# Cryptography Toolbox Report

In this report I will cover the efficiency and the strategies applied for the algorithms used for Credit Card validation, BCH(10,6), two SHA1 Brute Forcers and Factorisation using Fermat’s and Dixons formulas. I will then sum up the key points learnt from both code development and further research.

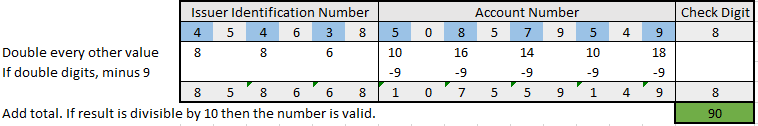
## Validating Card Numbers with the Luhn Check AlgorithmCredit Card

Credit card numbers are formed using three key sections. The first six digits form the bank identification number, the next nine digits are the account number and the final number is the check digit.

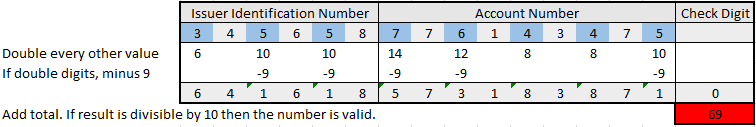
Credit cards are one of many identification numbers that use the Luhn Algorithm for validation. This algorithm is what’s used to generate the final check digit in the card number.

As shown in the examples below, alternate digits on the card are doubled, with any resulting double digits having 9 taken away from them. With these new numbers the total, including the check digit, should be a multiple of 10 for the number to be valid.

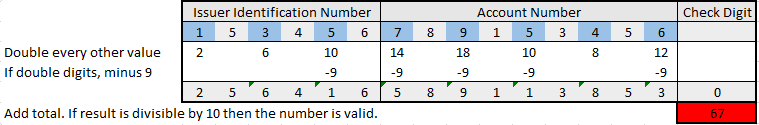
This example shows a valid credit card number:

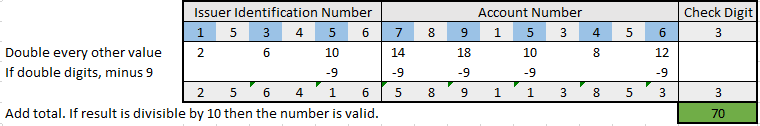


This example shows an invalid credit card number.



This example shows the generation of the final check digit when creating a card number. The same process is run and a total is found, but using one less digit. To make the card number valid, the check digit is appended of a value that brings the total up to a multiple of ten. As shown, a three is added to the generated number to form a complete, valid credit card number.





### Efficiency

The Luhn algorithm is very simple and can detect any single digit error at the cost of only a single digit. It will also detect transpositions of adjacent digits. For example in the image above, if the INN started with 51, instead of 15, the total would result in 65, rather than 70.

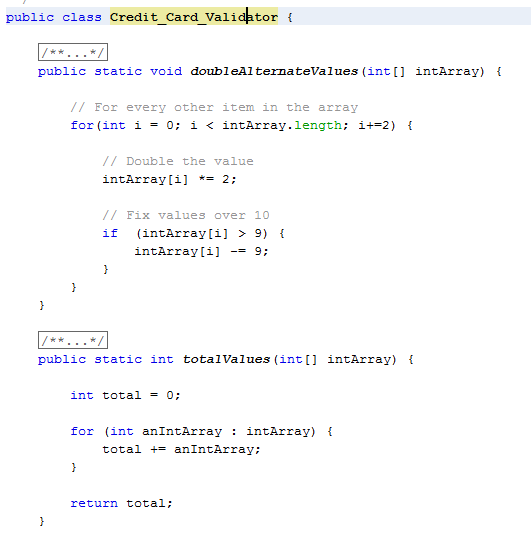
After running several tests and researching online, it appears the only transposition it does not detect is 09 to 90. It will also detect 7 of the 10 possible twin errors, not including 22 ↔ 55, 33 ↔ 66 or 44 ↔ 77[[1]](#footnote-1). These are weaknesses that seem to have been resolved in Verhoeff’s and Damm’s algorithms[[2]](#footnote-2),

As only nine of the possible sixteen digits can be used to generate valid unique card numbers, is this method efficient? Well nine digits gives a possibility of 1,000,000,000 unique accounts. That’s around one card for every seven people for the world. Not enough.

