Recursive Power Function Java

Implement pow(x, n), which calculates x raised to the power n.Implement using recursion in Java

Here is an implementation of the `pow(x, n)` function using recursion in Java:

```java
public class PowerFunction {

    public static double pow(double x, int n) {
        // Base case: any number raised to the power 0 is 1
        if (n == 0) {
            return 1;
        }

        // If n is negative, calculate the reciprocal
        if (n < 0) {
            return 1 / pow(x, -n);
        }

        // Recursive step: calculate half power to optimize performance
        double half = pow(x, n / 2);

        // If n is even, the result is half * half
        if (n % 2 == 0) {
            return half * half;
        } else {
            // If n is odd, multiply an additional x
            return half * half * x;
        }
    }

    public static void main(String[] args) {
        System.out.println(pow(2, 10)); // Output: 1024.0
        System.out.println(pow(2, -2)); // Output: 0.25
        System.out.println(pow(3, 0));  // Output: 1.0
    }
}
```

Explanation:

1. **Base Case:**
   When `n == 0`, the result is `1`, as any number raised to the power of 0 is `1`.
2. **Negative Exponent:**
   If `n < 0`, we calculate the reciprocal by calling `pow(x, -n)` and return `1 / result`.
3. **Recursive Optimization:**
   Instead of directly calculating `x * pow(x, n - 1)` which can lead to higher stack usage, we calculate `pow(x, n / 2)` and reuse it:
   - If `n` is even: $x^n = (x^{n/2}) \times (x^{n/2})$

- If `n` is odd: $x^n = (x^{n//2}) \times (x^{n//2}) \times x$

Example Runs:

- `pow(2, 10)`
  Computes $2^{10} = 2^5 \times 2^5$ (recursively calculates $2^5 = 2^2 \times 2^2 \times 2$).
- `pow(2, -2)`
  Computes $2^{-2} = 1/(2^2) = 1/4 = 0.25$.

This recursive solution is efficient due to its logarithmic complexity $O(\log n)$.

You are given a stack St. You have to reverse the stack using recursion.

To reverse a stack using recursion in Java, you can implement the following code:

Code:

```java
import java.util.Stack;

public class ReverseStack {

    // Function to reverse the stack
    public static void reverseStack(Stack<Integer> stack) {
        // Base case: if stack is empty, return
        if (stack.isEmpty()) {
            return;
        }

        // Remove the top element
        int top = stack.pop();

        // Recursively reverse the remaining stack
        reverseStack(stack);

        // Insert the top element at the bottom of the stack
        insertAtBottom(stack, top);
    }

    // Helper function to insert an element at the bottom of the stack
    private static void insertAtBottom(Stack<Integer> stack, int element) {
        // Base case: if stack is empty, push the element
        if (stack.isEmpty()) {
            stack.push(element);
            return;
        }

        // Remove the top element
        int top = stack.pop();

        // Recursively call to reach the bottom
        insertAtBottom(stack, element);

        // Push the top element back after inserting the element at the bottom
        stack.push(top);
    }
```

```java
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);
        stack.push(4);

        System.out.println("Original Stack: " + stack);

        reverseStack(stack);

        System.out.println("Reversed Stack: " + stack);
    }
}
```

Explanation:

1. `reverseStack` **Function:**
   - The function pops the top element of the stack.
   - Recursively calls itself to reverse the remaining stack.
   - Uses the helper function `insertAtBottom` to place the popped element at the bottom of the stack.
2. `insertAtBottom` **Function:**
   - Inserts the given element at the bottom of the stack by recursively removing all elements and then adding the given element.
3. **Base Cases:**
   - For `reverseStack`, the base case is when the stack is empty, i.e., there's nothing to reverse.
   - For `insertAtBottom`, the base case is when the stack is empty, at which point the element is pushed.

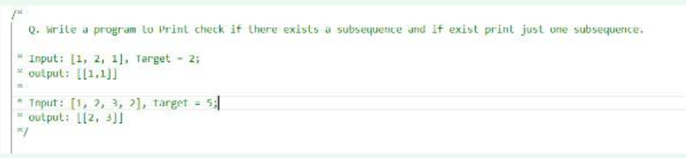
Example:

Input:

Stack: `[1, 2, 3, 4]`

Output:

Reversed Stack: `[4, 3, 2, 1]`



```
/*
  Q. Write a program to Print check if there exists a subsequence and if exist print just one subsequence.

 * Input: [1, 2, 1], Target - 2;
 * output: [[1,1]]
 *
 * Input: [1, 2, 3, 2], target = 5;
 * output: [[2, 3]]
 */
```

Write the above program using recurison in Java

Here's a Java program that uses recursion to find if there exists a subsequence with a sum equal to the target. If such a subsequence exists, it prints one such subsequence.

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

public class SubsequenceSum {

    public static boolean findSubsequence(int[] arr, int index, int target, List<Integer> result) {
        // Base case: if target becomes 0, we found a subsequence with the required sum
        if (target == 0) {
            System.out.println(result);
            return true;
        }

        // If we've reached the end of the array and target isn't 0, return false
        if (index == arr.length) {
            return false;
        }

        // Include the current element in the subsequence and recurse
        result.add(arr[index]);
        if (findSubsequence(arr, index + 1, target - arr[index], result)) {
            return true;
        }

        // Backtrack and remove the element from the result
        result.remove(result.size() - 1);

        // Exclude the current element and recurse
        if (findSubsequence(arr, index + 1, target, result)) {
            return true;
        }

        return false;
    }

    public static void main(String[] args) {
        int[] arr1 = {1, 2, 1};
        int target1 = 2;
        List<Integer> result1 = new ArrayList<>();
        System.out.print("Input: [1, 2, 1], Target = 2, Output: ");
        if (!findSubsequence(arr1, 0, target1, result1)) {
            System.out.println("No subsequence found");
        }

        int[] arr2 = {1, 2, 3, 2};
        int target2 = 5;
        List<Integer> result2 = new ArrayList<>();
        System.out.print("Input: [1, 2, 3, 2], Target = 5, Output: ");
        if (!findSubsequence(arr2, 0, target2, result2)) {
            System.out.println("No subsequence found");
        }
    }
}
```

Explanation:

1. **Base Case:**
    - If the `target` becomes `0`, it means we've found a subsequence with the desired sum, so we print it and return `true`.
    - If `index` reaches the end of the array and `target` is not `0`, we return `false` as there's no valid subsequence in this path.

2. **Recursive Steps:**
   - We have two choices for each element:
     1. **Include** the current element in the subsequence, subtract its value from `` `target` ``, and call the function recursively with the next index.
     2. **Exclude** the current element and recursively call the function with the next index.
   - If any of these recursive calls returns `` `true` ``, we stop and return `` `true` `` as we only need one valid subsequence.

Example Outputs:

For the inputs given:

- `` `Input: [1, 2, 1], Target = 2` `` will output `` `[[1, 1]]` ``
- `` `Input: [1, 2, 3, 2], Target = 5` `` will output `` `[[2, 3]]` ``

This code efficiently finds and prints one subsequence with the desired sum using recursion and backtracking.

```
/*
 * Combination Sum - 1
   Problem Statement:

   Given an array of distinct integers and a target, you have to return the list of all unique
   combinations where the chosen numbers sum to target. You may return the combinations in any order.

   The same number may be chosen from the given array an unlimited number of times.
   Two combinations are unique if the frequency of at least one of the chosen numbers is different.

   It is guaranteed that the number of unique combinations that sum up to target is
   less than 150 combinations for the given input.

   Examples:

   Example 1:
   Input: array = [2,3,6,7], target = 7
   Output: [[2,2,3],[7]]

   Explanation: 2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
                7 is a candidate, and 7 = 7.
                These are the only two combinations.


   Example 2:
   Input: array = [2], target = 1
   Output: []
   Explaination: No combination is possible.
*/
```

Write the above program using recurison in Java

Here is the Java program that uses recursion to solve the "Combination Sum" problem as described:

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

public class CombinationSum {

    public static List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<>();
        findCombinations(candidates, target, 0, new ArrayList<>(), result);
        return result;
    }

    private static void findCombinations(int[] candidates, int target, int index, List<Integ
er> current, List<List<Integer>> result) {
        // Base case: if target is zero, add the current list to result
        if (target == 0) {
```

```
            result.add(new ArrayList<>(current));
            return;
        }

        // If the target becomes negative, there's no point in exploring further
        if (target < 0) {
            return;
        }

        // Loop through the candidates starting from 'index' to allow repetition of elements
        for (int i = index; i < candidates.length; i++) {
            current.add(candidates[i]); // Include the candidate
            findCombinations(candidates, target - candidates[i], i, current, result); // Rec
ursive call with reduced target
            current.remove(current.size() - 1); // Backtrack to try other combinations
        }
    }

    public static void main(String[] args) {
        int[] candidates1 = {2, 3, 6, 7};
        int target1 = 7;
        System.out.println("Input: array = [2,3,6,7], target = 7");
        System.out.println("Output: " + combinationSum(candidates1, target1));

        int[] candidates2 = {2};
        int target2 = 1;
        System.out.println("Input: array = [2], target = 1");
        System.out.println("Output: " + combinationSum(candidates2, target2));
    }
}
```

Explanation:

1. `combinationSum` **function**:
   - This function initializes the result list and starts the recursive search with `findCombinations`.
2. `findCombinations` **function**:
   - **Base Case 1**: If `target` is `0`, it means we have found a combination that adds up to the target, so we add the current combination to the result list.
   - **Base Case 2**: If `target` is less than `0`, it means the current combination has exceeded the target, so we return without adding it to the result list.
   - **Loop through candidates**: Starting from the current index, it adds each candidate to the current list and calls the recursive function with the reduced target. The recursive call allows the same element to be used multiple times.
   - **Backtracking**: After each recursive call, it removes the last added element (backtracks) to explore other combinations.

