**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. [😊1](https://www.geeksforgeeks.org/how-to-write-a-pseudo-code/)[2](https://stackoverflow.com/questions/39883025/how-to-check-whether-a-number-is-prime-or-not-algorithm-using-brute-force)

**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:**

A screenshot of a table

Description automatically generated

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).