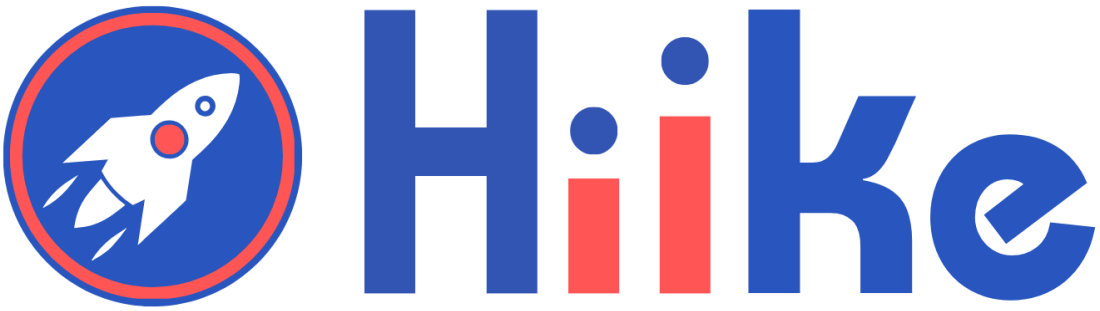
Class 3

Detailed Notes



Problem 1: GCD of Two Numbers

Link: GCD of Two Numbers

Solution:

```
class Solution {
public:
    int gcd(int A, int B) {
        if((max(A,B) % min(A,B)) == 0) {
            return min(A,B);
        }
        return gcd((max(A,B) % min(A,B)), min(A,B));
    }
};
```

Explanation:

The function calculates the Greatest Common Divisor (GCD) of two numbers using the Euclidean algorithm, which is based on the principle that the GCD of two numbers also divides their difference. This recursive approach repeatedly replaces the larger number by its remainder when divided by the smaller number until the remainder is zero. The smallest number at this point is the GCD.

Why it works:

The Euclidean algorithm works because it progressively reduces the problem size, ensuring that the final non-zero remainder of this division process is the GCD.

Time Complexity: O(log(min(A, B))), since the algorithm converges quickly by reducing the problem size in each step logarithmically.

Space Complexity: O(log(min(A, B))) for recursive stack space.

Problem 2: Prime Numbers

Link: Prime Numbers

Solution:

```
vector<int> Solution::sieve(int A) {
    vector<int> vec(A + 1, 0);
    vector<int> ans;
    for(int i = 2; i <= A; i++) {
        if(vec[i] == 0) {
            vec[i] = 1;
            ans.push_back(i);
            for(int j = i; (long long) i * j <= (long long) A; j++) {
                vec[i * j] = 1;
            }
        }
    }
    return ans;
}
```

Explanation:

The function generates all prime numbers up to a given number A using the Sieve of Eratosthenes. It iteratively marks the multiples of each prime number starting from 2. Numbers that remain unmarked after all multiples have been accounted for are prime.

Why it works:

The Sieve of Eratosthenes is efficient for finding all primes smaller than N because it minimizes the number of redundant checks for multiplicity by ensuring each number is marked by its smallest prime divisor only once.

Time Complexity: O(n log(log(n))), where n is the number A. This is due to the nature of the progression of marking multiples.

Space Complexity: O(n) for storing the sieve array.

Problem 3: Prime Factors

Link: Prime Factors

Solution:

```
class Solution {
public:
    vector<int> AllPrimeFactors(int N) {
        vector<int> ans;
        int num = 2;
        while(N > 1) {
            if(N % num == 0) {
                ans.push_back(num);
                while(N % num == 0) {
                    N = N / num;
                }
            }
            num++;
        }
        return ans;
    }
};
```

Explanation:

This function lists all unique prime factors of a number N. It sequentially tests each integer starting from 2 and checks if it is a divisor of N. If it is, the number is a prime factor, and it is repeatedly divided out of N until it no longer divides N.

Why it works:

It systematically checks each number's divisibility, ensuring that once a number divides N, all of its multiples are inherently not prime factors. This direct method guarantees all prime factors are found.

Time Complexity: O(sqrt(N)), as it only needs to check divisibility up to the square root of N in the worst case.

Space Complexity: O(1), ignoring the output list, as no additional space is used for computation.

Problem 4: Number of Factors of a Number

Link: [Number of Factors of a Number]

Solution:

```
#include <cmath>
#include <iostream>
using namespace std;

int main() {
    int N;
    cin >> N;
    int count = 0;
    for (int i = 1; i <= ((int)sqrt(N)); i++) {
        if (N % i == 0) {
            count += 2; // Count both divisors i and N/i
        }
    }
    if (((int)sqrt(N)) * ((int)sqrt(N)) == N) { // Check if N is a perfect
square
        count--; // Decrement count for a repeated factor at the square
root
    }
    cout << count;
    return 0;
}
```

Explanation:

This code snippet calculates the number of factors of N by iterating up to the square root of N. For each integer i that divides N without leaving a remainder, two factors are identified: i and N/i, except when i equals N/i i.e., N is a perfect square).

Why it works:

Each factor less than the square root has a corresponding factor greater than the square root. This approach effectively reduces the number of necessary checks to about the square root of N, making it much more efficient than checking up to N.

Time Complexity: O(sqrt(N)) — The loop iterates up to the square root of N.

Space Complexity: O(1) — Uses a constant amount of space.

Problem 5: Number of Factors for Multiple Queries

Link: [Number of Factors for Multiple Queries]

Solution:

```
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int Q;
    cin >> Q;
    vector<int> factors(10000, 0); // Precomputing smallest factors up to
9999
    for (int i = 2; i < 10000; i++) {
        if (factors[i] == 0) { // i is prime
            for (int j = i; i * j < 10000; j++) {
                if (factors[i * j] == 0) factors[i * j] = i;
            }
        }
    }
    while (Q--) {
        int temp, ans = 1;
        cin >> temp;
        int original = temp;
        while (temp > 1) {
            int smallestFactor = factors[temp];
            int count = 0;
            while (temp % smallestFactor == 0) {
                temp /= smallestFactor;
                count++;
            }
            ans *= (count + 1);
        }
        cout << ans << " ";
    }
    return 0;
}
```

Explanation:

This program first precomputes the smallest prime factor for each number up to 9999. For each query, it repeatedly divides the number by its smallest prime factor (found in the precomputed list) until all factors are extracted, counting the occurrences of each prime factor. The total number of factors is then the product of (count of each prime factor + 1) for each unique prime factor.

Why it works:

The precomputation speeds up factorization for each query, making the program efficient for multiple queries. It uses the property of numbers where the number of divisors can be derived from the exponent of its prime factorization.

Time Complexity: The preprocessing is O(n*log log n) using a sieve-like method. Each query is processed in O(log n) assuming that division and factor retrieval are constant time operations.

Space Complexity: O(1), ignoring the storage for the smallest prime factors and the input storage, as no additional significant space is used in the calculation.