Example Runs:

- **Example 1**: `Input: array = [2,3,6,7], target = 7`
  Output: `[[2, 2, 3], [7]]`
- **Example 2**: `Input: array = [2], target = 1`
  Output: `[]`

This solution uses recursion with backtracking to generate all unique combinations that sum up to the target. The solution allows the use of the same number multiple times as required by the problem.

```
/*
 * Combination Sum II – Find all unique combinations
   In this article we will solve the most asked interview question "Combination Sum II –
   Problem Statement: Given a collection of candidate numbers (candidates) and
   a target number (target), find all unique combinations in candidates where the candidate numbers
   sum to target. Each number in candidates may only be used once in the combination.

   Note: The solution set must not contain duplicate combinations.

   Examples:

   Example 1:
   Input: candidates = [10,1,2,7,6,1,5], target = 8
   Output:
   [[1,1,6],[1,2,5],[1,7],[2,6]]

   Explanation: These are the unique combinations whose sum is equal to target.

   Example 2:
   Input: candidates = [2,5,2,1,2], target = 5
   Output: [[1,2,2],[5]]
   Explanation: These are the unique combinations whose sum is equal to target.
*/
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion and backtracking to solve the "Combination Sum II" problem as described:

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

public class CombinationSumII {

    public static List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(candidates); // Sort the array to handle duplicates
        findCombinations(candidates, target, 0, new ArrayList<>(), result);
        return result;
    }

    private static void findCombinations(int[] candidates, int target, int index, List<Integ
er> current, List<List<Integer>> result) {
        // Base case: if target is zero, add the current list to result
        if (target == 0) {
            result.add(new ArrayList<>(current));
            return;
        }

        for (int i = index; i < candidates.length; i++) {
            // Skip duplicates
            if (i > index && candidates[i] == candidates[i - 1]) {
                continue;
            }

            // If the current candidate is greater than the remaining target, break to avoid
unnecessary calls
            if (candidates[i] > target) {
                break;
            }

            current.add(candidates[i]); // Include the candidate
            findCombinations(candidates, target - candidates[i], i + 1, current, result); //
Recurse with next index
            current.remove(current.size() - 1); // Backtrack to explore other combinations
        }
    }

    public static void main(String[] args) {
        int[] candidates1 = {10, 1, 2, 7, 6, 1, 5};
        int target1 = 8;
        System.out.println("Input: candidates = [10,1,2,7,6,1,5], target = 8");
        System.out.println("Output: " + combinationSum2(candidates1, target1));
```

```java
        int[] candidates2 = {2, 5, 2, 1, 2};
        int target2 = 5;
        System.out.println("Input: candidates = [2,5,2,1,2], target = 5");
        System.out.println("Output: " + combinationSum2(candidates2, target2));
    }
}
```

Explanation:

1. **`combinationSum2` Method:**
    - This method initializes the result list and sorts the input array to help in handling duplicates.
    - Calls the recursive `findCombinations` method to explore all combinations.
2. **`findCombinations` Method:**
    - **Base Case:** If `target` becomes zero, it means we have a combination that adds up to the target. Add a copy of `current` to `result`.
    - **Loop through Candidates:**
        - Start from `index` to avoid reusing the same element in the same position.
        - **Skip Duplicates:** If the current candidate is the same as the previous one, skip it to avoid duplicate combinations.
        - **Target Exceeded Check:** If `candidates[i]` is greater than the remaining target, break out of the loop, as further elements will only increase the sum.
        - **Include Candidate and Recurse:** Add the candidate to the current combination, reduce the target, and recurse with the next index.
        - **Backtrack:** After the recursive call, remove the last added candidate to explore other combinations.

Example Outputs:

1. **Example 1:**
    - **Input:** `candidates = [10, 1, 2, 7, 6, 1, 5], target = 8`
    - **Output:** `[[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]`
    - **Explanation:** The unique combinations whose sum equals 8 are found as follows:
        - `[1, 1, 6]` (uses `1` twice and `6` once)
        - `[1, 2, 5]` (uses `1`, `2`, and `5`)
        - `[1, 7]` (uses `1` and `7`)
        - `[2, 6]` (uses `2` and `6`)
2. **Example 2:**
    - **Input:** `candidates = [2, 5, 2, 1, 2], target = 5`
    - **Output:** `[[1, 2, 2], [5]]`
    - **Explanation:** The unique combinations whose sum equals 5 are:
        - `[1, 2, 2]` (uses `1` and `2` twice)
        - `[5]` (uses `5` alone)

This solution effectively handles duplicates and ensures each number is used at most once per combination while generating unique combinations that sum to the target.

```
/*
 Find all valid combinations of k numbers that sum up to n such that the following conditions are true:

 Only numbers 1 through 9 are used.
 Each number is used at most once.
 Return a list of all possible valid combinations. The list must not contain the same combination twice,
  and the combinations may be returned in any order.

 Example 1:
 Input: k = 3, n = 7
 Output: [[1,2,4]]
 Explanation:
 1 + 2 + 4 = 7
 There are no other valid combinations.

 Example 2:
 Input: k = 3, n = 9
 Output: [[1,2,6],[1,3,5],[2,3,4]]
 Explanation:
 1 + 2 + 6 = 9
 1 + 3 + 5 = 9
 2 + 3 + 4 = 9
 There are no other valid combinations.

 Example 3:
 Input: k = 4, n = 1
 Output: []
 Explanation: There are no valid combinations.
 Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4 = 10 and since 10 > 1,
  there are no valid combination.
*/
```

Write the above program using recurison in Java and explain it using output

Here is the Java program to find all valid combinations of `k` numbers that sum up to `n` using recursion and backtracking:

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

public class CombinationSumIII {

    public static List<List<Integer>> combinationSum3(int k, int n) {
        List<List<Integer>> result = new ArrayList<>();
        findCombinations(k, n, 1, new ArrayList<>(), result);
        return result;
    }

    private static void findCombinations(int k, int n, int start, List<Integer> current, List<List<Integer>> result) {
        // Base case: if the current combination has k numbers and the target sum is reached
        if (k == 0 && n == 0) {
            result.add(new ArrayList<>(current));
            return;
        }

        // If k becomes 0 or n is less than 0, stop the recursion
        if (k == 0 || n < 0) {
            return;
        }

        // Iterate from the current number (start) to 9, as only numbers 1 to 9 are allowed
        for (int i = start; i <= 9; i++) {
            current.add(i); // Include i in the current combination
            findCombinations(k - 1, n - i, i + 1, current, result); // Recurse with updated k, n, and start
            current.remove(current.size() - 1); // Backtrack
        }
    }

    public static void main(String[] args) {
        // Example 1
        int k1 = 3, n1 = 7;
        System.out.println("Input: k = 3, n = 7");
        System.out.println("Output: " + combinationSum3(k1, n1));

        // Example 2
        int k2 = 3, n2 = 9;
        System.out.println("Input: k = 3, n = 9");
```

```
        System.out.println("Output: " + combinationSum3(k2, n2));

        // Example 3
        int k3 = 4, n3 = 1;
        System.out.println("Input: k = 4, n = 1");
        System.out.println("Output: " + combinationSum3(k3, n3));
    }
}
```

Explanation:

1. **`combinationSum3` Method**:
   - Initializes the result list and calls the helper method `findCombinations` to find all valid combinations.
2. **`findCombinations` Method**:
   - **Base Case 1**: If `k == 0` (i.e., we've selected `k` numbers) and `n == 0` (i.e., the target sum is reached), it means we found a valid combination. Add a copy of `current` to `result`.
   - **Base Case 2**: If `k == 0` or `n < 0`, the recursion stops because no valid combination can be formed from this point.
   - **Loop Through Possible Numbers**: Iterate from `start` to `9` (since only numbers `1` to `9` are allowed):
     - **Include the Number**: Add `i` to the current combination.
     - **Recurse**: Call `findCombinations` with `k - 1`, `n - i`, and `i + 1` (to ensure each number is used at most once and in ascending order).
     - **Backtrack**: Remove the last added element to explore other combinations.

Example Outputs:

1. **Example 1**:
   - **Input:** `k = 3`, `n = 7`
   - **Output:** `[[1, 2, 4]]`
   - **Explanation:** The only valid combination of 3 numbers that adds up to 7 is `[1, 2, 4]`.
2. **Example 2**:
   - **Input:** `k = 3`, `n = 9`
   - **Output:** `[[1, 2, 6], [1, 3, 5], [2, 3, 4]]`
   - **Explanation:** The valid combinations of 3 numbers that add up to 9 are:
     - `[1, 2, 6]`
     - `[1, 3, 5]`
     - `[2, 3, 4]`
3. **Example 3**:
   - **Input:** `k = 4`, `n = 1`
   - **Output:** `[]`
   - **Explanation:** It's impossible to find 4 different numbers that sum up to 1, so the output is an empty list.

This code efficiently finds all unique combinations of `k` numbers that sum to `n` using recursion and backtracking, with constraints on the range of numbers (1 to 9) and that each number is used only once per combination.

```
// Write a program to print all the SubSequences with sum k.
/*
  Q. Write a program to count all the Subsequences with sum k.

* Input: [1, 2, 1], Target = 2;
* output: ([[1,1], [2]]) = 2
*
* Input: [1, 2, 3, 2], target = 5;
* output: ([[1, 2, 2], [2, 3], [3, 2]]) = 3
*/

```

Write the above program using recurison in Java and explain it using output

Here's a Java program that prints all subsequences of an array whose elements sum up to a given target `k` and counts the total number of such subsequences.

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

public class SubsequencesWithSumK {

    public static int countSubsequences(int[] arr, int target) {
        List<List<Integer>> result = new ArrayList<>();
        findSubsequences(arr, 0, target, new ArrayList<>(), result);
        System.out.println("Subsequences with sum " + target + ": " + result);
        return result.size();
    }

    private static void findSubsequences(int[] arr, int index, int target, List<Integer> current, List<List<Integer>> result) {
        // Base case: if target is reached, add the current subsequence to result
        if (target == 0) {
            result.add(new ArrayList<>(current));
            return;
        }

        // If index goes out of bounds or target is negative, return
        if (index >= arr.length || target < 0) {
            return;
        }

        // Include the current element
        current.add(arr[index]);
        findSubsequences(arr, index + 1, target - arr[index], current, result);
        current.remove(current.size() - 1); // Backtrack

        // Exclude the current element
        findSubsequences(arr, index + 1, target, current, result);
    }

    public static void main(String[] args) {
        // Example 1
        int[] arr1 = {1, 2, 1};
        int target1 = 2;
        System.out.println("Input: [1, 2, 1], Target = 2");
        int count1 = countSubsequences(arr1, target1);
        System.out.println("Number of subsequences: " + count1);

        // Example 2
        int[] arr2 = {1, 2, 3, 2};
        int target2 = 5;
        System.out.println("Input: [1, 2, 3, 2], Target = 5");
        int count2 = countSubsequences(arr2, target2);
        System.out.println("Number of subsequences: " + count2);
```

```
        }
    }
```

Explanation:

1. `` `countSubsequences` `` **Method**:
    - This method initializes a list to store valid subsequences and calls the helper method `` `findSubsequences` `` to find all subsequences that sum up to the target.
    - It also prints all subsequences and returns the count of such subsequences.
2. `` `findSubsequences` `` **Method**:
    - **Base Case 1**: If `` `target` `` is `` `0` ``, it means we have found a subsequence with the desired sum. Add a copy of `` `current` `` to `` `result` ``.
    - **Base Case 2**: If `` `index` `` exceeds the array length or if `` `target` `` becomes negative, stop recursion, as no valid subsequence can be formed in this path.
    - **Recursive Case**:
        - **Include the Current Element**: Add `` `arr[index]` `` to `` `current` ``, reduce `` `target` `` by `` `arr[index]` ``, and call `` `findSubsequences` `` for the next index.
        - **Backtrack**: Remove the last added element from `` `current` `` to explore subsequences that exclude the current element.
        - **Exclude the Current Element**: Call `` `findSubsequences` `` without adding `` `arr[index]` `` to `` `current` ``.

Example Outputs:

1. **Example 1**:
    - **Input**: `` `arr = [1, 2, 1]` ``, `` `target = 2` ``
    - **Output**: `` `Subsequences with sum 2: [[1, 1], [2]]` ``
    - **Explanation**: The valid subsequences with sum equal to 2 are `` `[1, 1]` `` and `` `[2]` ``.
    - **Count**: `` `2` ``
2. **Example 2**:
    - **Input**: `` `arr = [1, 2, 3, 2]` ``, `` `target = 5` ``
    - **Output**: `` `Subsequences with sum 5: [[1, 2, 2], [2, 3], [3, 2]]` ``
    - **Explanation**: The valid subsequences with sum equal to 5 are `` `[1, 2, 2]` ``, `` `[2, 3]` ``, and `` `[3, 2]` ``.
    - **Count**: `` `3` ``

This code efficiently finds all subsequences that sum to the target and counts them using recursion and backtracking.

1980. Find Unique Binary String

Medium | Topics | Companies | Hint

Given an array of strings `nums` containing `n` **unique** binary strings each of length `n`, return *a binary string of length* `n` *that **does not appear** in* `nums`*. If there are multiple answers, you may return **any** of them.*

Example 1:

```
Input: nums = ["01","10"]
Output: "11"
Explanation: "11" does not appear in nums. "00" would also be correct.
```

Example 2:

```
Input: nums = ["00","01"]
Output: "11"
Explanation: "11" does not appear in nums. "10" would also be correct.
```

Example 3:

```
Input: nums = ["111","011","001"]
Output: "101"
Explanation: "101" does not appear in nums. "000", "010", "100", and "110" would also be correct.
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion to find a unique binary string that does not appear in the input array. This approach generates all possible binary strings of length `n` and checks if they are present in the given array `nums`. If a unique string is found, it is returned.

```java
import java.util.HashSet;
import java.util.Set;

public class UniqueBinaryString {

    public static String findDifferentBinaryString(String[] nums) {
        Set<String> numSet = new HashSet<>();
        for (String num : nums) {
            numSet.add(num); // Add all binary strings from nums to a set
        }

        return findUniqueBinaryString(nums.length, "", numSet);
    }

    private static String findUniqueBinaryString(int n, String current, Set<String> numSet)
{
        // Base case: if the binary string has reached length n
        if (current.length() == n) {
            if (!numSet.contains(current)) { // Check if it's not in nums
                return current; // Return the unique binary string
            }
            return null; // Return null if this string is in nums
        }

        // Recursive case: try adding '0' or '1' to the current string
        String withZero = findUniqueBinaryString(n, current + "0", numSet);
        if (withZero != null) {
            return withZero;
        }

        String withOne = findUniqueBinaryString(n, current + "1", numSet);
        return withOne;
    }

    public static void main(String[] args) {
        // Example 1
        String[] nums1 = {"01", "10"};
        System.out.println("Input: nums = [\"01\", \"10\"]");
        System.out.println("Output: " + findDifferentBinaryString(nums1));

        // Example 2
        String[] nums2 = {"00", "01"};
```

```java
        System.out.println("Input: nums = [\"00\", \"01\"]");
        System.out.println("Output: " + findDifferentBinaryString(nums2));

        // Example 3
        String[] nums3 = {"111", "011", "001"};
        System.out.println("Input: nums = [\"111\", \"011\", \"001\"]");
        System.out.println("Output: " + findDifferentBinaryString(nums3));
    }
}
```

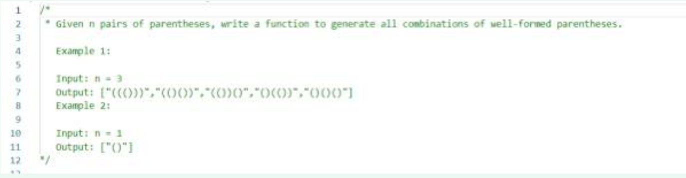
Explanation

1. **`findDifferentBinaryString` Method**:
   - This method initializes a `Set` to store all the binary strings present in `nums`. Using a `Set` provides O(1) average-time complexity for checking if a string is in `nums`.
   - Calls the recursive helper function `findUniqueBinaryString` to find a unique binary string of the given length.
2. **`findUniqueBinaryString` Method**:
   - **Base Case**: If the length of the `current` string equals `n`, check if it is in the `numSet`. If it is not, return it as a unique binary string.
   - **Recursive Case**: Try appending `'0'` and `'1'` to the `current` string and recursively search for a unique binary string. If a unique string is found (i.e., `withZero` or `withOne` is non-null), return it immediately.
3. **Backtracking**: The recursion tries all possible binary strings by building them character by character, ensuring it doesn't miss any combination.

Example Outputs

1. **Example 1**:
   - **Input**: `nums = ["01", "10"]`
   - **Possible Unique Outputs**: `"11"` (does not appear in `nums`)
   - **Explanation**: The function generates all binary strings of length `2`: `"00"`, `"01"`, `"10"`, and `"11"`. Since `"11"` is not in `nums`, it returns `"11"`.
2. **Example 2**:
   - **Input**: `nums = ["00", "01"]`
   - **Possible Unique Outputs**: `"11"`
   - **Explanation**: The function generates all binary strings of length `2` and finds that `"11"` is not present in `nums`.
3. **Example 3**:
   - **Input**: `nums = ["111", "011", "001"]`
   - **Possible Unique Outputs**: `"101"`
   - **Explanation**: The function generates all binary strings of length `3` and finds that `"101"` is not in `nums`. Other possible outputs could include `"000"`, `"010"`, `"100"`, and `"110"`.

This approach ensures that all binary strings are considered, and recursion allows for easy exploration of all possible binary strings by building them step-by-step. The backtracking prevents redundant checks, returning the first unique string found.



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

   Example 1:

   Input: n = 3
   Output: ["((()))","(()())","(())()","()(())","()()()"]
   Example 2:

   Input: n = 1
   Output: ["()"]
*/
```

Write the above program using recurison in Java and explain it using output

Here’s a Java program that generates all combinations of well-formed parentheses using recursion and backtracking.

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

public class GenerateParentheses {

    public static List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        generateCombinations(result, "", n, n);
        return result;
    }

    private static void generateCombinations(List<String> result, String current, int open,
int close) {
        // Base case: if there are no more open or close parentheses to add, add the current
combination
        if (open == 0 && close == 0) {
            result.add(current);
            return;
        }

        // If there are open parentheses left, add an open parenthesis
        if (open > 0) {
            generateCombinations(result, current + "(", open - 1, close);
        }

        // If there are more closing parentheses than open ones, add a closing parenthesis
        if (close > open) {
            generateCombinations(result, current + ")", open, close - 1);
        }
    }

    public static void main(String[] args) {
        // Example 1
        int n1 = 3;
        System.out.println("Input: n = " + n1);
        System.out.println("Output: " + generateParenthesis(n1));

        // Example 2
        int n2 = 1;
        System.out.println("Input: n = " + n2);
        System.out.println("Output: " + generateParenthesis(n2));
    }
}
```

Explanation

1. **`generateParenthesis` Method**:

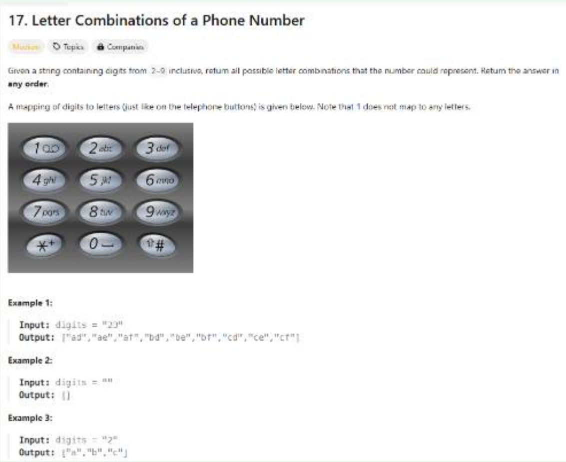
- This is the main method that initializes the result list and calls the recursive helper function `generateCombinations`.

2. `generateCombinations` **Method**:
   - **Base Case**: If both `open` and `close` are `0`, it means we have used up all pairs of parentheses and the current combination is well-formed. We add `current` to the result list.
   - **Recursive Case**:
     - **Add an Open Parenthesis**: If there are open parentheses left (`open > 0`), we add an open parenthesis to the current combination and decrease the `open` count.
     - **Add a Close Parenthesis**: If the number of `close` parentheses is greater than `open`, we add a close parenthesis to the current combination and decrease the `close` count. This condition ensures that we only add a close parenthesis if it won't make the parentheses unbalanced.

Example Outputs

1. **Example 1**:
   - **Input**: `n = 3`
   - **Output**: `["((()))", "(()())", "(())()", "()(())", "()()()"]`
   - **Explanation**:
     - The program generates all valid combinations with 3 pairs of parentheses.
     - The result list contains well-formed parentheses such as:
       - `"((()))"`: Three open parentheses followed by three close parentheses.
       - `"(()())"`: Two open, one close, one open, and two close parentheses.
       - `"(())()"`: One open, one close, one open, one close, and then two more parentheses.
       - `"()(())"`: One open, one close, two open, and two close parentheses.
       - `"()()()"`: Alternating open and close parentheses.
2. **Example 2**:
   - **Input**: `n = 1`
   - **Output**: `["()"]`
   - **Explanation**:
     - Only one pair of parentheses is needed, so the only possible valid combination is `"()"`.

This code uses recursion and backtracking to explore all possible well-formed combinations of parentheses by keeping track of the remaining open and close parentheses to ensure the balance.


17. Letter Combinations of a Phone Number

Medium Topics Companies

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1:

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Example 2:

```
Input: digits = ""
Output: []
```

Example 3:

```
Input: digits = "2"
Output: ["a","b","c"]
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion (backtracking) to generate all possible letter combinations for a given phone number string. Each digit from `2` to `9` maps to a set of letters, similar to the mapping on a traditional phone keypad.

