ by (4) of the IH and the bijectivity of HG_{d-1}. Also, $K(x, \mathrm{HG}_d(x)) = K(x_1^{d-1}, \mathrm{HG}_{d-1}^*(x_1^{d-1})) = 1$ by (3) of the IH. $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*(\mathrm{HG}_{d-1}^*(x_1^{d-1})m) = \mathrm{HG}_{d-1}(\mathrm{HG}_{d-1}^*(x_1^{d-1}))m = x_1^{d-1}a = x$, where we use that $\mathrm{HG}_{d-1}^*(x_1^{d-1}) \neq (0, .., 0, m)$ and (2) of the IH.

- x is of the form $x_1^{d-1}a$ with $2 \nmid a$ and $x_1^{d-1} \neq (0, .., 0, m)$: Then $\mathrm{HG}_d(x) = \mathrm{HG}_{d-1}(x_1^{d-1})a$. Since, $x_1^{d-1} \neq (0, .., 0, m, 0)$, $M(x_1^{d-1}) = M(\mathrm{HG}_{d-1}(x_1^{d-1}))$, (4) of IH, so $M(x) = M(\mathrm{HG}_d(x))$. Also, $K(x, \mathrm{HG}_d(x)) = K(x_1^{d-1}, \mathrm{HG}_{d-1}(x_1^{d-1})) = 1$ by (3) of the IH. $\mathrm{HG}_d^*(\mathrm{HG}_d(x)) = \mathrm{HG}_d^*(\mathrm{HG}_{d-1}(x_1^{d-1})a) = \mathrm{HG}_{d-1}^*(\mathrm{HG}_{d-1}(x_1^{d-1}))a = x_1^{d-1}a = x$, where we use that $\mathrm{HG}_{d-1}(x_1^{d-1}) \neq (0, .., 0, m, 0)$ and (1) of the IH.

We will now prove condition (2):

Again we first look at the situation where the max digit is odd: Let $x \in \mathbb{N}^d \setminus \{(0, .., 0)\}$ with $2 \nmid M(x)$ and $m = M(x)$. 6 cases:

- x is of the form $(0,..,0,m)$: Then $\mathrm{HG}_d^*(x) = (0,..,0,m-1)$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d((0,..,0,m-1)) = (0,..,0,m) = x$, since $2|M(\mathrm{HG}_d^*(x))$.

- x is of the form $(0,..0,m,a)$ with $2 \nmid a$ and $a \neq m$: Then $\mathrm{HG}_d^*(x) = (0..0,m,a+1)$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d((0,..,0,m,a+1)) = (0,..,0,m,a) = x$, since $a+1 \neq 0$ and $2|(a+1)$.

- x is of the form $(0,..,0,m,0,a)$ with $2|a$: Then $\mathrm{HG}_d^*(x) = (0,..,0,m,0,a+1)$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d((0,..,0,m,0,a+1)) = (0,..,0,m,0,a) = x$, since $2 \nmid (a+1)$.

- x is of the form $x_1^{d-1}a$ with $2 \nmid a$ and $x_1^{d-1} \neq (0,..,0,m)$. Then $\mathrm{HG}_d^*(x) = \mathrm{HG}_{d-1}^*(x_1^{d-1})a$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d(\mathrm{HG}_{d-1}^*(x_1^{d-1})a) = \mathrm{HG}_{d-1}(\mathrm{HG}_{d-1}^*(x_1^{d-1}))a = x_1^{d-1}a = x$, since $M(\mathrm{HG}_{d-1}^*(x_1^{d-1})) = M(x_1^{d-1})$, because $x_1^{d-1} \neq (0,..,0,m)$ and the IH, so $M(\mathrm{HG}_d^*(x)) = M(x)$ and $2 \nmid M(\mathrm{HG}_d^*(x))$ and also $\mathrm{HG}_{d-1}^*(x_1^{d-1}) \neq (0,..,0,m,0)$, because if it were then $x_1^{d-1} = \mathrm{HG}_{d-1}((0,..,0,m,0)) = (0,..,0,m+1,0)$ (IH) and then $M(x) > m$.

- x is of the form $(0,..,0,m,m)$: Then $\mathrm{HG}_d^*(x) = (0,..,0,m-1,m)$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_{d-1}((0,..,0,m-1))m = (0,..,0,m,m) = x$, since $2|(m-1)$.

- x is of the form $x_1^{d-1}a$ with $2|a$ and $x_1^{d-1} \neq (0,..,0,m,0)$. Then $\mathrm{HG}_d^*(x) = \mathrm{HG}_{d-1}(x_1^{d-1})a$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d(\mathrm{HG}_{d-1}(x_1^{d-1})a) = \mathrm{HG}_{d-1}^*(\mathrm{HG}_{d-1}(x_1^{d-1}))a = x_1^{d-1}a = x$, since $2|a$, $2 \nmid \mathrm{HG}_d^*(x)$ and $\mathrm{HG}_{d-1}(x_1^{d-1}) \neq (0,..,0,m)$.

Now look at the situation where the max digit is even: Let $x \in \mathbb{N}^d \setminus \{(0,..,0)\}$ with $2|M(x)$ and $m = M(x)$. 5 cases:

- x is of the form $(0,..,0,m,0)$: Then $\mathrm{HG}_d^*(x) = (0,..,0,m-1,0)$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d((0,..,0,m-1,0)) = (0,..,0,m,0) = x$, since $2 \nmid (m-1)$.

- x is of the form $(0,..,0,m,a)$ with $2|a$: In this case $\mathrm{HG}_d^*(x) = (0,..,0,m,a-1)$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d((0,..,0,m,a-1)) = (0,..,0,m,a) = x$, since $2 \nmid (a-1)$.

- x is of the form $(0,..,0,m,0,a)$ with $2 \nmid a$: Then $\mathrm{HG}_d^*(x) = (0,..,0,m,0,a-1)$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d((0,..,0,m,0,a-1)) = (0,..,0,m,0,a) = x$, since $2|(a-1)$.

- x is of the form $(x_1^{d-1}a)$ with $2 \nmid a$ and $x_1^{d-1} \neq (0,..,0,m,0)$: Then $\mathrm{HG}_d^*(x) = \mathrm{HG}_{d-1}^*(x_1^{d-1})a$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d(\mathrm{HG}_{d-1}^*(x_1^{d-1})a) = \mathrm{HG}_{d-1}(x_1^{d-1})a = x_1^{d-1}a = x$, since $2 \nmid a$ and $2|M(\mathrm{HG}_d^*(x))$, because $M(x_1^{d-1}) = m$ and $M(\mathrm{HG}_{d-1}^*(x_1^{d-1})) = M(x_1^{d-1})$, because $x_1^{d-1} \notin V_{d-1}^*$ and $a \leq m$, and also $\mathrm{HG}_{d-1}^*(x_1^{d-1}) \neq (0,..,0,m)$, because if $\mathrm{HG}_{d-1}^*(x_1^{d-1}) = (0,..,0,m)$, then $x_1^{d-1} = (0,..,0.m+1)$ and then $M(x) \neq m$.

- x is of the form $(x_1^{d-1}a)$ with $2|a$: Then $\mathrm{HG}_d^*(x) = \mathrm{HG}_{d-1}(x_1^{d-1})a$. $\mathrm{HG}_d(\mathrm{HG}_d^*(x)) = \mathrm{HG}_d(\mathrm{HG}_{d-1}(x_1^{d-1})a) = \mathrm{HG}_{d-1}^*(\mathrm{HG}_{d-1}(x_1^{d-1}))a = x_1^{d-1}a = x$, since $2|a$ and $2|\mathrm{HG}_d^*(x)$ and $\mathrm{HG}_{d-1}(x_1^{d-1}) \neq (0,..,0,m,0)$.

Now we have proven that the four conditions hold for the function HG_d. What remains is to prove that we will indeed visit all coordinates by starting at $(0,..,0)$ and repeatedly applying function HG_d. We will show that if we start at $(0,..,0)$ and repeatedly apply HG_d, if $2|m$, after $(m+1)^d - 1$ steps, we will reach $(0,..,0,m)$ and if $2 \nmid m$ after $(m+1)^d - 1$ steps we will reach $(0,..,0,m,0)$. From conditions (1) and (2), which imply bijectivity, we can conclude that the set of elements visited, S_m, is of size $|S_m| = (m+1)^d$ and from condition (4) we can conclude that $S_m \subset \{0,1,..,m\}^d$. $|S_m| = |\{0,1,..,m\}^d|$, so $S_m = \{0,1,..,m\}^d$.

Induction basis: HG_1: Starting at 0 after $(m+1)^1 - 1 = m$ steps we reach m. This is what we want, because in the 1-dimensional case $(0,..,0,m)$ and $(0,..,0,m,0)$ both translate to m.

HG_2: We will use induction over m to prove it for dimension $d = 2$.

IB: $m = 0$: It takes $(0+1)^2 - 1 = 1 - 1 = 0$ steps to reach $(0,0)$ from $(0,0)$.

IH: Take $m > 0$. Assume that if $2|(m-1)$, then after $m^2 - 1$ steps we reached $(0, m-1)$ and if $2 \nmid (m-1)$, after $m^2 - 1$ steps we reached $(m-1, 0)$.

IS: If $2|m$: Then after $m^2 - 1$ steps we reached $(m-1, 0)$. From the definition we see: $\mathrm{HG}_2((m-1,0)) = (m,0)$, $\mathrm{HG}_2^m((m,0)) = (m,m)$ and $\mathrm{HG}_2^m((m,m)) = (0,m)$, so $\mathrm{HG}_2^{2\cdot m+1}((m-1,0)) = (0,m)$. This means that it takes $m^2 - 1 + 2 \cdot m + 1 = (m+1)^2 - 1$ to get to $(0,m)$ from $(0,0)$.

