Exercise 1)

What is the Big-Oh of the following computation?

int sum = 0;

for (int counter = n; counter > 0; counter = counter - 2)

sum = sum + counter;

The for loop starts with counter = n and decrements by 2 in each iteration. This means the number of iterations is approximately n/2, which represents half of n. The constant factor 1/2 is irrelevant in Big-O analysis because Big-O focuses on the growth rate as n increases.

Therefore, n/2 simplifies to O(n).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Suppose your implementation of a particular algorithm appears in Java as follows:

for (int pass = 1; pass <= n; pass++)

{

for(int index = 0; index < n; index++)

{

for(int count = 1; count < 10; count++)

{

} //end for

} // end for

} //end for

The algorithm involves an array of "n" items. The previous code shows only the repetition in the algorithm, but it does not show the computations that occur within the loops.Those computations, however, are independent of "n." What is the order of the algorithm?

The outer for loop runs n times.

For each iteration of the outer loop, the middle loop also runs n times.

Within the middle loop, the inner loop always runs a constant number of times (9).

This results in a total of n \* n \* 9 = 9n^2 iterations.

In Big-O analysis, we ignore constant factors like 9, so the complexity simplifies to O(n²).

Exercise 3)

Consider two programs, A and B. Program A requires 1000 x n^2 operations and Program B requires 2^n operaitons. For which values of n will Program A execute faster than Program B?

Based on the explanation, Program A = 1000n^2 and Program B = 2^n.

To determine when Program A is faster than Program B, we need to compare their formulas at different values of n.

A table with numbers and letters

AI-generated content may be incorrect.

As noted in this table, at first Program B performs better than Program A, up until a point in between n = 15 and n = 20.

From this point forward, Program B continues to exponentially grow so quickly that it more than duplicated itself before even reaching 100 iterations.

As shown by this graph, Program A is faster than Program B when n > 18 (non-inclusive of n = 18).

Exercise 4)

Consider an array of length "n" containing unique integers in random order and in the range 1 to n + 1. For example an array of length 5 would contain 5 unique integers selected randomly from the integers 1 through 6. Thus the array might contain 3 6 5 1 4. Of the integers 1 through 6, notice that 2 was not selected and is not in the array. Write Java code that finds the integer that does not appear in such an array. Explain the Big-Oh in your code.

We are tasked with finding the missing integer from an array of size n containing unique integers in the range 1 to n+1

For example, if the array is [3, 6, 5, 1, 4], the missing number is 2.

This one was tricky; after some googling I found a clean solution in which we determine the sum of all present numbers and subtract that from the expected total sum to find the missing value - which makes perfect sense and will be how I would solve this type of problem in the future.

However, it really felt like something I wouldn't be able to come up on my own, so for the purpose of this assignment and my own practice I gave it another try with a different approach.

Since it is guaranteed that only 1 number is missing, then that means all other numbers are present and these will be either preceding or proceeding the missing number.

In this implementation the most important is that I’m using two maps/caches to keep track of found numbers and missing numbers throughout a single iteration.

As we iterate, we use the current number to "discover" the other numbers around it that are expected to the present. These are num + 1 and num - 1.

This also required binding into scope (1 - n + 1 means that 0 cannot be included, and n + 1 + 1 should ALSO not be included).

As we check each num, we remove "possible" missing numbers and mark them as found.

If num + 1 and num - 1 have not been found, they can be considered possibly missing respectively until found otherwise.

At the end, only the last missing number (the one that is possibly missing and was not found) will be left.

In terms of complexity: This is a single loop, with no nested looping operations at each iteration, so its time complexity is O(n).

A computer screen shot of a program code

AI-generated content may be incorrect.

Here is the snippet of the code implementation and I’ll also include the GitHub URL in the comments of the submission.

<https://github.com/GMtnezS/Sharable_Code_Snippets/blob/main/School_Snippets/CSC400/wk3/cta/Main.java>