Absolutely. Let's proceed with a structured explanation and breakdown of all the provided code snippets using the **"Processed and Unprocessed" recursion pattern**.

🔁 Understanding the "Processed and Unprocessed" Pattern

This pattern is widely used in recursion when solving problems like:

- Subsets
- Subsequences
- Skipping characters
- Generating permutations
- Keypad combinations

🔍 Core Idea:

We divide the input into two parts:

- **Processed (p)**: What we've built so far in the recursive path.
- **Unprocessed (up)**: What we are yet to handle or consume.

At each recursive call:

1. We choose whether or how to include the next character (or element) from the unprocessed string/array into the processed.
2. We then move forward with recursive calls on the updated processed and the reduced unprocessed.

🧠 Why is this pattern powerful?

- It encourages branching logic: Every decision leads to different recursive paths.
- It helps visualize recursion trees.
- It's modular: Any decision made at a level is isolated from others, ideal for backtracking-based problems.

📁 SubSet.java

🔹 subSet — Generates **all subsets** (including duplicates)

```
public static List<List<Integer>> subSet(int [] arr) {

    List<List<Integer>> outer = new ArrayList<>();

    outer.add(new ArrayList<>()); // start with empty subset
```

```
    for(int num : arr) {
        int n = outer.size();
        for(int i = 0 ; i < n; i++) {
            ArrayList<Integer> internal = new ArrayList<>(outer.get(i)); // copy current subset
            internal.add(num); // include current number
            outer.add(internal); // add to result
        }
    }

    return outer;
}
```

📊 **Time Complexity**:

- Each element can be included or excluded ⇒ 2^n subsets
- O(n × 2^n) (because copying list takes O(n) time)

📊 **Space Complexity**:

- O(n × 2^n) (for storing all subsets)

🔎 **Example**:

arr = [1, 2]

Start: [[]]

Add 1: [], [1]

Add 2: [], [1], [2], [1,2]

🔹 subSet2 — **Avoids duplicates** when input contains duplicate elements

```
public static List<List<Integer>> subSet2(int [] arr) {
    List<List<Integer>> outer = new ArrayList<>();
    outer.add(new ArrayList<>());
    int start = 0, end = 0;

    for(int i = 0; i < arr.length; i++) {
```

```
        start = 0;
        if(i > 0 && arr[i] == arr[i-1]) {
            start = end + 1; // only extend the subsets created in the last round
        }
        end = outer.size() - 1;

        int n = outer.size();
        for(int j = start; j < n; j++) {
            ArrayList<Integer> internal = new ArrayList<>(outer.get(j));
            internal.add(arr[i]);
            outer.add(internal);
        }
    }

    return outer;
}
```

📊 **Time Complexity**: O(n × 2^n)
📊 **Space Complexity**: O(n × 2^n)

🔎 **Example**:

Input: [1,2,2]

Result: [[], [1], [2], [1,2], [2,2], [1,2,2]]

📁 SubSeq.java

🔹 subSeq — Print all **subsequences** of a string

```
public static void subSeq(String p , String up) {
    if(up.isEmpty()) {
        System.out.println(p); // base case
        return;
```

```
    }


    char ch = up.charAt(0);

    subSeq(p + ch, up.substring(1)); // include current char

    subSeq(p, up.substring(1));      // exclude current char

}
```

📊 **Time Complexity**: O(2^n)
📊 **Space Complexity**: O(n) (stack depth)

🔎 **Example**:

Input: "ab"

Output: ab, a, b, ""

🔹 subSeq2 — Returns the list of subsequences

```
public static ArrayList<String> subSeq2(String p , String up, ArrayList<String> sArr) {

    if(up.isEmpty()) {

        ArrayList<String> list = new ArrayList<>();

        list.add(p);

        return list;

    }


    char ch = up.charAt(0);


    ArrayList<String> left = subSeq2(p + ch, up.substring(1), sArr);

    ArrayList<String> right = subSeq2(p, up.substring(1), sArr);


    left.addAll(right);

    return left;

}
```

📊 **Time Complexity**: O(2^n)
📊 **Space Complexity**: O(2^n × n)

