Experiment- 4

Student Name: Kamal Mehta **UID:** 22BET10097

Branch: B.E - IT **Section/Group:** 22BET-701/A

Semester: 6th **Date of Performance:** 14-02-25

Subject Name: AP Lab -2 **Subject Code:** 22ITP-351

Problem 4.1: Max Subarray

1. **Problem Statement:** To find the contiguous subarray within a one-dimensional array of numbers that has the largest sum.

2. **Objective:**

I. Implement an efficient algorithm (like Kadane's algorithm) to find the maximum sum of a contiguous subarray in linear time.

II. Develop a search algorithm that leverages the sorted properties of the matrix to locate a target value in O(m + n) time complexity.

III. Create a function that calculates the result of a base raised to a power represented by an array of digits, using modular arithmetic to handle large numbers.

IV. Design an algorithm to construct an array that meets the criteria of having a specific arrangement of odd and even integers, ensuring the output is valid.

V. Use a sweep line algorithm or a priority queue to compute the critical points of the skyline formed by overlapping rectangles, producing a list of key points that outline the skyline.

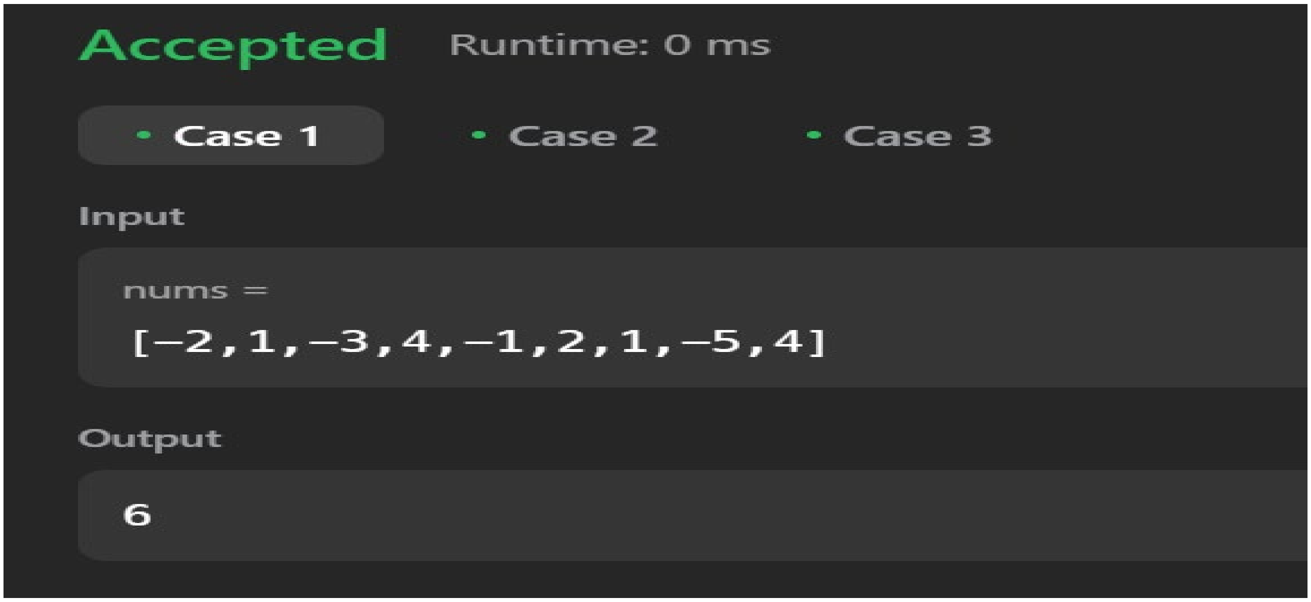
VI. Implement a modified merge sort algorithm to efficiently count the number of important reverse pairs in an array, achieving a time complexity of O(n log n).

VII. Utilize dynamic programming or binary search techniques to find the length of the longest increasing subsequence in a more complex input scenario, optimizing for performance.