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

public class LetterCombinationsPhoneNumber {

    // Mapping of digits to corresponding letters
    private static final String[] KEYPAD = {
        "",    // 0
        "",    // 1
        "abc", // 2
        "def", // 3
        "ghi", // 4
        "jkl", // 5
        "mno", // 6
        "pqrs",// 7
        "tuv", // 8
        "wxyz" // 9
    };

    public static List<String> letterCombinations(String digits) {
        List<String> result = new ArrayList<>();
        if (digits == null || digits.length() == 0) {
            return result;
        }
        backtrack(result, digits, 0, "");
        return result;
    }

    private static void backtrack(List<String> result, String digits, int index, String curr
ent) {
        // Base case: if the current string length equals the input digits length
        if (index == digits.length()) {
            result.add(current);
            return;
        }

        // Get the letters that the current digit maps to
        String letters = KEYPAD[digits.charAt(index) - '0'];
        for (char letter : letters.toCharArray()) {
            // Recursive call to build the combinations
            backtrack(result, digits, index + 1, current + letter);
        }
    }

    public static void main(String[] args) {
        // Example 1
        String digits1 = "23";
        System.out.println("Input: digits = \"" + digits1 + "\"");
        System.out.println("Output: " + letterCombinations(digits1));

        // Example 2
        String digits2 = "";
        System.out.println("Input: digits = \"" + digits2 + "\"");
        System.out.println("Output: " + letterCombinations(digits2));

        // Example 3
        String digits3 = "2";
        System.out.println("Input: digits = \"" + digits3 + "\"");
        System.out.println("Output: " + letterCombinations(digits3));
```

```
        }
    }
```

Explanation:

1. `` `letterCombinations` `` **Method**:
   - This method initializes the result list and checks if the input string `` `digits` `` is empty. If it is, it returns an empty list.
   - It then calls the recursive helper method `` `backtrack` `` to build the combinations.
2. `` `backtrack` `` **Method**:
   - **Base Case**: If `` `index` `` equals the length of `` `digits` ``, it means we have constructed a complete combination of letters corresponding to each digit in the input. We add the current combination (`` `current` ``) to `` `result` ``.
   - **Recursive Case**: For each digit, get the corresponding letters from `` `KEYPAD` `` and iterate through each letter. For each letter, append it to the `` `current` `` combination and call `` `backtrack` `` with the next index.
   - **Backtracking**: By appending each letter and moving to the next index recursively, this function builds all possible combinations of letters for the given digits.

Example Outputs:

1. **Example 1**:
   - **Input**: `` `digits = "23"` ``
   - **Output**: `` `["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]` ``
   - **Explanation**: The digit `` `2` `` maps to `` `["a", "b", "c"]` ``, and `` `3` `` maps to `` `["d", "e", "f"]` ``. The function generates all combinations of letters for these digits.
2. **Example 2**:
   - **Input**: `` `digits = ""` ``
   - **Output**: `` `[]` ``
   - **Explanation**: Since the input is empty, there are no digits to map to letters, so the output is an empty list.
3. **Example 3**:
   - **Input**: `` `digits = "2"` ``
   - **Output**: `` `["a", "b", "c"]` ``
   - **Explanation**: The digit `` `2` `` maps to `` `["a", "b", "c"]` ``. Since there is only one digit, the output is simply each letter corresponding to `` `2` ``.

This program efficiently generates all possible combinations of letters for each digit in the input using recursion and backtracking. Each recursive call builds a partial combination, and backtracking ensures that all potential combinations are explored.

```
/*
 * Power Set: Print all the possible subsequences of the String
   Problem Statement: Given a string, find all the possible subsequences of the string.

   * To calculate the total subseques, the formula is 2^n.
   Examples:

   Example 1:
   Input: str = "abc"
   Output: a ab abc ac b bc c
   Explanation: Printing all the 7 subsequence for the string "abc".

   Example 2:
   Input: str = "aa"
   Output: a a aa
   Explanation: Printing all the 3 subsequences for the string "aa"
*/
```

Write the above program using recurison in Java and explain it using output

Here is a Java program that uses recursion to generate all possible subsequences (the power set) of a given string.

```java
public class PowerSet {

    public static void generateSubsequences(String str) {
        generateSubsequencesHelper(str, 0, "");
    }

    private static void generateSubsequencesHelper(String str, int index, String current) {
        // Base case: if we have processed all characters, print the current subsequence
        if (index == str.length()) {
            if (!current.isEmpty()) { // To avoid printing empty string as a subsequence
                System.out.print(current + " ");
            }
            return;
        }

        // Recursive case:
        // 1. Include the current character in the subsequence
        generateSubsequencesHelper(str, index + 1, current + str.charAt(index));

        // 2. Exclude the current character from the subsequence
        generateSubsequencesHelper(str, index + 1, current);
    }

    public static void main(String[] args) {
        // Example 1
        String str1 = "abc";
        System.out.println("Input: str = \"" + str1 + "\"");
        System.out.print("Output: ");
        generateSubsequences(str1);
        System.out.println();

        // Example 2
        String str2 = "aa";
        System.out.println("Input: str = \"" + str2 + "\"");
        System.out.print("Output: ");
        generateSubsequences(str2);
        System.out.println();
    }
}
```

Explanation

1. **`generateSubsequences` Method**:

  - This is the main method that calls the recursive helper function `generateSubsequencesHelper` with an empty current subsequence (`""`) and the initial index (`0`).

2. `generateSubsequencesHelper` **Method**:
  - **Base Case**: If `index` is equal to the length of the string `str`, it means we have processed all characters. If `current` is not empty, it represents a valid subsequence, so we print it.
  - **Recursive Case**:
    - **Include the Current Character**: We add the character at `index` to `current` and recursively call the function to process the next character.
    - **Exclude the Current Character**: We do not add the character at `index` to `current` and recursively call the function to process the next character.

This recursive approach explores all possible combinations by deciding at each character whether to include it in the current subsequence or not. This process generates 2^n subsequences for a string of length n.

Example Outputs

1. **Example 1**:
  - **Input**: `str = "abc"`
  - **Output**: `a ab abc ac b bc c`
  - **Explanation**:
    - The subsequences are generated by including or excluding each character in the string.
    - The complete set of subsequences for "abc" is: `"a"`, `"ab"`, `"abc"`, `"ac"`, `"b"`, `"bc"`, `"c"`.
2. **Example 2**:
  - **Input**: `str = "aa"`
  - **Output**: `a a aa`
  - **Explanation**:
    - Since the string has duplicate characters, some subsequences will appear the same, but each combination is valid.
    - The complete set of subsequences for "aa" is: `"a"`, `"a"`, `"aa"`.

This code uses recursion and backtracking to generate all possible subsequences efficiently, considering each character in the string for inclusion or exclusion at each recursive level.

```
// Write a program to print all the SubSequences with sum k.
/*
 * Input: [1, 2, 1], Target = 2;
 * output: [[1,1], [2]]
 *
 * Input: [1, 2, 3, 2], target = 5;
 * output: [[2, 3], [3, 2], [1, 2, 2]]
 */
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion to print all subsequences of an array whose elements sum up to a given target `k`.

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

public class SubsequencesWithTargetSum {

    public static void printSubsequencesWithSum(int[] arr, int target) {
        List<Integer> current = new ArrayList<>();
        findSubsequences(arr, 0, target, current);
    }

    private static void findSubsequences(int[] arr, int index, int target, List<Integer> cur
rent) {
        // Base case: if target becomes 0, print the current subsequence
        if (target == 0) {
            System.out.println(new ArrayList<>(current));
            return;
        }

        // If index goes out of bounds or target is negative, return
        if (index >= arr.length || target < 0) {
            return;
        }

        // Include the current element in the subsequence
        current.add(arr[index]);
        findSubsequences(arr, index + 1, target - arr[index], current);

        // Backtrack: Exclude the last added element and try the next possibility
        current.remove(current.size() - 1);
        findSubsequences(arr, index + 1, target, current);
    }

    public static void main(String[] args) {
        // Example 1
        int[] arr1 = {1, 2, 1};
        int target1 = 2;
        System.out.println("Input: [1, 2, 1], Target = " + target1);
        System.out.print("Output: ");
        printSubsequencesWithSum(arr1, target1);

        // Example 2
        int[] arr2 = {1, 2, 3, 2};
        int target2 = 5;
        System.out.println("Input: [1, 2, 3, 2], Target = " + target2);
        System.out.print("Output: ");
        printSubsequencesWithSum(arr2, target2);
    }
}
```

Explanation

1. **`printSubsequencesWithSum` Method**:
    - This method initializes an empty list `current` to store the current subsequence and calls the helper method `findSubsequences`.
2. **`findSubsequences` Method**:
    - **Base Case**: If the `target` becomes `0`, it means we have found a subsequence with the desired sum. We print this subsequence and return.
    - **Out of Bounds/Negative Target Check**: If `index` goes beyond the array length or `target` is negative, we return because no valid subsequence can be formed in this path.
    - **Recursive Case**:

- **Include the Current Element**: Add the element at `arr[index]` to `current`, reduce `target` by this element's value, and recursively call `findSubsequences` for the next index.
- **Backtrack**: Remove the last element added to `current` (to explore subsequences that exclude this element), and recursively call `findSubsequences` without including the current element.

Example Outputs

1. **Example 1**:
   - **Input**: `arr = [1, 2, 1]`, `target = 2`
   - **Output**:

     ```csharp
     [1, 1]
     [2]
     ```

   - **Explanation**: The subsequences with a sum equal to `2` are `[1, 1]` and `[2]`.
2. **Example 2**:
   - **Input**: `arr = [1, 2, 3, 2]`, `target = 5`
   - **Output**:

     ```csharp
     [2, 3]
     [3, 2]
     [1, 2, 2]
     ```

   - **Explanation**: The subsequences with a sum equal to `5` are `[2, 3]`, `[3, 2]`, and `[1, 2, 2]`.

This program uses recursion and backtracking to explore all subsequences of the array, checking each subsequence's sum against the target. If a subsequence meets the target, it is printed. This approach ensures all possible subsequences are considered, including those with duplicate values.

