Check Digit Algorithm

Why bother with check digits?

The purpose of check digits is simple. Any time identifiers (typically number +/- letters) are being manually entered via keyboard, there will be errors. Inadvertent keystrokes or fatigue can cause digits to be rearranged, dropped, or inserted. Have you ever mis-dialed a phone number? It happens.

Check digits help to reduce the likelihood of errors by introducing a final digit that is calculated from the prior digits. Using the proper algorithm, the final digit can always be calculated. Therefore, when a number is entered into the system (manually or otherwise), the computer can instantly verify that the final digit matches the digit predicted by the check digit algorithm. If the two do not match, the number is refused. The end result is fewer data entry errors.

Calculate a check digit:

Valid MRN:

mod10+letters mod10 mod25 mod30

show bulk check digit tool

What is the Luhn algorithm?

We use a variation of the Luhn algorithm. This algorithm, also known as the "modulus 10" or "mod 10" algorithm, is very common. For example, it's the algorithm used by credit card companies to generate the final digit of a credit card.

Given an identifier, let's say "139", you travel right to left. Every other digit is doubled and the other digits are taken unchanged. All resulting digits are summed and the check digit is the amount necessary to take this sum up to a number divisible by ten.

Got it? All right, lets try the example.

Work right-to-left, using "139" and doubling every other digit.

9 x 2 = 18

3 = 3

1 x 2 = 2

Now sum all of the **digits** (note '18' is two digits, '1' and '8'). So the answer is '1 + 8 + 3 + 2 = 14' and the check digit is the amount needed to reach a number divisible by ten. For a sum of '14', the check digit is '6' since '20' is the next number divisible by ten.

Our variation on the Luhn algorithm

Allowing for Letters

We have borrowed the variation on the Luhn algorithm used by Regenstrief Institute, Inc. In this variation, we allow for letters as well as numbers in the identifier (i.e., alphanumeric identifiers). This allows for an identifier like "139MT" that the original Luhn algorithm cannot handle (it's limited to numeric digits only).

Allowing letters ~~even limited to capital letters~~ does not increase the accuracy of data entry. In fact, the potential for mistaking numbers and letters likely increases the chance for errors. In our case (Regenstrief with the AMPATH Medical Record System), we were forced to come up with a simple method for generating identifiers in disparate, disconnected location without collision (giving out the same number twice). Adding a 2-3 letter suffix to the identifer was our solution.

To handle alphanumeric digits (numbers **and** letters), we actually use the ASCII value (the computer's internal code) for each character and subtract 48 to derive the "digit" used in the Luhn algorithm. We subtract 48 because the characters "0" through "9" are assigned values 48 to 57 in the ASCII table. Subtracting 48 lets the characters "0" to "9" assume the values 0 to 9 we'd expect. The letters "A" through "Z" are values 65 to 90 in the ASCII table (and become values 17 to 42 in our algorithm after subtracting 48). To keep life simple, we convert identifiers to uppercase and remove any spaces before applying the algorithm.

The Luhn CheckDigit Validator uses this variation to allow for letters, whereas the Luhn Mod-10 Check-Digit Validator uses the standard Luhn Algorithm using only numbers 0-9.

Mod 25 and Mod 30

The idgen module supports additional algorithms, including Mod25 and Mod30 algorithms. These algorithms not only allow letters and numbers to be used throughout the identifier, but also allow the check "digit" to be a letter. Typically, letters than can easily be confused with numbers (B, I, O, Q, S, and Z) are omitted. In fact, the Mod25 algorithm omits both numbers and letters that look similar and can be confused with each other (0, 1, 2, 5, 8, B, I, O, Q, S, and Z); the Mod30 algorithm omits only the potentially confusing letters. The LuhnModNIdentifierValidator.java class contains the code that computes a check digit using "baseCharacters" as the set of possible characters for the identifier or check digit.

