<https://leetcode.com/problems/split-array-into-fibonacci-sequence/description/>

You are given a string of digits num, such as "123456579". We can split it into a Fibonacci-like sequence [123, 456, 579].

Formally, a **Fibonacci-like** sequence is a list f of non-negative integers such that:

0 <= f[i] < 231, (that is, each integer fits in a **32-bit** signed integer type),

f.length >= 3, and

f[i] + f[i + 1] == f[i + 2] for all 0 <= i < f.length - 2.

Note that when splitting the string into pieces, each piece must not have extra leading zeroes, except if the piece is the number 0 itself.

Return any Fibonacci-like sequence split from num, or return [] if it cannot be done.

**Example 1:**

**Input:** num = "1101111"

**Output:** [11,0,11,11]

**Explanation:** The output [110, 1, 111] would also be accepted.

**Example 2:**

**Input:** num = "112358130"

**Output:** []

**Explanation:** The task is impossible.

**Example 3:**

**Input:** num = "0123"

**Output:** []

**Explanation:** Leading zeroes are not allowed, so "01", "2", "3" is not valid.

**Constraints:**

1 <= num.length <= 200

num contains only digits.

**Attempt 1: 2025-05-27**

**Solution 1: Backtracking (60 min)**

**Style 1: No early terminate (3ms finish 74 test cases)**

class Solution {

    public List<Integer> splitIntoFibonacci(String num) {

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

        helper(num, 0, result);

        return result;

    }

    private boolean helper(String num, int index, List<Integer> result) {

        int curSize = result.size();

        // Base case

        // When we've processed the entire string and have

        // at least 3 numbers in our sequence

        if(index == num.length() && curSize >= 3) {

            return true;

        }

        for(int i = index; i < num.length(); i++) {

            // Avoid leading 0

            if(num.charAt(index) == '0' && i > index) {

                break;

            }

            // Parse current integer

            // We parse it Long since it might over maximum integer

// Current number ending till i, to include must use substring till i + 1

            long cur = Long.parseLong(num.substring(index, i + 1));

            // Check if within 32-bit integer

            if(cur > Integer.MAX\_VALUE) {

                break;

            }

            // Check if the current number fits the Fibonacci property

            // (sum of last two numbers)

            if(curSize <= 1 || (long) result.get(curSize - 1) + (long) result.get(curSize - 2) == cur) {

                result.add((int) cur);

                if(helper(num, i + 1, result)) {

                    return true;

                }

                result.remove(result.size() - 1);

            }

        }

        return false;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Style 2: Early terminate (2ms finish 74 test cases)**

class Solution {

    public List<Integer> splitIntoFibonacci(String num) {

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

        helper(num, 0, result);

        return result;

    }

    private boolean helper(String num, int index, List<Integer> result) {

        int curSize = result.size();

        // Base case

        // When we've processed the entire string and have

        // at least 3 numbers in our sequence

        if(index == num.length() && curSize >= 3) {

            return true;

        }

        for(int i = index; i < num.length(); i++) {

            // Avoid leading 0

            if(num.charAt(index) == '0' && i > index) {

                break;

            }

            // Parse current integer

            // We parse it Long since it might over maximum integer

// Current number ending till i, to include must use substring till i + 1

            long cur = Long.parseLong(num.substring(index, i + 1));

            // Check if within 32-bit integer

            if(cur > Integer.MAX\_VALUE) {

                break;

            }

            // Early terminate

            // Check if the current number fits the Fibonacci property

            // (sum of last two numbers)

            if(curSize >= 2) {

                long sum = (long) result.get(curSize - 1) + (long) result.get(curSize - 2);

                if(sum > cur) {

                    continue; // Need to build a larger number

                } else if(sum < cur) {

                    break; // Cannot form a sequence

                }

            }

            result.add((int) cur);

            if(helper(num, i + 1, result)) {

                return true;

            }

            result.remove(result.size() - 1);

        }

        return false;

    }

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/split-array-into-fibonacci-sequence/solutions/139690/logical-thinking-with-clear-java-code/>

If we create a pointer curIdx, such that the sequence to the left of it is Fibonacci-like, and result, to save the Fibonacci-like sequence split from S, then

**Start State**: curIdx = 0 and result.size = 0;

**End(Aim) State**: curIdx = S.length() and result.size() >= 3

**State Transformation**:

for the current sequence, its start index is curIdx (its previous sequence's end index + 1), and its end index should be in range [curIdx, S.length() - 1]. We simply list all the possibilities, check if the current sequence is Fibonacci-like for the sequence before, i.e.

if (result.size() <= 1 || num == (long) result.get(result.size() - 1) + (long) result.get(result.size() - 2)), and check if it can form a Fibonacci-like sequence afterwards, i.e., if (splitIntoFibonacciFrom(i + 1, result, S)). We terminate the Recursion if we meet corner cases or the base case, i.e., the End State.

