Q0: Recursion on Arraylist

1. Problem Understanding

- Recursion on ArrayList involves processing elements by index or removing elements recursively.
- Useful for:
- Traversal
- Sum / Product
- Searching
- Maximum / Minimum
- Backtracking (subsets / permutations)

---

2. Constraints

- ArrayList size = n
- Base case: index i == list.size()
- Avoid concurrent modification when modifying list during recursion
- Stack depth = O(n)

---

3. Edge Cases

- Empty list
- Single element list
- Duplicate elements
- Modifications during recursion (add/remove) must be carefully handled

---

4. Examples

```
Input: [1, 2, 3] → Output: 1 2 3

Input: [1, 2, 3] → Sum = 6

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

---

5. Approaches

Approach 1: Traversal (Forward / Backward)

Java Code:

```
Forward Traversal

void traverse(ArrayList<Integer> list, int i) {
    if (i == list.size()) return;
    System.out.print(list.get(i) + " ");
    traverse(list, i + 1);
}


Backward Traversal

void traverseReverse(ArrayList<Integer> list, int i) {
    if (i == list.size()) return;
    traverseReverse(list, i + 1);
    System.out.print(list.get(i) + " ");
}
```

Complexity (Time & Space):

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

Approach 2: Sum / Product

Java Code:

```
Sum

int sumList(ArrayList<Integer> list, int i) {
    if (i == list.size()) return 0;
    return list.get(i) + sumList(list, i + 1);
}


Product

int productList(ArrayList<Integer> list, int i) {
    if (i == list.size()) return 1;
    return list.get(i) * productList(list, i + 1);
}
```

Complexity (Time & Space):

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

Approach 3: Search Element

Java Code:

```
Linear Search

boolean searchList(ArrayList<Integer> list, int i, int key) {
    if (i == list.size()) return false;
    if (list.get(i) == key) return true;
    return searchList(list, i + 1, key);
}
```

Complexity (Time & Space):

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

Approach 4: Find Maximum / Minimum

Java Code:

```
Maximum

int maxList(ArrayList<Integer> list, int i) {
    if (i == list.size() - 1) return list.get(i);
    return Math.max(list.get(i), maxList(list, i + 1));
}


Minimum

int minList(ArrayList<Integer> list, int i) {
    if (i == list.size() - 1) return list.get(i);
    return Math.min(list.get(i), minList(list, i + 1));
}
```

Complexity (Time & Space):

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

Approach 5: Remove Element Recursively

Idea:

- Skip or remove elements while traversing.

Java Code:

```
void removeElement(ArrayList<Integer> list, int i, int key) {
    if (i == list.size()) return;
    if (list.get(i) == key) {
```

```java
            list.remove(i);
            removeElement(list, i, key); // do not increment index
        } else {
            removeElement(list, i + 1, key);
        }
    }
```

Complexity (Time & Space):

- Time: $O(n^2)$ in worst case (due to remove shift)
- Space: O(n)

Approach 6: Subsets (Backtracking)

Idea:

- Include/exclude elements recursively.

Java Code:

```java
void subsets(ArrayList<Integer> list, ArrayList<Integer> curr, int i) {
    if (i == list.size()) {
        System.out.println(curr);
        return;
    }
    curr.add(list.get(i));
    subsets(list, curr, i + 1);   // include
    curr.remove(curr.size() - 1);
    subsets(list, curr, i + 1);   // exclude
}
```

Complexity (Time & Space):

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

Approach 7: Permutations (Backtracking)

Idea:

- Swap elements to generate all permutations.

Java Code:

```java
void permute(ArrayList<Integer> list, int i) {
    if (i == list.size()) {
        System.out.println(list);
        return;
    }
    for (int j = i; j < list.size(); j++) {
```

```java
            Collections.swap(list, i, j);
            permute(list, i + 1);
            Collections.swap(list, i, j); // backtrack
        }
    }
```

Complexity (Time & Space):

- Time: O(n × n!)
- Space: O(n)

Approach 8: Count / Sum with Condition

Idea:

- Example: Sum of even elements

Java Code:

```java
int sumEven(ArrayList<Integer> list, int i) {
    if (i == list.size()) return 0;
    int sum = (list.get(i) % 2 == 0) ? list.get(i) : 0;
    return sum + sumEven(list, i + 1);
}
```

Complexity (Time & Space):

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

Approach 9: Reverse ArrayList Recursively

Java Code:

```java
void reverseList(ArrayList<Integer> list, int l, int r) {
    if (l >= r) return;
    Collections.swap(list, l, r);
    reverseList(list, l + 1, r - 1);
}
```

Complexity (Time & Space):

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

Approach 10: Remove Duplicates Recursively

Java Code:

```java
void removeDuplicates(ArrayList<Integer> list, int i) {
    if (i >= list.size() - 1) return;
    if (list.get(i).equals(list.get(i + 1))) {
        list.remove(i + 1);
        removeDuplicates(list, i);
    } else {
        removeDuplicates(list, i + 1);
    }
}
```

Complexity (Time & Space):

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

---

6. Justification / Proof of Optimality

- ArrayList recursion is almost identical to array recursion.
- Key difference: methods like .get(), .add(), .remove() introduce extra cost.
- Useful for backtracking and combinatorial problems.

---

7. Variants / Follow-Ups

- Recursion on ArrayList of Strings
- 2D ArrayList recursion (list of lists)
- Nested backtracking (subsets of subsets)

---

8. Tips & Observations

- Always use index-based recursion to avoid ConcurrentModificationException.
- Recursive depth = size of list = O(n)
- Avoid modifying list during iteration without careful indexing.
- Backtracking requires restore state after recursion (remove / swap).

Q83: Get Subsequences

---

1. Problem Understanding

- Find all subsequences of a given string.
- A subsequence is formed by deleting some characters without changing the order.
- Return all subsequences in lexicographical order.
- Ignore empty string.

---

2. Constraints

- 1 <= s.length <= 100

---

3. Edge Cases

- Single character string → only 1 subsequence (itself).
- Repeated characters → subsequences may repeat; use a Set to ensure uniqueness.

---

4. Examples

```
Input: "abc"

Output:

a
ab
abc
ac
b
bc
c


Explanation: All non-empty subsequences in sorted order.
```

---

5. Approaches

Approach 1: Recursion

Idea:

- For each character, choose to include or exclude it recursively.
- Base case: empty string → return list containing empty string (later filter it out).
- Combine all subsequences from including and excluding current character.
- Sort the final list lexicographically.

Steps:

- If string empty → return list containing "".
- Take first character ch.
- Recurse on rest of string → get list restSubs.
- For each string in restSubs:
  - Add it to result (exclude ch).
  - Add ch + string to result (include ch).
- Remove empty string and sort the list.

- Return the list.

Java Code:

```
static List<String> generateSubsequence(String s) {
    if (s.length() == 0) {
        List<String> base = new ArrayList<>();
        base.add("");
        return base;
    }

    char ch = s.charAt(0);
    List<String> restSubs = generateSubsequence(s.substring(1));
    List<String> result = new ArrayList<>();

    for (String str : restSubs) {
        result.add(str);           // exclude current character
        result.add(ch + str);      // include current character
    }

    result.remove(""); // remove empty string
    Collections.sort(result); // lexicographical order
    return result;
}
generateSubsequence("ab")
           /       \
exclude 'a'          include 'a'
      "b"                  "a" + "b"
     /   \                 /   \
exclude 'b' include 'b'  exclude 'b' include 'b'
  ""       "b"               "a"      "ab"
Result before removing empty: ["", "b", "a", "ab"]

After removing empty and sorting: ["a", "ab", "b"]

another way

public static ArrayList<String> generateSubsequences(String str) {
    ArrayList<String> l = new ArrayList<>();
    helper(str, 0, "", l);
    Collections.sort(l);
    return l;
}

public static void helper(String str, int i, String ans, ArrayList<String> l) {
    if (i >= str.length()) {
        if (ans.length() > 0) {
            l.add(ans);
        }
        return;
    }
    helper(str, i + 1, ans + str.charAt(i), l);
```

```
        helper(str, i + 1, ans, l);
    }
```

Complexity (Time & Space):

- Time: O(2^n * n) → 2^n subsequences, each of length up to n for sorting
- Space: O(2^n * n) → recursion stack + storing subsequences

Approach 2: Iterative using Bitmasking

Idea:

- A string of length n has 2^n subsequences.
- Each subsequence corresponds to a bitmask of length n:
- 1 → include the character
- 0 → exclude the character
- Loop through all bitmasks from 1 to 2^n - 1 (skip 0 to ignore empty string).
- For each bitmask, build the subsequence by including characters where bit = 1.

Steps:

- Let n = s.length().
- Loop mask from 1 to (1 << n) - 1:
- Initialize empty string subseq.
- For each bit i from 0 to n-1:
- If (mask & (1 << i)) != 0 → include s.charAt(i) in subseq.
- Add subseq to list of subsequences.
- Sort the list lexicographically.
- Return the list.

Java Code:

```java
static List<String> generateSubsequenceBitmask(String s) {
    int n = s.length();
    List<String> result = new ArrayList<>();

    for (int mask = 1; mask < (1 << n); mask++) { // skip 0 to avoid empty string
        StringBuilder subseq = new StringBuilder();
        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) != 0) {
                subseq.append(s.charAt(i));
            }
        }
        result.add(subseq.toString());
    }

    Collections.sort(result); // lexicographical order
    return result;
}
```

Complexity (Time & Space):

- Time: O(2^n * n) → 2^n masks, each can take up to n operations to build string
- Space: O(2^n * n) → storing subsequences

---

6. Justification / Proof of Optimality

- Approach 2
  - Correct because it explores all inclusion/exclusion choices for every character.
  - Ensures all subsequences are generated.
  - Lexicographical order achieved by sorting at the end.
- Approach 2
  - Generates all possible subsequences by representing choices using bits.
  - Excludes empty string by starting mask from 1.
  - Sorting ensures lexicographical order.

---

7. Variants / Follow-Ups

- Return subsequences as Set to remove duplicates.
- Return only subsequences of length k.
- Generate subsequences iteratively using bitmasking.

---

8. Tips & Observations

- Approach 1
  - Maximum number of subsequences for string of length n = 2^n.
  - Use recursion template:
    - Include first character + recurse on rest
    - Exclude first character + recurse on rest
  - Removing empty string ensures only valid subsequences are returned.
  - Sorting can be done using Collections.sort() for lexicographic order.
- Approach 2
  - Useful for iterative solution when recursion depth might be an issue.
  - Works best for n <= 20, since 2^n grows quickly.
  - Can combine with Set to avoid duplicates if the string has repeated characters.

Q84: Old Phone Keypad

---

1. Problem Understanding

- Given a sequence of key presses on an old phone keypad, generate all possible letter combinations.
- Keys map to multiple letters:
  - 1 -> ABC

    - 2 -> DEF
    - 3 -> GHI
    - 4 -> JKL
    - 5 -> MNO
    - 6 -> PQRS
    - 7 -> TU
    - 8 -> VWX
    - 9 -> YZ
    - 0, *, # -> ignored
- Output must be lexicographically sorted.
- The order of key presses must be maintained.

---

2. Constraints

- 1 ≤ n ≤ 10 → number of key presses
- 1 ≤ key[i] ≤ 9 → keys pressed
- Output must contain uppercase letters only

---

3. Edge Cases

- Single key press → just return letters of that key
- Repeated key → allow repetition in combinations
- Keys 0, *, # → ignored (no letters assigned)

---

4. Examples

```
Input:

2
2 5


Output:

DM DN DO EM EN EO FM FN FO
```

---

5. Approaches

Approach 1: Recursion (Backtracking)

Idea:

- For each key, iterate over its letters and recursively generate combinations for the remaining keys.

Steps:

- Map each key to its corresponding letters in an array.
- Start recursion with index = 0 and empty string ans.
- For each character corresponding to keys[index]:
    - Append to ans
    - Recurse for index + 1
- Base case: if index == n → add ans to result list

Java Code:

```
public static ArrayList<String> keypadCombinations(int[] keys) {
    String[] map = {"", "ABC", "DEF", "GHI", "JKL", "MNO", "PQRS", "TU", "VWX",
"YZ"};
    ArrayList<String> res = new ArrayList<>();
    helper(keys, 0, "", map, res);
    Collections.sort(res);
    return res;
}

static void helper(int[] keys, int index, String ans, String[] map,
ArrayList<String> res) {
    if (index == keys.length) {
        res.add(ans);
        return;
    }
    String letters = map[keys[index]];
    for (char ch : letters.toCharArray()) {
        helper(keys, index + 1, ans + ch, map, res);
    }
}
Recursion Tree for keys = [2,5] (2 -> DEF, 5 -> MNO):

("",0)
   /   |   \
"D",1   "E",1   "F",1
 /|\     /|\     /|\
"M N O" "M N O" "M N O"


Produces: DM, DN, DO, EM, EN, EO, FM, FN, FO
```

Complexity (Time & Space):

- Time: O(letters_per_key ^ n) → For worst-case key 6 → 4 letters, max n = 10 → O(4^10)
- Space: O(n) recursion depth + O(total combinations) storage

---

6. Justification / Proof of Optimality

- Generates all possible combinations by recursion maintaining order of key presses.
- Sorted after generation to ensure lexicographical order.

7. Variants / Follow-Ups

- Can print directly instead of storing if memory is tight.
- Can adapt to lowercase letters easily by changing mapping.

8. Tips & Observations

- Each key press branches into multiple letters → classic recursion tree problem
- Lexicographic order is ensured by sorting at the end or by storing letters in order in mapping
- Works best for small number of key presses; for large n, iterative combinatorial generation may be