Lecture 7
Information Redundancy

ECE 422: Reliable and Secure Systems Design

UNIVERSITY OF ALBERTA

Instructor: An Ran Chen
Term: 2024 Winter

Schedule for today

- Key concepts from last class
- Code, codespace, codeword, word
- Hamming distance
- Code distance in error detection/correction
- Repetition codes
- Parity codes
- Next class: Hamming codes

Fault localization

Fault localization techniques have been proposed to assist in locating and understanding the root causes of faults.

Why? Fault localization as a debugging technique to maintain the system:

- Pinpoint the location to fix in the code
- Recover fast from bugs, and reduce its impact on the users
- Reduce manual debugging efforts, more time for new feature

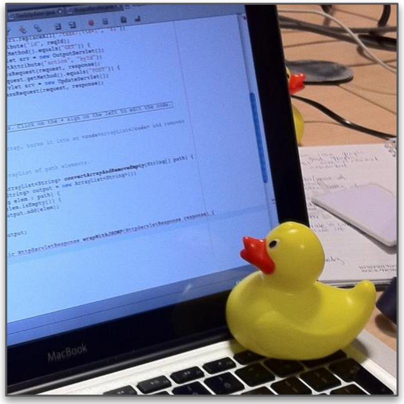
Rubber duck debugging

Introduced by Dave Thomas and Andy Hunt (credited Andrew Hunt) in “The Pragmatic Programmer” book.

From a [lists.ethernal.org post]() by Andy:

- Beg, borrow, steal, buy, fabricate, or otherwise obtain a rubber duck.
- Place rubber duck on desk and inform it you are just going to go over some code with it, if that’s all right.
- Explain to the duck what your code does line by line.
- At some point, you will realize what you are telling to the duck is not in fact what you are actually doing.

If you don’t have a rubber duck, a co-worker works too.



Fault localization

Fault localization present automated techniques for locating the faults in the source code.

There are many families of fault localization techniques:

- Spectrum-based techniques
- Information retrieval-based techniques
- Mutation-based techniques (not covered in class)
- Historical-based techniques (not covered in class)
- and more …

Redundancy and fault tolerance

Main goal of designing reliable and secure systems:

- Maintain availability
  - Uptime
  - The system is up and running
- Maintain reliability
  - Mean Time Between Failure
  - The system continues to function without failure
- No system/software/hardware failure
  - Fault tolerance

Redundancy

To achieve fault tolerance, redundancy is usually used.

Types of redundancy:

- Software redundancy (e.g., N-version programming)
- Information redundancy (e.g., parity bit)
- Hardware redundancy (e.g., replication of the processor)
- Time redundancy (e.g., extra time)

For fault tolerant systems, there is often a combination of these redundancies.

Information redundancy

Information redundancy tolerate faults by adding information to the original data.

- Tolerate faults by means of coding
- Avoid unwanted information changes
- E.g., information loss during data storage or transmission

Used in data applications to ensure data integrity:

- Communication systems
- Storage devices
- Computer networks

The term “coding” is used in the context of communication and data storage

Information redundancy

There are two commons forms of information redundancy:

- Error detection code
- Error correction code

to ensure the accuracy, integrity and reliability of transmitted or stored data.

Code selection depends on the nature of the faults:

- Rate
- Consequences
- Overhead in encoding and decoding

Code

- Code of length n is a set of n-tuples satisfying some well-defined set of rules.

0000	0
0001	1
0010	2
…	
1001	9

Example: binary-coded decimal (BCD)

- A BCD with code length 4 is a set of 4-tuples for each decimal digit, from 0 to 9.
- Since there are 10 unique decimal digits, the codespace only includes the first 10 binary 4-tuples, from 0000 to 1001.
- Code: {0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001}
- 1010 is not part of the code, the decimal digit 10 is represented by 0001 0000.

Code

- A codeword is an element of the code satisfying the rules of the code.
- A word is an n-tuple not satisfying the rules of the code.
- For example:

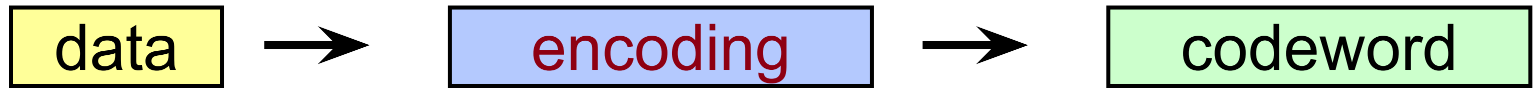
  In binary-coded decimal, 1010 is invalid, thus a word and not a codeword.

  In binary code, 2024 is invalid, thus a word and not a codeword.

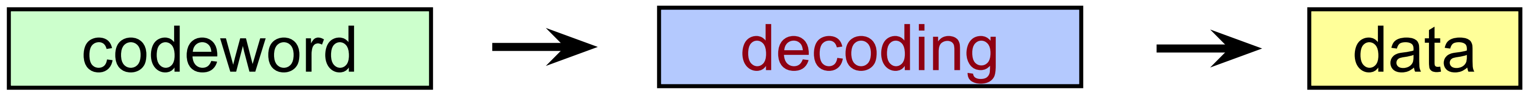
- To make code error detection/correction possible, codewords should be a subset of all possible 2^n binary tuples in the code.
- The number of codewords in a code C is called the size of C.

