Task 2 : Triangle

Team Number: 18

ID Name

20210876 محمود محمد احمد سيد

20210481 عادل علي إبراهيم علي

20220306 عمار محمد عبدالمنعم الجوهري

20220156 رحمة جمال ابوكلوب محمد

20220045 احمد هاني إبراهيم الفرماوي

20220085 اوليفر عطية عيد كامل

Non-recursive algorithm:

Code:

**function** isTriangular**(**arr**)** **{**

**let** size **=** 0**;**

**while** **(**arr**[**size**]** **!==** undefined**)** **{**

size**++;**

**}**

**for** **(let** i **=** 0**;** i **<=** size **-** 3**;** i**++)** **{**

**for** **(let** j **=** i **+** 1**;** j **<=** size **-** 2**;** j**++)** **{**

**for** **(let** k **=** j **+** 1**;** k **<=** size **-** 1**;** k**++)** **{**

**if** **(**

arr**[**i**]** **+** arr**[**j**]** **>** arr**[**k**]** **&&**

arr**[**j**]** **+** arr**[**k**]** **>** arr**[**i**]** **&&**

arr**[**i**]** **+** arr**[**k**]** **>** arr**[**j**]**

**)**

**return** 1**;**

**}**

**}**

**}**

**return** 0**;**

**}**

**const** nums1 **=** **[**3**,** 1**,** 1**];**

**const** nums2 **=** **[**10**,** 2**,** 5**,** 1**,** 8**,** 20**];**

console**.**log**(**"Example 1:"**);**

console**.**log**(**"Output:"**,**isTriangular**(**nums1**));**

console**.**log**(**"\nExample 2:"**);**

console**.**log**(**"Output:"**,**isTriangular**(**nums2**));**

Pseudocode:

Algorithm isTriangular**(**arr**)**

//arr is an array of size n

{

n := 0

while arr[n] != undefined do

{

n++

}

**for** i **:=** 0 **to** n**-**3 **do**

{

for j := i+1 to n-2 do

{

for k := j+1 to n-1 do

{

if (arr[i] + arr[j] > arr[k]) AND (arr[j] + arr[k] > arr[i]) AND

(arr[i] + arr[k] > arr[j]) then

return 1

}

**}**

**}**

return 0

**}**

Initialization:

Set n to 0.

Start a while loop that iterates until it finds an undefined element in the array arr, incrementing n each time to determine the size of the array.

Main Loop:

Once the size of the array is determined, iterate through the array using three nested loops.

The outer loop (i) iterates from 0 to n-3, ensuring that enough elements remain in the array for a triangular triplet.

The middle loop (j) iterates from i+1 to n-2, ensuring that the middle element in the triplet is ahead of the first element (i) and that there are enough elements after it for the third element (k).

The innermost loop (k) iterates from j+1 to n-1, ensuring that the last element in the triplet is ahead of both the first and second elements.

Triangular Check:

Inside the innermost loop, it checks if the current triplet (arr[i], arr[j], arr[k]) satisfies the triangular inequality condition: the sum of the lengths of any two sides of a triangle must be greater than the length of the third side.

If the condition is met for the current triplet, it returns 1, indicating that a triangular triplet exists in the array.

Return:

If no triangular triplet is found after checking all possible combinations, the function returns 0, indicating that no triangular triplet exists in the array.

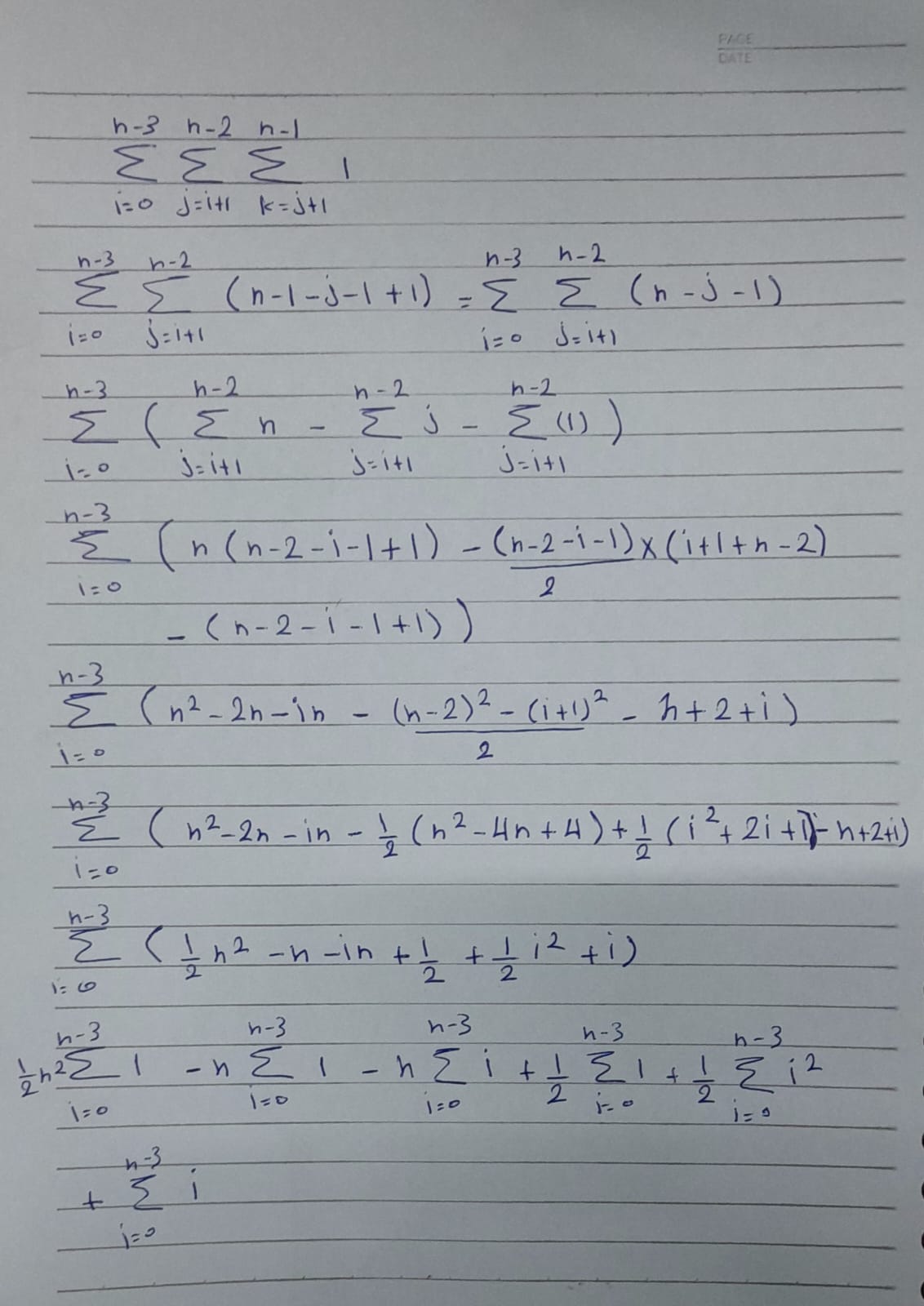
Complexity:

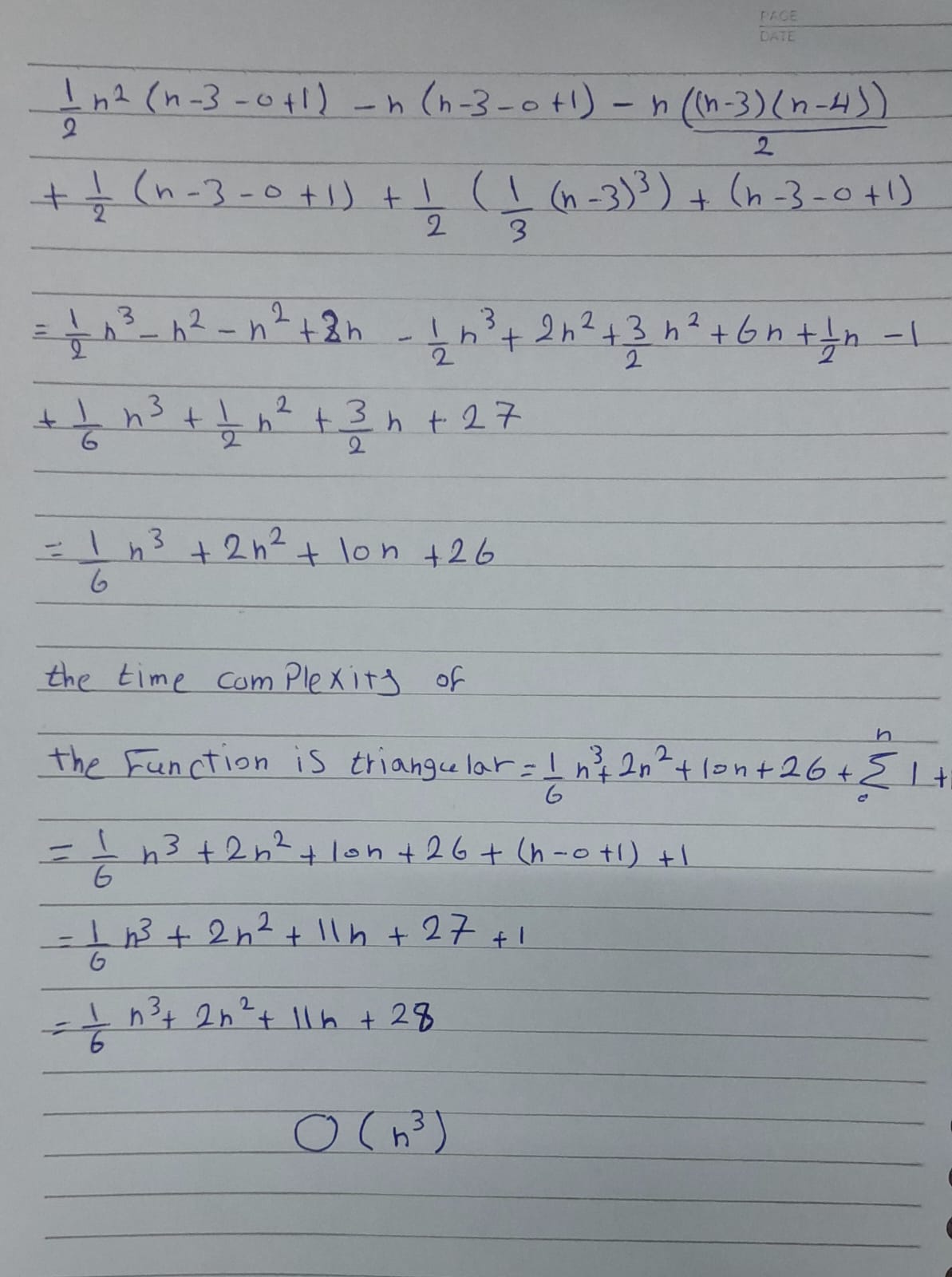
The time complexity of this algorithm is O(n^3), where n is the size of the array. This is because of the three nested loops that iterate through all possible combinations of array elements.

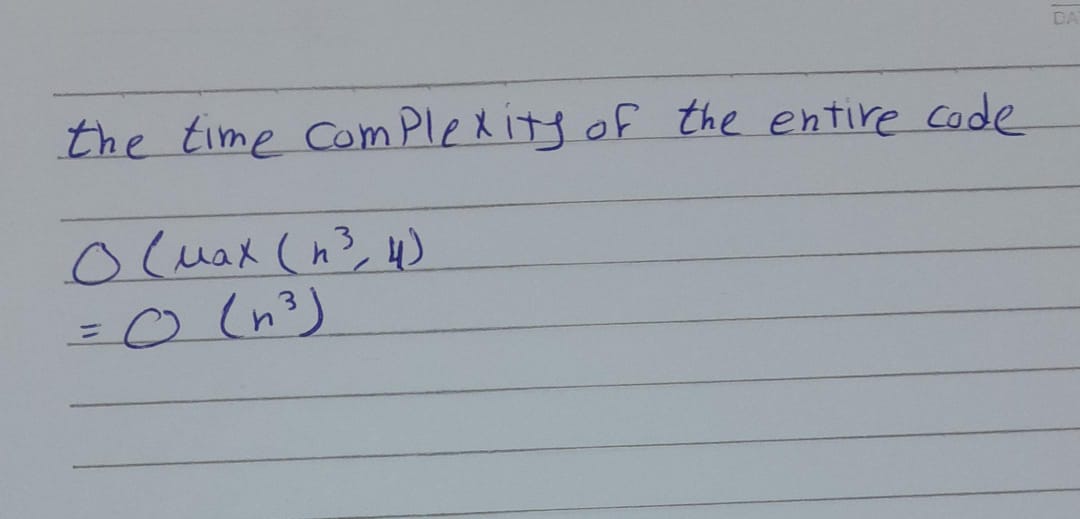
The space complexity is O(1), as the algorithm only uses a constant amount of extra space regardless of the size of the input array.

The time complexity:

The worst case:







The best case is O(n) because it is the lower bound, which occurs when there are a triangular at the first of the array.

Recursive Algorithm:

Code:

**function** sortArray**(**arr**,** size**)** **{**

**for** **(let** i **=** 0**;** i **<** size**;** i**++)** **{**

**for** **(let** j **=** 0**;** j **<** size **-** i **-** 1**;** j**++)** **{**

**if** **(**arr**[**j**]** **>** arr**[**j **+** 1**])** **{**

// Swap arr[j] and arr[j+1]

**let** temp **=** arr**[**j**];**

arr**[**j**]** **=** arr**[**j **+** 1**];**

arr**[**j **+** 1**]** **=** temp**;**

**}**

**}**

**}**

**}**

**function** getArraySize**(**nums**)** **{**

**let** count **=** 0**;**

**while** **(**nums**[**count**]** **!==** undefined**)** **{**

count**++;**

**}**

**return** count**;**

**}**

**function** checkTriangular**(**nums**,** start**,** size**)** **{**

**if** **(**start **+** 2 **>=** size**)** **return** 0**;**

**if** **(**nums**[**start**]** **+** nums**[**start **+** 1**]** **>** nums**[**start **+** 2**])**

**return** 1**;** // Triangular triplet found

**else** **return** checkTriangular**(**nums**,** start **+** 1**,** size**);** // Move to the next starting position

**}**

**function** isTriangular**(**nums**)** **{**

**let** size **=** getArraySize**(**nums**);**

sortArray**(**nums**,** size**);** // Sort the array in non-decreasing order

**return** checkTriangular**(**nums**,** 0**,** size**);**

**}**

**const** nums1 **=** **[**10**,** 50**,** 1**];**

**const** nums2 **=** **[**10**,** 2**,** 5**,** 1**,** 8**,** 20**];**

console**.**log**(**"Example 1:"**);**

console**.**log**(**"Output:"**,** isTriangular**(**nums1**));**

console**.**log**(**"\nExample 2:"**);**

console**.**log**(**"Output:"**,** isTriangular**(**nums2**));**

A black rectangular object with white lines

Description automatically generated

Pseudocode:

**function** sortArray**(**arr**,** n**)**

// arr is an array of size n

{

for i := 0 to size do

{

for j := 0 to n-i-1 do

{

if arr[j] > arr[j + 1] then

{

// Swap arr[j] and arr[j+1]

temp := arr[j]

arr[j] := arr[j + 1]

arr[j + 1] := temp

}

**}**

**}**

**}**

**function** getArraySize**(**nums**)**

{

count := 0

while nums[count] != undefined do

count := count +1

return count

}

**function** checkTriangular**(**nums**,** start**,** size**)**

{

if start + 2 >= size then

return 0

if nums[start] + nums[start + 1] > nums[start + 2] then

// Triangular triplet found

return 1

// Move to the next starting position

else return checkTriangular(nums, start + 1, size)

}

**function** isTriangular**(**nums**)**

{

// Sort the array in non-decreasing order

n := getArraySize(nums)

sortArray(nums)

return checkTriangular(nums, 0, n)

}

Analysis:

1. Sorting Function (sortArray):

- The sorting function implements the bubble sort algorithm to sort the array in non-decreasing order.

- Bubble sort has a time complexity of (O(n^2)) in the worst-case scenario, where (n) is the size of the input array.

2. Function to Get Array Size (getArraySize):

- The function iterates over the array to count the number of elements until it encounters an undefined value.

- This operation has a time complexity of (O(n)), where (n) is the size of the input array.

3. Function to Check for Triangular Triplets (checkTriangular):

- The function recursively checks for triangular triplets in the sorted array.

- In the worst-case scenario, when there are no triangular triplets, it traverses the entire array, resulting in a time complexity of (O(n)).

4. Overall Time Complexity (isTriangular):

- The isTriangular function first obtains the size of the input array using the getArraySize function, which has a time complexity of (O(n)).

- Then, it sorts the array using the sortArray function, which has a time complexity of (O(n^2)).

- Finally, it calls the checkTriangular function, which has a time complexity of (O(n)).

- Therefore, the overall time complexity of the isTriangular algorithm is dominated by the sorting function and is (O(n^2)).

In summary, the time complexity of the provided algorithm is (O(n^2)), where (n) is the size of the input array. This is because the sorting operation dominates the overall time complexity, while the other operations contribute to a linear time complexity.

The time complexity:

function sortArray :

n^2

O(n^2)

getArraySize :

-The loop iterates 'n' times, where 'n' is the number of elements in the array.

- for this time comlpexity is : O(n)

checkTriangular :

The function recursively checks for triangular triplets in the sorted array.

In the worst-case scenario, when there are no triangular triplets, it traverses the entire array, resulting in a time complexity of 𝑂(𝑛)

isTriangular :

It calls the getArraySize function, which has a time complexity of O(n)

It calls the 'sortArray' function, which has a time complexity of O(n^2)

It calls the 'checkTriangular' function, which has a time complexity of O(n)

O(max(g1(x),g2(x),g3(x)), so:

O(max(n, n^2,n)=O(n^2)

The best case is O(n), which occurs when the input array is already sorted in non-decreasing order before sorting.

The comparison:

|  |  |  |
| --- | --- | --- |
| The algorithm | The worst case | The best case |
| Non-recursive | O(n^3) | O(n) |
| Recursive | O(n2) | O(n) |

Complexity:

The first algorithm has a time complexity of O(n^3).

The second algorithm has a time complexity of O(n^2), making it more efficient in terms of time complexity.

Both have the same best case.

Approach:

Both algorithms follow different approaches, with the first directly checking combinations and the second sorting before checking for triangular triplets.

Efficiency:

The second algorithm is more efficient in terms of time complexity, especially for larger arrays, due to its O(n^2) complexity compared to the O(n^3) complexity of the first algorithm.