```
/*
 * Given an integer array nums of unique elements, return all possible subsets (the power set).

  The solution set must not contain duplicate subsets. Return the solution in any order.

  Example 1:

  Input: nums = [1,2,3]
  Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
  Example 2:

  Input: nums = [0]
  Output: [[],[0]]
 */
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion to generate all possible subsets (the power set) of an array of unique integers.

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

public class PowerSetUniqueElements {

    public static List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        generateSubsets(nums, 0, new ArrayList<>(), result);
        return result;
    }

    private static void generateSubsets(int[] nums, int index, List<Integer> current, List<L
ist<Integer>> result) {
        // Base case: Add the current subset to the result
        result.add(new ArrayList<>(current));

        // Recursive case: Explore each element by including or excluding it
        for (int i = index; i < nums.length; i++) {
            // Include the current element
            current.add(nums[i]);
            generateSubsets(nums, i + 1, current, result);

            // Backtrack: remove the last element to explore subsets without it
            current.remove(current.size() - 1);
        }
    }

    public static void main(String[] args) {
        // Example 1
        int[] nums1 = {1, 2, 3};
        System.out.println("Input: nums = " + java.util.Arrays.toString(nums1));
        System.out.println("Output: " + subsets(nums1));

        // Example 2
        int[] nums2 = {0};
        System.out.println("Input: nums = " + java.util.Arrays.toString(nums2));
        System.out.println("Output: " + subsets(nums2));
    }
}
```

Explanation

1. **`subsets` Method**:
   - This method initializes an empty list `result` to store all subsets and calls the recursive helper function `generateSubsets`.
2. **`generateSubsets` Method**:
   - **Base Case**: At each call, we add the `current` subset (which can be empty or partially filled) to `result`.
   - **Recursive Case**: For each element in the array starting from `index`, we explore two possibilities:
     1. **Include the Element**: Add `nums[i]` to `current`, then recursively call `generateSubsets` with `i + 1` to avoid re-using elements.
     2. **Backtrack**: After returning from the recursive call, remove the last element from `current` to explore subsets that do not include `nums[i]`.

This approach uses backtracking to ensure all possible combinations are considered. Each subset is generated by deciding for each element whether to include it or not, leading to a total of 2^n subsets for an array of length n.

Example Outputs

1. **Example 1**:
   - **Input**: `nums = [1, 2, 3]`
   - **Output**:

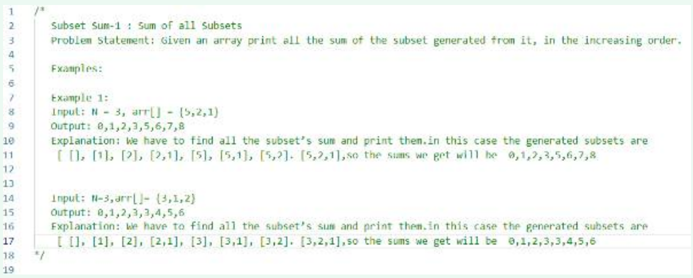
     ```css
     [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
     ```

   - **Explanation**:
     - The program generates all possible subsets by including or excluding each element in the array.
     - The subsets of `[1, 2, 3]` include:
       - The empty subset `[]`
       - Subsets with one element: `[1]`, `[2]`, `[3]`
       - Subsets with two elements: `[1, 2]`, `[1, 3]`, `[2, 3]`
       - The full subset `[1, 2, 3]`
2. **Example 2**:
   - **Input**: `nums = [0]`
   - **Output**:

     ```lua
     [[], [0]]
     ```

   - **Explanation**:
     - For a single-element array `[0]`, there are only two subsets: the empty subset `[]` and the subset containing the element `[0]`.

This code effectively generates the power set (all subsets) for any array of unique integers using recursion and backtracking. Each recursive call explores a new subset by adding the current element or skipping it, ensuring all combinations are covered.



```
/*
    Subset Sum-1 : Sum of all Subsets
    Problem Statement: Given an array print all the sum of the subset generated from it, in the increasing order.

    Examples:

    Example 1:
    Input: N = 3, arr[] = {5,2,1}
    Output: 0,1,2,3,5,6,7,8
    Explanation: We have to find all the subset's sum and print them.in this case the generated subsets are
     [ [], [1], [2], [2,1], [5], [5,1], [5,2]. [5,2,1],so the sums we get will be  0,1,2,3,5,6,7,8


    Input: N=3,arr[]= {3,1,2}
    Output: 0,1,2,3,3,4,5,6
    Explanation: We have to find all the subset's sum and print them.in this case the generated subsets are
     [ [], [1], [2], [2,1], [3], [3,1], [3,2]. [3,2,1],so the sums we get will be  0,1,2,3,3,4,5,6
*/
```

Write the above program using Brute force and recurison in Java and explain it using output

Here's a Java program that uses recursion and brute force to calculate the sum of all subsets of a given array. Each subset's sum is calculated recursively, and the results are stored in a list, which is then sorted in increasing order.

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

public class SubsetSum {

    public static List<Integer> subsetSums(int[] arr) {
        List<Integer> result = new ArrayList<>();
        calculateSubsetSums(arr, 0, 0, result);
        Collections.sort(result); // Sort the sums in increasing order
        return result;
    }

    private static void calculateSubsetSums(int[] arr, int index, int currentSum, List<Integ
er> result) {
        // Base case: If we have processed all elements, add the current sum to the result
        if (index == arr.length) {
            result.add(currentSum);
            return;
        }

        // Recursive case:
        // 1. Include the current element in the subset
        calculateSubsetSums(arr, index + 1, currentSum + arr[index], result);

        // 2. Exclude the current element from the subset
        calculateSubsetSums(arr, index + 1, currentSum, result);
    }

    public static void main(String[] args) {
        // Example 1
        int[] arr1 = {5, 2, 1};
        System.out.println("Input: arr = " + java.util.Arrays.toString(arr1));
        System.out.println("Output: " + subsetSums(arr1));

        // Example 2
        int[] arr2 = {3, 1, 2};
        System.out.println("Input: arr = " + java.util.Arrays.toString(arr2));
        System.out.println("Output: " + subsetSums(arr2));
    }
}
```

Explanation:

1. **`subsetSums` Method**:
    - This is the main method that initializes the result list and calls the recursive helper function **`calculateSubsetSums`**.
    - It also sorts the result list to ensure the sums are in increasing order before returning.
2. **`calculateSubsetSums` Method**:
    - **Base Case**: If **`index`** equals the length of the array, it means we have processed all elements in the array, and the current sum represents the sum of a subset. We add this sum to

`result`.

- **Recursive Case**:
  - **Include the Current Element**: We add the element at `arr[index]` to `currentSum` and recursively call `calculateSubsetSums` with the next index.
  - **Exclude the Current Element**: We do not add the element at `arr[index]` to `currentSum` and recursively call `calculateSubsetSums` with the next index.
- By including and excluding each element, the recursion generates all possible subsets and calculates their sums.

Example Outputs:

1. **Example 1**:
   - **Input**: `arr = [5, 2, 1]`
   - **Output**: `[0, 1, 2, 3, 5, 6, 7, 8]`
   - **Explanation**:
     - The subsets are: `[]`, `[1]`, `[2]`, `[2, 1]`, `[5]`, `[5, 1]`, `[5, 2]`, `[5, 2, 1]`.
     - Their sums are: `0`, `1`, `2`, `3`, `5`, `6`, `7`, `8`.
     - The sums are sorted in increasing order.
2. **Example 2**:
   - **Input**: `arr = [3, 1, 2]`
   - **Output**: `[0, 1, 2, 3, 3, 4, 5, 6]`
   - **Explanation**:
     - The subsets are: `[]`, `[1]`, `[2]`, `[2, 1]`, `[3]`, `[3, 1]`, `[3, 2]`, `[3, 2, 1]`.
     - Their sums are: `0`, `1`, `2`, `3`, `3`, `4`, `5`, `6`.
     - The sums are sorted in increasing order.

This code uses recursion to explore all possible subsets by deciding for each element whether to include it in the current subset or not. The approach is brute-force, as it considers every subset. The result is sorted to satisfy the requirement of printing sums in increasing order.

```
/*
    Subset - II | Print all the Unique Subsets
    Problem Statement: Given an array of integers that may contain duplicates the task is
    to return all possible subsets. Return only unique subsets and they can be in any order.

    Examples:

    Example 1:
    Input: array[] - [1,2,2]
    Output: [ [ ],[1],[1,2],[1,2,2],[2],[2,2] ]
    Explanation: We can have subsets ranging from  length 0 to 3. which are listed above.
     Also the subset [1,2] appears twice but is printed only once as we require only unique subsets.

    Input: array[] - [1]
    Output: [ [ ], [1] ]
    Explanation: Only two unique subsets are available
*/
```

Write the above program using Brute force and recurison in Java and explain it using output

Here is a Java program that uses recursion and backtracking to generate all unique subsets of an array that may contain duplicate elements. We use a brute-force approach to generate all subsets and then ensure only unique subsets are added to the result.