Encoding and decoding

- Encoding: transforms data into code word



- Decoding: transforms code word back to data



- There is a difference in length between the codeword and data.
- This difference gives us the number of check bits which are required to make the encoding (for separable codes).

Encoding and decoding

There are 2 possible scenario when an error happens during encoding/decoding:

1. Correct codeword → another codeword
2. Correct codeword → word

Error

Error happens when the data is corrupted in transmission or storage.

- E.g., bad spots on disks, scratches on DVD or CD, electrical interference

The most common error is a bit flip during a stream of data

- Bit flip: a 1 becomes a 0, or a 0 becomes a 1
- E.g., the sender passes 1010, the receiver gets 1000 instead.

It is also possible for a bit to get deleted or for an extra bit to be inserted.

Error

Common error patterns:

- Single-bit error = one bit has been corrupted
- Burst error = more than one bits have been changed

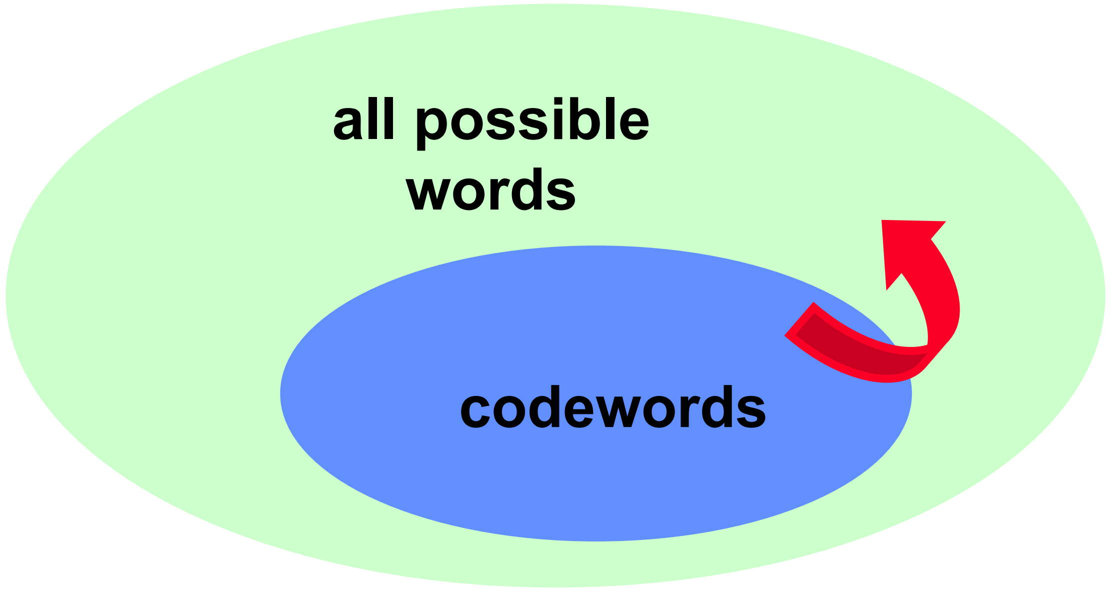
Error detecting/correcting code

Error detecting/correcting code is used to detect and correct corrupted bits during transmission.

- Characterized by the number of bits that can be detected/corrected
  - E.g., double-bit detecting code can detect two single-bit errors
  - E.g., single-bit correcting code can correct one single-bit error

