**Problem Statement**

Given string str, find the longest repeating non-overlapping substring in it. In other words, find two identical substrings of maximum length that do not overlap. If there is more than one such substring, return any one of them.

Write a program for the same.

**Note:** Use Dynamic Programming

**Example:**

**Input:** str = "aabaabaaba"

**Output:** aaba

**Input format :**

The input consists of a string as str.

**Output format :**

The output prints string, representing the longest repeating, non-overlapping substring in it.

**Refer to the sample output for the formatting specifications.**

**Code constraints :**

The strings contain 100 characters.

**Sample test cases :**

**Input 1 :**

malayalam

**Output 1 :**

ala

**Input 2 :**

aabaabaaba

**Output 2 :**

aaba

import java.util.\*;

class Main {

    // Method to find the longest repeated substring

    static String longestRepeatedSubstring(String str) {

        // Length of the input string

        int n = str.length();

        // Create a 2D array to store lengths of longest common suffixes

        int[][] LCSRe = new int[n + 1][n + 1];

        // To store the result (longest repeated substring)

        String res = "";

        // Length of the longest repeated substring

        int res\_length = 0;

        // Index to store the starting index of the longest repeated substring in the string

        int i, index = 0;

        // Loop through the string

        for (i = 1; i <= n; i++) {

            for (int j = i + 1; j <= n; j++) {

                // Check if characters match and suffix length is valid

                if (str.charAt(i - 1) == str.charAt(j - 1)

                        && LCSRe[i - 1][j - 1] < (j - i)) {

                    // Update the length of the longest common suffix ending at i and j

                    LCSRe[i][j] = LCSRe[i - 1][j - 1] + 1;

                    // If this length is greater than the previously found length

                    if (LCSRe[i][j] > res\_length) {

                        // Update the length and the index where this substring starts

                        res\_length = LCSRe[i][j];

                        index = Math.max(i, index);

                    }

                } else {

                    // No common suffix if characters don't match

                    LCSRe[i][j] = 0;

                }

            }

        }

        // Construct the longest repeated substring from the index and length found

        if (res\_length > 0) {

            for (i = index - res\_length + 1; i <= index; i++) {

                res += str.charAt(i - 1);

            }

        }

        return res;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Read the input string from the user

        String str = sc.nextLine();

        // Print the longest repeated substring

        System.out.println(longestRepeatedSubstring(str));

    }

}

**Explanation with Example**

Let's consider an example string "banana".

1. **Initialization**:
   * n = 6 (length of "banana").
   * LCSRe is a 7x7 matrix initialized with zeros.
   * res will hold the longest repeated substring.
   * res\_length is the length of this substring.
   * index is the index of this substring's starting position.
2. **First loop**:
   * The outer loop starts with i = 1 and goes up to 6.
   * The inner loop starts with j = i + 1 and goes up to 6.
3. **Matching and Updating**:
   * For i = 1 and j = 2, characters 'b' and 'a' do not match, so LCSRe[1][2] remains 0.
   * When i = 1 and j = 3, characters 'b' and 'n' do not match, so LCSRe[1][3] remains 0.
   * Continue this process for all combinations of i and j.
4. **Substring Construction**:
   * Once the longest length is found, the index is used to extract the substring. For "banana", the longest repeated substring is "ana".
5. **Output**:
   * For input "banana", the output will be "ana".

This approach ensures we efficiently find the longest repeated substring using dynamic programming to track common suffixes.

**Problem Statement**

Write a program to replace every array element with the sum of its previous and next elements using **dynamic memory allocation.**

**Input format :**

The first line consists of an integer n, representing the number of elements of an array.

The second line consists of n space-separated elements representing the elements of the array.

**Output format :**

The output prints the replaced array elements with the sum of its previous and next elements separated by a space.

