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
Català
Čeština
Deutsch
Español
Français
Polski
Русский
Simple English
Svenska
Українська
Tiếng Việt
Edit links

Exponentiation by squaring

From Wikipedia, the free encyclopedia
(Redirected from Exponentiating by squaring)

In mathematics and computer programming, **exponentiating by squaring** is a general method for fast computation of large positive integer powers of a number, or more generally of an element of a semigroup, like a polynomial or a square matrix. Some variants are commonly referred to as **square-and-multiply** algorithms or **binary exponentiation**. These can be of quite general use, for example in modular arithmetic or powering of matrices. For semigroups for which additive notation is commonly used, like elliptic curves used in cryptography, this method is also referred to as **double-and-add**.

Contents [hide]

Basic method [edit]

The method is based on the observation that, for a positive integer *n*, we have

$$x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

This may be easily implemented as the following recursive algorithm:

```
Function exp-by-squaring(x, n )
  if n < 0  then return exp-by-squaring(1 / x, - n );
  else if n = 0  then return  1;
  else if n = 1  then return  x ;
  else if n is even  then return exp-by-squaring(x * x,  n / 2);
  else if n is odd  then return x * exp-by-squaring(x * x, (n - 1) / 2).
```

Although not tail-recursive, this algorithm may be rewritten into a tail recursive algorithm by introducing an auxiliary function:

```
Function exp-by-squaring(x, n)
  exp-by-squaring2(1, x, n)
Function exp-by-squaring2(y, x, n)
  if n < 0  then return exp-by-squaring2(y, 1 / x, - n);
```

```
    else if n = 0  then return  y;
    else if n = 1  then return  x * y;
    else if n is even  then return exp-by-squaring2(y, x * x,  n / 2);
    else if n is odd  then return exp-by-squaring2(x * y, x * x, (n - 1) / 2).
```

The iterative version of the algorithm also uses a bounded auxiliary space, and is given by

```
  Function exp-by-squaring-iterative(x, n)
    if n < 0 then
      x := 1 / x;
      n := -n;
    if n = 0 then return 1
    y := 1;
    while n > 1 do
      if n is even then
        x := x * x;
        n := n / 2;
      else
        y := x * y
        x := x * x;
        n := (n - 1) / 2;
    return x * y
```

Computational complexity [edit]

A brief analysis shows that such an algorithm uses $\lfloor \log_2 n \rfloor$ squarings and at most $\lfloor \log_2 n \rfloor$ multiplications, where $\lfloor \; \rfloor$ denotes the floor function. More precisely, the number of multiplications is one less than the number of 1 in the binary expansion of *n*. For *n* greater than about 4 this is computationally more efficient than naively multiplying the base with itself repeatedly.

Each squaring results in approximately double the number of digits of the previous, and so, if multiplication of two *d* digit numbers is implemented in O(d^k) operations for some fixed *k* then the complexity of computing x^n is given by:

$$\sum_{i=0}^{O(\log(n))} (2^i O(\log(x)))^k = O((n\log(x))^k)$$

2^k-ary method [edit]

This algorithm calculates the value of x^n after expanding the exponent in base 2^k. It was first proposed by Brauer in 1939. In the algorithm below we make use of the following function f(0) = (k,0) and f(m) = (s,u) where m = u·2^s with *u* odd.

Algorithm:

Input

An element x of G, a parameter k > 0, a non-negative integer $n = (n_{l-1}, n_{l-2}, \ldots, n_0)_{2^k}$ and the precomputed values $x^3, x^5, \ldots, x^{2^k-1}$.

Output

The element x^n in *G*

```
 1. y := 1; i := l-1
 2. while i>=0 do
 3.    (s,u) := f($n_i$)
 4.    for j:=1 to k-s do
 5.        y := $y^2$
 6.    y := y*$x^u$
 7.    for j:=1 to s do
 8.        y := $y^2$
 9.    i := i-1
10. return y
```

For optimal efficiency, *k* should be the smallest integer satisfying [1]

$$\log(n) < \frac{k(k+1)\cdot 2^{2k}}{2^{k+1}-k-2}+1.$$

Sliding window method [edit]

