Experiment-7(A)

Student Name: Om Ankur Prajapati
UID: 22BCS15926
Branch: CSE
Section/Group: NTPP_602-A
Semester: 6
Date of Performance: 10-03-25
Subject Name: Advanced Programming Lab-2
Subject Code: 22CSH-359

1. **Title:** Greedy (Maximum Units on a Truck)

2. **Objective:** The objective is to maximize the total number of units loaded onto a truck given an array of box types and a truck size constraint.

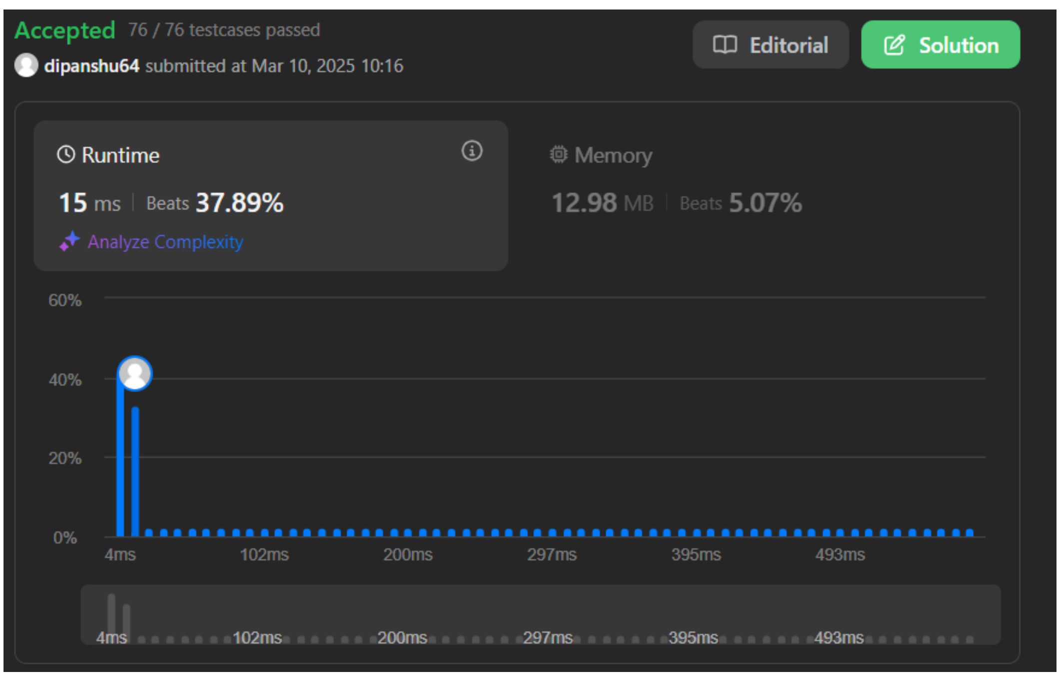
3. **Algorithm:**
   - **Input:** An array boxTypes where each element represents [numberOfBoxes, unitsPerBox] and an integer truckSize.
   - **Sorting Step:** Sort boxTypes in **descending order** based on the number of units per box.
   - **Initialization:**
     - maxUnits = 0 □ Stores the total units collected.
   - **Iteration:**
   - For each box in the sorted list:
     - If truckSize == 0, stop the iteration.
     - Add min(box[0], truckSize) * box[1] to maxUnits.
     - Subtract min(box[0], truckSize) from truckSize.
   - **Return maxUnits** as the maximum units loaded onto the truck.

4. **Implementation/Code:**

```
class Solution:
    def maximumUnits(self, boxTypes, truckSize):
        # Sort box types in descending order of units per box
```

```
        boxTypes.sort(key=lambda x: x[1], reverse=True)

        maxUnits = 0
        for boxes, units in boxTypes:
            if truckSize == 0:

                break
            count = min(boxes, truckSize)
            maxUnits += count * units
            truckSize -= count

        return maxUnits

# Example Usage
print(Solution().maximumUnits([[1,3],[2,2],[3,1]], 4))  # Output: 8
print(Solution().maximumUnits([[5,10],[2,5],[4,7],[3,9]], 10))  # Output:
91
```

5. <u>Output:</u>



6. <u>Time Complexity:</u> O (n log n)

7. <u>Space Complexity:</u> O(1)

Experiment 7(B)

1. **Title:** Maximum Subarray

2. **Objective:** To find the contiguous subarray with the largest sum in a given integer array.

