Math Algorithms C++ Code

8. Mathematical Algorithms
- GCD and LCM (Euclidean algorithm)
- Modular Arithmetic
- Sieve of Eratosthenes (Prime Numbers)
- Exponentiation by Squaring (Fast Power)
- Matrix Exponentiation
- Combinatorics: nCr, nPr i want to do this revision give one by algorithm explaination time cmplexity and code in c++

Let's revise these mathematical algorithms with detailed explanations, time complexity, and C++ implementations.

---

1. GCD and LCM (Euclidean Algorithm)

Algorithm for GCD:

1. If $b = 0$, return a as the GCD.
2. Otherwise, recursively compute $\text{GCD}(b, a\%b)$.

Time Complexity:

- $O(\log(\min(a, b)))$

C++ Code:

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

int gcd(int a, int b) {
    return (b == 0) ? a : gcd(b, a % b);
}

int lcm(int a, int b) {
    return (a / gcd(a, b)) * b;
}

int main() {
    int a = 24, b = 36;
    cout << "GCD: " << gcd(a, b) << endl;
    cout << "LCM: " << lcm(a, b) << endl;
    return 0;
}
```

2. Modular Arithmetic

Concept:

- $(a + b)\%m = [(a\%m) + (b\%m)]\%m$
- $(a \times b)\%m = [(a\%m) \times (b\%m)]\%m$

Application: Modular Exponentiation

To compute $(base^{exp}) \mod m$.

Time Complexity:

- $O(\log(exp))$

C++ Code:

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

int modularExponentiation(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;

    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int main() {
    int base = 3, exp = 5, mod = 7;
    cout << "Modular Exponentiation: " << modularExponentiation(base, exp, mod) << endl;
    return 0;
}
```

---

3. Sieve of Eratosthenes (Prime Numbers)

Algorithm:

1. Create a boolean array `` `isPrime` `` of size $n + 1$, initialized to `` `true` ``.
2. Mark `` `isPrime[0]` `` and `` `isPrime[1]` `` as `` `false` ``.
3. For each number p from 2 to $\sqrt{n}$:
   - If `` `isPrime[p]` `` is `` `true` ``, mark all multiples of p as `` `false` ``.

Time Complexity:

- $O(n \log \log n)$

C++ Code:

```
cpp


#include <iostream>
#include <vector>
using namespace std;

void sieveOfEratosthenes(int n) {
    vector<bool> isPrime(n + 1, true);
    isPrime[0] = isPrime[1] = false;

    for (int p = 2; p * p <= n; p++) {
        if (isPrime[p]) {
            for (int i = p * p; i <= n; i += p) {
                isPrime[i] = false;
            }
        }
    }

    for (int i = 2; i <= n; i++) {
        if (isPrime[i]) {
            cout << i << " ";
        }
    }
    cout << endl;
}

int main() {
    int n = 50;
    sieveOfEratosthenes(n);
    return 0;
}
```

---

4. Exponentiation by Squaring (Fast Power)

Algorithm:

1. If $exp = 0$, return 1.
2. If exp is even, compute $(base^{exp/2})^2$.
3. If exp is odd, compute $base \times (base^{exp-1})$.

Time Complexity:

- $O(\log(exp))$

C++ Code:

```
cpp


#include <iostream>
using namespace std;

long long fastPower(long long base, long long exp) {
    if (exp == 0) return 1;
    long long half = fastPower(base, exp / 2);
    if (exp % 2 == 0) {
        return half * half;
    } else {
        return base * half * half;
    }
}

int main() {
    long long base = 2, exp = 10;
    cout << "Fast Power: " << fastPower(base, exp) << endl;
    return 0;
}
```

---

5. Matrix Exponentiation

Concept:

- Used to compute terms of linear recurrence relations efficiently.
- Multiply the matrix with itself for $\log(n)$ times.

Time Complexity:

- $O(k^3 \log(n))$ where k is the matrix dimension.