**Refer to the sample output for formatting specifications.**

**Code constraints :**

1 ≤ n ≤ 30

**Sample test cases :**

**Input 1 :**

5

1 2 3 4 5

**Output 1 :**

3 4 6 8 9

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        // Create a Scanner object to read input

        Scanner sc = new Scanner(System.in);

        // Read the number of elements in the array

        int n = sc.nextInt();

        // Initialize the array to hold the input elements

        int[] arr = new int[n];

        // Read the elements of the array from user input

        for (int i = 0; i < n; i++) {

            arr[i] = sc.nextInt();

        }

        // Initialize the result array

        int[] result = new int[n];

        // Handle the case where the array has more than 1 element

        if (n > 1) {

            // The first element of result is the sum of the first two elements of arr

            result[0] = arr[0] + arr[1];

            // Variable to store the previous element

            int prev = arr[0];

            // Iterate through the array to calculate the result elements

            for (int i = 1; i < n - 1; i++) {

                int curr = arr[i];

                // Calculate result[i] as the sum of prev and the next element in arr

                result[i] = prev + arr[i + 1];

                // Update prev to the current element

                prev = curr;

            }

            // The last element of result is the sum of the last element and the second to last element in arr

            result[n - 1] = prev + arr[n - 1];

        } else if (n == 1) {

            // If there is only one element, the result is the same as the input

            result[0] = arr[0];

        }

        // Print the result array

        for (int i = 0; i < n; i++) {

            System.out.print(result[i] + " ");

        }

    }

}

Sure! Let’s go through the example where the input is 5 (number of elements) and the array elements are 1 2 3 4 5. We’ll break down how the code processes this input step by step.

**Input**

* Number of elements (n): 5
* Array elements (arr): {1, 2, 3, 4, 5}

**Initial Setup**

* arr = [1, 2, 3, 4, 5]
* result array will be initialized with zeros: result = [0, 0, 0, 0, 0]

**Processing**

1. **For the first element (result[0])**:
   * We calculate the sum of the first two elements of arr: arr[0] + arr[1].
   * result[0] = 1 + 2 = 3
   * Now, result = [3, 0, 0, 0, 0]
2. **For the elements between the first and the last**:
   * We loop from i = 1 to i = n - 2 (i.e., from 1 to 3 in this case).
   * **For i = 1**:
     + prev is arr[0] which is 1.
     + curr is arr[1] which is 2.
     + Calculate result[1] as prev + arr[i + 1] which is 1 + arr[2].
     + result[1] = 1 + 3 = 4
     + Update prev to curr, so prev = 2.
     + Now, result = [3, 4, 0, 0, 0]
   * **For i = 2**:
     + prev is arr[1] which is 2.
     + curr is arr[2] which is 3.
     + Calculate result[2] as prev + arr[i + 1] which is 2 + arr[3].
     + result[2] = 2 + 4 = 6
     + Update prev to curr, so prev = 3.
     + Now, result = [3, 4, 6, 0, 0]
   * **For i = 3**:
     + prev is arr[2] which is 3.
     + curr is arr[3] which is 4.
     + Calculate result[3] as prev + arr[i + 1] which is 3 + arr[4].
     + result[3] = 3 + 5 = 8
     + Update prev to curr, so prev = 4.
     + Now, result = [3, 4, 6, 8, 0]
3. **For the last element (result[n - 1])**:
   * result[4] is calculated as prev + arr[n - 1] which is 4 + arr[4].
   * result[4] = 4 + 5 = 9
   * Final result = [3, 4, 6, 8, 9]

**Output**

* The result array is printed: 3 4 6 8 9.

**Summary**

For the input array {1, 2, 3, 4, 5}, the result array is calculated as follows:

* The first element in result is the sum of the first two elements in arr.
* Each element between the first and last in result is the sum of the previous element and the next element in arr.
* The last element in result is the sum of the second to last element and the last element in arr.

The final output of the code is 3 4 6 8 9.

**Problem Statement**

You are given an array nums consisting of positive integers.

We call a subarray of nums nice if the bitwise AND of every pair of elements in different positions is equal to 0. Return the length of the longest nice subarray. A subarray is a contiguous part of an array.

