

Assignment - 4

Name: Aanchal Negi

Branch: BE-CSE

Semester: 6th

Subject Name: AP Lab-2

UID: 22BCS14969

Section/Group: IOT-609/B

Date :18/03/25

Subject Code: 22CSP-351

➤ Longest Nice Substring

Code:

```
class Solution {
    public String longestNiceSubstring(String s) {
        if (s.length() < 2) {
            return "";
        }
        for (int i = 0; i < s.length(); i++) {
            if (!s.contains(String.valueOf(s.charAt(i)).toLowerCase()) ||
                !s.contains(String.valueOf(s.charAt(i)).toUpperCase())) {
                String left = longestNiceSubstring(s.substring(0, i));
                String right = longestNiceSubstring(s.substring(i + 1));
                return left.length() >= right.length() ? left : right;
            }
        }
        return s;
    }
}
```

Output:

The screenshot displays a coding platform interface with a dark theme. On the left, the 'Submissions' tab is active, showing a submission by 'sapphire040812' that was 'Accepted' with 73/73 testcases passed. The submission details include a runtime of 1 ms (beats 97.78%) and a memory usage of 42.43 MB (beats 75.11%). A green 'Solution' button is visible. The main area shows the Java code for the 'Longest Nice Substring' problem. The code is a recursive function that iterates through the string, checking for characters that are not part of a 'nice' substring (i.e., characters whose lowercase and uppercase versions are not both present). If such a character is found, the function recursively finds the longest nice substring to the left and right of the character and returns the longer one. If no such character is found, the entire string is returned. The code is syntax-highlighted and includes line numbers. At the bottom, there is a 'Testcase' tab with a 'Test Result' section showing a table with columns for 'Case', 's', and 'Result'. The 'Case 1' tab is selected, and the input 's =' is visible. The 'Run' and 'Submit' buttons are at the bottom right of the code editor.

➤ Reverse Bits

Code:

```
public class Solution {
    // you need treat n as an unsigned value
    public int reverseBits(int n) {
        int result = 0;

        // Iterate over all 32 bits
        for (int i = 0; i < 32; i++) {
            // Extract the last bit of n
            int bit = n & 1;

            // Shift the bit to its reversed position and accumulate it in result
            result = (result << 1) | bit;

            // Right shift n to process the next bit
            n >>= 1;
        }

        return result;
    }
}
```

Output:

Problem List

Description | Accepted × | Editorial | Solutions | Submissions

All Submissions

Accepted 600 / 600 testcases passed

sapphir... submitted at Mar 17, 2025 21:24

Runtime

0 ms | Beats 100.00%

Analyze Complexity

Memory

41.51 MB | Beats 94.56%

Editorial

Solution

Code

Java Auto

```
1 public class Solution {  
2     // you need treat n as an unsigned value  
3     public int reverseBits(int n) {  
4         int result = 0;  
5  
6         // Iterate over all 32 bits  
7         for (int i = 0; i < 32; i++) {  
8             // Extract the last bit of n  
9             int bit = n & 1;  
10            // Shift the bit to its reversed position and accumulate it in result  
11            result = (result << 1) | bit;  
12        }  
13    }  
14 }
```

Ln 19, Col 1 | Saved

Run Submit

Testcase | Test Result

Expected

321225471 (10111111111111111111111111111111)

Contribute a testcase

➤ Number of 1 Bits

Code:

```
public class Solution {
    // Function to calculate Hamming weight
    public int hammingWeight(int n) {
        int count = 0; // Initialize count of set bits

        // Loop until all bits are processed
        while (n != 0) {
            count += (n & 1); // Add the last bit to count if it's 1
            n >>= 1;          // Right shift n (unsigned) by 1 to process the next bit
        }

        return count;
    }
}
```

Output:

