Q0: Recursion on Integer

---

1. Problem Understanding

- Recursion on integers means performing recursive operations where the changing variable is an integer — typically n, representing size, count, or numerical value.
- Each recursive call usually reduces or divides n until reaching a base case (e.g., n == 0 or n == 1).

---

2. Constraints

- Must have a base case to prevent infinite recursion.
- Integer parameter should change in every recursive call.
- Recursion depth ≤ value of n (or log(n) if divided each time).

---

3. Edge Cases

- n = 0 or n = 1 (most base cases).
- Negative integers (usually invalid unless explicitly handled).
- Large n may cause StackOverflowError (too deep recursion).

---

4. Examples

```
Print numbers 1 → n

Print numbers n → 1

Sum of first n numbers

Factorial n!

Reverse digits

Count digits

Fibonacci numbers

GCD of two numbers
```

---

5. Approaches

Approach 1: Printing / Counting Numbers

Java Code:

```
a) 1 → n
void print1ToN(int n) {
    if (n == 0) return;
    print1ToN(n - 1);
    System.out.print(n + " ");
}


b) n → 1
void printNTo1(int n) {
    if (n == 0) return;
    System.out.print(n + " ");
    printNTo1(n - 1);
}


c) Count numbers in a range
int countRange(int start, int end) {
    if (start > end) return 0;
    return 1 + countRange(start + 1, end);
}
```

Complexity (Time & Space):

- a) print1ToN(int n)
  - ⏱ Time: O(n)
  - 💾 Space: O(n)
- b) printNTo1(int n)
  - ⏱ Time: O(n)
  - 💾 Space: O(n)
- c) countRange(int start, int end)
  - ⏱ Time: O(end − start + 1) ≈ O(n)
  - 💾 Space: O(n)

Approach 2: Sum / Product

Java Code:

```
a) Sum of first n numbers
int sumN(int n) {
    if (n == 0) return 0;
    return n + sumN(n - 1);
}


b) Sum of digits
int sumDigits(int n) {
```

```
    if (n == 0) return 0;
    return (n % 10) + sumDigits(n / 10);
}


c) Product of digits
int productDigits(int n) {
    if (n == 0) return 1;
    return (n % 10) * productDigits(n / 10);
}


d) Sum of squares
int sumSquares(int n) {
    if (n == 0) return 0;
    return n * n + sumSquares(n - 1);
}
```

Complexity (Time & Space):

- a) sumN(int n)
  - ⏱ Time: O(n)
  - 💾 Space: O(n)
- b) sumDigits(int n)
  - ⏱ Time: $O(\log_{10} n)$ → proportional to number of digits
  - 💾 Space: $O(\log_{10} n)$
- c) productDigits(int n)
  - ⏱ Time: $O(\log_{10} n)$
  - 💾 Space: $O(\log_{10} n)$
- d) sumSquares(int n)
  - ⏱ Time: O(n)
    - 💾 Space: O(n)

Approach 3: Factorial / Power

Java Code:

```
a) Factorial
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}


b) Power
long power(long x, long n) {
    if (n == 0) return 1;
    return x * power(x, n - 1);
}
```

```
c) Power Optimized (Exponentiation by Squaring)
long power(long x, long n) {
    if (n == 0) return 1;
    long half = power(x, n / 2);
    if (n % 2 == 0) return half * half;
    else return x * half * half;
}


d) Power with modulo
long powerMod(long x, long n, long m) {
    if (n == 0) return 1;
    long half = powerMod(x, n / 2, m);
    long res = (half * half) % m;
    if (n % 2 != 0) res = (res * x) % m;
    return res;
}
```

Complexity (Time & Space):

- a) factorial(int n)
- ⏱ Time: O(n)
- 💾 Space: O(n)
- b) power(long x, long n)
    - ⏱ Time: O(n)
    - 💾 Space: O(n)
- c) powerOptimized(long x, long n) (Exponentiation by Squaring)
    - ⏱ Time: O(log n)
    - 💾 Space: O(log n)
- d) powerMod(long x, long n, long m)
    - ⏱ Time: O(log n)
    - 💾 Space: O(log n)

