

# ALGORITHM TASK

Task number: 8

Task name: Maximal Subarray

	id	Name
1	20210656	Fady hany abdelmalak messyha
2	20210544	Abdallah Raafat Abdelhamid Helal
3	20210586	Ali moataz ali mahmoud
4	20210947	مصطفى محمود السيد محمود
5	20210594	Omar ahmed mohamed aly

# For the non Recursive code

## pseudocode :

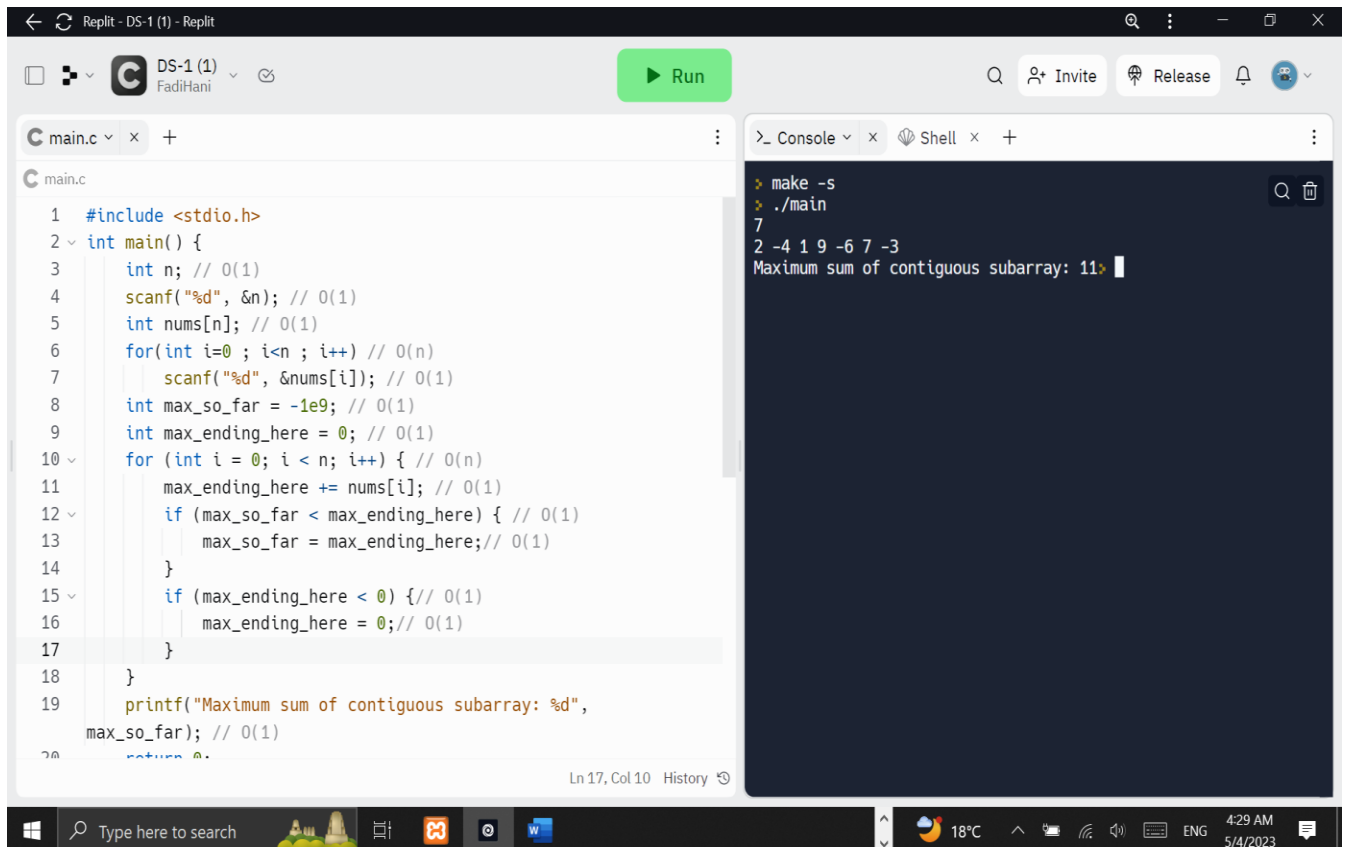
1. Declare an integer variable n ( Array size ).
2. Read an integer value into n using scanf.
3. Declare an integer array nums of size n.
4. Read n integer values into the array nums using a for loop and scanf.
5. Declare two integer variables max\_so\_far and max\_ending\_here, both initialized to 0.
6. Set max\_so\_far to -1e9.
7. Iterate over the elements of the array nums using a for loop:
8. {
9. Add the current element of the array to max\_ending\_here.
10. If max\_so\_far is less than max\_ending\_here, set max\_so\_far to max\_ending\_here.
11. If max\_ending\_here is less than 0, set it to 0.
12. }
13. Print the value of max\_so\_far using printf.
14. End the program.

