Lab Assignment No. 02 ZKP

Q1) ① completeness
② Soundness
③ Zero-knowledge

Q2) - Prover: knows the secret x and wants to prove knowledge of it without revealing it
- Verifier: Sends a random challenge and checks that

Q3) Because it allows the Prover to prove the truth of ~~a~~ a statement without revealing any information about the secret

Q4) The random choice of challenge from Large Set ensures that a cheating Prover a very slim chance of convincing the verifier

Q5) It replaces the verifier's random challenge with a hash, allowing the Prover to compute the response without waiting for the verifier

ZKP

Q6)) we can repeat multiple cheating attempts and count how many times the prover succeeds then calculate the empirical Probability of success

Q7)) cheating Prover could precompute the commitment and response to pass verification easily, compromising the security of Protocol

Q8)) ZKPs are used in cryptocurrencies like Zcash to prove transaction validity without revealing the sender or the amount

Q9)) Interactive: The verifier and prover exchange multiple messages

Non-Interactive: The challenge is derived from hash removing the need for message exchanges

```python
import random
import hashlib

# ---------- Parameters (small primes for lab/demo) ----------
# In practice p should be large; for lab we use small primes so outputs are easy to read.
p = 23        # prime
q = 11        # q divides p-1 (22) -> here 11 divides 22
h = 2         # candidate
g = pow(h, (p - 1) // q, p)  # generator of subgroup of order q

# Secret and Public key (Prover)
x = 7                 # secret (in Z_q)
y = pow(g, x, p)      # public key y = g^x mod p

print("Parameters:")
print(f"p = {p}, q = {q}, h = {h}, g = {g}")
print(f"Secret x = {x}  (keep secret)")
print(f"Public y = g^x mod p = {y}")
print("-" * 50)

# ------------------ Honest Interactive Schnorr ------------------
def honest_interactive_run():
    # Prover picks random r in Z_q, sends t = g^r
    r = random.randrange(q)
    t = pow(g, r, p)
    print("--- Honest Interactive Run ---")
    print(f"Prover commitment t = {t}  (r = {r})")
    # Verifier picks random challenge c in Z_q
    c = random.randrange(q)
    print(f"Verifier challenge e = {c}")
    # Prover computes response s = r + c*x (mod q)
    s = (r + c * x) % q
    print(f"Response s = {s}")
    # Verifier checks g^s ?= t * y^c  (mod p)
    left = pow(g, s, p)
    right = (t * pow(y, c, p)) % p
    ok = (left == right)
    print("Verification:", "Passed" if ok else "Failed")
    print("-" * 50)
    return {"t": t, "c": c, "r": r, "s": s, "ver_ok": ok}

honest_interactive_run()

# ------------------ Cheating Prover (interactive) ------------------
# Cheating strategy used here: the cheating prover **predicts** the challenge value e_guess
# before committing. If prediction is correct, they can craft t so that verification passes.
# Success probability ≈ 1/q for uniform challenge.
def cheating_attempt_once():
    # Cheater guesses a challenge e_guess uniformly from Z_q
    e_guess = random.randrange(q)
    # Cheater picks a random s in Z_q (s will be used if guess is correct)
    s_fake = random.randrange(q)
    # Compute t such that verification would pass for challenge = e_guess:
    # We need t = g^s * y^{-e_guess}  mod p  (because g^s ?= t * y^{e_guess})
    # To compute y^{-e_guess} use modular inverse.
    # y^e_guess mod p:
    y_pow = pow(y, e_guess, p)
    # inverse of y_pow modulo p (p is prime so use pow(y_pow, p-2, p))
    inv_y_pow = pow(y_pow, p - 2, p)
    t_fake = (pow(g, s_fake, p) * inv_y_pow) % p

    # Now verifier issues actual challenge c (unknown to cheater at commit time)
    c = random.randrange(q)
    # Verification step:
    left = pow(g, s_fake, p)
    right = (t_fake * pow(y, c, p)) % p
    ok = (left == right)
    # Success happens exactly when c == e_guess in this constructed strategy
    return {"e_guess": e_guess, "s_fake": s_fake, "t_fake": t_fake, "c": c, "ver_ok": ok}

