Task 2: Majority Element

(Find the Majority element of an array A[1 ... n]: An array is said to have a majority element if more than half of its entries are the same.)

Team members (Team 24):

Seif-Elden Mohammed Saleh [20210439]

Abdelrahman Khaled Mohammed [20210503]

Aly Mohammed Aly Ibrahim [20210581]

Omar Zakaria Mohy El-deen [20210603]

Abdelrahman Mohammed Ahmed Saber [20210523]

Ali Ahmed Abdelmawgood [20210573]

1.Recursive pseudocode:

```
Function: recursive()

Input: None
Output: an integer x

1. freq_size ← 5 * 10^4
2. Allocate memory for freq with size freq_size * sizeof(int)
3. small_array_size ← read integer from input
4. Allocate memory for arr with size small_array_size * sizeof(int)
5. For i from 0 to 5 * 10^4 do
6.    freq[i] ← 0
7. End for
8. For i from 0 to small_array_size - 1 do
9.    arr[i] ← read integer from input
10. End for
11. Call recursive_frequency_array(arr, small_array_size, freq)
12. x ← solve(freq, small_array_size, arr)
13. Free memory allocated for freq
14. Free memory allocated for arr
15. Return x

Function: recursive_frequency_array(arr, small_array_size, freq)

Input: an array arr of integers, an integer small_array_size representing the size of arr, an array freq
of integers representing the frequency of each number in arr
Output: None

1. If small_array_size equals 0, return
2. first ← arr[0]
3. freq[first] ← freq[first] + 1
4. Call recursive_frequency_array(arr + 1, small_array_size - 1, freq)
```

```
Function: solve(freq, small_array_size, arr)


Input: an array freq of integers representing the frequency of each number in arr, an integer
small_array_size representing the size of arr, an array arr of integers

Output: an integer x


1. mx ← INT_MIN

2. num ← INT_MIN

3. For i from 0 to small_array_size - 1 do

4.    If mx is less than freq[arr[i]] then

5.        mx ← freq[arr[i]]

6.        num ← arr[i]

7.    End if

8. End for

9. If mx is greater than small_array_size / 2 then

10.   Return num

11. End if

12. Return INT_MIN
```

- Time complexity:

  $T(n) = T(n-1) + O(1)$

  $= T(n-2) + 2 * O(1)$

  $= T(n-3) + 3 * O(1)$

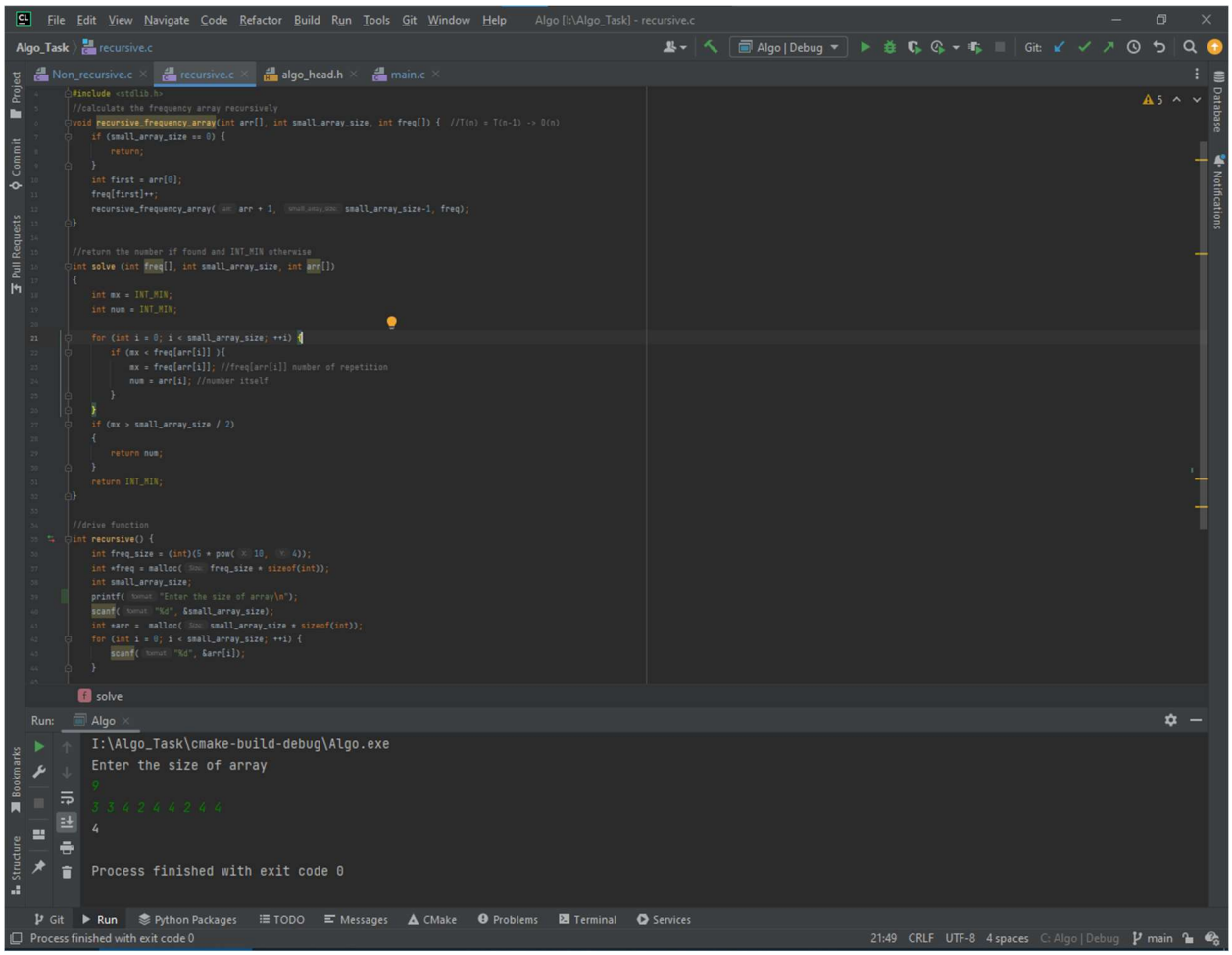
  $= ...$

  $= T(0) + n * O(1)$

  $= O(n)$

- Space complexity: O(n) size of input array.