3.1. Code 4.1:

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        max_sum = current_sum = nums[0]
        for num in nums[1:]:
            current_sum = max(num, current_sum + num)
            max_sum = max(max_sum, current_sum)
        return max_sum
```

1.1. Output 4.1:



Accepted Runtime: 0 ms

Input

nums =

```
[-2,1,-3,4,-1,2,1,-5,4]
```

Output

```
6
```

Fig 1: Output for Problem 4.1

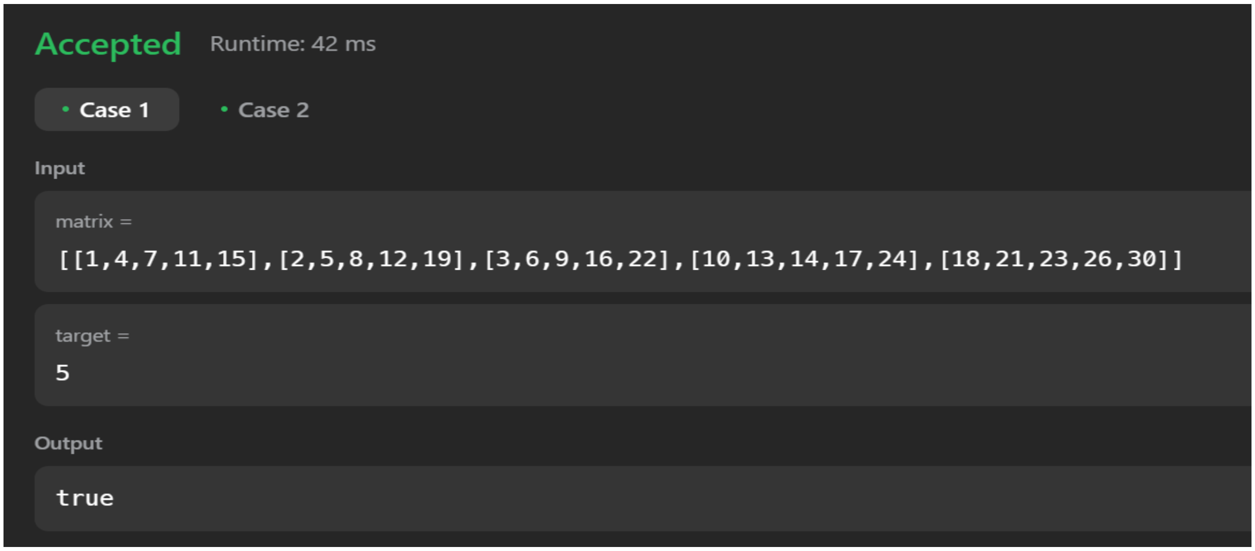
Problem 4.2: Search 2d matrix 2

Problem Statement: To search for a target value in a 2D matrix where each row and column is sorted in ascending order.

3.2. Code 4.2:

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        if not matrix or not matrix[0]:
            return False
        rows, cols = len(matrix), len(matrix[0])
        row, col = rows - 1, 0
        while row >= 0 and col < cols:
            if matrix[row][col] == target:
                return True
            elif matrix[row][col] > target:
                row -= 1
            else:
                col += 1
        return False
```

1.2. Output 4.2:



```
Accepted   Runtime: 42 ms

• Case 1   • Case 2

Input

matrix =
[[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]]

target =
5

Output

true
```

