Q4: Optimus Prime — Print All Primes up to N

---

1. Understand the Problem

- **Read & Identify:** Given an integer n, print all prime numbers between 1 and n (inclusive).
- **Goal:** Find all primes ≤ n.
- **Paraphrase:** Numbers greater than 1 that have no divisors other than 1 and themselves.

---

2. Input, Output, & Constraints

- **Input:**

```
8
```

- **Output:**

```
2 3 5 7
```

Constraints:

- 1 ≤ n ≤ 100,000
- Output size ≤ n
- Target time complexity: O(n log log n) with sieve

3. Approaches

Approach 1: Naive Check (Trial Division)

- **Idea:**
  - For every number from 2 to n, check if it's prime by trying to divide by numbers up to √num.

Pseudocode:

```
function printPrimesNaive(n):
    for i from 2 to n:
        isPrime = true
        for j from 2 to sqrt(i):
            if i % j == 0:
                isPrime = false
                break
```

```
        if isPrime:
            print i
```

Complexity:

- Time: $O(n\sqrt{n})$
- Space: $O(1)$

Approach 2: Sieve of Eratosthenes (Optimized)

- **Idea:**
    - Assume all numbers 2..n are prime.
    - Start from 2, mark all its multiples as non-prime.
    - Repeat for next unmarked number.
    - Remaining unmarked numbers are primes.

Pseudocode:

```
function sieve(n):
    isPrime = array[0..n] filled with true
    isPrime[0] = false, isPrime[1] = false

    for i from 2 to sqrt(n):
        if isPrime[i]:
            for j from i*i to n step i:
                isPrime[j] = false

    for i from 2 to n:
        if isPrime[i]:
            print i
```

Complexity:

- Time: $O(n \log \log n)$
- Space: $O(n)$

4. Justification / Proof of Optimality

- Naive method is too slow for n up to 10^5.
- Sieve of Eratosthenes runs in $O(n \log \log n)$ which is optimal for this range.
- The sieve guarantees correctness by systematically eliminating composites.

5. Variants / Follow-Ups

- Print primes between L and R (Segmented Sieve).
- Count primes up to n (Prime Counting Function).
- Find the k-th prime ≤ n.
- Applications in number theory (Goldbach's conjecture, twin primes).

Q5: Calculate nCr

---

1. Understand the Problem

- **Read & Identify:** Given two integers, n (total items) and r (items to choose), the goal is to calculate the binomial coefficient, nCr, which represents the number of distinct combinations.
- **Goal:** Compute the value of nCr using the standard combinatorial formula.
- **Paraphrase:** Find the number of distinct subsets of size r from a set of size n.

---

2. Input, Output, & Constraints

- **Input:**

```
Two non-negative integers, n and r.
```

- **Output:**

```
A single integer representing the calculated value of nCr.
```

Constraints:

- $1 \le n \le 20$
- $1 \le r \le n$
- The result will fit within a standard 64-bit integer ($\approx 1.8 \times 10^{19}$).

3. Approaches

Approach 1: Direct Factorial Calculation (Naive)

- **Idea:**
  - Directly compute the factorials for n, r, and $(n-r)$, then perform the division. $nCr = n!/(r! * (n-r)!)$

Pseudocode:

```
function Calculate_nCr(n, r):
    // Requires a data type that can hold 20! (long long/64-bit integer)
    numerator = Factorial(n)
    denominator = Factorial(r) * Factorial(n - r)
    return numerator / denominator
```

Complexity:

- Time: O(n)

- Space: O(1) space.

Approach 2: Optimized Multiplicative Formula (Preferred)

- **Idea:**
    - Simplify the fraction before calculation to minimize the size of intermediate numbers, which is crucial for larger n. The simplified form cancels out the largest factorial term, $(n-r)!$.

Pseudocode:

```
function Calculate_nCr(n, r):
    // Use nCr = nC(n-r) property for fewer iterations
    if r > n / 2:
        r = n - r

    result = 1
    // Loop r times
    for i from 1 to r:
        // Current calculation: result * (n-i+1) / i
        result = result * (n - i + 1)
        result = result / i // Division is guaranteed to be exact
    return result
```

Complexity:

- Time: $O(\min(r,n-r))$, which is O(n)
- Space: O(1) space

4. Justification / Proof of Optimality

- Approach 2 is the better solution. While both approaches are O(n) time complexity, Approach 2 keeps the intermediate values much smaller, which minimizes the risk of overflow. It directly computes nCr step-by-step, ensuring each partial product is a valid integer combination value, which is inherently safer than computing three separate, massive factorials (as in Approach 1) and hoping their ratio fits the integer type.