Output preview:



2.Non-Recursive pseudocode:

```
Function: non_recursive()

Input: None
Output: an integer x

 1.  size ← 5 * 10^4
 2. Allocate memory for freq with size size * sizeof(int)
 3. small_array_size ← read integer from input
 4. For i from 0 to size - 1 do
 5.    freq[i] ← 0
 6. End for
 7. For i from 0 to small_array_size - 1 do
 8.    x ← read integer from input
 9.    freq[x] ← freq[x] + 1
 10. End for
 11. mx ← INT_MIN
 12. num ← INT_MIN
 13. For i from 0 to size - 1 do
 14.    If mx is less than freq[i] then
 15.        mx ← freq[i]
 16.        num ← i
 17.    End if
 18. End for
 19. If mx is greater than small_array_size / 2 then
 20.   Free memory allocated for freq
 21.   Return num
 22. End if
 23. Free memory allocated for freq
 24. Return INT_MIN
```

- Time complexity:

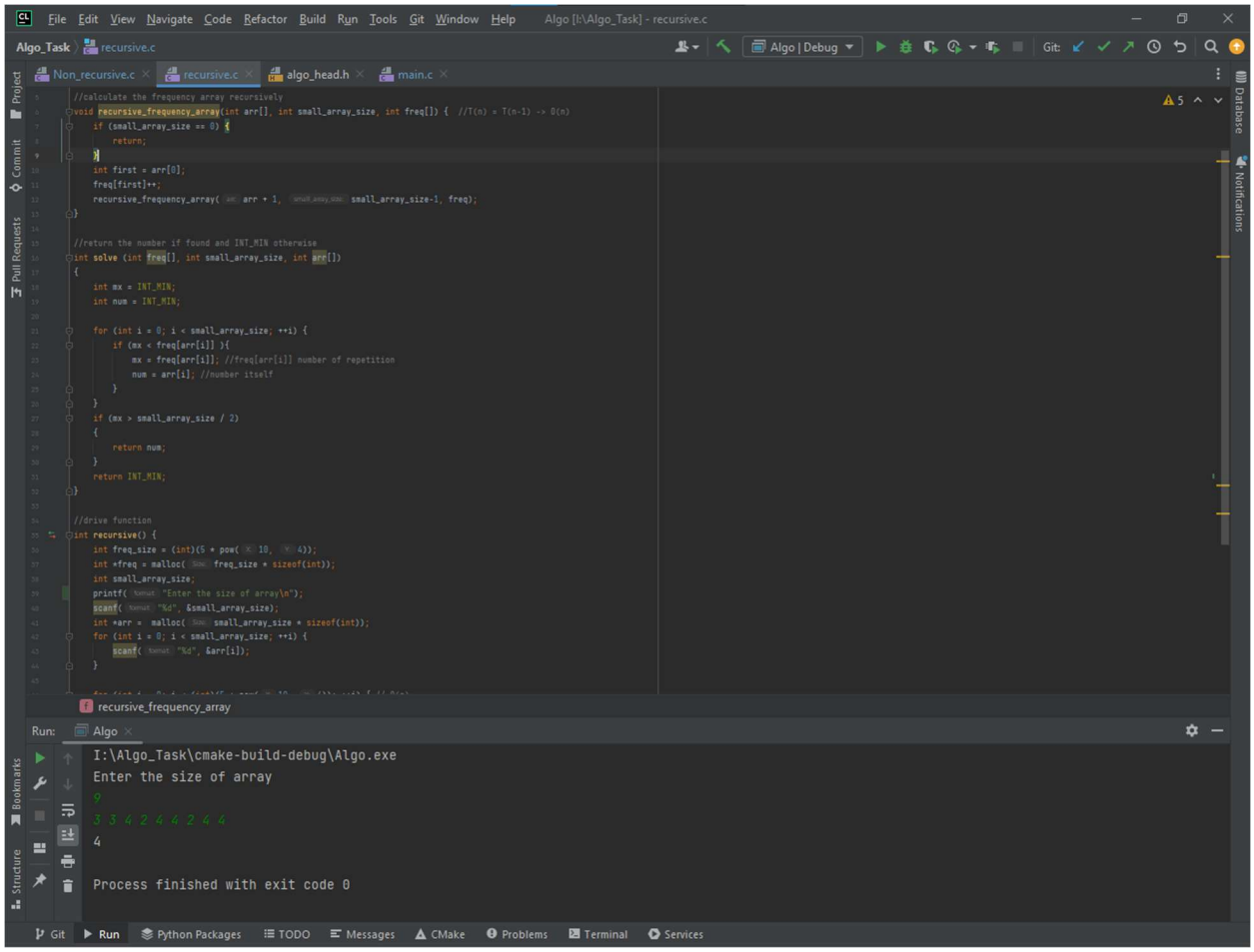
  $\sum_{i=0}^{n} 1$

  = n – 0 + 1

  = O(n)

- Space complexity: O(1) The size is constant for all inputs.

Output preview:



Comparison Table:

	Recursive	Non-Recursive
Time complexity	O(n)	O(n)
Space complexity	O(n)	O(1)

- Time complexity: both are the same.
- Space complexity: non-recursive algorithm consumes less space.