Discrete logarithm computation with generic algorithms.

- Section 9.2 of Monain
- Chapter 5 of Washington
- Shanks Baby Step Giant Step (5.2.1)
- Pollard ρ: inspired by Floyd path finding algorithm monograph.
- Pollard - Shalkman: 5.2.3.

These algorithms apply to generic groups $G$: it can be the multiplicative subgroup of a finite field, or the group of points of an elliptic curve over a finite field.

Notation: $G$ is a group with a multiplication: $x, y \in G \rightarrow x \cdot y \in G$.

$\Rightarrow$ we have exponentiation in $G$: $x^n$ where $n \in \mathbb{Z}$,

$x^0 = 1$ (the neutral element of $G$; this is $1$ in $\mathbb{F}_q^*$, this is $0$ on $E(\mathbb{F}_p)$).

$x^{-n} = (1/x)^n$ and inversion is in the usual sense in $\mathbb{F}_q^*$,

inversion is $-P = (x_p, -y_p)$ on an elliptic curve.

Discrete logarithm ($D$-$\log$): given $G$, a generator $g$, and a target $h \in G$,

compute $x$ in $\mathbb{Z}$, $1, \ldots, |G| - 1$ such that $g^x = h$ in $G$.

Example: if $G = \mathbb{F}_p^*$ the multiplicative subgroup of a finite field where $p$ is prime,

then $|G| = p - 1$ and the discrete log is such that $g^x = h$ in $\mathbb{F}_p^*$ (mod $p$)

- if $G = E(\mathbb{F}_p)$, then $|G| = |E(\mathbb{F}_p)| = p + 1 - t$ where $t$ is the trace of the curve,

and if $P$ is a generator of $E(\mathbb{F}_p)$ (of order $p + 1 - t = q$), then the $D$-$\log$ is such that

$Q \in E(\mathbb{F}_p)$ has $D$-$\log \ [\log_p Q] P = Q$ where exponentiation becomes scalar multiplication.

In Sage Math, $P \cdot \text{order}(P)$ computes $\mathbb{Z}$, the order of the point (costly).

for a point $G \in E$ and another point $P \in E$,

$G \cdot \text{discrete_log}(P)$ computes the discrete log of $P$ in basis $G$ (costly too).
Polihy-Hellman method.

Group homomorphism. Let \( G \) be a multiplicative group of order \( N \).

Suppose we can factor \( N \) into distinct prime factors:

\[
N = \prod_{i} p_i^{e_i} = p_1^{e_1} \cdot p_2^{e_2} \cdots p_n^{e_n}
\]

One can map \( g, h \in G \) to subgroups by exponentiation to the cofactors of the subgroups:

\[
g \rightarrow g^{N/e_i}, \quad h \rightarrow h^{N/e_i}
\]

so we can solve independently the discrete log problem in distinct subgroups of order a prime power \( p_i^{e_i} \):

\[
g_i = g^{p_i^{e_i}}, \quad h_i = h^{p_i^{e_i}}, \quad \text{compute } \log_{g_i}(h_i) \text{ in } g_i^{p_i^{e_i}}
\]

we have

\[
\log g \cdot h \mod p_i^{e_i}
\]

if we have \( n \) processes, we can parallelize over \( n \) tasks the \( d \)-log computation

it means that the complexity in time is not function of \( N \) but function of \( \max e_i \).

Now, we are left with the computation of \( \log_{g_i}(h_i) \mod p_i^{e_i} \).

\[
h_i = g_i^a, \quad \text{where } a = a_0 + a_1 p_i + a_2 p_i^2 + \cdots + a_{e_i-1} p_i^{e_i-1} \text{ in base } p_i.
\]

ex: \( p_i = 2^2 \), \( a = 7 = 4 + 2 \times 3 \) for example.

How to compute the \( a_i \)?

Let's cancel the \( a_1, a_2, \ldots, a_{e_i-1} \) to compute \( a_0 \):

\[
\begin{align*}
\log_{g_i^{p_i^{e_i-1}}}(g_i^{p_i^{e_i-1}}) & = a \cdot p_i^{e_i-1} + a_1 p_i + a_2 p_i^2 + \cdots + a_{e_i-1} p_i^{e_i-1} \\
\log_{h_i^{p_i^{e_i}}}(h_i^{p_i^{e_i}}) & = g_i^{p_i^{e_i-1}}
\end{align*}
\]

but what is the order of \( g_i^{p_i^{e_i-1}} \)? \( g_i^{p_i^{e_i-1}} \) has order \( p_i \) \( \Rightarrow g_i^{p_i^{e_i-1}} = 1 \) hence all the \( a_i \) cancel.

