Recursion code snippets

Here are the current code snippets with beginner-friendly comments and concise explanations. You can copy-paste them directly into your notes:

🔍 1. **Linear Search using Recursion**

// Searches for the target value in the array using recursion

```
static int find(int[] arr, int target, int index) {
    if (index >= arr.length) {
        System.out.println("Not found");  // Base case: reached end of array without finding target
        return -1;
    }
    if (arr[index] == target) {
        return index;  // Target found at current index
    }
    return find(arr, target, index + 1);  // Recursive call: move to the next index
}
```

🧠 **How it works:**
This function checks each element from left to right. If the target is found, it returns the index. If it reaches the end, it returns -1, indicating the element is not found.

🔁 2. **Return All Occurrences of a Target in Array**

// Returns a list of all indices where the target is found in the array

```
static ArrayList<Integer> find(int[] arr, int target, int index, ArrayList<Integer> list) {
    if (index == arr.length) {
        return list;  // Base case: return collected indices
    }
    if (arr[index] == target) {
```

```
        list.add(index);  // Target found, add index to the list
    }
    return find(arr, target, index + 1, list);  // Recursive call with next index
}
```

🧠 **How it works:**
This function searches through the array recursively, adding indices where the target is found to a list, and returns the list after the entire array has been traversed.

✅ 3. **Check if Array is Sorted (Ascending Order)**

```
// Checks if the array is sorted in strictly increasing order
static boolean sorted(int[] arr, int index) {
    if (index == arr.length - 1) {
        return true;  // Base case: reached the end, array is sorted
    }
    return arr[index] < arr[index + 1] && sorted(arr, index + 1);  // Check current and recurse forward
}
```

Certainly! Here's a consolidated and beginner-friendly explanation of your three recursive Java programs, each broken down into understandable parts with proper inline comments and concise overviews.

📦 1. Return All Indices of Target Using Recursion – With External List Passed

```
// Finds all indices where the target element occurs in the array using recursion
static ArrayList<Integer> find(int[] arr, int target, int index, ArrayList<Integer> list) {
    if (index == arr.length) {
```

```
        return list; // Base case: reached end of array, return collected indices
    }
    if (arr[index] == target) {
        list.add(index); // If current element matches target, add index to list
    }
    return find(arr, target, index + 1, list); // Recur for the next element
}
```

🟦 **Overview:**
This function collects all the positions (indices) of a given number in the array. The list is passed in each recursive call and filled as matching elements are found.

📦 2. Return All Indices – Create List in Function

🔹 Version 1: Pre-Processing

```
// Pre-processing: Adds index to the list before recursive call
static ArrayList<Integer> find(int[] arr, int target, int index) {
    ArrayList<Integer> list = new ArrayList<>();

    if (index == arr.length) {
        return list; // Base case: return empty list if end is reached
    }

    if (arr[index] == target) {
        list.add(index); // Add current index if it matches the target
    }

    list.addAll(find(arr, target, index + 1)); // Add all matches from rest of the array
```

```
    return list;
}
```

🔹 **Version 2: Post-Processing**

```
// Post-processing: Adds index to the list after recursive call
static ArrayList<Integer> find2(int[] arr, int target, int index) {
    ArrayList<Integer> list = new ArrayList<>();

    if (index == arr.length) {
        return list; // Base case: return empty list
    }

    list = find2(arr, target, index + 1); // First get all matches from rest of the array

    if (arr[index] == target) {
        list.add(index); // Then add current index if it matches the target
    }

    return list;
}
```

🟦 **Overview:**
Both methods return all indices of the target, but differ in *when* they add the matching index:

- **Pre-processing: Adds index before recursion.**
- **Post-processing: Adds index after recursion.**

🔍 3. Rotated Binary Search Using Recursion

```
// Searches for target in a rotated sorted array using recursive binary search
static int search(int[] arr, int target, int s, int e) {
    if (s > e) {
        return -1; // Base case: not found
    }

    int mid = s + (e - s) / 2; // Avoid overflow

    if (arr[mid] == target) {
        return mid; // Target found
    }

    // Check if left half is sorted
    if (arr[s] <= arr[mid]) {
        // Target lies in left sorted half
        if (target >= arr[s] && target <= arr[mid]) {
            return search(arr, target, s, mid - 1);
        } else {
            return search(arr, target, mid + 1, e);
        }
    }

    // Right half is sorted
    if (target >= arr[mid] && target <= arr[e]) {
        return search(arr, target, mid + 1, e);
    }
```

```
    return search(arr, target, s, mid - 1);
}
```

🟦 **Overview:**
This recursive function performs binary search in a rotated sorted array. It determines whether the left or right half is sorted and recursively narrows the search range accordingly.

Sorting using recursion

Here are cleaned code snippets from each of your programs, with minimal yet insightful comments for your quick reference and understanding. These are perfect for your notes and revision.

