1 Blind Counting

The usefulness of the topic is in the coding theory.

Historically, Gray's (?) code was used to encode and transmit strings.

Suppose you have two sequences: 0011 and 0100. Since these two numbers differ in two digits, trivial encoding is not going to be efficient. The problem is that transmission can be dissynchronised, which would increase the likelihood of errors.

Modern ethernet lines work most efficiently if the message contains roughly the same number of 0s and 1s. Another requirement is that there should not be many consecutive digits.

To counteract this problem, error-correcting codes are used.

How can we formalise the operations?

1.1 Gray's Code

By definition, Gray's code is the method of writing numbers from 0 to $2^n - 1$ using n bits such that addition of 1 would change only one symbol in the encoding.

We can easily construct such a code for 4 bits. Can we do it in general?

One of the ways to approach this problem is to look at the hypercubes. For 4 bits, a 3-dimensional cube is enough, and by traversing the vertices of the cube sequentially in S-shaped paths we can obtain the code. Imagining the hypercubes in higher dimensions, however, requires more work.

We can, for example, give an algorithm to construct Gray's code. Fir this, let n denote the position of the digit in the encoded number. For example, if $n = 1$, it changes from 0, to 1, to 0.

If $n \neq 1$, write 0 for each code for n, and then write 1 to the left of the codes for n in the reverse order.

This construction is called mirrored Gray's code, and it has its own disadvantages. For example, its construction requires writing out all the possible binary encodings at first, and then trying to execute the algorithm, which is not efficient memory-wise.

Suppose we have a standard binary representation of a number, and let $k < 2^{n-1}$ be some integer. We can introduce the following operation: we will write 0 before the code for k of length $n - 1$, and we will write 1 for $(2^n - 1) - k$ of length $n - 1$.

On the other hand, we can try the other operation. If $k \geq 2^{n-1}$, we will write 1 to the left of the code for $\overline{k}$.

We can try introduce some rules. To do this, we will again look at the binary representation of an encoded number. The first symbol is copied to the code as is. If the next digit in the encoded number is the same as the previous one, we will write 0, otherwise, write 1 to the code.

This seems to be the easiest method we have seen so far. But what about the inverse problem? If we have Gray's code, how can we obtain the encoded number?

One way is to track the parity of the sum of the digits from left to right so far. The first number in the code is decoded as is. Then we will write the same digits as in the code until we meet 1, which flips the parity of the sum, and thus we will flip the digit from the code to the decoding.

1.2 Addition and Gray's Code

How can we add 1 to the encoded number?

The obvious method is just decode the input, add 1, and encode it again. Can we do better?

What can happen when we add 1 to the number in binary?

We can add 1 to 0, and that's it. We can also add 1 to 1 in the least significant position, in which case we obtain a carry. We can also add 1 to the number which is an n-bit repunit of 1, and thus we would obtain the overflow.

We can try the following rules.

If the number of 1's in the encoding is even, we will flip the last digit. Otherwise, we will flip the digit to the left from the rightmost digit. If we obtain the overflow, we put zero instead of the rightmost digit.

Is this the only way to obtain Gray's code?

Note that, if we write Gray's code in rows, all the columns obtained are unique, and thus even if we interchange the columns, we can deduce its original position, even if it was flipped.

Is there Gray's code such that it would not require the knowledge of parity or the entire code to add 1?

First, we need to formalise the notion of not reading the entire code.

Suppose that we build a tree of reading operations.

At the root, we state the bit which we read, and then branch out depending on the bit we have read. At some point, we reach leaves, and then decide on what digit we put there.

On the other hand, we can define a program which would read the code. However, with just an algorithm, it becomes difficult to look at all the possible options we have.