Here's how we handle non-numeric characters

For the second-to-last (2nd from the right) character and every other (even-positioned) character moving to the left, we just add 'ASCII value - 48' to the running total. Non-numeric characters will contribute values >10, but these digits are **not** added together; rather, the value 'ASCII value - 48' (even if over 10) is added to the running total. For example, '"M"' is ASCII 77. Since '77 - 48 = 29', we add 29 to the running total, **not** '2 + 9 = 11'.

For the rightmost character and every other (odd-positioned) character moving to the left, we use the formula '2 * n - 9 x INT(n/5)' (where INT() rounds off to the next lowest whole number) to calculate the contribution of every other character. If you use this formula on the numbers 0 to 9, you will see that it's the same as doubling the value and then adding the resulting digits together (e.g., using 8: '2 x 8 = 16' and '1 + 6 = 7'. Using the formula: '2 x 8 - 9 x INT(8/5) = 16 - 9 x 1 = 16 - 9 = 7') – identical to the Luhn algorithm. But using this formula allows us to handle non-numeric characters as well by simply plugging 'ASCII value - 48' into the formula. For example, '"Z"' is ASCII 90. '90 - 48 = 42' and '2 x 42 - 9 x INT(42/5) = 84 - 9 x 8 = 84 - 72 = 12'. So we add 12 (**not** '1 + 2 = 3') to the running total.

So, here's how we would use the Luhn algorithm for the identifier "139MT":

T (ASCII 84) -> 84 - 48 = 36 -> 2 x 36 - 9 x INT(36/5) = 72 - 9 x 7 = 72 - 63 = 9

M (ASCII 77) -> 77 - 48 = 29

9 x 2 = 18 -> 1 + 8 = 9 or 9 => 2 x 9 - 9 x INT(9/5) = 18 - 9 x 1 = 18 - 9 = 9

3 = 3

1 x 2 = 2 or 1 => 2 x 1 - 9 x INT(1/5) = 2 - 9 x 0 = 2

Summing the results we get '9 + 29 + 9 + 3 + 2 = 52'. The next number divisible by ten is 60. So, our check digit (the difference) is 8.

Java

The modified mod10 algorithm implemented in Java

```
public int checkdigit(String idWithoutCheckdigit) {

// allowable characters within identifier
String validChars = "0123456789ABCDEFGHIJKLMNOPQRSTUVYWXZ_";

// remove leading or trailing whitespace, convert to uppercase
idWithoutCheckdigit = idWithoutCheckdigit.trim().toUppercase();

// this will be a running total
int sum = 0;

// loop through digits from right to left
for (int i = 0; i < idWithoutCheckdigit.length(); i++) {

//set ch to "current" character to be processed
char ch = idWithoutCheckdigit
.charAt(idWithoutCheckdigit.length() - i - 1);

// throw exception for invalid characters
if (validChars.indexOf(ch) == \-1)
throw new InvalidIdentifierException(
"\"" + ch + "\" is an invalid character");

// our "digit" is calculated using ASCII value - 48
int digit = (int)ch - 48;

// weight will be the current digit's contribution to
// the running total
int weight;
if (i % 2 == 0) {

      // for alternating digits starting with the rightmost, we
      // use our formula this is the same as multiplying x 2 and
      // adding digits together for values 0 to 9.  Using the
      // following formula allows us to gracefully calculate a
      // weight for non-numeric "digits" as well (from their
      // ASCII value - 48).
      weight = (2 * digit) - (int) (digit / 5) * 9;

    } else {

      // even-positioned digits just contribute their ascii
      // value minus 48
      weight = digit;

    }

// keep a running total of weights
sum \+= weight;

}

// avoid sum less than 10 (if characters below "0" allowed,
// this could happen)
sum = Math.abs(sum) + 10;

// check digit is amount needed to reach next number
// divisible by ten
return (10 - (sum % 10)) % 10;

}
```

