Date:10/3/25

Week-10

Aim : Implement Digital Fingerprint Algorithm

Description:

Digital fingerprinting aims to create a unique and compact representation of digital content, allowing for efficient comparison and verification of data integrity and authenticity.

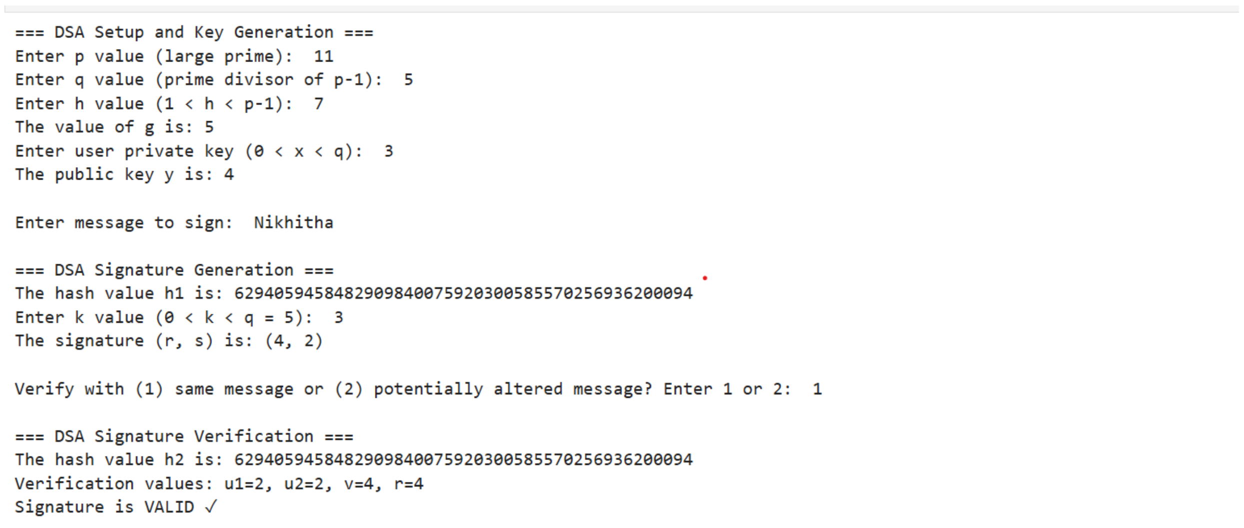
Code:

```
import hashlib

import sys

import random

from math import gcd


def hash_message(message):

    """Create a SHA-1 hash of the message and convert to integer"""

    result = hashlib.sha1(message.encode())

    hex_digest = result.hexdigest()

    return int(hex_digest, 16)


def mod_inverse(a, m):

    """Calculate the modular multiplicative inverse using the extended Euclidean algorithm"""

    if gcd(a, m) != 1:

        raise ValueError("Modular inverse does not exist")


    def extended_gcd(a, b):

        if a == 0:

            return b, 0, 1

        else:

            gcd, x1, y1 = extended_gcd(b % a, a)

            x = y1 - (b // a) * x1

            y = x1
        return gcd, x, y

    _, x, _ = extended_gcd(a, m)
    return (x % m + m) % m  # Ensure the result is positive

# DSA parameter generation and key generation
def dsa_setup():
    print("=== DSA Setup and Key Generation ===")

    # Get parameters p, q, and h
    p = int(input("Enter p value (large prime): "))
    q = int(input("Enter q value (prime divisor of p-1): "))

    # Validate that q is a divisor of p-1
    if (p - 1) % q != 0:
        print("Error: q must be a divisor of p-1")
        sys.exit(1)

    h = int(input("Enter h value (1 < h < p-1): "))

    # Validate h
    if h <= 1 or h >= p-1:
        print("Error: h must be in range (1, p-1)")
        sys.exit(1)

    # Calculate generator g
    g = pow(h, (p-1)//q, p)
    # Validate g
    if g <= 1:
        print("Error: g must be > 1. Try a different h value.")
        sys.exit(1)

    print(f"The value of g is: {g}")

    # Generate private and public keys
    x = int(input("Enter user private key (0 < x < q): "))

    # Validate private key
    if x <= 0 or x >= q:
        print("Error: Private key must be in range (0, q)")
        sys.exit(1)

    # Calculate public key
    y = pow(g, x, p)
    print(f"The public key y is: {y}")

    return p, q, g, x, y

# DSA signature generation
def dsa_sign(p, q, g, x, message):
    print("\n=== DSA Signature Generation ===")

    # Hash the message
    h1 = hash_message(message)
    print(f"The hash value h1 is: {h1}")
```

```
# Generate k (typically random, but user-input here for demonstration)
k = int(input(f"Enter k value (0 < k < q = {q}): "))

# Validate k
if k <= 0 or k >= q or gcd(k, q) != 1:
    print("Error: k must be in range (0, q) and coprime to q")
    sys.exit(1)

# Calculate r component of signature
r = pow(g, k, p) % q

if r == 0:
    print("Invalid signature: r = 0. Try a different k value.")
    sys.exit(1)

# Calculate modular inverse of k
k_inverse = mod_inverse(k, q)

# Calculate s component of signature
s = (k_inverse * (h1 + x * r)) % q

if s == 0:
    print("Invalid signature: s = 0. Try a different k value.")
    sys.exit(1)

print(f"The signature (r, s) is: ({r}, {s})")
```

```
    return r, s, h1

# DSA signature verification
def dsa_verify(p, q, g, y, message, r, s, original_hash=None):
    print("\n=== DSA Signature Verification ===")

    # Check if r and s are in the valid range
    if r <= 0 or r >= q or s <= 0 or s >= q:
        print("Invalid signature: r or s out of range")
        return False
    # Calculate the modular inverse of s
    s_inverse = mod_inverse(s, q)

    # Calculate u1 and u2
    u1 = (h2 * s_inverse) % q
    u2 = (r * s_inverse) % q

    # Calculate v = ((g^u1 * y^u2) mod p) mod q
    v = ((pow(g, u1, p) * pow(y, u2, p)) % p) % q

    print(f"Verification values: u1={u1}, u2={u2}, v={v}, r={r}")

def main():
    # Setup and key generation
    p, q, g, x, y = dsa_setup()

    # Signature generation
    message = input("\nEnter message to sign: ")
    r, s, h1 = dsa_sign(p, q, g, x, message)


    # Verification process
    verify_option = input("\nVerify with (1) same message or (2) potentially altered message? Enter 1 or 2: ")


    if verify_option == "1":
        # Verify with the same message
        dsa_verify(p, q, g, y, message, r, s, h1)
    else:
        # Verify with a potentially different message
        received_message = input("Enter message after transmission: ")
        dsa_verify(p, q, g, y, received_message, r, s)


if __name__ == "__main__":
    main()
```

Output:



```
=== DSA Setup and Key Generation ===
Enter p value (large prime):  11
Enter q value (prime divisor of p-1):  5
Enter h value (1 < h < p-1):  7
The value of g is: 5
Enter user private key (0 < x < q):  3
The public key y is: 4

Enter message to sign:  Nikhitha

=== DSA Signature Generation ===
The hash value h1 is: 6294059458482909840075920300585570256936200094
Enter k value (0 < k < q = 5):  3
The signature (r, s) is: (4, 2)

Verify with (1) same message or (2) potentially altered message? Enter 1 or 2:  1

=== DSA Signature Verification ===
The hash value h2 is: 6294059458482909840075920300585570256936200094
Verification values: u1=2, u2=2, v=4, r=4
Signature is VALID ✓
```