Fig 2: Output for Problem 4.2

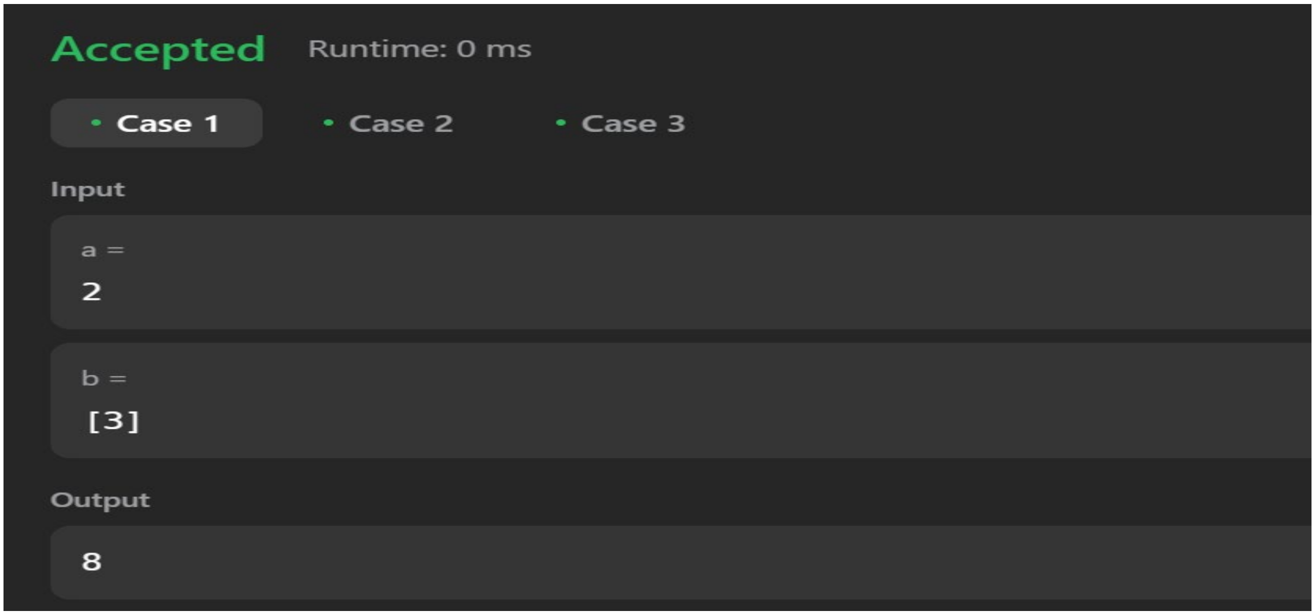
Problem 4.3: Super Pow

Problem Statement: To compute a large power of a number efficiently, given the base and an array of exponents.

3.3. Code 4.3:

```
class Solution:
    MOD = 1337
    def pow(self, a: int, b: int) -> int:
        result = 1
        a %= self.MOD  # Taking mod to prevent overflow
        for _ in range(b):
            result = (result * a) % self.MOD
        return result
    def superPow(self, a: int, b: list[int]) -> int:
        result = 1
        for i in range(len(b) - 1, -1, -1):
            result = (result * self.pow(a, b[i])) % self.MOD
            a = self.pow(a, 10)  # Power up for the next iteration
        return result
```

1.3. Output 4.3:



Accepted Runtime: 0 ms

• Case 1 • Case 2 • Case 3

Input

a =

2

b =

[3]

Output

8

Fig 3: Output for Problem 4.3

Problem 4.4: Beautiful Array

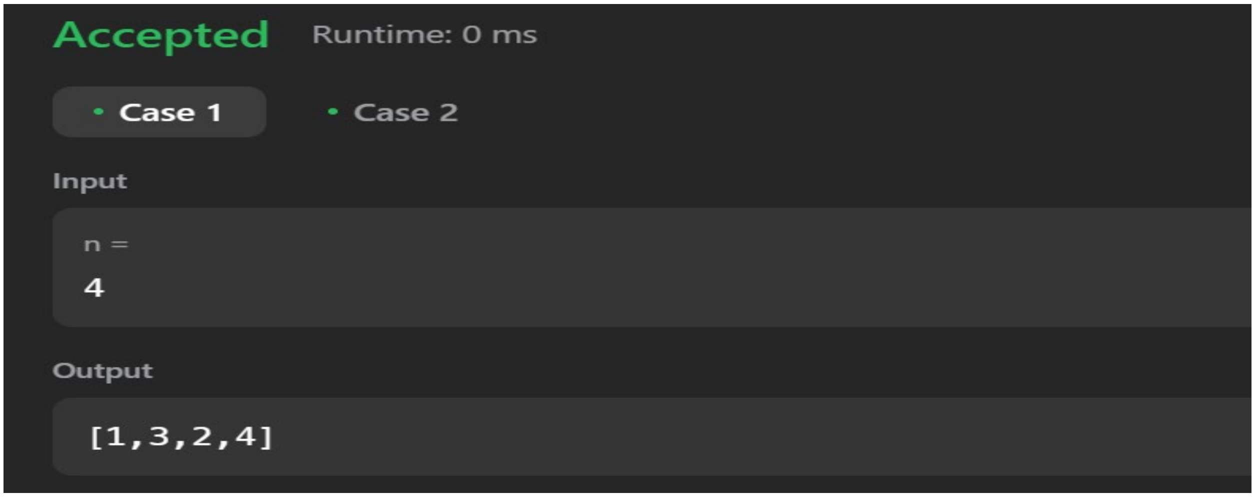
Problem Statement: To generate an array of integers that satisfies specific conditions regarding the arrangement of odd and even numbers.