The screenshot displays a code editor interface for a problem titled "Number of 1 Bits". The left sidebar shows the submission status as "Accepted" with 598/598 testcases passed, submitted by "sapphir..." on Mar 17, 2025 at 21:47. The runtime is 1 ms, beating 17.40% of solutions, and the memory usage is 40.88 MB, beating 42.18%. The right pane shows the Java code for the solution, which uses a while loop to count the number of set bits in an integer n. The code is as follows:

```
1 public class Solution {
2     // Function to calculate Hamming weight
3     public int hammingWeight(int n) {
4         int count = 0; // Initialize count of set bits
5
6         // Loop until all bits are processed
7         while (n != 0) {
8             count += (n & 1); // Add the last bit to count if it's 1
9             n >>= 1;          // Right shift n (unsigned) by 1 to process the next bit
10        }
11    }
```

Below the code editor, the "Test Result" section shows "Accepted" with a runtime of 0 ms. There are three test cases listed: Case 1, Case 2 (selected), and Case 3. The input field is currently empty.

➤ Maximum Subarray

Code:

```
public class Solution
{
    public int maxSubArray(int[] nums)
    {
        int maxSum = nums[0];
        int currentSum = nums[0];

        for (int i = 1; i < nums.length; i++)
        {
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }
        return maxSum;
    }
}
```

Output:

The screenshot shows a LeetCode submission interface. On the left, the 'All Submissions' tab is active, showing the submission status as 'Accepted' with 210 / 210 testcases passed. The user 'sapphir...' submitted the solution on Mar 17, 2025 at 22:41. The 'Runtime' section shows a time of 1 ms, beating 99.51% of solutions. The 'Memory' section shows a memory usage of 57.02 MB, beating 56.35% of solutions. On the right, the 'Code' editor shows the Java code for the Maximum Subarray problem. The code is as follows:

```
1 public class Solution
2 {
3     public int maxSubArray(int[] nums)
4     {
5         int maxSum = nums[0];
6         int currentSum = nums[0];
7
8         for (int i = 1; i < nums.length; i++)
9         {
10             currentSum = Math.max(nums[i], currentSum + nums[i]);
11             maxSum = Math.max(maxSum, currentSum);
12         }
13         return maxSum;
14     }
15 }
```

Below the code editor, the 'Testcase' section shows the input array: `nums = [-2,1,-3,4,-1,2,1,-5,4]`. The 'Run' and 'Submit' buttons are visible at the bottom right of the code editor.

➤ Search a 2D Matrix II

Code:

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        // Get the dimensions of the matrix
        int rows = matrix.length;
        int cols = matrix[0].length;

        // Start from the top-right corner
        int row = 0, col = cols - 1;

        // Traverse the matrix
        while (row < rows && col >= 0) {
            if (matrix[row][col] == target) {
                return true; // Target found
            } else if (matrix[row][col] > target) {
                col--; // Move left
            } else {
                row++; // Move down
            }
        }

        return false; // Target not found
    }
}
```

The screenshot displays a code editor interface with a dark theme. On the left, a sidebar shows the 'Problem List' and 'Accepted' status for the problem. The main editor area displays the Java code for the 'searchMatrix' function. The code is as follows:

```
1 public class Solution {
2     public boolean searchMatrix(int[][] matrix, int target) {
3         // Get the dimensions of the matrix
4         int rows = matrix.length;
5         int cols = matrix[0].length;
6
7         // Start from the top-right corner
8         int row = 0, col = cols - 1;
9
10        // Traverse the matrix
11        while (row < rows && col >= 0) {
12            if (matrix[row][col] == target) {
13                return true;
14            } else if (matrix[row][col] > target) {
15                col--;
16            } else {
17                row++;
18            }
19        }
20        return false;
21    }
22 }
```

Below the code editor, the 'Testcase' tab is active, showing 'Accepted' status and 'Runtime: 0 ms'. The 'Input' field is empty.

➤ Super Pow

Code:

```
public class Solution {
    private static final int MOD = 1337;

    public int superPow(int a, int[] b) {
        // Reduce `a` modulo 1337 initially
        a %= MOD;

        // Start with result = 1
        int result = 1;
        for (int digit : b) {
            result = powerMod(result, 10) * powerMod(a, digit) % MOD;
        }

        return result;
    }

    // Helper method to calculate (base^exp) % MOD
    private int powerMod(int base, int exp) {
        int result = 1;
        while (exp > 0) {
            if (exp % 2 == 1) {
                result = result * base % MOD;
            }
            base = base * base % MOD;
            exp /= 2;
        }
        return result;
    }
}
```

Output:

The screenshot shows a code editor interface for a problem named 'Super Pow'. The code is written in Java and is accepted. The runtime is 6 ms, and the memory is 44.43 MB. The test result is 'Accepted' with a runtime of 0 ms.

Code:

```
1 public class Solution {
2     private static final int MOD = 1337;
3
4     public int superPow(int a, int[] b) {
5         // Reduce `a` modulo 1337 initially
6         a %= MOD;
7
8         // Start with result = 1
9         int result = 1;
10        for (int digit : b) {
11            result = powerMod(result, 10) * powerMod(a, digit) % MOD;
12        }
13
14        return result;
15    }
16
17    // Helper method to calculate (base^exp) % MOD
18    private int powerMod(int base, int exp) {
19        int result = 1;
20        while (exp > 0) {
21            if (exp % 2 == 1) {
22                result = result * base % MOD;
23            }
24            base = base * base % MOD;
25            exp /= 2;
26        }
27        return result;
28    }
29 }
```

Testcase: Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

➤ Beautiful Array

Code:

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

public class Solution {
    public int[] beautifulArray(int n) {
        List<Integer> result = new ArrayList<>();
        result.add(1); // Start with the base case

        // Build the beautiful array using divide-and-conquer
        while (result.size() < n) {
            List<Integer> temp = new ArrayList<>();
            // Add odd numbers to preserve the beautiful property
            for (int num : result) {
                if (num * 2 - 1 <= n) {
                    temp.add(num * 2 - 1);
                }
            }
            // Add even numbers to preserve the beautiful property
            for (int num : result) {
                if (num * 2 <= n) {
                    temp.add(num * 2);
                }
            }
            result = temp;
        }

        // Convert the result list to an array
        int[] beautifulArray = new int[n];
        for (int i = 0; i < n; i++) {
            beautifulArray[i] = result.get(i);
        }
        return beautifulArray;
    }
}
```

The screenshot displays a coding platform interface with the following components:

- Problem List:** Shows the problem is "Accepted" with 38/38 testcases passed. The user "sapphir..." submitted it on Mar 18, 2025 at 10:41. Buttons for "Editorial" and "Solution" are visible.
- Runtime/Memory Analysis:** Shows a runtime of 4 ms (Beats 51.67%) and memory usage of 42.46 MB (Beats 64.38%). A bar chart at the bottom shows performance relative to other solutions.
- Code Editor:** Contains the Java code for the "Beautiful Array" problem, which uses a divide-and-conquer approach to build the array.
- Test Result:** Shows the solution is "Accepted" with a runtime of 0 ms. It lists "Case 1" and "Case 2" as test cases.

