Contents

1 Sum-Check Protocol [LFKN90]

1.1 Definition

Input: Verifier V given oracle access to a ℓ polynomial g over field F
Goal: Compute the quantity:

$$\sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \dots \sum_{x_\ell \in \{0,1\}} g(x_1, x_2, \dots, x_\ell)$$

1.2 Explanation:

The Sum-Check protocol is a fundamental technique in interactive proof systems, particularly in the context of arithmetic circuits and polynomial computations. It allows a verifier to efficiently check the result of a multi-dimensional summation over a polynomial f without actually needing to evaluate f at all the possible points in the domain.

- **Input:** The verifier V has *oracle access* to the polynomial g This means that V can query the value of g at any point within the domain without knowing the explicit representation of g

  "oracle access" is a theoretical computer science term that means:

  1. Black-box access:
  2. You can query input/output pairs
  3. You don't see how it's computed inside
  4. Like a black box with input/output slots
  5. Cost still exists:
  6. Each query counts as one operation
  7. If you query n times, complexity is O(n)
  8. Not "free" access

  Example in Python:

```python
class Polynomial_Oracle:
    def __init__(self):
        # Internal representation hidden from user
        self._coefficients = [1, 2, 3]  # ax² + bx + c

    def query(self, x):
        # User only sees input/output
        # Doesn't know it's a quadratic polynomial
        return sum(c * x**i for i, c in
        ↪  enumerate(self._coefficients))
```

```
# Verifier only gets this oracle access
oracle = Polynomial_Oracle()
result = oracle.query(2)  # Can only see input=2, output=17
```

- **Goal:** The goal is to compute the sum of f over all possible combinations of $x_1, x_2, ..., x_\ell$ where each x_i can be either 0 or 1.

1.3 Significance:

The Sum-Check protocol is crucial for several reasons:

- **Efficiency:** It allows the verifier to check the result of a complex summation with significantly less computational effort than evaluating the polynomial at all points.
- **Zero-Knowledge:** It can be adapted to provide zero-knowledge proofs, meaning the verifier can be convinced of the result without learning anything about the polynomial itself.

  Here's the high-level idea of making Sum-Check zero-knowledge:

  1. **Main Strategy**: Hide the real polynomial f by adding randomness

```
# Instead of revealing f directly, do:
def masked_polynomial(x):
    r = random_polynomial()  # mask
    return f(x) + r(x)       # masked value
```

  2. **Key Steps**:
  3. Prover sends masked values
  4. Uses randomization to hide real values
  5. Sum still verifies correctly due to:
     – Random parts cancel out
     – Original sum preserved
  3. **Verifier learns**:
  4. Final sum is correct
  5. Nothing about specific f values
  6. Only sees masked values

  Like showing your bank balance is >$1000 without revealing exact amount.

  This is just outline; actual protocol has more complex masking techniques.

1.4 How it works:

The protocol is interactive, involving a prover P who claims to know the value of the sum. The prover interacts with the verifier in a series of rounds, reducing the size of the summation by one dimension in each round, until the verifier can finally compute the final sum.

1.4.1 Sum-Check Protocol: Round 1

Start:

- **Prover (P)** sends a claimed answer C The protocol must check that:

$$C = \sum_{x_1 \in \{0,1\}} \sum_{x_2 \in \{0,1\}} \dots \sum_{x_\ell \in \{0,1\}} f(x_1, x_2, ..., x_\ell)$$

Round 1:

- **Prover (P)** sends a univariate polynomial $S_1(x_1)$ claimed to equal:

$$H_1(x_1) = \sum_{x_2 \in \{0,1\}} \sum_{x_3 \in \{0,1\}} \dots \sum_{x_\ell \in \{0,1\}} f(x_1, x_2, ..., x_\ell)$$

- **Verifier (V)** checks that $C = S_1(0) + S_1(1)$
- **Key Point:** If this check passes, it is safe for V to believe that C is the correct answer *as long as* V believes that $S_1 = H_1$

How to check S1 = H1?

- **Verifier's Approach:** The verifier V cannot directly compute $H_1(x_1)$ (because it would require knowing f which is unknown to V Instead, V chooses a random value r_1 in the field.
- **Verification Step:** V checks if $S_1(r_1)$ equals $H_1(r_1)$ To do this:
  - V can compute $S_1(r_1)$ from the polynomial $S_1(x_1)$ sent by the prover P.
  - V cannot compute $H_1(r_1)$ directly (it doesn't know f

```
class SumCheckProtocol:
    def __init__(self, ell):
        self.ell = ell  # number of variables
        self.field_size = 23  # example prime field

    def f(self, x):
        # Example polynomial f(x1,x2,x3) = x1x2 + x2x3 + x3x1
        # This would normally be unknown to verifier
        return (x[0]*x[1] + x[1]*x[2] + x[2]*x[0]) % self.field_size

    # Prover's side
    def compute_true_sum(self):
        # Compute actual sum over {0,1}^ell
        result = 0
        for bits in product([0,1], repeat=self.ell):
            result = (result + self.f(bits)) % self.field_size
        return result

    def S1(self, x1):
        # First round polynomial S1(x1)
        result = 0
        for bits in product([0,1], repeat=self.ell-1):
            point = (x1,) + bits
            result = (result + self.f(point)) % self.field_size
        return result

    # Verifier's side
    def verify_round1(self, claimed_C, S1):
        # Check C = S1(0) + S1(1)
        if claimed_C != (S1(0) + S1(1)) % self.field_size:
            return False

        # Pick random r1
        r1 = random.randrange(self.field_size)
        return r1, S1(r1)

# Example usage
protocol = SumCheckProtocol(ell=3)

# Prover computes and sends C
C = protocol.compute_true_sum()
print(f"Claimed sum C: {C}")

# Prover sends S1
S1 = protocol.S1
print(f"S1(0): {S1(0)}, S1(1): {S1(1)}")

# Verifier checks
r1, S1_r1 = protocol.verify_round1(C, S1)
```

```
print(f"Verifier picked r1={r1}, S1(r1)={S1_r1}")
```