VBA

The modified mod10 algorithm implemented in VBA

```
Function checkdigit(idWithoutCheckDigit)

ucIdWithoutCheckdigit = UCase(idWithoutCheckDigit)
total = 0
For i = Len(ucIdWithoutCheckdigit) To 1 Step \-2
digit = Asc(Mid(ucIdWithoutCheckdigit, i, 1)) - 48
total = total + (2 * digit) - Int(digit / 5) * 9
If (i > 1) Then
digit = Asc(Mid(ucIdWithoutCheckdigit, i - 1, 1)) - 48
total = total + digit
End If
Next i
total = Abs(total) + 10
checkdigit = (10 - (total Mod 10)) Mod 10

End Function
```


Note:

This VBA algorithm should probably check each character and return an error if any invalid characters are found (as the Java example above does by throwing an exception).

Groovy

The modified mod10 algorithm implemented in Groovy

```
def checkdigit(idWithoutCheckDigit) {
        idWithoutCheckDigit = idWithoutCheckDigit.trim().toUpperCase()
        sum = 0
        (0..<idWithoutCheckDigit.length()).each { i ->
            char ch = idWithoutCheckDigit[-(i+1)]
            if (!'0123456789ABCDEFGHIJKLMNOPQRSTUVYWXZ_'.contains(ch.toString()))
                throw new Exception("$ch is an invalid character")
            digit = (int)ch - 48;
            sum += i % 2 == 0 ? 2*digit - (int)(digit/5)*9 : digit
          }
        (10 - ((Math.abs(sum)+10) % 10)) % 10
}

// Validate our algorithm
assert checkdigit('12') == 5
assert checkdigit('123') == 0
assert checkdigit('1245496594') == 3
assert checkdigit('TEST') == 4
assert checkdigit('Test123') == 7
assert checkdigit('00012') == 5
assert checkdigit('9') == 1
assert checkdigit('999') == 3
assert checkdigit('999999') == 6
assert checkdigit('CHECKDIGIT') == 7
assert checkdigit('EK8XO5V9T8') == 2
assert checkdigit('Y9IDV90NVK') == 1
assert checkdigit('RWRGBM8C5S') == 5
assert checkdigit('OBYY3LXR79') == 5
assert checkdigit('Z2N9Z3F0K3') == 2
assert checkdigit('ROBL3MPLSE') == 9
assert checkdigit('VQWEWFNY8U') == 9
assert checkdigit('45TPECUWKJ') == 1
assert checkdigit('6KWKDFD79A') == 8
assert checkdigit('HXNPKGY4EX') == 3
assert checkdigit('91BT') == 2
try {
checkdigit ("12/3")
assert false
} catch(e) { }
```

Python

Implemented in Python, by Daniel Watson

```
import math
def return_checkdigit(self, id_without_check):

        # allowable characters within identifier
        valid_chars = "0123456789ABCDEFGHIJKLMNOPQRSTUVYWXZ_"

        # remove leading or trailing whitespace, convert to uppercase
        id_without_checkdigit = id_without_check.strip().upper()

        # this will be a running total
        sum = 0;

        # loop through digits from right to left
        for n, char in enumerate(reversed(id_without_checkdigit)):

            if not valid_chars.count(char):
                raise Exception('InvalidIDException')

            # our "digit" is calculated using ASCII value - 48
            digit = ord(char) - 48

            # weight will be the current digit's contribution to
            # the running total
            weight = None
            if (n % 2 == 0):

                # for alternating digits starting with the rightmost, we
                # use our formula this is the same as multiplying x 2 and
                # adding digits together for values 0 to 9.  Using the
                # following formula allows us to gracefully calculate a
                # weight for non-numeric "digits" as well (from their
                # ASCII value - 48).
                weight = (2 * digit) - (digit / 5) * 9
            else:
                # even-positioned digits just contribute their ascii
                # value minus 48
                weight = digit

            # keep a running total of weights
            sum += weight

        # avoid sum less than 10 (if characters below "0" allowed,
        # this could happen)
        sum = math.fabs(sum) + 10

        # check digit is amount needed to reach next number
        # divisible by ten. Return an integer
        return int((10 - (sum % 10)) % 10)
```