If $2 \nmid m$: Then after $m^2 - 1$ steps we reached $(0, m-1)$. From the definition we see: $\mathrm{HG}_2((0,m-1)) = (0,m)$, $\mathrm{HG}_2^m((0,m)) = (m,m)$ and $\mathrm{HG}_2^m((m,m)) = (m,0)$, so $\mathrm{HG}_2^{2\cdot m+1}((0,m-1)) = (m,0)$. This means that it indeed takes $m^2 - 1 + 2 \cdot m + 1 = (m+1)^2 - 1$ to get to $(m,0)$ from $(0,0)$.

Now we have shown that it holds for HG_2.

Induction hypothesis: Let $d > 2$, $\forall m \in \mathbb{N}\ 2|m \implies \mathrm{HG}_{d-1}^{(m+1)^{d-1}-1}((0,..,0)) = (0,..,0,m)$ and $2 \nmid m \implies \mathrm{HG}_{d-1}^{(m+1)^{d-1}-1}((0,..,0)) = (0,..,0,m,0)$.

Induction step: We use induction over m to prove that $\forall m \in \mathbb{N}$: $2|m \implies \mathrm{HG}_d^{(m+1)^d-1}((0,..,0)) = (0,..,0,m)$ and $2 \nmid m \implies \mathrm{HG}_d^{(m+1)^d-1}((0,..,0)) = (0,..,0,m,0)$.

IB: If $m = 0$, $(m+1)^d - 1 = 1 - 1 = 0$, and we start at $(0,..,0)$, so indeed after 0 steps we end up at $(0,..,0,m) = (0,..,0)$.

IH: Let $m > 0$, we assume that if $2|(m-1)$, $\mathrm{HG}_d^{m^d-1}((0,..,0)) = (0,..,0,m-1)$ and if $2 \nmid (m-1)$, $\mathrm{HG}_d^{m^d-1}((0,..,0)) = (0,..,0,m-1,0)$

IS: First we look at the case where $2|m$: After $m^d - 1$ steps we reached $(0,..,0,m-1,0)$, so after m^d applications of HG_d on $(0,..,0)$ we reached $(0,..,0,m,0)$. We now want to calculate how many steps it takes to get from $(0,..,0,m,0)$ to $(0,..,0,m)$. Note that in this case the rightmost digit of the tuple only changes at particular points, and can only increase.

By the outer induction hypothesis, we know that, starting at $(0,..,0)$ we must apply HG_{d-1} $(m+1)^d - 1$ times to reach $(0,..,0,m)$ and apply HG_{d-1} $m^d - 1$ times to reach $(0,..,0,m-1,0)$, since $(0,..,0,m,0) = \mathrm{HG}_{d-1}((0,..,0,m-1,0))$, it will take $(m+1)^{d-1} - m^{d-1} - 1$ steps to reach $(0,..,0,m)$ from $(0,..,0,m,0)$. Likewise, we must apply HG_{d-1}^{-1} $(m+1)^{d-1} - m^{d-1} - 1$ times to get from $(0,..,0,m)$ to $(0,..,0,m,0)$. It can be formally proven with induction, but it is also very clear from the definition of the function HG_d that for every odd $0 \leq a < m$, we will start at $(0,..,0,m,0,a)$ and reach $(0,..,0,m,a)$ after $(m+1)^{d-1} - m^{d-1} - 1$ steps and at the next step we reach $(0,..,0,m,a+1)$. And for every even $0 \leq a < m$, we will start at $(0,..,0,m,a)$ and reach $(0,..,0,m,0,a)$ after $(m+1)^{d-1} - m^{d-1} - 1$ steps and at the next step we reach $(0,..,0,m,0,a+1)$. And the first tuple where the rightmost digit is m will be $(0,..0,m,m)$. Then by the outer IH it will take $(m+1)^{d-1} - 1$ steps to reach $(0,..,0,m)$. So the total number of steps to reach $(0,..,0,m)$ from $(0,..,0,m,0)$ is $m \cdot ((m+1)^{d-1} - m^{d-1} - 1) + m \cdot 1 + (m+1)^{d-1} - 1 = (m+1)^d - m^d - 1$. The number of steps to reach $(0,..,0,m)$ from $(0,..,0)$ will then be the sum of the number of steps to reach $(0,..,0,m,0)$ from $(0,..,0)$ and the number of steps to get from $(0,..,0,m,0)$ to $(0,..,0,m)$. Using the inner IH this sum is equal to $m^d - 1 + 1 + (m+1)^d - m^d - 1 = (m+1)^d - 1$, which is what we wanted to show.

Now we look at the case where $2 \nmid m$: This is very similar to the case where $2|m$. The difference is that we want to find how many times we must apply HG_d to get from $(0,..,0,m)$ to $(0,..,0,m,0)$. We again want to find for each $0 \leq a \leq m$, how many steps the rightmost digit remains this value a, and then we sum these steps. We again use the

outer induction hypothesis to find that in dimension $d-1$, it takes $(m+1)^{d-1} - m^{d-1} - 1$ steps to get from $(0, .., 0, m)$ to $(0, .., 0, m, 0)$.

With that we have proven that our function HG_d suffices. □

3.3.1 Alternative half growing algorithm

In the definition in 3.3, when the most significant digit of the input tuple $(x_1, .., x_d)$ is odd, we generally apply HG_{d-1} to the first $d-1$ digits, and when the most significant digit is even, we generally apply HG_{d-1}^{-1} to the first $d-1$ digits. In other words, when the most significant digit is odd, the direction of the rest of the tuple is "forwards" and it is "backwards" otherwise. This is the opposite of the reflected Gray codes in 2.2, where even implied "forwards". The definition in 3.3 can actually easily be tweaked to have even imply "forwards". As a matter of fact if we keep the same recursive principles, but in the two dimensional case move from $(0, 0)$ to $(1, 0)$ instead of $(0, 1)$, this new definition naturally follows. One major difference between these two definitions, is that in 3.3 the path through the edge $\mathbb{A}_m^d \setminus \mathbb{A}_{m-1}^d$ moves from $(0, .., 0, 0, m)$ to $(0, .., 0, m, 0)$ or vice versa, and in this new definition the path through the edge moves from $(0, .., 0, 0, m)$ to $(m, 0, .., 0)$ or vice versa. For the rest of the chapter, we use the definitions in 3.3, but the algorithms could be changed to work for the other definition too.

3.4 Recursive half growing cube algorithm implementation

This recursive algorithm uses the assumption that the algorithm is already solved for a lower dimension. The tuples of digits are modeled as lists in this algorithm. It uses two functions, `forwardsHGCube` and `backwardsHGCube`. `forwardsHGCube` is the normal algorithm: you input a list and it changes this original list to the new coordinates, a so called in-situ algorithm. `backwardsHGCube` is the reverse of `forwardsHGCube`: you give it a list and it changes the value of this list to the previous coordinates in the algorithm. Furthermore, these functions take as input an index value. The use of this index value is to indicate which part of the input list we want to look at. If we put as input the coordinate $x_0x_1...x_{n-1}$, but we only want to know which coordinates follow $x_0x_1..x_{j-1}$ in the j-dimensional case, for some $0 \leq j < n$, the input of the function `forwardsHGCube` is the list $x_0x_1..x_{n-1}$ and this j.

Algorithm 1: `forwardsHGCube`

```
Input: list,index
Output: no output
m ← max(list,index)
if index = 0 then list[0] ← list[0] + 1
else if index = 1 then
    if 2|m then
        if list[0] > 0 and list[1] == m then list[0] ← list[0] − 1
        else list[1] ← list[1] + 1
    else
        if list[0] = m and list[1] > 0 then list[1] ← list[1] − 1
        else list[0] ← list[0] + 1
else
    if 2|m then
        if allZeroes(list,index − 1) then list[index] ← list[index] + 1
        else if list[index] = m then backwardsHGCube(list,index − 1)
        else if 2|list[index] then
            if is00m0Shape(list,index − 1) then list[index] ← list[index] + 1
            else backwardsHGCube(list,index − 1)
        else
            if is00mShape(list,index − 1) then list[index] ← list[index] + 1
            else forwardsHGCube(list,index − 1)
    else
        if is00m0Shape(list,index) then list[index − 1] ← list[index − 1] + 1
        else if 2|list[index] then
            if is00mShape(list,index − 1) then list[index] ← list[index] − 1
            else backwardsHGCube(list,index − 1)
        else
            if is00m0Shape(list,index − 1) then list[index] ← list[index] − 1
            else forwardsHGCube(list,index − 1)
```

The backwardsHGCube is very similar with lots of case by case situations. We make use of the auxiliary functions `is00mshape`, `is00m0shape` and `allZeroes` to check if the tuple up to the index has a $0..0m$, $0..0m0$ or $0..0$ shape respectively. Since, in the actual implementation, we have the list inside a struct that also has an array containing the maximum values of the tuple up to index i, we can check if the tuple has the correct shape in constant time.

Justification of algorithm: This implementation follows the definition of the algorithm in 3.3 line for line and therefore is indeed an implementation of the definition.

Complexity of algorithm: The function is recursive, but each new call lowers the index by 1. Therefore, when deciding the successor of a tuple, we have to call `forwardsHGCube` or `backwardsHGCube` $O(d)$ times. If we were to naively decide the maximum value of a tuple or naively check the shape of the input, we would have to loop through the tuple, which is $O(d)$. This would mean that deciding the successor takes $O(d^2)$ time. However, in the implementation we use an extra array to track the maximum value up to index i for every i. This allows us to check the input tuple in constant time, resulting in the total algorithm being in $O(d)$ time. We do however have to update this auxiliary array when we change a digit of the input. This can be done in $O(d)$ time, but since this only has to be done once per tuple, the total algorithm takes $O(d)$ time.

3.5 Constant time half growing cube algorithm

In Section 3.4 we have a linear implementation of the path in Section 3.3. It is linear in the size of the tuple: if the tuple is twice as long, the maximum time to determine the next tuple is twice as long. Intuitively, you might think that a linear complexity is as fast as we can go, since it takes linear time to even determine what the input tuple is, but just like in 2.2.6 it is possible to make the speed not depend on the length of the tuple. It does require some extra memory. The size of the extra memory required is linear in the length of the tuples.