Using sympy to understand the algebraic structure more clearly:

```
from sympy import symbols, expand, Poly
from itertools import product

class SymbolicSumCheck:
    def __init__(self, ell=3):
        self.ell = ell
        # Create symbolic variables x1, x2, x3
        self.vars = symbols(f'x1:{ell+1}')

    def f(self, x):
        # f(x1,x2,x3) = x1*x2 + x2*x3 + x3*x1
        x1, x2, x3 = self.vars
        return x1*x2 + x2*x3 + x3*x1

    def compute_true_sum(self):
        result = 0
        # Sum over {0,1}^ell
        for bits in product([0, 1], repeat=self.ell):
            subs_dict = {self.vars[i]: bit for i, bit in enumerate(bits)}
            result += self.f(bits).subs(subs_dict)
        return result

    def S1(self, x1_val):
        x1 = self.vars[0]
        result = 0
        # Sum over remaining variables
        for bits in product([0, 1], repeat=self.ell-1):
            subs_dict = {self.vars[i+1]: bit for i, bit in
             ↪  enumerate(bits)}
            # Keep x1 symbolic, substitute others
            result += self.f(self.vars).subs(subs_dict)
        # Convert to polynomial in x1
        return Poly(expand(result), x1)

# Example usage
protocol = SymbolicSumCheck()
C = protocol.compute_true_sum()
print(f"C = {C}")
```

```
S1 = protocol.S1(symbols('x1'))
print(f"S1 = {S1}")
print(f"S1(0) = {S1.eval(0)}")
print(f"S1(1) = {S1.eval(1)}")
```

This illustrates: 1. Prover computes true sum C 2. Prover sends polynomial S1 3. Verifier checks C = S1(0) + S1(1) 4. Verifier picks random r1 5. Process continues (not shown: subsequent rounds)

Zero-Knowledge Aspect:

By checking only at a random point, r_1 the verifier does not gain knowledge of f They are only checking the consistency of the claimed intermediate sum $H_1(x_1)$ with the actual value sent by the prover in the form of $S_1(x_1)$

Why This Works:

If the prover is honest, S_1 will indeed equal H_1 and this random check is likely to pass. If the prover is dishonest, it is unlikely that S_1 will match H_1 at a random point, and the verifier will catch the deception.

1.4.2 Sum-Check Protocol: Round 2

Next Rounds: The protocol continues for $\ell - 1$ more rounds, reducing the sum one dimension at a time, using similar random checks to maintain zero-knowledge.

Round 2: They recursively check that $S_1(r_1) = H_1(r_1)$

i.e. $s_1(r_1) = \sum_{b_2 \in \{0,1\}} \cdots \sum_{b_\ell \in \{0,1\}} f(r_1, b_2, \dots, b_\ell)$.

```
from sympy import symbols, expand, Poly
from itertools import product

class SymbolicSumCheckRounds:
    def __init__(self, ell=3):
        self.ell = ell
        self.vars = symbols(f'x1:{ell+1}')
        print(f"Initialized with {ell} variables: {self.vars}")

    def f(self, x):
        x1, x2, x3 = self.vars
        return x1*x2 + x2*x3 + x3*x1

    def compute_true_sum(self):
        result = 0
        print(f"\nComputing true sum over {2**self.ell} points:")
        for bits in product([0, 1], repeat=self.ell):
            subs_dict = {self.vars[i]: bit for i, bit in enumerate(bits)}
            term = self.f(bits).subs(subs_dict)
```

```python
            result += term
            print(f"  f{bits} = {term}")
        print(f"True sum: {result}")
        return result

    def S1(self, x1):
        print(f"\nComputing S1(x1):")
        result = 0
        for bits in product([0, 1], repeat=self.ell-1):
            subs_dict = {self.vars[i+1]: bit for i, bit in
             ↪  enumerate(bits)}
            term = self.f(self.vars).subs(subs_dict)
            result += term
            print(f"  Term for {bits}: {term}")
        poly = Poly(expand(result), x1)
        print(f"S1(x1) = {poly}")
        return poly

    def S2(self, r1, x2):
        print(f"\nComputing S2(x2) with r1 = {r1}:")
        result = 0
        for bits in product([0, 1], repeat=self.ell-2):
            subs_dict = {self.vars[0]: r1, self.vars[1]: x2}
            for i, bit in enumerate(bits):
                subs_dict[self.vars[i+2]] = bit
            term = self.f(self.vars).subs(subs_dict)
            result += term
            print(f"  Term for {bits}: {term}")
        poly = Poly(expand(result), x2)
        print(f"S2(x2) = {poly}")
        return poly

# Example usage
protocol = SymbolicSumCheckRounds()
C = protocol.compute_true_sum()
print(f"C = {C}")

# Round 1
S1 = protocol.S1(symbols('x1'))
print(f"S1(0) = {S1.eval(0)}")
print(f"S1(1) = {S1.eval(1)}")

# Verifier picks r1 = 2
r1 = 2
```

```
S1_at_r1 = S1.eval(r1)
print(f"S1({r1}) = {S1_at_r1}")

# Round 2
S2 = protocol.S2(r1, symbols('x2'))
print(f"S2(0) = {S2.eval(0)}")
print(f"S2(1) = {S2.eval(1)}")
```

---

if $S_1 \neq H_1$ the probability V accepts is at most:
$\Pr_{T_1 \in F}[s_1(r_1) = H(r_1)] + \Pr_{r_2,\ldots,r_\ell \in F}[\text{V accepts}|s_1(r_1) \neq H(r_1)]$
$\leq \frac{d}{|F|} + \frac{d(\ell-1)}{|F|} \leq \frac{d}{|F|}$

```
from sympy import symbols, expand, Poly
import random
from fractions import Fraction

class SumCheckProbability:
    def __init__(self, d, field_size, ell):
        """
        d: degree of polynomial
        field_size: size of field F
        ell: number of variables
        """
        self.d = d
        self.F = field_size
        self.ell = ell

    def simulate_verification(self, num_trials=1000):
        """Simulate the verification process and compute probabilities"""
        print(f"\nSimulating with parameters:")
        print(f"d = {self.d}, |F| = {self.F}, $\\ell$ = {self.ell}")

        # Probability of $s_1(r_1) = H(r_1)$
        prob_s1_equals_h = 1 - Fraction(self.d, self.F)
        print(f"\nPr[$s_1(r_1) = H(r_1)$] $\\leq$ {prob_s1_equals_h}")

        # Probability of accepting when $s_1(r_1) \neq H(r_1)$
        prob_accept_given_different = Fraction(self.d * (self.ell - 1),
        ↪  self.F)
        print(f"Pr[V accepts | $s_1(r_1) \\neq H(r_1)$] $\\leq$
        ↪  {prob_accept_given_different}")

        # Total probability bound
        total_bound = Fraction(self.d, self.F) + Fraction(self.d *
        ↪  (self.ell - 1), self.F)
        print(f"\nTotal probability bound: {total_bound}")

        # Verify bound is $\leq d\ell/|F|$
        theoretical_bound = Fraction(self.d * self.ell, self.F)
        print(f"Theoretical bound ($d\\ell/|F|$): {theoretical_bound}")
        print(f"Bound satisfied: {total_bound <= theoretical_bound}")

# Example usage
params = [
    (2, 17, 3),    # small field
    (3, 101, 4),   # medium field
    (5, 257, 5)    # larger field
]

for d, F, ell in params:
    simulator = SumCheckProbability(d, F, ell)
    simulator.simulate_verification()
    print("\n" + "="*50)

def schwartz_zippel_bound(d, F):
    """Compute Schwartz-Zippel lemma bound"""
    return Fraction(d, F)

# Additional verification
print("\nVerifying Schwartz-Zippel bounds:")
for d, F, _ in params:
    bound = schwartz_zippel_bound(d, F)
    print(f"d={d}, |F|={F}: Pr[error] $\\leq$ {bound}")
```

1.4.3 Round ℓ (Final Round)

- **Prover (P)** sends a univariate polynomial $S_\ell(x_\ell)$ claimed to equal $H_\ell(x_\ell) := f(r_1, r_2, ..., r_{\ell-1}, x_\ell)$
- **Verifier (V)** checks that $S_\ell(r_\ell) = S_\ell(0) + S_\ell(1)$
- **V picks r_ℓ at random** and needs to check that $S_\ell(r_\ell) = f(r_1, ..., r_{\ell-1}, r_\ell)$
- **No need for more rounds.** V can perform this check with one oracle query.

Explanation:
In the final round of the Sum-Check protocol:

1. **The Prover's Role:** The prover has reduced the multidimensional sum to a single variable x_ℓ by successively fixing the previous variables $x_1, x_2, ..., x_{\ell-1}$ to random values $r_1, r_2, ..., r_{\ell-1}$ They send a polynomial $S_\ell(x_\ell)$ that is claimed to represent the sum over the remaining variable.

