Q0: Recursion on Strings

1. Problem Understanding

- Recursion on strings involves processing one character at a time, reducing the string in each call.
- Each recursive step handles a smaller substring (like s.substring(1) or index increment).
- Common operations:
  - Traversal / Printing
  - Reversal
  - Character search or count
  - Subsequence / Subset generation
  - Permutations
  - Encoding or decoding patterns

---

2. Constraints

- String length = n
- String is immutable → each operation may create new substrings
- Base case: when i == n or s.length() == 0
- Avoid excessive substring creation for efficiency

---

3. Edge Cases

- Empty string ""
- Single character strings
- Repeated characters (for permutations)
- Case sensitivity (e.g., ‘A’ vs ‘a’)
- Palindromic strings

---

4. Examples

```
Input: "abc" → Output (print): a b c

Input: "abc" → Reverse: "cba"

Input: "abc" → Subsets: ["", "a", "b", "c", "ab", "ac", "bc", "abc"]

Input: "abc" → Permutations: ["abc", "acb", "bac", "bca", "cab", "cba"]
```

---

5. Approaches

Approach 1: Character Processing (Traversal)

Idea:

- Process one character and move forward.

Java Code:

```
void process(String s, int i) {
    if (i == s.length()) return;
    System.out.print(s.charAt(i) + " ");
    process(s, i + 1);
}
```

Complexity (Time & Space):

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

Approach 2: Reverse a String

Idea:

- Process from end to start recursively.

Java Code:

```
String reverse(String s) {
    if (s.length() == 0) return "";
    return reverse(s.substring(1)) + s.charAt(0);
}
```

Complexity (Time & Space):

- Time: $O(n^2)$ → due to substring creation
- Space: O(n)
- ☑ Optimization: use character array instead of substring for O(n).

Approach 3: Count Characters

Idea:

- Count characters recursively using index.

Java Code:

```
int count(String s, int i) {
    if (i == s.length()) return 0;
```

```java
        return 1 + count(s, i + 1);
    }
```

Complexity (Time & Space):

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

Approach 4: Count Specific Character Occurrences

Idea:

- Increment count when a match is found.

Java Code:

```java
int countChar(String s, int i, char ch) {
    if (i == s.length()) return 0;
    int count = (s.charAt(i) == ch) ? 1 : 0;
    return count + countChar(s, i + 1, ch);
}
```

Complexity (Time & Space):

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

Approach 5: Remove Specific Character

Idea:

- Build new string excluding given character.

Java Code:

```java
String removeChar(String s, char ch, int i) {
    if (i == s.length()) return "";
    if (s.charAt(i) == ch) return removeChar(s, ch, i + 1);
    return s.charAt(i) + removeChar(s, ch, i + 1);
}
```

Complexity (Time & Space):

- Time: $O(n^2)$ due to string concatenation
- Space: O(n)
- ☑ Use StringBuilder for O(n) optimized version.

Approach 6: Check Palindrome

Idea:

- Compare start and end characters recursively.

Java Code:

```
boolean isPalindrome(String s, int l, int r) {
    if (l >= r) return true;
    if (s.charAt(l) != s.charAt(r)) return false;
    return isPalindrome(s, l + 1, r - 1);
}
```

Complexity (Time & Space):

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

Approach 7: Subsequence Generation

Idea:

- For each character → include or exclude.

Java Code:

```
void subsequences(String s, String ans, int i) {
    if (i == s.length()) {
        System.out.println(ans);
        return;
    }
    subsequences(s, ans + s.charAt(i), i + 1);  // include
    subsequences(s, ans, i + 1);                // exclude
}
```

Complexity (Time & Space):

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

Approach 8: Subsets (Same as subsequences but conceptual)

Idea:

- Treat each character as a choice (include/exclude).

Steps:

- void subsets(String s, String curr, int i) {
    - if (i == s.length()) {

        - System.out.println(curr);
        - return;
    - }
    - subsets(s, curr + s.charAt(i), i + 1);
    - subsets(s, curr, i + 1);
- }

Complexity (Time & Space):

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

Approach 9: Permutations

Idea:

- Fix one character and recursively permute the rest.

Java Code:

```
void permutations(String s, String ans) {
    if (s.length() == 0) {
        System.out.println(ans);
        return;
    }
    for (int i = 0; i < s.length(); i++) {
        char ch = s.charAt(i);
        String ros = s.substring(0, i) + s.substring(i + 1);
        permutations(ros, ans + ch);
    }
}
```

Complexity (Time & Space):

- Time: O(n × n!)
- Space: O(n)

Approach 10: Replace Character

Idea:

- Replace all occurrences of a target character.

Java Code:

```
String replaceChar(String s, char oldChar, char newChar, int i) {
    if (i == s.length()) return "";
    char curr = s.charAt(i);
    if (curr == oldChar) curr = newChar;
```

```
    return curr + replaceChar(s, oldChar, newChar, i + 1);
}
```

Complexity (Time & Space):

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

Approach 11: Remove Consecutive Duplicates

Idea:

- Skip same consecutive characters.

