Fast Exponentiation and Inversion

Introducing Computational Number Theory

Ananyapam De

Indian Institute of Science Education and Research, Kolkata

Exponentiation

Let a and n be integers with $n \geq 0$. Then the nth power of a, denoted by a^n, is defined as

$$a^n = a \cdot a \cdot \dots \cdot a$$

where the n factors of a are multiplied together.

```
def power(a, n):
    ans = 1
    for i in range(n):
        ans *= a
    return ans
```

Time Complexity: $O(n)$

Space Complexity: $O(1)$

Motivating Example

Suppose we want to compute 2^{1000}. We can do this by multiplying 2 by itself 1000 times. Very slow!

How can we compute 2^{1000} faster?

The Idea

Suppose we know 2^{500}. Then we can compute 2^{1000} by squaring 2^{500} (in one step). This is much faster!

How can we compute 2^{500}? We can compute 2^{250} by squaring 2^{125} (in one step). And so on.

$$a^n = \begin{cases} 1 & \text{if } n == 0 \\ \left(a^{\frac{n}{2}}\right)^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ \left(a^{\frac{n-1}{2}}\right)^2 \cdot a & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases}$$

Fast Exponentiation

```
def fastpower(a, n) {
    if (n == 0)
        return 1
    res = fastpower(a, n / 2)
    if (n % 2)
        ans = res * res * a
        return ans
    else
        ans = res * res
        return ans
}
```

Time Complexity: $O(\log n)$

Space Complexity: ??

Fast Exponentiation

```
def fastpower(a, n) {
    if (n == 0)
        return 1
    res = fastpower(a, n / 2)
    if (n % 2)
        ans = res * res * a
        return ans
    else
        ans = res * res
        return ans
}
```

Time Complexity: $O(\log n)$

Space Complexity: $O(\log n)$ (due to the recursive stack)

Iterative Implementation

Consider the binary representation of n.

Ex: $n = 1000 = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^3 = (1111101000)_2$

```python
def fastpower_iterative(a, n):
    ans = 1
    while n > 0:
        if n & 1:
            ans *= a
        a *= a
        n >>= 1
    return ans
```

Time Complexity: $O(\log n)$

Space Complexity: $O(1)$

Fibonacci Numbers!

Let F_n be the nth Fibonacci number. Then we have the following recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with $F_0 = 0$ and $F_1 = 1$.

How can we compute F_n?

Naive Recursive Implementation

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

Time Complexity: $O(2^n)$

Space Complexity: $O(n)$ (due to the recursive stack)

Memoization

We can use memoization to reduce the time complexity to $O(n)$.

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    if dp[n] != -1:
        return dp[n]
    dp[n] = fib(n-1) + fib(n-2)
    return dp[n]
```

Time Complexity: $O(n)$

Space Complexity: $O(n)$ (due to the memoization array)

Iterative Implementation

```python
def fib_iterative(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    a = 0
    b = 1
    for i in range(2, n+1):
        c = a + b
        a = b
        b = c
    return b
```

Time Complexity: $O(n)$

Space Complexity: $O(1)$

Can we do even better?

Matrix Exponentiation

Idea: Computing all required Fibonacci numbers in one step. (This is a very general technique.)

$$\begin{pmatrix} F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n-2} & F_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} F_n & F_{n+1} \end{pmatrix} = \begin{pmatrix} F_0 & F_1 \end{pmatrix} \cdot P^n$$

Time Complexity: $O(8 * \log n)$

Modular Exponentiation

$$a^n \bmod m = (a \cdot a \cdot \dots \cdot a) \bmod m$$

where the n factors of a are multiplied together.

Assume that a and m are coprime integers.