2. **The Verifier's Role:** The verifier checks if the prover's final polynomial S_ℓ is consistent with the previous rounds. The first check ensures that the polynomial is well-formed. Then, the verifier picks a random value r_ℓ for the final variable and checks if the polynomial $S_\ell(r_\ell)$ is equal to the value of the original function f at the point $(r_1, r_2, ..., r_\ell)$

3. **One Oracle Query:** Crucially, the verifier can perform this final check with just one query to the oracle. They no longer need to perform any other calculations because the final polynomial S_ℓ represents the entire sum.

4. **Zero-Knowledge:** This entire process ensures that the verifier only gains knowledge about the final sum but nothing about the function f itself.

```python
from sympy import symbols, expand, Poly
from itertools import product
import random

class FullSumCheckProtocol:
    def __init__(self, ell=3):
        self.ell = ell
        self.vars = symbols(f'x1:{ell+1}')
        print(f"Initialized with variables: {self.vars}")
        self.random_points = []   # Store ri values

    def f(self, x):
        x1, x2, x3 = self.vars
        return x1*x2 + x2*x3 + x3*x1

    def compute_Si(self, i, prev_random_points):
        """
        Compute Si for round i, given previous random points
        i: current round (0 to ell)
        prev_random_points: list of r1,...,r_{i-1}
        """
        print(f"\nComputing S{i}({'C' if i==0 else f'x_{i}'}):")
        result = 0

        # Variables to sum over
        variables_to_sum = self.ell if i == 0 else self.ell - i
        # Create substitution dictionary for fixed points
        fixed_subs = {
            self.vars[j]: r
            for j, r in enumerate(prev_random_points)
        }
        print(f"Fixed points: {fixed_subs}")

        # Sum over remaining variables
        for bits in product([0, 1], repeat=variables_to_sum):
            subs_dict = fixed_subs.copy()
            # Add remaining variable assignments
            for j, bit in enumerate(bits):
                subs_dict[self.vars[j if i == 0 else i + j]] = bit

            term = self.f(self.vars).subs(subs_dict)
            result += term
            print(f"  Term for {bits}: {term}")

        if i == 0:
            print(f"C = {result}")
            return result
        else:
            current_var = self.vars[i-1]  # i starts from 1 for actual
            ↪  rounds
            poly = Poly(expand(result), current_var)
            print(f"S{i}({current_var}) = {poly}")
            return poly

    def verify_round(self, i, Si, ri):
        """Verify round i with random point ri"""
        print(f"\nVerifying round {i}:")
        print(f"Checking S{i}(0) + S{i}(1) = previous value")
        sum_01 = Si.eval(0) + Si.eval(1)
        print(f"S{i}(0) + S{i}(1) = {sum_01}")

        # Evaluate at random point
        Si_at_ri = Si.eval(ri)
        print(f"S{i}({ri}) = {Si_at_ri}")
        return Si_at_ri

# Example usage
protocol = FullSumCheckProtocol()
```

```python
# Initial sum
C = protocol.compute_Si(0, [])
print(f"\nInitial sum C = {C}")

# All rounds
random_points = []
for i in range(1, protocol.ell + 1):
    # Compute Si
    Si = protocol.compute_Si(i, random_points)

    # Verifier picks random ri
    ri = i + 1  # For demonstration, using i+1 as random point
    random_points.append(ri)

    # Verify this round
    result = protocol.verify_round(i, Si, ri)

    print(f"\nRound {i} complete. Random point r{i} = {ri}")
    if i == protocol.ell:
        print(f"Final round: Verifier makes oracle query
        ↪  f({random_points}) = {result}")

class CheatingProver(FullSumCheckProtocol):
    def __init__(self, ell=3, cheat_probability=0.5):
        super().__init__(ell)
        self.cheat_probability = cheat_probability
        print(f"Initialized Cheating Prover with cheat probability:
        ↪  {self.cheat_probability}")

    def compute_Si(self, i, prev_random_points):
        """Compute Si, potentially cheating"""
        honest_Si = super().compute_Si(i, prev_random_points)

        if i == 0 or random.random() > self.cheat_probability:
            print("Prover is honest this round.")
            return honest_Si

        print("Prover is cheating this round!")
        if i == self.ell:
            # In the final round, just return a different constant
            return honest_Si + 1

        # For earlier rounds, perturb the honest polynomial
        x = self.vars[i-1]
        cheat_poly = honest_Si + x  # Add a linear term to cheat
        print(f"Cheating S{i}({x}) = {cheat_poly}")
        return cheat_poly

class Verifier:
    def __init__(self, prover):
        self.prover = prover
        self.previous_value = None

    def run_protocol(self):
        C = self.prover.compute_Si(0, [])
        print(f"\nClaimed initial sum C = {C}")
        self.previous_value = C

        random_points = []
        for i in range(1, self.prover.ell + 1):
            Si = self.prover.compute_Si(i, random_points)
            ri = random.randint(2, 10)  # Random point, avoiding 0 and 1
            random_points.append(ri)

            if not self.verify_round(i, Si, ri):
                print(f"Verification failed at round {i}!")
                return False

        print("All rounds verified successfully.")
        return True

    def verify_round(self, i, Si, ri):
        print(f"\nVerifying round {i}:")
        sum_01 = Si.eval(0) + Si.eval(1)
        print(f"S{i}(0) + S{i}(1) = {sum_01}")
        print(f"Previous value = {self.previous_value}")

        if sum_01 != self.previous_value:
            return False

        Si_at_ri = Si.eval(ri)
        print(f"S{i}({ri}) = {Si_at_ri}")
        self.previous_value = Si_at_ri
        return True

prover = CheatingProver(ell=3, cheat_probability=0.7)
verifier = Verifier(prover)
protocol_result = verifier.run_protocol()
```

```
print(f"\nProtocol {'succeeded' if protocol_result else 'failed'}")

prover = CheatingProver(ell=3, cheat_probability=0.7)
verifier = Verifier(prover)
protocol_result = verifier.run_protocol()
print(f"\nProtocol {'succeeded' if protocol_result else 'failed'}")
```