Java Code:

```
String removeDuplicates(String s, int i) {
    if (i == s.length() - 1) return s.charAt(i) + "";
    String next = removeDuplicates(s, i + 1);
    if (s.charAt(i) == s.charAt(i + 1)) return next;
    return s.charAt(i) + next;
}
```

Complexity (Time & Space):

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

Approach 12: String to Integer (Parse Recursively)

Idea:

- Convert each char to number from left to right.

Java Code:

```
int stringToInt(String s, int i) {
    if (i == s.length()) return 0;
    int num = s.charAt(i) - '0';
    return num * (int)Math.pow(10, s.length() - i - 1) + stringToInt(s, i + 1);
}
```

Complexity (Time & Space):

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

Approach 13: Encoding / Decoding (Advanced)

Idea:

- Example: "1 → a, 2 → b ..." → Decode numeric strings to alphabets.

Java Code:

```
void encode(String s, String ans) {
    if (s.length() == 0) {
        System.out.println(ans);
        return;
    }
    char ch1 = (char) ('a' + (s.charAt(0) - '1'));
    encode(s.substring(1), ans + ch1);
    if (s.length() > 1) {
        int num = Integer.parseInt(s.substring(0, 2));
        if (num <= 26) {
            char ch2 = (char) ('a' + num - 1);
            encode(s.substring(2), ans + ch2);
        }
    }
}
```

Complexity (Time & Space):

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

---

6. Justification / Proof of Optimality

- Recursion simplifies complex string manipulations.
- Reduces looping logic into smaller subproblems.
- Backbone for backtracking & combinatorial string problems.

---

7. Variants / Follow-Ups

- String recursion with multiple strings (comparisons)
- Pattern-based recursion (decoding, parentheses matching)
- Using character arrays to optimize performance

---

8. Tips & Observations

- Always use base case like i == s.length() or s.length() == 0.
- Be careful with string immutability (avoid excessive concatenation).
- For large inputs, prefer StringBuilder to optimize performance.
- Visualize recursive stack for better debugging.
- Backtracking problems on strings often follow include/exclude pattern.

Q70: No X

---

1. Problem Understanding

- You are given a string s.
- You must remove all characters 'x' recursively.
- Return or print the new string after all 'x' are removed.

---

2. Constraints

- 1 <= s.length() <= 10^4
- String may contain lowercase alphabets including 'x'.

---

3. Edge Cases

- String has no 'x' → return same string.
- String is all 'x' → return empty string.
- 'x' appears at start, middle, or end — handle all positions.

---

4. Examples

```
Input → "xaaax"
Output → "aaa"
```

---

5. Approaches

Approach 1: Expanded Recursive Template (for clarity)

Idea:

- Work on the first character and solve the rest of the string recursively.

Steps:

- Base case: if string is empty → return "".
- Extract first char.
- Recurse for rest of string.
- Combine results:
- If char == 'x' → skip it.
- Else → append it to result.

Java Code:

```
static String removeX(String s) {
    if (s.length() == 0) return "";
    char ch = s.charAt(0);
    String smallAns = removeX(s.substring(1));
    if (ch == 'x') return smallAns;
    else return ch + smallAns;
}

removeX("axbxc")
|
├── ch = 'a'
|   └── smallAns = removeX("xbxc")
|         |
|         ├── ch = 'x'
|         |   └── smallAns = removeX("bxc")
|         |         |
|         |         ├── ch = 'b'
|         |         |   └── smallAns = removeX("xc")
|         |         |         |
|         |         |         ├── ch = 'x'
|         |         |         |   └── smallAns = removeX("c")
|         |         |         |         |
|         |         |         |         ├── ch = 'c'
|         |         |         |         |   └── smallAns = removeX("")
|         |         |         |         |         Base case → return ""
|         |         |         |         └── return 'c' + "" → "c"
|         |         |         └── ch == 'x' → skip 'x' → return "c"
|         |         └── return 'b' + "c" → "bc"
|         └── ch == 'x' → skip 'x' → return "bc"
└── return 'a' + "bc" → "abc"
☑ Final Answer → "abc"
```

Complexity (Time & Space):

- Time Complexity: $O(n^2)$ (due to substring copying in each call)
- Space Complexity: O(n) (recursion stack)

Approach 2: Concise Recursive Template (for contests/interviews)

Idea:

- Same as above but written compactly in one return chain.

Java Code:

```
static String noX(String s) {
    if (s.length() == 0) return "";
    if (s.charAt(0) == 'x') return noX(s.substring(1));
    return s.charAt(0) + noX(s.substring(1));
}
```

Complexity (Time & Space):

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

---

6. Justification / Proof of Optimality

- Both methods are equivalent — Approach 1 is great for concept building,
- Approach 2 is ideal for fast coding once the pattern is familiar.