```
def modularpower(a, n, m):
    ans = 1
    while n > 0:
        if n & 1:
            ans = (ans * a) % m
        a = (a * a) % m
        n >>= 1
    return ans
```

Time Complexity: $O(\log n)$

Can we do even better?

Euler's Theorem

Let a and m be coprime integers. Then we have the following theorem:

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

where $\varphi(m)$ is the Euler totient function.

$\varphi(m)$ counts the number of integers between 1 and m inclusive that are coprime to m.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$\phi(n)$	1	1	2	2	4	2	6	4	6	4	10	4	12	6	8	8	16	6	18	8	12

How do we use this?

Problem: Compute $a^n \bmod m$.

Solution: Write $n = \varphi(m) \cdot k + r$, where k and r are integers and $0 \le r < \varphi(m)$.

Then we have the following:

$$a^n \bmod m = a^{\varphi(m) \cdot k + r} \bmod m \equiv a^r \bmod m$$

Time Complexity: $O(\log \varphi(m))$

Very very fast!

But how to compute $\varphi(m)$?

Properties of $\varphi(m)$

1. $\varphi(1) = 1$
2. $\varphi(p) = p - 1$ (where p is a prime)
3. $\varphi(p^k) = p^k - p^{k-1}$ (where p is a prime)
4. $\varphi(mn) = \varphi(m) \cdot \varphi(n)$ (where m and n are coprime)

Method 1: Naive Prime Factorization

Let $m = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_k^{a_k}$, where p_i are distinct primes and $a_i \geq 1$.

Then we have the following:

$$\varphi(m) = m \cdot \left(1 - \frac{1}{p_1}\right) \cdot \left(1 - \frac{1}{p_2}\right) \cdot \dots \cdot \left(1 - \frac{1}{p_k}\right)$$

Implementation

```
def eulerphi(m):
    ans = m
    for i in range(2, int(m**0.5) + 1):
        if m % i == 0:
            ans -= ans // i
            while m % i == 0:
                m //= i
    if m > 1:
        ans -= ans // m
    return ans
```

Time Complexity: $O(\sqrt{m})$

Method 2: Gauss's Divisor Sum Formula

Let m be a positive integer. Then we have the following formula:

$$\sum_{d|m} \varphi(d) = m$$

where d are the distinct prime factors of n

Example: $\varphi(12) = 4$. We have the following:

$$\sum_{d|12} \varphi(d) = \varphi(1) + \varphi(2) + \varphi(3) + \varphi(4) + \varphi(6) + \varphi(12) = 12$$

Implementation

```
def eulerphi(m):
    phi[0] = 0
    phi[1] = 1
    for i in range(2, m+1):
        phi[i] = i - 1

    for i in range(2, m+1):
        for j in range(2 * i, m+1, i):
            phi[j] -= phi[i]

    return phi[m]
```

Time Complexity: $O(m \log m)$

Modular Inverse

Let a and m be coprime integers. Then if a has a modular inverse modulo m, then there exists x such that:

$$a \cdot x \equiv 1 \pmod{m}$$

x is called the modular inverse of a modulo m, denoted by $a^{-1} \pmod{m}$.

How do we compute a^{-1}?

Method 1: Extended Euclidean Algorithm

If a and m be coprime integers, then we can find x and y such that:

$$ax + my = 1$$

using the extended Euclidean algorithm.

Take the modulo m of both sides:

$$ax \equiv 1 \pmod{m}$$

Thus the modular inverse of a modulo m is x.

Time Complexity: $O(\log min(a, m))$

Method 2: Fast Exponentiation (and Euler's Theorem)

$$a^{\varphi(m)-1} \equiv a^{-1} \pmod{m}$$

Time Complexity: $O(\log \varphi(m))$

Method 3: Euclidean Division

$$m = k \cdot i + r$$

where $k = \lfloor \frac{m}{i} \rfloor$ and $r = m \pmod{i}$

Then we have the following:

$$\begin{gathered} 0 \equiv k \cdot i + r \pmod{m} \\ r \equiv -k \cdot i \pmod{m} \\ r \cdot i^{-1} \equiv -k \pmod{m} \\ i^{-1} \equiv -k \cdot r^{-1} \pmod{m} \end{gathered}$$

Implementation

```
def modinv(a, m):
    if a ⩽ 1:
        return a
    else:
        return m - (m/a) * inv(m % a) % m
```

Time Complexity: ~ $O(\frac{logm}{loglogm})$

Space Complexity: $O(\log min(a, m))$

References

- https://cp-algorithms.com/algebra
- https://arxiv.org/abs/2211.08374 (On the length of Pierce expansions)
- https://artofproblemsolving.com/community/c90633h1291397
- https://www.cse.iitd.ac.in/~rjaiswal/2011/csl866/Notes/w-cnt.pdf (Chapter 9), Prof Ragesh Jaiswal (IIT Delhi)