Result:

```
Initialized with variables: (x1, x2, x3)

Computing S0(C):
Fixed points: {}
  Term for (0, 0, 0): 0
  Term for (0, 0, 1): 0
  Term for (0, 1, 0): 0
  Term for (0, 1, 1): 1
  Term for (1, 0, 0): 0
  Term for (1, 0, 1): 1
  Term for (1, 1, 0): 1
  Term for (1, 1, 1): 3
C = 6

Initial sum C = 6

Computing S1(x_1):
Fixed points: {}
  Term for (0, 0): 0
  Term for (0, 1): x1
  Term for (1, 0): x1
  Term for (1, 1): 2*x1 + 1
S1(x1) = Poly(4*x1 + 1, x1, domain='ZZ')

Verifying round 1:
Checking S1(0) + S1(1) = previous value
S1(0) + S1(1) = 6
S1(2) = 9

Round 1 complete. Random point r1 = 2

Computing S2(x_2):
Fixed points: {x1: 2}
  Term for (0,): 2*x2
  Term for (1,): 3*x2 + 2
S2(x2) = Poly(5*x2 + 2, x2, domain='ZZ')

Verifying round 2:
Checking S2(0) + S2(1) = previous value
S2(0) + S2(1) = 9
S2(3) = 17

Round 2 complete. Random point r2 = 3

Computing S3(x_3):
Fixed points: {x1: 2, x2: 3}
  Term for (): 5*x3 + 6
S3(x3) = Poly(5*x3 + 6, x3, domain='ZZ')

Verifying round 3:
Checking S3(0) + S3(1) = previous value
S3(0) + S3(1) = 17
S3(4) = 26

Round 3 complete. Random point r3 = 4
Final round: Verifier makes oracle query f([2, 3, 4]) = 26
Initialized with variables: (x1, x2, x3)
Initialized Cheating Prover with cheat probability: 0.7

Computing S0(C):
Fixed points: {}
  Term for (0, 0, 0): 0
  Term for (0, 0, 1): 0
  Term for (0, 1, 0): 0
  Term for (0, 1, 1): 1
  Term for (1, 0, 0): 0
  Term for (1, 0, 1): 1
  Term for (1, 1, 0): 1
  Term for (1, 1, 1): 3
C = 6
Prover is honest this round.

Claimed initial sum C = 6

Computing S1(x_1):
Fixed points: {}
  Term for (0, 0): 0
  Term for (0, 1): x1
  Term for (1, 0): x1
  Term for (1, 1): 2*x1 + 1
S1(x1) = Poly(4*x1 + 1, x1, domain='ZZ')
Prover is cheating this round!
Cheating S1(x1) = Poly(5*x1 + 1, x1, domain='ZZ')

Verifying round 1:
S1(0) + S1(1) = 7
Previous value = 6
Verification failed at round 1!

Protocol failed
Initialized with variables: (x1, x2, x3)
Initialized Cheating Prover with cheat probability: 0.7

Computing S0(C):
Fixed points: {}
  Term for (0, 0, 0): 0
  Term for (0, 0, 1): 0
  Term for (0, 1, 0): 0
  Term for (0, 1, 1): 1
  Term for (1, 0, 0): 0
  Term for (1, 0, 1): 1
  Term for (1, 1, 0): 1
  Term for (1, 1, 1): 3
C = 6
Prover is honest this round.

Claimed initial sum C = 6

Computing S1(x_1):
Fixed points: {}
  Term for (0, 0): 0
  Term for (0, 1): x1
  Term for (1, 0): x1
  Term for (1, 1): 2*x1 + 1
S1(x1) = Poly(4*x1 + 1, x1, domain='ZZ')
Prover is cheating this round!
Cheating S1(x1) = Poly(5*x1 + 1, x1, domain='ZZ')

Verifying round 1:
S1(0) + S1(1) = 7
Previous value = 6
Verification failed at round 1!
```

```
Protocol failed
```

1.5 IP=PSPACE

- **#SAT is a #P-complete problem:** This means that any problem in the complexity class #P (counting problems) can be reduced to #SAT in polynomial time.