---

7. Tips & Observations

- Always check the first character, then recurse on the remaining.
- Avoid + concatenation in large strings → use StringBuilder for optimization.
- Similar pattern used in:
- "Replace Pi with 3.14"
- "Remove duplicates"
- "Replace character in string"

Q71: Keypad Combination

1. Problem Understanding

- Given a string str consisting of digits (0–9), print all possible combinations of letters corresponding to each digit based on a phone keypad mapping.
- Each digit maps to a specific set of characters.
- For example:
- 0 → .;
- 1 → abc
- 2 → def
- 3 → ghi
- 4 → jkl
- 5 → mno
- 6 → pqrs
- 7 → tu
- 8 → vwx
- 9 → yz
- We must generate all possible strings that could be formed by pressing the given keys in sequence.

---

2. Constraints

- 0 <= str.length() <= 10

- Input string contains digits only.

---

3. Edge Cases

- Empty input string → should return nothing (no output).
- Digits like 0 or 1 → still must map correctly using given mapping.
- Long strings (length ≥ 10) → handle via recursion efficiently.

---

4. Examples

```
Input:
78
Output:
tv
tw
tx
uv
uw
ux
```

---

5. Approaches

Approach 1: Recursive Expansion (Character Mapping + Combination)

Idea:

- For each digit, get all possible characters it can represent.
- Recursively solve for the remaining string and append each possible combination.

Steps:

- Base Case → If str is empty, print "" (an empty string as one valid combination).
- Get first digit (e.g., '7' → "tu").
- Recursively call for the remaining substring.
- Combine each character of the current digit with all strings returned from smaller problem.

Java Code:

```
import java.util.*;

public class Main {
    static String[] codes = {".;", "abc", "def", "ghi", "jkl", "mno", "pqrs",
"tu", "vwx", "yz"};

    public static void printKPC(String str, String ans) {
        // Base case
```

```
        if (str.length() == 0) {
            System.out.println(ans);
            return;
        }

        // Current digit
        char ch = str.charAt(0);
        String code = codes[ch - '0'];

        // Smaller problem
        String ros = str.substring(1);

        // Combine current digit's characters with recursive results
        for (int i = 0; i < code.length(); i++) {
            printKPC(ros, ans + code.charAt(i));
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String str = sc.next();
        printKPC(str, "");
    }
}
printKPC("23", "")
│
├── ch = '2' → code = "def"
│   ├── i=0 → 'd' → printKPC("3", "d")
│   │           ├── ch = '3' → code = "ghi"
│   │           ├── 'g' → printKPC("", "dg") → prints "dg"
│   │           ├── 'h' → printKPC("", "dh") → prints "dh"
│   │           └── 'i' → printKPC("", "di") → prints "di"
│   │
│   ├── i=1 → 'e' → printKPC("3", "e")
│   │           ├── 'g' → printKPC("", "eg") → prints "eg"
│   │           ├── 'h' → printKPC("", "eh") → prints "eh"
│   │           └── 'i' → printKPC("", "ei") → prints "ei"
│   │
│   └── i=2 → 'f' → printKPC("3", "f")
│               ├── 'g' → printKPC("", "fg") → prints "fg"
│               ├── 'h' → printKPC("", "fh") → prints "fh"
│               └── 'i' → printKPC("", "fi") → prints "fi"
│
Printed Output
dg
dh
di
eg
eh
ei
fg
fh
fi
```

Complexity (Time & Space):

- Time Complexity:
- $O(k^n)$ — where k is average number of characters per key (max 4), and n is length of input string.
- Each character can branch into multiple recursive calls.
- Space Complexity:
- O(n) — recursion depth (stack space).

Approach 2: Concise Recursive Template (Contest Friendly)

Java Code:

```
static String[] codes = {".;", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tu",
"vwx", "yz"};

static void printKPC(String str, String ans) {
    if (str.isEmpty()) {
        System.out.println(ans);
        return;
    }
    for (char ch : codes[str.charAt(0) - '0'].toCharArray())
        printKPC(str.substring(1), ans + ch);
}
```

Complexity (Time & Space):

- Time Complexity:
- $O(k^n)$ — where k is average number of characters per key (max 4), and n is length of input string.
- Each character can branch into multiple recursive calls.
- Space Complexity:
- O(n) — recursion depth (stack space).

---

6. Justification / Proof of Optimality

- Base case handles termination cleanly.
- Each recursion level branches according to key mapping — pure combinatorial recursion.
- Common recursion pattern: “one choice per character → recurse on the rest.”

---

7. Variants / Follow-Ups

- Return list of all combinations instead of printing them.
- Add support for invalid digits.
- Lexicographical sorting of results.

---

8. Tips & Observations

- Whenever the problem says “print all combinations” or “all possible outputs”, think recursion/backtracking.
- The number of recursive calls = product of number of mappings per digit.
- The base case (empty string) is crucial — it marks one complete combination.
- Use String[] codes array globally to avoid repeated creation in each call.
- The pattern here is similar to subsequence generation → choose one char per level and recurse.
- For interviews: this problem tests your recursive tree visualization — one node per digit, each branch per possible letter.

Q72: Print Stair Paths

---

1. Problem Understanding

- You are given a number n representing the number of stairs.
- You start from the top (or equivalently, at n) and want to reach the ground (0).
- You can jump 1, 2, or 3 steps at a time.
- You must print all possible paths (as strings) showing which jumps were taken.
- Each path represents one valid sequence of jumps that reaches exactly 0.

---

2. Constraints

- $0 \le n \le 10$
- Output can grow exponentially, but within limits for small n.

---

3. Edge Cases

- If n == 0: You’re already on the ground → print "" (empty path).
- If n < 0: Invalid jump → don’t print anything.

---

4. Examples

```
Input
3

Output
111
12
21
3

Explanation:
Paths →

1 + 1 + 1
```

```
1 + 2

2 + 1

3
```

---

5. Approaches

Approach 1: Recursive Backtracking (Printing Paths Directly)

Idea:

- At each stair n, you can jump 1, 2, or 3 steps.
- → Recursively explore all paths by appending the jump (1, 2, or 3) to the current path string.

Steps:

- If n == 0 → print the path (you've reached the ground).
- If n < 0 → invalid move, return.
- Make recursive calls for:
  - printStairPaths(n-1, path + "1")
  - printStairPaths(n-2, path + "2")
  - printStairPaths(n-3, path + "3")

Java Code:

```
public static void printStairPaths(int n, String path) {
        if (n == 0) {
            System.out.println(path);
            return;
        }
        if (n < 0) return;

        printStairPaths(n - 1, path + "1");
        printStairPaths(n - 2, path + "2");
        printStairPaths(n - 3, path + "3");
    }

    printStairPaths(3, "")
├── take 1 step → printStairPaths(2, "1")
│    ├── take 1 step → printStairPaths(1, "11")
│    │    ├── take 1 step → printStairPaths(0, "111") → prints "111"
│    │    ├── take 2 steps → printStairPaths(-1, "112") → invalid
│    │    └── take 3 steps → printStairPaths(-2, "113") → invalid
│    │
│    ├── take 2 steps → printStairPaths(0, "12") → prints "12"
│    └── take 3 steps → printStairPaths(-1, "13") → invalid
│
├── take 2 steps → printStairPaths(1, "2")
```

```
│   ├── take 1 step → printStairPaths(0, "21") → prints "21"
│   ├── take 2 steps → printStairPaths(-1, "22") → invalid
│   └── take 3 steps → printStairPaths(-2, "23") → invalid
│
└── take 3 steps → printStairPaths(0, "3") → prints "3"
```

Complexity (Time & Space):

- 🧮 Time Complexity:
- Each call makes up to 3 recursive calls → roughly $O(3^n)$
- Because each stair can branch into 3 possible paths.
- 💾 Space Complexity:
- O(n) for recursion stack (maximum depth = n).

---

6. Justification / Proof of Optimality

- Recursion explores all combinations of jumps.
- Base case ensures only valid paths reaching exactly 0 are printed.

---

7. Variants / Follow-Ups

- Count total paths instead of printing.
- Store all paths in a list and return.
- Change jump range (e.g., 1 or 2 only).

---

8. Tips & Observations

- Pattern: whenever we explore all possible moves from a given state, recursion with multiple branches (like this) works best.
- This is similar to "staircase problem" or "dice roll to target" problems.
- If asked to count paths instead of printing, replace print with return count.

Q73: Number of ways to form Natural Number using Recursion

---

1. Problem Understanding

- You need to find how many ways an integer N can be represented as the sum of unique natural numbers.
- Order does not matter (e.g., 1+2+3 and 3+2+1 are the same).

---

2. Constraints

- 0≤N≤120
- Natural numbers: 1,2,3,...
- Each number can be used at most once.

---

3. Edge Cases

- N=0: 1 way (empty sum).
- N<0: 0 ways (invalid sum).
- N=1: 1 way (1 itself).

---

4. Examples

```
Input:
6
Output:
4
Explanation:
6 = (1+2+3), (1+5), (2+4), (6)
```

---

5. Approaches

Approach 1: Recursion with Backtracking

Idea:

- At every step, decide whether to include the current number or not.

Steps:

- Maintain a variable curr = current natural number to consider.
- Base Cases:
    - If n == 0, return 1 (valid combination found).
    - If n < 0, return 0 (invalid sum).
    - If curr > n, return 0 (no more numbers left to use).
- Recursive Cases:
    - Include curr: call for n - curr, curr + 1
    - Exclude curr: call for n, curr + 1
- Add both results.