The function is designed to take a tuple and change this tuple to the one that follows in the half growing algorithm from 3.3. For this purpose, the tuple is put inside a class that contains attributes that help determine the next change that has to happen, for instance the index of the digit that has to change. When we apply the function `constantHG` to an instance of this class, we change the tuple accordingly and also set the auxiliary attributes to the values necessary to determine the next change of the tuple. An explanation of the extra attributes can be found below the pseudocode. In this pseudocode, we do not use a class, but instead have the attributes of the struct as input of the function.

Algorithm 2: `constantHG`

```
Input: list,index,D,M,DM,U,indexRef
Output: no output
if index = 0 or index = 1 then
    if !U[1] then
        if D[index] then list[index] ← list[index] + 1
        else list[index] ← list[index] − 1
        if list[index] = M[1] then
            D[index] ←!D[index]
            index ← (index + 1) mod 2
        else if list[index] = 0 then
            D[index] ←!D[index]
            if len(list) = 2 or ( list[2] = M[2] and !( M[1] = M[2] and DM[1] ) )
             then U[1] ← true
            else
                if list[2] = M[2] then DM[1] ←!(DM[1])
                index ← indexRef[2]
                indexRef[2] ← 2
    else
        U[1] ← false
        if DM[1] then M[1] ← M[1] + 1
        else M[1] ← M[1] − 1
        if list[1] ≥ list[0] then
            list[1] ← M[1]
            if M[1] = 0 then
                index ← indexRef[2]
                indexRef[2] ← 2
                U[1] ← true
                if list[2] = M[2] then DM[1] ←!DM[1]
            else index ← 0
        else
            list[0] ← M[1]
            index ← 1
else
    if !U[index] then
        if D[index] then list[index] ← list[index] + 1
        else list[index] ← list[index] − 1
        if list[index] = 0 or list[index] = M[index] then
            if Upgrade next time then U[index] ← true
            else
                indexRef[index] ← indexRef[index + 1]
                indexRef[index + 1] ← index + 1
            D[index] ←!D[index]
        if list[index − 1] = 0 then index ← |index − 3|
        else index ← index − 2
    else
        U[index] ← false
        if DM[index] then M[index] ← M[index] + 1
        else M[index] ← M[index] − 1
        if list[index − 1] ≠ 0 then
            M[index − 1] ← M[index]
            list[index − 1] ← M[index − 1]
            index ← index − 2
        else
            list[index] ← M[index]
            if M[index] = 0 then
                U[index] ← true
                DM[index] ←!DM[index]
                i ← index
                index ← indexRef[i + 1]
                indexRef[i + 1] ← i + 1
            else index ← index − 1
```

Informal explanation algorithm: Let d be the dimension of the input. The input will be of the form $x = (x_1, .., x_d)$. Notice that the indices go from 1 to d, whilst in the code the indices go from 0 to $d-1$ and the digit at index i will be the digit at index $i-1$ in the code. Before we explain any further, we have to introduce some auxiliary matrices and variables: The array of booleans D, where D[i] is True if the direction of the digit at index i is positive and False otherwise. The array of non-negative integers M, where M[i] contains the maximum value so far: M[i] = $\max\{x_1, .., x_i\}$. The array of booleans DM, where DM[i] is True if the direction of the maximum value at index i is positive and False otherwise. The array of booleans U, where U[i] indicates whether the next time we would change the value at index i, we change the maximum value at i and a value at either i or $i-1$. And lastly, the array indexRef, that contains a reference to a higher index. This array has the same function as the array of focus pointers in Knuth's algorithm in 2.2.6

...	...	...	...
$[0,0,1,1,3,0]$	$[0,0,1,1,4,1]$	$[0,1,1,4,0,2]$	$[0,0,0,1,1,4]$
$[0,0,0,1,3,0]$	$[0,0,0,1,4,1]$	$[0,0,1,4,0,2]$	$[0,0,0,0,1,4]$
$[0,0,0,0,3,0]$	$[0,0,0,0,4,1]$	$[0,0,0,4,0,2]$	$[0,0,0,0,0,4]$
$[0,0,0,0,4,0]$	$[0,0,0,0,4,2]$	$[0,0,0,4,0,3]$	$[0,0,0,0,0,5]$
$[0,0,0,1,4,0]$	$[0,0,0,1,4,2]$	$[0,0,1,4,0,3]$	$[0,0,0,0,1,5]$
$[0,0,1,1,4,0]$	$[0,0,1,1,4,2]$	$[0,1,1,4,0,3]$	$[0,0,0,1,1,5]$
...	...	...	...

Figure 5: In the second and third example we just change the most significant digit. In the other two examples we update the most significant digit, that is we change the maximum value.

First we will just look at how the most significant digit, x_d, behaves in the case where $d > 2$, see figure 5: The most significant digit is moving from 0 to m_i or from m_i to 0, depending on whether m_i is odd or even. At these steps, the shape of the input is either $0..0ma$ or $0..0m0a$. Eventually he will reach the "border": M[i] if the direction is positive and 0 if the direction is negative. In this case we must set U[d] to True, indicating that next time we normally change the value at index d, we change the maximum value at index d. This eventually happens at shape $0..0m0$ or $0..0m$. In the former case we move from $0..0m0$ to $0..0(m+1)0$, in the latter case we move $0..0m$ to $0..0(m+1)$. For the most significant maximum value, the direction always remains positive, i.e., DM[d] remains True.

Now we look at how the second most significant digit, x_{d-1}, behaves, where we assume that $d-1 > 2$: This digit also moves from 0 to M[$d-1$] or vice versa, depending on the direction D[$d-1$]. When it reaches a border, there is a couple of possibilities: If $x_d \neq$ M[d], we swap the direction D[$d-1$] and refer to index d, which means that the next time we would normally do something at index $d-1$, we instead do something at index d. This is similar to the reflect step in 2.2.6. If $x_d =$ M[d], we have a more difficult situation. When $x_d \neq$ M[d], we have to visit all tuples for which atleast one of x_1,..,x_{d-1} is equal to M[d]. When $x_d =$ M[d], this restriction does not hold, so we have to visit a lot more possibilities for the first $d-1$ digits. As a result of this, the value at index $d-1$ does not just go between 0 and M[d], but we actually have to either increase M[$d-1$] from 0 to M[d] or decrease M[$d-1$] from M[d] to 0, where between every step of M[$d-1$] x_{d-1} also moves between 0 and M[$d-1$]. Eventually M[$d-1$] reaches its own "border" 0 or M[d], depending on its direction, and in that situation we do refer to index d the next time x_{d-1} reaches its "border".

The definition of the algorithm in 3.3 (not this implementation) is recursive, so the

...	...	...	...
$[0,4,0,2]$	$[0,0,4,3]$	$[0,4,0,4]$	$[0,2,2,4]$
$[0,4,0,3]$	$[0,0,4,4]$	$[0,3,0,4]$	$[0,2,1,4]$
...	...	...	...
$[4,0,0,3]$	$[0,4,4,4]$	$[3,0,0,4]$	$[2,0,1,4]$
$[4,0,1,3]$	$[0,4,3,4]$	$[3,0,1,4]$	$[2,0,0,4]$
...	...	...	...
$[0,4,1,3]$	$[4,0,3,4]$	$[0,3,1,4]$	$[0,2,0,4]$
$[0,4,2,3]$	$[4,0,2,4]$	$[0,3,2,4]$	$[0,1,0,4]$
...	...	...	...
$[4,0,2,3]$	$[0,4,2,4]$	$[3,0,2,4]$	$[1,0,0,4]$
$[4,0,3,3]$	$[0,4,1,4]$	$[3,0,3,4]$	$[1,0,1,4]$
...	...	...	...
$[0,4,3,3]$	$[4,0,1,4]$	$[0,0,3,4]$	$[0,0,1,4]$
$[0,4,4,3]$	$[4,0,0,4]$	$[0,0,2,4]$	$[0,0,0,4]$
...	...	...	...

Figure 6: All the points between $[0,4,0,2]$ and $[0,0,0,4]$ where the second most significant digit changes, or changes its max value.

other digits behave similarly to the digit at index $d-1$. The exceptions are the digits at indices 1 and 2. This is partly because in the normal case when changing the value at index i, we afterwards have to check the shape of the values at $i-2$ and $i-1$, to determine the next index we should change. For this reason, instead of putting extra if statements everywhere, the algorithm for changing index 1 and 2 is programmed differently. Let $m = \mathrm{M}[2]$, then if we change a value at index 1 or 2, we are either in the process of moving from $m0$ to $0m$ or vice versa. The route from $m0$ to $0m$ goes $m0$,$m1$,$m2$,..,mm,$(m-1)m$,$(m-2)m$,..,$0m$. When we reach the end, we either change a higher index digit at the next iteration, or we change the maximum value of index 1 and 2 and the corresponding digit. Note that we do not actually use M[1].

An important part of the algorithm is deciding what the next index is that should be changed. Generally, when changing index i, $x_1x_2..x_{i-1}$ is either of the form $0..0m$ or $0..0m0$, where $m = \mathrm{M}[i]$. In the first case, the next index to change is $i-2$ and in the second case the next index to change is $i-3$. However, when changing $\mathrm{M}[i]$, $x_1x_2..x_i$ is of the form $0..0m$ or $0..0m0$. Then in the first case, the next index to change is $i-1$ and in the second case the next index to change is $i-2$. In the special case where the $\mathrm{M}[i]$ gets changed to 0, the next index to change is the one referenced to by $i+1$.

Complexity analysis: When determining the successor of a tuple, the number of commands is bounded and does not depend on the length of the tuple, therefore the algorithm runs in constant time.