The clear code in Java is as below:

class Solution {

    public List<Integer> splitIntoFibonacci(String S) {

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

        splitIntoFibonacciFrom(0, result, S); // start state

        return result;

    }

    private boolean splitIntoFibonacciFrom(int curIdx, List<Integer> result, String S) {

        if (curIdx == S.length() && result.size() >= 3) { // end state (base cases)

            return true;

        }

        for (int i = curIdx; i <= S.length() - 1; i++) {

            if (S.charAt(curIdx) == '0' && i > curIdx) {

                break;

            }

            long num = Long.valueOf(S.substring(curIdx, i + 1));

            if (num > Integer.MAX\_VALUE) {

                break;

            }

            if (result.size() <= 1 || num == (long) result.get(result.size() - 1) + (long) result.get(result.size() - 2)) {

                result.add((int) num);

                if (splitIntoFibonacciFrom(i + 1, result, S)) {

                    return true;

                }

                result.remove(result.size() - 1);

            }

        }

        return false;

    }

}

**Refer to Deepseek**

**Problem Description**

Given a string of digits num, we need to split it into a Fibonacci-like sequence where:

Each number is the sum of the two preceding ones

No number has leading zeros (except for 0 itself)

All numbers must fit in a 32-bit signed integer

The sequence must contain at least 3 numbers

If such a split exists, return any valid sequence; otherwise, return an empty list .

**Approach**

The solution uses backtracking (DFS) to explore all possible ways of splitting the string into valid Fibonacci sequences:

Base Case: When we've processed the entire string and have at least 3 numbers in our sequence.

Number Construction: Build numbers digit by digit from the current position, ensuring no leading zeros.

Validation: Check if the current number fits the Fibonacci property (sum of last two numbers) and is within 32-bit integer range.

Backtracking: If a number is valid, add it to the sequence and recurse; if not, backtrack and try other splits .

**Solution Code**

class Solution {

public List<Integer> splitIntoFibonacci(String num) {

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

backtrack(num, 0, result);

return result;

}

private boolean backtrack(String num, int index, List<Integer> sequence) {

// Base case: reached end with valid sequence

if (index == num.length() && sequence.size() >= 3) {

return true;

}

for (int i = index; i < num.length(); i++) {

// Avoid leading zeros

if (num.charAt(index) == '0' && i > index) {

break;

}

// Parse current number

long current = Long.parseLong(num.substring(index, i + 1));

// Check 32-bit integer limit

if (current > Integer.MAX\_VALUE) {

break;

}

int size = sequence.size();

// If we have at least 2 numbers, check Fibonacci property

if (size >= 2) {

long sum = (long)sequence.get(size-2) + sequence.get(size-1);

if (current < sum) {

continue; // Need larger number

} else if (current > sum) {

break; // Can't form sequence

}

}

// Add to sequence and recurse

sequence.add((int)current);

if (backtrack(num, i + 1, sequence)) {

return true;

}

// Backtrack

sequence.remove(sequence.size() - 1);

}

return false;

}

}

**Explanation**

Initialization: Start with an empty result list and begin backtracking from index 0.

Number Building: For each position, build numbers by taking substrings while avoiding leading zeros.

Validation Checks:

Ensure numbers don't exceed 32-bit integer limit (2³¹-1)

For sequences with ≥2 numbers, verify current number equals the sum of last two

Recursion & Backtracking:

If valid, add number to sequence and recurse to next position

If sequence completes successfully, return true up the recursion stack

Otherwise, backtrack by removing last number and try next possibility

**Complexity Analysis**

Time Complexity: O(n²) - In worst case, we explore all possible splits

Space Complexity: O(n) - For recursion stack and storing the sequence

This solution efficiently explores all possible valid splits while respecting the problem constraints through careful backtracking.