Java Code:

```java
int countWays(int n, int curr) {
    if (n == 0) return 1;
```

```
    if (n < 0 || curr > n) return 0;

    // include current number + exclude current number
    return countWays(n - curr, curr + 1) + countWays(n, curr + 1);
}
countWays(4,1)
├── include 1 → countWays(3,2)
│   ├── include 2 → countWays(1,3)
│   │   ├── include 3 → countWays(-2,4) → 0
│   │   └── exclude 3 → countWays(1,4)
│   │       ├── include 4 → countWays(-3,5) → 0
│   │       └── exclude 4 → countWays(1,5) → 0
│   │      → total = 0
│   │
│   └── exclude 2 → countWays(3,3)
│       ├── include 3 → countWays(0,4) → 1 ☑
│       └── exclude 3 → countWays(3,4)
│           ├── include 4 → countWays(-1,5) → 0
│           └── exclude 4 → countWays(3,5) → 0
│          → total = 0
│      → total = 1
│  → total = 1
│
└── exclude 1 → countWays(4,2)
    ├── include 2 → countWays(2,3)
    │   ├── include 3 → countWays(-1,4) → 0
    │   └── exclude 3 → countWays(2,4)
    │       ├── include 4 → countWays(-2,5) → 0
    │       └── exclude 4 → countWays(2,5) → 0
    │      → total = 0
    │  → total = 0
    │
    └── exclude 2 → countWays(4,3)
        ├── include 3 → countWays(1,4)
        │   ├── include 4 → countWays(-3,5) → 0
        │   └── exclude 4 → countWays(1,5) → 0
        │  → total = 0
        └── exclude 3 → countWays(4,4)
            ├── include 4 → countWays(0,5) → 1 ☑
            └── exclude 4 → countWays(4,5) → 0
           → total = 1
       → total = 1
   → total = 1
→ FINAL TOTAL = 2
```

Complexity (Time & Space):

- Time Complexity: O(2^N) — each number has two choices (include/exclude).
- Space Complexity: O(N) — recursion depth proportional to N.

---

6. Justification / Proof of Optimality

- Each recursive branch explores one subset of natural numbers that sum to N.
- Avoids repetition because each natural number is considered only once.

---

7. Variants / Follow-Ups

- Return actual combinations: instead of counting, store lists of numbers forming N.
- Allow repetition of numbers: modify recursion to allow including the same number again.
- Use memoization/DP to optimize from O(2^N) to O(N^2).
- Find combinations with exactly K numbers: add extra parameter to track count.
- Subset sum for array input: generalizes this approach to arbitrary arrays.

---

8. Tips & Observations

- This is similar to the Subset Sum Problem but with numbers 1..N.
- You can optimize it using memoization if needed (DP).
- The recursion tree grows exponentially, but N ≤ 120 makes it feasible for moderate inputs.

Q74: Climbing Stairs

1. Problem Understanding

- You are climbing n steps.
- Each move: climb 1 step or 2 steps.
- Find number of distinct ways to reach the top.

---

2. Constraints

- 1 <= n <= 45
- Only 1 or 2 steps at a time.

---

3. Examples

```
n = 2 → (1+1), (2) → 2 ways

n = 3 → (1+1+1), (1+2), (2+1) → 3 ways
```

---

4. Approaches

Approach 1: Recursive (Brute Force)

Idea:

- Number of ways to reach step n = ways to reach n-1 + ways to reach n-2.
- n == 0 → return 1 (reached top)
- n < 0 → return 0 (invalid path)

Java Code:

```
public int ClimbingStairs(int n) {
    if (n == 0) return 1;
    if (n < 0) return 0;
    return ClimbingStairs(n - 1) + ClimbingStairs(n - 2);
}
ClimbingStairs(3)
      /                    \
ClimbingStairs(2)       ClimbingStairs(1)
     /       \                 \
ClimbingStairs(1)  ClimbingStairs(0)   (returns 1)
     |                  |
   (returns 1)        (returns 1)

→ From ClimbingStairs(2): 1 + 1 = 2
→ From ClimbingStairs(3): 2 + 1 = 3
Output
3
```

Complexity (Time & Space):

- Time Complexity: O(2^n)
- Space Complexity: O(n) (recursion stack)

Approach 2: Recursive + Memoization (Top-Down DP)

Idea:

- Store results for each n to avoid recomputation.

Java Code:

```
public int ClimbingStairsMemo(int n, int[] dp) {
    if (n == 0) return 1;
    if (n < 0) return 0;
    if (dp[n] != -1) return dp[n];
    dp[n] = ClimbingStairsMemo(n-1, dp) + ClimbingStairsMemo(n-2, dp);
    return dp[n];
}
```

Complexity (Time & Space):

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

Approach 3: Iterative DP (Bottom-Up)

Idea:

- Build dp[i] = dp[i-1] + dp[i-2] for all i = 2 to n.

Java Code:

```
public int ClimbingStairsDP(int n) {
    if (n == 1) return 1;
    int[] dp = new int[n+1];
    dp[0] = 1; dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

Complexity (Time & Space):

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

Approach 4: Optimized Iterative (Space O(1))

Idea:

- Only keep last two results.

Java Code:

```
public int ClimbingStairsOptimized(int n) {
    if (n == 1) return 1;
    int a = 1, b = 1;
    for (int i = 2; i <= n; i++) {
        int temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

Complexity (Time & Space):

- Time Complexity: O(n)
- Space Complexity: O(1)

5. Justification / Proof of Optimality

- Recursive solution works because each step branches into 1-step and 2-step moves.
- Total ways = sum of ways from n-1 and n-2 → Fibonacci sequence.