🔹 1. Standard Merge Sort (Returns New Array)

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

    int mid = arr.length / 2;
    // Recursively sort both halves
```

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

    return merge(left, right);
}

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

    // Compare elements and merge
    while (i < left.length && j < right.length) {
        mix[k++] = left[i] < right[j] ? left[i++] : right[j++];
    }

    // Copy remaining elements
    while (i < left.length) mix[k++] = left[i++];
    while (j < right.length) mix[k++] = right[j++];

    return mix;
}
```

🔸 **Key Points:**

- **Uses additional arrays to return a new sorted array.**
- **Fully recursive with Arrays.copyOfRange() splitting.**

🔹 2. In-Place Merge Sort

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

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

    // Sort left and right halves in-place
    mergeSortInPlace(arr, s, mid);
    mergeSortInPlace(arr, mid, e);

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

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

    // Merge two sorted parts
    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++];
```

```
    // Copy back to original array
    System.arraycopy(mix, 0, arr, s, mix.length);
}
```

🔸 **Key Points:**

- **Uses index boundaries instead of array splitting.**
- **More memory-efficient than standard merge sort.**

✅ QuickSort (Recursive Implementation)

```
package recursionAll.recursionSorting;

import java.util.Arrays;

public class QuickSort {

    public static void main(String[] args) {
        int[] arr = new int[] {3, 2, 4, 5, 1, 6, 8, 7};

        sort(arr, 0, arr.length - 1);  // Sort full array from index 0 to length - 1

        System.out.println(Arrays.toString(arr));
    }

    static void sort(int[] arr, int low, int high) {
        if (low >= high) {
            return; // Base condition: subarray has 0 or 1 element
```

```
}

int s = low;
int e = high;
int mid = s + (e - s) / 2;
int pivot = arr[mid];  // Choosing middle element as pivot

// Partitioning logic
while (s <= e) {
   // Find first element from left greater than or equal to pivot
   while (arr[s] < pivot) {
      s++;
   }
   // Find first element from right smaller than or equal to pivot
   while (arr[e] > pivot) {
      e--;
   }

   // Swap if needed and move pointers
   if (s <= e) {
      int temp = arr[s];
      arr[s] = arr[e];
      arr[e] = temp;
      s++;
      e--;
   }
```

```
    }

    // Recursive calls to left and right subarrays
    sort(arr, low, e);  // Sort left partition
    sort(arr, s, high); // Sort right partition
  }
}
```

🧠 Quick Notes:

- **QuickSort is a divide-and-conquer algorithm.**
- **It works by partitioning the array around a pivot, such that elements smaller than pivot go to the left and larger go to the right.**
- **After partitioning, it recursively sorts the left and right subarrays.**
- **This implementation uses middle element as pivot and in-place partitioning with two pointers (s and e).**
- **Time Complexity:**
  - **Best & Average Case: O(n log n)**
  - **Worst Case (when array is already sorted or reverse sorted): O(n^2)**
- **Space Complexity: O(log n) due to recursion stack.**

🔹 3. Recursive Bubble Sort

```
static int[] bubble(int[] arr, int r, int c) {
  if (r < 0) return arr;

  if (c < r) {
    if (arr[c] > arr[c + 1]) {
      // Swap
      int temp = arr[c];
```

```
            arr[c] = arr[c + 1];
            arr[c + 1] = temp;
        }
        return bubble(arr, r, c + 1);  // Next comparison
    } else {
        return bubble(arr, r - 1, 0);  // Move to next pass
    }
}
```

🔸 **Key Points:**

- **Simulates nested loop structure using recursion.**
- **r indicates the range of unsorted part (like outer loop).**

🔹 4. Recursive Selection Sort

```
static int[] selection(int[] arr, int r, int c, int max) {
    if (r < 0) return arr;

    if (c < r) {
        // Track index of maximum element
        return selection(arr, r, c + 1, arr[c] > arr[max] ? c : max);
    } else {
        // Swap max with end element
        int temp = arr[max];
        arr[max] = arr[r];
        arr[r] = temp;
        return selection(arr, r - 1, 0, 0);  // Move to next range
    }
```

```
}
```

🔸 **Key Points:**

- **Finds the maximum in each pass and places it at the end.**
- **Works like classic selection sort but with recursive control.**

🔹 5. Recursive Pattern Printing

🔸 **Pattern 1: Decreasing Stars per Row (Triangle)**

```
static void pattern(int r, int c) {
   if (r == 0) return;

   if (c < r) {
      System.out.print("* ");
      pattern(r, c + 1);
   } else {
      System.out.println();
      pattern(r - 1, 0);  // Move to next row
   }
}
```

Output for n=4:

*** * * ***

*** * ***

*** ***

*****

🔸 **Key Points:**

- **Simple decreasing triangle.**
- **Controlled by rows (r) and columns (c).**

Below is your QuickSort Java code snippet with clean formatting, helpful comments, and concise notes for your study/reference. This will help you grasp the core concepts quickly during revision: