# Baby-step giant-step

From Wikipedia, the free encyclopedia

In group theory, a branch of mathematics, the **baby-step giant-step** is a meet-in-the-middle algorithm computing the discrete logarithm. The discrete log problem is of fundamental importance to the area of public key cryptography. Many of the most commonly used cryptography systems are based on the assumption that the discrete log is extremely difficult to compute; the more difficult it is, the more security it provides a data transfer. One way to increase the difficulty of the discrete log problem is to base the cryptosystem on a larger group.

**Contents** [hide]



## Theory [edit]

The algorithm is based on a space-time tradeoff. It is a fairly simple modification of trial multiplication, the naive method of finding discrete logarithms.

Given a cyclic group $G$ of order $n$, a generator $\alpha$ of the group and a group element $\beta$, the problem is to find an integer $x$ such that

$$\alpha^x = \beta\,.$$

The baby-step giant-step algorithm is based on rewriting $x$ as $x = im + j$, with $m = \left\lceil \sqrt{n} \right\rceil$ and $0 \leq i < m$ and $0 \leq j < m$. Therefore, we have:

$$\beta(\alpha^{-m})^i = \alpha^j\,.$$

The algorithm precomputes $\alpha^j$ for several values of $j$. Then it fixes an $m$ and tries values of $i$ in the left-hand side of the congruence above, in the manner of trial multiplication. It tests to see if the congruence is satisfied for any value of $j$, using the precomputed values of $\alpha^j$.

## The algorithm [edit]

**Input**: A cyclic group *G* of order *n*, having a generator α and an element β.

**Output**: A value *x* satisfying $\alpha^x = \beta$.

1. $m \leftarrow \text{Ceiling}(\sqrt{n})$
2. For all *j* where $0 \leq j < m$:
   1. Compute $\alpha^j$ and store the pair $(j, \alpha^j)$ in a table. (See section "In practice")
3. Compute $\alpha^{-m}$.
4. $\gamma \leftarrow \beta$. (set $\gamma = \beta$)
5. For $i = 0$ to $(m - 1)$:
   1. Check to see if γ is the second component ($\alpha^j$) of any pair in the table.
   2. If so, return $im + j$.
   3. If not, $\gamma \leftarrow \gamma \cdot \alpha^{-m}$.

### C algorithm with the GNU MP lib [edit]

```
void baby_step_giant_step (mpz_t g, mpz_t h, mpz_t p, mpz_t n, mpz_t x ){
   unsigned long int i;
```

```
    long int j = 0;
    mpz_t N;
    mpz_t* gr ; /* list g^r */
    unsigned long int* indices; /* indice[ i ] = k <=> gr[ i ] = g^k */
    mpz_t hgNq ; /* hg^(Nq) */
    mpz_t inv ; /* inverse of g^(N) */
    mpz_init (N) ;
    mpz_sqrt (N, n ) ;
    mpz_add ui (N, N, 1 ) ;

    gr = malloc (mpz_get_ui (N) * sizeof (mpz_t) ) ;
    indices = malloc ( mpz_get_ui (N) * sizeof (long int ) ) ;
    mpz_init_set_ui (gr[ 0 ], 1);

    /* find the sequence {g^r} r = 1 ,.. ,N (Baby step ) */
    for ( i = 1 ; i <= mpz_get_ui (N) ; i++) {
       indices[i - 1] = i - 1 ;
       mpz_init (gr[ i ]) ;
       mpz_mul (gr[ i ], gr[ i - 1 ], g ); /* multiply gr[i - 1] for g */
       mpz_mod (gr[ i ], gr[ i ], p );
    }
    /* sort the values (k , g^k) with respect to g^k */
    qsort ( gr, indices, mpz_get_ui (N), mpz_cmp ) ;
    /* compute g^(-Nq)   (Giant step) */
    mpz_init_set (inv, g);
    mpz_powm (inv, inv, N, p);  /* inv <- inv ^ N (mod p)  */
    mpz_invert (inv, p, inv) ;

    mpz_init_set (hgNq, h);

    /* find the elements in the two sequences */
    for ( i = 0 ; i <= mpz_get_ui (N) ; i++){
       /* find hgNq in the sequence gr ) */
       j = bsearch (gr, hgNq, 0, mpz_get_ui (N), mpz_cmp ) ;
       if ( j >= 0 ){
          mpz_mul_ui (N, N, i);
          mpz_add_ui (N, N, indices [j]);
          mpz_set (x, N) ;
          break;
       }
       /* if j < 0, find the next value of g^(Nq) */
       mpz_mul (hgNq, hgNq, inv);
       mpz_mod (hgNq, hgNq, p);
    }
}
```