C++ Code:

```cpp
#include <iostream>
#include <vector>
using namespace std;

typedef vector<vector<int>> Matrix;

Matrix multiply(Matrix &A, Matrix &B, int mod) {
    int n = A.size();
    Matrix result(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                result[i][j] = (result[i][j] + (A[i][k] * B[k][j]) % mod) % mod;
            }
        }
    }
    return result;
}

Matrix matrixExponentiation(Matrix base, int exp, int mod) {
    int n = base.size();
    Matrix result(n, vector<int>(n, 0));
    for (int i = 0; i < n; i++) result[i][i] = 1; // Identity matrix

    while (exp > 0) {
        if (exp % 2 == 1) result = multiply(result, base, mod);
        base = multiply(base, base, mod);
        exp /= 2;
    }
    return result;
}

int main() {
    Matrix base = {{1, 1}, {1, 0}};
    int exp = 10, mod = 1e9 + 7;
    Matrix result = matrixExponentiation(base, exp, mod);

    for (const auto &row : result) {
        for (int val : row) cout << val << " ";
        cout << endl;
    }
    return 0;
}
```

---

6. Combinatorics: nCr, nPr

Algorithm for nCr:

- $nCr = \frac{n!}{r! \times (n-r)!}$

Time Complexity:

- $O(n)$ for factorial computation.

C++ Code:

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

long long factorial(int n) {
    long long result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

long long nCr(int n, int r) {
    return factorial(n) / (factorial(r) * factorial(n - r));
}

long long nPr(int n, int r) {
    return factorial(n) / factorial(n - r);
}

int main() {
    int n = 5, r = 2;
    cout << "nCr: " << nCr(n, r) << endl;
    cout << "nPr: " << nPr(n, r) << endl;
    return 0;
}
```

Let me know which part you'd like more clarification on!

is there algorithm left

We've covered the main algorithms in the requested list. Here's a summary of what's been addressed:

Covered Algorithms:

1. **GCD and LCM (Euclidean Algorithm)**
    - Algorithm and code provided.
2. **Modular Arithmetic**
    - Concepts, modular exponentiation, and code provided.
3. **Sieve of Eratosthenes (Prime Numbers)**
    - Algorithm and code provided.
4. **Exponentiation by Squaring (Fast Power)**
    - Algorithm and code provided.
5. **Matrix Exponentiation**
    - Algorithm and code provided.
6. **Combinatorics: nCr, nPr**
    - Algorithm and code provided.

Any Missing Algorithms?

From your initial list, we've covered all the topics. However, if you're interested in related mathematical algorithms or want to dive deeper into specific areas like:

- **Fermat's Little Theorem** (for modular inverses)
- **Chinese Remainder Theorem**
- **Extended Euclidean Algorithm** (to find coefficients of $ax + by = gcd(a, b)$)
- **Number of Divisors / Sum of Divisors**
- **Segmented Sieve**

Let me know, and I can provide those as well!

give missing also

Let's cover the **missing algorithms** related to mathematical computation:

---

1. Fermat's Little Theorem (Modular Inverse)

Concept:

- Fermat's Little Theorem states:
  For a prime p, if a is not divisible by p, then:
  $a^{p-1} \equiv 1 \mod p$
- Modular inverse of $a \mod p$ (where p is prime) is:
  $a^{-1} \equiv a^{p-2} \mod p$

Time Complexity:

- $O(\log(p))$ using modular exponentiation.

C++ Code:

cpp

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

int modularExponentiation(int base, int exp, int mod) {
    int result = 1;
    base = base % mod;

    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        base = (base * base) % mod;
        exp /= 2;
    }
    return result;
}

int modularInverse(int a, int p) {
    return modularExponentiation(a, p - 2, p);
}

int main() {
    int a = 3, p = 7; // p must be prime
    cout << "Modular Inverse: " << modularInverse(a, p) << endl;
    return 0;
}
```

---

2. Chinese Remainder Theorem (CRT)

Concept:

- Solves systems of simultaneous congruences:

$$\begin{aligned} x &\equiv a_1 \mod m_1 \\ x &\equiv a_2 \mod m_2 \\ &\dots \end{aligned}$$

  where $m_1, m_2, \dots$ are pairwise coprime.

Algorithm:

1. Calculate the product $M = m_1 \times m_2 \times \dots$.
2. For each congruence $x \equiv a_i \mod m_i$:
   - Compute $M_i = M/m_i$.
   - Find y_i, the modular inverse of $M_i \mod m_i$.
   - Add $a_i \times M_i \times y_i$ to the result.
3. Result x is $\text{result} \mod M$.