Note that subarrays of length 1 are always considered nice.

**Example 1**

**Input:**

5

1 3 8 48 10

**Output:**

3

**Explanation:**

The longest nice subarray is [3,8,48]. This subarray satisfies the conditions:

- 3 AND 8 = 0.

- 3 AND 48 = 0.

- 8 AND 48 = 0.

It can be proven that no longer nice subarray can be obtained, so we return 3.

**Example 2**

**Input:**

5

3 1 5 11 13

**Output:**

1

**Explanation:**

The length of the longest nice subarray is 1. Any subarray of length 1 can be chosen.

**Input format :**

The first line contains an integer N representing the size of the array.

The second line contains N space-separated integers arr[i], representing the elements of the array.

**Output format :**

The output displays a single integer, the length of the longest nice subarray.

**Refer to the sample output for the formatting specifications.**

**Code constraints :**

1 ≤ N ≤ 10

1 ≤ arr[i] ≤100

**Sample test cases :**

**Input 1 :**

5

1 3 8 48 10

**Output 1 :**

3

**Input 2 :**

5

3 1 5 11 13

**Output 2 :**

1

import java.util.Scanner;

public class Main {

    // Method to find the length of the longest "nice" subarray

    public static int longestNiceSubarray(int[] nums) {

        int n = nums.length; // Length of the input array

        int maxLength = 1; // Initialize maximum length of "nice" subarray to 1

        // Iterate over all possible start points of the subarray

        for (int start = 0; start < n; start++) {

            // Iterate over all possible end points of the subarray

            for (int end = start; end < n; end++) {

                boolean isNice = true; // Assume the current subarray is "nice"

                // Check if all pairs in the current subarray are "nice"

                for (int i = start; i <= end; i++) {

                    for (int j = i + 1; j <= end; j++) {

                        // If any pair of elements has overlapping bits, it's not "nice"

                        if ((nums[i] & nums[j]) != 0) {

                            isNice = false; // Mark subarray as not "nice"

                            break;

                        }

                    }

                    if (!isNice) break; // Exit outer loop if subarray is not "nice"

                }

                // Update the maximum length if the current subarray is "nice"

                if (isNice) {

                    maxLength = Math.max(maxLength, end - start + 1);

                }

            }

        }

        return maxLength; // Return the length of the longest "nice" subarray

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt(); // Read the number of elements

        int[] nums = new int[n];

        // Read the elements into the array

        for (int i = 0; i < n; i++) {

            nums[i] = sc.nextInt();

        }

        // Output the length of the longest "nice" subarray

        System.out.println(longestNiceSubarray(nums));

        sc.close(); // Close the scanner

    }

}

Let's analyze the input 5 and the array 3 1 5 11 13 using the provided code. We'll go through each step to determine the length of the longest "nice" subarray.