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

public class UniqueSubsets {

    public static List<List<Integer>> uniqueSubsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums);  // Sort the array to handle duplicates
        generateSubsets(nums, 0, new ArrayList<>(), result);
        return result;
    }

    private static void generateSubsets(int[] nums, int index, List<Integer> current, List<L
ist<Integer>> result) {
        // Add the current subset to the result (no duplicates because of backtracking)
        result.add(new ArrayList<>(current));

        // Recursive case: Explore subsets by including each element
        for (int i = index; i < nums.length; i++) {
            // Skip duplicates by checking if the current element is the same as the previou
s one
            if (i > index && nums[i] == nums[i - 1]) {
                continue;
            }

            // Include nums[i] in the current subset
            current.add(nums[i]);
            generateSubsets(nums, i + 1, current, result);  // Recurse with the next index

            // Backtrack to explore subsets without nums[i]
            current.remove(current.size() - 1);
        }
    }

    public static void main(String[] args) {
        // Example 1
        int[] nums1 = {1, 2, 2};
        System.out.println("Input: array = " + Arrays.toString(nums1));
        System.out.println("Output: " + uniqueSubsets(nums1));

        // Example 2
        int[] nums2 = {1};
        System.out.println("Input: array = " + Arrays.toString(nums2));
        System.out.println("Output: " + uniqueSubsets(nums2));
    }
}
```

Explanation

1. **`uniqueSubsets` Method**:
    - This method initializes the result list and sorts the input array `nums`. Sorting helps to identify duplicates easily and ensures that subsets are generated in a sorted order.
    - It calls the recursive helper method `generateSubsets` to generate all unique subsets.
2. **`generateSubsets` Method**:
    - **Base Case**: At each call, we add the `current` subset (which may be empty or partially filled) to `result`.
    - **Recursive Case**:
        - For each element in the array starting from `index`, we check if the element is a duplicate of the previous one. If it is, we skip it to avoid duplicate subsets.

- **Include the Current Element**: Add `nums[i]` to `current` and recursively call `generateSubsets` with the next index `i + 1`.
- **Backtrack**: After returning from the recursive call, we remove the last element added to `current` (to explore subsets without the current element).

This approach ensures that all unique subsets are generated by avoiding duplicates when the current element is the same as the previous one. The result list `result` accumulates all unique subsets.

Example Outputs

1. **Example 1**:
   - **Input**: `array = [1, 2, 2]`
   - **Output**:

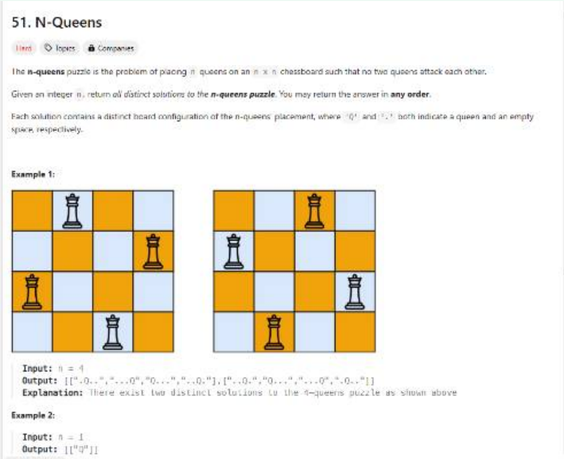
     ```css
     [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]
     ```

   - **Explanation**:
     - The subsets generated are `[]`, `[1]`, `[1, 2]`, `[1, 2, 2]`, `[2]`, `[2, 2]`.
     - The subset `[1, 2]` is included only once, as we skip duplicate values to ensure uniqueness.
2. **Example 2**:
   - **Input**: `array = [1]`
   - **Output**:

     ```lua
     [[], [1]]
     ```

   - **Explanation**:
     - For a single-element array `[1]`, there are only two subsets: the empty subset `[]` and the subset containing the element `[1]`.

This code effectively generates unique subsets of an array with possible duplicate elements using recursion and backtracking. By sorting and skipping duplicate elements, it ensures that only unique subsets are included in the result.



51. N-Queens

Hard · Topics · Companies

The **n-queens** puzzle is the problem of placing `n` queens on an `n x n` chessboard such that no two queens attack each other.

Given an integer `n`, return *all distinct solutions to the **n-queens puzzle***. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where `'Q'` and `'.'` both indicate a queen and an empty space, respectively.

Example 1:

```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above
```

Example 2:

```
Input: n = 1
Output: [["Q"]]
```

Write the above program using recurison in Java and explain it using output

Here is a Java program that uses recursion and backtracking to solve the N-Queens problem. This program finds all possible arrangements of `n` queens on an `n x n` chessboard such that no two queens attack each other.

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

public class NQueens {

    public static List<List<String>> solveNQueens(int n) {
        List<List<String>> solutions = new ArrayList<>();
        char[][] board = new char[n][n];

        // Initialize the board with empty cells
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = '.';
            }
        }

        // Start solving with the first row
        solve(solutions, board, 0, n);
        return solutions;
    }

    private static void solve(List<List<String>> solutions, char[][] board, int row, int n)
{
        // Base case: if row equals n, all queens are placed
        if (row == n) {
            solutions.add(createBoard(board));
            return;
        }

        // Try placing a queen in each column of the current row
        for (int col = 0; col < n; col++) {
            if (isSafe(board, row, col, n)) {
                board[row][col] = 'Q';  // Place queen
                solve(solutions, board, row + 1, n);  // Move to the next row
                board[row][col] = '.';  // Backtrack and remove the queen
            }
        }
    }

    // Helper method to check if a position is safe for a queen
    private static boolean isSafe(char[][] board, int row, int col, int n) {
```

```
        // Check column
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') {
                return false;
            }
        }

        // Check upper-left diagonal
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }

        // Check upper-right diagonal
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }

        return true;
    }

    // Helper method to create the board configuration as a list of strings
    private static List<String> createBoard(char[][] board) {
        List<String> result = new ArrayList<>();
        for (char[] row : board) {
            result.add(new String(row));
        }
        return result;
    }

    public static void main(String[] args) {
        // Example 1
        int n1 = 4;
        System.out.println("Solutions for n = " + n1 + ":");
        List<List<String>> solutions1 = solveNQueens(n1);
        for (List<String> solution : solutions1) {
            for (String row : solution) {
                System.out.println(row);
            }
            System.out.println();
        }

        // Example 2
        int n2 = 1;
        System.out.println("Solutions for n = " + n2 + ":");
        List<List<String>> solutions2 = solveNQueens(n2);
        for (List<String> solution : solutions2) {
            for (String row : solution) {
                System.out.println(row);
            }
            System.out.println();
        }
    }
}
```

Explanation:

1. `solveNQueens` **Method**:
    - Initializes an `n x n` chessboard filled with `'.'` (empty cells).
    - Calls the `solve` method to begin solving the N-Queens problem starting from the first row (`row = 0`).
    - Returns a list of all solutions, where each solution is a board configuration represented by a list of strings.
2. `solve` **Method**:

- **Base Case**: If `row` equals `n`, it means all queens have been placed on the board without conflicts, so we add the current board configuration to the solutions list.
- **Recursive Case**: For each column in the current row, it checks if placing a queen is safe using the `isSafe` method.
- If it is safe, it places a queen (`'Q'`) at the current cell, recursively calls `solve` for the next row, and then backtracks by removing the queen (setting the cell back to `'.'`).

3. `isSafe` **Method**:
   - Checks if it is safe to place a queen at the given position `(row, col)` by ensuring:
     - No other queen is in the same column.
     - No other queen is in the upper-left diagonal.
     - No other queen is in the upper-right diagonal.
4. `createBoard` **Method**:
   - Converts the `char[][] board` into a list of strings to represent the board configuration for each solution.

Example Outputs:

1. **Example 1**:
   - **Input**: `n = 4`
   - **Output**:

```css
.Q..
...Q
Q...
..Q.

..Q.
Q...
...Q
.Q..
```

   - **Explanation**: There are two distinct solutions for the 4-queens problem. Each solution represents a unique arrangement of 4 queens on a 4x4 board where no two queens attack each other.
2. **Example 2**:
   - **Input**: `n = 1`
   - **Output**:

```css
Q
```

   - **Explanation**: For `n = 1`, there is only one solution where a single queen is placed on a 1x1 board.

