PW SKILLS

Lesson Plan

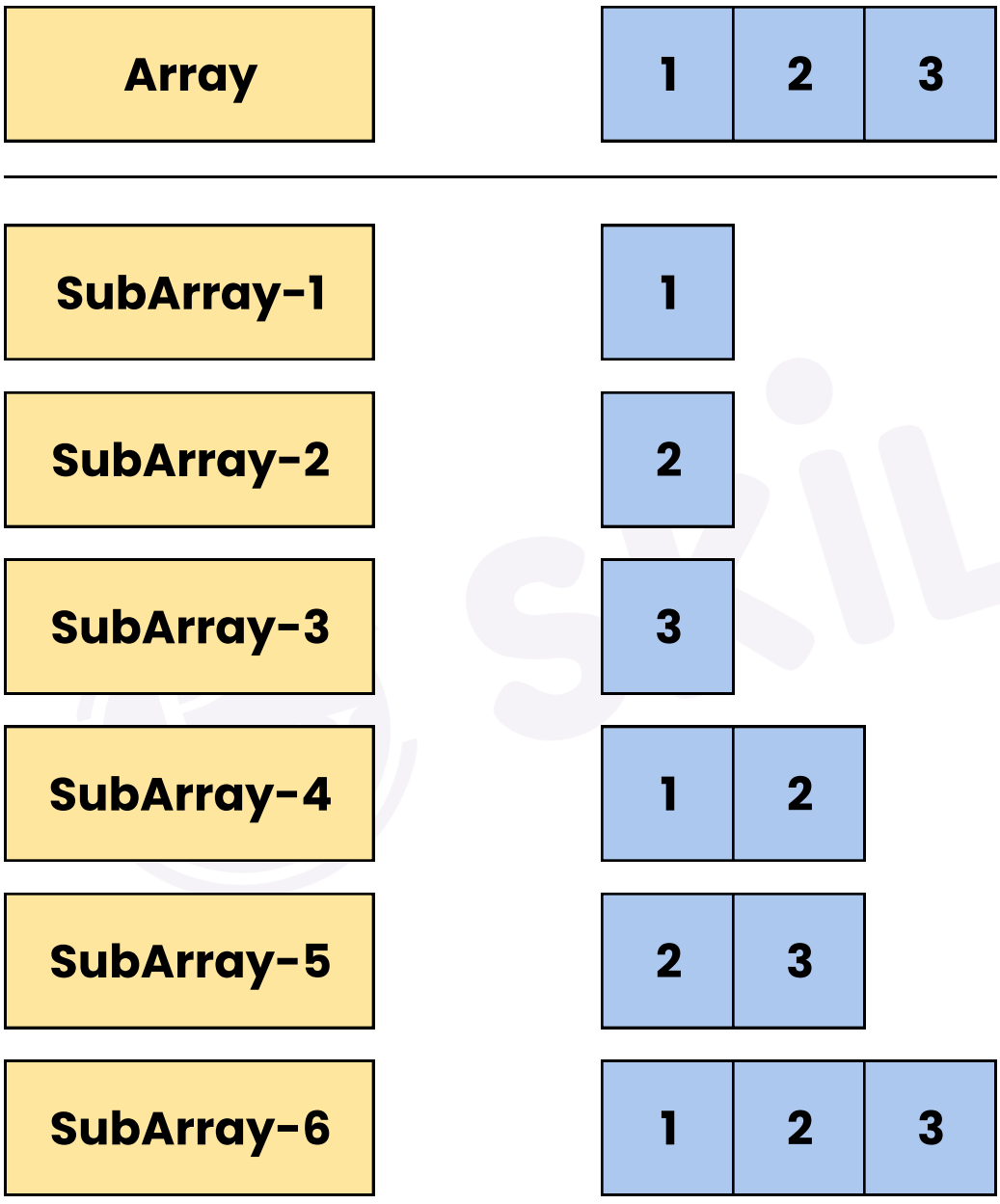
Java
Recursion 3

PW SKILLS

Recursion is a programming concept where a function calls itself directly or indirectly to solve a problem by breaking it down into smaller instances of the same problem. It involves a base case to stop the recursion and prevent an infinite loop.