**Input**

* Number of elements (n): 5
* Array elements (nums): {3, 1, 5, 11, 13}

**Process**

1. **Initialization**:
   * nums = [3, 1, 5, 11, 13]
   * maxLength = 1 (initial maximum length)
2. **Iterate Over All Subarrays**:
   * **Subarray starting at index 0**:
     + **Subarray [3]**:
       - Single element, so it's "nice".
       - maxLength remains 1.
     + **Subarray [3, 1]**:
       - Check pairs: (3, 1).
       - 3 & 1 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
     + **Subarray [3, 1, 5]**:
       - Check pairs: (3, 1), (3, 5), (1, 5).
       - 3 & 1 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
     + **Subarray [3, 1, 5, 11]**:
       - Check pairs: (3, 1), (3, 5), (3, 11), (1, 5), (1, 11), (5, 11).
       - 3 & 1 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
     + **Subarray [3, 1, 5, 11, 13]**:
       - Check pairs: (3, 1), (3, 5), (3, 11), (3, 13), (1, 5), (1, 11), (1, 13), (5, 11), (5, 13), (11, 13).
       - 3 & 1 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
   * **Subarray starting at index 1**:
     + **Subarray [1]**:
       - Single element, so it's "nice".
       - maxLength remains 1.
     + **Subarray [1, 5]**:
       - Check pairs: (1, 5).
       - 1 & 5 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
     + **Subarray [1, 5, 11]**:
       - Check pairs: (1, 5), (1, 11), (5, 11).
       - 1 & 5 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
     + **Subarray [1, 5, 11, 13]**:
       - Check pairs: (1, 5), (1, 11), (1, 13), (5, 11), (5, 13), (11, 13).
       - 1 & 5 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
   * **Subarray starting at index 2**:
     + **Subarray [5]**:
       - Single element, so it's "nice".
       - maxLength remains 1.
     + **Subarray [5, 11]**:
       - Check pairs: (5, 11).
       - 5 & 11 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
     + **Subarray [5, 11, 13]**:
       - Check pairs: (5, 11), (5, 13), (11, 13).
       - 5 & 11 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
   * **Subarray starting at index 3**:
     + **Subarray [11]**:
       - Single element, so it's "nice".
       - maxLength remains 1.
     + **Subarray [11, 13]**:
       - Check pairs: (11, 13).
       - 11 & 13 = 9 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
   * **Subarray starting at index 4**:
     + **Subarray [13]**:
       - Single element, so it's "nice".
       - maxLength remains 1.
3. **Final Output**:
   * The length of the longest "nice" subarray is 1.

**Summary**

For the input array {3, 1, 5, 11, 13}, no subarray longer than a single element meets the "nice" criteria where no two elements overlap in their bitwise representation. Therefore, the length of the longest "nice" subarray is 1.

Let's determine the length of the longest "nice" subarray for the input 5 with the array {1, 3, 8, 48, 10}. We'll use the provided code to analyze this step by step.

**Input**

* Number of elements (n): 5
* Array elements (nums): {1, 3, 8, 48, 10}

**Process**

1. **Initialization**:
   * nums = [1, 3, 8, 48, 10]
   * maxLength = 1 (initial maximum length)
2. **Iterate Over All Subarrays**:
   * **Subarray starting at index 0**:
     + **Subarray [1]**:
       - Single element, so it's "nice".
       - maxLength remains 1.
     + **Subarray [1, 3]**:
       - Check pairs: (1, 3).
       - 1 & 3 = 1 (overlap detected, subarray is not "nice").
       - maxLength remains 1.
     + **Subarray [1, 3, 8]**:
       - Check pairs: (1, 3), (1, 8), (3, 8).
       - 1 & 3 = 1 (overlap detected).
       - maxLength remains 1.
     + **Subarray [1, 3, 8, 48]**:
       - Check pairs: (1, 3), (1, 8), (1, 48), (3, 8), (3, 48), (8, 48).
       - 1 & 3 = 1 (overlap detected).
       - maxLength remains 1.
     + **Subarray [1, 3, 8, 48, 10]**:
       - Check pairs: (1, 3), (1, 8), (1, 48), (1, 10), (3, 8), (3, 48), (3, 10), (8, 48), (8, 10), (48, 10).
       - 1 & 3 = 1 (overlap detected).
       - maxLength remains 1.
   * **Subarray starting at index 1**:
     + **Subarray [3]**:
       - Single element, so it's "nice".
       - maxLength remains 1.
     + **Subarray [3, 8]**:
       - Check pairs: (3, 8).
       - 3 & 8 = 0 (no overlap), so subarray is "nice".
       - Update maxLength to 2.
     + **Subarray [3, 8, 48]**:
       - Check pairs: (3, 8), (3, 48), (8, 48).
       - 3 & 8 = 0, 3 & 48 = 0, 8 & 48 = 0 (all pairs are "nice").
       - Update maxLength to 3.
     + **Subarray [3, 8, 48, 10]**:
       - Check pairs: (3, 8), (3, 48), (3, 10), (8, 48), (8, 10), (48, 10).
       - 3 & 8 = 0, 3 & 48 = 0, 3 & 10 = 2, 8 & 48 = 0, 8 & 10 = 8, 48 & 10 = 8 (overlap detected).
       - maxLength remains 3.
   * **Subarray starting at index 2**:
     + **Subarray [8]**:
       - Single element, so it's "nice".
       - maxLength remains 3.
     + **Subarray [8, 48]**:
       - Check pairs: (8, 48).
       - 8 & 48 = 8 (overlap detected, subarray is not "nice").
       - maxLength remains 3.
     + **Subarray [8, 48, 10]**:
       - Check pairs: (8, 48), (8, 10), (48, 10).
       - 8 & 48 = 8, 8 & 10 = 8, 48 & 10 = 8 (overlap detected).
       - maxLength remains 3.
   * **Subarray starting at index 3**:
     + **Subarray [48]**:
       - Single element, so it's "nice".
       - maxLength remains 3.
     + **Subarray [48, 10]**:
       - Check pairs: (48, 10).
       - 48 & 10 = 8 (overlap detected, subarray is not "nice").
       - maxLength remains 3.
   * **Subarray starting at index 4**:
     + **Subarray [10]**:
       - Single element, so it's "nice".
       - maxLength remains 3.
3. **Final Output**:
   * The length of the longest "nice" subarray is 3.