3.4. Code 4.4:

```
class Solution:
    def beautifulArray(self, N: int) -> List[int]:
        def helper(n):
            if n == 1:
                return [1]
            odd = helper((n + 1) // 2)
            even = helper(n // 2)
            return [x * 2 - 1 for x in odd] + [x * 2 for x in even]

        return helper(N)
```

1.4. Output 4.4:



Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

```
n =
4
```

Output

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

Fig 4: Output for Problem 4.4

Problem 4.5: The Skyline Problem

Problem Statement: To determine the outline of a city skyline formed by a collection of rectangular buildings.

3.5. Code 4.5:

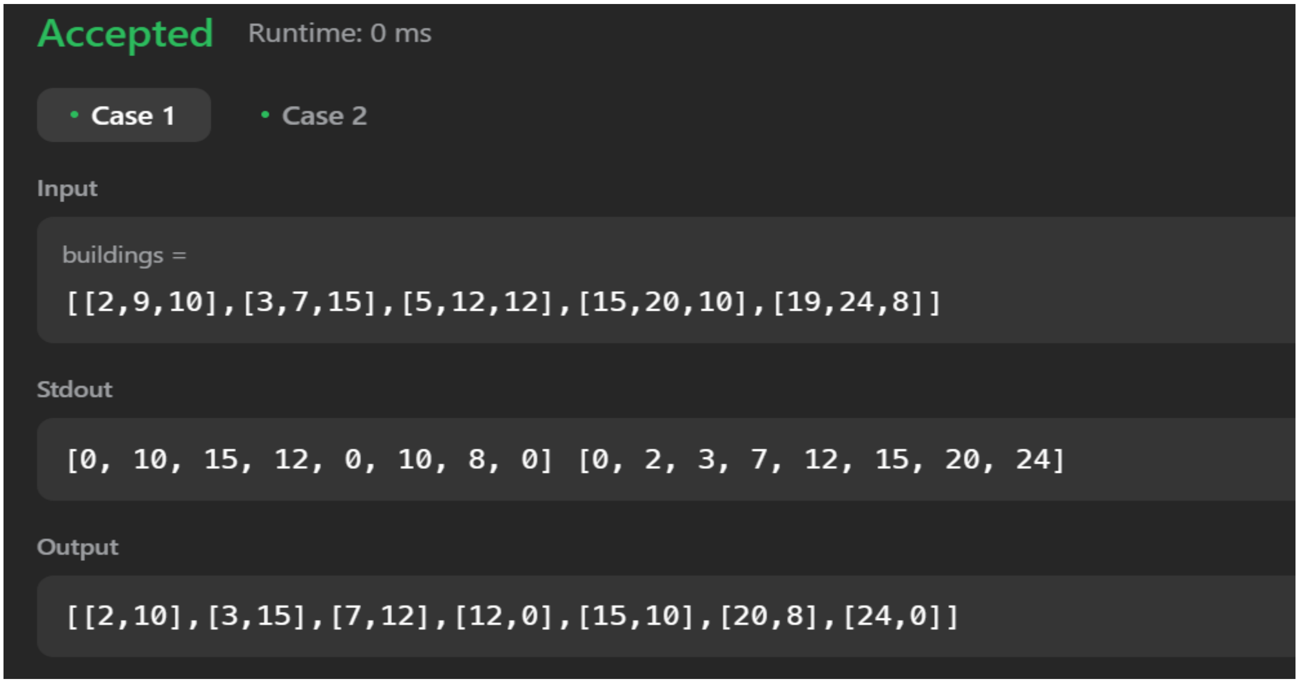
```
from sortedcontainers import SortedList
class Solution:
    def getSkyline(self, buildings: List[List[int]]) -> List[List[int]]:
        if len(buildings) == 0:
            return []

        buildings.sort(key=lambda v: v[2])
        pos, height = [0], [0]
        for left, right, h in buildings:
            i = bisect_left(pos, left)
            j = bisect_right(pos, right)
            height[i:j] = [h, height[j-1]]
            pos[i:j] = [left, right]
        print(height, pos)
        res = []
```

```
prev = 0
for v, h in zip(pos, height):
    if h != prev:
        res.append([v,h])
        prev = h

return res
```

1.5. Output 4.5:



Input

```
buildings =
[[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
```