Question :

Find out all the subarrays of an array.



Sample Input and Output :

- **Input :** [1, 2, 3]
- **Output :** [[1], [1, 2], [1, 2, 3], [2], [2, 3], [3]]

Code :

```
import java.util.*;

public class PrintSubarrayMain {

    public static void main(String args[]){
        PrintSubarrayMain psm=new PrintSubarrayMain();
        int arr[]= {1,2,3};
        psm.printSubArray(arr);
    }


    void printSubArray(int arr[])
    {
        int n=arr.length;
        for (int i=0; i <n; i++)
        {
            for (int j=i; j<n; j++)
            {
                for (int k=i; k⩽j; k++)
                {
                    System.out.print( arr[k]+" ");
                }
                System.out.println();
            }
        }
    }
}
```

Explanation :

This Java code defines a class PrintSubarrayMain with a method printSubArray that prints all possible subarrays of a given integer array using nested loops. The main method creates an instance and invokes the method with an example array [1,2,3].

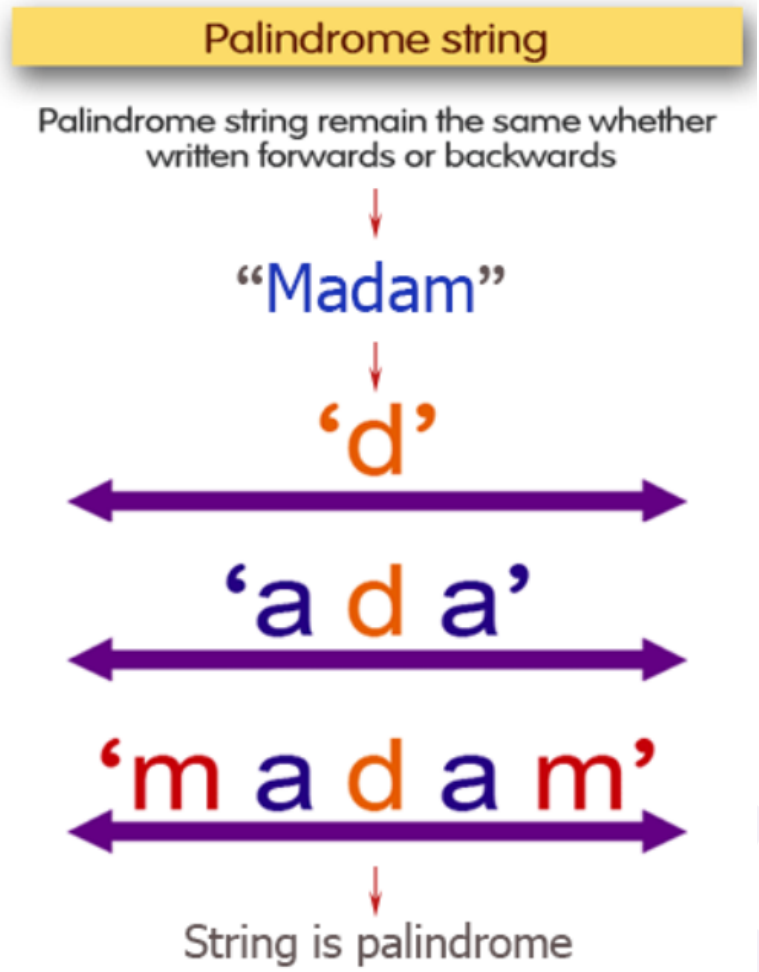
Time and Space Complexity :

- **Time Complexity :** The code has a cubic time complexity of O(n^3) due to three nested loops. For each element in the array, it generates and prints all possible subarrays, resulting in a cubic growth in time as the input size increases.