  #P-complete and NP-complete are different:

  NP-complete: - Decision problems ("Is there a solution?") - Example: SAT asks "Is this boolean formula satisfiable?" - Answer is YES/NO

  #P-complete: - Counting problems ("How many solutions?") - Example: #SAT asks "How many satisfying assignments are there?" - Answer is a number

  Example to illustrate:

```
def SAT(formula):  # NP-complete
    # Returns True if formula has ANY satisfying assignment
    return any(is_satisfying(assignment) for assignment in
    ↪  all_possible())

def sharp_SAT(formula):  # #P-complete (NOT hash_SAT)
    # Returns COUNT of ALL satisfying assignments
    return sum(1 for assignment in all_possible() if
    ↪  is_satisfying(assignment))

# For formula: (x1 OR x2)
formula = "(x1 OR x2)"
print(f"SAT: {SAT(formula)}")          # True: solution exists
print(f"#SAT: {sharp_SAT(formula)}")   # 3: counts (1,0), (0,1),
↪  (1,1)

# Another example:
def satisfying_assignments(formula):
    return [(1,0), (0,1), (1,1)]  # for (x1 OR x2)

print(f"SAT just needs one: {satisfying_assignments(formula)[0]}")
print(f"#SAT counts all: {len(satisfying_assignments(formula))}")
```

  #P-complete problems are generally harder than NP-complete problems because they require counting ALL solutions rather than just finding ONE.

- **The protocol we just saw implies every problem in #P has an interactive proof with a polynomial-time verifier:** The protocol refers to the Sum-Check protocol. Because #SAT is #P-complete, and the Sum-Check protocol can handle #SAT with a polynomial-time verifier, this implies that any problem in #P can be solved with an interactive proof (using the same protocol and a reduction from the problem to #SAT).

#SAT is #P-complete for two key reasons:

1. **#SAT is in #P:**

```
def verify_SAT_solution(formula, assignment):
    # Polynomial-time verification
    return formula.evaluate(assignment)

def count_solutions(formula):  # in #P
    # Non-deterministic counting
    return sum(1 for assignment in all_possible_assignments
               if verify_SAT_solution(formula, assignment))
```

2. **Every #P problem reduces to #SAT:**
   - Any counting problem in #P can be transformed to counting SAT solutions
   - Example: counting valid graph colorings → #SAT

```
def reduce_graph_coloring_to_SAT(graph):
    # Transform graph coloring instance to SAT formula
    # Such that:
    # number of valid colorings = number of satisfying
    ↪  assignments
    variables = create_variables_for_vertices()
    clauses = create_coloring_constraints()
    return SAT_formula(variables, clauses)
```

This combination of being in #P (membership) and having all #P problems reducible to it (hardness) makes #SAT #P-complete.

Note:

Here's a more detailed implementation showing how SAT formulas work:

```
class SAT_Formula:
    def __init__(self):
        self.variables = []
        self.clauses = []

```

```python
    def add_variable(self, name):
        self.variables.append(name)
        return len(self.variables) - 1  # return variable index

    def add_clause(self, literals):
        # literal: (var_index, is_positive)
        self.clauses.append(literals)

def reduce_graph_coloring_to_SAT(graph, colors=3):
    """
    Transform k-coloring problem to SAT
    graph: {vertex: [neighbors]}
    colors: number of colors (typically 3)
    """
    formula = SAT_Formula()

    # 1. Create variables for each vertex-color pair
    # x_{v,c} means "vertex v has color c"
    color_vars = {}
    for v in graph:
        color_vars[v] = []
        for c in range(colors):
            var_idx = formula.add_variable(f"x_{v}_{c}")
            color_vars[v].append(var_idx)

    # 2. Each vertex must have at least one color
    for v in graph:
        formula.add_clause([(var, True) for var in color_vars[v]])

    # 3. No vertex can have two colors
    for v in graph:
        for c1 in range(colors):
            for c2 in range(c1+1, colors):
                formula.add_clause([
                    (color_vars[v][c1], False),
                    (color_vars[v][c2], False)
                ])

    # 4. Adjacent vertices must have different colors
    for v1 in graph:
        for v2 in graph[v1]:  # neighbors
            for c in range(colors):
                formula.add_clause([
                    (color_vars[v1][c], False),
                    (color_vars[v2][c], False)
                ])

    return formula

# Example usage
graph = {
    0: [1, 2],
    1: [0, 2],
    2: [0, 1]
}

formula = reduce_graph_coloring_to_SAT(graph)
print(f"Variables: {formula.variables}")
print(f"Clauses: {formula.clauses}")

def count_satisfying_assignments(formula):
    """Count solutions (this is #SAT)"""
    count = 0
    for assignment in product([0,1], repeat=len(formula.variables)):
        if is_satisfying(formula, assignment):
            count += 1
    return count
```

This shows: 1. How to encode graph coloring as SAT 2. The structure of SAT formulas (variables and clauses) 3. Why counting solutions (#SAT) is harder than finding one (SAT) 4. Why any #P problem can reduce to #SAT (through similar encodings)

The #P-completeness comes from: - Any counting problem can be encoded this way - The counting preserves the number of solutions - The reduction is polynomial-time

To further show NP-complete vs #P-complete problems:

A problem A is NP-complete if the following conditions hold:

1. $A \in NP$: Problem A is in the class NP (Nondeterministic Polynomial time). This means there exists a nondeterministic Turing machine that can verify a solution to A in polynomial time.
2. $SAT \leq^p A$: SAT (Boolean Satisfiability) can be reduced to A in polynomial time. This means there is a polynomial-time algorithm that transforms any instance of SAT into an instance of A, such that the solution to the SAT instance can be obtained from the solution to the A instance.
3. $A \leq^p SAT$: A can be reduced to SAT in polynomial time. This means there

is a polynomial-time algorithm that transforms any instance of A into an instance of SAT.

In simpler terms:

- **NP:** Problems where you can quickly check if a given solution is correct (but finding the solution might be hard).
- **SAT:** The classic NP-complete problem, determining if a Boolean formula has a satisfying assignment.
- **Reduction ($\leq^p$):** Transforming one problem into another in a way that preserves the solution (in polynomial time).

NP-Completeness Implications:

If a problem is NP-complete, it means:

- **Hardness:** The problem is at least as hard as SAT.
- **Universality:** Any other problem in NP can be reduced to this problem.

Therefore, finding efficient algorithms (polynomial-time) for NP-complete problems would imply that all problems in NP can be solved efficiently. However, this remains an open question in computer science.

```
def is_SAT(formula):
    return any(is_satisfying(assignment) for assignment)

def solve_NP_problem(problem):
    sat_formula = convert_to_SAT(problem)   # polynomial-time
     ↪   reduction
    return is_SAT(sat_formula)
```

#P-complete: Similarly, problem A is #P-complete if:

1. A $\in$ #P
2. #SAT $\leq^p$ A (#SAT reduces to A)
3. A $\leq^p$ #SAT (A reduces to #SAT)

```
def count_SAT(formula):
    return sum(1 for assignment if is_satisfying(assignment))

def solve_sharp_P_problem(problem):
    sat_formula = convert_to_SAT(problem)   # polynomial-time
     ↪   reduction
    return count_SAT(sat_formula)   # counts ALL solutions
```

Key difference: NP is about finding ONE solution, #P is about counting ALL solutions.

- **It is not much harder to show that this in fact holds for every problem in PSPACE [LFKN, Shamir]:** The result, known as IP=PSPACE, states that the class of problems solvable by interactive proofs (IP) is equivalent to the class of problems solvable in polynomial space (PSPACE).

In simpler terms:

- #P problems are those that ask "How many solutions are there?"
- Interactive proofs are like "conversations" between a prover and a verifier to determine a truth.
- PSPACE refers to problems that can be solved using a polynomial amount of memory (space), regardless of time:

  Definition: A problem is in PSPACE if it can be solved by a Turing machine using $O(n^k)$ space, where: - n is input size - k is some constant - Space means memory used

  Example in Python:

```python
def PSPACE_example(n):
    # Space complexity: O(n)
    # Time complexity: Could be exponential!
    def recursive_solve(depth, space):
        if depth == n:
            return check_solution(space)

        # Only uses polynomial space
        # But might take exponential time to try all possibilities
        for choice in possible_choices:
            space[depth] = choice
            if recursive_solve(depth + 1, space):
                return True
        return False

    # Only uses n cells of memory
    space = [0] * n
    return recursive_solve(0, space)
```

  Key properties:

  – PSPACE includes NP and coNP

  - PSPACE = NPSPACE (Savitch's Theorem)
  - Problems in PSPACE can be very hard to solve (time-wise) but need only polynomial memory

1.6 Double-Efficient Interactive Proof for Counting Triangles

- **Input:** $A \in \{0,1\}^{n\times n}$ representing the adjacency matrix of a graph.
- **Desired Output:**

$$\frac{1}{6} \cdot \sum_{(i,j,k)\in\{n\}^3} A_{ij}A_{jk}A_{ik}$$

```python
import numpy as np

def count_triangles(A):
    """
    A: adjacency matrix where
    A[i][j] = 1 if edge exists between vertices i and j
    A[i][j] = 0 if no edge exists
    """
    n = len(A)
    count = 0


    # This implements $\sum_{(i,j,k)\in[n]^3} (A_{ij} * A_{jk} *
    ↪  A_{ik})$
    for i in range(n):
        for j in range(n):
            for k in range(n):
                # A_ij: edge from i to j exists?
                # A_jk: edge from j to k exists?
                # A_ik: edge from i to k exists?
                triangle = A[i][j] * A[j][k] * A[i][k]
                count += triangle

    # Divide by 6 because each triangle is counted 6 times
    return count // 6

# Example
graph = np.array([
    [0, 1, 1],  # vertex 0 connected to 1 and 2
    [1, 0, 1],  # vertex 1 connected to 0 and 2
    [1, 1, 0]   # vertex 2 connected to 0 and 1
])
```

```
print(f"Number of triangles: {count_triangles(graph)}")  # Should
↪  print 1
```