Fortunately, this is not the maximum number of possibilities as the IIN has not yet been considered. The number of cards that any issuer can issue depends on how many 6-digit issuer IDs they have. For example, Visa has the entire range of IDs starting with the digit 4[[3]](#footnote-3). That's 100,000 IDs.

This means that Visa alone are able to issue 1,000,000,000 x 100,000, 100 trillion unique card numbers. With a minimum of a billion possible combinations per issuer, it is unlikely for any to ever run out. If they ever did so they would simple need to request a new issuer number to gain another billion possibilities.

### Code



The code that I have written is not the most efficient, but works reliably and gives an instant result when validating a single credit card number.

The reason for the efficiency is because I wanted to make the code easily readable. Each function completes a specific task. An example of this is the code block shown to the right.

These two methods double alternate values and totals all digits together. The two could be merged under the same for loop to increase performance, but I wanted to capture the idea of having individual steps to make it easy to view when looking back on.

### What was learnt

Learning Luhn’s algorithm was very useful for understanding the key concepts of error detection through the use of a parity bit. Having this knowledge really helped when tackling the BCH algorithm, moving on from just detecting an error to fixing it as well.

## BCH(10,6)

BCH (Bose, Chaudhuri, Hocquenghem) codes form a large class of multiple random error-correcting codes. BCH codes are cyclic codes and can be adapted to detect and fix different numbers of errors, depending on the number of parity bits used.

BCH(10,6) can correct up to two errors and detect a triple error with the addition of 4 bits onto a 6 bit string.

### Encoding

To calculate the four parity bits for a six digit number, the following formulas are used:

d7 = (4d1+10d2+9d3+2d4+d5+7d6) mod 11 d8= (7d1+8d2+7d3+d4+9d5+6d6) mod 11

d9 = (9d1+d2+7d3+8d4+7d5+7d6) mod 11 d10 = (d1+2d2+9d3+10d4+4d5+d6) mod 11

### Decoding

To decode the following steps are followed:

s1= (d1+d2+d3+d4+d5+d6+d7+d8+d9+d10) mod 11

s2= (d1+2\*d2+3\*d3+4\*d4+5\*d5+6\*d6+7\*d7+8\*d8+9\*d9+10\*d10) mod 11

s3= (d1+22\*d2+32\*d3+42\*d4+52\*d5+62\*d6+72\*d7+82\*d8+92\*d9+102\*d10) mod 11

s4= (d1+23\*d2+33\*d3+43\*d4+53\*d5+63\*d6+73\*d7+83\*d8+93\*d9+103\*d10) mod 11

If all of these values work out as 0, then there is no error in the transmission. If there are not, then the following values must be calculated:

P = S22 - S1 S3 Q = S1 S4 - S2 S3 R = S32 - S2 S4

If these three values work out to be 0 then there is a single error. The position or the error is s2/s1, and the magnitude is s1. This information can be used to correct the error. Otherwise four other pieces of information are required:

i = (- Q + √(Q2-4\*P\*R)) / 2\*P j = (- Q - √(Q2-4\*P\*R)) / 2\*P b = (i\*s1- s2) / (i - j) a = s1 – b

If (Q2-4\*P\*R) doesn’t have a square root under mod 11, or the values of i or j is zero, there are three or more errors. Otherwise there are two errors. The position of the first error is i, of magnitude a. The position of the second error is j of magnitude b.

