0.1 Square roots mod p (Tonelli-Shanks)

- *Algorithm:* Tonelli-Shanks (algo. 1)
- *Input:* A prime number p and a remainder n.
- *Complexity:* $\mathcal{O}(\log^2 p)$
- *Data structure compatibility:* N/A
- *Common applications:* Find the square root modulo a prime, applying to Legendre symbol and Jacobi symbol.

Problem. Square roots mod p (Tonelli-Shanks)

Given a prime p and a remainder n, find the square root r such that $r^2 \equiv n \mod p$.

Description

According to Euler's criterion, if n has a square root modulo p, it is equivalent to the formula below:

$$n^{\frac{p-1}{2}} \equiv 1 \mod p$$

To the opposition, for a number z with no square root, it means that

$$z^{\frac{p-1}{2}} \equiv -1 \mod p$$

Since about half of the number in the finite field $\mathbb{Z}_p$ will have no square root modulo p, it is easy to find such a z.

Then the method of Tonelli-Shanks algorithm works as follows. Firstly, we can transform $p-1$ into $Q2^S$, thus Q is a odd number. If we take R as

$$R \equiv n^{\frac{Q+1}{2}} \mod p$$

Then $R^2 \equiv n \cdot n^Q \mod p$. In this sense, if $t \equiv n^Q \equiv 1 \mod p$, R will be a square root of n modulo p. If not, we assign $M = S$, and we can have R and t like:

- $R^2 \equiv nt \mod p$
- t is a 2^{M-1}_{th} root of 1, as:
$$t^{2^{M-1}} \equiv t^{2^{S-1}} \equiv n^{Q2^{S-1}} \equiv n^{\frac{p-1}{2}} \equiv 1 \mod p$$

Then, we can again choose a pair of R and t for $M-1$ satisfying the above conditions, and finally stop when t is the 2^0_{th} root of 1 modulo p. When this is reached, the corresponding R at that stage would be the square root of n modulo p.

To make the decrease of M within iterations more specific, we can think as follows. If we find that t is a 2^{M-2}_{th} root of 1, we can simply keep the same R and t to the next iteration. If not, then t must be a 2^{M-2}_{th} root of -1 modulo p, since t is always the 2^{M-1}_{th} root of 1 modulo p. Then, our goal would be to find a b such that new R would be the old R multiplied by b, which means that new t will be old t multiplied by b^2 to maintain $R^2 \equiv nt \mod p$. Therefore, b should be another 2^{M-2}_{th} root of -1 [1].

According to the definition of z, z^Q will be the 2^{S-1}_{th} root of -1, since

$$z^{Q2^{S-1}} \equiv z^{\frac{p-1}{2}} \equiv -1 \mod p$$

Therefore, we can take the square of z^Q again and again, and will get a sequence of 2^i_{th} root of -1, for $i \in \{0, 1, \ldots, S-1\}$.

Combining all the thesis above, we can iterate from $M = S$ initially, and finally get the square root of n when $t = 1 \mod p$.

The time complexity of Tonelli-Shanks Algorithm is $\log^2 p$, since $M = S$ which can be roughly seen as the binary length of p, which is $\log_2 p$. Also, for confirming what 2^i_{th}root t is, it should be found between $0 < i < M$, which is also at most $\log_2 p$.

Therefore, the total time complexity will be the multiplication of those two, which gives $\log^2 p$.

Pseudocoe for Tonelli-Shanks Algorithm

Algorithm 1: Tonelli-Shanks

Input : A prime number p, and a congruence remainder n
Output: R, the square root of n modulo p

```
Factorize $p - 1$ into $Q \cdot 2^S$                /* $Q$ is odd */;
while The Jacobi symbol $\left(\frac{z}{p}\right) = 1$ do
    $z \leftarrow z + 1$;
end while
$M \leftarrow S \mod p$;
$c \leftarrow z^Q \mod p$;
$t \leftarrow n^Q \mod p$;
$R \leftarrow n^{\frac{Q+1}{2}} \mod p$;
while $M > 0$ do
    if $t = 0$ then
        return 0;
    end if
    if $t = 1$ then
        return $R$;
    end if
    $i \leftarrow 1$;
    while $t^{2^i} \neq 1 \mod$ and $i < M$ do
        $i \leftarrow i + 1$;
    end while
    if $i = M$ then
        return None        /* $n$ is not quadratic residue, so no $R$ is available */;
    end if
    $b \leftarrow c^{2^{M-i-1}}$;
    $M \leftarrow i$;
    $c \leftarrow b^2$;
    $t \leftarrow tb^2$;
    $R \leftarrow Rb$;
end while
return $R$;
```

References.