➤ The Skyline Problem

Code:

```
import java.util.*;

public class Solution {
    public List<List<Integer>> getSkyline(int[][] buildings) {
        List<int[]> points = new ArrayList<>();
        for (int[] building : buildings) {
            // Add building start point (negative height to distinguish from end points)
            points.add(new int[] {building[0], -building[2]});
            // Add building end point
            points.add(new int[] {building[1], building[2]});
        }

        // Sort points:
        // 1. By x-coordinate
        // 2. By height (start points before end points for same x)
        Collections.sort(points, (a, b) -> {
            if (a[0] != b[0]) return a[0] - b[0]; // Sort by x
            return a[1] - b[1]; // Start (-height) before end (+height)
        });

        // Use a max-heap to keep track of building heights
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        maxHeap.add(0); // Initialize with ground level
        int prevMaxHeight = 0;

        List<List<Integer>> result = new ArrayList<>();
        for (int[] point : points) {
            int x = point[0];
            int height = point[1];

            if (height < 0) {
                // Start of a building: add its height
                maxHeap.add(-height);
            } else {
                // End of a building: remove its height
                maxHeap.remove(height);
            }

            // Get current max height
            int currMaxHeight = maxHeap.peek();

            // If max height changes, add a key point
            if (currMaxHeight != prevMaxHeight) {
                result.add(Arrays.asList(x, currMaxHeight));
                prevMaxHeight = currMaxHeight;
            }
        }
    }
}
```



```

        return result;
    }
}

```

Output:

The screenshot shows a LeetCode submission for the 'Trapping Water' problem. The solution is accepted, with a runtime of 238 ms and memory usage of 51.41 MB. The code is written in Java and uses a two-pointer approach to calculate the trapped water.

```

1  import java.util.*;
2
3  public class Solution {
4      public List<List<Integer>> getSkyline(int[][] buildings) {
5          List<int[]> points = new ArrayList<>();
6          for (int[] building : buildings) {
7              // Add building start point (negative height to distinguish from end points)
8              points.add(new int[] {building[0], -building[2]});
9              // Add building end point
10             points.add(new int[] {building[1], building[2]});
11         }
12     }
13 }

```

➤ Reverse Pairs :

Code:

```

public class Solution {
    public int reversePairs(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        return mergeSort(nums, 0, nums.length - 1);
    }

    private int mergeSort(int[] nums, int left, int right) {
        if (left >= right) {
            return 0;
        }

        int mid = left + (right - left) / 2;
        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);

        // Count reverse pairs in the current range
        int j = mid + 1;
        for (int i = left; i <= mid; i++) {
            while (j <= right && nums[i] > 2L * nums[j]) {
                j++;
            }
            count += (j - (mid + 1));
        }
    }
}

```

```

        // Merge the two halves
        merge(nums, left, mid, right);
        return count;
    }

    private void merge(int[] nums, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;

        while (i <= mid && j <= right) {
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                temp[k++] = nums[j++];
            }
        }

        while (i <= mid) {
            temp[k++] = nums[i++];
        }

        while (j <= right) {
            temp[k++] = nums[j++];
        }

        for (int p = 0; p < temp.length; p++) {
            nums[left + p] = temp[p];
        }
    }
}

```

Output:

The screenshot displays a LeetCode submission page for the problem "Reverse Pairs". The submission is by user "sapphir..." and is marked as "Accepted" with 140/140 testcases passed. The submission time is Mar 18, 2025 11:26.

Runtime Performance:
 42 ms | Beats 78.48%
 Analyze Complexity

Memory Performance:
 55.03 MB | Beats 55.54%

Code Editor:
 The code is written in Java. It defines a class `Solution` with a public method `reversePairs` and a private recursive method `mergeSort`. The `reversePairs` method checks for null or empty arrays and then calls `mergeSort` to count the reverse pairs. The `mergeSort` method implements a merge sort algorithm to count reverse pairs during the merging process.

```

1 public class Solution {
2     public int reversePairs(int[] nums) {
3         if (nums == null || nums.length == 0) {
4             return 0;
5         }
6         return mergeSort(nums, 0, nums.length - 1);
7     }
8
9     private int mergeSort(int[] nums, int left, int right) {
10        if (left >= right) {
11            return 0;
12        }
13        int mid = (left + right) / 2;
14        int count = mergeSort(nums, left, mid) + mergeSort(nums, mid + 1, right);
15        count += merge(nums, left, mid, right);
16        return count;
17    }
18
19    private int merge(int[] nums, int left, int mid, int right) {
20        int[] temp = new int[right - left + 1];
21        int i = left, j = mid + 1, k = 0;
22
23        while (i <= mid && j <= right) {
24            if (nums[i] <= nums[j]) {
25                temp[k++] = nums[i++];
26            } else {
27                temp[k++] = nums[j++];
28            }
29        }
30
31        while (i <= mid) {
32            temp[k++] = nums[i++];
33        }
34
35        while (j <= right) {
36            temp[k++] = nums[j++];
37        }
38
39        for (int p = 0; p < temp.length; p++) {
40            nums[left + p] = temp[p];
41        }
42
43        return 0;
44    }
45 }