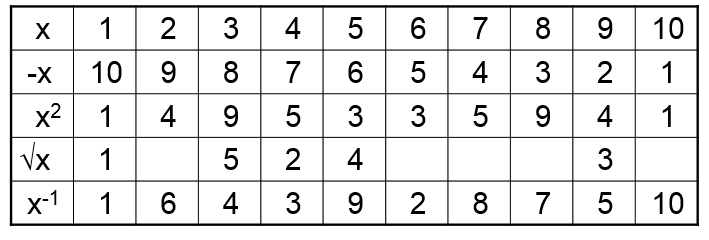
### Efficiency

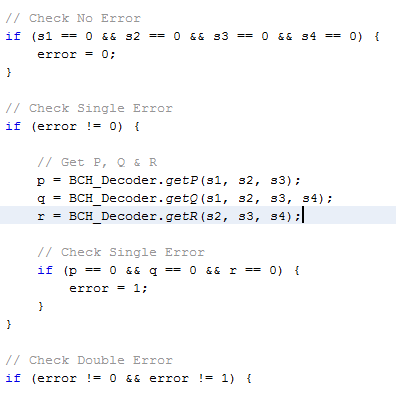
With six digits there are a total of one million possible numbers. However, when the parity bits are generated there is a chance that one or more parity bits result as a two digit number. In fact, 32% of all possible numbers produce at least one double digit number, which is extremely high. This makes 320,000 possibilities invalid, as there are only four available slots for parity digits.

If the number of parity bits were extended, there would be no way to tell which of the values the two digit number was. For the example 10101, how would you tell if the parity bits were 10, 1, 0, 1 or 1, 0, 10, 0 etc. One way to resolve this issue would be to add an additional bit stating the location of a multi digit number. However, if there was more than one double digit value you would need a further bit, so 6 parity digits and 2 check digits.

At this point it would be better to use eight bits to simply allow for multiple digit values. However, it would be a very high cost to have eight parity bits for a six digit number. Another possible alternative would be to use a different set of characters instead of numbers. For example the alphabet has 26 letters that could be used to represent double digit numbers, e.g. H = 8, I = 9, J = 10. This would allow for all the numbers to be usable, but would come at the programming cost of having to decode the value of each letter when decoding.

### Code

The code for this task was challenging, as certain operators did not function as expected. For example, in certain cases Math.pow(x, 2) would generate a different result to x \* x. Calculating the square root and inverse power of numbers also proved much more difficult than expected.

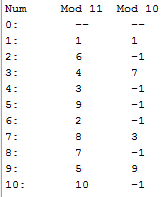
The strategy used was to start by creating a method for each of the functions to the right and ensure these worked correctly. From here, I wrote individual methods for calculating the syndromes, PQR, the quadratic equation i & j, then finally for a & b.

From here, these methods could be inserted into if statements, assigning the number of errors depending on the results. As shown to the side, if all syndromes are equal to 0, then all other checks can be skipped and the result output. If not, then the next check is performed and so on.

Coding in this way avoided the code becoming too nested, so simpler to follow and break apart into clear steps. Based on the result of error, a switch statement would determine the correct message to output.

### What was learnt

During the process I learnt that modular arithmetic is essentially doing arithmetic not on a line, but round a circle. The values wrap around, always staying less than a fixed number called the modulus. Mod 11 used in the formulas ensure all values range between 0 and 10, and as 11 is prime there is less chance of collisions.

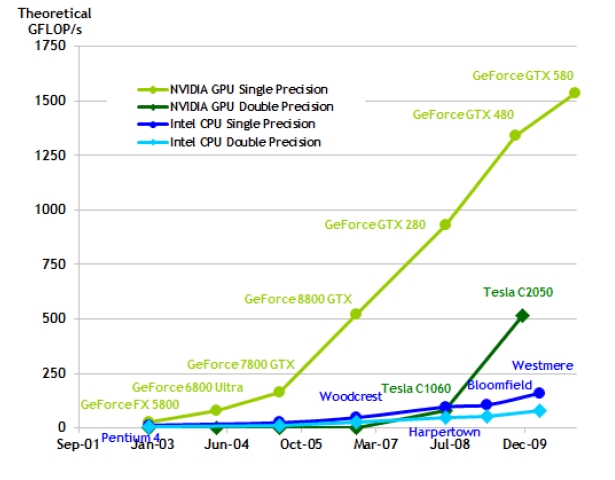
When thinking through the efficiency of the algorithm, I considered using mod 10 would resolve the issue of having double-digit values. However, I learnt from the process of attempting to implement it, under modular 10 many numbers don’t have a multiplicative inverse. The screenshot to the side shows this, with no errors using mod 11 and over half of the results for mod 10 giving an error.

