02. Solved Coding PYQs - Array & Math Problems

Problem 1: Second Smallest and Second Largest Element

Problem Statement

Given an array of N integers, write a program to find the **second smallest** and **second largest** distinct elements in the array in $O(N)$ time complexity without sorting. Return `-1` if the second smallest or second largest does not exist.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

void findSecondElements(const vector<int>& arr) {
    if (arr.size() < 2) {
        cout << "Second Smallest: -1, Second Largest: -1" << endl;
        return;
    }

    int smallest = INT_MAX, secondSmallest = INT_MAX;
    int largest = INT_MIN, secondLargest = INT_MIN;

    for (int num : arr) {
        // Smallest tracking
        if (num < smallest) {
            secondSmallest = smallest;
            smallest = num;
        } else if (num < secondSmallest && num != smallest) {
            secondSmallest = num;
        }

        // Largest tracking
        if (num > largest) {
            secondLargest = largest;
            largest = num;
        } else if (num > secondLargest && num != largest) {
            secondLargest = num;
        }
    }
```

```
    int resSmallest = (secondSmallest == INT_MAX) ? -1 : secondSmallest;
    int resLargest = (secondLargest == INT_MIN) ? -1 : secondLargest;

    cout << "Second Smallest: " << resSmallest << ", Second Largest: " << resLargest <<
endl;
}

int main() {
    vector<int> arr = {12, 35, 1, 10, 34, 1};
    findSecondElements(arr);
    return 0;
}
```

Java Solution

```
import java.util.*;

public class SecondElements {
    public static void findSecondElements(int[] arr) {
        if (arr == null || arr.length < 2) {
            System.out.println("Second Smallest: -1, Second Largest: -1");
            return;
        }

        int smallest = Integer.MAX_VALUE, secondSmallest = Integer.MAX_VALUE;
        int largest = Integer.MIN_VALUE, secondLargest = Integer.MIN_VALUE;

        for (int num : arr) {
            if (num < smallest) {
                secondSmallest = smallest;
                smallest = num;
            } else if (num < secondSmallest && num != smallest) {
                secondSmallest = num;
            }

            if (num > largest) {
                secondLargest = largest;
                largest = num;
            } else if (num > secondLargest && num != largest) {
                secondLargest = num;
            }
        }

        int resSmallest = (secondSmallest == Integer.MAX_VALUE) ? -1 : secondSmallest;
        int resLargest = (secondLargest == Integer.MIN_VALUE) ? -1 : secondLargest;

        System.out.println("Second Smallest: " + resSmallest + ", Second Largest: " +
resLargest);
```

```
    }

    public static void main(String[] args) {
        int[] arr = {12, 35, 1, 10, 34, 1};
        findSecondElements(arr);
    }
}
```

Python Solution

```
def find_second_elements(arr):
    if len(arr) < 2:
        return -1, -1

    smallest = second_smallest = float('inf')
    largest = second_largest = float('-inf')

    for num in arr:
        if num < smallest:
            second_smallest = smallest
            smallest = num
        elif num < second_smallest and num != smallest:
            second_smallest = num

        if num > largest:
            second_largest = largest
            largest = num
        elif num > second_largest and num != largest:
            second_largest = num

    res_smallest = -1 if second_smallest == float('inf') else second_smallest
    res_largest = -1 if second_largest == float('-inf') else second_largest

    return res_smallest, res_largest

# Driver test
arr = [12, 35, 1, 10, 34, 1]
sec_small, sec_large = find_second_elements(arr)
print(f"Second Smallest: {sec_small}, Second Largest: {sec_large}")
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — Single pass over array of size N.
- **Space Complexity**: $\mathcal{O}(1)$ — Constant auxiliary variables.

Problem 2: Find Single Non-Repeating Element (Bitwise XOR)

Problem Statement

Given an array where every element appears twice except for one element which appears once, find that single element in $O(N)$ time and $O(1)$ space.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>

using namespace std;

int findSingle(const vector<int>& arr) {
    int res = 0;
    for (int num : arr) {
        res ^= num; // Bitwise XOR: x ^ x = 0, x ^ 0 = x
    }
    return res;
}

int main() {
    vector<int> arr = {4, 1, 2, 1, 2};
    cout << "Single Element: " << findSingle(arr) << endl; // Output: 4
    return 0;
}
```

Java Solution

```
public class SingleElement {
    public static int findSingle(int[] arr) {
        int res = 0;
        for (int num : arr) {
            res ^= num; // Bitwise XOR properties
        }
        return res;
    }

    public static void main(String[] args) {
        int[] arr = {4, 1, 2, 1, 2};
        System.out.println("Single Element: " + findSingle(arr)); // Output: 4
    }
}
```

Python Solution

```
def find_single(arr):
    res = 0
    for num in arr:
        res ^= num  # Using property: x ^ x = 0 and x ^ 0 = x
    return res

# Driver test
arr = [4, 1, 2, 1, 2]
print("Single Element:", find_single(arr))  # Output: 4
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — One linear scan of the input array.
- **Space Complexity**: $\mathcal{O}(1)$ — In-place XOR accumulation.

