# Modular square root: find r such that *r2 ≡ x* modulo *m*, given *x* and *m*

## Outline Procedure:

Step 1: Factorise ***m***; get a series of factors ***p1i1, p2i2***, ….

Step 2: For each unique prime factor of ***m***, solve ***r2 ≡ x*** modulo ***pi***. If ***x*** has no roots modulo any one of the prime factors, there are no roots modulo ***m***.

Step 3: Combine the roots, obtained separately for each unique prime factor in step 2, using the Chinese Remainder theorem.

Steps 1 and 3 will not be discussed further here as factorisation is a large subject, well described elsewhere and the Chinese Remainder Theorem is also well known, but step 2 has been analysed further. The Tonelli-Shanks method is well known but is only applicable for odd primes where ***x*** and ***p*** are mutually prime. Several methods are used, depending on the values of ***x*** and ***m***.

Note that whenever we find a solution ***r*** such that ***r2 ≡ x modulo m***, ***-r*** is also a solution. To avoid negative numbers, we can use ***m-r*** which is congruent to –r, modulo ***m***. If there are multiple solutions: r1, r2, ...then there is a corresponding solution –r1, -r2, … for each one.

## Step 2:

For each unique prime factor of ***m***, solve ***r2 ≡ x*** modulo ***pi***. If *x* is not in the range *0 ≤ x < pi*, ‘normalise’ it by adding or subtracting **pi** until it is in range.

If ***x = 0***, the smallest root is 0. Any other roots form an arithmetic progression where the common difference is the smallest power of ***p ≥ √pi = pi/2***, so additional solutions exist if ***I ≥ 2***.It is easy to see that this generates valid roots, not so obvious that it generates all the roots. *I have not seen this described anywhere, but it works.*

If ***x ≠ 0*** continue with 2(a) or 2(b) below; If ***pi*** and ***x*** are mutually prime go to step 2(a), if not go to step 2(b)

### 2(a) x ≠ 0, *pi* and *x* are mutually prime:

If ***p >2***, solve ***r2 ≡ x*** modulo ***p*** using Tonelli-Shanks or otherwise. Note that there are no solutions if the Legendre symbol (x/p) is not 1. ‘Lift’ the roots obtained from modulo ***p*** to modulo ***pi*** using the method from [Wkipedia](https://en.wikipedia.org/wiki/Tonelli%E2%80%93Shanks_algorithm#Tonelli's_algorithm_will_work_on_mod_p^k) or [Henshell’s](https://en.wikipedia.org/wiki/Hensel%27s_lemma) Lemma.

If ***p=2,*** there are solutions only if ***x ≡ 1*** modulo ***8***. Find a solution ***r1*** (see ***Note****(a)* below) then generate the other 3 solutions as follows:   
***r2 = pi – r1***

If ***r2 > r1; r3 = r1 + pi-1, r4 = r2 - pi-1***

If ***r2 < r1; r3 = r1 - pi-1, r4 = r2 + pi-1***

### 2(b) x ≠ 0, *pi* and *x* are not mutually prime:

*I worked out method 2(b) here myself. I cannot find a description anywhere, nor can I prove mathematically that it works in all cases.*

#### 2(b)1.

First calculate the GCD of ***pi*** and ***x***. The GCD will be a power of ***p***; ***pk*** where 0 < ***k < i***. If ***k*** is odd there are no solutions:

#### If k is even replace x with x/pk × pk , and find the roots of x/pk and pk separately. Solve r2 ≡ x/pk modulo pi as described in 2(a) above. Then:

#### 2(b)2.

#### Solve r2 ≡ pk modulo pi (k is even).

#### The smallest root r1 is pk/2 and the largest is pi - pk/2. The roots form 2 arithmetic progressions.

If ***p > 2*** let ***increment = pi-k/2***If ***p = 2*** let ***increment = max (pi-1 -k/2, p2+k/2)***   
 i.e. if ***i > k+3*** then ***increment = pi-1 -k/2*** otherwise ***increment = p2+k/2***

The 1st arithmetic progression is: ***r1, r1 + increment, r1 + 2 × increment, …*** while ***r1 + n × increment*** < ***pi.*** The second set of roots is ***pi – r1, pi – r2***, … etc.

Combine the roots from **2(b)1** and **2(b)2** by multiplying (modulo ***pi***) each of the roots obtained in **2(b)2** by the smallest root obtained in **2(b)1**. Any or all of the roots from step **2(b)1** could be used, but the same set of combined roots would be obtained each time.

## *Note(a)*:

To obtain a modular square root ***r*** such that ***r2 ≡ x (modulo 2i)***, where ***x ≡ 1*** (modulo 8)

Ref: <https://math.stackexchange.com/questions/845486/square-roots-modulo-powers-of-2> explains that there are 4 roots where x ≡ 1 (modulo 8): If a root ***r*** is known the other three roots are r+2i-1, -r, -r+2i-1.