\[
h_i^{p_i^{e_i-1}} = g_i^{a_0 p_i^{e_i-1}} = (g_i^{p_i^{e_i-1}})^{a_0} \quad \text{and} \quad g_i^{p_i^{e_i-1}} \text{ has order } p_i.
\]

we are left with the computation of \( a \cdot d \)-log \( \mod p_i \).
Once we know $a_0$, we have:

\[ h_i = q_i = a_0 + a_1 p_i + a_2 p_i^2 + \ldots + a_{e_i - 1} p_i^{e_i - 1} \]

unknowns

\[ h_i = g_i \]

\[ h_i = h_i / a_0 \]

raise $h_i$ to the power $p_i^{e_i - 2}$ to cancel all the $a_2, a_3, \ldots a_{e_i - 1}$ coefficients.

we have $h_i = (q_i p_i^{e_i - 1})^{a_n}$ → compute a $d$-log $a_n \mod p_i$.

the next step will set $h_i = h_i / a_0 + a_1 p_i$, and compute the $d$-log $a_1 \mod p_i$.

in total, the procedure computes sequentially $e_i$ times a $d$-log $\mod p_i$.

the complexity is $e_i (\text{time to compute a } d\text{-log } \mod p_i)$.

and it is not parallel this time.

In conclusion, Pollard - Hellman has complexity

\[ \max_{1 \leq i \leq n} (e_i \cdot (\text{complexity of } d\text{-log } \mod p_i)) \] where

\[ N = \prod_{i=1}^{n} p_i^{e_i} \]

We will see just after that the complexity of computing a $d$-log $\mod p_i$ is in $O(\sqrt{p_i})$ with generic algorithms.

See Gallant’s book Sect. 13.2 and Alg. 13 for more details.
Baby Step Giant Step to compute a discrete log in a cyclic subgroup.

\[
h = g^a, \text{ and we want to compute } a \mod p.\]

Let \( m = \lceil \sqrt{p} \rceil \) that is the ceiling of \( \sqrt{p} \).

Then \( a = a_0 + a_1 m \) where \( m = \lceil \sqrt{p} \rceil \) and \( 0 \leq a_0, a_1 < m \).

(Write the Euclidean division of \( a \) by \( m \) to get \( a_0 \) and \( a_1 \)).

\[
h = g^{a_0 + a_1 m} = g^{a_0} g^{a_1 m} = g^{a_0} (g^m)^{a_1}.
\]

Since \( h \cdot (g^{-m})^{a_1} = g^{a_0} \) baby step: enumerate all \( g^{a_0} \) for \( 0 \leq i \leq m \).

Giant step: enumerate all \( h \cdot (g^m)^j \) for \( j = 0, 1, \ldots \) until a match is found.

Requires companion and reach in the baby step database to be as fast as possible.


input: \( g, h \in G \) of order \( p \), output: \( a \) such that \( h = g^a \), or reject \( 1 \) (fail).

\( m = \lceil \sqrt{p} \rceil \) \( \lceil \sqrt{p} \rceil \) floor \( \lfloor \sqrt{p} \rfloor \) floor in latex \( \lfloor \sqrt{p} \rfloor \) floor \( \lfloor \sqrt{p} \rfloor \) floor in latex \( \lfloor \sqrt{p} \rfloor \) floor

easily searched structure \( L \) (hash table, binary tree).

\( f = 1 \)

for \( i = 0 \) to \( m \) do:

\( \text{store } (f, i) \in L \) (with \( f \) the key of the hash table for \( f \) in \( L \)).

end for

end for

\( L \) contains all the \( g^i \), \( 0 \leq i \leq m \). \( \text{storage } (\log p) \sqrt{p} \).

\( y = h, j = 0 \)

while \( y \) does not match a key in \( L \) do

\( y \leftarrow y \cdot m, \quad j \leftarrow j + 1 \)

end while

if \( \exists (f, i) \in L \) such that \( f = y \) then

return \( i + m \cdot j \)

else

return \( 1 \)

end if.

Theorem 13.3.4. Let \( G \) be a group of order \( p \). Suppose the elements of \( G \) are represented with \( O(\log p) \) bits and that the group operations can be performed in \( O(\log p)^2 \) bit operations. The BS & S algo for DLP in \( G \) has running time \( O(\sqrt{p} \log p)^2 \) bit operations. The algo requires \( O(\sqrt{p} \log p) \) bit of storage.
What is a hash table? (What is a binary search tree?)

- A hash table is a data structure that allows for efficient searching, insertion, and deletion of elements.
- It uses a hash function to map keys to indices in an array.

**Hash Function**: \( f : \mathbb{Z} \rightarrow \{0, 1, ..., m\}^* \)

**Hash Table Array**: \( T[0, 1, ..., m] \)

**Elements**: \( \{q_1, q_2, ..., q_n\} \)

**Hash Value**:
- **Hash Function**: \( h(q_i) \)
- **Conversion**: \( f(q_i) \rightarrow \) some encoding to \( \mathbb{N} \)
- **Hash Value**: \( H(q_i) = f(q_i) \mod (m+1) \)

**Initialization**: Create an array \( T \) of size \( N = m+1 \).

**Insertion**: For each new item \( q_i \), compute \( H(q_i) \). We would like to store it at index \( H(q_i) \).

**Collision Resolution**: What do we do if there is a collision?
- **Hash Table Array**: \( T[H(q_i)] \) is actually a linked list: we store buckets of values that are distinct but sharing the same hash function.
- **Null at Initialization**: \( T[n] \rightarrow \text{NULL} \) at initialization.