- **Space Complexity :** The space complexity is constant, denoted as O(1), since the memory usage remains fixed regardless of the input size. The code uses a few variables (i, j, k, n) that do not scale with the input array length, contributing to constant space requirements.

Question :

Find out whether a given string is palindrome or not using recursion.



Sample Input and Output :

- **Input :** “Madam”
- **Output :** true

Code :

```
public class PalindromeCheck {

    public static void main(String[] args) {
        String input1 = "Madam";

        System.out.println(isPalindrome(input1));

    }

    static boolean isPalindrome(String str) {
        int start = 0;
        int end = str.length() - 1;
        return isPalindromeRecursive(str, start, end);
    }
```

```
    private static boolean isPalindromeRecursive(String str,
int start, int end) {
        if (start ⩾ end) {
            return true;
        }

        if (str.charAt(start) ≠ str.charAt(end)) {
            return false;
        }

        return isPalindromeRecursive(str, start + 1, end -
1);
    }
}
```

Explanation :

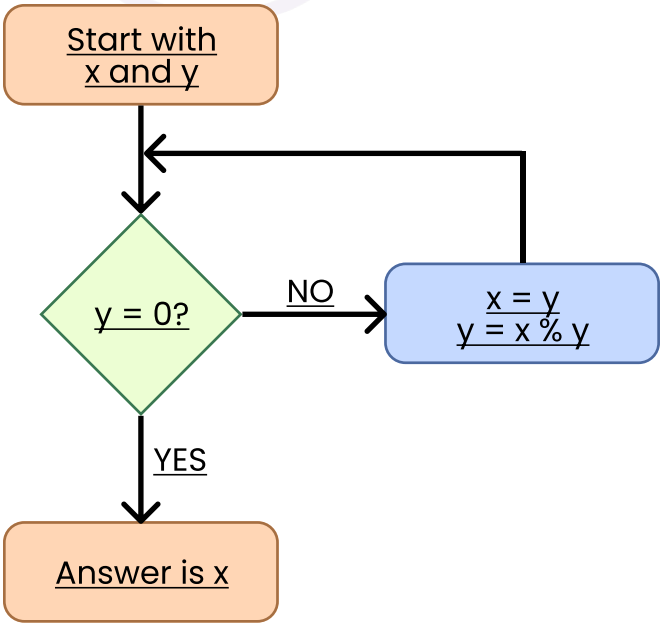
This Java code determines if a given string, like "Madam", is a palindrome using a recursive approach. It compares characters from both ends towards the center, and the recursive function returns true if the characters match. The main method tests the string "Madam" and prints the result (true).

Time and Space Complexity :

- **Time Complexity :** O(n), where n is the length of the input string. The recursive function traverses each character at most once.
- **Space Complexity :** O(n), where n is the length of the input string. The recursion stack grows linearly with the length of the input string.

Question :

Calculate Greatest Common Divisor of two numbers.



Sample Input and Output :

- **Input :** 12 , 18
- **Output :** 6

Code :

```
public class GCDCalculator {

    public static void main(String[] args) {
        int num1 = 12;
        int num2 = 18;
        int gcd = findGCD(num1, num2);
        System.out.println("GCD of " + num1 + " and " + num2
+ " is: " + gcd);
    }

       static int findGCD(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
}
```

Explanation :

The program defines a class GCDCalculator. The main method initializes two numbers (num1 and num2) and calls the findGCD function with these numbers. The findGCD function implements the Euclidean algorithm for finding the GCD. It uses a while loop to repeatedly update the values of a and b until b becomes 0. In each iteration, it calculates the remainder of the division (a % b) and updates a and b accordingly. Once the loop exits, the GCD is stored in the variable a and returned to the main method for display.

Time and Space Complexity :

- **Time Complexity :** The time complexity of the Euclidean algorithm is O(log min(a, b)). In the worst case, it performs log(min(a, b)) divisions.

- **Space Complexity :** The space complexity is O(1) as the algorithm uses only a constant amount of extra space.