## Time complexity analysis:

1. Initializing variables n, nums[n], max\_so\_far, and max\_ending\_here takes **O(1) time**.
2. The first for loop iterates over n elements of the array, taking **O(n) time**.
3. Inside the first for loop, each operation takes **O(1) time**, which includes taking input from the user using scanf, and adding elements to nums.
4. The second for loop also iterates over n elements of the array, taking **O(n) time**.
5. Inside the second for loop, each operation takes **O(1) time**, which includes adding elements to max\_ending\_here, comparing max\_so\_far and max\_ending\_here, and resetting max\_ending\_here if it becomes negative.
6. The printf statement takes **O(1) time**.
7. Therefore, the time complexity of the code is **O(n)**.

# Space complexity analysis:

1. The code declares an integer variable  $n$ , which takes  $O(1)$  space.
2. The array `nums[n]` takes  $O(n)$  space.
3. The other integer variables `max_so_far` and `max_ending_here` take  $O(1)$  space.
4. Therefore, the space complexity of the code is  $O(n)$  due to the array.



```
1 #include <stdio.h>
2 int main() {
3     int n; // O(1)
4     scanf("%d", &n); // O(1)
5     int nums[n]; // O(n)
6     for(int i=0 ; i<n ; i++) // O(n)
7         scanf("%d", &nums[i]); // O(1)
8     int max_so_far = -1e9; // O(1)
9     int max_ending_here = 0; // O(1)
10    for (int i = 0; i < n; i++) { // O(n)
11        max_ending_here += nums[i]; // O(1)
12        if (max_so_far < max_ending_here) { // O(1)
13            max_so_far = max_ending_here; // O(1)
14        }
15        if (max_ending_here < 0) { // O(1)
16            max_ending_here = 0; // O(1)
17        }
18    }
19    printf("Maximum sum of contiguous subarray: %d",
20    max_so_far); // O(1)
21    return 0;
}
```

```
> make -s
> ./main
7
2 -4 1 9 -6 7 -3
Maximum sum of contiguous subarray: 11>
```

## For the Recursive code

### pseudocode :

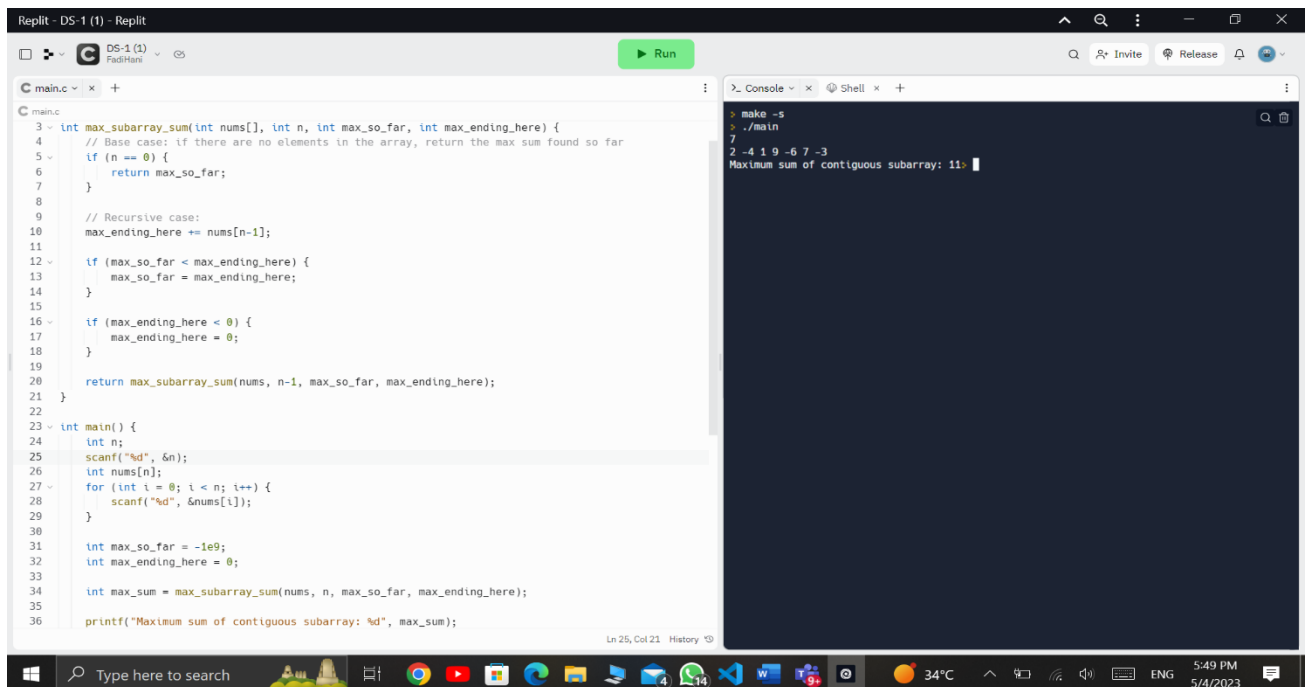
1. Declare an integer variable `n` ( Array size ).
2. Read an integer value into `n` using `scanf`.
3. Declare an integer array `nums` of size `n`.
4. Read `n` integer values into the array `nums` using a for loop and `scanf`.
5. Declare two integer variables (`max_so_far`) and (`max_ending_here`), both initialized to 0.
6. Set `max_so_far` to `-1e9`.
7. Declare (`max_sum`) as a variable to store the return value of our function (`max_subarray_sum`)
8. Calling the function (`max_subarray_sum`) that takes the array ,his size , (`max_so_far`), (`max_ending_here`)
9. If the array size equal to zero return (`max_so_far`)
10. Add the current element of the array to `max_ending_here` without the last array's element
11. If (`max_so_far`) is less than (`max_ending_here`), set (`max_so_far`) to (`max_ending_here`).
12. If `max_ending_here` is less than 0, set it to 0.
13. Return the function by calling it again (recursive function) but with (`N-1`) (array size-1)
14. After the code finish the array will find that (array size(`n`)=0)
15. return (`max_so_far`)
16. set `max_so_far`(function's return) to `max_sum`
17. Print the value of `max_sum` using `printf`.
18. End the program.

### Time complexity analysis:

1. The time complexity of the given code is  $O(n)$ , where `n` is the number of elements in the array. The function `max_subarray_sum` is called recursively `n` times, each time reducing the size of the array by The operations inside the function `max_subarray_sum` are all constant time operations, so the total time complexity of the code is  $O(n)$ .

# Space complexity analysis:

1. The space complexity of the code is  $O(1)$  because the amount of space used does not depend on the size of the input array. The only variables used are `n`, `nums`, `max\_so\_far`, `max\_ending\_here`, and `max\_sum`, all of which take constant space. The recursive calls to the `max\_subarray\_sum` function do not use any additional space on the stack because the arguments are passed by value and not by reference.



```
main.c
3 int max_subarray_sum(int nums[], int n, int max_so_far, int max_ending_here) {
4     // Base case: if there are no elements in the array, return the max sum found so far
5     if (n == 0) {
6         return max_so_far;
7     }
8
9     // Recursive case:
10    max_ending_here += nums[n-1];
11
12    if (max_so_far < max_ending_here) {
13        max_so_far = max_ending_here;
14    }
15
16    if (max_ending_here < 0) {
17        max_ending_here = 0;
18    }
19
20    return max_subarray_sum(nums, n-1, max_so_far, max_ending_here);
21 }
22
23 int main() {
24     int n;
25     scanf("%d", &n);
26     int nums[n];
27     for (int i = 0; i < n; i++) {
28         scanf("%d", &nums[i]);
29     }
30
31     int max_so_far = -1e9;
32     int max_ending_here = 0;
33
34     int max_sum = max_subarray_sum(nums, n, max_so_far, max_ending_here);
35
36     printf("Maximum sum of contiguous subarray: %d", max_sum);
37 }
```

```
_ Console
> make -s
> ./main
2 -4 1 9 -6 7 -3
Maximum sum of contiguous subarray: 11
```

## Comparison Table:

	Non recursive	recursive
Time complexity	the time complexity of the this code is $O(n)$ .	the total time complexity of the code is $O(n)$
Space complexity	the space complexity of the code is $O(n)$	The space complexity of the code is $O(1)$

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