Problem 3: Maximum Subarray Sum (Kadane's Algorithm)

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int maxSubArraySum(const vector<int>& nums) {
    if (nums.empty()) return 0;
    int maxSoFar = nums[0];
    int currentMax = nums[0];

    for (size_t i = 1; i < nums.size(); ++i) {
        currentMax = max(nums[i], currentMax + nums[i]);
        maxSoFar = max(maxSoFar, currentMax);
    }
    return maxSoFar;
}
```

```
int main() {
    vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
    cout << "Maximum Subarray Sum: " << maxSubArraySum(nums) << endl; // Output: 6
    return 0;
}
```

Java Solution

```
public class Kadane {
    public static int maxSubArraySum(int[] nums) {
        if (nums == null || nums.length == 0) return 0;
        int maxSoFar = nums[0];
        int currentMax = nums[0];

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

    public static void main(String[] args) {
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("Maximum Subarray Sum: " + maxSubArraySum(nums)); // Output: 6
    }
}
```

Python Solution

```
def max_subarray_sum(nums):
    if not nums:
        return 0
    max_so_far = nums[0]
    current_max = nums[0]

    for num in nums[1:]:
        current_max = max(num, current_max + num)
        max_so_far = max(max_so_far, current_max)
    return max_so_far

# Driver test
nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("Maximum Subarray Sum:", max_subarray_sum(nums))  # Output: 6
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — Traverses the array once.

- **Space Complexity**: $\mathcal{O}(1)$ — Uses constant extra memory.

Problem 4: Equilibrium Index of an Array

Problem Statement

An equilibrium index of an array is an index such that the sum of elements at lower indices is equal to the sum of elements at higher indices. Return the first equilibrium index found, or `-1` if no equilibrium index exists.

Solution Implementations

C++ Solution

```
#include <iostream>
#include <vector>

using namespace std;

int findEquilibriumIndex(const vector<int>& arr) {
    long long totalSum = 0;
    for (int num : arr) totalSum += num;

    long long leftSum = 0;
    for (size_t i = 0; i < arr.size(); ++i) {
        totalSum -= arr[i]; // totalSum is now rightSum
        if (leftSum == totalSum) {
            return (int)i;
        }
        leftSum += arr[i];
    }
    return -1;
}

int main() {
    vector<int> arr = {-7, 1, 5, 2, -4, 3, 0};
    cout << "Equilibrium Index: " << findEquilibriumIndex(arr) << endl; // Output: 3
    return 0;
}
```

Java Solution

```
public class EquilibriumIndex {
    public static int findEquilibriumIndex(int[] arr) {
        if (arr == null) return -1;
        long totalSum = 0;
        for (int num : arr) totalSum += num;
```

```
        long leftSum = 0;
        for (int i = 0; i < arr.length; i++) {
            totalSum -= arr[i]; // totalSum now acts as rightSum
            if (leftSum == totalSum) {
                return i;
            }
            leftSum += arr[i];
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] arr = {-7, 1, 5, 2, -4, 3, 0};
        System.out.println("Equilibrium Index: " + findEquilibriumIndex(arr)); // Output:
3
    }
}
```

Python Solution

```
def find_equilibrium_index(arr):
    total_sum = sum(arr)
    left_sum = 0

    for i, num in enumerate(arr):
        total_sum -= num  # total_sum now represents right_sum
        if left_sum == total_sum:
            return i
        left_sum += num

    return -1

# Driver test
arr = [-7, 1, 5, 2, -4, 3, 0]
print("Equilibrium Index:", find_equilibrium_index(arr))  # Output: 3
```

Complexity Analysis

- **Time Complexity**: $\mathcal{O}(N)$ — Two linear iterations (one for total sum, one for left/right sum check).
- **Space Complexity**: $\mathcal{O}(1)$ — No additional array memory allocated.