Question :

Generate all binary strings of length n without consecutive 1's

Sample Input and Output :

- **Input :** 3
- **Output :** ["000", "001", "010", "100", "101"]

Code :

```
import java.util.ArrayList;
import java.util.List;

public class BinaryStringsGenerator {

    public static void main(String[] args) {
        int n = 3;
        List<String> result = generateBinaryStrings(n);
        System.out.println("Binary strings of length " + n +
" without consecutive 1's: " + result);
    }

    static List<String> generateBinaryStrings(int n) {
        List<String> result = new ArrayList<>();
        generateStrings("", n, result);
        return result;
    }

    static void generateStrings(String current, int n,
List<String> result) {
        if (n == 0) {
            result.add(current);
            return;
        }

        if (current.length() == 0 ||
current.charAt(current.length() - 1) == '0') {
            generateStrings(current + "0", n - 1, result);
            generateStrings(current + "1", n - 1, result);
        } else {
            generateStrings(current + "0", n - 1, result);
        }
    }
}
```

Explanation :

The class BinaryStringsGenerator uses a main method to initialize n and call generateBinaryStrings. The latter constructs binary strings without consecutive 1's through recursive appending of '0' or '1' based on the last character. The recursion halts when n is 0, and the current string is added to the result list.

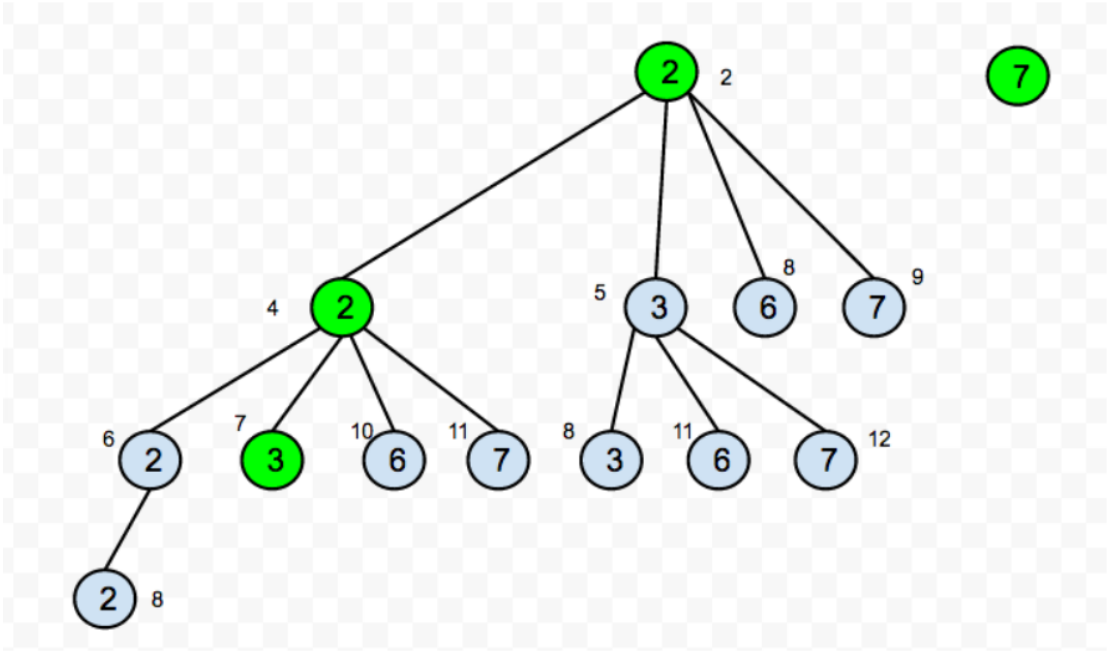
Time and Space Complexity :

- **Time Complexity :** O(2^n) - Each binary string of length n is generated, and at each position, there are two choices (0 or 1).
- **Space Complexity :** O(2^n) - The space required for storing all binary strings.

Question :

Combination Sum :

Return a list of all unique combinations of candidates where the chosen numbers sum to target.