For the input array {1, 3, 8, 48, 10}, the longest "nice" subarray found is {3, 8, 48}, which has a length of 3.

**Problem Statement**

Given an integer array nums, return the length of the longest strictly increasing subsequence.

**Example 1:**

**Input:** nums = [10,9,2,5,3,7,101,18]

**Output:** 4

**Explanation:** The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

**Example 2:**

**Input:** nums = [0,1,0,3,2,3]

**Output:** 4

**Example 3:**

**Input:** nums = [7,7,7,7,7,7,7]

**Output:** 1

**Input format :**

The first line of input contains an integer **n**, which represents the length of the array.

The second line of input contains **n** space-separated integers, which are the elements of the array.

**Output format :**

The output prints an integer, representing the length of the longest strictly increasing subsequence of the given array.

**Refer to the sample output for the formatting specifications.**

**Code constraints :**

In this scenario, the given test cases will fall under the following constraints:

1 ≤ n ≤ 50

0 ≤ Elements of the array ≤ 150

**Sample test cases :**

**Input 1 :**

7

7 7 7 7 7 7 7

**Output 1 :**

1

**Input 2 :**

6

0 1 0 3 2 3

**Output 2 :**

4

**Input 3 :**

8

10 9 2 5 3 7 101 8

**Output 3 :**

4

import java.util.Scanner;

public class Main {

    /\*\*

     \* Calculates the length of the Longest Increasing Subsequence (LIS).

     \*

     \* @param nums Array of integers

     \* @return Length of the LIS

     \*/

    public static int lengthOfLIS(int[] nums) {

        int n = nums.length; // Get the length of the input array

        // If the array is empty, the LIS length is 0

        if (n == 0) return 0;

        // Initialize the dp array where dp[i] will store the length of LIS ending at index i

        int[] dp = new int[n];

        for (int i = 0; i < n; i++) {

            dp[i] = 1; // Each element is an LIS of length 1 by itself

        }

        // Fill the dp array

        for (int i = 1; i < n; i++) { // Iterate over each element starting from the second element

            for (int j = 0; j < i; j++) { // Check all previous elements

                // If the current element is greater than the previous element, it can extend the LIS

                if (nums[i] > nums[j]) {

                    // Update dp[i] to be the maximum of its current value or extending LIS ending at j

                    dp[i] = Math.max(dp[i], dp[j] + 1);

                }

            }

        }

        // Find the maximum value in dp array which represents the length of the longest increasing subsequence

        int maxLength = 0;

        for (int length : dp) {

            maxLength = Math.max(maxLength, length);

        }

        return maxLength;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        // Read the number of elements in the array

        int n = sc.nextInt();

        int[] nums = new int[n];

        // Read the elements of the array

        for (int i = 0; i < n; i++) {

            nums[i] = sc.nextInt();

        }

        // Compute the length of the longest increasing subsequence

        System.out.println(lengthOfLIS(nums));

        sc.close(); // Close the scanner

    }

}

Let's determine the length of the Longest Increasing Subsequence (LIS) for the input 6 with the array {0, 1, 0, 3, 2, 3} using the provided code.

