Luhn algorithm

The **Luhn algorithm** or **Luhn formula**, also known as the "modulus 10" or "mod 10" algorithm, named after its creator, IBM scientist Hans Peter Luhn, is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers, IMEI numbers, National Provider Identifier numbers in the United States, Canadian Social Insurance Numbers, Israeli ID Numbers, South

African ID Numbers, Greek Social Security Numbers (AMKA), and survey codes appearing on McDonald's , Taco Bell , and Tractor Supply Co. receipts. It is described in U.S. Patent No. 2,950,048 , filed on January 6, 1954, and granted on August 23, 1960.

The algorithm is in the public domain and is in wide use today. It is specified in ISO/IEC 7812-1.[1] It is not intended to be a cryptographically secure hash function; it was designed to protect against accidental errors, not malicious attacks. Most credit cards and many government identification numbers use the algorithm as a simple method of distinguishing valid numbers from mistyped or otherwise incorrect numbers.

Description

The formula verifies a number against its included check digit, which is usually appended to a partial account number to generate the full account number. This number must pass the following test:

1. From the rightmost digit (excluding the check digit) and moving left, double the value of every second digit. The check digit is neither doubled nor included in this

calculation; the first digit doubled is the digit located immediately left of the check digit. If the result of this doubling operation is greater than 9 (e.g., 8 × 2 = 16), then add the digits of the result (e.g., 16: 1 + 6 = 7, 18: 1 + 8 = 9) or, alternatively, the same final result can be found by subtracting 9 from that result (e.g., 16: 16 − 9 = 7, 18: 18 − 9 = 9).

2. Take the sum of all the digits (including the check digit).
3. If the total modulo 10 is equal to 0 (if the total ends in zero) then the

number is valid according to the Luhn formula; otherwise it is not valid.

Example for computing check digit

Assume an example of an account number "7992739871" that will have a check digit added, making it of the form 7992739871x:

	7	9	9	2	7	3	9	8	7	1	x
Double every other	7	**18**	9	**4**	7	**6**	9	**16**	7	**2**	x
Sum digits	7	**9**	9	4	7	6	9	**7**	7	2	x

The sum of all the digits in the third row, the sum of the sum digits, is 67.

The check digit (x) is obtained by computing the sum of the sum digits then computing 9 times that value modulo 10 (in equation form, ((67 × 9) mod 10)). In algorithm form:

1. Compute the sum of the sum digits (67).
2. Multiply by 9 (603).
3. 603 mod 10 is then 3, which is the check digit. Thus, **x=3**.

(Alternative method) The check digit (x) is obtained by computing the sum of the other digits (third row) then subtracting the units digit from 10 (67 => Units digit 7; 10 − 7 = check digit 3). In algorithm form:

1. Compute the sum of the sum digits (67).
2. Take the units digit (7).
3. Subtract the units digit from 10.
4. The result (3) is the check digit. In case the sum of digits ends in 0 then 0 is the check digit.

This makes the full account number read 79927398713.

Example for validating check digit

...

Each of the numbers 79927398710, 79927398711, 79927398712, 79927398713, 79927398714, 79927398715, 79927398716, 79927398717, 79927398718, 79927398719 can be validated as follows.

1. Double every second digit, from the rightmost: (1×2) = 2, (8×2) = 16, (3×2) = 6, (2×2) = 4, (9×2) = 18
2. Sum all the *individual* digits (digits in parentheses are the products from Step 1): x (the check digit) + (2) + 7 + (1+6) + 9 + (6) + 7 + (4) + 9 + (1+8) + 7 = x + 67.

3. If the sum is a multiple of 10, the account number is possibly valid. Note that **3** is the only valid digit that produces a sum (70) that is a multiple of 10.
4. Thus these account numbers are all invalid except possibly 79927398713 which has the correct check digit.

Alternately, you can use the same checksum creation algorithm, ignoring the checksum already in place as if it had not yet been calculated. Then calculate the checksum and compare this calculated checksum to the original checksum included with the credit card number. If the included checksum matches the calculated checksum, then the number is valid.

Strengths and weaknesses

The Luhn algorithm will detect any single-digit error, as well as almost all transpositions of adjacent digits. It will not, however, detect transposition of the two-digit sequence *09* to *90* (or vice versa). It will detect most of the possible twin errors (it will not detect *22* ↔ *55*, *33* ↔ *66* or *44* ↔ *77*).