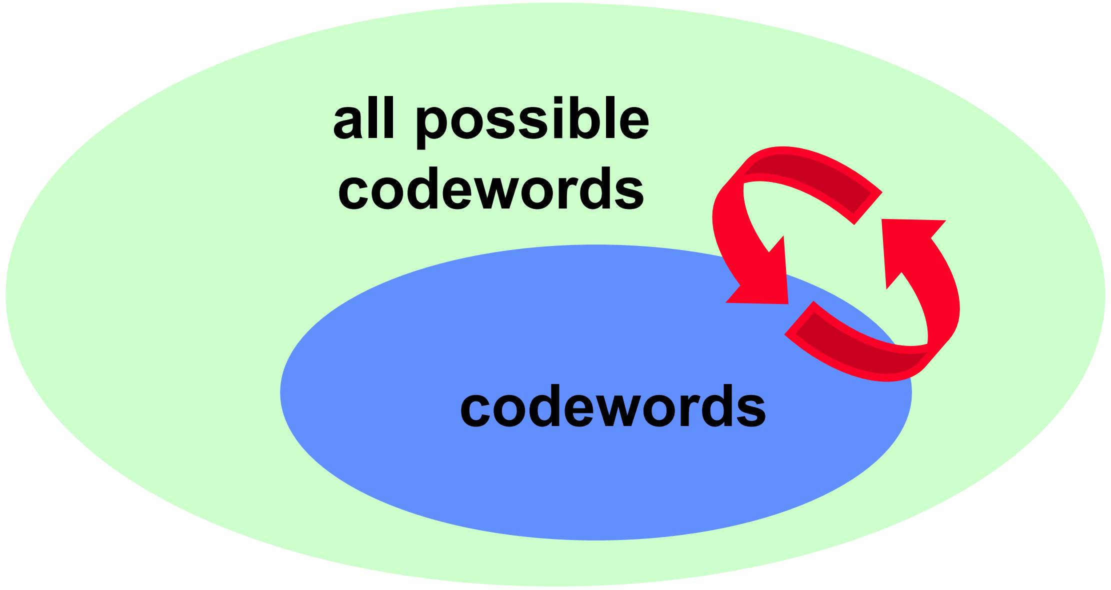
Error detecting code

Basic intuition: we define a code so that errors introduced in a codeword force it to lie outside the range of codewords.



Error correcting code

Basic intuition: we define a code so that it is possible to determine the correct codeword.



Hamming distance

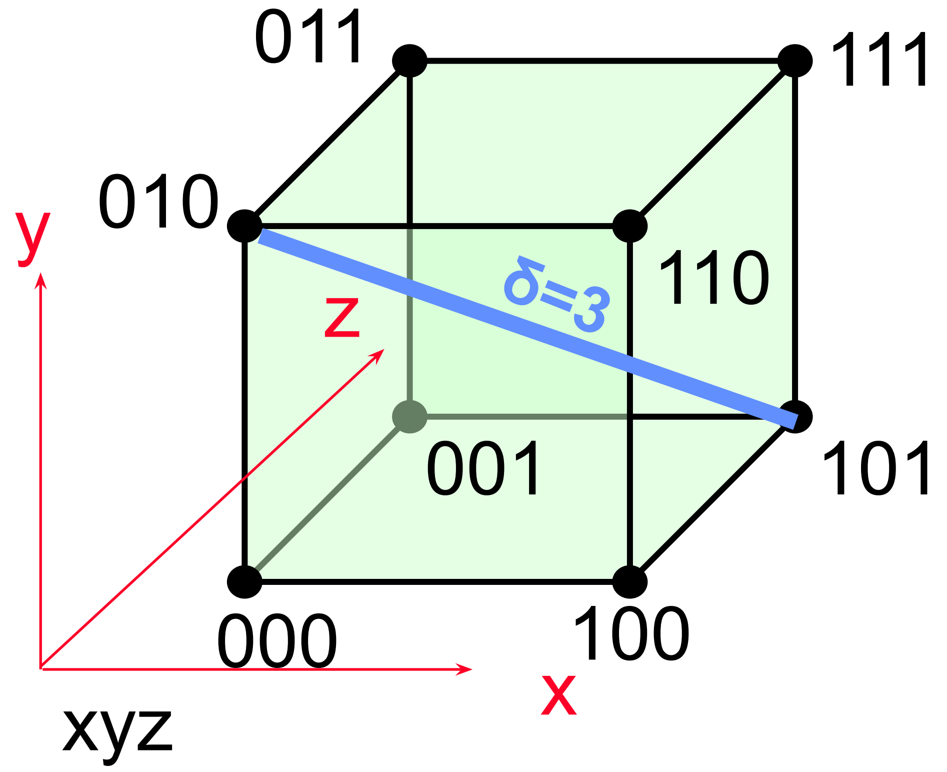
Hamming distance measures the number of bit positions in which two n-tuples differ.

x 0000

y 0101

$\delta(x,y) = 2$

Code in 3-dimensional space



Hamming distance has 3 properties:

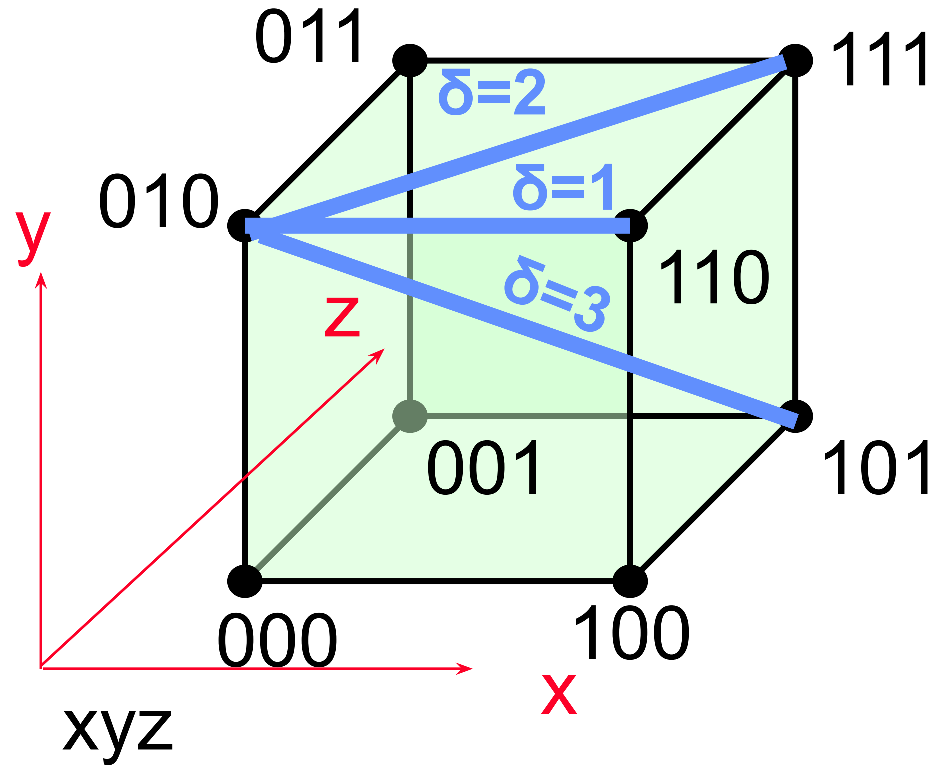
- Reflexivity:

  $H_d(x, y) = 0$ if and only if $x = y$
- Symmetry

  $H_d(x, y) = H_d(y, x)$
- Triangle inequality

  $H_d(x, y) + H_d(y, z) \geq H_d(x, z)$

Code in 3-dimensional space



Hamming distance has 3 properties:

- Reflexivity:

  $H_d(x, y) = 0$ if and only if $x = y$

  $H_d(010, 010) = 0$

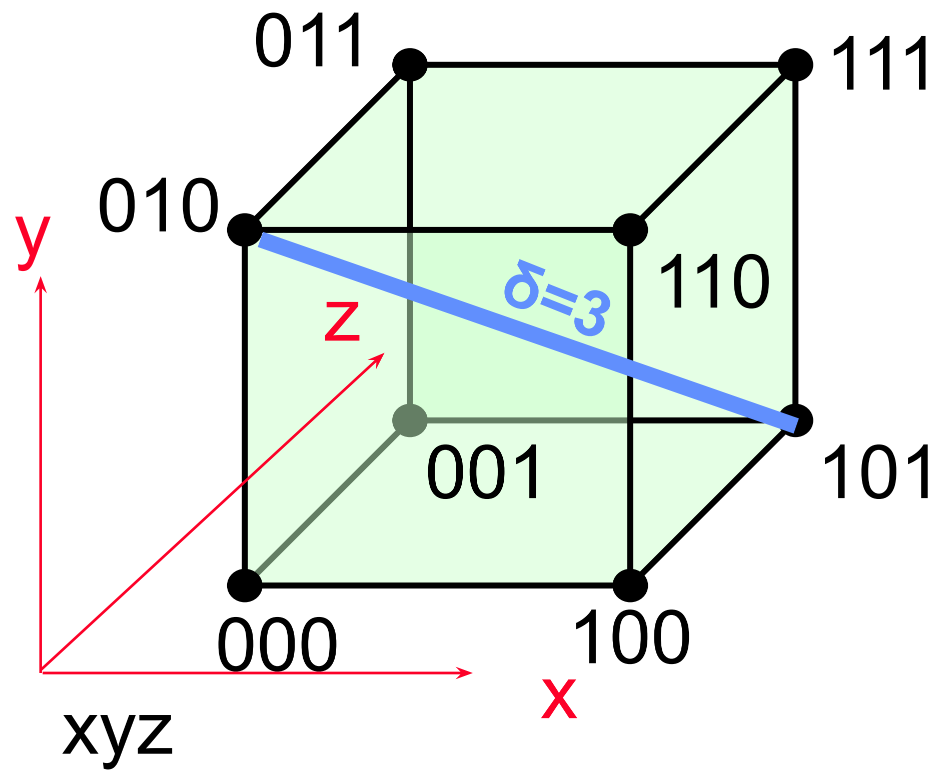
  $H_d(010, 110) = 1$

  $H_d(010, 111) = 2$

  $H_d(010, 101) = 3$

- Symmetry
- Triangle inequality

Code in 3-dimensional space



Hamming distance has 3 properties:

- Reflexivity:

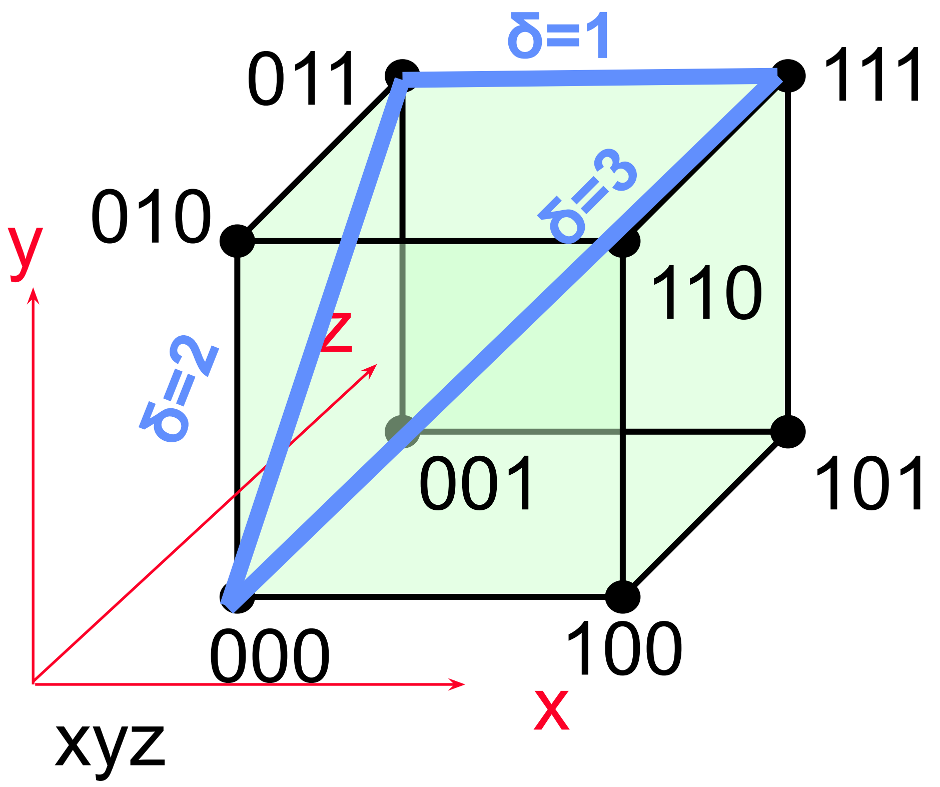
  $H_d(x, y) = 0$ if and only if $x = y$
- Symmetry

  $H_d(x, y) = H_d\,(y, x)$

  $H_d\,(101, 010) = 3 = H_d(010, 101)$
- Triangle inequality

  $H_d(x, y) + H_d\,(y, z) \geq H_d\,(x, z)$

Code in 3-dimensional space



Hamming distance has 3 properties:

- Reflexivity:

  $H_d(x, y) = 0$ if and only if $x = y$
- Symmetry

  $H_d(x, y) = H_d(y, x)$
- Triangle inequality

  $H_d(x, y) + H_d(y, z) \geq H_d(x, z)$

  $H_d(000, 011) + H_d(011, 111)$

  $= 2 + 1 = 3 \geq H_d(000, 111)$

Code distance

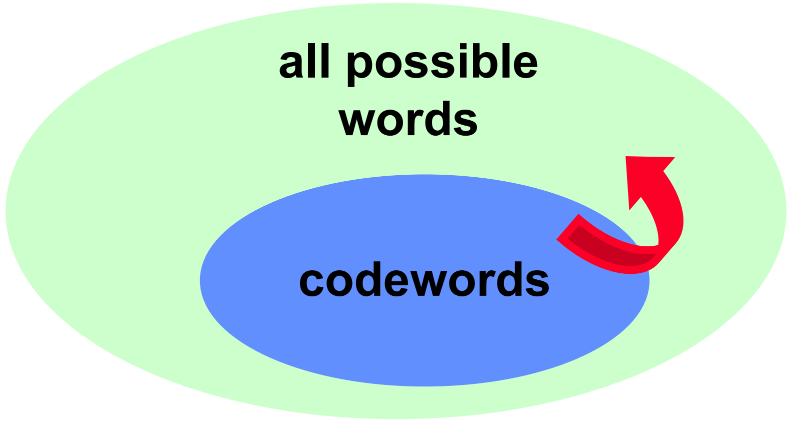
Code distance, denoted as C_d, is the minimum Hamming distance between any two distinct pairs of codewords of the code.

- Determines the error-detecting and error-correcting capabilities of a code.

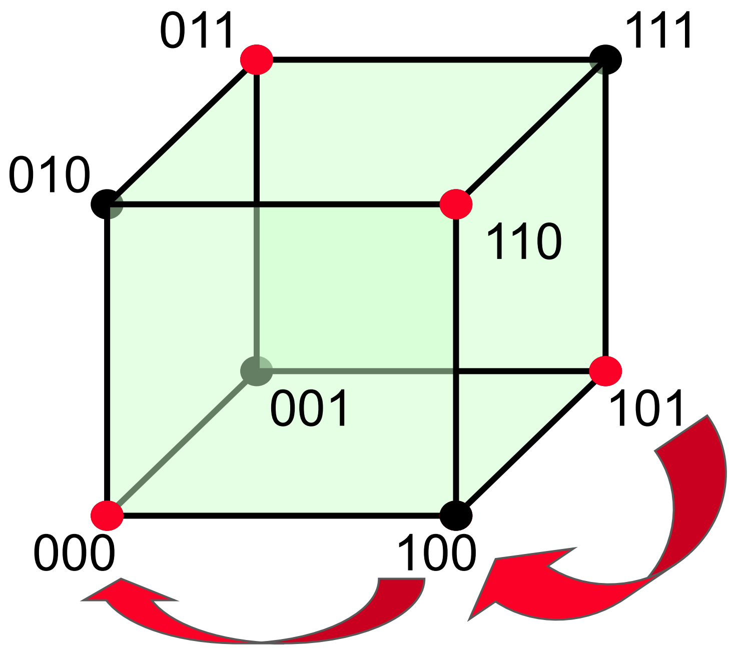
Code distance in detection

To design a code that can detect *d* bit errors, the code distance for the codewords must be larger or equal to *d+1*.