# Demonstrate one cheating attempt and print
res_cheat = cheating_attempt_once()
print("--- Cheating Attempt (single) ---")
print(f"Prover fakes t = {res_cheat['t_fake']}")
print(f"Prover guessed challenge = {res_cheat['e_guess']}")
print(f"Verifier challenge e = {res_cheat['c']}")
print("Verification:", "Passed" if res_cheat['ver_ok'] else "Failed")
print("-" * 50)

# ------------------ Cheating Probability Experiment ------------------
def cheating_experiment(trials=1000):
    successes = 0
    for _ in range(trials):
        r = random.randrange(q)
        # cheater strategy as above
        e_guess = random.randrange(q)
        s_fake = random.randrange(q)
        y_pow = pow(y, e_guess, p)
        inv_y_pow = pow(y_pow, p - 2, p)
        t_fake = (pow(g, s_fake, p) * inv_y_pow) % p
        # real verifier challenge
        c = random.randrange(q)
        left = pow(g, s_fake, p)
        right = (t_fake * pow(y, c, p)) % p
        if left == right:
            successes += 1
    success_rate = successes / trials
    return {"trials": trials, "successes": successes, "success_rate": success_rate}

exp = cheating_experiment(trials=100)
print("--- Cheating Probability Experiment ---")
print(f"Cheating success rate = {exp['success_rate']:.4f} (after {exp['trials']} runs)")
print("Expected theoretical ≈ 1/q =", 1/q)
print("-" * 50)

# ------------------ Fiat-Shamir (Non-Interactive) ------------------
# Use SHA-256 as the hash, reduce mod q to get challenge e = H(t || msg) mod q
def fiat_shamir_noninteractive(message=b"Schnorr ZKP example"):
    # Prover picks r, computes t
    r = random.randrange(q)
    t = pow(g, r, p)
    # Compute hash-based challenge: H( t || message ) -> integer mod q
    # Represent t in bytes deterministically (e.g., decimal ASCII)
    hash_input = f"{t}|{message.decode() if isinstance(message, bytes) else str(message)}".encode()
    digest = hashlib.sha256(hash_input).digest()
    # Interpret digest as integer and reduce mod q
    e = int.from_bytes(digest, byteorder="big") % q
    # Prover computes s = r + e*x (mod q)
    s = (r + e * x) % q
    # Verifier recomputes e' from t||message and checks
    hash_input_v = f"{t}|{message.decode() if isinstance(message, bytes) else str(message)}".encode()
    e_v = int.from_bytes(hashlib.sha256(hash_input_v).digest(), byteorder="big") % q
    left = pow(g, s, p)
    right = (t * pow(y, e_v, p)) % p
    ok = (left == right)
    return {"t": t, "e": e, "s": s, "ver_ok": ok}

fs = fiat_shamir_noninteractive()
print("--- Fiat-Shamir (Non-Interactive) ---")
print(f"Hash-based challenge = {fs['e']}")
print("Verification:", "Passed" if fs['ver_ok'] else "Failed")
print("-" * 50)

# ------------------ Multiple Fiat-Shamir Trials (optional) ------------------
# Show a few runs to ensure deterministic verification
for i in range(3):
    out = fiat_shamir_noninteractive(message=b"Protocol run " + str(i).encode())
    print(f"FS run {i}: challenge={out['e']}  Verification={'Passed' if out['ver_ok'] else 'Failed'}")
```

```
Parameters:
p = 23, q = 11, h = 2, g = 4
Secret x = 7  (keep secret)
Public y = g^x mod p = 8
--------------------------------------------------
--- Honest Interactive Run ---
Prover commitment t = 6  (r = 10)
Verifier challenge e = 8
Response s = 0
Verification: Passed
--------------------------------------------------
--- Cheating Attempt (single) ---
Prover fakes t = 3
Prover guessed challenge = 5
Verifier challenge e = 5
Verification: Passed
--------------------------------------------------
--- Cheating Probability Experiment ---
Cheating success rate = 0.1100 (after 100 runs)
Expected theoretical ≈ 1/q = 0.09090909090909091
--------------------------------------------------
--- Fiat-Shamir (Non-Interactive) ---
Hash-based challenge = 9
Verification: Passed
--------------------------------------------------
FS run 0: challenge=8  Verification=Passed
FS run 1: challenge=7  Verification=Passed
FS run 2: challenge=9  Verification=Passed
```