Approach 4: Reverse / Digit Operations

Java Code:

```
Reverse digits
int rev(int n, int ans) {
    if (n == 0) return ans;
    return rev(n / 10, ans * 10 + n % 10);
}
// call: rev(1234, 0)


b) Count digits
int countDigits(int n) {
    if (n == 0) return 0;
    return 1 + countDigits(n / 10);
```

```
}


c) Count digits with property (even/odd)
int countEvenDigits(int n) {
    if (n == 0) return 0;
    int count = (n % 10) % 2 == 0 ? 1 : 0;
    return count + countEvenDigits(n / 10);
}


d) Check palindrome
// Reverse number using recursion
int rev(int n, int ans) {
    if (n == 0) return ans;
    return rev(n / 10, ans * 10 + n % 10);
}

// Check palindrome
boolean isPalindrome(int n) {
    return n == rev(n, 0);
}
```

Complexity (Time & Space):

- ⏱ Time: $O(\log_{10} n)$
- 💾 Space: $O(\log_{10} n)$

Approach 5: Fibonacci / Sequence

Java Code:

```
Fibonacci number
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}


b) Print Fibonacci sequence
void printFib(int a, int b, int n) {
    if (n == 0) return;
    int c = a + b;
    System.out.print(c + " ");
    printFib(b, c, n - 1);
}
```

Complexity (Time & Space):

- a) fib(int n)
    - ⏱ Time: $O(2^n)$
    - 💾 Space: O(n)
- b) printFib(int a, int b, int n)
    - ⏱ Time: O(n)
    - 💾 Space: O(n)

Approach 6: GCD / LCM

Java Code:

```
a) GCD (Euclid's Algorithm)
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}


b) LCM using GCD
int lcm(int a, int b) {
    return (a * b) / gcd(a, b);
}
```

Complexity (Time & Space):

- ⏱ Time: O(log(min(a, b)))
- 💾 Space: O(log n)

Approach 7: Integer Backtracking / Combinatorics

Java Code:

```
a) Generate all subsequences of a number
void subsequences(int n, int curr) {
    if (n == 0) {
        System.out.println(curr);
        return;
    }
    // include last digit
    subsequences(n / 10, curr * 10 + n % 10);
    // exclude last digit
    subsequences(n / 10, curr);
}


b) Generate all k-digit numbers
void printKDigits(int n, int curr) {
    if (n == 0) {
        System.out.println(curr);
```

```
            return;
        }
        for (int i = 0; i <= 9; i++) {
            printKDigits(n - 1, curr * 10 + i);
        }
    }
```

Complexity (Time & Space):

- a) subsequences(int n, int curr)
  - ⏱ Time: $O(2^d)$ (where d = number of digits)
  - 💾 Space: O(d)
- b) printKDigits(int n, int curr)
  - ⏱ Time: $O(10^n)$ (each level has 10 branches)
  - 💾 Space: O(n)

Approach 8: Special Patterns

Java Code:

```
a) Reverse printing range
void reverseRange(int start, int end) {
    if (start > end) return;
    reverseRange(start + 1, end);
    System.out.print(start + " ");
}


b) Odd / Even numbers in range
void printEven(int n) {
    if (n == 0) return;
    printEven(n - 1);
    if (n % 2 == 0) System.out.print(n + " ");
}
```

Complexity (Time & Space):

- a) reverseRange(int start, int end)
  - ⏱ Time: O(end − start + 1) ≈ O(n)
  - 💾 Space: O(n)
- b) printEven(int n)
  - ⏱ Time: O(n)
  - 💾 Space: O(n)

---

6. Justification / Proof of Optimality

- Each call reduces the integer, approaching base case.
- Can be linear recursion (O(n)) or branching recursion (subsequence, combinatorial).