- This ensures that no errors in *d* or fewer bits can change one (valid) codeword into another (valid) codeword.



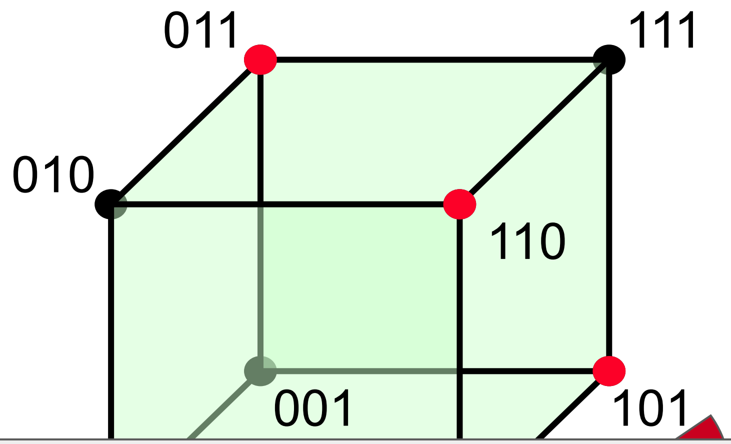
Code distance in detection



Example:

- Consider the code {000, 011, 101, 110}.
- The code distance is 2.
- Suppose an error has occurred in the first bit of the codeword 000.
- The resulting word 100 has a distance of 1 from the affected codeword 000.
- Since all codeword have a distance of 2 from each other, we detect 100 to be a single-bit error.

Code distance in detection



To detect *d* bit errors, the code distance for the codewords must be larger or equal to *d+1*.

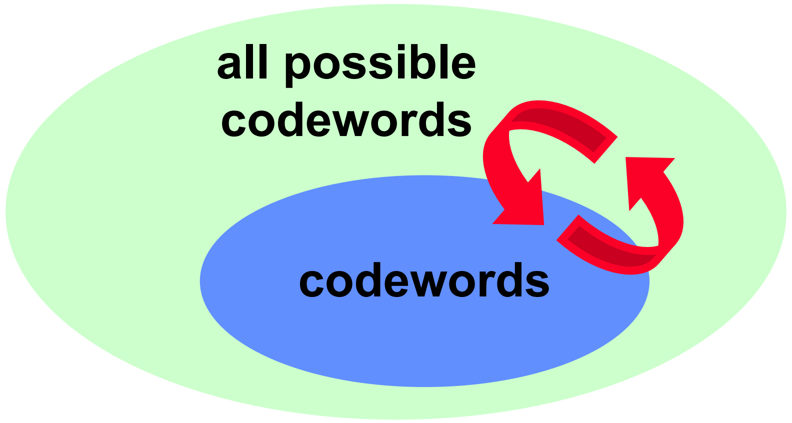
Example:

- Consider the code {000, 011, 101, 110}.
- The code distance is 2.
- Suppose an error has occurred in the first bit of the codeword 000.
- The resulting word 100 has a distance of 1 from the affected codeword 000.
- Since all codeword have a distance of 2 from each other, we detect 100 to be a single-bit error.

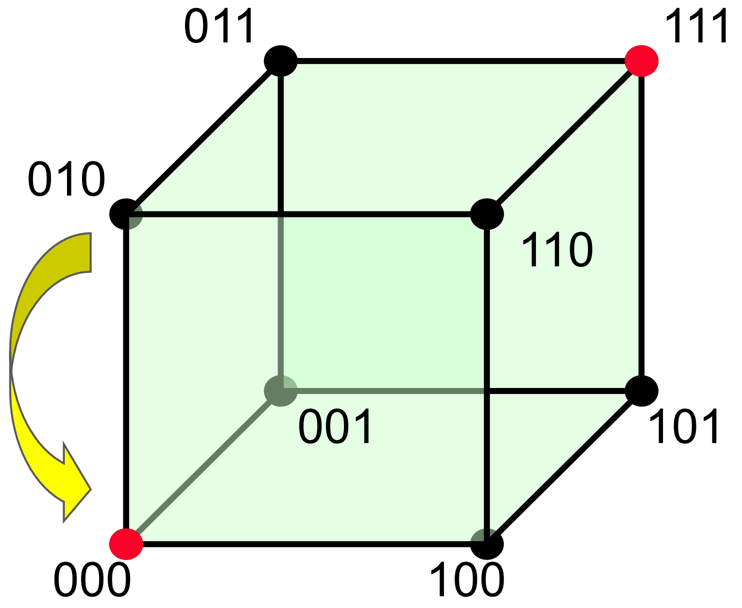
Code distance in correction

To design a code that can correct *d* bit errors, the code distance for the codewords must be larger or equal to *2d+1*.

- This puts the valid codewords so far apart that even after d bit errors, it is still less than half the distance to another valid codeword.