4 Fully growing cube

4.1 The problem

The problem of the "fully growing" cube is similar to the problem of the "half growing" cube, except for one major difference: instead of using the alphabet $\mathbb{A}_m = \{0, .., m\}$ at step m, we use the alphabet $\mathbb{B}_m = \{-m, -m+1, .., -1, 0, 1, .., m-1, m\}$. So for a dimension $d \in \mathbb{N}_{>0}$ we want to order all tuples of size d with digits from alphabet $\mathbb{Z}$, such that the first $(2 \cdot m + 1)^d$ tuples only contain digits from alphabet $\mathbb{B}_m = \{-m, .., m\}$ and furthermore, two consecutive tuples differ in only one index and the difference between the two digits at this index is 1.

Definition 4.1. A **fully growing cube code of dimension** d is a bijection $g : \mathbb{Z}^d \to \mathbb{N}$, such that

$$g(x) - g(y) = 1 \implies K(x, y) = 1$$

and

$$g(x) \leq g(y) \implies \bar{M}(x) \leq \bar{M}(y),$$

where $K((x_1, ..x_d), (y_1, .., y_d)) = \sum_{i=1}^{d} |x_i - y_i|$ and $\bar{M}((x_1, ..x_d)) = \max\{|x_1|, .., |x_d|\}$.

Just like in the last chapter, an application of this code could be to traverse polynomials of a certain size, where we want to try the ones with a minimal absolute value of the coefficients first and want to minimize the changes at each step.

4.2 Modeled as a Hamiltonian path through a growing hypercube

4.2.1 One dimension

In the one-dimensional case, it is not possible to find a path. We start at 0 and then have the option to move to either 1 or -1. After moving to 1, for example, we are not able to move to -1, since we would have to jump two squares.

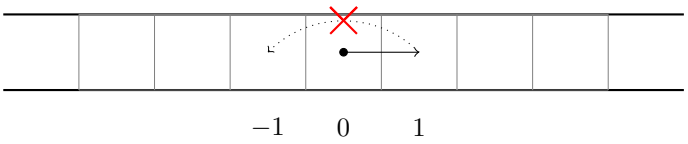

Figure 7: No route exists in the one-dimensional case, since we have to jump two squares.

4.2.2 Two dimensions

We can visualize this as a square. Starting at 00, we have four options to move to: 01, 10, -10, $0(-1)$. Next, we want to fill the rest of the edge $E_1 = \{ab \in \mathbb{B}_1^2 : |a| = 1 \text{ or } |b| = 1\}$. We can fill this edge either clockwise or counterclockwise. Every time we fill an edge $E_m = \{ab \in \mathbb{B}_m^2 : |a| = m \text{ or } |b| = m\}$, we can step out towards edge E_{m+1} and then we, again, have the option to fill this edge either clockwise or counterclockwise.

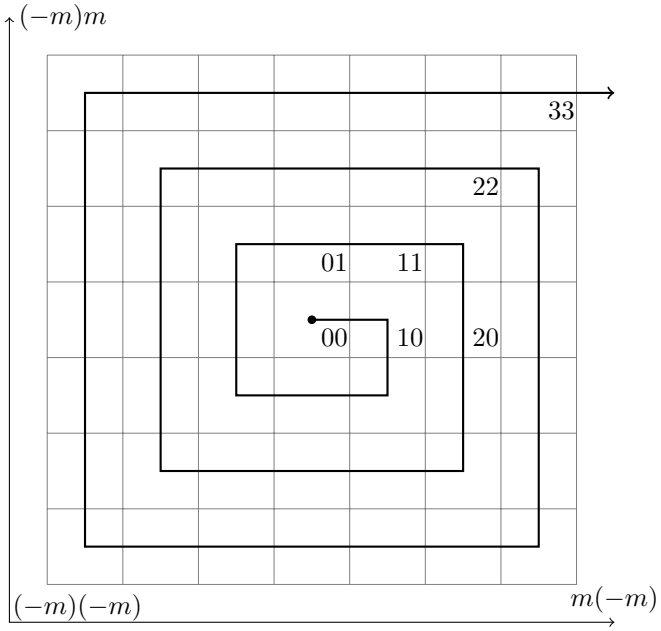

Figure 8: The route in the two-dimensional case. The spiral pattern resembles a snail shell.

In our eventual algorithm that we use for higher dimensional solutions too, from 00 we move to 10 and then fill the edge clockwise. In general, we move from $(m-1)(m-1)$ to $m(m-1)$ and then fill the E_m edge clockwise, ending in mm. This route resembles the shell of a snail and will henceforth be called the *snail path*. See Figure 8.

4.2.3 Three dimensions

We can visualize this situation as a cube. The situation for alphabet $\mathbb{B}_1$ is a cube where we start in the middle cube, at 000, and have to visit all the other individual cubes of this $3 \times 3 \times 3$ cube. You can try, but you will find that each time you will have at least one unreachable cube left. Sadly, this is the case for all odd dimensions, which will be proven in 4.2.4.

4.2.4 Chessboard argument

We will now prove that it is impossible to have a Hamiltonian path through the hypercube $\{-m, .., 0, .., m\}^d$ starting at $(0, .., 0)$ for an odd dimension $d \in \mathbb{N}$ and $m \in \mathbb{Z}_{>0}$. We define a function $W : \cup_{n \in N} \mathbb{Z}^n \to \{-1, 1\}$ recursively with $W(()) = 1$ and for $d \geq 1$

$$W((x_1, x_2, .., x_d)) = \begin{cases} W((x_1, x_2, .., x_{d-1})), & \text{if } 2 | x_d \\ -W((x_1, x_2, .., x_{d-1})), & \text{if } 2 \nmid x_d \end{cases}$$

The intuition behind this function is that, if we have have the $\{-m, .., m\}^d$ hypercube and we paint the cells in this hypercube either white or black, such that $(0, .., 0)$ is painted white and two direct neighbours (not diagonal) always differ in color, then $W((x_1, x_2, .., x_d))$ equals 1 if the cell is white and it equals -1 if the cell is painted black.

Another useful function is the following: $G : \mathbb{Z}_{>0} \times \mathbb{N} \to \mathbb{Z}$ with $G(d, m) = \sum_{x \in \{-m, .., m\}^d} W(x)$. This gives the difference between the number of white cells and the number of black cells in the $\{-m, .., m\}^d$ hypercube.

Theorem 4.2. *If $2|d$ or $2|m$, then $G(d,m) = 1$ and if both $2 \nmid d$ and $2 \nmid m$, then $G(d,m) = -1$.*

Proof: We will prove this with induction over the dimension d.

Induction basis: The dimension d equals $d = 1$. Take $m \in \mathbb{N}$. Every even number in $\{-m, .., m\}$ is white and every odd is black. The number of white squares is then equal to $1 + 2 \cdot \lfloor m/2 \rfloor$ and the number of black squares is equal to $2 \cdot \lceil m/2 \rceil$. This means that if $2|m$, then $G(1,m) = 1 + 2 \cdot m/2 - 2 \cdot m/2 = 1$ and if $2 \nmid m$, then $G(1,m) = 1 + 2 \cdot (m-1)/2 - 2 \cdot (m+1)/2 = -1$.

Induction hypothesis: We assume that $\forall c < d$ and $\forall m \in \mathbb{N}$, $G(c,m) = 1$ if $2|c$ or $2|m$ and $G(c,m) = -1$ if $2 \nmid c$ and $2 \nmid m$.

Induction step: Let d be the dimension. Take $m \in \mathbb{N}$ arbitrarily.

$$\begin{aligned}
G(d,m) &= \sum_{(x_1,..,x_d) \in \mathbb{B}_m^d} W((x_1,..,x_d)) \\
&= \sum_{a \in \mathbb{B}_m} \sum_{(x_1,..,x_{d-1}) \in \mathbb{B}_m^{d-1}} W((x_1,..,x_{d-1},a)) \\
&= \sum_{a \in \mathbb{B}_m} \sum_{(x_1,..,x_{d-1}) \in \mathbb{B}_m^{d-1}} (W((x_1,..,x_{d-1})) \cdot W(a)) \\
&= \sum_{a \in \mathbb{B}_m} \Big(W(a) \cdot \sum_{(x_1,..,x_{d-1}) \in \mathbb{B}_m^{d-1}} W((x_1,..,x_{d-1})) \Big) \\
&= \sum_{a \in \mathbb{B}_m} (W(a) \cdot G(d-1,m)) \\
&= G(d-1,m) \cdot \sum_{a \in \mathbb{B}_m} W(a) \\
&= G(d-1,m) \cdot G(1,m)
\end{aligned}$$

Now, if $2|d$, then $G(d-1,m) = G(1,m)$, so $G(d,m) = 1$. If $2 \nmid d$ and $2|m$, then $G(d-1,m) = 1$ and $G(1,m) = 1$, so $G(d,m) = 1$. Lastly, if $2 \nmid d$ and $2 \nmid m$, then $G(d-1,m) = 1$ and $G(1,m) = -1$, so $G(d,m) = -1$. □

Now we know that in the hypercube $\mathbb{B}_m^d$ the number of white cells is one more than the number of black cells if $2|d$ or $2|m$ However in the case where $2 \nmid d$ and $2 \nmid m$, there are more black cells than white cells, and since we start at a white cell, it is impossible to have a Hamiltonian path without passing over some white cell twice, or making an illegal jump. Furthermore, if the dimension is odd, every new layer either has two extra black cells or two extra white cells, so we can not solve the problem by simply starting at a black cell.

4.2.5 Four dimensions

We can visualize the four dimensions as a square grid, where at every point there is another square grid. For example, in the situation where the alphabet is $\mathbb{B}_2 = \{-2, .., 2\}$, we can visualize 5 by 5 big squares, where each big square itself consists of 5 by 5 smaller squares. For the tuple *abcd*, the last two digits *cd* indicate in which big square the coordinate lies, and the first two digits *ab* indicate the location inside this big square. When going from alphabet $\mathbb{B}_1$ to $\mathbb{B}_2$, assuming that we visited all points in $\mathbb{B}_1^4$, there are two types of coordinates that get added. Firstly, we add an entire layer of big squares that lies around the original 9 big squares. These new big squares are totally unvisited. Secondly, around each original big square, an edge $E_2 = \{ab \in \mathbb{B} : |a| = 2 \text{ or } |b| = 2\}$ gets added that increases the size of these big squares from 3 by 3 to 5 by 5. Of these

9 big squares, only this new outer edge is unvisited. These new unvisited squares are colored white in the following figures.