**Input**

* Number of elements (n): 6
* Array elements (nums): {0, 1, 0, 3, 2, 3}

**Process**

1. **Initialization**:

java

Copy code

int n = nums.length; // n = 6

if (n == 0) return 0;

int[] dp = new int[n];

for (int i = 0; i < n; i++) {

dp[i] = 1;

}

* + dp = [1, 1, 1, 1, 1, 1] (each position initially assumes the LIS is just that element itself)

1. **Compute LIS**:

java

Copy code

for (int i = 1; i < n; i++) {

for (int j = 0; j < i; j++) {

if (nums[i] > nums[j]) {

dp[i] = Math.max(dp[i], dp[j] + 1);

}

}

}

* + For i = 1 (nums[1] = 1):
    - j = 0 (nums[0] = 0):
      * 1 > 0, so dp[1] = max(dp[1], dp[0] + 1) = max(1, 2) = 2
    - dp = [1, 2, 1, 1, 1, 1]
  + For i = 2 (nums[2] = 0):
    - j = 0 (nums[0] = 0):
      * 0 <= 0, no update.
    - j = 1 (nums[1] = 1):
      * 0 <= 1, no update.
    - dp = [1, 2, 1, 1, 1, 1]
  + For i = 3 (nums[3] = 3):
    - j = 0 (nums[0] = 0):
      * 3 > 0, so dp[3] = max(dp[3], dp[0] + 1) = max(1, 2) = 2
    - j = 1 (nums[1] = 1):
      * 3 > 1, so dp[3] = max(dp[3], dp[1] + 1) = max(2, 3) = 3
    - j = 2 (nums[2] = 0):
      * 3 > 0, no update as dp[2] + 1 does not exceed current dp[3].
    - dp = [1, 2, 1, 3, 1, 1]
  + For i = 4 (nums[4] = 2):
    - j = 0 (nums[0] = 0):
      * 2 > 0, so dp[4] = max(dp[4], dp[0] + 1) = max(1, 2) = 2
    - j = 1 (nums[1] = 1):
      * 2 > 1, so dp[4] = max(dp[4], dp[1] + 1) = max(2, 3) = 3
    - j = 2 (nums[2] = 0):
      * 2 > 0, no update as dp[2] + 1 does not exceed current dp[4].
    - j = 3 (nums[3] = 3):
      * 2 <= 3, no update.
    - dp = [1, 2, 1, 3, 3, 1]
  + For i = 5 (nums[5] = 3):
    - j = 0 (nums[0] = 0):
      * 3 > 0, so dp[5] = max(dp[5], dp[0] + 1) = max(1, 2) = 2
    - j = 1 (nums[1] = 1):
      * 3 > 1, so dp[5] = max(dp[5], dp[1] + 1) = max(2, 3) = 3
    - j = 2 (nums[2] = 0):
      * 3 > 0, no update as dp[2] + 1 does not exceed current dp[5].
    - j = 3 (nums[3] = 3):
      * 3 <= 3, no update.
    - j = 4 (nums[4] = 2):
      * 3 > 2, so dp[5] = max(dp[5], dp[4] + 1) = max(3, 4) = 4
    - dp = [1, 2, 1, 3, 3, 4]

1. **Find Maximum LIS Length**:

java

Copy code

int maxLength = 0;

for (int length : dp) {

maxLength = Math.max(maxLength, length);

}

* + Maximum value in dp is 4.

1. **Return Result**:

java

Copy code

return maxLength; // returns 4

**Output**

The length of the longest increasing subsequence for the input array {0, 1, 0, 3, 2, 3} is 4.

The LIS in this case could be [0, 1, 2, 3].