By Hand:

7	42	18	9	31	25	12	3	56	37	14	29	5	23

Average = (7+42+18+9+31+25+12+3+56+37) / 10

Average = 24.0

7	42	18	9	31	25	12	3	56	37	14	29	5	23

Average = (42+18+9+31+25+12+3+56+37+14) / 10

Average = 24.7

7	42	18	9	31	25	12	3	56	37	14	29	5	23

Average = (18+9+31+25+12+3+56+37+14+29) / 10

Average = 23.4

7	42	18	9	31	25	12	3	56	37	14	29	5	23

Average = (9+31+25+12+3+56+37+14+29+5) / 10

Average = 22.1

7	42	18	9	31	25	12	3	56	37	14	29	5	23

Average = (31+25+12+3+56+37+14+29+5+23) / 10

Average = 23.5

OVERALL HIGHEST AVERAGE = 24.7

Approach:

The goal of this program is to take an array of numbers of any length and then iterate through the array, creating as many sub-arrays of a specified length as possible. From there, the average of each sub-array will be found. The program will then determine the highest of those averages and return that number back to the user. To accomplish the actual task of creating the sub-arrays the program will start at the first index of the main array and then go as many indices as is specified by the user for the length of the sub-array and then find the average of those numbers and store it in a separate array. At the beginning of the next iteration, the starting point (and thus the ending point as well) will move one index to the left and repeat the process, so on and so forth until it is no longer possible to create another sub-array of the specified length.

Pseudocode Attempt #1:

```
power_algorithm(array, subarray_length)
        FOR i <- (length(array), i++)
                IF i+subarray_length <= length(array)
                        SET subarray <- array[i:i+subarray_length]
                        SET average <-sum(subarray) / subarray_length
                        averages.append(average)

determine_max_average(averages)
        SET highest_average <- averages[0]
        FOR average IN averages
                IF average > highest_average
                        SET highest_average <- average
        RETURN highest_average
```

Microsoft Copilot Pseudocode:

Me:

Create a pseudocode solution to the following: The goal of this program is to take an array of numbers of any length and then iterate through the array, creating as many sub-arrays of a specified length as possible. From there, the average of each sub-array will be found. The program will then determine the highest of those averages and return that number back to the user. To accomplish the actual task of creating the sub-arrays the program will start at the first index of the main array and then go as many indices as is specified by the user for the length of the sub-array and then find the average of those numbers and store it in a separate array. At the beginning of the next iteration, the starting point (and thus the ending point as well) will move one index to the left and repeat the process, so on and so forth until it is no longer possible to create another sub-array of the specified length.

Copilot:

Certainly! Let's break down the problem and create a pseudocode solution step by step. I'll provide a high-level representation of the algorithm:

1. **Input:**
   - An array of numbers (main_array).
   - A specified sub-array length (sub_array_length).
2. **Initialize:**
   - Create an empty array to store the averages (averages_array).
   - Set max_average to a very small value (e.g., negative infinity).
3. **Iterate through the main array:**
   - Start at index 0.
   - While the current index + sub_array_length is within the bounds of the main array:
     - Calculate the average of the sub-array from the current index to current_index + sub_array_length - 1.
     - Append the average to averages_array.
     - If the calculated average is greater than max_average, update max_average.
4. **Output:**
   - Return max_average.

Here's the pseudocode representation:

```
function find_max_average(main_array, sub_array_length):
    // Initialize variables
    averages_array = []  // Stores averages of sub-arrays
```

```
max_average = -∞

// Iterate through the main array
for current_index from 0 to length(main_array) - sub_array_length:
    // Calculate average of sub-array
    sum = 0
    for i from current_index to current_index + sub_array_length - 1:
        sum += main_array[i]
    average = sum / sub_array_length

    // Update max_average if necessary
    if average > max_average:
        max_average = average

    // Store average in averages_array (optional)
    averages_array.append(average)

// Return the maximum average
return max_average
```

Remember to replace comments with actual code when implementing this in a programming language. 😊[12]

Compare and Contrast:

The two solutions have very different approaches for solving the problem, which they both solve. My solution prioritizes clarity and readability by separating the tasks into two distinct functions while the Copilot solution emphasizes efficiency, and performance by doing it all in one function and updating the highest average on the fly rather than storing the averages in a new array and finding the highest average later. This does make the second solution more confusing as it all occurs within one function, so you sacrifice understandability/readability for increased efficiency. In other words, I could make mine more efficient based on what Copilot gave me and Copilot's solution could be more readable and easier to understand. By combining the on-the fly average update from Copilot with the easy readability of my solution, we could get the best solution possible in terms of efficiency (as the averages array and second function would not be needed) and readability (non of the confusing multi-level iterating done by the Copilot solution). I also believe that I can make my solution a bit more simple if I use a while loop. In the end, they both accomplish the task I performed in the first part of this assignment.

Final Pseudocode (Attempt 2):

```
find_highest_subarray_average(array, subarray_length)
    SET highest_average <- 0
    SET starting_index <- 0
    WHILE  starting_index + (subarray_length – 1) < length(array)
        SET subarray <- array[starting_index : starting_index + subarray_length]
        SET average <- sum(subarray) / subarray_length
        IF average > highest_average
            SET highest_average <- average
        SET starting_index <- starting_index + 1
    RETURN highest_average

main()
    SET numbers <- [1, 12, 5, 6, 50, 3]
    SET subarray_length <- 4
    SET result <- find_max_average(numbers, subarray_length)
     PUT result

CALL main()
```

Program Trace:

Line	numbers	subarray_length	result	highest_average	starting_index	subarray	average
13	[41, 45, 47, 32, 49, 40, 32]	/	/	/	/	/	/
14	[41, 45, 47, 32, 49, 40, 32]	4	/	/	/	/	/
2	[41, 45, 47, 32, 49, 40, 32]	4	/	0	/	/	/
3	[41, 45, 47, 32, 49, 40, 32]	4	/	0	0	/	/
5	[41, 45, 47, 32, 49, 40, 32]	4	/	0	0	[41, 45, 47, 32]	/
6	[41, 45, 47, 32, 49, 40, 32]	4	/	0	0	[41, 45, 47, 32]	41.25
8	[41, 45, 47, 32, 49, 40, 32]	4	/	41.25	0	[41, 45, 47, 32]	41.25
9	[41, 45, 47, 32, 49, 40, 32]	4	/	41.25	1	[41, 45, 47, 32]	41.25
5	[41, 45, 47, 32, 49, 40, 32]	4	/	41.25	1	[45, 47, 32, 49]	41.25
6	[41, 45, 47, 32, 49, 40, 32]	4	/	41.25	1	[45, 47, 32, 49]	43.25
8	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	1	[45, 47, 32, 49]	43.25
9	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	2	[45, 47, 32, 49]	43.25
5	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	2	[47, 32, 49, 40]	43.25
6	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	2	[47, 32, 49, 40]	42
9	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	3	[47, 32, 49, 40]	42
5	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	3	[32, 49, 40, 32]	42
6	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	3	[32, 49, 40, 32]	38.25
9	[41, 45, 47, 32, 49, 40, 32]	4	/	43.25	4	[32, 49, 40, 32]	38.25
15	[41, 45, 47, 32, 49, 40, 32]	4	43.25	43.25	4	[32, 49, 40, 32]	38.25

Algorithmic Efficiency:

For this portion I will focus only on the function that contains the actual algorithm rather than trying to include the main function in this analysis. The first 2 lines of code in the algorithm are both O(1) as we are setting variables to values. From there, everything within the while loop is also O(1) because we are setting values to variables and doing comparisons based on those values, all of which takes constant time. However, the number of iterations the while loop does is entirely dependent on the size of the array and the desired size of each of the sub-arrays analyzed by the algorithm, which means it has O(n) efficiency. This O(n) efficiency dominates and overrides all of the O(1) portions of the algorithm as those are run numerous times depending on the size of the input for the O(n) while loop. This means the overall efficiency of the algorithm is O(n).