There are many different paths to take, so we can set some restrictions for the algorithm. For example, we can say that we always want to end at $mmmm$. Another restriction is that we first want to visit all the inner big squares and afterwards visit all the outer big squares. In other words, we first want to visit all the elements of $(\mathbb{B}_m^2 \setminus \mathbb{B}_{m-1}^2) \times \mathbb{B}_{m-1}^2 = E_m \times \mathbb{B}_{m-1}^2$ and afterwards the elements of $\mathbb{B}_m^2 \times (\mathbb{B}_m^2 \setminus \mathbb{B}_{m-1}^2) = \mathbb{B}_m^2 \times E_m$. We would also like to use our solution for two dimensions as much as possible, as using lower dimensional solutions will help us to find an algorithm for higher dimensions.

The following algorithm, visualize in the figures, meets these requirements: First we step out from $(m-1)(m-1)(m-1)(m-1)$ to $m(m-1)(m-1)(m-1)$. Next, we move to $m(m-1)00$, by using the snail path backwards on the last two coefficients and not moving in the first two coefficients. We now want to fill the rest of the edge of big square 00. The edge filling path for the two-dimensional case first moves down from $m(m-1)$ to $m(m-2)$ and then fills the rest of the edge and ends up in mm. We fill up the rest of the edge of big square 00 by using this path and moving from $m(m-1)00$ to $mm00$. Now we move to a new big square, from $mm00$ to $mm10$. In this big square we also need to fill the rest of the edge. We use the path for E_m backwards, which goes from $mm10$ to $m(m-1)10$, but we already visited $m(m-1)10$, so we stop on the prior square, $m(m-2)10$. Now we move through the big squares using the snail path, and at each big square we use the path for the two-dimensional edge, alternating between backwards and forwards. This way, we will eventually end up at $mm(m-1)(m-1)$ and the entirety of $\mathbb{B}_m^2 \times \mathbb{B}_{m-1}^2$ has been visited.

We now want to visit $B_m^2 \times E_m$. First, we move from $mm(m-1)(m-1)$ to $mmm(m-1)$, where we continue our snail path through the big squares. The difference is that in big square $m(m-1)$, we have to visit all small squares instead of just the outer edge. We can use the snail path backwards to move from $mmm(m-1)$ to $00m(m-1)$. After stepping to $00m(m-2)$ we use the snail path forwards to move to $mmm(m-2)$. This way we use the snail path to move through the big squares, but we also use the snail path alternatingly forwards and backwards inside these big squares to visit all the small squares. Finally, we end up in $mmmm$.

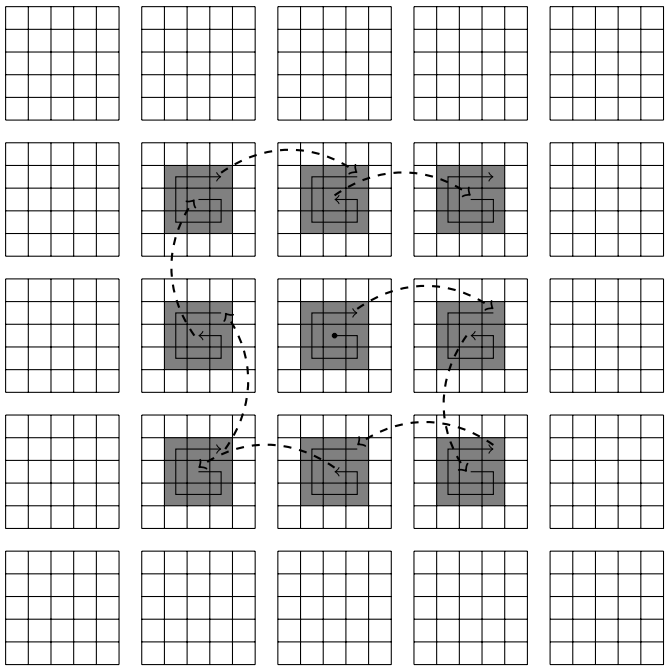

Figure 9: The route through $\{-1, 0, 1\}^4$.

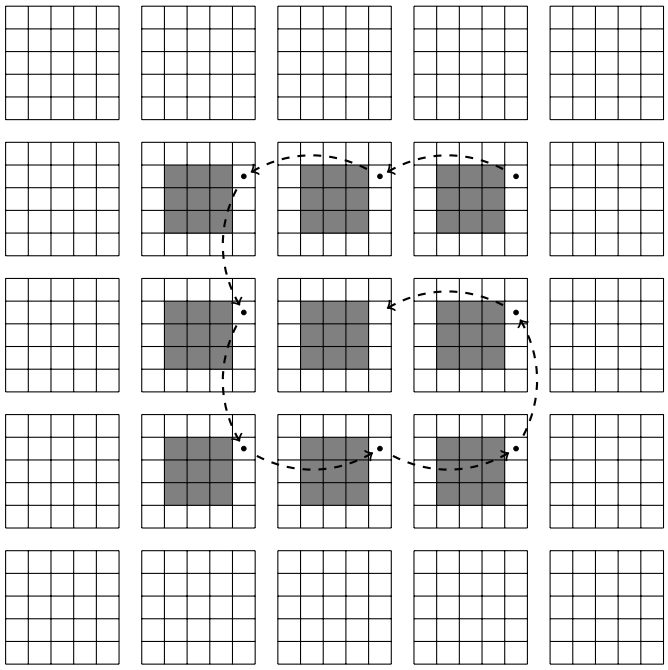

Figure 10: The first step of the route through $\mathbb{B}_m^4$ is moving from $(m, m-1, m-1, m-1)$ to $(m, m-1, 0, 0)$.

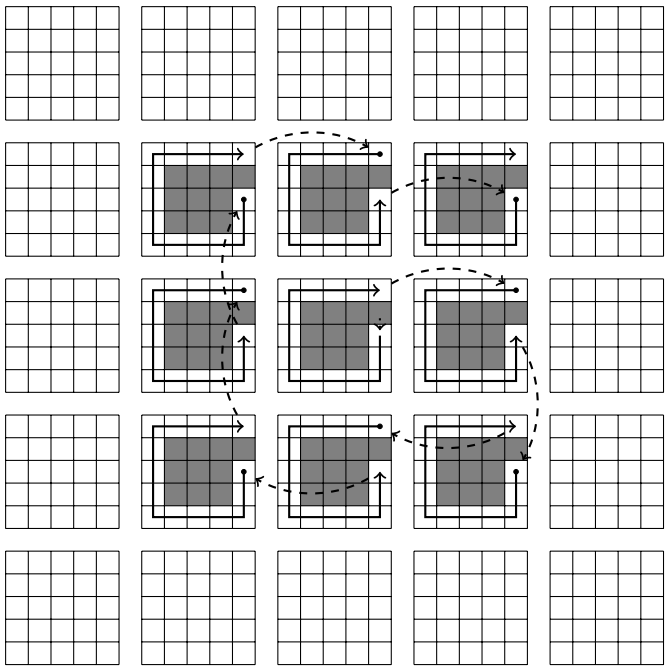

Figure 11: Next, we move through the rest of $\mathbb{B}^2_m \times \mathbb{B}^2_{m-1}$.

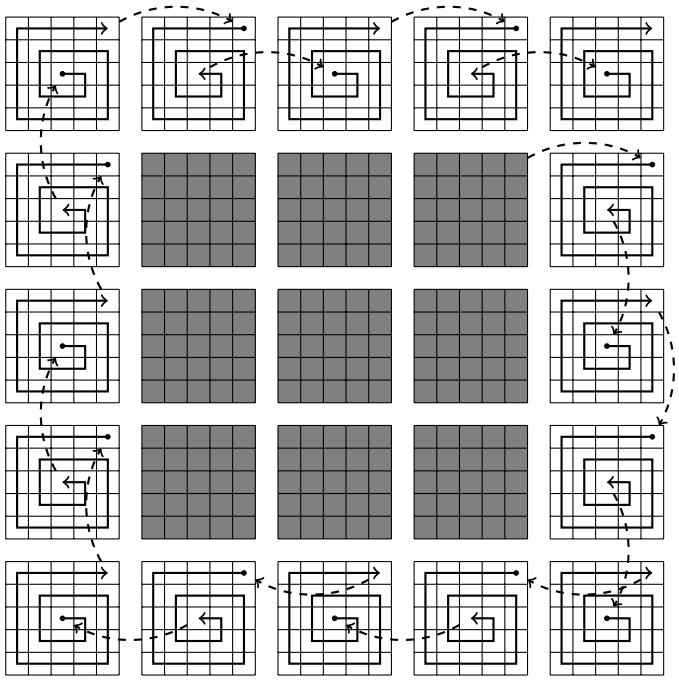

Figure 12: Lastly, we move through $\mathbb{B}^2_m \times (\mathbb{B}^2_m \setminus \mathbb{B}^2_{m-1})$. We use the route through $\mathbb{B}^2$ from 4.2.2, alternating between forwards and backwards.

4.2.6 Even dimensions

The algorithm for four dimensions can be generalized for the higher even dimensions. Let $d \in \mathbb{N}$ be an even number higher than 2. For the algorithm for d dimensions, we require some prerequisites:

- The snail path for traversing $\mathbb{B}_m^2$ that moves from 00 to mm.
- The algorithm for the $(d-2)$-dimensional case, that starts at 00..00 and ends at $mm..mm$ while visiting all elements in $\mathbb{B}_m^{d-2}$.
- An algorithm for the $(d-2)$-dimensional edge, so for visiting all elements of $E_m = B_m^{d-2} \setminus B_{m-1}^{d-2}$. This algorithm start at $(m, m-1, .., m-1, m-1)$ and ends at $(m, m, .., m, m)$, but in this case we leave out the first square, so it starts from the second square $(m, m-1, .., m-1, m-2, m-1)$ and moves through $E_m \setminus \{(m, m-1, .., m-1, m-1)\}$.

