Recitation 14 – Error detection and correction

Index code

data	EC

We want to send *data* of length $k = 2^m - 1$ bits.
EC = bitwise XOR over all active indices (containing '1') in *data* (indices start at 1)

What will be the length of *EC*?
The length of *EC* is be equal to the number of bits required to represent an index $\leq k$. In order to write k in binary the number of bits required is $\lfloor \log_2 k + 1 \rfloor$.
For $k = 2^m - 1$, we get: $|EC| = \lfloor \log_2 (2^m - 1) + 1 \rfloor = m$.

We saw in class *d*=2.

Improvement 1: transmit *EC* twice.

Now *d*=3 (why?)

data	EC1	EC2

Decoding algorithm, given that we expect no more than 1 error:

```
decode (message):
1. compute EC ' from first k bits (data)
2. if EC ' = EC1 or EC ' = EC2        #if both hold then 0 errors
3.      return message[:k]            # no error in data
4. else:                              # EC1 = EC2, single error in data
5.      i = EC ' ⊕ EC1                #or EC2, doesn't matter. Index of error.
6.      return message[:k] with index i switched
```

How would we interpret different scenarios of 2 errors?
Assume p is small (so we always prefer an interpretation with fewer errors).

- 2 errors in data: we would think it's 1 error. We would "fix" and insert a third error!

  example: $0\overline{0}10\overline{0}10\underline{010010}$

  2+5 = 010+101 = 111

  we'd conclude the single error is at 111 = 7

- 2 errors at EC1 and EC2:
  - if at the same bit of EC1 and EC2, we'd conclude 1 error in data.
  - if at different bits of EC1 and EC2, we'll know there is >1 error.
- 2 errors at e.g. EC1: we'll know this (the alternative is 3 errors: 1 at data, 2 at EC2 – which is less probable).
- 1 error in data and 1 at e.g. EC1 – the 2 options are possible (detecting or "fixing").

Note that we can write an algorithm that corrects up to 1 error, or a different algorithm that detects up to 2 errors. But we cannot have an algorithm that does both, since as we saw in some cases we cannot distinguish between 1 and 2 errors.

Improvement 2: add parity bit at the end.

data	*EC1*	*EC2*	*p*

Now d=4 (why?)

Explanation: earlier d was 3. This means that the closest words were 3 bits apart. So such words had different parity (3 is odd). Therefore with the addition of a parity bit such words will be 4 bits apart. Note that if d was even, a parity bit would not change it.

Now we can detect up to 3 errors and fix up to 1 error.

Claim: we can now distinguish between scenarios of 1 and 2 errors (which we couldn't before).

Proof: parity bit will notify error if and only if number of errors is odd (no matter where they are).

Not done in class:

A summary of the possible interpretations for various numbers of errors:

#errors	EC'= EC1=EC2	parity
0	V	V
1	X or V*	X
2	X	V
3	X or V**	X

V - indicates this part of the error correction code does not notify error.
X - indicates this part notifies an error.
X or V – some cases are erroneous and some are not.

* if error was in parity bit
** try to find a case in which this happens