5. Variants / Follow-Ups

- Large Constraints ($n,r \approx 10^9$): When n and r are very large and the answer must be computed modulo p. This requires number theory techniques like Lucas Theorem or calculating factorials and their modular inverses using Fermat’s Little Theorem.
- Dynamic Programming (Pascal's Identity): For scenarios where many nCr values are needed, the relation $nCr=(n-1)C(r-1)+(n-1)Cr$ allows filling a table (Pascal's Triangle) in $O(n^2)$ time.
- Permutations (nPr): The problem of calculating $nPr=n!/(n-r)!$ involves a similar multiplicative approach, simply stopping before dividing by r!.

Q6: Binary To Decimal Conversion

1. Input, Output, & Constraints

- **Input:**

```
1011
```

- **Output:**

```
11
```

Constraints:

- Binary number contains only 0 and 1.
- Length of binary number ≤ 32 (or as per system integer limit).

2. Approaches

Approach 1: Positional Value (Iterative)

- **Idea:**
    - Each binary digit represents a power of 2. Starting from the least significant bit (LSB), multiply each bit by 2^position and sum them.

Pseudocode:

```
function binaryToDecimal(binary):
    decimal = 0
    length = len(binary)
    for i in range(0, length):
        bit = int(binary[length - 1 - i])
        decimal += bit * (2^i)
    return decimal
```

Complexity:

- Time: O(n), n = number of bits
- Space: O(1)

Approach 2: Left-to-Right Multiplication (Accumulation)

- **Idea:**
    - Traverse the binary number from left to right, multiply accumulated result by 2, then add current bit.
    - Works for very large binary numbers.
    - Slightly more efficient than computing powers explicitly.

Pseudocode:

```
function binaryToDecimal(binary):
    decimal = 0
    for bit in binary:
        decimal = decimal * 2 + int(bit)
    return decimal
```

Complexity:

- Time: O(n)
- Space: O(1)

3. Justification / Proof of Optimality

- Optimality: Approach 2 is optimal in terms of simplicity and efficiency.
- Comparison:
- Approach 1: Direct calculation using powers → more verbose.
- Approach 2: Accumulative method → elegant, in-place.

4. Variants / Follow-Ups

- Decimal → Binary conversion
- Binary → Hexadecimal conversion
- Large binary strings beyond integer limit → Use BigInt or string manipulation
- Summing multiple binary numbers efficiently

Q7: Decimal to Binary Conversion

1. Input, Output, & Constraints

- **Input:**

```
11
```

- **Output:**

```
1011
```

Constraints:

- Decimal number ≥ 0
- Decimal number ≤ maximum integer limit of language

2. Approaches

Approach 1: Repeated Division by 2

- **Idea:**
  - Keep dividing the decimal number by 2. The remainder at each step forms the binary digits from least significant bit (LSB) to most significant bit (MSB).

Pseudocode:

```
function decimalToBinary(n):
    if n == 0:
        return "0"
    binary = ""
    while n > 0:
        remainder = n % 2
        binary = str(remainder) + binary
        n = n // 2
    return binary
```

Complexity:

- Time: O(log n)
- Space: O(log n) (for storing binary digits)

Approach 2: Using Bit Manipulation

- **Idea:**
  - Extract bits from decimal number using bitwise AND and right shift operations.

Pseudocode:

```
function decimalToBinary(n):
    if n == 0:
        return "0"
    binary = ""
    while n > 0:
        bit = n & 1
        binary = str(bit) + binary
        n = n >> 1
    return binary
```

Complexity:

- Time: O(log n)
- Space: O(log n

3. Justification / Proof of Optimality

- Optimality: Approach 1 & 2 are optimal and educational.
- Comparison:
- Approach 1: Classic method using division → easy to understand.
- Approach 2: Bitwise method → faster in low-level operations.

4. Variants / Follow-Ups

- Binary → Decimal conversion
- Decimal → Hexadecimal conversion
- Decimal → Binary for negative numbers (2's complement)
- Fast conversion using recursion or stack

Q8: Print Continuous Character Pattern

1. Input, Output, & Constraints

- **Input:**

```
5
```

- **Output:**

```
A
BC
CDE
DEFG
EFGHI
```

2. Approaches

Approach 1: Using ASCII Values

- **Idea:**
  - Use ASCII values of characters. Start from 'A' (ASCII 65), and for each row, print consecutive letters using (ASCII value) % 26 + 65 to handle cyclic behavior.

