Luhn mod N algorithm

From Wikipedia, the free encyclopedia

The **Luhn mod N algorithm** is an extension to the Luhn algorithm (also known as mod 10 algorithm) that allows it to work with sequences of non-numeric characters. This can be useful when a check digit is required to validate an identification string composed of letters, a combination of letters and digits or even any arbitrary set of characters.

Informal explanation [edit]

The Luhn mod N algorithm generates a check digit (more precisely, a check character) within the same range of valid characters as the input string. For example, if the algorithm is applied to a string of lower-case letters (*a* to *z*), the check character will also be a lower-case letter. Apart from this distinction, it resembles very closely the original algorithm.

The main idea behind the extension is that the full set of valid input characters is mapped to a list of code-points (i.e., sequential integers beginning with zero). The algorithm processes the input string by converting each character to its associated code-point and then performing the computations in mod N (where N is the number of valid input characters). Finally, the resulting check code-point is mapped back to obtain its corresponding check character.

Mapping characters to code-points [edit]

Initially, a mapping between valid input characters and code-points must be created. For example, consider that the valid characters are the lower-case letters from *a* to *f*. Therefore, a suitable mapping would be:

Character	*a*	*b*	*c*	*d*	*e*	*f*
Code-point	0	1	2	3	4	5

Note that the order of the characters is completely irrelevant. This other mapping would also be acceptable (although possibly more cumbersome to implement):

Character	*c*	*e*	*a*	*f*	*b*	*d*
Code-point	0	1	2	3	4	5

It is also possible to intermix letters and digits (and possibly even other characters). For example, this mapping would be appropriate for lower-case hexadecimal digits:

Character	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*a*	*b*	*c*	*d*	*e*	*f*
Code-point	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Add links

Algorithm [edit]

Assuming the following functions are defined:

```
int CodePointFromCharacter(char character) {...}

char CharacterFromCodePoint(int codePoint) {...}

int NumberOfValidInputCharacters() {...}
```

The function to generate a check character is:

```
char GenerateCheckCharacter(string input) {

 int factor = 2;
 int sum = 0;
 int n = NumberOfValidInputCharacters();

 // Starting from the right and working leftwards is easier since
 // the initial "factor" will always be "2"
 for (int i = input.Length - 1; i >= 0; i--) {
  int codePoint = CodePointFromCharacter(input[i]);
  int addend = factor * codePoint;

  // Alternate the "factor" that each "codePoint" is multiplied by
  factor = (factor == 2) ? 1 : 2;

  // Sum the digits of the "addend" as expressed in base "n"
  addend = (addend / n) + (addend % n);
  sum += addend;
 }

 // Calculate the number that must be added to the "sum"
 // to make it divisible by "n"
 int remainder = sum % n;
 int checkCodePoint = (n - remainder) % n;

 return CharacterFromCodePoint(checkCodePoint);
}
```

And the function to validate a string (with the check character as the last character) is:

```
bool ValidateCheckCharacter(string input) {

 int factor = 1;
 int sum = 0;
 int n = NumberOfValidInputCharacters();

 // Starting from the right, work leftwards
 // Now, the initial "factor" will always be "1"
 // since the last character is the check character
 for (int i = input.Length - 1; i >= 0; i--) {
  int codePoint = CodePointFromCharacter(input[i]);
  int addend = factor * codePoint;

  // Alternate the "factor" that each "codePoint" is multiplied by
  factor = (factor == 2) ? 1 : 2;

  // Sum the digits of the "addend" as expressed in base "n"
  addend = (addend / n) + (addend % n);
  sum += addend;
 }

 int remainder = sum % n;

 return (remainder == 0);
}
```

Example [edit]

Generation [edit]

Consider the above set of valid input characters and the example input string *abcdef*. To generate the check character, start with the last character in the string and move left doubling every other code-point. The "digits" of the code-points as written in base 6 (since there are 6 valid input characters) should then be summed up:

Character	*a*	*b*	*c*	*d*	*e*	*f*
Code-point	0	1	2	3	4	5
Double		2		6 (base 10) 10 (base 6)		10 (base 10) 14 (base 6)
Reduce	0	2	2	1 + 0	4	1 + 4
Sum of digits	0	2	2	1	4	5

The total sum of digits is **14** (0 + 2 + 2 + 1 + 4 + 5). The number that must be added to obtain the next multiple of 6 (in this case, **18**) is **4**. This is the resulting check code-point. The associated check character is **e**.

Validation [edit]

The resulting string *abcdefe* can then be validated by using a similar procedure:

Character	*a*	*b*	*c*	*d*	*e*	*f*	*e*
Code-point	0	1	2	3	4	5	4
Double		2		6 (base 10) 10 (base 6)		10 (base 10) 14 (base 6)	
Reduce	0	2	2	1 + 0	4	1 + 4	4
Sum of digits	0	2	2	1	4	5	4

The total sum of digits is **18**. Since it is divisible by 6, the check character is **valid**.

Implementation [edit]

The mapping of characters to code-points and back can be implemented in a number of ways. The simplest approach (akin to the original Luhn algorithm) is to use ASCII code arithmetic. For example, given an input set of *0* to *9*, the code-point can be calculated by subtracting the ASCII code for '0' from the ASCII code of the desired character. The reverse operation will provide the reverse mapping. Additional ranges of characters can be dealt with by using conditional statements.

Non-sequential sets can be mapped both ways using a hard-coded *switch/case* statement. A more flexible approach is to use something similar to an Associative Array. For this to work, a pair of arrays is required to provide the two-way mapping.

An additional possibility is to use an array of characters where the array indexes are the code-points associated with each character. The mapping from character to code-point can then be performed with a linear or binary search. In this case, the reverse mapping is just a simple array lookup.

Weakness [edit]

This extension shares the same weakness as the original algorithm, namely, it cannot detect the transposition of the sequence *<first-valid-character><last-valid-character>* to *<last-valid-character><first-valid-character>* (or vice versa). This is equivalent to the transposition of *09* to *90* (assuming a set of valid input characters from *0* to *9* in order). On a positive note, the larger the set of valid input characters, the smaller the impact of the weakness.

See also [edit]

- International Securities Identification Number

Categories: Modular arithmetic | Checksum algorithms