[1] Daniel Shanks. "Five Number Theoretic Algorithms". In: the Second Manitoba Conference on Numerical Mathematics, 1973, pp. 51–70 (cit. on p. 1).

0.2 Lempel Ziv Welch

- *Algorithm:* LZW Encoding (algo. 2), LZW Decoding (algo. 3)

- *Input:* A string for encoding, or a list of integers ranging from 0–4095 (ASCII representation) for decoding
- *Complexity:* $\mathcal{O}(n)$
- *Data structure compatibility:* N/A
- *Common applications:* Unix file compression and used in GIF image format
- *Note:* LZW stands for "Lempel Ziv Welch"

Problem. Lempel Ziv Welch

Encoding: Given a string s, compress it into a list of ASCII code.

Decoding: Given a list of ASCII code, recover the original string.

Description

Nowadays, almost everyone needs compression, the reasons for this are as follows:

- Original data without compression may occupy a lot space on disk, limiting the space to put important things. Also, without compression, if we want to download some files from the internet, it will take quite a long time since network speed is limited.
- Algorithms of reducing data size like compressing can help improve the development of hardwares.

In this sense, the Lempel-Ziv-Welch Algorithm is introduced as a lossless compression algorithm for reducing data size. In practice, an English text file with large size can be compressed using LZW and its size can be reduced to about half the original size.

Method of LZW compression/Encoding

LZW compression works as reading input of string containing different symbols, and then converting the strings of symbols into codes, usually ranging from 0 to 4095 [2]. To be more specific, the following steps show the exact procedure of LZW compression:

- First, create a code table, with 4096 the usual number of entries. Codes 0 – 255 represents the ASCII codes for all the symbols in character.
- Then read characters of the input string one by one. If the sequence of the previous character and the current one does exist in the table for corresponding code, then assign a new entry in the table of this sequence, and keep the sequence to the next iteration. If the sequence does not have a corresponding entry in the table, just output the code for the current sequence, and assign the sequence as the current character, and resume the remaining iterations [1].
- Finally, after reading the last character of the string, output the last sequence's code.

Method of LZW Decompression/decoding

The steps for LZW decompression is shown below:

- Create a default table containing indexes ranging from 0 – 255 and their corresponding characters.
- Then it operates similarly as the compression method, just reverse the code into original string.

Time Complexity

Since we just need the number of iterations equal to the length of input string, the time complexity for both compression and decompression is $\mathcal{O}(n)$.

Applications

The Graphics Interchange Format (GIF) uses LZW for encoding and decoding. Different codes in the color stand for different pixels, and all of which would be combined as an animation picture.

Pseudocode for LZW Encoding

Algorithm 2: LZW Encoding

```
Input  : A string of characters $s$
Output: a list of codes compressed from $s$
table $\leftarrow$ a table with 256 single characters as keys and 0 – 255 as values;
output $\leftarrow$ an empty list;
$c \leftarrow s[0]$;
index $\leftarrow 256$;
for $i = 1$ to $s.length - 1$ do
    $c' \leftarrow s[i]$;
    if $c + c'$ is in table then
        $c \leftarrow c + c'$;
    else
        push $c$ into output;
        table[$c + c'$] $\leftarrow$ index;
        index $\leftarrow$ index + 1;
        $c \leftarrow c'$;
    end if
end for
return output;
```

Pseudocode for LZW Decoding

Algorithm 3: LZW Decoding

Input : A list l of codes
Output: The original string s compressed into l

```
table ← a table with 0 – 255s as keys and 256 single character as values;
$s$ ← an empty string;
$c \leftarrow l[0]$;
$s \leftarrow s+$ table[$c$];
index ← 256;
for $i = 0$ to $l.length - 1$ do
    $c' \leftarrow l[i]$;
    if $c'$ not in table then
        temp ← table[$c$];
        temp ← temp + $x$;
    else
        temp ← table[$c'$];
    end if
    $s \leftarrow s+$ temp;
    $x \leftarrow$ temp[0];
    table[index] ← table[$c$] + $x$;
    index ← index + 1;
    $c \leftarrow c'$;
end for
return $s$;
```

References.

[1] Welch. “A technique for high-performance data compression”. In: *Computer* 17.6 (1984), pp. 8–19. DOI: 10.1109/mc.1984.1659158 (cit. on p. 3).

[2] J. Ziv and A. Lempel. “Compression of individual sequences via variable-rate coding”. In: *IEEE Transactions on Information Theory* 24.5 (1978), pp. 530–536. DOI: 10.1109/tit.1978.1055934 (cit. on p. 3).