---

6. Variants / Follow-Ups

- Climb with 1, 2, 3 steps at a time.
- Steps with forbidden steps.
- Minimum steps instead of counting ways.

---

7. Tips & Observations

- Recursive brute force is easy but slow → use memoization/DP.
- Recognize Fibonacci pattern.
- DP can be implemented top-down or bottom-up.

Q75: Print all subsequences of a string

1. Problem Understanding

- We are given a string str, and we need to print all possible subsequences (not substrings).
- A subsequence is formed by including or excluding each character while maintaining the order.

---

2. Constraints

- 1 ≤ str.length() ≤ 15
- Total possible subsequences = 2^n

---

3. Edge Cases

- Empty string → only one subsequence: ""
- Repeated characters → subsequences may not be unique.
- Long string (n > 20) → exponential output, recursion may overflow.

---

4. Examples

```
Input:
abc
Output:
abc ab ac a bc b c
```

```
Justification
Every subsequence is a unique combination of including/excluding characters in
order.
The recursion tree for "abc" looks like:

                 ""
           /          \
         "a"           ""
        /   \        /   \
    "ab"    "a"    "b"     ""
    / \     / \   / \    / \
"abc" "ab" "ac" "a" "bc" "b" "c" ""
```

---

5. Approaches

Approach 1: With Index Parameter

Idea:

- At each index, we have two choices:
  - Include the current character.
  - Exclude the current character.
- This branching generates all possible subsequences.

Steps:

- Base Case: if i == str.length(), print the accumulated string ans.
- Recursive Case:
- Include str.charAt(i) in ans → call f(str, i + 1, ans + str.charAt(i))
- Exclude str.charAt(i) → call f(str, i + 1, ans)

Java Code:

```java
void printSubsequences(String str, int i, String ans) {
    if (i == str.length()) {
        System.out.print(ans + " ");
        return;
    }

    // include
    printSubsequences(str, i + 1, ans + str.charAt(i));
    // exclude
    printSubsequences(str, i + 1, ans);
}
Call: printSubsequences(str, 0, "")
```

Complexity (Time & Space):

- Time Complexity: $O(2^n)$ → each character has 2 choices
- Space Complexity: $O(n)$ → recursion stack

Approach 2: Without Using Index Parameter

Idea:

- Instead of tracking index, break the string from the front:
- At each step, take the first character and work on the remaining substring.

Java Code:

```
void printSubsequences(String str, String ans) {
    if (str.isEmpty()) {
        System.out.print(ans + " ");
        return;
    }

    char ch = str.charAt(0);
    String rest = str.substring(1);

    // include
    printSubsequences(rest, ans + ch);
    // exclude
    printSubsequences(rest, ans);
}
▨ Call: printSubsequences(str, "")
```

Complexity (Time & Space):

- Time Complexity: $O(2^n)$
- Space Complexity: $O(n)$
- Both versions have identical performance.
- Difference: this one uses substring decomposition instead of index tracking.

---

6. Variants / Follow-Ups

- Return subsequences as a List: return ArrayList instead of printing.
- Count total subsequences: return int instead of printing.
- Generate subsequences with ASCII values: also include ASCII codes of characters (useful in recursion practice).

---

7. Tips & Observations

- This is a 2-branch recursion pattern: include/exclude.
- Number of subsequences = 2^n.
- Often used as a base template for subset, power set, or combination problems.
- In backtracking, this is one of the core "choice-making" structures.

Q78: String Permutations

1. Problem Understanding

- You are given a string str, and you need to print all possible permutations of its characters.
- Each character must appear exactly once per permutation, and all permutations should be unique and printed in lexicographic order.

2. Constraints

- 1 ≤ |str| ≤ 5
- Small enough for recursion (O(n!) total permutations).

3. Edge Cases

- Empty string → no output
- String with duplicates → ensure uniqueness (e.g., "aab")
- Lexicographic order → sort before generating permutations

4. Examples

```
Input
abc
Output
abc
acb
bac
bca
cab
cba
```

5. Approaches

Approach 1: Simple Recursion (No Duplicates)

Idea:

- At each step:
- Pick one character ch
- Recurse on the remaining string ros
- Add ch to the current answer ans

Steps:

- Base case: If str is empty → print ans
- For every character ch in str:
- Remove it → get ros
- Recurse with (ros, ans + ch)

Java Code:

```
static void printPermutations(String str, String ans) {
    if (str.isEmpty()) {
        System.out.println(ans);
        return;
    }

    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        String ros = str.substring(0, i) + str.substring(i + 1);
        printPermutations(ros, ans + ch);
    }
}
printPermutations("abc", "")

                 ("abc","")
             /         |          \
          a            b             c
     ("bc","a")   ("ac","b")   ("ab","c")
     /     \        /     \       /     \
  b         c       a        c      a       b
("c","ab")("b","ac")("c","ba")("a","bc")("b","ca")("a","cb")
     ↓         ↓        ↓        ↓        ↓        ↓
   "abc"     "acb"    "bac"    "bca"    "cab"    "cba"
```

Complexity (Time & Space):

- Time: O(n × n!) → Each level makes n choices, total n! permutations
- Space: O(n) → Recursion depth (stack)

Approach 2: Handle Duplicates (Unique Permutations)

Idea:

- If string has duplicate characters (e.g., "aab"), naive recursion prints duplicates.
- To avoid that:
- Sort the string first.
- Use a boolean array or Set at each recursion level to skip duplicate choices.