Stdout

```
[0, 10, 15, 12, 0, 10, 8, 0] [0, 2, 3, 7, 12, 15, 20, 24]
```

Output

```
[[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
```

Fig 5: Output for Problem 4.5

Problem 4.6: Reverse Pairs

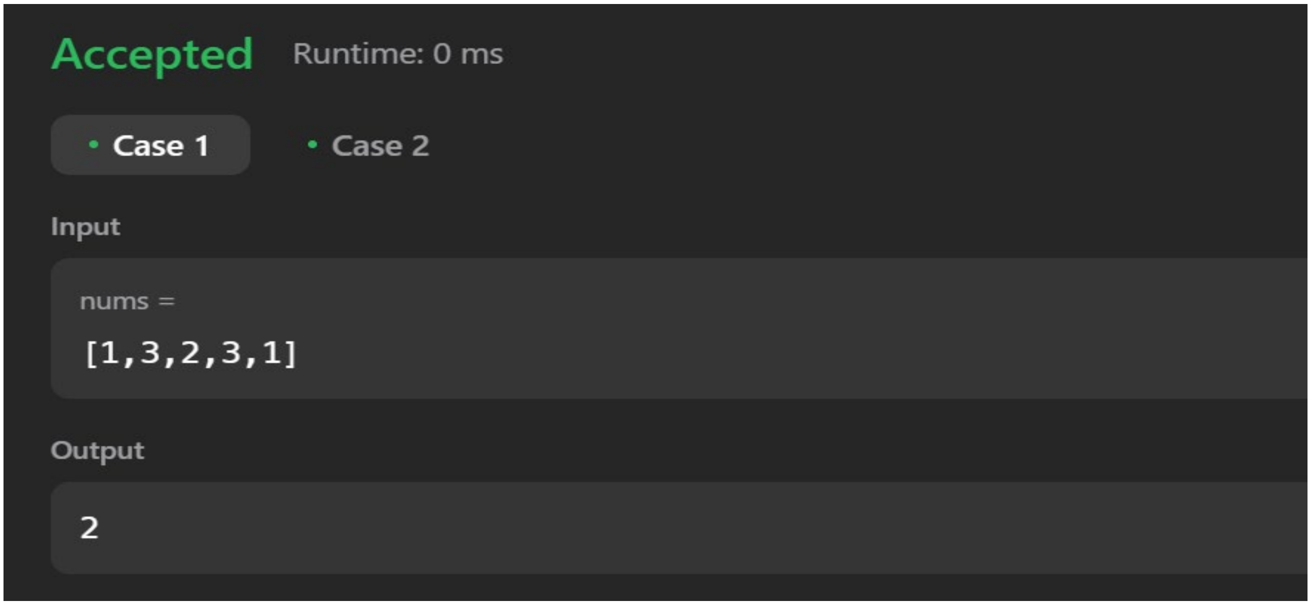
Problem Statement: To count the number of important reverse pairs in an array, where a pair (i, j) is considered important if i < j and nums[i] > 2 * nums[j].

3.6. Code 4.6:

```
class Solution:
```

```
def reversePairs(self, nums: List[int]) -> int:
    def merge(arr1,arr2):
        i = 0
        j = 0
        count = 0
        arr =[]
        x = 0
        while i < len(arr1) and j < len(arr2) :
            if arr1[i] <= arr2[j] :
                arr.append(arr1[i])
                while x < len(arr2) and arr1[i] > 2 * arr2[x] :
                    x = x + 1
                count = count + x
                i+=1
            else :
                arr.append(arr2[j])
                j +=1
        while i < len(arr1) :
            arr.append(arr1[i])
            while x < len(arr2) and arr1[i] > 2 * arr2[x] :
                x = x + 1
            count = count + x
            i+=1
        while j < len(arr2) :
            arr.append(arr2[j])
            j+=1
        return count , arr
    def helper(arr) :
        if len(arr) > 1:
            count1,arr1 = helper(arr[:len(arr)//2])
            count2,arr2 = helper(arr[len(arr)//2 : ])
            count3,arr3 = merge(arr1,arr2)
            return count1 + count2 + count3 , arr3
        return 0,arr
    return helper(nums)[0]
```

1.6. Output 4.6:



Accepted Runtime: 0 ms

Input

nums =

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

Output

```
2
```

Fig 6: Output for Problem 4.6

Problem 4.7: Longest increasing subsequence 2

Problem Statement: To find the length of the longest increasing subsequence in a sequence of numbers, allowing for a more complex input structure.