This method is an efficient variant of the 2^k-ary method. For example, to calculate the exponent 398 which has binary expansion $(110\ 001\ 110)_2$, we take a window of length 3 using the 2^k-ary method algorithm we calculate $1,x^3,x^6,x^{12},x^{24},x^{48},x^{49},x^{98},x^{99},x^{198},x^{199},x^{398}$. But, we can also compute $1,x^3,x^6,x^{12},x^{24},x^{48},x^{96},x^{192},x^{199}$, x^{398} which saves one multiplication and amounts to evaluating $(110\ 001\ 110)n_2$

Here is the general algorithm:

Algorithm:

Input

An element *x* of *G*,a non negative integer $n=(n_l,n_{l-1},...,n_0)_2$, a parameter k>0 and the pre-computed values $x^3, x^5, ..., x^{2^k-1}$.

Output

The element $x^n \in G$

Algorithm:

```
y := 1; i := l-1
while i > -1 do
    if $n_i$=0 then $y:=y^2$' i:=i-1
    else
        s:=max{i-k+1,0}
        while $n_s$=0 do s:=s+1 [2]
        for h:=1 to i-s+1 do $y:=y^2$
        $u:=(n_i,n_{i-1},....,n_s)_2$
        $y:=y*x^u$
        i:=s-1
return y
```

Montgomery's ladder technique [edit]

Many algorithms for exponentiation do not provide defence against side-channel attacks. Namely, an attacker observing the sequence of squarings and multiplications can (partially) recover the exponent involved in the computation. This is a problem if the exponent should remain secret, as with many public-key cryptosystems. A technique called Montgomery's Ladder[3] addresses this concern.

Given the binary expansion of a positive, non-zero integer $n=(n_{k-1}...n_0)_2$ with $n_{k-1}=1$ we can compute x^n as follows:

```
$x_1$=x; $x_2=x^2$
for i=k-2 to 0 do
  If $n_i$=0 then
    $x_2=x_1*x_2$; $x_1=x_1^2$
  else
    $x_1=x_1*x_2$; $x_2=x_2^2$
return $x_1$
```

The algorithm performs a fixed sequence of operations (up to log n): a multiplication and squaring takes place for each bit in the exponent, regardless of the bit's specific value.

This specific implementation of Montgomery's ladder is not yet protected against cache timing attacks: memory access latencies might still be observable to an attacker as you access different variables depending on the value of bits of the secret exponent.

Fixed base exponent [edit]

There are several methods which can be employed to calculate x^n when the base is fixed and the exponent varies. As one can see, precomputations play a key role in these algorithms.

Yao's method [edit]

Yao's method is orthogonal to the 2^k-ary method where the exponent is expanded in radix $b=2^k$ and the computation is as performed in the algorithm above. Let "n", "n_i", "b", and "b_i" be integers.

Let the exponent "n" be written as

$$n = \sum_{i=0}^{w-1} n_i b_i \text{ where } 0 \leqslant n_i < h \text{ for all } i \in [0, w-1]$$

Let $x_i = x^{b_i}$. Then the algorithm uses the equality

$$x^n = \prod_{i=0}^{w-1} {x_i}^{n_i} = \prod_{j=1}^{h-1} \left[\prod_{n_i=j} x_i \right]^j$$

Given the element 'x' of G, and the exponent 'n' written in the above form, along with the precomputed values x^{b_0}....$x^{b_{w-1}}$ the element x^n is calculated using the algorithm below.

```
1. y=1,u=1 and j=h-1
2. while j > 0 do
    1. for i=0 to w-1 do
        1. if $n_i$=j then u=u*$x^{b_i}$
    2. y=y*u
    3. j=j-1
3. return y
```

If we set $h=2^k$ and $b_i = h^i$ then the n_i 's are simply the digits of n in base h. Yao's method collects in u first those x_i which appear to the highest power h-1; in the next round those with power h-2 are collected in u as well etc. The variable y is multiplied h-1 times with the initial u, h-2 times with the next highest powers etc. The algorithm uses w+h-2 multiplications and w+1 elements must be stored to compute x^n (see [1]).

Euclidean method [edit]

The Euclidean method was first introduced in *Efficient exponentiation using precomputation and vector addition chains* by P.D Rooij.