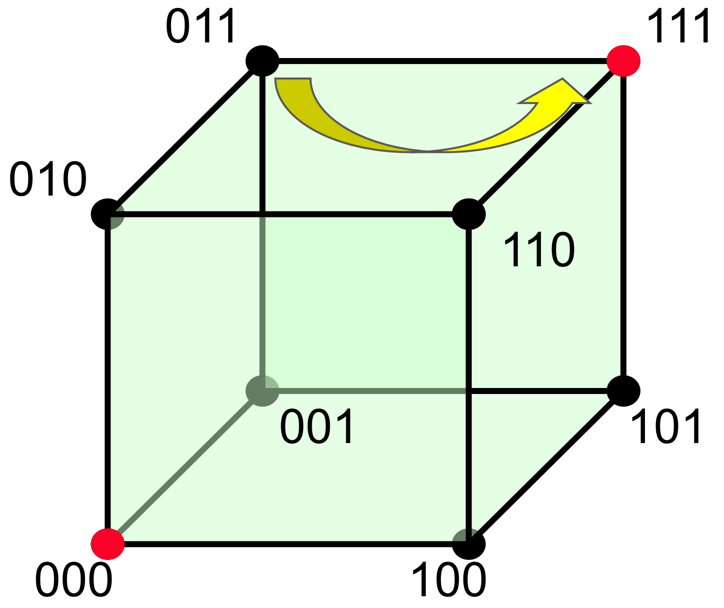
Code distance in correction



Example:

- Consider the code {000, 111}.
- The code distance is 3.
- Suppose a single-bit error has occurred in the second bit of the codeword 000.
- The resulting word 010 has a distance of 1 from the affected codeword 000.
- We correct 010 to the codeword 000, since it is the closest codeword.

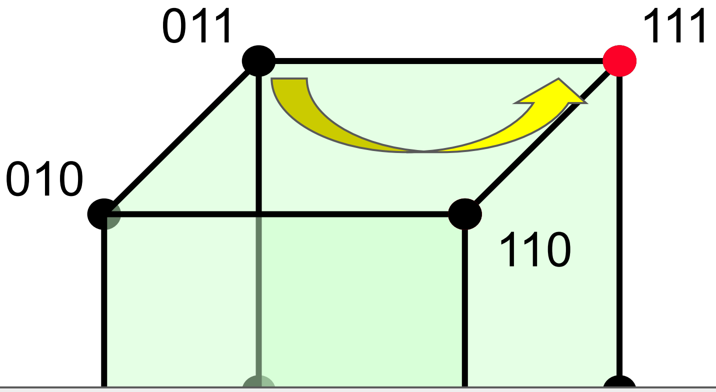
Code distance in correction



Example:

- Consider the code {000, 111}.
- The code distance is 3.
- Suppose a double-bit error has occurred in the second and third bit of the codeword 000.
- The resulting word 011 has a distance of 2 from the affected codeword 000.
- We cannot correct the error because the word 011 is the closer to the codeword 111.

Code distance in correction



To correct *d* bit errors, the code distance for the codewords must be larger or equal to *2d+1*.

Example:

- Consider the code {000, 111}.
- The code distance is 3.
- Suppose a double-bit error has occurred in the second and third bit of the codeword 000.
- The resulting word 011 has a distance of 2 from the affected codeword 000.
- We cannot correct the error because the word 011 is the closer to the codeword 111.

Code distance

Code distance is the minimum Hamming distance between any two distinct pairs of codewords of the code.

- Determines the error-detecting and error-correcting capabilities of a code.

Error detecting code

Suppose $C_d = 2$, code detects all single-bit errors.

Code: {00, 11}

Invalid words: 01 and 10

Error correcting code

Suppose $C_d = 3$, code corrects all single-bit errors.

Code: {000, 111}

Invalid words: 001, 010, 100,
101, 011, 110

Separable/non-separable code

- Separable code
  - Codeword = data + check bits
  - E.g., Parity: 11011 = 1101 + 1
- Non-separable code
  - Codeword = data mixed with check bits
  - E.g., Cyclic: 1010001 -> 1101
- Decoding process is much easier for separable codes (remove check bits)

Information rate

- The information rate of the code is the ratio k/n, where
  - k is the number of data bits
  - n is the number of data + check bits
- Example: a code obtained by repeating data three times has the information rate ⅓.
  - In other words, 3 bits of message for each bit of data
  - Bigger code rates, stronger codes

Types of codes

- Repetition codes
- Parity codes
- Hamming codes
- Reed-Solomon codes
- Berger codes
- … and more

Repetition codes

Repetition codes is used to detect errors by repeating the data several times.

- Raises the probability of the correct bit being uncorrupted

For example:

- Code {1 1 0} can be sent as {1 1 1 1 0 0}
- If the bits are not matching in each group of two, then an error is detected

However, two repetitions does not allow us to determine what the original values was.

Repetition codes

We can repeat each bit three times:

- Code {1 1 0} can be sent as {1 1 1 1 1 1 0 0 0}
- If the bits are not matching in each group of three, then an error is detected
- We can assume the majority results for each group of three is likely to be the original code
- However, this solution comes with high overhead
  - E.g., if we transfer 8 bits, it will require 24 bits, 200% increase in data size

Parity codes

Parity codes detect errors using parity bits, decreasing the number of bits sent.

Parity/check bits are used to determine if the total 1s in the sent code is even or odd.