## Brute Force

SHA-1 is a cryptographic hash function that encrypts a string into a 40 digit cryptographic hash. Even a slight change results in a completely different hash, so there is no way to guess a hash based on the result of a similar word. A hash is considered practically impossible to reverse, with the only real tactic to check millions of combinations of words / characters until a match is found.

### Efficiency

Brute force is the tactic of running through every possible combination using a set of predefined characters. In this project, all lower case letters and numbers were used, totalling 36 unique characters. If the password were 6 characters, the number of possible combinations would be 366 = 2,176,782,336. Using a reasonably speced computer it would likely take around 20 minutes to run through every combination.

If increased to 8 characters, 368 = 101,559,956,668,416 combinations would take around 17 days[[4]](#footnote-4). Only a few years ago a password of this length would have been near impossible to break for a regular computer, showing how quickly the development of GPUs has moved on, as this length password is no longer safe even against brute force[[5]](#footnote-5).

As a comparison, if the key space was increased to include all lowercase and uppercase letters, numbers and special characters, the number of possibilities for a 8 character password would be 948, which would take over 113 years. Finally for a 10 character password using the original key space, it would take over 63 years. At both these points the password would likely have expired past its usefulness, making the hash successful.

Brute force is generally not considered efficient compared to something like a dictionary attack. Dictionary attacks contain a list of commonly used passwords. Reverse lookup tables and rainbow tables are similar, but have the hashes of these passwords already pre-computed.

|  |  |  |
| --- | --- | --- |
| Password | Brute Force | Dictionary |
| TestPassword123 | 2.43 x 10^7 Years | 11.22 Seconds |
| pass12345 | 101.56 Seconds | 0.01 Seconds |
| PA55W0RD | 2.82 Seconds | 3.98 x 10^6 Years |

This table shows that for commonly used words, a dictionary will find passwords extremely quickly, but will be unable to find anything not held within the dictionary, so the efficiency depends of the password and quality of the dictionary[[6]](#footnote-6).

### Code

Writing the code for the brute force algorithm required a lot of time planning to avoid falling into the trap of having a large number of for loops nested within other loops and repeating code for checking strings of different lengths.

All code is held within the main for loop, which runs from a word length of 1 to the maximum set word length, or until the correct word is found. Within this loop is another for loop which runs for the number of possible combinations of that word length.

Within this loop the indices are generated, e.g. [0, 0, 1], [0, 1, 0], [0, 1, 1] etc. and the matching character from the defined charset is assigned. The array of characters is then converted to a string and its hash is compared against the user entered hash value. If there is a match it breaks, otherwise the next index is generated.

For the specialised brute force program I used the same class and added a few additional lines. The GUI offers two separate inputs and submission buttons. The first sets the variable ‘special’ to false and the second to true. I added an additional check when assigning a character to the index value to count the amount of numbers in the combination.

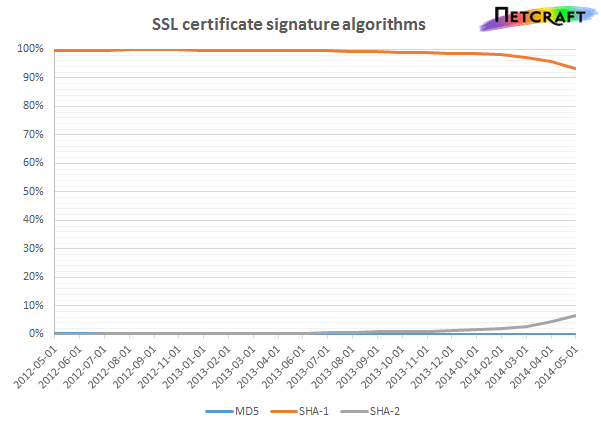
I then surrounded the code that converts the string to a hash and checks it in an IF statement. If special is true and there are exactly two numbers it will run the check. If special is false then the normal version is running and it will perform the check. Otherwise the check will be skipped, as the specialised version is running, but the string did not contain two numbers so must be invalid, so there is no point converting it and checking as we already know it’s incorrect.