This program uses recursion and backtracking to explore all possible arrangements of queens on an `n x n` board, ensuring no two queens can attack each other. The `isSafe` method checks for conflicts, and `solve` method places queens one row at a time, backtracking as needed. The program collects and displays all distinct solutions.

```
/*
    Rat in a Maze
    Rat in a Maze

    Consider a rat placed at (0, 0) in a square matrix of order N * N. It has to reach the destination at
    (N - 1, N - 1). Find all possible paths that the rat can take to reach from source to destination.
     The directions in which the rat can move are 'U'(up), 'D'(down), 'L' (left), 'R' (right).
      Value 0 at a cell in the matrix represents that it is blocked and the rat cannot move to it
      while value 1 at a cell in the matrix represents that rat can travel through it.

    Note: In a path, no cell can be visited more than one time.

    Print the answer in lexicographical(sorted) order

    Examples:

    Example 1:

    Input:
    N = 4
    m[][] = {{1, 0, 0, 0},
            {1, 1, 0, 1},
            {1, 1, 0, 0},
            {0, 1, 1, 1}}

    Output: DDRDRR DRDDRR

    Explanation:
    The rat can reach the destination at (3, 3) from (0, 0) by two paths - DRDDRR and DDRDRR, when printed in sorted order we get DDRDRR DRDDRR.

    Example 2:

    Input: N = 2
           m[][] = {{1, 0},
                    {1, 0}}

    Output:
    No path exists and the destination cell is blocked.
*/
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion and backtracking to solve the "Rat in a Maze" problem. The program finds all possible paths from the top-left corner `(0, 0)` to the bottom-right corner `(N-1, N-1)` of an `N x N` matrix, where the rat can move in four directions: 'D' (down), 'R' (right), 'U' (up), and 'L' (left). The paths are printed in lexicographical (sorted) order.

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

public class RatInMaze {

    public static List<String> findPath(int[][] maze, int n) {
        List<String> paths = new ArrayList<>();
        if (maze[0][0] == 1) {
            solveMaze(maze, n, 0, 0, "", paths, new boolean[n][n]);
        }
        Collections.sort(paths); // Sort the paths lexicographically
        return paths;
    }

    private static void solveMaze(int[][] maze, int n, int row, int col, String path, List<S
tring> paths, boolean[][] visited) {
        // Base case: If the rat has reached the bottom-right corner
        if (row == n - 1 && col == n - 1) {
            paths.add(path);
            return;
        }

        // Mark the current cell as visited
        visited[row][col] = true;

        // Explore in lexicographical order: Down, Left, Right, Up

        // Move Down
        if (isSafe(maze, n, row + 1, col, visited)) {
            solveMaze(maze, n, row + 1, col, path + "D", paths, visited);
        }
```

```
        // Move Left
        if (isSafe(maze, n, row, col - 1, visited)) {
            solveMaze(maze, n, row, col - 1, path + "L", paths, visited);
        }

        // Move Right
        if (isSafe(maze, n, row, col + 1, visited)) {
            solveMaze(maze, n, row, col + 1, path + "R", paths, visited);
        }

        // Move Up
        if (isSafe(maze, n, row - 1, col, visited)) {
            solveMaze(maze, n, row - 1, col, path + "U", paths, visited);
        }

        // Backtrack: Unmark the current cell as visited
        visited[row][col] = false;
    }

    // Helper method to check if the cell (row, col) is safe to visit
    private static boolean isSafe(int[][] maze, int n, int row, int col, boolean[][] visite
d) {
        return row >= 0 && row < n && col >= 0 && col < n && maze[row][col] == 1 && !visited
[row][col];
    }

    public static void main(String[] args) {
        // Example 1
        int[][] maze1 = {
            {1, 0, 0, 0},
            {1, 1, 0, 1},
            {1, 1, 0, 0},
            {0, 1, 1, 1}
        };
        System.out.println("Paths for maze 1:");
        System.out.println(findPath(maze1, 4));

        // Example 2
        int[][] maze2 = {
            {1, 0},
            {1, 0}
        };
        System.out.println("Paths for maze 2:");
        System.out.println(findPath(maze2, 2));
    }
}
```

Explanation

1. **`findPath` Method**:
    - Initializes the `paths` list to store all valid paths from the start to the destination.
    - Checks if the starting cell `(0, 0)` is open (`maze[0][0] == 1`). If it is, it calls the recursive `solveMaze` function.
    - Sorts the `paths` list lexicographically to ensure the output is in the correct order.
2. **`solveMaze` Method**:
    - **Base Case**: If the rat has reached the bottom-right corner `(N-1, N-1)`, it means a complete path has been found, so `path` is added to the `paths` list.
    - Marks the current cell `(row, col)` as visited to avoid revisiting it in the current path.
    - **Recursive Case**: Tries moving the rat in four directions (Down, Left, Right, Up) in lexicographical order. For each move, it checks if the destination cell is safe using `isSafe`.
    - **Backtracking**: After exploring all possible moves from the current cell, it unmarks the cell (backtracks) to allow other paths to explore this cell.
3. **`isSafe` Method**:

- Checks if a cell `(row, col)` is within the bounds of the maze, has a value of `1` (open path), and has not been visited yet.

Example Outputs

1. **Example 1**:
    - **Input**:

```java
int[][] maze1 = {
    {1, 0, 0, 0},
    {1, 1, 0, 1},
    {1, 1, 0, 0},
    {0, 1, 1, 1}
};
```

    - **Output**:

```csharp
Paths for maze 1:
[DDRRDR, DDRDRR]
```

    - **Explanation**: The rat can reach the destination at `(3, 3)` from `(0, 0)` by two distinct paths:
        - `DDRRDR`: Down, Down, Right, Right, Down, Right
        - `DDRDRR`: Down, Down, Right, Down, Right, Right
    - The output is sorted lexicographically as required.
2. **Example 2**:
    - **Input**:

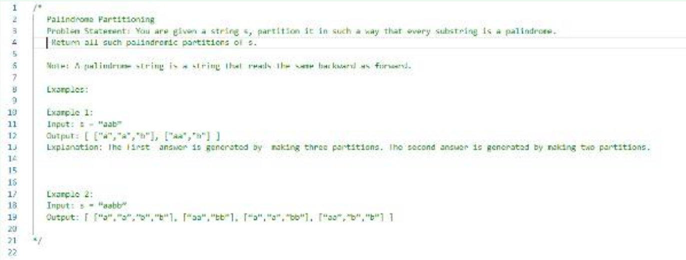
```java
int[][] maze2 = {
    {1, 0},
    {1, 0}
};
```

    - **Output**:

```less
Paths for maze 2:
[]
```

    - **Explanation**: There is no valid path from `(0, 0)` to `(1, 1)` because the destination cell `(1, 1)` is blocked. Thus, the output is an empty list.

This solution uses recursion and backtracking to explore all possible paths from the start to the destination, ensuring that each path is unique and no cell is revisited in a single path. The paths are stored in lexicographical order for easy sorting, and the solution is efficient for reasonably sized grids.



```
/*
    Palindrome Partitioning
    Problem Statement: You are given a string s, partition it in such a way that every substring is a palindrome.
    Return all such palindromic partitions of s.

    Note: A palindrome string is a string that reads the same backward as forward.

    Examples:

    Example 1:
    Input: s = "aab"
    Output: [ ["a","a","b"], ["aa","b"] ]
    Explanation: The first  answer is generated by  making three partitions. The second answer is generated by making two partitions.



    Example 2:
    Input: s = "aabb"
    Output: [ ["a","a","b","b"], ["aa","bb"], ["a","a","bb"], ["aa","b","b"] ]

*/
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion and backtracking to solve the "Palindrome Partitioning" problem. The program partitions a given string such that every substring in each partition is a palindrome, and it returns all such partitions.

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

public class PalindromePartitioning {

    public static List<List<String>> partition(String s) {
        List<List<String>> result = new ArrayList<>();
        findPartitions(s, 0, new ArrayList<>(), result);
        return result;
    }

    private static void findPartitions(String s, int start, List<String> current, List<List<
String>> result) {
        // Base case: if we have reached the end of the string, add the current partition to
the result
        if (start == s.length()) {
            result.add(new ArrayList<>(current));
            return;
        }

        // Recursive case: explore each substring starting from 'start'
        for (int end = start; end < s.length(); end++) {
            // If the substring s[start:end+1] is a palindrome
            if (isPalindrome(s, start, end)) {
                // Include this substring in the current partition
                current.add(s.substring(start, end + 1));
                // Recurse for the remaining part of the string
                findPartitions(s, end + 1, current, result);
                // Backtrack to explore other partitions
                current.remove(current.size() - 1);
            }
        }
    }

    // Helper method to check if a substring is a palindrome
    private static boolean isPalindrome(String s, int left, int right) {
        while (left < right) {
            if (s.charAt(left++) != s.charAt(right--)) {
                return false;
```

```
            }
        }
        return true;
    }

    public static void main(String[] args) {
        // Example 1
        String s1 = "aab";
        System.out.println("Input: s = \"" + s1 + "\"");
        System.out.println("Output: " + partition(s1));

        // Example 2
        String s2 = "aabb";
        System.out.println("Input: s = \"" + s2 + "\"");
        System.out.println("Output: " + partition(s2));
    }
}
```

Explanation

1. **`partition` Method**:
    - This is the main method that initializes the result list and calls the recursive helper method **`findPartitions`**.
2. **`findPartitions` Method**:
    - **Base Case**: If **`start`** equals the length of the string, it means we have processed the entire string and generated a valid partition, so we add **`current`** to the result list.
    - **Recursive Case**: For each possible substring **`s[start:end+1]`**, it checks if the substring is a palindrome using **`isPalindrome`**.
        - If it is a palindrome, it includes this substring in the **`current`** partition, then recursively calls **`findPartitions`** for the remaining substring starting from **`end + 1`**.
        - After the recursive call, it backtracks by removing the last added substring to explore other partition possibilities.
3. **`isPalindrome` Method**:
    - Checks if a substring **`s[left:right+1]`** is a palindrome by comparing characters from both ends.

Example Outputs

1. **Example 1**:
    - **Input**: **`s = "aab"`**
    - **Output**:

    ```lua
    Input: s = "aab"
    Output: [[a, a, b], [aa, b]]
    ```

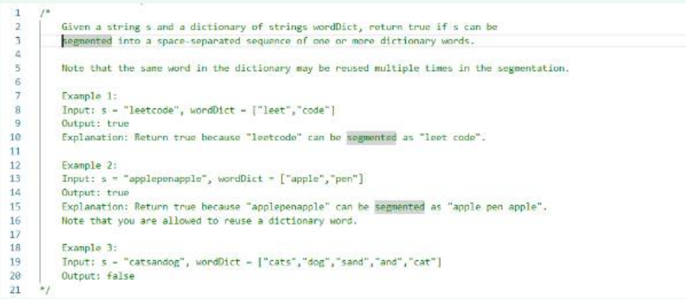
    - **Explanation**:
        - The string "aab" can be partitioned into palindromic substrings in two ways:
            - **`["a", "a", "b"]`** (three partitions, each letter is a palindrome)
            - **`["aa", "b"]`** (two partitions, "aa" is a palindrome and "b" is a palindrome)
2. **Example 2**:

- **Input**: `s = "aabb"`
- **Output**:

```lua
Input: s = "aabb"
Output: [[a, a, b, b], [a, a, bb], [aa, b, b], [aa, bb]]
```

- **Explanation**:
  - The string "aabb" can be partitioned into palindromic substrings in four ways:
    - `["a", "a", "b", "b"]` (each character is a palindrome)
    - `["a", "a", "bb"]` ("bb" is also a palindrome)
    - `["aa", "b", "b"]` ("aa" is a palindrome)
    - `["aa", "bb"]` (both "aa" and "bb" are palindromes)