- Extra parameters are useful to accumulate results (sum, reversed number, current number).
- Forms the foundation for arrays, strings, trees, and DP recursion problems.

---

7. Variants / Follow-Ups

- Sum/product of odd/even digits
- Count numbers with specific digit
- Generate all numbers in a range satisfying a property
- Fibonacci using memoization for efficiency
- Factorial / Power using iterative + recursive combination

---

8. Tips & Observations

- Base case is critical. Always define clearly.
- For ascending order, recurse first, then process.
- For descending order, process first, then recurse.
- For branching recursion, number of calls = (branches)^depth.
- Visualize stack frames for debugging.

Q69: Smallest Number in an Array using Recursion

---

1. Problem Understanding

- You are given an array arr of size n.
- You need to find the minimum element in the array using recursion (no loops).
- The recursion should process one element at a time and eventually return the smallest value.

---

2. Constraints

- 1 <= n <= 10^3
- -10^4 <= arr[i] <= 10^4
- Time limit allows simple O(n) recursive solution.

---

3. Edge Cases

- Array has only one element → return that element.
- All elements are same → return that element.
- Array contains negative numbers → recursion should handle comparisons properly.

---

4. Examples

```
Input
5
5 4 0 -8 67
Output
-8
Explanation
→ Recursive calls compare elements one by one until the smallest (-8) is found.
```

---

5. Approaches

Approach 1: Linear Recursive Comparison (Left to Right)

Idea:

- Compare the first element with the minimum of the rest of the array.
- Base case: when array size = 1 → return that element.

Steps:

- Define a recursive function minElement(arr, n).
- Base case → if n == 1, return arr[0].
- Recursive call → minElement(arr, n-1) to get the min of first (n-1) elements.
- Compare last element with recursive result.
- Return the smaller one.

Java Code:

```
int minElement(int[] arr, int n) {
    if (n == 1) return arr[0];
    int small = minElement(arr, n - 1);
    return Math.min(arr[n - 1], small);
}
minElement([5, 2, 4, 1], 4)
        |
        🔁 Recursive call → minElement([5, 2, 4], 3)
                |
                🔁 Recursive call → minElement([5, 2], 2)
                        |
                        🔁 Recursive call → minElement([5], 1)
                                Base case → return 5
                        🔙 Return → min(2, 5) = 2
                🔙 Return → min(4, 2) = 2
        🔙 Return → min(1, 2) = 1
✅ Final Answer → 1
```

Complexity (Time & Space):

- Time Complexity
    - O(n) → one call per element.
- Space Complexity
    - O(n) → recursion stack.

Approach 2: Index-Based Recursion

Idea:

- Pass the index i as a changing variable.
- Move forward from 0 to n-1, comparing elements along the way.

Steps:

- Base case → if i == n - 1, return arr[i].
- Recursive call to get min of rest: minElement(arr, i + 1, n).
- Compare current element arr[i] with result of recursion.

Java Code:

```java
int minElement(int[] arr, int i, int n) {
    if (i == n - 1) return arr[i];
    int minRest = minElement(arr, i + 1, n);
    return Math.min(arr[i], minRest);
}
```

Complexity (Time & Space):

- Time Complexity
    - O(n)
- Space Complexity
    - O(n) (recursion depth)

---

6. Justification / Proof of Optimality

- Each recursive call reduces the problem size by 1.
- Base case ensures termination at the smallest subproblem (single element).
- Works correctly for both positive and negative numbers.

---

7. Variants / Follow-Ups

- Find Maximum element (replace Math.min with Math.max)
- Find both Min and Max recursively (return pair or use helper class)
- Find Minimum index recursively (return index instead of value)

---

8. Tips & Observations

- Recursion helps visualize problems as smaller subarrays.
- Always identify a shrinking condition (n-1 or i+1).
- Use Math.min() and Math.max() to avoid manual if-else checks.
- Dry run on small arrays to ensure indices are correct.
