Problem 180: Trust, But Verify

Difficulty: Hard

Author: Dr. Francis Manning, Owego, New York, United States

Originally Published: Code Quest 2022

Problem Background

A key issue in software design and development is the concept of data validation. How do you know if the input data you are receiving has been tampered with? One of the primary mechanisms to accomplish this is by using a hash algorithm.

A hash algorithm is a way of condensing any amount of data into a smaller byte array of a specific size. These are one-way functions, so the original data cannot be reconstructed from the hash alone; additionally, they are deterministic, meaning any given block of data will always produce the same hash. While hashes aren't quite unique - there's an infinite number of ways to put together data, but a non-infinite number of hash values - it's nearly impossible to intentionally create a set of data that results in the same hash as another. These properties make these hash functions ideal for data validation.

Once a hash has been calculated for any particular set of data, that hash can be published as a "checksum." Anyone can re-run the same hash algorithm on the source data, and if the result doesn't match the provided checksum, that proves the data has been tampered with.

Problem Description

Your team is working with Lockheed Martin's Corporate Information Security office (LMCIS) to implement a new automated security process for verifying documents. Your program will receive a string of text representing a document being downloaded from Lockheed Martin's servers. It must calculate a hash of that data, then compare that hash against the one provided by a different server. If the hash matches, everything is fine; if not, you'll need to flag the error so Lockheed Martin employees know not to open that document - it's been tampered with!

LMCIS has developed a new hash algorithm for this purpose. (Note that this is actually a bad idea in reality; you should always use previously published and cryptographically secure hash algorithms whenever one is needed to ensure the best security.) The hash algorithm works using the following process:

1. Convert the given text string to binary by replacing each character in the string with the binary equivalent shown in the ASCII table in the reference information, retaining any leading zeroes. For example, 'A' would be replaced with '1000001', and '.' would be replaced with '0101110'.
2. Add additional '1' bits to the end of the binary string until the length is evenly divisible by 128.

3. Split the resulting binary string into "chunks" of 32 bits each.
4. Initialize variables, called A, B, C, and D, to the 32-bit binary equivalents of the following numbers:
    a. A = 792,250,721
    b. B = 683,117,105
    c. C = 1,215,387,974
    d. D = 1,767,829,900
5. For each chunk (from step 3), calculate new values for A, B, C, and D as follows. M represents the current chunk; N represents the index of that chunk (first chunk has index 0, second has index 1, etc.).
    a. S = (B XOR D) AND (C OR (NOT B))
    b. S = A + S
    c. S = M + S
    d. Left rotate S by (N modulo 32) bits
    e. Set A = D, D = C, C = B, and B = S
6. Concatenate the final values of A, B, C, and D into a single binary string (in that order)
7. Convert the result into a 32-character hexadecimal string by replacing every four bits with the corresponding hex character (0000 = 0, 1111 = F, etc.)

In case you're not familiar with the various operations described in step 5:

- A OR B compares each bit in A and B in turn. If at least one bit in A or B is '1', the corresponding bit in the result is also '1'. (e.g. 1100 OR 1010 = 1110)
- A AND B compares each bit in A and B in turn. If both bits in A and B are '1', the corresponding bit in the result is also '1'. (e.g. 1100 AND 1010 = 1000)
- A XOR B compares each bit in A and B in turn. If exactly one bit in A or B is '1', the corresponding bit in the result is also '1'. (e.g. 1100 XOR 1010 = 0110)
- NOT A inverts the value of each bit in A. (e.g. NOT 10 = 01)
- A + B adds the values of A and B together, then truncates the result to match the lengths of A and B. (e.g. 1111 + 1010 = 1001)
- A left rotation of X bits moves the X most significant (leftmost) bits to the end of the number. (e.g. a left rotation of 1 bit for 101010 results in 010101; 2 bits results in 101010.)

Sample Input

The first line of your program's input, received from the standard input channel, will contain a positive integer representing the number of test cases. Each test case will include:

- A line containing a string of text to be hashed, up to 2000 characters long. This line may contain any character listed in the US ASCII table in the reference information.
- A line containing 32 uppercase hexadecimal characters representing the checksum hash to compare against.

```
2
This is a sample string for which you will need to calculate a hash.
45F1D2C216714CA65BE85211A364B2DB
Once calculated, compare the hash to the given hash and see if they match.
C2935B67EE5204A671D231E94676C101
```

Sample Output

For each test case, your program must print a single line with the word "TRUE" if the hash you calculate matches the provided checksum value, or "FALSE" otherwise.

```
TRUE
FALSE
```

To assist with your debugging efforts, the text for the second test case should produce the hash value shown below. Please note that this is not part of the expected output:

```
43A442826ED0CCB0E89B4160E76087B3
```