Ref <https://www.johndcook.com/blog/quadratic_congruences/> explains, amongst other things:   
“Suppose xk2 ≡ a (mod 2k) for k ≥ 3. By definition, this means x2 – a is divisible by 2k. If (x2 – a)/2k is odd, let i = 1. Otherwise let i = 0. Then xk+1 defined by xk + i 2k-1 is a solution to xk+12 ≡ a (mod 2k+1).”

<https://projecteuler.chat/viewtopic.php?t=3506> is a discussion of how to obtain modular square roots: It includes sample code in Python. The relevant snippet of the code (based on John Cook’s method) is:

# Handle prime 2 special case

if p == 2:

if e >= 3 and a % 8 == 1:

res = []

for x in [1, 3]:

for k in xrange(3, e):

i = (x\*x - a)/(2\*\*k) % 2

x = x + i\*2\*\*(k-1)

res.append(x)

res.append(p\*\*e - x)

return res

# No solution if a is odd and a % 8 != 1

if e >= 3 and a % 2 == 1:

return []

# Force brut if a is even or e < 3 (for now)

return [x for x in xrange(0, p\*\*e) if x\*x % p\*\*e == a % p\*\*e]

This code does produce valid roots, but it generally only gives 2 of the four roots. The roots are duplicated.

My C++ code based on this is as follows:

/\* modulus AKA mod is 2^lambda, Znum is a typedef for a class of Extended Precision integers, pow2(k) returns 2^k \*/

else if ((lambda >= 3) && (c % 8 == 1)) {

x2 = 1; /\* x2 is modular square root of c (mod 2^k) \*/

for (int k = 3; k <= lambda; k++) {

Znum i = ((x2 \* x2 - c) / pow2(k)) % 2;

if (i == -1) i = 1;

x2 = x2 + i \* pow2(k - 1);

}

while (x2 < 0)

x2 += mod;

if (x2 >= mod)

x2 %= mod; /\* ensure x2 is in range 0 to mod-1 \*/

/\* x2 is 1 of 4 roots. We can easily generate the other 3 given any 1

of the 4 roots. \*/

roots.push\_back(x2);

roots.push\_back(mod - x2);

if (x2 < mod / 2) {

roots.push\_back(mod / 2 + x2);

roots.push\_back(mod / 2 - x2);

}

else {

roots.push\_back(x2 - mod / 2);

roots.push\_back(mod \* 3 / 2 - x2);

}

}

This code does generate all 4 roots. The roots are saved as values in the range 0 to ***2i ‑ 1***. The variables are extended precision integers so that integers > 264 can be handled.

## Note (b). Dario Alpern has developed a web page

<https://www.alpertron.com.ar/QUADMOD.HTM> which solves quadratic modular equations of the form **ax*² +* bx *+* c *≡ 0 (mod*n*)***. Given ***a, b, c***, and ***n*** it will calculate all the possible integer values of ***x*** in the range ***0 ≤* x< n**. In particular, it can find modular square roots by setting **a = -1**, **b = 0**, **c** = number whose root we want to find and **n** = modulus.

This has been used to verify independently the results of the method described above. He has not explained the methods he uses, but in the code there is a reference to a paper “Complete solving the quadratic equation mod 2^n of Dehnavi, Shamsabad and Rishakani[. https://arxiv.org/pdf/1711.03621.pdf](.%20https:/arxiv.org/pdf/1711.03621.pdf)”. This paper seems to cover the modulus = 2n case comprehensively.

## Example 1:

Find square roots of 36 modulo 7776:

Step 1: factorise 7776; 7776 = 25 × 35.

Step 2: find square roots of 36, modulo 32 (=25), and modulo 243( = 35)

Note that 36 ≡ 4 (modulo 32), so we look for the roots of 4 modulo 32

4 and 32 are not mutually prime so go to step 2(b). gcd(4, 32) = 4.

*If solving by hand the following step would be skipped because 4/4 = 1. However, the computer program proceeds as follows.*

Use step 2(a) to solve ***x2 ≡ 1 (mod 32)***   
modsqrt(1, 32) = 1, 31, 15, 17,

We only need the 1st root ***r=1***.

Use step2(b)2 to solve ***x2 ≡ 4 (mod 32)***

We have p=2, k=2, i=5, so common difference = 23 = 8.

So the 1st sequence of roots is 2, 10, 18, 26

The second set of roots is 32-r1, 32-r2, …etc. i.e. 30, 22, 14, 6.

So we have:

modsqrt(4, 32) = 2, 30, 10, 22, 18, 14, 26, 6,

Multiplying each root by 1 we have the same *(Working by hand this step would be skipped)*

modsqrt(4, 32) = ***2, 30, 10, 22, 18, 14, 26, 6***. More concisely: the roots are ±2 (mod 4)

Use step 2(b) to solve x2 ≡ 36 modulo 243.