This method for computing x^n in group $\mathbf{G}$, where n is a natural integer, whose algorithm is given below, is using the following equality recursively:

$$x_0^{n_0} \cdot x_1^{n_1} = (x_0 \cdot x_1^{q})^{n_0} \cdot x_1^{n_1 \mod n_0}, \text{ where } q = \left\lfloor \frac{n_1}{n_0} \right\rfloor$$

(in other words a Euclidean division of the exponent n_1 by n_0 is used to return a quotient q and a rest $n_1 \bmod n_0$).

Given the base element x in group $\mathbf{G}$, and the exponent n written as in Yao's method, the element x^n is calculated using l precomputed values $x^{b_0}, ..., x^{b_{l_i}}$ and then the algorithm below.

```
Begin loop
    Find $M \in [0, l-1]$, such that $\forall i \in [0, l-1], n_M \geq n_i$;
    Find $N \in ([0, l-1] - M)$, such that $\forall i \in ([0, l-1] - M), n_N \geq n_i$;
    Break loop if $n_N = 0$;
    Let $q = \lfloor n_M / n_N \rfloor$, and then let $n_N = (n_M \mod n_N)$;
    Compute recursively $x_M^q$, and then let $x_N = x_N \cdot x_M^q$; End loop;
Return $x^n = x_M^{n_M}$.
```

The algorithm first finds the largest value amongst the n_i and then the supremum within the set of $\{ n_i \setminus i \neq M \}$. Then it raises x_M to the power q, multiplies this value with x_N, and then assigns x_N the result of this computation and n_M the value n_M modulo n_N.

Further applications [edit]

The same idea allows fast computation of large exponents modulo a number. Especially in cryptography, it is useful to compute powers in a ring of integers modulo *q*. It can also be used to compute integer powers in a group, using the rule

Power(*x*, −*n*) = (Power(*x*, *n*))$^{-1}$.

The method works in every semigroup and is often used to compute powers of matrices.

For example, the evaluation of

13789^{722341} (mod 2345)

would take a very long time and lots of storage space if the naïve method were used: compute 13789^{722341} then take the remainder when divided by 2345. Even using a more effective method will take a long time: square 13789, take the remainder when divided by 2345, multiply the result by 13789, and so on. This will take less than $2\log_2(722340) \leq 40$ modular multiplications.

Applying above *exp-by-squaring* algorithm, with "*" interpreted as *x*y* = *xy* mod 2345 (that is a multiplication followed by a division with remainder) leads to only 27 multiplications and divisions of integers which may all be stored in a single machine word.

Example implementations [edit]

Computation by powers of 2 [edit]

This is a non-recursive implementation of the above algorithm in Ruby.

In most statically typed languages, `result=1` must be replaced with code assigning an identity matrix of the same size as `x` to `result` to get a matrix exponentiating algorithm. In Ruby, thanks to coercion, `result` is automatically upgraded to the appropriate type, so this function works with matrices as well as with integers and floats. Note that n=n-1 is redundant when n=n/2 implicitly rounds towards zero, as lower level languages would do. n[0] is the rightmost bit of the binary representation of n, so if it is 1, the number is odd, if it is zero, the number is even.

```ruby
def power(x,n)
  result = 1
  while n.nonzero?
    if n[0].nonzero?
      result *= x
      n -= 1
    end
    x *= x
    n /= 2
  end
  return result
end
```

Runtime example: compute 3^{10} [edit]

```
parameter x =  3
parameter n = 10
result := 1

Iteration 1
  n = 10 -> n is even
  x := x^2 = 3^2 = 9
  n := n / 2 = 5

Iteration 2
  n = 5 -> n is odd
      -> result := result * x = 1 * x = 1 * 3^2 = 9
         n := n - 1 = 4
  x := x^2 = 9^2 = 3^4 = 81
  n := n / 2 = 2

Iteration 3
  n = 2 -> n is even
  x := x^2 = 81^2 = 3^8 = 6561
  n := n / 2 = 1

Iteration 4
  n = 1 -> n is odd
      -> result := result * x = 3^2 * 3^8 = 3^10 = 9 * 6561 = 59049
         n := n - 1 = 0

return result
```