Pseudocode:

```
function printPattern(n):
    for row in range(1, n+1):
        start_char = 65 + (row - 1)   # 'A' = 65
        for col in range(row):
            char_to_print = chr(65 + ((start_char - 65 + col) % 26))
```

```
            print(char_to_print, end="")
        print()   # New line after each row
```

Complexity:

- Time: O(n^2) → Each row has up to n letters
- Space: O(1) → Only loop variables

Approach 2: Using String Arithmetic (Optional)

- **Idea:**
  - Pre-generate the alphabet string "ABCDEFGHIJKLMNOPQRSTUVWXYZ" and use slicing with modulo to handle cyclic letters.

Pseudocode:

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
function printPattern(n):
    for row in range(1, n+1):
        start_index = row - 1
        for col in range(row):
            index = (start_index + col) % 26
            print(alphabet[index], end="")
        print()
```

Complexity:

- Time: O(n^2)
- Space: O(1)

3. Justification / Proof of Optimality

- Optimality: Both approaches are efficient; Approach 1 is straightforward using ASCII, Approach 2 is more intuitive for beginners.
- Comparison:
- ASCII arithmetic → Less memory, direct computation
- String-based → Easier to read and maintain, especially for cyclic operations

4. Variants / Follow-Ups

- Change starting letter for the first row (instead of always 'A')
- Print pattern in reverse order
- Allow lowercase letters or custom alphabet sets
- Print continuous character diamond pattern

Q12: Count All Digits of a Number

1. Input, Output, & Constraints

- **Input:**

```
234
```

- **Output:**

```
3
```

Constraints:

- 0 ≤ n ≤ 5000
- n has no leading zeros except if n = 0

---

2. Approaches

Approach 1: Using Division (Iterative)

- **Idea:**
  - Divide n by 10 repeatedly, counting how many times until n becomes 0.

Java Code:

```java
public static int countDigits(int n) {
    if (n == 0) return 1;  // Edge case

    int count = 0;
    while (n > 0) {
        n /= 10;
        count++;
    }
    return count;
}
```

Complexity:

- Time: O(log n) → number of digits
- Space: O(1)

Approach 2: Using String Conversion

- **Idea:**
  - Convert the integer to a string and count the number of characters.

Java Code:

```
public static int countDigits(int n) {
    return String.valueOf(n).length();
}
```

Complexity:

- Time: O(log n) → traverses digits to convert to string
- Space: O(log n) → stores string representation

Approach 3: Using Logarithm (Math.log10)

- **Idea:**
  - The number of digits in a positive integer n is floor(log10(n)) + 1.
  - Edge case: if n = 0, the number of digits is 1.

Java Code:

```
public static int countDigits(int n) {
    if (n == 0) return 1;  // Edge case
    return (int)(Math.log10(n)) + 1;
}
```

Complexity:

- Time: O(1) → single mathematical operation
- Space: O(1)

---

3. Justification / Proof of Optimality

- Division → simple and memory efficient
- String → concise and intuitive
- Logarithm → fastest for large numbers

---

4. Variants / Follow-Ups

- Count digits in negative numbers
- Count digits in very large numbers (BigInteger in Java)
- Count digits in binary, octal, or hexadecimal representation

Q13: Check for Perfect Number

1. Understand the Problem

- **Paraphrase:** Proper divisors = all positive divisors excluding the number itself. A perfect number is equal to the sum of its proper divisors.

---

2. Input, Output, & Constraints

- **Input:**

```
n
```

- **Output:**

```
Boolean true or false
```

Constraints:

- 1 ≤ n ≤ 5000

---

3. Approaches

Approach 1: Iterating Over Divisors

- **Idea:**
  - Sum all divisors from 1 to n/2 (proper divisors)
  - If sum equals n, return true; else return false

Java Code:

```java
public static boolean isPerfectNumber(int n) {
    if (n == 1) return false;  // 1 is not a perfect number

    int sum = 0;
    for (int i = 1; i <= n / 2; i++) {
        if (n % i == 0) {
            sum += i;
        }
```

```
    }
    return sum == n;
}
```

Complexity:

- Time: O(n) → iterate up to n/2
- Space: O(1)

Approach 2: Iterating up to √n (Optimized)

- **Idea:**
  - Proper divisors come in pairs (i, n/i)
  - Iterate i from 1 to √n and add both divisors to sum
  - Exclude n itself from the sum

Java Code:

```
public static boolean isPerfectNumber(int n) {
    if (n == 1) return false;  // Edge case

    int sum = 1;  // 1 is always a proper divisor
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            sum += i;
            int pair = n / i;
            if (pair != i) sum += pair;  // Avoid adding sqrt twice
        }
    }
    return sum == n;
}
```

Complexity:

- Time: O(√n) → faster for larger numbers
- Space: O(1)

---

4. Justification / Proof of Optimality

- Approach 1 is simple and easy to implement.
- Approach 2 is more efficient, especially for larger n, as it avoids unnecessary iterations.

---

5. Variants / Follow-Ups

- Check for abundant numbers (sum of divisors > n) or deficient numbers (sum < n)
- Find all perfect numbers up to a given limit
- Handle very large numbers using optimized divisor sum formulas

Q14: GCD/HCFof Two Numbers

1. Understand the Problem

- **Paraphrase:** Find the highest number that both n1 and n2 are divisible by.

---

2. Input, Output, & Constraints

- **Input:**

```
4, 6
```

- **Output:**

```
Output: 2

Divisors of 4: 1, 2, 4

Divisors of 6: 1, 2, 3, 6

GCD = 2
```

Constraints:

- 1 ≤ n1, n2 ≤ 1000

---

3. Approaches

Approach 1: Using Brute Force

- **Idea:**
  - Iterate from min(n1, n2) down to 1
  - First number that divides both is the GCD

Java Code:

```
public static int gcdBruteForce(int n1, int n2) {
    int min = Math.min(n1, n2);
    for (int i = min; i >= 1; i--) {
```

```
        if (n1 % i == 0 && n2 % i == 0) {
            return i;
        }
    }
    return 1; // This line is never really reached because 1 always divides
}
```

Complexity:

- Time: O(min(n1, n2))
- Space: O(1)

Approach 2: Using Euclidean Algorithm (Optimized)

- **Idea:**
  - GCD(a, b) = GCD(b, a % b)
  - Repeat until b = 0, then GCD = a

Java Code:

```
public static int gcdEuclidean(int n1, int n2) {
    while (n2 != 0) {
        int temp = n2;
        n2 = n1 % n2;
        n1 = temp;
    }
    return n1;
}
```

Complexity:

- Time: O(log(min(n1, n2))) → very efficient
- Space: O(1)

---

4. Justification / Proof of Optimality

- Brute force is simple but inefficient for large numbers.
- Euclidean algorithm is optimal, widely used, and handles large inputs efficiently.

---

5. Variants / Follow-Ups

- Find LCM using GCD: LCM(a, b) = (a * b) / GCD(a, b)
- Extend to more than two numbers
- Find GCD of an array using pairwise GCD

Q15: LCM of Two Numbers

1. Understand the Problem

- **Paraphrase:** Find the least number that both n1 and n2 divide evenly. Can be computed efficiently using GCD: LCM(a, b) = (a * b) / GCD(a, b)

---

2. Input, Output, & Constraints

- **Input:**

```
4, 6
```

- **Output:**

```
12

Multiples of 4: 4, 8, 12, ...

Multiples of 6: 6, 12, 18, ...

LCM = 12
```

---

3. Approaches

Approach 1: Using Formula LCM = (n1 * n2) / GCD

- **Idea:**
  - Compute GCD first using Euclidean algorithm
  - Then LCM = (n1 * n2) / GCD(n1, n2)

Java Code:

```java
public static int lcm(int n1, int n2) {
    int a = n1, b = n2;
    while (b != 0) {
        int temp = b;
        b = a % b;
        a = temp;
    }
    int gcd = a;
```

```java
    return (n1 * n2) / gcd;
}
```

Complexity:

- Time: O(log(min(n1, n2))) → for computing GCD
- Space: O(1)

Approach 2: Brute Force Multiples (Less Efficient)

- **Idea:**
  - Start from max(n1, n2) and check each number incrementally until divisible by both

Java Code:

```java
public static int lcmBruteForce(int n1, int n2) {
    int lcm = Math.max(n1, n2);
    while (true) {
        if (lcm % n1 == 0 && lcm % n2 == 0) return lcm;
        lcm++;
    }
}
```

Complexity:

- Time: O(n1 * n2) → inefficient for large numbers
- Space: O(1)

---

4. Justification / Proof of Optimality

- Formula using GCD is efficient and widely used.
- Brute force is simple but slow for larger numbers.

---

5. Variants / Follow-Ups

- LCM of more than two numbers (compute pairwise LCM)
- LCM using prime factorization
- LCM of large numbers using BigInteger