Maths Based Problems and Solutions

1.Prime Numbers

Approach 1: Naive Method (Check Divisibility)

- Description: To check if a number n is prime, iterate through all numbers from 2 to $\sqrt{n}$, and check if any number divides n. If no number divides n, then n is prime.
- Time Complexity: O(n)
- Space Complexity: O(1)

Pseudocode (C++):

```
bool isPrime(int n) {
if (n <= 1) return false;
for (int i = 2; i * i <= n; ++i) {
if (n % i == 0) return false;
}
return true;
}
```

Dry Run:

For n = 11:

Check divisibility from 2 to $\sqrt{11}$, i.e., 2, 3.

11 is not divisible by 2 or 3 → Prime

- Output:

True

Approach 2: Optimized Check (Skip Even Numbers)

- Description: Similar to the naive method, but skip even numbers after checking for 2.
- Time Complexity: $O(\sqrt{n})$
- Space Complexity: $O(1)$

Pseudocode (C++):

```
bool isPrime(int n) {
if (n <= 1) return false;
if (n == 2) return true;  //2 is prime
if (n % 2 == 0) return false;  //Skip even numbers

for (int i = 3; i * i <= n; i += 2) {
if (n % i == 0) return false;
}
return true;
}
```

Dry Run:

For n = 19:

Check divisibility for numbers up to $\sqrt{19}$, i.e., 2, 3, 5.

19 is not divisible by 2, 3, or 5 →

Prime

- Output: True

2. Sieve of Eratosthenes

Approach 1: Simple Sieve of Eratosthenes

- Description: Use the Sieve of Eratosthenes to generate all primes up to N. Mark multiples of each prime starting from 2 as non-prime.

- Time Complexity: O(N log log N)

- Space Complexity: O(N)

Pseudocode (C++):

```
vector sieve(int N) {
vector prime(N + 1, true);
prime[0] = prime[1] = false;  //0 and 1 are not prime numbers

for (int i = 2; i * i <= N; ++i) {
if (prime[i]) {
for (int j = i * i; j <= N; j += i) {
prime[j] = false;
}
}
}
return
```

prime;

}

Dry Run:

For N = 10:

Initialize array prime: [false, false, true, true, true, true, true, true, true, true, true]

Mark multiples of each prime starting from 2:

For i = 2: Mark multiples of 2.

For i = 3: Mark multiples of 3.

For i = 5: Mark multiples of 5.

Prime numbers: [2, 3, 5, 7]

- Output: [false, false, true, true, false, true, false, true, false, false, false]

3. GCD/LCM

Approach 1: Naive GCD (Iterative Method)

- Description: Compute the GCD by checking all divisors of both numbers from 1 to min(a, b).

- Time Complexity: O(min(a, b))

- Space Complexity: O(1)

Pseudocode (C++):

int gcd(int a, int b) {

int result =

1;
for (int i = 1; i <= min(a, b); ++i) {
if (a % i == 0 && b % i == 0) {
result = i;
}
}
return result;
}
Dry Run:
For a = 36 and b = 60:

Check divisors: 1, 2, 3, 4, 6, 9, 12, 18, 36 for both numbers.
GCD = 12
- Output: 12

Approach 2: Euclidean Algorithm

- Description: Use the Euclidean algorithm to find the GCD efficiently. Keep reducing the problem by replacing a with b and b with a % b until b = 0.
- Time Complexity: O(log min(a, b))
- Space Complexity: O(1)

- Pseudocode (C++):
int gcd(int a, int b) {
while (b != 0)

{

a %= b;

swap(a, b);

}

return a;

}

Dry Run:

For a = 36 and b = 60:

a = 36, b = 60: a % b = 36, swap → a = 60, b = 36

a = 60, b = 36: a % b = 24, swap → a = 36, b = 24

a = 36, b = 24: a % b = 12, swap → a = 24, b = 12

a = 24, b = 12: a % b = 0, GCD = 12

- Output: 12

LCM Calculation

- Description: LCM can be computed using the formula: LCM(a, b) = (a * b) GCD(a, b).

- Time Complexity: O(log min(a, b))

- Space Complexity: O(1)

Pseudocode (C++):

int lcm(int a, int b) {

return (a * b) gcd(a,

b);
}

4. Fast Power (Modular Arithmetic)

Approach 1: Naive Power Calculation
- Description: Compute a pow b by repeatedly multiplying a for b times.
- Time Complexity: O(b)
- Space Complexity: O(1)
Pseudocode (C++):

```
int power(int a, int b) {
int result = 1;
for (int i = 0; i < b; ++i) {
result *= a;
}
return result;
}
```

Dry Run:
For a = 2, b = 5:

2 pow 5 = 32
- Output:

Approach 2: Exponentiation by Squaring (Efficient)

- Description: Use exponentiation by squaring to calculate a pow b in O(log b) time by breaking down the power into smaller parts.
- Time Complexity: O(log b)
- Space Complexity: O(1)
Pseudocode (C++):

```
int fastPower(int a, int b) {
int result = 1;
while (b > 0) {
if (b % 2 == 1) result *= a;
a *= a;
b /= 2;
}
return result;
}
```

Dry Run:
For a = 2, b = 5:
2(pow)5 = (2(pow)2)pow(2) * 2 = 32

- Output:

5. Pigeonhole Principle

Approach 1: Basic Principle

- Description: If n items are placed into m containers, and n > m, at least one container will have more than one item.
- Time Complexity: Depends on the problem.
- Space Complexity: Depends on the problem.
- Example:

Given 11 socks and 10 colors, at least two socks will have the same color.

- Output: True

6. Catalan Numbers

Approach 1: Recursive Formula

- Time Complexity: O(n) (factorial computation)
- Space Complexity: O(1)

Pseudocode (C++):

```
long long catalan(int n) {
long long res = 1;
for (int i = 0; i < n; ++i) {
res *= (2 * n -
```