Runtime example: compute 3^{10} [edit]

```
result := 3
bin := "1010"

Iteration for digit 2:
  result := result^2 = 3^2 = 9
  1010bin - Digit equals "0"

Iteration for digit 3:
  result := result^2 = (3^2)^2 = 3^4  = 81
  1010bin - Digit equals "1" --> result := result*3 = (3^2)^2*3 = 3^5  = 243

Iteration for digit 4:
  result := result^2 = ((3^2)^2*3)^2 = 3^10  = 59049
  1010bin - Digit equals "0"

return result
```

JavaScript-Demonstration: http://home.mnet-online.de/wzwz.de/temp/ebs/en.htm

Calculation of products of powers [edit]

Exponentiation by squaring may also be used to calculate the product of 2 or more powers. If the underlying group or semigroup is commutative then it is often possible to reduce the number of multiplications by computing the product simultaneously.

Example [edit]

The formula $a^7 \times b^5$ may be calculated within 3 steps:

$((a)^2 \times a)^2 \times a$ (four multiplications for calculating a^7)
$((b)^2)^2 \times b$ (three multiplications for calculating b^5)
$(a^7) \times (b^5)$ (one multiplication to calculate the product of the two)

so one gets eight multiplications in total.

A faster solution is to calculate both powers simultaneously:

$((a \times b)^2 \times a)^2 \times a \times b$

which needs only 6 multiplications in total. Note that a×b is calculated twice, the result could be stored after the first calculation which reduces the count of multiplication to 5.

Example with numbers:

$2^7 \times 3^5 = ((2\times3)^2\times2)^2\times2\times3 = (6^2\times2)^2\times6 = 72^2\times6 = 31{,}104$

Calculating the powers simultaneously instead of calculating them separately always reduces the count of multiplications if at least two of the exponents are greater than 1.

Using transformation [edit]

The example above $a^7 \times b^5$ may also be calculated with only 5 multiplications if the expression is transformed before calculation:

$a^7 \times b^5 = a^2 \times (ab)^5$ with ab := a×b

ab := a×b (one multiplication)
$a^2 \times (ab)^5 = ((ab)^2 \times a)^2 \times ab$ (four multiplications)

Generalization of transformation shows the following scheme:
For calculating $a^A \times b^B \times \ldots \times m^M \times n^N$
1st: define ab := a×b, abc = ab×c, ...
2nd: calculate the transformed expression $a^{A-B} \times ab^{B-C} \times \ldots \times abc..m^{M-N} \times abc..mn^N$

Transformation before calculation often reduces the count of multiplications but in some cases it also increases the count (see the last one of the examples below), so it may be a good idea to check the count of multiplications before using the transformed expression for calculation.

Examples [edit]

For the following expressions the count of multiplications is shown for calculating each power separately, calculating them simultaneously without transformation and calculating them simultaneously after transformation.

Example: $a^7 \times b^5 \times c^3$
separate: $[((a)^2 \times a)^2 \times a] \times [((b)^2)^2 \times b] \times [(c)^2 \times c]$ (**11** multiplications)
simultaneous: $((a \times b)^2 \times a \times c)^2 \times a \times b \times c$ (**8** multiplications)
transformation: a := 2   ab := a×b   abc := ab×c (2 multiplications)
calculation after that: $(a \times ab \times abc)^2 \times abc$ (4 multiplications ⇒ **6** in total)

Example: $a^5 \times b^5 \times c^3$
separate: $[((a)^2)^2 \times a] \times [((b)^2)^2 \times b] \times [(c)^2 \times c]$ (**10** multiplications)
simultaneous: $((a \times b)^2 \times c)^2 \times a \times b \times c$ (**7** multiplications)
transformation: a := 2   ab := a×b   abc := ab×c (2 multiplications)
calculation after that: $(ab \times abc)^2 \times abc$ (3 multiplications ⇒ **5** in total)

Example: $a^7 \times b^4 \times c^1$
separate: $[((a)^2 \times a)^2 \times a] \times [((b)^2)^2] \times [c]$ (**8** multiplications)
simultaneous: $((a \times b)^2 \times a)^2 \times a \times c$ (**6** multiplications)
transformation: a := 2   ab := a×b   abc := ab×c (2 multiplications)
calculation after that: $(a \times ab)^2 \times a \times ab \times abc$ (5 multiplications ⇒ **7** in total)

Signed-digit recoding [edit]

In certain computations it may be more efficient to allow negative coefficients and hence use the inverse of the base, provided inversion in G is 'fast' or has been precomputed. For example, when computing x^{2^k-1} the binary method requires k−1 multiplications and k−1 squarings. However one could perform k squarings to get x^{2^k} and then multiply by x^{-1} to obtain x^{2^k-1}.