Explanation: - `A_ij` is `A[i][j]`: indicates if edge (i,j) exists - `A_jk` is `A[j][k]`: indicates if edge (j,k) exists - `A_ik` is `A[i][k]`: indicates if edge (i,k) exists - When all three are 1, we've found a triangle - Divide by 6 because each triangle is counted in all possible permutations ($3! = 6$)

- **Fastest known algorithm:** Runs in matrix multiplication time, currently about $O(n^{2.37})$

This suggests that an interactive proof system could offer a more efficient way to verify the count of triangles in a graph, potentially achieving a faster runtime than traditional algorithms.

1.7 A Better Doubly-Efficient Interactive Proof for Counting Triangles

1. Counting Triangles
   - **Input:** $A \in \{0,1\}^{n\times n}$ representing the adjacency matrix of a graph.
   - **Desired Output:** $\frac{1}{6} \cdot \sum_{(i,j,k)\in\{n\}^3} A_{ij}A_{jk}A_{ik}$
   - **View A and A^2 as functions mapping $\{0,1\}^{\log n} \times \{0,1\}^{\log n}$ to $\mathbb{F}$:**
2. Defining the Polynomial
   - Define the polynomial:

   $h(X,Y) = \widetilde{(A^2)}(X,Y)\ \widetilde{A}(X,Y)$
3. The Protocol
   a) Apply the sum-check protocol to h
   b) **At the end of the protocol, V needs to evaluate:** $h(r_1,r_2) = \widetilde{(A^2)}(r_1,r_2)\ \widetilde{A}(r_1,r_2)$
   c) **V can evaluate $\widetilde{A}(r_1,r_2)$ on its own in $O(n^2)$ time.**
   d) **V uses the MatMult protocol to force P to compute $\widetilde{(A^2)}(r_1,r_2)$ for her.**

```
from sympy import symbols, expand, Poly
import numpy as np
from itertools import product

class TriangleCountingProtocol:
    def __init__(self, n):
        self.n = n
        # Create symbolic variables
        self.X, self.Y = symbols('X Y')

    def setup_adjacency(self, edges):
        """Setup adjacency matrix A"""
        self.A = np.zeros((self.n, self.n))
        for i, j in edges:
            self.A[i][j] = self.A[j][i] = 1

    def compute_A_squared(self):
        """Compute A^2"""
        return np.matmul(self.A, self.A)

    def MLE_A(self, x, y):
        """
        Multilinear extension of A
        $\\tilde{A}(x,y)$ - extension of adjacency matrix
        """
        result = 0
        for i, j in product(range(self.n), repeat=2):
            if self.A[i][j] == 1:
                term = 1
                # Create multilinear term for each 1 in matrix
                for bit, var in zip(format(i,
                ↪  f'0{self.n.bit_length()}b'), [x]):
                    term *= var if bit == '1' else (1-var)
                for bit, var in zip(format(j,
                ↪  f'0{self.n.bit_length()}b'), [y]):
                    term *= var if bit == '1' else (1-var)
                result += term
        return result

    def MLE_A_squared(self, x, y):
        """
        Multilinear extension of A^2
        $\\tilde{A}^2(x,y)$ - extension of A^2
        """
        A_squared = self.compute_A_squared()
        result = 0
        for i, j in product(range(self.n), repeat=2):
            if A_squared[i][j] != 0:
                term = A_squared[i][j]
                for bit, var in zip(format(i,
                ↪   f'0{self.n.bit_length()}b'), [x]):
                    term *= var if bit == '1' else (1-var)
                for bit, var in zip(format(j,
                ↪   f'0{self.n.bit_length()}b'), [y]):
                    term *= var if bit == '1' else (1-var)
                result += term
        return result

    def h(self, x, y):
        """
        h(X,Y) = $\\tilde{A}^2(X,Y)\\cdot\\tilde{A}(X,Y)$
        """
        return self.MLE_A_squared(x, y) * self.MLE_A(x, y)

    def verify(self, r1, r2):
        """Verifier's check at random points r1, r2"""
        A_val = self.MLE_A(r1, r2)
        A_squared_val = self.MLE_A_squared(r1, r2)
        h_val = A_val * A_squared_val
        print(f"$\\tilde{{A}}({r1},{r2}) = {A_val}$")
        print(f"$\\tilde{{A}}^2({r1},{r2}) = {A_squared_val}$")
        print(f"h({r1},{r2}) = {h_val}")
        return h_val

# Example usage
protocol = TriangleCountingProtocol(n=3)
# Example graph with one triangle
edges = [(0,1), (1,2), (0,2)]
protocol.setup_adjacency(edges)

print("Adjacency matrix:")
print(protocol.A)
print("\nA^2:")
print(protocol.compute_A_squared())

# Verifier picks random points
r1, r2 = 0.5, 0.5
result = protocol.verify(r1, r2)
```