## In practice [edit]

The best way to speed up the baby-step giant-step algorithm is to use an efficient table lookup scheme. The best in this case is a hash table. The hashing is done on the second component, and to perform the check in step 1 of the main loop, γ is hashed and the resulting memory address checked. Since hash tables can retrieve and add elements in O(1) time (constant time), this does not slow down the overall baby-step giant-step algorithm.

The running time of the algorithm and the space complexity is O($\sqrt{n}$), much better than the O(n) running time of the naive brute force calculation.

## Notes [edit]

- The baby-step giant-step algorithm is a generic algorithm. It works for every finite cyclic group.
- It is not necessary to know the order of the group *G* in advance. The algorithm still works if *n* is merely an upper bound on the group order.
- Usually the baby-step giant-step algorithm is used for groups whose order is prime. If the order of the group is composite then the Pohlig–Hellman algorithm is more efficient.
- The algorithm requires O(*m*) memory. It is possible to use less memory by choosing a smaller *m* in the first step of the algorithm. Doing so increases the running time, which then is O(*n*/*m*). Alternatively one can use Pollard's rho algorithm for logarithms, which has about the same running time as the baby-step giant-step algorithm, but only a small memory requirement.

- The algorithm was originally developed by Daniel Shanks.

## References [edit]


- H. Cohen, A course in computational algebraic number theory, Springer, 1996.
- D. Shanks. Class number, a theory of factorization and genera. In Proc. Symp. Pure Math. 20, pages 415—440. AMS, Providence, R.I., 1971.
- A. Stein and E. Teske, Optimized baby step-giant step methods, Journal of the Ramanujan Mathematical Society 20 (2005), no. 1, 1–32.
- A. V. Sutherland, Order computations in generic groups, PhD thesis, M.I.T., 2007.
- D. C. Terr, A modification of Shanks' baby-step giant-step algorithm, Mathematics of Computation 69 (2000), 767–773.


| V · T · E | Number-theoretic algorithms [hide] |
|---|---|
| **Primality tests** | AKS TEST · APR TEST · Baillie–PSW · ECPP TEST · Elliptic curve · Pocklington · Fermat · Lucas · *LUCAS–LEHMER* · *LUCAS–LEHMER–RIESEL* · *PROTH'S THEOREM* · *PÉPIN'S* · Quadratic Frobenius test · Solovay–Strassen · Miller–Rabin |
| **Prime-generating** | Sieve of Atkin · Sieve of Eratosthenes · Sieve of Sundaram · Wheel factorization |
| **Integer factorization** | Continued fraction (CFRAC) · Dixon's · Lenstra elliptic curve (ECM) · Euler's · Pollard's rho · $p-1$ · $p+1$ · Quadratic sieve (QS) · General number field sieve (GNFS) · *Special number field sieve (SNFS)* · Rational sieve · Fermat's · Shanks' square forms · Trial division · Shor's |
| **Multiplication** | Ancient Egyptian · Long · Karatsuba · Toom–Cook · Schönhage–Strassen · Fürer's |
| **Discrete logarithm** | **Baby-step giant-step** · Pollard rho · Pollard kangaroo · POHLIG–HELLMAN · Index calculus · Function field sieve |
| **Greatest common divisor** | Binary · Euclidean · Extended Euclidean · Lehmer's |
| **Modular square root** | Cipolla · Pocklington's · Tonelli–Shanks |
| **Other algorithms** | Chakravala · Cornacchia · Integer relation · Integer square root · Modular exponentiation · Schoof's |
| | *Italics* indicate that algorithm is for numbers of special forms · SMALLCAPS indicate a deterministic algorithm |

Categories: Group theory | Number theoretic algorithms