Below is a table that contains a list of SHA1 hashes, the decrypted plain text and the time taken to crack using the standard and special brute force algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hash | Decrypted | Time | Time | Efficiency |
| c2543fff3bfa6f144c2f06a7de6cd10c0b650cae | this | 00:00:03 | - |  |
| b47f363e2b430c0647f14deea3eced9b0ef300ce | is | 00:00:00 | - |  |
| e74295bfc2ed0b52d40073e8ebad555100df1380 | very | 00:00:03 | - |  |
| 0f7d0d088b6ea936fb25b477722d734706fe8b40 | simple | 01:14:29 | - |  |
| 77cfc481d3e76b543daf39e7f9bf86be2e664959 | fail7 | 00:00:34 | - |  |
| 5cc48a1da13ad8cef1f5fad70ead8362aabc68a1 | 5you5 | 00:03:33 | 00:00:22 | 90% |
| 4bcc3a95bdd9a11b28883290b03086e82af90212 | 3crack | 01:41:46 | - |  |
| 7302ba343c5ef19004df7489794a0adaee68d285 | 1you1 | 00:03:04 | 00:00:18 | 90% |
| 21e7133508c40bbdf2be8a7bdc35b7de0b618ae4 | 00if00 | 01:42:18 | - |  |
| 6ef80072f39071d4118a6e7890e209d4dd07e504 | cannot | 00:11:21 | - |  |
| 02285af8f969dc5c7b12be72fbce858997afe80a | 4this4 | 02:02:17 | 00:14:34 | 88% |
| 57864da96344366865dd7cade69467d811a7961b | 6will | 00:01:27 | - |  |

### What was learnt

The table shows that when dealing with a string with two numbers, the specialised brute force program is around 90% more efficient than the regular version. This shows the amount of processing power that goes into converting a string to a hash and how much easier it can be to crack a password if you have some prior knowledge of what characters it contains.

I also learnt the difference between a reverse lookup table and a rainbow table is that rainbow tables use run time searching, which trades memory for computation time. Instead of storing an individual password hash, you create a chain of password hashes and only store the beginning and the end of the chain. Enough of these chains can cover all passwords in a given alphabet.

I also learnt that SHA1 is close to becoming vulnerable to collision attacks. It is predicted that a collision attack is well within the range of what an organized crime syndicate can practically budget by 2018, and a university research project by 2021[[7]](#footnote-7).

This is worrying, as currently around 90% of websites that use SSL encryption use SHA1[[8]](#footnote-8). Companies such a Google and Microsoft have stated they will no longer accept SHA1 certificates as of 2017. Thankfully as of this year the estimated cost to break a single SHA1 is $700k, although the cost in 2012 was $2.7M and the price is rapidly decreasing.

## Factorising

The way of breaking down a composite number into smaller non-trivial divisors.

**Fundamental Theorem of Arithmetic**: Any integer *x* greater than 1 can be uniquely represented as:

*x* = p1a1p2a2 …pnan (pi are primes)

Fermat’s method is based on that fact that every odd integer can be represented as the difference of two squares, n = a2 − b2 = (a+b) (a-b). To find the first square, a = ceiling(sqrt(n)). If a2 – n = a square, the value of ‘a’ is the second square ‘b’, otherwise ‘a’ is incremented and checked again.

Dixon’s method is based on Fermat’s, but instead of looking for squares, it looks for numbers with small prime factors using a factor base. Rather than having a defined starting number, it is picked at random. The other key difference is that even if a pair isn’t found, if a single square number is discovered it is stored and checked against other stored squares. This helps to find the correct pair far faster than running through sequentially.