📁 SkipString.java & Skip.java

🔹 Skip apple from a string

```
public static String skipString(String up) {
    if(up.length() == 0) return "";

    if(up.startsWith("apple")) {
        return skipString(up.substring(5));
    } else {
        return up.charAt(0) + skipString(up.substring(1));
    }
}
```

📊 **Time Complexity**: O(n)
📊 **Space Complexity**: O(n) (due to recursion)

🔹 Skip app but not apple

```
public static String skipString2(String up) {
    if(up.length() == 0) return "";

    if(up.startsWith("app") && !up.startsWith("apple")) {
        return skipString2(up.substring(3));
    } else {
        return up.charAt(0) + skipString2(up.substring(1));
    }
}
```

📊 **Time Complexity**: O(n)
📊 **Space Complexity**: O(n)

📁 Permutation.java

🔹 Print all **permutations** of a string

```
public static void permutation(String p , String up) {
    if(up.isEmpty()) {
        System.out.println(p);
        return;
    }

    char ch = up.charAt(0);
    for(int i = 0; i <= p.length(); i++) {
        String f = p.substring(0, i);
        String s = p.substring(i);
        permutation(f + ch + s, up.substring(1));
    }
}
```

📊 **Time Complexity**: O(n!)
📊 **Space Complexity**: O(n)

🔎 Example (Input = "abc"):

Output: abc, acb, bac, bca, cab, cba

🔹 Return permutations in a list

```
public static ArrayList<String> permutationArray(String p , String up) {
    if(up.isEmpty()) {
        ArrayList<String> list = new ArrayList<>();
        list.add(p);
        return list;
    }

    ArrayList<String> ans = new ArrayList<>();
```

```
    char ch = up.charAt(0);
    for(int i = 0; i <= p.length(); i++) {
        String f = p.substring(0, i);
        String s = p.substring(i);
        ans.addAll(permutationArray(f + ch + s, up.substring(1)));
    }
    return ans;
}
```

📊 **Time Complexity**: O(n!)
📊 **Space Complexity**: O(n! × n)

🔹 Count permutations

```
public static int permutationCount(String p , String up) {
    if(up.isEmpty()) return 1;

    int count = 0;
    char ch = up.charAt(0);
    for(int i = 0; i <= p.length(); i++) {
        String f = p.substring(0, i);
        String s = p.substring(i);
        count += permutationCount(f + ch + s, up.substring(1));
    }
    return count;
}
```

📊 **Time Complexity**: O(n!)
📊 **Space Complexity**: O(n)

📁 PhoneNumberCombination.java

🔹 Generate combinations like mobile keypad (1 -> abc, 2 -> def, etc.)

```
static ArrayList<String> combination(String p, String up) {
```

```
    if(up.isEmpty()) {
        ArrayList<String> list = new ArrayList<>();
        list.add(p);
        return list;
    }

    ArrayList<String> ans = new ArrayList<>();
    int n = Character.getNumericValue(up.charAt(0));
    int num = up.charAt(0) + '0'; // This is incorrect usage
    System.out.println(num);      // This should be removed in production

    for(int i = (n-1)*3; i < n*3; i++) {
        char ch = (char)('a' + i);
        ans.addAll(combination(p + ch, up.substring(1)));
    }

    return ans;
}
```

📌 ⚠ **Bug**: Keypad mapping logic is incorrect. '1' doesn't map to "abc". This logic should use a proper mapping array like:

String[] map = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};

📊 **Time Complexity**: O(3^n) or O(4^n) depending on digits
📊 **Space Complexity**: O(n × 3^n)

Absolutely. Let's continue in the exact format as before for each of your provided recursive and sorting algorithms, along with detailed breakdowns. We'll also address your second request at the end.

🔹 Merge Sort (Returning a New Array)

```
static int[] mergeSort(int[] arr) {
    if(arr.length == 1) {
        return arr;
    }

    int mid = arr.length / 2;

    int[] left = mergeSort(Arrays.copyOfRange(arr, 0, mid));
    int[] right = mergeSort(Arrays.copyOfRange(arr, mid, arr.length));

    return merge(left, right);
}

static int[] merge(int[] left, int[] right) {
    int[] mix = new int[left.length + right.length];
    int i = 0, j = 0, k = 0;

    while(i < left.length && j < right.length) {
        mix[k++] = (left[i] < right[j]) ? left[i++] : right[j++];
```

```
    }

    while(i < left.length) mix[k++] = left[i++];

    while(j < right.length) mix[k++] = right[j++];

    return mix;
}
```

📌 ⚠ Notes:

- No bugs, clean and classic functional approach.
- Doesn't sort in place; creates new arrays.