Perl

Implemented in Perl, by Steve Cayford

```
sub checkdigit {
    my ( $tocheck ) = @_;

    $tocheck = uc $tocheck;
    die 'Invalid characters' if $tocheck =~ m/ [^0-9A-Z_] /xms;

    my $sum  = 0;
    my $even = 0;

    for my $char ( reverse split( qr//, $tocheck ) ) {
        my $n = ord( $char ) - 48;
        $sum +=
              $even
            ? $n
            : 2 * $n - 9 * int( $n / 5 );
        $even = ( $even + 1 ) % 2;
    }

    $sum = abs $sum + 10;
    return ( 10 - $sum % 10 ) % 10;
}
```

C#

C# direct translation, by Yves Rochon

```
        private static int D3CustomerCheckDigit(string idWithoutCheckdigit) {

            // allowable characters within identifier
            const string validChars = "0123456789ABCDEFGHIJKLMNOPQRSTUVYWXZ_";

            // remove leading or trailing whitespace, convert to uppercase
            idWithoutCheckdigit = idWithoutCheckdigit.Trim().ToUpper();

            // this will be a running total
            int sum = 0;

            // loop through digits from right to left
            for (int i = 0; i < idWithoutCheckdigit.Length; i++) {

                //set ch to "current" character to be processed
                char ch = idWithoutCheckdigit[idWithoutCheckdigit.Length - i - 1];

                // throw exception for invalid characters
                if (validChars.IndexOf(ch) == -1)
                    throw new Exception(ch + " is an invalid character");

                // our "digit" is calculated using ASCII value - 48
                int digit = (int)ch - 48;

                // weight will be the current digit's contribution to
                // the running total
                int weight;
                if (i % 2 == 0)
                {

                  // for alternating digits starting with the rightmost, we
                  // use our formula this is the same as multiplying x 2 and
                  // adding digits together for values 0 to 9.  Using the
                  // following formula allows us to gracefully calculate a
                  // weight for non-numeric "digits" as well (from their
                  // ASCII value - 48).
                  weight = (2 * digit) - (int) (digit / 5) * 9;
                }
                else
                {
                    // even-positioned digits just contribute their ascii
                    // value minus 48
                    weight = digit;
                }

                // keep a running total of weights
                sum += weight;

            }

            // avoid sum less than 10 (if characters below "0" allowed,
            // this could happen)
            sum = Math.Abs(sum) + 10;

            // check digit is amount needed to reach next number
            // divisible by ten
            return (10 - (sum % 10)) % 10;

        }
```

JavaScript

Implemented in JavaScript, by Owais Hussain

```
 function luhnCheckDigit(number) {
  var validChars = "0123456789ABCDEFGHIJKLMNOPQRSTUVYWXZ_";
  number = number.toUpperCase().trim();
  var sum = 0;
  for (var i = 0; i < number.length; i++) {
    var ch = number.charAt(number.length - i - 1);
    if (validChars.indexOf(ch) < 0) {
      alert("Invalid character(s) found!");
      return false;
    }
    var digit = ch.charCodeAt(0) - 48;
    var weight;
    if (i % 2 == 0) {
      weight = (2 * digit) - parseInt(digit / 5) * 9;
    }
    else {
      weight = digit;
    }
    sum += weight;
  }
  sum = Math.abs(sum) + 10;
  var digit = (10 - (sum % 10)) % 10;
  return digit;
}
```

Excel Formula

Input the number in cell "A1" and assign the formula below to cell "A2", which will give you the check digit.

Implemented in MS Excel, by Owais Hussain

```
A2=MOD(SUMPRODUCT(-MID(TEXT(MID(A1,ROW(INDIRECT("1:"&LEN(A1))),1)*(MOD(ROW(INDIRECT("1:"&LEN(A1)))+1,2)+1),"
00"),{1,2},1)),10)
```