**Time Complexity:**

The algorithm explores all possible ways to split the string into valid Fibonacci sequences.

**Key Observations:**

Branching Factor:

At each step, we can split the string into a number of varying lengths (from 1 digit up to ~10 digits, since Integer.MAX\_VALUE has 10 digits).

However, due to Fibonacci constraints, many branches are pruned early.

Maximum Recursion Depth:

The recursion depth is bounded by the length of the string n (worst case: every digit is a single number in the sequence).

Pruning Effects:

Leading zeros (break instead of continue) reduce unnecessary branches.

The Fibonacci property (current == prev1 + prev2) further limits valid paths.

**Worst-Case Time Complexity:**

**Without pruning: O(2ⁿ) (since each digit can either start a new number or extend the current one).**

**With pruning (due to leading zeros and Fibonacci constraints): O(n²) in practice.**

**Why O(n²)?**

**For each starting index i, we check up to O(n) possible splits (substrings of length 1 to n-i).**

**Due to Fibonacci constraints, the branching is limited, making it closer to O(n²) in practice.**

**Space Complexity:**

Recursion Stack:

Maximum depth is O(n) (if every digit is a separate number in the sequence).

Output Storage (sequence list):

Stores up to O(n) numbers (if each digit is a single number).

**Total Space Complexity: O(n)**

(Dominated by recursion stack and the sequence list.)

**Optimized Bound (Tighter Analysis)**

Best Case: If the string cannot form a valid sequence, we may exit early → O(1).

Average Case: Due to pruning, it performs better than brute-force, typically O(n²).

Worst Case: Still O(2ⁿ) theoretically (if no pruning happens), but practically O(n²) due to constraints.

**Final Answer:**

|  |  |  |
| --- | --- | --- |
| Complexity | Bound (General Case) | Explanation |
| **Time** | **O(n²)** | Due to substring checks and Fibonacci constraints pruning branches. |
| **Space** | **O(n)** | Recursion depth + sequence storage. |

This solution efficiently prunes invalid branches, making it feasible for reasonable input sizes (n ≤ 200 as per LeetCode constraints).

**For below statement, why it use 'break' not 'continue' ?**

// Avoid leading zeros

if (num.charAt(index) == '0' && i > index) {

break;

}

Great question! The choice between break and continue here is crucial for correct handling of numbers with leading zeros. Let me explain the reasoning:

**Why break is used instead of continue:**

Leading Zero Check:

When num.charAt(index) == '0' && i > index is true, it means:

The current number starts with '0' (num.charAt(index) == '0')

We're trying to make a multi-digit number (i > index)

Invalid Number Case:

Numbers like "01", "02", etc. are invalid (except for "0" itself)

Once we detect we're trying to create a multi-digit number starting with 0:

break exits the entire loop for this position

continue would just skip to the next iteration (longer numbers)

Logical Reason:

If we're at position i where first digit is '0' and we're trying to take more digits (i > index):

All longer numbers starting at this position will also have leading zeros ("012", "0123" etc.)

No point continuing the loop - all remaining numbers will be invalid

Hence we break out of the loop completely for this starting position

**Example Scenario:**

For input "10203":

When index=1 (second digit '0'):

First iteration (i=1): takes "0" (valid)

Second iteration (i=2): would take "02" → invalid → break

Without break, it would continue and try "020", "0203" etc. (all invalid)

**Why Not continue?**

continue would keep trying longer invalid numbers ("02", "020", "0203")

This wastes time on numbers we know will be invalid

break optimizes by stopping early when we know no valid numbers can be formed from this starting position

**Key Insight:**

The break statement here implements an important optimization - it recognizes that once we have a leading zero with multiple digits, all longer numbers starting at that position will also be invalid, so we can stop checking further possibilities from that starting index.

This is more efficient than using continue which would keep checking invalid cases unnecessarily.

**Refer to**

[L509.Fibonacci Number (Ref.L70,L746,L842,L873)](note://WEB5e91be389fb43d760647ae414e6240e8)

[L306.Additive Number (Ref.L842)](note://WEBb2ae906b7a77a2b65a82efde9ef30f36)