To this end we define the signed-digit representation of an integer n in radix b as

$$n = \sum_{i=0}^{l-1} n_i b^i \text{ with } |n_i| < b$$

Signed binary representation corresponds to the particular choice $b = 2$ and $n_i \in \{-1, 0, 1\}$. It is denoted by $(n_{l-1} \dots n_0)_s$. There are several methods for computing this representation. The representation is not unique, for example take $n = 478$. Two distinct signed-binary representations are given by $(10\bar{1}1100\bar{1}10)_s$ and $(100\bar{1}1000\bar{1}0)_s$, where $\bar{1}$ is used to denote -1. Since the binary method computes a multiplication for every non-zero entry in the base 2 representation of n, we are interested in finding the signed-binary representation with the smallest number of non-zero entries, that is, the one with *minimal* Hamming weight. One method of doing this is to compute the representation in non-adjacent form, or NAF for short, which is one that satisfies $n_i n_{i+1} = 0 \text{ for all } i \geqslant 0$ and denoted by $(n_{l-1} \dots n_0)_{\text{NAF}}$. For example the NAF representation of 478 is equal to $(1000\bar{1}000\bar{1}0)_{\text{NAF}}$. This representation always has minimal Hamming weight. A simple algorithm to compute the NAF representation of a given integer $n = (n_l n_{l-1} \dots n_0)_2$ with $n_l = n_{l-1} = 0$ is the following:

1. $c_0 = 0$
2. for $i = 0$ to $l - 1$ do
3. $c_{i+1} = \left\lfloor \frac{1}{2}(c_i + n_i + n_{i+1}) \right\rfloor$
4. $n_i' = c_i + n_i - 2c_{i+1}$
5. return $(n_{l-1}' \dots n_0')_{\text{NAF}}$

Another algorithm by Koyama and Tsuruoka does not require the condition that $n_i = n_{i+1} = 0$; it still minimizes the Hamming weight.

Alternatives and generalizations [edit]

Main article: Addition-chain exponentiation

Exponentiation by squaring can be viewed as a suboptimal addition-chain exponentiation algorithm: it computes the exponent via an addition chain consisting of repeated exponent doublings (squarings) and/or incrementing exponents by *one* (multiplying by *x*) only. More generally, if one allows *any* previously computed exponents to be summed (by multiplying those powers of *x*), one can sometimes perform the exponentiation using fewer multiplications (but typically using more memory). The smallest power where this occurs is for *n*=15:

$a^{15} = x \times (x \times [x \times x^2]^2)^2$ (squaring, 6 multiplies)
$a^{15} = x^3 \times ([x^3]^2)^2$ (optimal addition chain, 5 multiplies if x^3 is re-used)

In general, finding the *optimal* addition chain for a given exponent is a hard problem, for which no efficient algorithms are known, so optimal chains are typically only used for small exponents (e.g. in compilers where the chains for small powers have been pre-tabulated). However, there are a number of heuristic algorithms that, while not being optimal, have fewer multiplications than exponentiation by squaring at the cost of additional bookkeeping work and memory usage. Regardless, the number of multiplications never grows more slowly than Θ(log *n*), so these algorithms only improve asymptotically upon exponentiation by squaring by a constant factor at best.

See also [edit]

- Modular exponentiation
- Vectorial addition chain
- Montgomery reduction
- Non-adjacent form
- Addition chain

Notes [edit]

1. ^ *a* *b* Cohen, H., Frey, G. (editors): Handbook of elliptic and hyperelliptic curve cryptography. Discrete Math.Appl., Chapman & Hall/CRC (2006)
2. ^ In this line, the loop finds the longest string of length less than or equal to 'k' which ends in a non zero value. And not all odd powers of 2 up to x^{2^k-1} need be computed and only those specifically involved in the computation need be considered.
3. ^ Montgomery, P. L. "Speeding the Pollard and Elliptic Curve Methods of Factorization." Math. Comput. 48, 243-264, 1987.

Categories: Exponentials | Computer arithmetic algorithms | Computer arithmetic