Java Code:

```java
static void printUniquePermutations(String str, String ans) {
    if (str.isEmpty()) {
        System.out.println(ans);
        return;
    }

    HashSet<Character> used = new HashSet<>();

    for (int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        if (used.contains(ch)) continue; // skip duplicate char at this level
        used.add(ch);

        String ros = str.substring(0, i) + str.substring(i + 1);
        printUniquePermutations(ros, ans + ch);
    }
}
                ("aab","")
            /              \
       a("ab","a")        b("aa","b")
      /      \              |
a("b","aa") b("a","ab")  a("a","ba")
   ↓            ↓            ↓
 "aab"        "aba"        "baa"
```

Complexity (Time & Space):

- Time: O(n × n!) worst case
- Space: O(n) recursion depth + O(n) set storage
- But avoids repeated permutations → more efficient on duplicate inputs

---

6. Justification / Proof of Optimality

- Approach 1
  - This approach explores all possible character arrangements recursively.
  - Every level "fixes" one character while permuting the rest.
  - It's a direct application of recursion for branching problems (each branch = choice of one character).
- Approach 2
  - Sorting ensures lexicographic order.
  - Using a Set ensures no duplicate character is chosen in the same recursion level.
  - Hence, all generated permutations are unique and ordered.

---

7. Variants / Follow-Ups

- Return instead of Print
  - Return List instead of printing directly.
- Count Total Permutations

    - Instead of printing, return count → n! / (freq of each duplicate)!
- Lexicographically Next Permutation
    - Given a string, find its next permutation in sorted order.
- Kth Permutation Sequence
    - Find the k-th permutation directly (e.g., LeetCode #60).
- Permutations of Array / List
    - Apply same recursion logic on integer arrays.
- Backtracking (In-place Swap) Version
    - Instead of building substrings, swap characters in an array and backtrack.

---

8. Tips & Observations

- Recursive permutation generation is a classic backtracking pattern.
- Base case is always when all choices are made (empty string or full selection).
- Lexicographic order needs sorted input.
- Always visualize recursion as a tree of choices (branch = one pick).
- For large n, recursion becomes infeasible (factorial explosion).

Q79: String Encodings

---

1. Problem Understanding

- You are given a numeric string str.
- Each number (1–26) represents a lowercase alphabet letter:
    - 1 → a, 2 → b, 3 → c, ..., 26 → z.
- You must print all possible valid encodings of the string.
- Invalid cases:
    - Substrings starting with '0'.
    - Numbers > 26 (like 30, 40).
    - Empty string should result in a valid (printed) path.

---

2. Constraints

- 0 ≤ str.length ≤ 10
- String consists only of digits '0'–'9'.

---

3. Edge Cases

- String starts with '0' → invalid → print nothing.
- Substrings containing '0' (like "303", "06") → invalid paths.
- Empty string → print accumulated encoding.

---

4. Examples

```
Input:
123
Output:
abc
aw
lc
Explanation:
{1,2,3} → "abc"
{1,23} → "aw"
{12,3} → "lc"

Input:
013
Output:
(no output)
Invalid because it starts with 0.

Input:
303
Output:
(no output)
Invalid because "30" and "03" are not encodable.
```

---

5. Approaches

Approach 1: Recursive Encoding

Idea:

- At each step:
  - Take one digit → if between 1–9, convert and recurse.
  - Take two digits → if between 10–26, convert and recurse.
- Base case: when string is empty, print accumulated encoding.

Steps:

- If string starts with '0', return (invalid).
- Take one digit → map to a character → recurse on remaining string.
- If two digits form a number ≤ 26 → map and recurse.
- Continue until string becomes empty (print valid encoding).