📊 Time Complexity: **O(n log n)**
📊 Space Complexity: **O(n)** due to array copying.

✅ **Use case:** When immutability is preferred, and memory isn’t a concern (e.g., functional programming environments).

🔹 Merge Sort (In-Place)

```
static void mergeSortInPlace(int[] arr, int s, int e) {
    if(e - s == 1) return;

    int mid = s + (e - s) / 2;
    mergeSortInPlace(arr, s, mid);
    mergeSortInPlace(arr, mid, e);

    mergeInPlace(arr, s, mid, e);
}

static void mergeInPlace(int[] arr, int s, int mid, int e) {
    int[] mix = new int[e - s];
    int i = s, j = mid, k = 0;
```

```
    while(i < mid && j < e) {
        mix[k++] = (arr[i] < arr[j]) ? arr[i++] : arr[j++];
    }

    while(i < mid) mix[k++] = arr[i++];
    while(j < e) mix[k++] = arr[j++];

    for(int l = 0; l < mix.length; l++) {
        arr[s + l] = mix[l];
    }
}
```

📌 ⚠ Notes:

- Efficient memory usage by working directly on the input array.
- Stable sort if merge logic is handled properly.

📊 Time Complexity: **O(n log n)**
📊 Space Complexity: **O(n)** for temporary mix[], but still better than copying subarrays.

✅ **Use case:** Useful for constrained memory environments and in-place requirements.

🔹 Quick Sort (In-Place using Middle Element as Pivot)

```
static void sort(int[] arr, int low, int high) {
    if(low >= high) return;

    int s = low, e = high;
    int mid = s + (e - s) / 2;
    int pivot = arr[mid];

    while(s <= e) {
        while(arr[s] < pivot) s++;
```

```
        while(arr[e] > pivot) e--;


        if(s <= e) {
            int temp = arr[s];
            arr[s++] = arr[e];
            arr[e--] = temp;
        }
    }


    sort(arr, low, e);
    sort(arr, s, high);
}
```

📌 ⚠ Notes:

- No bug. Pivot is chosen as the middle element for balance.
- Not stable; elements with equal value may be rearranged.

📊 Time Complexity:

- **Average:** O(n log n)
- **Worst Case:** O(n^2) (when array is already sorted)

📊 Space Complexity: **O(log n)** for recursive call stack.

✅ **Use case:** When speed is prioritized over stability and memory is limited.

🎲 Dice Combinations (Backtracking)

```
static void dice(String p, int target) {
    if(target == 0) {
        System.out.println(p);
        count++;
        return;
    }
```

```
    for(int i = 1; i <= target; i++) {
        dice(p + i, target - i);
    }
}
```

📌 ⚠ Notes:

- No bug. Backtracking solution that explores all dice combinations summing to the target.

📊 Time Complexity: **O(2^target)** (Upper bound due to recursive calls, varies depending on target)
📊 Space Complexity: **O(target)** for recursion stack

✅ **Use case:** To find combinations/paths in problems like stairs, dice throws, number partitions, etc.

🔍 2. Process and Unprocess Approach: Where and How to Use

Process and Unprocess is a conceptual pattern for recursive backtracking used when:

- We process part of a problem now and defer the rest (unprocessed).
- Frequently seen in **string permutations**, **subset generation**, and **combinations**.

🔑 Use Cases:

1. **Permutations and Combinations** (e.g., abc -> [abc, acb, bac, bca, cab, cba])
2. **Generating Keypad Combinations** (like mobile keys or T9 predictive text)
3. **Pathfinding in Grids or Mazes**
4. **Subsequence Problems** (print or count all subsequences)

📌 Approach Summary:

- p: processed string (part we've already handled)
- up: unprocessed string (what's left to explore)

✅ **When to use:**

- When you want to explore **all decision branches** (like subsets, permutations)
- When output depends on intermediate "processed" state
- To reduce dependencies on indices and instead build outputs directly