### Efficiency

The table below shows the time taken to factorise a set of numbers using both Fermat’s and Dixon’s algorithms:

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Fermat | Dixon | Efficiency |
| 2450609331732137 | 00:00:01 | 00:00:00 | - |
| 24506093317321377 | 00:00:03 | 00:00:12 | 300% |
| 245060933173213777 | 00:05:41 | 00:00:07 | 98% |
| 2450609331732137777 | 00:06:10 | 00:00:14 | 96% |
| 11111111111111111 | 00:01:51 | 00:00:32 | 71% |
| 764796435689543 | 00:02:49 | 00:01:03 | 63% |
| 235791113 | 00:00:07 | 00:00:04 | 56% |
| 245245245245245 | 00:00:07 | 00:00:00 | - |
| 245245245245245137 | 00:04:10 | 00:00:07 | 97% |
| 24506093317321 | 00:00:09 | 00:00:00 | - |

This table shows that although Dixon’s is in most cases a lot faster to find a correct pair, the percentage efficiency is in no way consistent. This is likely due to the random nature of the numbers selected, as I found rerunning tests results could vary with Dixon’s by up to 15 seconds. This is likely the cause of the second result where Dixon’s took longer than Fermat’s to complete. Despite this, it is a clear improvement over Fermat’s method, especially on larger numbers, and in the tests I ran in some cases performed up to 98% faster.

### Code

For Fermat’s Factorisation I broke the code down into three key methods. The first is calculating the first root, which calculates the initial guess, then calls the second method, which determines whether the current number is a square. If not, the number is incremented and looped through until a square is found. Finally, the final method calculates the second root based off the first, r2 = n / r1.

The code for Dixon’s is far more complex. The square root of the number is taken, then a number between that value and the original is randomly generated. If this value can be represented as a base then the pairs are generated. If the greatest common denominator of the exponent is a prime the pair has been found, otherwise the pair is stored and checked against every other stored pair to see if any can be combined. If not, another random number is generated and the previous steps repeat until a match is found.

I left my initial attempt at Dixon’s within my code that ran in a more simplified way, not storing pairs and running sequentially, much like Fermat’s. It was not a true Dixon’s, but performed faster than Fermat’s and helped me understand the process before starting on my final attempt.

### What was learnt

I learnt that many cryptographic protocols are based on the difficulty of factoring large composite integers and that an algorithm that could efficiently find factors would make encryption methods such as RSA insecure. I find it surprising that there is currently no efficient way of calculating this, but it shows how unique prime numbers are and their importance to cryptography.

Shor's algorithm has shown the possibility of efficiently factorising very large integers using quantum computing. Using qubits which represent a dynamic state of 0 or 1, a register of 32 qubits can represent all possible values between 0 to 232 -1. If x were stored in a ‘qubit’ register, ax = 1 mod n could be completed in a single step, a calculation that classic computing finds very difficult to compute[[9]](#footnote-9). However, currently the highest number factorised in this way is 21, so far more work is required to turn this into a reality[[10]](#footnote-10).

I have also learnt that to defend against these kinds of attacks both primes shouldn’t have many small primes factors, as this can reduce the risk of them becoming factorised. The length of primes used are also extremely large, often a thousand bits long.

1. <http://en.wikipedia.org/wiki/Luhn_algorithm> [↑](#footnote-ref-1)
2. <http://en.wikipedia.org/wiki/Verhoeff_algorithm>, <http://en.wikipedia.org/wiki/Damm_algorithm> [↑](#footnote-ref-2)
3. <http://en.wikipedia.org/wiki/Bank_card_number#Issuer_identification_number_.28IIN.29> [↑](#footnote-ref-3)
4. <http://calc.opensecurityresearch.com/> [↑](#footnote-ref-4)
5. <http://deprocess.org/lets-talk-graphics-cards/> [↑](#footnote-ref-5)
6. <http://daleswanson.org/things/password.htm> [↑](#footnote-ref-6)
7. <http://arstechnica.com/security/2012/10/sha1-crypto-algorithm-could-fall-by-2018/> [↑](#footnote-ref-7)
8. <http://news.netcraft.com/archives/2014/05/05/sha-2-very-cryptographic-so-secure-such-growth-wow.html> [↑](#footnote-ref-8)
9. <http://www.scottaaronson.com/blog/?p=208> [↑](#footnote-ref-9)
10. <http://en.wikipedia.org/wiki/Shor's_algorithm> [↑](#footnote-ref-10)