```

Testcase Results:
 The test result is "Accepted" with a runtime of 0 ms. Two test cases are shown: "Case 1" and "Case 2".

➤ Longest Increasing Subsequence II

Code:

```
import java.util.*;

class Solution {
    public int lengthOfLIS(int[] nums, int k) {
        // Find the maximum number in nums to determine the size of the segment tree
        int maxNum = 0;
        for (int num : nums) {
            if (num > maxNum) {
                maxNum = num;
            }
        }

        // Initialize the segment tree
        SegmentTree st = new SegmentTree(maxNum);

        int result = 0;
        for (int num : nums) {
            // Query the maximum subsequence length in the range [num - k, num - 1]
            int prevMax = st.query(Math.max(1, num - k), num - 1);
            int currentLength = prevMax + 1;

            // Update the segment tree with the new subsequence length for the current number
            st.update(num, currentLength);

            // Update the result with the maximum subsequence length found so far
            result = Math.max(result, currentLength);
        }

        return result;
    }
}

// Segment Tree implementation
class SegmentTree {
    int[] tree;
    int size;

    public SegmentTree(int size) {
        this.size = size;
        this.tree = new int[4 * (size + 1)]; // Ensure enough space for the tree
    }

    // Update the value at a specific index
    public void update(int index, int value) {
        update(0, 0, size, index, value);
    }

    private void update(int node, int start, int end, int index, int value) {
        if (start == end) {
            tree[node] = Math.max(tree[node], value); // Ensure we take the maximum value
            return;
        }

        int mid = (start + end) / 2;
        if (index <= mid) {
            update(2 * node + 1, start, mid, index, value);
        } else {
            update(2 * node + 2, mid + 1, end, index, value);
        }
        tree[node] = Math.max(tree[2 * node + 1], tree[2 * node + 2]);
    }
}
```

```

        update(2 * node + 1, start, mid, index, value);
    } else {
        update(2 * node + 2, mid + 1, end, index, value);
    }

    // Update the current node with the maximum value from its children
    tree[node] = Math.max(tree[2 * node + 1], tree[2 * node + 2]);
}

// Query the maximum value in a range [l, r]
public int query(int l, int r) {
    return query(0, 0, size, l, r);
}

private int query(int node, int start, int end, int l, int r) {
    if (r < start || end < l) {
        return 0; // Out of range
    }

    if (l <= start && end <= r) {
        return tree[node]; // Fully within range
    }

    int mid = (start + end) / 2;
    int left = query(2 * node + 1, start, mid, l, r);
    int right = query(2 * node + 2, mid + 1, end, l, r);

    return Math.max(left, right); // Return the maximum value in the range
}
}
}

```

Output:

The screenshot displays a code editor interface with the following components:

- Problem List:** Shows the current problem is "Accepted" with 84 / 84 testcases passed. The submission was made by "sapphire040812" on Mar 18, 2025 at 11:36.
- Runtime:** 102 ms, Beats 48.36%.
- Memory:** 57.44 MB, Beats 60.66%.
- Code Editor:** Contains the Java code for the segment tree implementation. The code is in Java and uses a recursive approach for both update and query operations.
- Test Result:** Shows "Accepted" with a runtime of 0 ms. The test cases are Case 1, Case 2, and Case 3.