Gcd(36, 243) = 9, so we need to solve ***x2 ≡ 4 modulo 243*** and ***x2 ≡ 9 modulo 243***  
To solve ***x2 ≡ 4 modulo 243*** use step 2(a). We have 243 = 35, so we first solve ***x2 ≡ 4 modulo 3***, then ‘lift’ the roots from modulo 3 to modulo 35.

4 ≡ 1 (modulo 3), so modsqrt(4, 3) ≡ modsqrt(1,3)

modsqrt(1, 3) = 1, 2, (trivial, don’t need Tonelli-Shanks)

***modsqrt(4, 243) = 241, 2***, (roots ‘lifted’)

Solve ***x2 ≡ 9 modulo 243*** using step2(b)2.  
We have p=3, k=2, i=5, so common difference = 34 = 81, smallest root = 3k/2 = 3

The 1st series of roots is 3, 84, 165, and the 2nd series is 243-3 = ***240***, 243-84 = ***159***, and 243 ‑ 165 = ***78***.

***modsqrt(9, 243) = 3, 240, 84, 159, 165, 78***

To solve ***x2 ≡ 36 modulo 243*** we multiply each of the roots ***3, 240, 84, 159, 165, 78*** by either ***241*** or ***2*** modulo 243. If we use 241 we get: 237, 6, 75, 168, 156, 87. If we use 2 we get 6, 237, 168, 75, 87, 56 i.e. we get the same set of roots in a different order.

To recap we now have roots modulo 32 = **2, 30, 10, 22, 18, 14, 26, 6** and roots modulo 243= **237, 6, 75, 168, 156, 87**. i.e. ±6 mod 81

In step 3 we use the Chinese Remainder Theorem to combine these 2 sets of roots to get roots modulo 7776, so we get 6 × 8 = 48 roots. E.g. 5826 ≡ 2 modulo 32 and 5826 ≡ 237 modulo 243,

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Step 3** | **237** | **6** | **75** | **168** | **156** | **87** |
| **2** | 5826 | 4866 | 3234 | 7458 | 642 | 2274 |
| **30** | 2910 | 1950 | 318 | 4542 | 5502 | 7134 |
| **10** | 3882 | 2922 | 1290 | 5514 | 6474 | 330 |
| **22** | 4854 | 3894 | 2262 | 6486 | 7446 | 1302 |
| **18** | 1938 | 978 | 7122 | 3570 | 4530 | 6162 |
| **14** | 6798 | 5838 | 4206 | 654 | 1614 | 3246 |
| **26** | 7770 | 6810 | 5178 | 1626 | 2586 | 4218 |
| **6** | 966 | **6** | 6150 | 2598 | 3558 | 5190 |

Interestingly, the roots can be sorted into two arithmetic progressions: 6, 330, 654, … 7458, and 318, 642, 996, … 7770, each with a common difference of 324 = 4x81.

## Example 2

Find the square roots of 17 modulo 512

Step 1: factorise the modulus: 512 = 29

Step 2: solve x2 ≡ 17 modulo 29

Find a solution as described in Note(a):

Firstly ***8x+1 ≡ 1*** modulo 8, so modsqrt(8x+1, 8) ≡ modsqrt(1,8) ≡ 1, 3, 5, 7, for any integer x. We only need one root so choose 1.

modsqrt(17, 8) = 1

modsqrt(17, 16) = 9

modsqrt(17, 32) = 9

modsqrt(17, 64) = 41

modsqrt(17, 128) = 105

modsqrt(17, 256) = 233

modsqrt(17, 512) = 233

There are actually 4 solutions at each step, but we only need to use one.

Given ***r1 = 233***, we get ***r2 = 512-233= 279, r3 = 256 +233= 489, r4 = 256-233 = 23***, thus giving us the four solutions.

## Example 3

Find the square roots of 0 modulo 7776

Step 1: factorise 7776; 7776 = 25 × 35.

Step 2: find square roots of 0, modulo 32 (=25), and modulo 243( = 35)

Find square roots modulo 2^5. The roots are an arithmetic progression where the 1st term is 0 and the common difference is 23 = 8, so the roots are 0, 8, 16, 24

Find square roots modulo 3^5. The roots are an arithmetic progression where the 1st term is 0 and the common difference is 33 = 27, so the roots are 0, 27, 54, 81, 108, 135, 162, 189, 216.

In Step 3, combining these two sets of roots using the Chinese Remainder Theorem gives us us 4×9 = 36 roots: 0, 216, 432, 648, 864, 1080, 1296, 1512, 1728, 1944, 2160, 2376, 2592, 2808, 3024, 3240, 3456, 3672, 3888, 4104, 4320, 4536, 4752, 4968, 5184, 5400, 5616, 5832, 6048, 6264, 6480, 6696, 6912, 7128, 7344, 7560. Again, these roots form an arithmetic progression, where the common difference is 216 = 8×27.