- Single-bit parity code: can determine an odd number of errors.
- Multiple parity bits code: can determine errors regardless of its number, and locate where they occur.

Single-bit parity codes

Single-bit parity codes add an extra bit (a parity bit) to data so that resulting codeword has either even or odd number of 1s.

- Even parity: even number of 1s
- Odd parity: odd number of 1s

If a single bit is flipped, the received data will have the wrong parity, then we know there is an error in the transmission or storage.

The parity bit can either be added to the front or the back, as long as the order is consistent.

Single-bit parity codes

Example 1: odd parity code

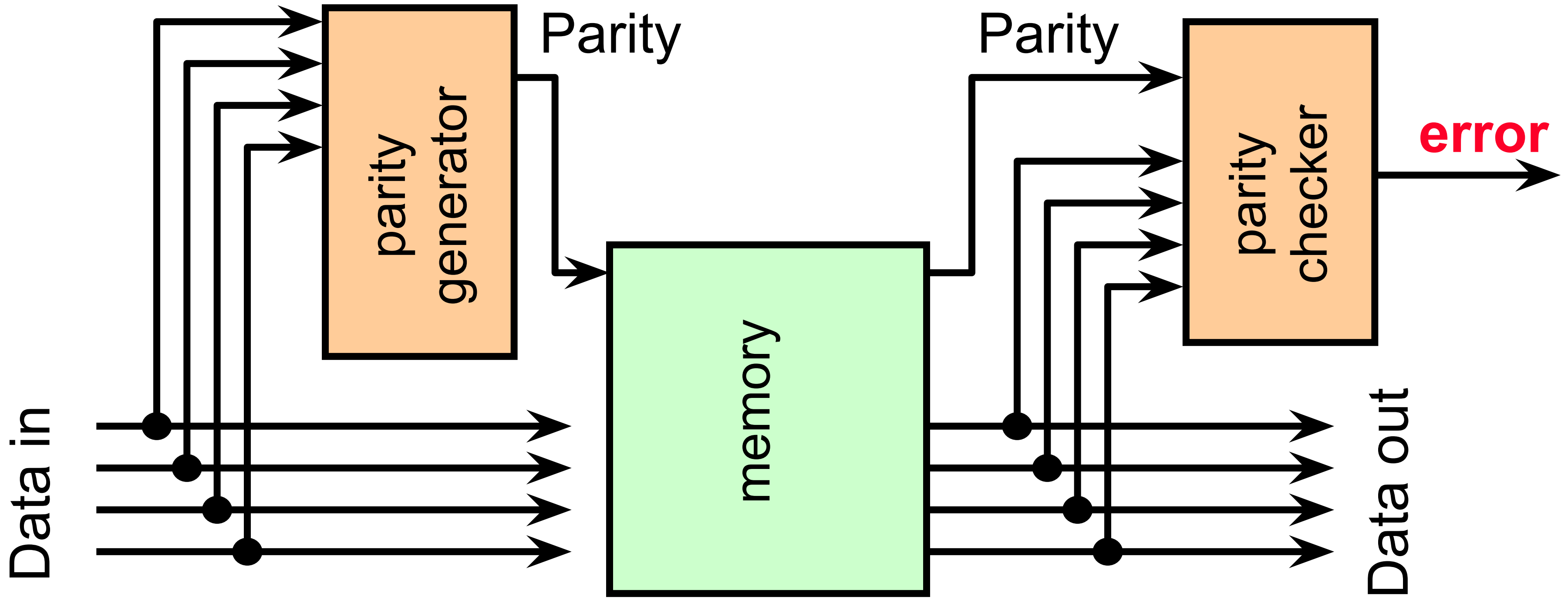
- Suppose the data “0011” is encoded using an odd parity code
- Because the data contain an even number of 1s
- We add a parity bit of “1” to make the number of 1s odd
- The resulting codeword is “0011 1”

Single-bit parity codes

Example 2: even parity code

- Suppose the data “0011” is encoded using an even parity code
- Because the data already contain an even number of 1s
- The resulting codeword is “0011 0”

Memory with built-in parity bits



Challenges with single-bit parity codes

Challenge 1: Multiple-bit errors which affect even number of bits cannot be detected

Scenario 1: with odd number of bits affected

- Consider the codeword “00111” with of a 5-bit odd parity code
- If an error affects the first bit, we receive the word “10111”
- “10111” has an even number of 1s, we detect the error.

Challenges with single-bit parity codes

Challenge 1: Multiple-bit errors which affect even number of bits cannot be detected

Scenario 2: with even number of bits affected

- Consider the codeword “00111” with of a 5-bit odd parity code
- If an error affects the first two bits, we receive the word “11111”
- “11111” has an odd number of 1s, the error is not detected.

Challenges with single-bit parity codes

Challenge 2: Limited error detection

- Can sometimes result in false positive (e.g., two bit flips)

Challenge 3: Unclear detection

- Can not tell which bit is corrupted

Schedule for today

- Key concepts from last class
- Code, codespace, codeword, word
- Hamming distance
- Code distance in error detection/correction
- Repetition codes
- Parity codes
- Next class: Hamming codes