3.7. Code 4.7:

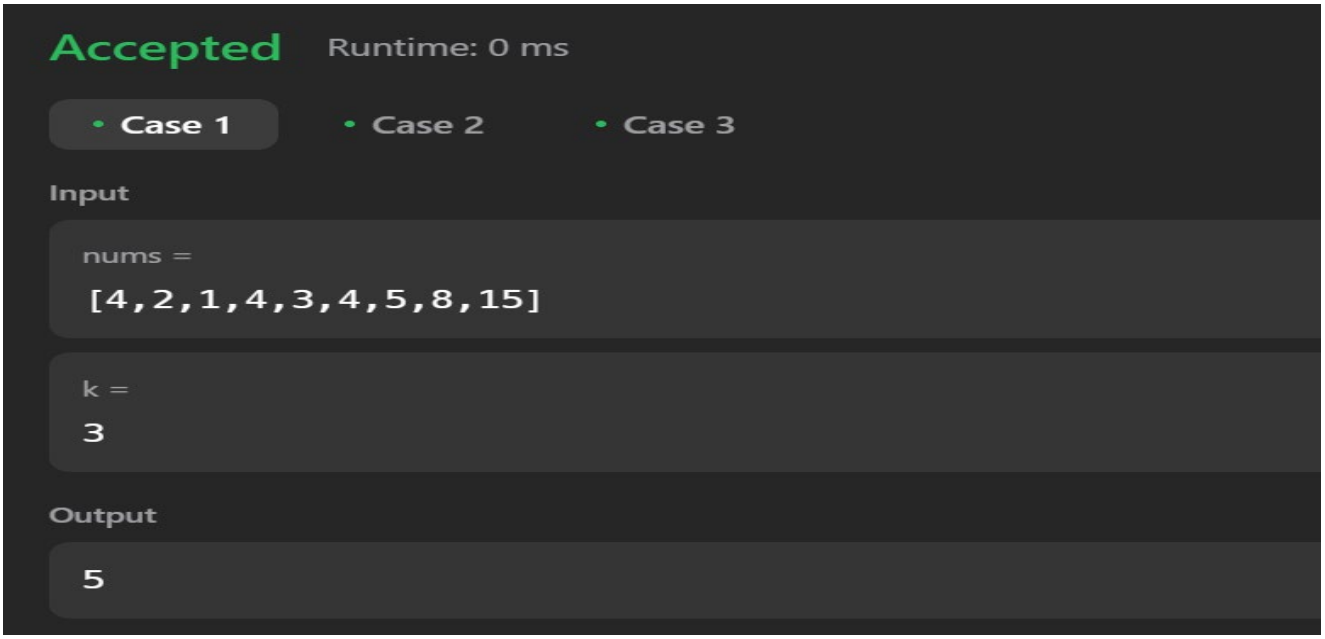
```
class SEG:
    def __init__(self, n):
        self.n = n
        self.tree = [0] * 2 * self.n

    def query(self, l, r):
        l += self.n
        r += self.n
        ans = 0
        while l < r:
            if l & 1:
                ans = max(ans, self.tree[l])
                l += 1
            if r & 1:
                r -= 1
                ans = max(ans, self.tree[r])
            l >>= 1
            r >>= 1
        return ans

    def update(self, i, val):
        i += self.n
        self.tree[i] = val
        while i > 1:
            i >>= 1
            self.tree[i] = max(self.tree[i * 2], self.tree[i * 2 + 1])

class Solution:
    def lengthOfLIS(self, A: List[int], k: int) -> int:
        n, ans = max(A), 1
        seg = SEG(n)
        for a in A:
            a -= 1
            premax = seg.query(max(0, a - k), a)
            ans = max(ans, premax + 1)
            seg.update(a, premax + 1)
        return ans
```

1.7. Output 4.7:



Accepted Runtime: 0 ms

```
Input

nums =
[4,2,1,4,3,4,5,8,15]

k =
3

Output

5
```

Fig 7: Output for Problem 4.7

5. Learning Outcome:

1) Algorithmic Thinking: Develop a deeper understanding of various algorithmic techniques, including dynamic programming, binary search, and divide-and-conquer strategies.
2) Problem-Solving Skills: Enhance your ability to break down complex problems into smaller, manageable parts and apply appropriate algorithms to solve them.
3) Data Structures: Gain familiarity with different data structures (arrays, lists, heaps) and their applications in solving algorithmic problems.
4) Efficiency: Learn to analyze the time and space complexity of algorithms, aiming for optimal solutions in competitive programming and technical interviews.