Q0: Recursion on Arrays

1. Problem Understanding

- Recursion on arrays means processing one element at a time using function calls.
- Each recursive call works on a smaller part of the array.
- Useful for operations like:
    - Traversal
    - Sum/Product
    - Searching
    - Sorting (Merge/Quick)
    - Backtracking (Subsets/Permutations)

---

2. Constraints

- Array size = n
- Usually index-based recursion (i, l, r)
- Base case should ensure termination (i == n, l >= r)
- Stack depth equals recursion depth (typically O(n))

---

3. Edge Cases

- Empty array (n = 0)
- Single element array
- Duplicate elements (for searching or subsets)
- Already sorted or completely reversed arrays

---

4. Examples

```
Input: [1, 2, 3] → Output: 1 2 3

Input: [2, 4, 6] → Sum = 12

Input: [1, 2, 3] → Reverse = [3, 2, 1]

Input: [1, 2] → Subsets = [], [1], [2], [1, 2]
```

---

5. Approaches

Approach 1: Traversal (Forward / Backward)

Java Code:

```
Forward Traversal

void printArray(int[] arr, int i) {
    if (i == arr.length) return;
    System.out.print(arr[i] + " ");
    printArray(arr, i + 1);
}


Backward Traversal

void printReverse(int[] arr, int i) {
    if (i == arr.length) return;
    printReverse(arr, i + 1);
    System.out.print(arr[i] + " ");
}
```

Complexity (Time & Space):

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

Approach 2: Sum / Product of Array

Java Code:

```
Sum

int sumArray(int[] arr, int i) {
    if (i == arr.length) return 0;
    return arr[i] + sumArray(arr, i + 1);
}


Product

int productArray(int[] arr, int i) {
    if (i == arr.length) return 1;
    return arr[i] * productArray(arr, i + 1);
}
```

Complexity (Time & Space):

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

Approach 3: Search in Array

Java Code:

```
Linear Search

boolean search(int[] arr, int i, int key) {
    if (i == arr.length) return false;
    if (arr[i] == key) return true;
    return search(arr, i + 1, key);
}
```

Complexity (Time & Space):

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

Approach 4: Last Occurrence

Idea:

- Go till the end and check while returning.

Java Code:

```
int lastIndex(int[] arr, int i, int key) {
    if (i == arr.length) return -1;
    int res = lastIndex(arr, i + 1, key);
    if (res == -1 && arr[i] == key) return i;
    return res;
}
```

Complexity (Time & Space):

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

Approach 5: Find Maximum / Minimum Element

Java Code:

```
Maximum

int maxElement(int[] arr, int i) {
    if (i == arr.length - 1) return arr[i];
    return Math.max(arr[i], maxElement(arr, i + 1));
}


Minimum
```

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

Complexity (Time & Space):

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

Approach 6: Reverse Array (In-Place)

Idea:

- Swap from both ends and move inward.

Java Code:

```
void reverse(int[] arr, int l, int r) {
    if (l >= r) return;
    int temp = arr[l];
    arr[l] = arr[r];
    arr[r] = temp;
    reverse(arr, l + 1, r - 1);
}
```

Complexity (Time & Space):

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

Approach 7: Check if Array is Sorted

Idea:

- Compare adjacent elements recursively.

Java Code:

```
boolean isSorted(int[] arr, int i) {
    if (i == arr.length - 1) return true;
    if (arr[i] > arr[i + 1]) return false;
    return isSorted(arr, i + 1);
}
```

Complexity (Time & Space):

- Time: O(n)

- Space: O(n)

Approach 8: Binary Search (Divide & Conquer)

Idea:

- Recursively search halves.

Java Code:

```
int binarySearch(int[] arr, int l, int r, int key) {
    if (l > r) return -1;
    int mid = (l + r) / 2;
    if (arr[mid] == key) return mid;
    else if (key < arr[mid]) return binarySearch(arr, l, mid - 1, key);
    else return binarySearch(arr, mid + 1, r, key);
}
```

Complexity (Time & Space):

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

Approach 9: Merge Sort

Idea:

- Divide array, sort halves, and merge.

Java Code:

```
void mergeSort(int[] arr, int l, int r) {
    if (l >= r) return;
    int mid = (l + r) / 2;
    mergeSort(arr, l, mid);
    mergeSort(arr, mid + 1, r);
    merge(arr, l, mid, r);
}
```

Complexity (Time & Space):

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

Approach 10: Quick Sort

Idea:

- Partition around pivot and sort halves.

Java Code:

```
void quickSort(int[] arr, int l, int r) {
    if (l >= r) return;
    int p = partition(arr, l, r);
    quickSort(arr, l, p - 1);
    quickSort(arr, p + 1, r);
}
```

Complexity (Time & Space):

- Time: O(n log n) average, $O(n^2)$ worst
- Space: O(log n) average, O(n) worst

Approach 11: Subsets (Backtracking)

Idea:

- For each element → include or exclude.

Java Code:

```
void subsets(int[] arr, List<Integer> curr, int i) {
    if (i == arr.length) {
        System.out.println(curr);
        return;
    }
    curr.add(arr[i]);
    subsets(arr, curr, i + 1);
    curr.remove(curr.size() - 1);
    subsets(arr, curr, i + 1);
}
```

Complexity (Time & Space):

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

Approach 12: Permutations (Backtracking)

Idea:

- Pick each element and mark used ones.

Java Code:

```
void permute(int[] arr, boolean[] used, List<Integer> curr) {
    if (curr.size() == arr.length) {
```

```
            System.out.println(curr);
            return;
        }
        for (int i = 0; i < arr.length; i++) {
            if (used[i]) continue;
            used[i] = true;
            curr.add(arr[i]);
            permute(arr, used, curr);
            used[i] = false;
            curr.remove(curr.size() - 1);
        }
    }
```

Complexity (Time & Space):

- Time: O(n × n!)
- Space: O(n)

Approach 13: Subset Sum

Idea:

- Either include or exclude element in sum.

Java Code:

```
void subsetSum(int[] arr, int i, int sum) {
    if (i == arr.length) {
        System.out.print(sum + " ");
        return;
    }
    subsetSum(arr, i + 1, sum + arr[i]);
    subsetSum(arr, i + 1, sum);
}
```

Complexity (Time & Space):

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

---

6. Justification / Proof of Optimality

- Recursion replaces loops for clarity in logic.
- Helps visualize array problems as smaller subproblems.
- Foundation for divide & conquer and backtracking algorithms.

---

7. Variants / Follow-Ups

- Recursion on lists or strings (same logic)
- 2D array recursion (matrix problems)
- Recursion with memoization for optimization

---

8. Tips & Observations

- Always define a clear base case (i == n or l >= r).
- Visualize using recursion tree or stack frames.
- Avoid extra space unless necessary.
- Restore array state after backtracking.
- Recursive depth generally equals array size.
