Q100: Power of Two Integers

1. Problem Understanding

- You are given a positive integer N (â‰¤ 10â□¹).
- You must determine if there exist two integers A > 0 and P > 1 such that:
- A^P=N
- If yes â†’ print 1,
- else â†’ print 0.

---

2. Constraints

- 0â‰¤Nâ‰¤10^9
- A>0
- P>1
- A and P are integers
- Must fit in 32-bit signed integer

---

3. Edge Cases

- N = 0 â†’ cannot be expressed â†’ 0
- N = 1 â†’ 1^ð□‘ƒ=1 for any P â†’ 1
- Perfect powers like 4 (=2Â²), 8 (=2Â³), 9 (=3Â²), 16 (=4Â²) â†’ 1
- Non-perfect powers like 10, 12, 20 â†’ 0

---

4. Examples

```
Input:

64


Output:

1


Explanation:
4^3=64 and also 8^2=64
```

---

5. Approaches

Approach 1: Brute Force (Iterative)

Idea:

- Try every possible base A from 2 to âˆšN.
- For each A, multiply repeatedly (A * A, A * A * A, etc.)
- until you exceed or match N.

Java Code:

```
public static int isPower(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    for (int a = 2; a * a <= n; a++) {
        long p = a; // use long to avoid overflow
        while (p <= n) {
            p *= a;
            if (p == n) return 1;
        }
    }
    return 0;
}
```

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - Outer loop: âˆšN
  - Inner loop: logâ‚□(N)
  - Overall: O(âˆšN Ã— logN)
- ðŸ'¾ Space Complexity
  - O(1)

Approach 2: Using Logarithms (Mathematical)

Idea:

- We know:
- N=A^PâŸ¹P=log(A)/log(N) â€‹
- If P is an integer, then N can be expressed as A^P.
- Use math and rounding to check if P is close to an integer.

Java Code:

```
public static int isPowerLog(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    for (int a = 2; a * a <= n; a++) {
```

```java
        double p = Math.log(n) / Math.log(a);
        if (Math.abs(p - Math.round(p)) < 1e-10 && Math.round(p) > 1)
            return 1;
    }
    return 0;
}
```

Complexity (Time & Space):

- â±ï¸ Time Complexity
- O(âˆšN)
- ðŸ'¾ Space Complexity
- O(1)
- Note
- Floating-point precision can cause issues for large N.
- To reduce errors, use Math.round() with tolerance (1e-10).

Approach 3: Recursive Power Check

Idea:

- Recursively try multiplying the base until the power exceeds N.
- For each base A, define:
  - check(A, current, N):
    - if current == N â†' true
    - if current > N â†' false
    - else â†' check(A, current * A, N)

Java Code:

```java
public static boolean check(long base, long current, long n) {
    if (current == n) return true;
    if (current > n) return false;
    return check(base, current * base, n);
}

public static int isPowerRecursive(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    for (int a = 2; a * a <= n; a++) {
        if (check(a, a, n)) return 1;
    }
    return 0;
}
```

Complexity (Time & Space):

- â±ï¸ Time Complexity

    - Similar to iterative: O(âˆšN Ã— logN)
- ðŸ’¾ Space Complexity
    - O(logN) (due to recursion depth)

Approach 4: Precompute Powers (Optimization)

Idea:

- Instead of computing on the fly,
- store all possible powers of 2â€“9 that are â‰¤ 10â□¹ in a HashSet.
- Then simply check if N exists in the set.

Java Code:

```
public static int isPowerPrecompute(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    HashSet<Long> powers = new HashSet<>();
    for (int a = 2; a <= 31622; a++) { // 31622^2 â‰ˆ 10^9
        long val = a * a;
        while (val <= 1_000_000_000L) {
            powers.add(val);
            val *= a;
        }
    }
    return powers.contains((long)n) ? 1 : 0;
}
```

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
    - Precomputation: O(âˆšN Ã— logN)
    - Query: O(1)
- ðŸ’¾ Space Complexity
    - O(M) where M = number of precomputed powers (~few thousand)

Approach 5 â€” Optimized Root-Based Method (Best One)

Idea: -For a given exponent p, compute the integer base a = round(n^(1/p)). Then verify if a^p == n.

- We only need to check exponents p from 2 to 31 (since 2^31 > 2,147,483,647).