3. **Algorithm:**

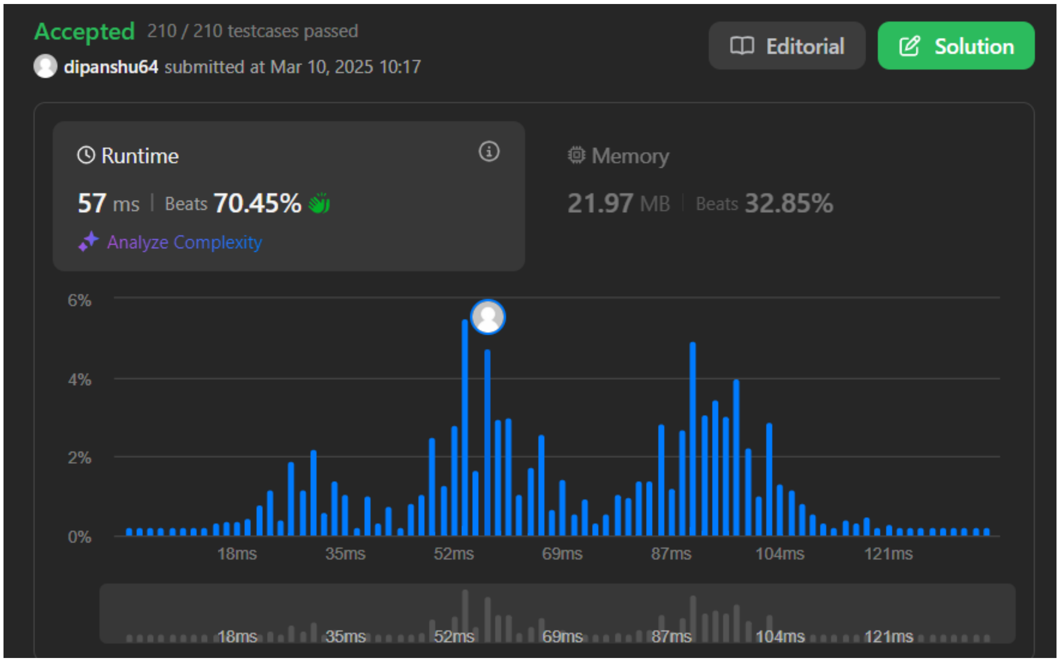
   1. **Input:** An array `nums` containing integers.
   2. **Initialization:**
      - `currentSum = 0` (Tracks the sum of the current subarray)
      - `maxSum = -∞` (Tracks the maximum subarray sum found so far)
   3. **Iteration:**
      - For each element `num` in `nums`:

   1. Add `num` to `currentSum`.
   2. Update `maxSum = max(maxSum, currentSum)`.
   3. If `currentSum` becomes negative, reset it to zero.
   4. **Return `maxSum`** as the maximum subarray sum.

4. **Implementation/Code:**

```
class Solution:
    def maxSubArray(self, nums):
        currentSum = 0
        maxSum = float('-inf')

        for num in nums:
            currentSum += num
            maxSum = max(maxSum, currentSum)
            if currentSum < 0:
                currentSum = 0
                return maxSum

# Example Usage
print(Solution().maxSubArray([-2,1,-3,4,-1,2,1,-5,4]))
print(Solution().maxSubArray([5,4,-1,7,8]))
```

6. Output:



8. Time Complexity: O(n)

9. Space Complexity: O(1)

10. Learning Outcome:

- Mastered Kadane's Algorithm for maximum subarray sum.
- Learned efficient handling of negative values in subarrays.
- Improved understanding of optimal subarray identification in `O(n)` time.

Experiment 7(C)

1. **Title:** Remove Stones to Minimize the Total

2. **Objective:** To minimize the total number of stones remaining in the piles by repeatedly removing floor(piles[i] / 2) stones from the pile with the maximum stones, exactly k times.

