The maximum contiguous subsequence sum (mcss) of an array is the largest sum you can acquire by adding up consecutive elements of an array. For example, consider the following array:

\[5, -8, 7, 7, -1, 9, -6, -4, 5, -7\]

The mcss of this array is 22, corresponding to the subsequence \([7, 7, -1, 9]\). Note that if the array consists of non-negative numbers then the subsequence is simply the entire array. We say that the mcss of an array consisting of all negative numbers is zero.

For this lab, you won’t be writing code to solve this problem. Instead, you’ll be analyzing the time complexity of various solutions to this problem. Download the source file in question (linked off the course schedule):

\[
\ldots/fileMaxContiguousSubsequenceSum.java
\]

The three functions—`compute1(arr)`, `compute2(arr)`, and `compute3(arr)`—all return the mcss of the given array.

## 1 Wall-clock Time

First, we will use the unix `time` utility to test how long each function takes to run. For example, we can compile and run the program at a terminal window as follows:

\[
\$> javac MaxContiguousSubsequenceSum.java\\
\$> time java MaxContiguousSubsequenceSum\\
\]

Generating a random array of size 10... complete!

\[
arr = [-3, 7, 6, -2, 1, -2, 3, -8, 3, 8]\\
compute1(arr) = 16\\
compute2(arr) = 16\\
compute3(arr) = 16
\]

real 0m0.171s
user 0m0.089s
sys 0m0.045s

`time` runs the program (and arguments) passed to it and reports the time taken for that program to execute—the total time `real`, the amount of that time spent in user code `user`, and the time spent in system code `sys`. If you are using a Windows machine, you will need to use a MathLAN computer for this lab as Windows does not have an equivalent `time` utility.

Now, modify the code and re-compile it so that the program runs only one of the functions, e.g., `compute1`. Run the program at least five times and record the average total time that you obtain for the function at a particular array size (the size is controlled by the `size` and `range` local variables in `main`).

Repeat this process for each function and the following array sizes:
• 10, 50, 100, 500, 1000, 5000, 10000, 100000,

record your results in the space provided, and graph the data you collect. The $x$-axis of your graph should be the size of the array and the $y$-axis should be the time taken. Please include your data and your graphs on the other side of this page. Note that for some combinations of function and larger array sizes, the code might take too long to execute! In these cases, you can let the code run for a few minutes to see if it will complete.

*(Please give your collected data and graphs for part 1 below.)*
2 Counting Operations

As discussed in the reading, while wall-clock is what ultimately matters when we talk about program performance, there are significant limitations to timing our programs over many inputs to assess its performs. One of those limitations is that wall-clock is highly sensitive to the particulars of the machine we run our programs on. We can avoid that limitation by instead counting the critical operations that a function performs. While the time a program takes to execute may vary widely on the state of computer, the program will perform the same number of critical operations no matter where the program executes\(^1\).

For mcss, we’ll consider the number of array accesses that each function performs. Let’s define an array access as any case where the function reads an array value (e.g., ```int x = arr[j]``` ) or writes an array value (e.g., ```arr[j] = 5``` ). Assuming that the array accesses dominates the runtime of the functions, then counting array accesses should be tantamount to measuring the time each function takes.

Augment the three functions so that rather than returning the mcss of the given array, they report the number of array accesses each function makes while computing the mcss. Use your augmented program to repeat the experiment from the first part of this lab: collect the number of array accesses required for each function for the following array sizes: 10, 50, 100, 500, 1000, 5000, 10000, 100000. For each function, graph the data you collected. The x-axis of your graph should be the size of the array and the y-axis should be the number of array accesses. Please include your data and your graphs on the other side of this page.

Finally, compare the graphs from both parts of the lab. How accurate is the operation counting method of measuring time complexity compared to the wall-clock method for understanding how the time complexity scales with the size of the input?

---

\(^1\)At least the deterministic programs that we write in this course. With more complex programs, other factors become relevant which can make this method of analyzing program complexity less accurate.
(Please give your collected data and graphs for part 2 below.)