Our first step is moving from $(m-1, .., m-1)$ to $(m, m-1, .., m-1)$. While fixing the first $d-2$ coordinates, we use the backwards snail path on the last two coordinates. This means that our next position is $(m, m-1, .., m-1, m-1, m-2, m-1)$ and we will eventually end up in $(m, m-1, .., m-1, m-1, 0, 0)$. Now we use the $(d-2)$-dimensional edge algorithm on the first $d-2$ coordinates to move to $(m, m, .., m, m, 0, 0)$. Next, we move to $(m, m, .., m, m, 1, 0)$ and do the backwards $(d-2)$-dimensional edge algorithm on the first $d-2$ coordinates to end at $(m, m-1, .., m-1, m-2, m-1, 1, 0)$. We use the snail path on the last two coordinates to move from big square to big square and at each big square we do the $(d-2)$-dimensional edge algorithm either forwards or backwards. At some point we will reach $(m, m, .., m, m, m-1, m-1)$ and from here on out we will continue using the snail path on the last two coordinates, but on each new big square, we will use the algorithm for the entire $(d-2)$-dimensional case on the first $d-2$ coordinates. We will end up in $(m, m, .., m, m, m, m)$ after visiting all elements in $\mathbb{B}_m^d$.

4.2.7 Some alternatives for odd dimensions

From section 4.2.4 we know that a path for odd dimensions does not exist if we want to follow all the conditions of the definition. We can however make paths by allowing slight deviations from the original definition. Without going into too much detail, some of the amendments that allow an algorithm to exist are:

- Allowing a minimal amount of bigger jumps.
- Allowing the path to pass specific coordinates twice.
- Going in a higher layer for a few steps, before going back and finishing the current layer.

4.3 Recursive definition fully growing cube algorithm for even dimensions

Our goal is to arrange, for a given dimension d, all elements of $\mathbb{Z}^d$, in such a way that for two following elements $x = x_1..x_d$ and $y = y_1..y_d$, $\sum_{i=1}^{d} |x_i - y_i| = 1$. Furthermore, for every $m \in \mathbb{N}$, the first $(2m+1)^d$ elements should only contain digits from $\{-m, -m+1, , .., m-1, m\}$. We model this as a bijective function FG from $\mathbb{Z}^d$ to $\mathbb{Z}^d \setminus \{(0, .., 0)\}$, where the arrangement we get is given by starting at element $(0, .., 0)$ and repeatedly applying this function FG. As shown in 4.2.4, the function can only be defined for even dimensions.

This function FG, corresponding to the Hamiltonian paths from 4.2, and its inverse FG^{-1} are defined as follows:

When the dimension is $d = 2$:

Let m be the maximal value of the absolute values of the input, i.e., if our input is $(x_1, x_2, .., x_d)$, $m = \max\{|x_1|, |x_2|, .., |x_d|\}$.

$\mathrm{FG}_2 : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N} \setminus \{(0,0)\}$

$$\begin{aligned}
(a, m) &\mapsto (a+1, m)\\
(-m, a) &\mapsto (-m, a+1)\\
(a, -m) &\mapsto (a-1, -m)\\
(m, a) &\mapsto (m, a-1)
\end{aligned}$$

$\mathrm{FG}_2^{-1} : \mathbb{N} \times \mathbb{N} \setminus \{(0,0)\} \to \mathbb{N} \times \mathbb{N}$

$$\begin{aligned}
(m, m-1) &\mapsto (m-1, m-1)\\
(m, a) &\mapsto (m, a+1)\\
(a, -m) &\mapsto (a+1, -m)\\
(-m, a) &\mapsto (-m, a-1)\\
(a, m) &\mapsto (a-1, m)
\end{aligned}$$

When the dimension is $d = 2k > 2$:

Notation: (**SLShape**) represents the tuple $(m, m-1, .., m-1, m-2, m-1)$. As defined before, if $x = (x_1, x_2, .., x_{d-1}, x_d)$, then $x_1^{d-1} = (x_1, x_2, .., x_{d-2}, x_{d-1})$.

$\mathrm{FG}_d : \mathbb{N}^d \to \mathbb{N}^d \setminus \{(0, .., 0)\}$

$$\begin{aligned}
&(m, .., m) && \mapsto (m+1, m, .., m) && \\
&(m, m-1, .., m-1, a, b) && \mapsto (m, m-1.., m-1)\mathrm{FG}_2^{-1}((a,b)) && \text{with } (a,b) \neq 0 \\
& && && \text{and } |a| \neq m \text{ and } |b| \neq m \\
&(m, .., m, a, b) && \mapsto (m, .., m)\mathrm{FG}_2((a,b)) && \text{with } 2|(a+b) \\
&x_1^{d-2}(a, b) && \mapsto \mathrm{FG}_{d-2}(x_1^{d-2})(a, b) && \text{with } 2|(a+b) \\
&(0, .., 0, a, b) && \mapsto (0, .., 0)\mathrm{FG}_2((a,b)) && \\
&(\mathbf{SLShape})(a, b) && \mapsto (\mathbf{SLShape})\mathrm{FG}_2((a,b)) && \text{with } |a| \neq m \\
& && && \text{and } |b| \neq m \\
&x_1^{d-2}(a, b) && \mapsto \mathrm{FG}_{d-2}^{-1}(x_1^{d-2})(a, b) &&
\end{aligned}$$

$\mathrm{FG}_d^{-1} : \mathbb{N}^d \setminus \{(0, .., 0)\} \to \mathbb{N}^d$

$$\begin{aligned}
&(m, m-1, .., m-1) && \mapsto (m-1, m-1, .., m-1) && \\
&(m, m-1, .., m-1, a, b) && \mapsto (m, m-1.., m-1)\mathrm{FG}_2((a,b)) && \text{with } |a| \neq m \\
& && && \text{and } |b| \neq m \\
&(m, .., m, a, b) && \mapsto (m, .., m)\mathrm{FG}_2^{-1}((a,b)) && \text{with } 2 \nmid (a+b) \\
&x_1^{d-2}(a, b) && \mapsto \mathrm{FG}_{d-2}(x_1^{d-2})(a, b) && \text{with } 2 \nmid (a+b) \\
&(0, .., 0, a, b) && \mapsto (0, .., 0)\mathrm{FG}_2^{-1}((a,b)) && \\
&(\mathbf{SLShape})(a, b) && \mapsto (\mathbf{SLShape})\mathrm{FG}_2^{-1}((a,b)) && \text{with } (a,b) \neq 0 \\
& && && \text{and } |a| \neq m \text{ and } |b| \neq m \\
&x_1^{d-2}(a, b) && \mapsto \mathrm{FG}_{d-2}^{-1}(x_1^{d-2})(a, b) &&
\end{aligned}$$

Proof correctness of algorithm: For this proof, we can use the same strategy as for the proof in 3.3. Therefore, it will not be as rigorous, but instead we will sketch the outlines and where necessary state in which way it is similar and in which way it differs from the proof in 3.3.

We want to prove that for a dimension $d = 2k$, the path given by starting at $(0, .., 0)$ and repeatedly applying the algorithm FG has the desired properties. These properties are that at every step we only change one digit, and this digit changes by plus or minus

1, we will eventually visit every element of $\mathbb{Z}^d$, we will not visit the same element twice and for every $m \in \mathbb{N}$, the elements of $\{-m, .., m\}^d$ will be visited before the elements of $\{-(m+1), .., m+1\}^d \setminus \{-m, .., m\}^d$.

To show that these properties hold, we will prove for FG that $\forall d \in \mathbb{Z}_{>0}$ with $2|d$:

(1) $\mathrm{FG}_d^* \circ \mathrm{FG}_d = \mathrm{id}_{\mathbb{Z}^d}$.

(2) $\mathrm{FG}_d \circ \mathrm{FG}_d^* = \mathrm{id}_{\mathbb{Z}^d \setminus \{(0,..,0)\}}$.

(3) $\forall x \in \mathbb{N}^d \; K_d(x, \mathrm{FG}_d(x)) = 1$.

(4) $\forall x \in \mathbb{Z}^d \setminus V_d \; \bar{M}(\mathrm{FG}_d(x)) = \bar{M}(x)$ and $\forall x \in V_d \; \bar{M}(\mathrm{FG}_d(x)) = \bar{M}(x) + 1$, with $V_d = \{(m, .., m) \in \mathbb{Z}^d\}$.

Where FG_d^* is the function defined by FG_d^{-1} above (we have yet to show that it is indeed the inverse) and we use the auxiliary functions: $\bar{M}((x_1, x_2, .., x_d)) = \max\{|x_1|, |x_2|, .., |x_d|\}$ and $K_d((x_1, ..x_d), (y_1, .., y_d)) = \sum_{i=1}^{d} |x_i - y_i|$.

From properties (1) and (2) it follows that FG is bijective, so when repeatedly applying FG we will not visit the same element twice, from property (3) it follows that only one digit changes, and this digit changes by plus or minus 1 and from property (4) it follows that the maximal value of the absolute values of the digits does not go down and only increases at specific points. Then it remains to show that we indeed visit every element of $\mathbb{Z}^d$ eventually.

Properties (1), (2), (3) and (4) can be proven with induction on dimension d, where the base case is $d = 2$ and we take steps of 2. We prove the base case and induction step by taking every type of input and manually checking that the properties hold. These types of input generally coincide with the individual lines of the definition of FG. Here we make extensive use of the induction hypothesis.

To prove that we visit each element when we start at $(0, .., 0)$ and repeatedly apply FG, we will show that for all dimensions $d \in 2\mathbb{Z}_{>0}$ and $m \in \mathbb{N}$, we reach $(m, .., m)_d$ after $(2 \cdot m + 1)^d - 1$ steps. Because we will not visit two elements twice and all the elements we visited lie in $\{-m, .., m\}^d$, and $\#\{-m, .., m\}^d = (2 \cdot m + 1)^d$, the elements visited is exactly equal to $\{-m, .., m\}^d$. We will prove that it takes $(2 \cdot m + 1)^d - 1$ steps to reach $(m, .., m)$ with induction over d where d starts at 2 and takes steps of 2 and then induction over $m \in \mathbb{N}$. For a given d and m, we want to count the number of steps to get from $(m, .., m)_d$ to $(m + 1, .., m + 1)_d$. This path consists of $(2 \cdot m + 1)^d$ $(d-2)$-dimensional paths from $(m + 1, m, .., m, m - 1, m)_{d-2}$ to $(m + 1, .., m + 1)_{d-2}$ or in reverse and $(2 \cdot (m+1) + 1)^d - (2 \cdot m + 1)^d$ $(d-2)$-dimensional paths from $(0, .., 0)_{d-2}$ to $(m + 1, .., m + 1)_{d-2}$ or in reverse. If we add the steps to move between the outer squares and use the induction hypothesis for dimension $d - 2$, we find that getting from $(m, .., m)$ to $(m + 1, .., m + 1)$ takes $(2 \cdot (m + 1) + 1)^d - (2 \cdot m + 1)^d$ steps. Then we conclude that it takes $(2 \cdot (m+1)+1)^d - 1$ steps to get to $(m+1, .., m+1)$ from $(0, .., 0)$.

4.4 Recursive fully growing Cube algorithm for even dimensions implementation

Just like in the half growing case, our algorithm uses the assumption that we have an algorithm for the lower dimensional case. We, again, model the tuples of digits as lists. For practical reasons, these lists are put in classes, which contain some extra information about the lists and some useful methods. This extra information we keep track of, helps to lower the complexity significantly. For simplicity, we just use the lists in the following pseudocode and not the classes. The main functions of the algorithm are called: `forwardsFGCube` and `backwardsFGCube`, which will change an input tuple to its successor or predecessor respectively. These functions also have an extra index as input. The use of this index value is to indicate which part of the input list we want to look at. If we put as input the coordinate $x_0x_1...x_{n-1}$, but we only want to know which

coordinates follow $x_0x_1..x_{j-1}$ in the j-dimensional case, for some $0 \leq j < n$, the input of the function `forwardsFGCube` is the list $x_0x_1..x_{n-1}$ and this j.

Algorithm 3: `forwardsFGCube`

```
Input: list,index
Output: no output
Result: The input list will
if index = 1 then snailPath(list,index)
else
    m ← max(list,index)
    if allM(list,index,m) then list[0] ← list[0] + 1
    else if !inOuterLayer(list,index) & snakeShape(list,index − 2,m) &
     !(list[index − 1] = 0 & list[index] = 0 ) then
        snailPathBack(list,index)
    else if even(list,index) then
        if allM(list,index − 2,m) then snailPath(list,index)
        else forwardsFGCube(list,index − 2)
    else
        if allZeroes(list,index − 2) then snailPath(list,index)
        else if !inOuterLayer(list,index) & snakeShapeLowTail(list,index − 2,m)
         then snailPath(list,index)
        else backwardsFGCube(list,index − 2)
```

Some of these methods will need some clarification:

`snailPath(list,index)`: this will move the the digits at index-1 and index as in the 2-dimensional case, see 4.2.2.

`snakeShape(list,i,m)`: Checks if the tuple up to index i is of the form: $(m, m-1, .., m-1)$.

`snakeShapeLowTail(list,i,m)`: Checks if the tuple up to index i is of the form: $(m, m-1, .., m-1, m-2, m-1)$.

`inOuterLayer(list,i)`: Checks if the absolute value of the value at index i or $i-1$ is equal to the maximum absolute power up to index i.

By the extra information given in the class, the methods such as max and snakeShape can be ran in constant time. This allows the algorithm to be in linear time, which will be shown in the next section.

Justification of algorithm: Just like in 3.4 the implementation follows the definition very closely.

Complexity analysis: The function is recursive and can have $O(d)$ recursive calls. Per call the complexity is $O(1)$, because the auxiliary functions are programmed to be constant time. This does require some auxiliary arrays though, so when changing the digit of the tuple, we also have to update these arrays. This can happen in $O(d)$ time, but only has to happen once per input tuple. Therefore, the total algorithm is in $O(d)$ time.

4.5 Constant time fully growing cube algorithm

In Section 3.5 we used a strategy to get a constant time algorithm for the half growing problem. There is no reason to believe this can not be applied to the fully growing problem too. However, due to time constraints, this has not been programmed in this thesis.

References

[1] Gideon Ehrlich James R. Bitner and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, 1976.

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

[3] Wikipedia contributors. Gray code — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Gray_code&oldid=1027748042`, 2021. [Online; accessed 6-July-2021].

Appendix

In this section we have placed the actual python code of the implementations of the algorithms described in this document.

Linear time memoryless Gray code implementation

Here we have the python implementation of the algorithm described in 2.2.3 and 2.2.5.

```
def first_non_max_index(list,alph_size):
    for i in range(0,len(list)):
        if(list[i]!=alph_size-1):
            return i
    return len(list)

def max_forwards(list,alph_size):
    if(alph_size%2==0):
        if(list[1]%2==0):
            list[1] += 1
        else:
            list[1] -= 1
    else:
        index = first_non_max_index(list,alph_size)
        if(list[index]%2==0):
            list[index] += 1
        else:
            list[index] -= 1

def first_non_zero_index(list):
    for i in range(0,len(list)):
        if(list[i]!=0):
            return i
    return len(list)

def zero_backwards(list, alph_size):
    index = first_non_zero_index(list)
    if( (list[index]==alph_size-1) and (alph_size%2==0)):
        if( list[index+1]%2==0 ):
            list[index+1] += 1
        else:
```

```
            list[index+1] -= 1
    else:
        if(list[index]%2==0):
            list[index] -= 1
        else:
            list[index] += 1

def go_forwards(list):
    b = True
    for i in range(1,len(list)):
        b ^= ((list[i]%2)!=0)        # b switches only when list[i] is
        ↪  odd.
    return b

def next(list,alph_size):
    if(go_forwards(list)):
        if(list[0]!=alph_size-1):
            list[0] += 1
        else:
            max_forwards(list,alph_size)
    else:
        if(list[0]!=0):
            list[0] -= 1
        else:
            zero_backwards(list,alph_size)
```

Linear time half growing implementation

The next code is the implementation of the linear half growing algorithm described in 3.4. The input tuple is represented as a list inside a class called Word. This class has attributes and methods to check some properties of the list in constant time.

```
class Word:
    def __init__(self,list):
        self.list = list
        self.length = len(list)
        self.largest_numbers = []
        largest = 0
        for x in list:
            if(x>largest):
                largest = x
            self.largest_numbers.append(largest)
        self.last_index=0
        self.last_dir=1

    def change_ele(self, index, add):
        self.last_index=index
        self.last_dir=add
        if(add==1):
            self.list[index] += 1
        else:
            self.list[index] -= 1
        x = self.list[index]
```

```
        if(index>0 and self.largest_numbers[index-1]>x):
            return
        for i in range(index, len(self.largest_numbers)):
            if(self.list[i]>x):
                return
            self.largest_numbers[i]=x

    def is_oom_shape(self,index,m):
        return self.largest_numbers[index]==m and (index==0 or
        ↪  self.largest_numbers[index-1]==0)

    def is_oomo_shape(self,index,m):
        return self.list[index]==0 and self.list[index-1]==m and
        ↪  (index==1 or self.largest_numbers[index-2]==0)

    def get_largest_number(self,index):
        return self.largest_numbers[index]

    def __getitem__(self, index):
        return self.list[index]

    def get_len(self):
        return len(self.list)

def is_even(n):
    return n%2==0


def forwardsHGCube(word,index):
    m = word.get_largest_number(index)
    if index==0:
        word.change_ele(index,1)
    elif index==1:
        if is_even(m):
            if(word[0]>0 and word[1]==m):
                word.change_ele(0,-1)
            else:
                word.change_ele(1,1)
        else:
            if(word[0]==m and word[1]>0):
                word.change_ele(1,-1)
            else:
                word.change_ele(0,1)
    else:
        if is_even(m):
            if word.get_largest_number(index-1)==0:
                word.change_ele(index,1)
            elif word[index]==m:
                backwardsHGCube(word,index-1)
            elif is_even( word[index] ):
                if word.is_oomo_shape(index-1, m):
                    word.change_ele(index,1)
```

```
                else:
                    backwardsHGCube(word,index-1)
            else:
                if word.is_oom_shape(index-1, m):
                    word.change_ele(index,1)
                else:
                    forwardsHGCube(word, index-1)
        else:
            if word.is_oomo_shape( index, m):
                word.change_ele( index-1, 1)
            elif is_even(word[index]):
                if word.is_oom_shape( index-1, m):
                    word.change_ele(index, -1)
                else:
                    backwardsHGCube(word, index-1)
            else:
                if word.is_oomo_shape( index-1, m):
                    word.change_ele(index, -1)
                else:
                    forwardsHGCube(word, index-1)


def backwardsHGCube(word,index):
    m = word.get_largest_number(index)
    if m==0:
        return
    if index==0:
        word.change_ele(index,-1)
    elif index==1:
        if is_even(m):
            if word[1]==0:
                word.change_ele(0,-1)
            elif word[0]==m:
                word.change_ele(1,-1)
            else:
                word.change_ele(0,1)
        else:
            if word[0]==0:
                word.change_ele(1,-1)
            elif word[1]==m:
                word.change_ele(0,-1)
            else:
                word.change_ele(1,1)
    else:
        if is_even(m):
            if word.is_oomo_shape( index, m):
                word.change_ele(index-1,-1)
            elif is_even( word[index] ):
                if word.is_oom_shape( index-1, m):
                    word.change_ele(index, -1)
                else:
```

```
                forwardsHGCube(word, index-1)
            else:
                if word.is_oomo_shape( index-1, m):
                    word.change_ele(index, -1)
                else:
                    backwardsHGCube(word, index-1)
        else:
            if word.get_largest_number(index-1)==0:
                word.change_ele(index, -1)
            elif is_even(word[index]):
                if word.is_oomo_shape( index-1, m):
                    word.change_ele(index, 1)
                else:
                    forwardsHGCube(word, index-1)
            elif word[index]<m and word.is_oom_shape( index-1, m):
                word.change_ele(index, 1)
            else:
                backwardsHGCube(word, index-1)
```

Constant time half growing implementation

The next code is the implementation of the constant time algorithm for the half growing cube. This algorithm is described in 3.5. In this implementation all the auxiliary memory is put inside the class Word that contains the list that represents the input tuple.

```
class Word:
    def __init__(self, size):
        self.values = [0]*size
        self.dir = [False]*size
        self.dir[0] = True
        self.max = [0]*size
        self.max[0] = -1
        self.d_max = [True]*size
        self.upgrade = [True]*size
        self.index_ref = list(range(size))
        self.index = size-1

    def print(self):
        print(self.values)

    def print_all(self):
        print("V: ",self.values)
        print("D: ",self.dir)
        print("M: ",self.max)
        print("D_M: ",self.d_max)
        print("U: ",self.upgrade)
        print("index ref: ",self.index_ref)
        print("next index: ",self.index)
        print()

    # In this function we assume we just hit a ``border" (0 or M[i]) at
    ↪  index i
    # and we are trying to decide if next time we get to index i,
```

```
# we should refer to a higher index
# or change the M[i]
def should_upgrade(self,i):
    if( i==len(self.values)-1 ):
        return True
    if( self.values[i+1]==self.max[i+1] ):
        if  ( not (self.max[i]==0 and  not self.d_max[i])
        and  not (self.max[i]==self.max[i+1] and self.d_max[i]) ):
            return True
        else:
            self.d_max[i] = not self.d_max[i]
            return False
    return False

def next(self):
    i = self.index
    if( i==0 or i==1 ):

        #A value at 0 or 1 gets changed
        if( not self.upgrade[1] ):
            if( self.dir[i] ):   self.values[i] += 1
            else:                self.values[i] -= 1
            if( self.values[i]==self.max[1] ):
                self.dir[i] = not self.dir[i]
                self.index = not i
            elif( self.values[i]==0 ):
                self.dir[i] = not self.dir[i]
                if( len(self.values)==2 or
                ↪  (self.values[2]==self.max[2] and
                not ( self.max[1]==self.max[2] and self.d_max[1] ) )
                ↪  ):
                    self.upgrade[1] = True
                else:
                    if(self.values[2]==self.max[2]):
                        self.d_max[1] = not self.d_max[1]
                    self.index = self.index_ref[2]
                    self.index_ref[2] = 2
        # both a value at 0 or 1 gets changed and the max element of
        ↪  0 and 1 gets changed
        else:
            self.upgrade[1] = False
            if( self.d_max[1] ):    self.max[1] += 1
            else:                   self.max[1] -= 1

            if( self.values[1]>=self.values[0] ):
                self.values[1] = self.max[1]
                if( self.max[1]==0 ):
                    self.index=self.index_ref[2]
                    self.index_ref[2]=2
                    self.upgrade[1] = True
                    if(self.values[2]==self.max[2]):
                        self.d_max[1] = not self.d_max[1]
```

```
            else:
                self.index = 0
        else:
            self.values[0] = self.max[1]
            self.index = 1
        if( self.max[1]==0 ):
            self.d_max


else:
    #This is the situation where we just change the value at i
    if(  not self.upgrade[i] ):
        if(self.dir[i]):    self.values[i] += 1
        else:                   self.values[i] -= 1

        #We reached a ``border" for i
        if( self.values[i]==0 or self.values[i]==self.max[i] ):
            if( self.should_upgrade(i) ):
                self.upgrade[i]=True
            else:
                self.index_ref[i]=self.index_ref[i+1]
                self.index_ref[i+1]=i+1

            self.dir[i] = not self.dir[i]

        if( self.values[i-1]==0 ): self.index = abs(i-3)
        else:                    self.index = i-2

    #This is the situation where we also change the max value at
    ↪  i
    else:
        self.upgrade[i] = False
        if( self.d_max[i] ): self.max[i] += 1
        else:           self.max[i] -= 1
        # shape 0..0m0
        if( self.values[i-1] !=0 ):
            self.max[i-1] = self.max[i]
            self.values[i-1] = self.max[i-1]
            self.index = i-2
        # shape 0..0m
        else:
            self.values[i] = self.max[i]
            if(self.max[i]==0):
                self.upgrade[i]=True
                self.index=self.index_ref[i+1]
                self.index_ref[i+1]= i+1
                self.d_max[i] = not self.d_max[i]
            else:
                self.index=i-1
```

Linear time fully growing implementation

The last code is the implementation of the linear algorithm described in 4.4. Once again the tuple is represented as a list inside a class.

```
class Word:

    def __init__(self,list):
        self.list = list
        self.length = len(list)
        self.largest_numbers = []
        m= abs(list[0])
        for x in list:
            if (abs(x)>m):
                m=abs(x)
            self.largest_numbers.append(m)
        j=1
        while( j<len(list) and list[j] == list[1] ):
            j += 1
        self.flatShapeMax = j-1 #The flatShapeMax is the index j such
        ↪  that 1<=i<=j => list[i]=list[1]

    def changeDigit(self, index, plusOne):
        if( plusOne ):
            self.list[index] += 1
        else:
            self.list[index] -= 1
        newm = abs(self.list[index])
        if(index==0 or self.largest_numbers[index-1]<=newm):
            index2 = index
            while( index2< self.length and abs(self.list[index2]) <=
            ↪  newm ):
                self.largest_numbers[index2] = newm
                index2 += 1

        if( index>1 and self.flatShapeMax >= index ):
            self.flatShapeMax = index-1
        elif( self.flatShapeMax == index-1 or index==1 ):
            index2=index
            while( index2 < self.length and self.list[index2] ==
            ↪  self.list[1] ):
                index2 += 1
            self.flatShapeMax = index2-1

    def max(self, index):
        return self.largest_numbers[index]

    def snakeShape(self, index, m):
        return self.list[0]==m and self.list[1]==m-1 and
        ↪  self.flatShapeMax>=index

    def snakeShapeLowTail(self, index, m):
        if(index==1):
```

```
            return self.list[0]==m and self.list[1]==m-2
        else:
            return self.list[0]==m and self.list[1]==m-1 and
            ↪  self.list[index-1]==m-2 and self.list[index]==m-1 and
            ↪  self.flatShapeMax==index-2

    def allM(self, index, m):
        return self.list[0]==m and self.list[1]==m and
        ↪  self.flatShapeMax>=index

    def all0(self, index):
        return self.list[0]==0 and self.list[1]==0 and
        ↪  self.flatShapeMax>=index

    def inOuterLayer(self,index):
        m = self.largest_numbers[index]
        return abs(self.list[index-1])==m or abs(self.list[index])==m

    def even(self,index):
        return (self.list[index-1]+self.list[index]) % 2 == 0

    def newPrint(self):
        print(self.list, self.largest_numbers, self.flatShapeMax)



def snailPath( word, index ):
    list = word.list
    m = max(abs(list[index]),abs(list[index-1]))
    if(list[index]==m):
        word.changeDigit(index-1,True)
    elif(list[index-1]==-m):
        word.changeDigit(index,True)
    elif(list[index]==-m):
        word.changeDigit(index-1,False)
    else:
        word.changeDigit(index,False)

def snailPathBack( word, index):
    list = word.list
    m = max( abs(list[index]), abs(list[index-1]) )
    if(list[index-1]==m and list[index]<m-1):
        word.changeDigit(index,True)
    elif(list[index]==-m):
        word.changeDigit(index-1,True)
    elif(list[index-1]==-m):
        word.changeDigit(index,False)
    else:
        word.changeDigit(index-1,False)
```

```
def forwardsFGCube(word, index):
    if(index==1):
        snailPath(word, index)
    else:
        m = word.max(index)
        if (word.allM(index, m)):
            word.changeDigit(0,True)
        elif( not word.inOuterLayer(index) and word.snakeShape(index-2,
        ↪  m) and  not (word.list[index-1]==0 and word.list[index]==0)
        ↪  ):
            snailPathBack(word, index)
        elif(word.even(index)):
            if( word.allM(index-2, m) ):
                snailPath(word, index)
            else:
                forwardsFGCube(word, index-2)
        else:
            if( word.all0(index-2) ):
                snailPath(word, index)
            elif(  not word.inOuterLayer(index) and
            ↪  word.snakeShapeLowTail(index-2, m) ):
                snailPath(word, index)
            else:
                backwardsFGCube(word, index-2)


def backwardsFGCube(word,index):
    if(index==1):
        snailPathBack(word, index)
    else:
        m = word.max(index)
        if(word.snakeShape(index,m)):
            word.changeDigit(0, False)
        elif( not word.inOuterLayer(index) and
        ↪  word.snakeShape(index-2,m)):
            snailPath(word, index)
        elif(  not word.even(index) ):
            if( word.allM(index-2,m) ):
                snailPathBack(word, index)
            else:
                forwardsFGCube(word, index-2)
        else:
            if( word.all0(index-2) ):
                snailPathBack(word, index)
            elif( not word.inOuterLayer(index) and
            ↪  word.snakeShapeLowTail(index-2,m) and (
            ↪  word.list[index]!=0 or word.list[index-1]!=0) ):
                snailPathBack(word, index)
            else:
                backwardsFGCube(word, index-2)
```