3.Algorithm:

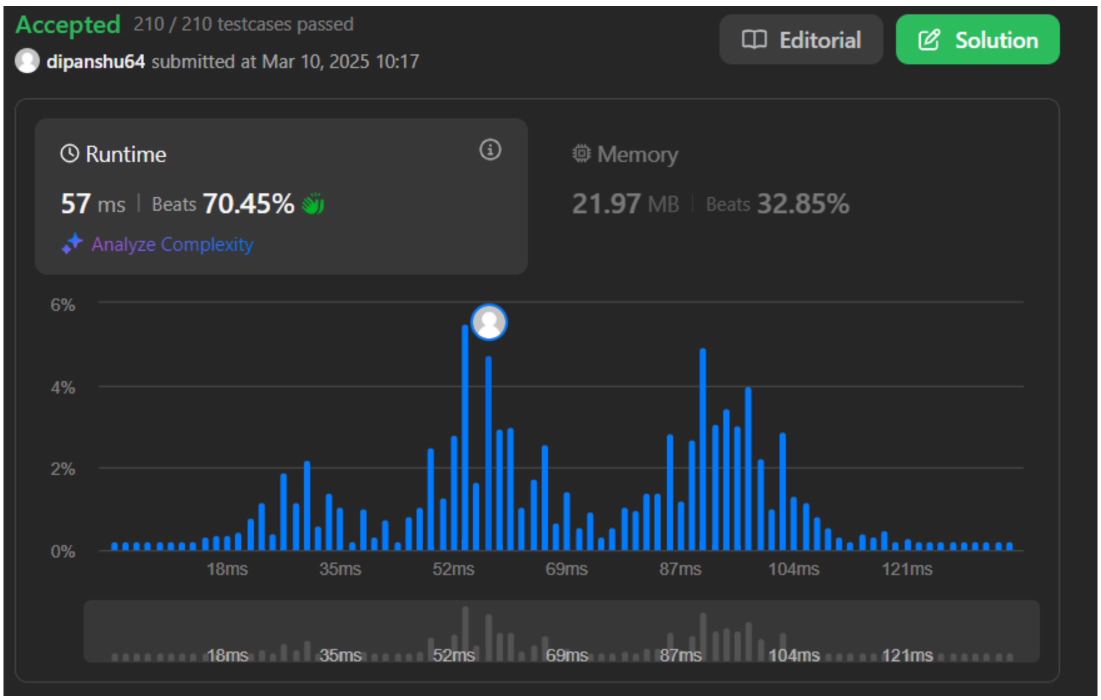
- **Input:** An array `piles` representing the number of stones in each pile, and an integer `k`.
- **Max Heap Conversion:**
  - Convert all elements of `piles` into their **negative values** and insert them into a **heap** (since Python's `heapq` only supports min-heaps, negating values makes it behave like a max-heap).
- **Perform `k` Operations:**
  - Repeat `k` times:
    - Extract the largest element from the heap (using `heapq.heappop`).
    - Remove `floor(largest / 2)` stones from it.
    - Insert the updated pile value back into the heap.
- **Calculate Remaining Stones:**
  - Sum all the remaining elements in the heap (converting them back to positive values).
- **Return** the total number of remaining stones.

5. Implementation/Code:

```
import heapq
import math

class Solution:
    def minStoneSum(self, piles, k):
        # Convert to negative values to simulate max heap
        max_heap = [-pile for pile in piles]
        heapq.heapify(max_heap)

        # Perform k operations
        for _ in range(k):
            largest = -heapq.heappop(max_heap)  # Get the max element
            reduced_stones = largest - largest // 2  # Correct floor
division
```

```
                heapq.heappush(max_heap, -reduced_stones)  # Push back the
reduced pile

        # Calculate remaining stones (Convert back to positive)
        return int(-sum(max_heap))

# Example Usage
print(Solution().minStoneSum([5, 4, 9], 2))  # Output: 12
print(Solution().minStoneSum([4, 3, 6, 7], 3))  # Output: 12
```

6. Output:



8. Time Complexity: O(k log n)

9. Space Complexity: O(n)

10. LearningOutcomes:

- Mastered Kadane's Algorithm for maximum subarray sum.
- Implemented DP solutions for optimal substructure problems.
- Applied Greedy Approach for maximizing outcomes.
- Utilized Max Heap for efficient element extraction and modification.
- Improved understanding of optimization techniques.