Time Complexity:

- $O(n \log(M))$ where n is the number of congruences.

C++ Code:

cpp

```cpp
#include <iostream>
#include <vector>
using namespace std;

int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

int modularInverse(int a, int mod) {
    int m0 = mod, x0 = 0, x1 = 1;
    if (mod == 1) return 0;
    while (a > 1) {
        int q = a / mod;
        int t = mod;
        mod = a % mod, a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }
    if (x1 < 0) x1 += m0;
    return x1;
}

int chineseRemainder(vector<int> nums, vector<int> rems) {
    int prod = 1;
    for (int num : nums) prod *= num;

    int result = 0;
    for (int i = 0; i < nums.size(); i++) {
        int pp = prod / nums[i];
        result += rems[i] * modularInverse(pp, nums[i]) * pp;
    }

    return result % prod;
}

int main() {
    vector<int> nums = {3, 5, 7};
    vector<int> rems = {2, 3, 2};
    cout << "Solution: " << chineseRemainder(nums, rems) << endl;
    return 0;
}
```

---

3. Extended Euclidean Algorithm

Concept:

- Solves the equation $ax + by = \gcd(a, b)$.
- It computes x and y (Bezout's coefficients).

Time Complexity:

- $O(\log(\min(a, b)))$

C++ Code:

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

int gcdExtended(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int x1, y1;
    int gcd = gcdExtended(b, a % b, x1, y1);
    x = y1;
    y = x1 - (a / b) * y1;
    return gcd;
}

int main() {
    int a = 30, b = 20, x, y;
    int gcd = gcdExtended(a, b, x, y);
    cout << "GCD: " << gcd << ", x: " << x << ", y: " << y << endl;
    return 0;
}
```

---

4. Number of Divisors / Sum of Divisors

Algorithm for Number of Divisors:

1. Perform prime factorization of n:
   $n = p_1^{e_1} \times p_2^{e_2} \times \dots$
2. Number of divisors is: $(e_1 + 1) \times (e_2 + 1) \times \dots$

Time Complexity:

- $O(\sqrt{n})$

C++ Code:

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

int countDivisors(int n) {
    int count = 0;
    for (int i = 1; i <= sqrt(n); i++) {
        if (n % i == 0) {
            if (i == n / i) count++; // Perfect square
            else count += 2;
        }
    }
    return count;
}

int main() {
    int n = 36;
    cout << "Number of Divisors: " << countDivisors(n) << endl;
    return 0;
}
```

---

5. Segmented Sieve

Concept:

- Used to find primes in a specific range $[L, R]$.

Algorithm:

1. Use Sieve of Eratosthenes to find primes up to $\sqrt{R}$.
2. Mark non-prime numbers in $[L, R]$ using these primes.

Time Complexity:

- $O((R - L + 1) \log \log R)$

C++ Code:

```cpp
#include <iostream>
#include <vector>
using namespace std;

void simpleSieve(int limit, vector<int> &prime) {
    vector<bool> isPrime(limit + 1, true);
    for (int p = 2; p * p <= limit; p++) {
        if (isPrime[p]) {
            for (int i = p * p; i <= limit; i += p) {
                isPrime[i] = false;
            }
        }
    }
    for (int p = 2; p <= limit; p++) {
        if (isPrime[p]) prime.push_back(p);
    }
}

void segmentedSieve(int L, int R) {
    int limit = sqrt(R);
    vector<int> prime;
    simpleSieve(limit, prime);

    vector<bool> isPrime(R - L + 1, true);
    for (int p : prime) {
        int start = max(p * p, (L + p - 1) / p * p);
        for (int j = start; j <= R; j += p) {
            isPrime[j - L] = false;
        }
    }

    for (int i = 0; i <= R - L; i++) {
        if (isPrime[i]) cout << i + L << " ";
    }
    cout << endl;
}

int main() {
    int L = 10, R = 50;
    segmentedSieve(L, R);
    return 0;
}
```

Let me know if you need further clarification or additional algorithms!