Sample Input and Output :

- **Input :** nums =[2,3,6,7]
- **Output :** target = 7

Code :

```
public class Solution{

public static void main(String[] args) {
        Solution solution = new Solution();

        int[] nums = {2, 3, 6, 7};
        int target = 7;

        List<List<Integer>> result =
solution.combinationSum(nums, target);
        for (List<Integer> combination : result) {
            System.out.println(combination);
        }
    }

public List<List<Integer>> combinationSum(int[] nums, int
target) {
```

```
    List<List<Integer>> list = new ArrayList<>();
    Arrays.sort(nums);
    backtrack(list, new ArrayList<>(), nums, target, 0);
    return list;
}

private void backtrack(List<List<Integer>> list,
List<Integer> tempList, int [] nums, int remain, int start){
    if(remain < 0) return;
    else if(remain == 0) list.add(new
ArrayList<>(tempList));
    else{
        for(int i = start; i < nums.length; i++){
            tempList.add(nums[i]);
            backtrack(list, tempList, nums, remain -
nums[i], i);           tempList.remove(tempList.size() - 1);
        }
    }
}
}
```

Explanation :

It uses a backtracking approach to explore and build combinations, considering reuse of elements. The combinationSum method initializes the process, and backtrack recursively generates valid combinations.

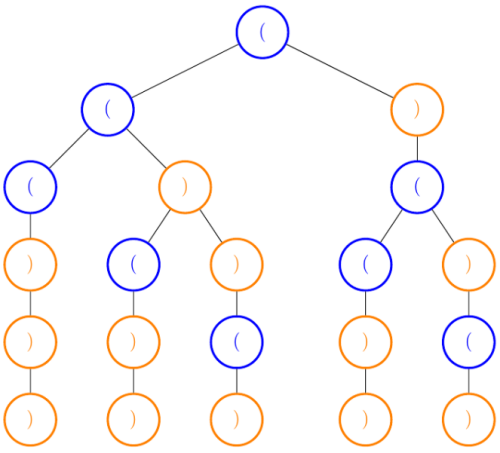
Time and Space Complexity :

- **Time Complexity :** O(2^n), where n is the length of nums.
- **Space Complexity :** O(n) due to the recursive call stack and space for the result and temporary lists.

Question :

Generate Parentheses :

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.



Sample Input and Output :

- **Input :** n=3
- **Output :** ["((()))","(()())","(())()","()(())","()()()"]

Code :

```
class Solution {
    List<String> result;
    public List<String> generateParenthesis(int n) {
        result = new ArrayList<>();
        generateParenthesis(n, n, new StringBuilder());
        return result;
    }
    public void generateParenthesis(int open, int close,
StringBuilder sb) {
        if(open==0 && close==0){
            result.add(sb.toString());
            return;
        }
        if(open > close || open < 0)
            return;
        generateParenthesis(open-1, close, sb.append('('));
        sb.deleteCharAt(sb.length() - 1);

        generateParenthesis(open, close-1, sb.append(')'));
        sb.deleteCharAt(sb.length() - 1);
    }
}
```

Explanation :

It uses backtracking to explore and build combinations, ensuring that the number of opening and closing parentheses is balanced. The generateParenthesis method initializes the process, and the helper function recursively generates valid combinations.

Time and Space Complexity :

- **Time Complexity :** O(4^n / sqrt(n)) - The number of valid combinations is a Catalan number, and each combination is constructed in O(n) time

- **Space Complexity :** O(n) - The space is used for the recursive call stack and the StringBuilder to store each combination.

Question :

Kth Symbol in Grammar :

We build a table of n rows (1-indexed). We start by writing 0 in the 1st row. Now in every subsequent row, we look at the previous row and replace each occurrence of 0 with 01, and each occurrence of 1 with 10. For example, for n = 3, the 1st row is 0, the 2nd row is 01, and the 3rd row is 0110. Given two integer n and k, return the kth (1-indexed) symbol in the nth row of a table of n rows.