Java Code:

```
public static void printEncodings(String str, String ans) {
    if (str.length() == 0) {
        System.out.println(ans);
```

```
            return;
        }

        if (str.charAt(0) == '0') return; // invalid start

        // Take one digit
        int oneDigit = str.charAt(0) - '0';
        char oneChar = (char)('a' + oneDigit - 1);
        printEncodings(str.substring(1), ans + oneChar);

        // Take two digits if possible
        if (str.length() >= 2) {
            int twoDigit = Integer.parseInt(str.substring(0, 2));
            if (twoDigit <= 26) {
                char twoChar = (char)('a' + twoDigit - 1);
                printEncodings(str.substring(2), ans + twoChar);
            }
        }
    }
Initial call:
printEncodings("123", "")

                         ("123","")
                        /           \
              take '1' (a)         take '12' (l)
                   /                    \
          ("23","a")                 ("3","l")
             /       \                    |
 take '2'(b)     take '23'(w)        take '3'(c)
      /               \                  ↓
("3","ab")         ("","aw")         ("","lc")
   |
take '3'(c)
 ↓
("","abc")
```

Complexity (Time & Space):

- Time Complexity: $O(2^n)$ — each digit can branch into 1 or 2 calls.
- Space Complexity: O(n) — recursion depth equals string length.

---

6. Justification / Proof of Optimality

- Each recursive call represents a choice point:
- Take a 1-digit encoding.
- Take a 2-digit encoding.
- Invalid branches (starting with 0 or >26) terminate early.
- The recursion tree ensures all valid combinations are explored.
- Base case ensures printing only when a full valid decoding is achieved.

---

7. Variants / Follow-Ups

- Count Encodings: Return the number of valid encodings instead of printing.
- Return List: Collect and return all encodings in a list.
- Memoized Version: Cache subproblems for optimization when string is large.
- Dynamic Programming: Convert recursion to bottom-up DP for performance.

---

8. Tips & Observations

- Always check for '0' at the start of a string/substring — it's invalid, because no letter maps to 0.
- Branching occurs at each character:
  - 1-digit number (1–9) → valid encoding.
  - 2-digit number (10–26) → valid encoding.
- Prune invalid branches early:
  - Substrings like "03", "30" are not valid → return immediately.
- Use recursion to build the encoded string incrementally:
  - Pass the accumulated string (ans) in each recursive call.
- Lexicographical order is automatically maintained:
  - Always process the 1-digit branch before the 2-digit branch.
- Base case is reached when the string is empty → print the accumulated encoding.
- Recursive depth = length of string → space complexity O(n).
- Number of total branches = 2^n in worst case (each character may split into 1-digit or 2-digit encoding).
- Observation:
  - Some digits may lead to only 1 choice (like '0' cannot start a branch).
  - Strings with repeated digits or multiple valid splits will create multiple valid encodings.
- For optimization:
  - Memoization is useful if you need to count encodings instead of printing them.
  - You can cache the number of encodings for a substring starting at each index.

Q82: Count String Encodings

---

1. Problem Understanding

- Given a string of digits, find all possible ways to encode it as letters a-z corresponding to 1-26.
- Return the total number of valid encodings.
- Examples:
  - "123" → "abc", "aw", "lc" → total 3.
  - "013" → invalid → total 0.

---

2. Constraints

- 0 <= str.length <= 10
- Digits are 0-9

---

3. Edge Cases

- Leading 0 → invalid.
- Any substring forming 0 or >26 → invalid.
- Single digit 1-9 → valid encoding.

---

4. Examples

```
Input: "123" → Output: 3

Input: "013" → Output: 0

Input: "303" → Output: 0
```

---

5. Approaches

Approach 1: Recursion

Idea:

- Try encoding 1-digit and 2-digit numbers recursively.
- Base case: empty string → return 1 (valid encoding).
- If first digit is '0' → return 0 (invalid).
- Count total encodings by summing results from 1-digit and 2-digit recursive calls.

Steps:

- If string empty → return 1.
- If first digit is '0' → return 0.
- Take first digit → recurse on rest of string.
- If first two digits ≤26 → recurse on remaining substring.
- Return sum of counts.

Java Code:

```java
static int countEncodings(String str) {
    if (str.length() == 0) return 1;
    if (str.charAt(0) == '0') return 0;

    // 1-digit encoding
    int count = countEncodings(str.substring(1));

    // 2-digit encoding
    if (str.length() >= 2) {
        int num = Integer.parseInt(str.substring(0, 2));
        if (num <= 26) count += countEncodings(str.substring(2));
    }
```

```
    return count;
}
countEncodings("123")
        /              \
  "23" (1-digit)     "3" (2-digit: 12)
   /      \                \
"3"        ""               ""
 |          |                |
""         ""                ""
Paths:

"1"-"2"-"3" → "abc"

"1"-"23" → "aw"

"12"-"3" → "lc"

Total = 3
```

Complexity (Time & Space):

- Time: O(2^n) → each step can branch into 1-digit or 2-digit encoding.
- Space: O(n) → recursion stack.

---

6. Justification / Proof of Optimality

- Correct because it recursively checks all valid splits of the string.
- Handles invalid cases with leading zeros and numbers >26.

---

7. Variants / Follow-Ups

- Return all possible encoded strings (not just count).
- Dynamic Programming / memoization to optimize for larger strings.

---

8. Tips & Observations

- Always check for leading zeros before considering a substring.
- If substring >26 → skip that branch.
- Maximum recursion depth = length of string.
- For counting, you don't need to generate strings; just sum valid paths.