**Insertion**: \( T[n] \rightarrow \begin{cases} 1 \quad (q_i, i) & \rightarrow \begin{cases} 2 \quad (q_i, 2) \end{cases} \end{cases} \) etc.

**Searching**: Finding an element in \( T \) is done in constant time:
- **Hash Function**: \( h(q_i) \)
- **Search**: Test whether \( T[H(h(q_i))] \) is NULL or some linked list. Search in the linked list.

---

**Binary Search Tree**:
- Elements at the left are smaller than the root.
- Elements at the right are larger (greater) than the root.

---

**Lexicographical Ordering**:
- Consider lexicographical ordering for examples.
Birthday paradox
Floyd algorithm of cycle path finding in a graph
Pollard's

Theorem (Birthday paradox) 14.1.1 in Galbraith book.

Let S be a set of N elements. If elements are sampled uniformly at random from S then the expected number of samples to be taken before some element is sampled twice is less than $\sqrt{\pi N/2} + 2 \approx 1.253 \sqrt{N}$.

Repeats, match or collision: something sampled twice.

The number of elements that need to be selected from S to get a collision with probability 1/2 is $\sqrt{2 \log(2) N} \approx 1.177 \sqrt{N}$.

Application: $\sqrt{2 \log(2) 365} \approx 22.49$, $\sqrt{\pi 365/2} = 23.94$

Thus with 23 people, the probability to have a birthday collision is $\approx 1/2$.

Note that this is about a collision: the birthday date is not set. (We are not fixing any value in S).

Floyd algorithm (Hare & Tortoise)

Let S be a sequence (a linked list, the values taken by a function)

equipped with a "next" to get the next element of the sequence

The end point is None (None in Python).

def has_cycle(node S):
    if S is None:
        return False
    tortoise = S
    hare = S.next()
    while (tortoise != hare):
        if hare is None:
            return False
        hare = hare.next()
        if hare is None:
            return False
        hare = hare.next()
        tortoise = tortoise.next
    return True

The hare gets twice faster than the tortoise.

• If there is no cycle (no collision) then the hare will reach None (head)

• If there is a cycle:
  • the tortoise gets closer to the cycle at each step
  • the hare, once in the cycle, will never hit "None"
  • So at some point, both will be going inside the cycle, and then the hare will hit the tortoise at some point.
Pollard $\rho$ algorithm applies this idea to finding the discrete logarithm in a cyclic group $G$ of order $p$.

Find $a, b, a', b'$ such that
\[ q^{a_i} h^{b_i} = q^{a_j} h^{b_j} \quad \text{and} \quad b_i \equiv b_j \mod p. \]

\[ a_i - a_j = b_j - b_i \implies (a_i - a_j)/(b_j - b_i) \mod p = h. \]

\[ \log_q h = \frac{a_i - a_j}{b_j - b_i} \mod p. \]

**Alg 16 p. 230 (Galbraith book ch. 14).**

```
input: q, h \in G
output: a s.t. h = q^a, a \perp
1. choose a function walk
2. x_0 = q, a_0 = 1, b_1 = 0
3. (x_1, a_1, b_1) = walk(x_0, a_0, b_1)
4. while x_0 \neq x_2:
   5. (x_2, a_2, b_2) = walk(x_1, a_1, b_1)
   6. (x_3, a_3, b_3) = walk(walk(x_2, a_2, b_2))
7. end while
8. if b_3 \equiv b_2 \mod p then
9. return 1
10. else:
11. return (a_2 - a_3)(b_4 - b_2)^{-1} \mod p
12. wait.
```

What do we choose for the walk function?

Define $R_2 \subseteq S$ subset of $G$ of the same size $\#G/S$.

Precompute $g^i = g^{\mu_i} h^{-i}$ for $0 \leq j \leq n - 1$, for a randomly chosen uniform distribution $0 \leq \mu_i, \nu_j < p$.

- Original $\rho$ walk: $x_{i+1} = f(x_i) = \begin{cases} x_i \cdot g^{\mu_i} \nu_j & \text{if } S(x_i) = 0 \\ x_i \cdot g^{\mu_i} & \text{if } S(x_i) = j, j \in \{1, \ldots, n-1\} \end{cases}$
- Additive $\rho$ walk: $x_{i+1} = f(x_i) = x_i + S(x_i)$

Exact: smallest $i$ s.t. $x_i = x_0$. Expected value $0.823 \sqrt{\pi p/2} = (2/\sqrt{\pi}) \sqrt{p}$, $\zeta(n(2))$.

Complexity of best algorithm $\frac{3.093 + o(1)}{\sqrt{p}}$ group operation.

Best method: $\frac{(1.253 + o(1))}{\sqrt{p}}$ group operations.