This program uses recursion and backtracking to generate all possible palindromic partitions of a given string. It systematically explores each substring, checks if it’s a palindrome, and recursively builds partitions, ensuring every substring in each partition is a palindrome.



```
/*
    Given a string s and a dictionary of strings wordDict, return true if s can be
    segmented into a space-separated sequence of one or more dictionary words.

    Note that the same word in the dictionary may be reused multiple times in the segmentation.

    Example 1:
    Input: s = "leetcode", wordDict = ["leet","code"]
    Output: true
    Explanation: Return true because "leetcode" can be segmented as "leet code".

    Example 2:
    Input: s = "applepenapple", wordDict = ["apple","pen"]
    Output: true
    Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
    Note that you are allowed to reuse a dictionary word.

    Example 3:
    Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
    Output: false
*/
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion with memoization to determine if a given string `s` can be segmented into a space-separated sequence of one or more dictionary words provided in `wordDict`.

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class WordBreak {

    public static boolean wordBreak(String s, List<String> wordDict) {
        Set<String> wordSet = new HashSet<>(wordDict);
        Map<String, Boolean> memo = new HashMap<>();
        return canBreak(s, wordSet, memo);
    }

    private static boolean canBreak(String s, Set<String> wordSet, Map<String, Boolean> mem
o) {
        // Check if we have already computed the result for this substring
```

```
        if (memo.containsKey(s)) {
            return memo.get(s);
        }

        // Base case: if the string is empty, it means we successfully segmented all parts
        if (s.isEmpty()) {
            return true;
        }

        // Try every possible prefix of the string
        for (int i = 1; i <= s.length(); i++) {
            String prefix = s.substring(0, i);

            // If the prefix is in the dictionary, check if the remaining string can be segm
ented
            if (wordSet.contains(prefix)) {
                String suffix = s.substring(i);

                // Recursively check the suffix
                if (canBreak(suffix, wordSet, memo)) {
                    memo.put(s, true); // Memoize the result for the current substring
                    return true;
                }
            }
        }

        // If no valid segmentation is found, memoize and return false
        memo.put(s, false);
        return false;
    }

    public static void main(String[] args) {
        // Example 1
        String s1 = "leetcode";
        List<String> wordDict1 = List.of("leet", "code");
        System.out.println("Input: s = \"" + s1 + "\", wordDict = " + wordDict1);
        System.out.println("Output: " + wordBreak(s1, wordDict1));

        // Example 2
        String s2 = "applepenapple";
        List<String> wordDict2 = List.of("apple", "pen");
        System.out.println("Input: s = \"" + s2 + "\", wordDict = " + wordDict2);
        System.out.println("Output: " + wordBreak(s2, wordDict2));

        // Example 3
        String s3 = "catsandog";
        List<String> wordDict3 = List.of("cats", "dog", "sand", "and", "cat");
        System.out.println("Input: s = \"" + s3 + "\", wordDict = " + wordDict3);
        System.out.println("Output: " + wordBreak(s3, wordDict3));
    }
}
```

Explanation

1. **`wordBreak` Method**:
    - This method initializes a `Set` called `wordSet` with the words in `wordDict` for quick lookup.
    - It also initializes a `Map` called `memo` to store the results of previously computed substrings, which optimizes the recursion by avoiding recomputation of the same substring.
2. **`canBreak` Method** (Recursive Helper with Memoization):
    - **Base Case**: If `s` is empty, it means we have successfully segmented the entire string, so we return `true`.
    - **Memoization Check**: Before attempting to segment `s`, we check if `s` has been computed before. If so, we return the stored result.
    - **Recursive Case**:
        - For each prefix of `s`, we check if it exists in `wordSet`.

- If the prefix is found in `wordSet`, we recursively check if the remaining part of the string (suffix) can be segmented.
- If a valid segmentation is found, we store `true` in `memo` for the current substring `s` and return `true`.

- **Backtracking**:
  - If none of the prefixes lead to a valid segmentation, we store `false` in `memo` for the current substring `s` and return `false`.

Example Outputs

1. **Example 1**:
   - **Input**: `s = "leetcode"`, `wordDict = ["leet", "code"]`
   - **Output**: `true`
   - **Explanation**: The string "leetcode" can be segmented as "leet code", both of which are in the dictionary, so the function returns `true`.
2. **Example 2**:
   - **Input**: `s = "applepenapple"`, `wordDict = ["apple", "pen"]`
   - **Output**: `true`
   - **Explanation**: The string "applepenapple" can be segmented as "apple pen apple", all of which are in the dictionary, so the function returns `true`.
3. **Example 3**:
   - **Input**: `s = "catsandog"`, `wordDict = ["cats", "dog", "sand", "and", "cat"]`
   - **Output**: `false`
   - **Explanation**: There is no way to segment "catsandog" such that each part is in the dictionary, so the function returns `false`.

This solution uses recursion with memoization to efficiently check if a given string can be segmented using words from a dictionary. The memoization avoids repeated work by storing the results for previously computed substrings, making the solution faster for larger inputs.

```
/*
Word Search - Leetcode
Given an m x n grid of characters board and a string word, return true if the word exists in the grid.
 The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or
 vertically neighboring. The same letter cell may not be used more than once.

Examples:

Example 1:
Input:
[
["A", "B", "C", "E"],
["S", "F", "C", "S"],
["A", "D", "E", "E"]
]
word = "ABCCED"
Output: true
Explanation: We can easily find the given word in the matrix.

Example 2:
Input:
[
["A", "B", "C", "E"],
["S", "F", "C", "S"],
["A", "D", "E", "E"]
]
word = "ABCB"
Output: false
Explanation:  There is no such word in the given matrix.
*/
```

Write the above program using recurison in Java and explain it using output

Here's a Java program that uses recursion and backtracking to solve the "Word Search" problem. The program checks if a given word can be constructed in an `m x n` grid of characters by sequentially adjacent cells (horizontally or vertically). Each cell can only be used once in constructing the word.

```java
public class WordSearch {

    public static boolean exist(char[][] board, String word) {
        int rows = board.length;
        int cols = board[0].length;

        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < cols; col++) {
                // Start searching from each cell
                if (dfs(board, word, row, col, 0)) {
                    return true;
                }
            }
        }
        return false;
    }

    private static boolean dfs(char[][] board, String word, int row, int col, int index) {
        // Base case: If we have matched the entire word
        if (index == word.length()) {
            return true;
        }

        // Boundary and character match check
        if (row < 0 || row >= board.length || col < 0 || col >= board[0].length || board[ro
w][col] != word.charAt(index)) {
            return false;
        }

        // Mark the current cell as visited by temporarily modifying it
        char temp = board[row][col];
        board[row][col] = '#';

        // Explore all four possible directions (down, up, right, left)
        boolean found = dfs(board, word, row + 1, col, index + 1) || // Down
                        dfs(board, word, row - 1, col, index + 1) || // Up
                        dfs(board, word, row, col + 1, index + 1) || // Right
                        dfs(board, word, row, col - 1, index + 1);   // Left

        // Backtrack by restoring the cell
        board[row][col] = temp;

        return found;
    }

    public static void main(String[] args) {
        // Example 1
        char[][] board1 = {
            {'A', 'B', 'C', 'E'},
            {'S', 'F', 'C', 'S'},
            {'A', 'D', 'E', 'E'}
        };
        String word1 = "ABCCED";
        System.out.println("Input: word = \"" + word1 + "\"");
        System.out.println("Output: " + exist(board1, word1)); // Expected: true

        // Example 2
        char[][] board2 = {
            {'A', 'B', 'C', 'E'},
            {'S', 'F', 'C', 'S'},
            {'A', 'D', 'E', 'E'}
        };
        String word2 = "ABCB";
        System.out.println("Input: word = \"" + word2 + "\"");
        System.out.println("Output: " + exist(board2, word2)); // Expected: false
```

```
        }
    }
```

Explanation

1. `` `exist` `` **Method**:
   - This is the main method that iterates over each cell in the `` `board` ``. For each cell, it calls the recursive `` `dfs` `` method to check if the word can be constructed starting from that cell.
   - If any `` `dfs` `` call returns `` `true` ``, it means the word exists in the grid, and the function immediately returns `` `true` ``. If no cell can construct the word, it returns `` `false` ``.
2. `` `dfs` `` **Method** (Recursive Depth-First Search):
   - **Base Case**: If `` `index` `` is equal to the length of `` `word` ``, it means we have matched all characters in the word, so we return `` `true` ``.
   - **Boundary and Character Check**: If the current cell is out of bounds or does not match the character at `` `word[index]` ``, it returns `` `false` ``.
   - **Mark Cell as Visited**: Temporarily change the cell's value to `` `#` `` to mark it as visited. This prevents revisiting the same cell in the current search path.
   - **Explore in Four Directions**:
     - It recursively checks the four possible directions: down, up, right, and left.
   - **Backtrack**: After exploring all directions, it restores the cell to its original character to allow it to be used in other potential paths.
3. **Backtracking**:
   - By marking a cell as visited and then restoring it after the recursive calls, the algorithm ensures that each cell is used only once in a single path while allowing it to be reused in different paths.

Example Outputs

1. **Example 1**:
   - **Input**:

     ```java
     board = {
         {'A', 'B', 'C', 'E'},
         {'S', 'F', 'C', 'S'},
         {'A', 'D', 'E', 'E'}
     };
     word = "ABCCED";
     ```

   - **Output**: `` `true` ``
   - **Explanation**: The word "ABCCED" can be constructed by starting at `` `(0,0)` `` and following the sequence of cells to `` `(2,1)` ``. The function returns `` `true` ``.
2. **Example 2**:
   - **Input**:

```java
java


  board = {
      {'A', 'B', 'C', 'E'},
      {'S', 'F', 'C', 'S'},
      {'A', 'D', 'E', 'E'}
  };
  word = "ABCB";
```

  - **Output**: `false`
  - **Explanation**: The word "ABCB" cannot be constructed because the cell containing 'B' would need to be reused, which is not allowed. Thus, the function returns `false`.

This solution uses recursion and backtracking to explore all possible paths from each starting cell in the grid, marking cells as visited temporarily to avoid revisiting within the same path. The algorithm checks all possible paths to see if they match the given word.