Other, more complex check-digit algorithms (such as the Verhoeff algorithm and the Damm algorithm) can detect more transcription errors. The Luhn mod N algorithm is an extension that supports non-numerical strings.

Because the algorithm operates on the digits in a right-to-left manner and zero digits affect the result only if they cause shift in position, zero-padding the beginning of a string of numbers does not affect the calculation. Therefore, systems that pad to a specific number of digits (by converting 1234 to 0001234 for instance) can perform Luhn validation before or after the padding and achieve the same result.

Prepending a 0 to odd-length numbers makes it possible to process the number from left to right rather than right to left, doubling the odd-place digits.

The algorithm appeared in a United States Patent[2] for a hand-held, mechanical device for computing the checksum. Therefore, it was required to be rather simple. The device took the mod 10 sum by mechanical means. The *substitution digits*, that is, the results of the double and reduce procedure, were not produced mechanically. Rather, the digits were marked in their permuted order on the body of the machine.

Pseudocode implementation

```
function checkLuhn(string purportedCC) {
    int sum := integer(purportedCC[length(purportedCC)-1])
    int nDigits := length(purportedCC)
    int parity := nDigits modulus 2
    for i from 0 to nDigits - 2 {
        int digit := integer(purportedCC[i])
        if i modulus 2 = parity
            digit := digit × 2
        if digit > 9
            digit := digit - 9
        sum := sum + digit
    }
    return (sum modulus 10) = 0
}
```

Python implementation

```
from random import randint
import time
from random import random
from random import seed

type = "random"
card = ""
card_types = ["random"]

def prefill(t):
    # typical number of
digits in credit card
    def_length = 16

    if t == card_types[0]:
        # discover card
starts with 6011 and is 16
digits long
        return [4, 0, 4,
0], def_length - 4
    else:
        # this section
probably not even needed
here
        return [],
def_length



def finalize(nums):
    check_sum = 0
    # is_even = True if
(len(nums) % 2) == 0 else
False

    check_offset =
len(nums) % 2

    for i, n in
enumerate(nums):
        if (i +
check_offset) % 2 == 0:
            check_sum += n
        else:
            check_sum += n
            n_ = n * 2
            check_sum += n_
```

```
- 9 if n_ > 9 else n_
    return nums + [10 -
(check_sum % 10)]



# main body
t = type.lower()
if t not in card_types:
    print("Unknown type:
'%s'" % type)
    print("Please pick one
of these supported types:
%s" % card_types)
else:
    initial, rem =
prefill(t)
```

```
    so_far = initial +
[randint(1, 9) for x in
range(rem - 1)]
    card = "".join(map(str,
finalize(so_far)))
    print("Card:", card)
```

PHP implementation

```
function luhnCheck(string
$vat): bool
{
    $length = strlen($vat);
    $sum=(int)
$vat[$length-1];
    $parity = $length % 2;
    for ($index = 0; $index
< $length-1; $index++) {
        $digit = (int)
$vat[$index];
        if($index % 2 ==
$parity){
            $digit *= 2;
        }
        if($digit > 9){
            $digit -= 9;
        }
        $sum += $digit;
    }
    return $sum % 10 == 0;
}
```

JavaScript implementation

```
const luhnNumber = (number)
=> {
    number =
String(number);

    let sum =
parseInt(number.charAt(numb
er.length - 1));

    for (let i = 0; i <
number.length - 1; i++) {
        let value =
parseInt(number.charAt(i));
```

```
        if (i % 2 === 0) {
            value *= 2;
        }

        if (value > 9) {
            value -= 9;
        }

        sum += value;
    }

    return sum % 10 === 0;
}
```

Usage

In addition to credit card numbers, this algorithm is also used to calculate the check digit on SIM card numbers.

For example. Take the SIM serial number (also known as ICCID) 89610195012344000018

The number printed on the SIM card is usually a subset of this.

The last two digits are check digits.

checkLuhn("896101950123440000") will return 1 - the first check digit

checkLuhn("950123440000") will return 8 - the second check digit. This shortened number, followed by its check digit is printed on the SIM card itself.

References

1. *ISO/IEC 7812-1:2017 Identification cards -- Identification of issuers -- Part 1: Numbering system*
2. *US Patent 2,950,048 – Computer for Verifying Numbers, Hans P Luhn, August 23, 1960*

External links

- Implementation in 88 languages on the Rosetta Code project