

Hamming Distance, Error Detecting and Error Correcting

Hamming distance is the number of features in which two objects differ. For example, the number of bit positions in which one string has '1' and the other string has '0' and vice versa:

```
String_1: 1001010111010011110100111111100101
String_2: 0101010110010011010100111011110011
          110000000010000000100000000100010110
```

In the last line, '1' are displayed where the bit positions are different.

The total number of these '1' is Hamming distance; in our case, it is $d=8$.

If the Hamming distance is d , we need d bit errors to convert one word into the other.

With m -bit words, we have 2^m different legal bit patterns (e.g., numbers). Altogether, if we use r additional redundant bits, we have $m+r=n$ bits, however, only 2^m of the 2^n codewords are valid.

Error Detecting Code

A simple method is to add one *parity* bit (even or odd parity) to the data. In this way, we can detect single-bit errors (if two bits change, the error cannot be detected).

```
          a parity bit (odd parity)
              ↓
10010101 | 1   (five '1', which is odd, OK)
11010101 | 1   (six '1', which is even, ERROR)
          ↑
a wrong (changed) bit
```

To detect e wrong bits, we need Hamming distance $d \geq e+1$. It takes 2 single-bit errors to go from a valid codeword to another valid codeword.

Error Correcting Code

To correct e wrong bits, we need $d \geq 2e+1$ Hamming distance. Larger distances between pairs of words enable a computer to correct e -bit errors:

```
0000000000
0000011111
1111100000
1111111111
```

Hamming distance for the given words is $5=2*2+1$, which allows a computer to correct double-bit errors:

```
0000000111
```

In the example above, if 1 or 2 errors occurred, it must have originally been

```
0000011111
```

because there is no other way to obtain a correct codeword.

Richard Hamming designed a method that can be used for constructing error-correcting codes for any size memory word.

Hamming Algorithm

To an m -bit word, r -redundant bits are added. The bits in the word are numbered from 1, with the bit number 1 at the leftmost position. All bits, which are on the position that corresponds to a power of 2, are *parity bits*. Other bits are *data bits*.

	16	8	4	2	1
1	0	0	0	0	1
2	0	0	0	1	0
3	0	0	0	1	1
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
7	0	0	1	1	1
8	0	1	0	0	0
9	0	1	0	0	1
10	0	1	0	1	0
11	0	1	0	1	1
12	0	1	1	0	0
13	0	1	1	0	1
14	0	1	1	1	0
15	0	1	1	1	1
16	1	0	0	0	0
17	1	0	0	0	1
18	1	0	0	1	0
19	1	0	0	1	1
20	1	0	1	0	0
21	1	0	1	0	1

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
0	0	1	0	1	1	1	0	0	0	0	0	1	0	1	1	0	1	1	1	0

An example: for a 16-bit word, to correct single-bit errors, we need 5 additional parity bits, i.e., we need $16+5=21$ bits. Bits on positions 1, 2, 4, 8, and 16 are *parity bits*; all the rest are *data bits*. We will use even parity in the example.

The table illustrates how parity bits control data bits in a 21-bit word.

Each bit in the upper header row controls those data bits that have '1' at a corresponding position.

Bit '16' controls data bits 16, 17, 18, 19, 20, 21.

Bit '8' controls data bits 8, 9, 10, 11, 12, 13, 14, 15.

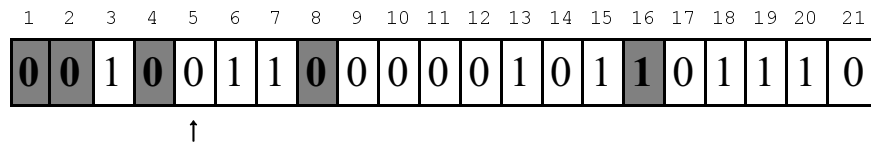
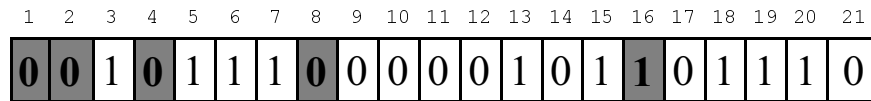
Bit '4' controls data bits 4, 5, 6, 7, 12, 13, 14, 15, 20, 21.

Bit '2' controls data bits 2, 3, 6, 7, 10, 11, 14, 15, 18, 19.

Bit '1' controls data bits 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21.

If the number of bits on controlled position is even, the parity bit is set to '0'; otherwise, it is set to '1'.

Let us suppose that the bit on the position 5 was changed:



After loading the word from memory, CPU checks the parity:

Parity bit 1 is incorrect (controlled positions contain five '1's).

Parity bit 2 is correct (controlled positions contain six '1's).

Parity bit 4 is incorrect (controlled bit positions contain five '1's).

Parity bit 8 is correct (controlled bit positions contain two '1's).

Parity bit 16 is correct (controlled bit positions contain four '1's).

Because parity bits 1 and 4 are wrong, the error must be somewhere at a position controlled by those two parity bits:

1: 1, 3, **5**, **7**, 9, 11, **13**, **15**, 17, 19, **21**
4: 4, **5**, 6, **7**, 12, **13**, 14, **15**, 20, **21**

The correct parity bit 2 eliminates its controlled positions 7 and 15.

The correct parity bit 8 eliminates its controlled position 13.

The correct parity bit 16 eliminates its controlled position 21.

The only left bit is bit **5**, which is the one in error. Because it was read as '1', it must be replaced by '0' to correct the error.

An algorithm:

- 1) For a loaded word, compute all the parity bits.
- 2) If the parity is incorrect, add up all the incorrect parity bit positions. The resulting sum is the position of the incorrect bit. E.g., in our case, $1+4 = 5$. Correct it.