Why Approach 5 Works (Reason):

- Instead of iterating all possible bases (which is huge for large n), we iterate over small exponents (2â€“31) â€” just 30 iterations total.
- Computing n^(1/p) gives a direct estimate of the base, avoiding billions of multiplications.
- Using long and rounding ensures accuracy and no overflow.

- For max input 2147483647, it completes instantly. **Steps:**
- Handle base cases: if n == 0 â†' return 0, if n == 1 â†' return 1.
- Loop p from 2 to 31.
- Compute approximate a = n^(1/p) using Math.pow.
- Round to nearest integer, then multiply a p times (using long to avoid overflow).
- If the result equals n, return 1.
- After loop ends, return 0.

Java Code:

```
public static int isPowerOptimized(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    for (int p = 2; p <= 31; p++) {
        double a = Math.pow(n, 1.0 / p);
        int base = (int) Math.round(a);

        long val = 1;
        for (int i = 0; i < p; i++) {
            val *= base;
            if (val > n) break;
        }

        if (val == n) return 1;
    }
    return 0;
}
```

Complexity (Time & Space):

- â□±ï¸□ Time Complexity
  - O(31 Ã— logN) â†' practically O(1)
- ðŸ'¾ Space Complexity
  - O(1)

---

6. Justification / Proof of Optimality

- Each approach ensures both A and P are integers.
- Brute-force and recursive approaches rely purely on integer arithmetic (no rounding errors).
- Logarithmic approach offers speed, but may have floating-point inaccuracies.
- Precomputation is best suited for multiple queries, trading off space for speed.

---

7. Variants / Follow-Ups

- Find All (A, P) pairs

- â†' Instead of returning 1, store or print all valid pairs where A^P=N.
- Find largest P for a given N
- â†' Find the highest exponent such that A^P=N.
- Check Power of Base K
- â†' Restrict A to a given base K (e.g., only check if N is a power of 2).
- For multiple queries
- â†' Use precomputation + HashSet lookup for O(1) query per number.

---

8. Tips & Observations

- Avoid using Math.pow() in integer problems (floating-point rounding errors).
- Always use long while multiplying to avoid integer overflow.
- For recursion, stop early when product exceeds N â€" this prunes large search space.
- Logarithmic method is fastest for single queries but less reliable for precision-critical code.
- Precomputation is perfect for repeated inputs in large datasets.
- A need only go up to âˆšN because A^2 must be â‰¤ N for P â‰¥ 2.

Q12: Count All Digits of a Number

1. Input, Output, & Constraints

- **Input:**

```
234
```

- **Output:**

```
3
```

Constraints:

- 0 â‰¤ n â‰¤ 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:

```
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) â†' 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) â†' traverses digits to convert to string
- Space: O(log n) â†' 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) â†' single mathematical operation
- Space: O(1)

---

3. Justification / Proof of Optimality

- Division â†' simple and memory efficient
- String â†' concise and intuitive
- Logarithm â†' 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 â‰¤ n â‰¤ 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:

```
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) â†’ iterate up to n/2
- Space: O(1)

Approach 2: Iterating up to âˆšn (Optimized)

- **Idea:**
  - Proper divisors come in pairs (i, n/i)
  - Iterate i from 1 to âˆšn 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(â^šn) â†' 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 â‰¤ n1, n2 â‰¤ 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))) â†' 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:

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

Complexity:

- Time: O(log(min(n1, n2))) â†’ 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:

```
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) â†’ 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

Q4: Optimus Prime â€” 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 â‰¤ 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 â‰¤ n â‰¤ 100,000
- Output size â‰¤ 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 itâ€™s prime by trying to divide by numbers up to âˆšnum.

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(nâˆšn)
- 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 10â□µ.
- 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 â‰¤ n.
- Applications in number theory (Goldbachâ€™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:

- 1â‰¤nâ‰¤20
- 1â‰¤râ‰¤n
- The result will fit within a standard 64-bit integer (â‰ˆ1.8Ã—10 ^19).

3. Approaches

Approach 1: Direct Factorial Calculation (Naive)

- **Idea:**

    - Directly compute the factorials for n, r, and (nâˆ’r), then perform the division. nCr=n!/(r!âˆ—(nâˆ’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, (nâˆ’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,nâˆ’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,râ‰ˆ10^9): When n and r are very large and the answer must be computed moduloÂ p. This requires number theory techniques like Lucas Theorem or calculating factorials and their modular inverses using Fermatâ€™sÂ LittleÂ Theorem.
- Dynamic Programming (Pascal's Identity): For scenarios where many nCr values are needed, the relation nCr=(nâˆ’1)C(râˆ’1)+(nâˆ’1)Cr allows filling a table (Pascal's Triangle) in O(n ^2) time.
- Permutations (nPr): The problem of calculating nPr=n!/(nâˆ’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 â‰¤ 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 â†’ more verbose.
- Approach 2: Accumulative method â†’ elegant, in-place.

4. Variants / Follow-Ups

- Decimal â†’ Binary conversion
- Binary â†’ Hexadecimal conversion
- Large binary strings beyond integer limit â†’ 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 â‰¥ 0
- Decimal number â‰¤ 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 â†' easy to understand.
- Approach 2: Bitwise method â†' faster in low-level operations.

4. Variants / Follow-Ups

- Binary â†' Decimal conversion
- Decimal â†' Hexadecimal conversion
- Decimal â†' Binary for negative numbers (2â€™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) â†' Each row has up to n letters
- Space: O(1) â†' 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 â†’ Less memory, direct computation
- String-based â†’ 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

Q99: Sum to N

---

1. Problem Understanding

- You need to find how many combinations of distinct digits (1â€“9) of size k have a sum equal to n.
- Each combination must use distinct numbers, and the order doesnâ€™t matter (i.e., {1,2,4} and {2,1,4} are the same).

---

2. Constraints

- 1 <= k <= 9
- 1 <= n <= 45
- Digits available = {1, 2, 3, 4, 5, 6, 7, 8, 9}

---

3. Edge Cases

- n < smallest possible sum â†’ return 0
- n > largest possible sum â†’ return 0
- If no valid combinations exist, return 0

---

4. Examples

```
Input:
9 3
Output:
3
Valid combinations:
{1, 2, 6}, {1, 3, 5}, {2, 3, 4}
```

---

5. Approaches

Approach 1: Recursive Backtracking

Idea:

- Use recursion to explore all combinations of numbers from 1â€“9.
- At each step, choose a number, reduce n by that number, and decrease k by 1.
- Stop when n == 0 and k == 0 (a valid combination found).

Steps:

- Define a helper function countCombinations(start, n, k)
  - start: current number to consider (ensures distinct + ascending order)
- If n == 0 and k == 0 â†' found a valid combination â†' return 1
- If n < 0 or k == 0 â†' invalid path â†' return 0
- Loop i from start to 9
  - Include i â†' call recursively with n - i, k - 1, i + 1
- Sum all valid recursive counts

Java Code:

```
public class Main {
    public static int sumOfN(int n, int k) {
        return helper(1, n, k);
    }

    private static int helper(int start, int n, int k) {
        // Base case: found valid combination
        if (n == 0 && k == 0) return 1;

        // Base case: invalid state
        if (n < 0 || k == 0) return 0;

        int count = 0;
        for (int i = start; i <= 9; i++) {
            count += helper(i + 1, n - i, k - 1);
        }
        return count;
    }

    public static void main(String[] args) {
        java.util.Scanner sc = new java.util.Scanner(System.in);
        int n = sc.nextInt();
        int k = sc.nextInt();
        System.out.println(sumOfN(n, k));
    }
}

ðŸŒ³ Recursion Tree (Example: n = 7, k = 3)
helper(1,7,3)
 â”œâ”€ i=1 â†' helper(2,6,2)
```

```
│   ├─ i=2 → helper(3,4,1)
│   │   ├─ i=3 → helper(4,1,0) ❌
│   │   ├─ i=4 → helper(5,0,0) ✅ (found 1)
│   │   └─ i>4 → exceeds 9 ❌
│   ├─ i>2 → other branches ❌
└─ i>1 → other branches ❌


✅ Found {1,2,4} → total = 1
```

Complexity (Time & Space):

- ⏱️ Time Complexity
  - Roughly O(2⁹) → since we explore include/exclude for 9 digits
  - Practically much less due to pruning (n < 0, k == 0)
- 💾 Space Complexity
  - O(k) → recursion depth (stack space)

---

6. Justification / Proof of Optimality

- This approach ensures:
- Distinct numbers (due to start parameter)
- No duplicates (combination order ignored)
- Checks all possible valid subsets efficiently

---

7. Variants / Follow-Ups

- Find actual combinations (store in list instead of counting)
- Allow repeated numbers (remove distinctness constraint)
- Use digits from 1–n instead of 1–9

---

8. Tips & Observations

- Use backtracking for combination-type problems
- Always control distinctness via the start parameter
- Recursion naturally handles combination depth (k here)