Sample Input and Output :

- **Input :** n=1 , k=1
- **Output :** 0

Code :

```
class Solution {
    public int kthGrammar(int n, int k) {
        if(n < 3)
            return k - 1;
        int half = (int)Math.pow(2, n - 2);
        if(k <= half)
            return kthGrammar(n - 1, k);
        return kthGrammar(n - 1, k - half) == 0? 1 : 0;
    }
}
```

Explanation :

It recursively reduces the problem by half at each step, using the fact that each row is constructed based on the previous row.

Time and Space Complexity :

- **Time Complexity :** O(log n) - The recursive calls reduce the problem size by half in each step.
- **Space Complexity :** O(log n) - The space is used for the recursive call stack.

Question :

Count and Say : The count-and-say sequence is a sequence of digit strings defined by the recursive formula: countAndSay(1) = "1" countAndSay(n) is the way you would "say" the digit string from countAndSay(n-1), which is then converted into a different digit string. To determine how you "say" a digit string, split it into the minimal number of substrings such that each substring contains exactly one unique digit. Then for each substring, say the number of digits, then say the digit. Finally, concatenate every said digit.

"3322251"

two 3's, three 2's, one 5, and one 1

2 3 + 3 2 + 1 5 + 1 1

"23321511"

Sample Input and Output :

- **Input :** n=1
- **Output :** "1"

Code :

```
class Solution {
    public String countAndSay(int n) {
        if(n==1)
        {
            return "1";
        }
        if(n==2)
        {
            return "11";
        }
        StringBuilder nm=new StringBuilder();
        nm.append("11");
        for(int i=3;i<=n;i++)
        {
            StringBuilder kk=new StringBuilder ();
            task(nm.toString(),kk);
            nm.setLength(0);
            nm.append(kk);
        }
        return nm.toString();
    }
    public void task(String s, StringBuilder nm)
    {
        int count=1;
        for(int i=0;i<s.length()-1;i++)
        {
            if(s.charAt(i) == s.charAt(i+1))
            {
                count++;
            }
            else
            {
```

```
                nm.append(count);
                nm.append(s.charAt(i));
                count=1;
            }
        }
        nm.append(count);
        nm.append(s.charAt(s.length()-1));
    }
}
```

Explanation : This Java code generates the nth term of the "Count and Say" sequence. It iteratively constructs each term based on the previous one by counting consecutive identical digits. The countAndSay method initializes the sequence and calls the helper method task to generate each term.

Time and Space Complexity :

- **Time Complexity :** O(n+k) where k is the variable length of the strings formed.
- **Space Complexity :** O(1) , as it works iteratively , it does not occupy any space in memory.

Question :

Permutation Sequence :

The set [1, 2, 3, ..., n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for n = 3:

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

- **Input :** n=3, k=3
- **Output :** “213”

Code :

```
class Solution {
    public String getPermutation(int n, int k) {
        List<Integer> lr = new ArrayList<>();
        int sum=1;
        for(int i=1;i<=n;i++) {lr.add(i);sum*=i;}
        StringBuilder sb = new StringBuilder();
        while(lr.size()!=0 && n>0)
        {
```

```
                        sum/=n--;

            if(k%sum==0){sb.append(lr.remove(k/(sum)-1));
for(int i=lr.size()-1;i>=0;i--) sb.append(lr.get(i)); return
sb.toString();}
                    sb.append(lr.remove(k/(sum)));

            k=k%sum;
        }
        return sb.toString();
    }
}
```

Explanation : This Java code generates the kth permutation of numbers from 1 to n in lexicographical order. It uses factorials to determine the number of permutations at each digit place, iteratively selecting digits based on the given value of k. The code achieves linear time complexity and space complexity.

Time and Space Complexity :

- **Time Complexity :** O(n) - The while loop iterates through n elements, and each iteration involves constant time operations.
- **Space Complexity :** O(n) - The list lr stores the numbers from 1 to n, resulting in linear space consumption.