Module 3 Critical Thinking

Critical Thinking Assignment (70 Points)

Exercises 10, 12, 16, and 24a

In Chapter 4 of Data Structures and Abstractions with Java (Carrano & Henry), complete the following:

10. What is the Big Oh of the following computation?

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

12. Suppose that 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 the only repetition in the algorithm, but it does not show the computations that occur within the loops. These computations, however, are independent of n. What is the order of the algorithm?

16. Consider two programs, A and B. Program A requires $1000 \times n^2$ operations and Program B requires 2^n operations. For which values of n will Program A execute faster than Program B?
NOTE: There is currently (Fall 2023) an error in the preferred answer key. Include something like: "If program A required $1000 + n^2$ operations, then Program A would execute faster at n = …"

24. 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 the 2 was not chosen and is not in the array.

Write Java code that finds the integer that does not appear in such an array. Your solution should use

a. $O(n^2)$ operations
b. $O(n)$ operations

For each exercise, show your work and all of the steps taken to determine the Big-Oh for each problem.

Partial credit cannot be awarded without showing work. Submit all work in a Word document.

Module 4 Critical Thinking

Critical Thinking Assignment (70 Points)

Program 2 (Java Postfix Converter)

ICreate an Java Postfix converter using the algorithm provided in Segment 5.16 on page 167 (Chapter 5 of Carrano & Henry, 2019, p. 167).

Ensure that your program has the required class and a test class. Submit screenshots of your program's execution and output compiled into a single document. Also attach all appropriate source code in a zip file:

5.11 Recall that in a postfix expression, a binary operator follows its two operands. Here are a few examples of infix expressions and their corresponding postfix forms:

Infix	Postfix
a + *b*	*a b* +
(*a* + *b*) * *c*	*a b* + *c* *
a + *b* * *c*	*a b c* * +

Note: Infix-to-postfix conversion

To convert an infix expression to postfix form, you take the following actions, according to the symbols you encounter, as you process the infix expression from left to right:

- Operand — Append each operand to the end of the output expression.
- Operator ^ — Push ^ onto the stack.
- Operator +, -, *, or / — Pop operators from the stack, appending them to the output expression, until the stack is empty or its top entry has a lower precedence than the new operator. Then push the new operator onto the stack.
- Open parenthesis — Push (onto the stack.
- Operator ^ — Pop operators from the stack, appending them to the output expression until an open parenthesis is popped. Discard both parentheses.

5.16 **The infix-to-postfix algorithm.** The following algorithm encompasses the previous observations about the conversion process. For simplicity, all operands in our expression are single-letter variables.

```
Algorithm convertToPostfix(infix)
{
    // Converts an infix expression to an equivalent postfix expression.
    operatorStack = a new empty stack
    postfix = a new empty string
    while (infix has characters left to parse)
    {
        nextCharacter = next nonblank character of infix
        switch (nextCharacter)
        {
            case variable:
                Append nextCharacter to postfix
                break
            case '^' :
                operatorStack.push(nextCharacter)
                break
            case '+' : case '-' : case '*' : case '/' :
                while (!operatorStack.isEmpty() and
                        precedence of nextCharacter <=
                          precedence of operatorStack.peek())
                {
                    Append operatorStack.peek() to postfix
                    operatorStack.pop()
                }
                operatorStack.push(nextCharacter)
                break
            case '( ' :
                operatorStack.push(nextCharacter)
                break
            case ')' : // stack is not empty if infix expression is valid
                topOperator = operatorStack.pop()
                while (topOperator != '(')
                {
                    Append topOperator to postfix
                    topOperator = operatorStack.pop()
                }
                break
            default: break
        }
        while (!operatorStack.isEmpty())
        {
            topOperator = operatorStack.pop()
            Append topOperator to postfix
        }
    return postfix
    }
}
```

Figure 5-9 traces this algorithm for the infix expression `a / b * (c + (d - e))`. The resulting postfix expression is `a b / c d e - + *`.

FIGURE 5-9 The steps in converting the infix expression `a / b * (c + (d - e))` to postfix form

Next Character from Infix Expression	Postfix Form	Operator Stack (bottom to top)
a	*a*	
/	*a*	/
b	*a b*	/
*	*a b* /	
	a b /	*
(	*a b* /	* (
c	*a b c*	* (
+	*a b c*	* (+
(	*a b c*	* (+ (
d	*a b c d*	* (+ (
–	*a b c d*	* (+ (–
e	*a b c d e*	* (+ (–
)	*a b c d e* –	* (+ (
	a b c d e –	* (+
)	*a b c d e* – +	* (
	a b c d e – +	*
	a b c d e – + *	

Module 5 Critical Thinking

Critical Thinking Assignment (70 Points)

Program 3 (Algorithm Analysis / Big-Oh Notation)

Implement the algorithm outlined in Exercise # 4 (Chapter 9 of Carrano & Henry, p. 288) in Java.

Analyze your algorithm in Big-Oh notation and provide the appropriate analysis, ensuring that your program has the required class and a test class.

In addition, compile and submit Exercise 5, the Big-Oh evaluations, and screenshots of your program's execution and output in a single document.

4. The factorial of a positive integer n—which we denote as $n!$—is the product of n and the factorial of $n - 1$. The factorial of 0 is 1. Write two different recursive methods that each return the factorial of n.

5. Write a recursive method that displays a portion of a given array backward. Consider the last entry in the portion first.

Module 6 Critical Thinking

Critical Thinking Assignment (70 Points)

Program 4 (Exercise #11)

Implement the algorithm outlined in Exercise # 11 (Chapter 15 of Carrano & Henry, p. 458) in Java.

Ensure that your program has the required class and a test class. Compile and submit Exercise #11, the screenshots of your program's execution and output in a single document. Also attach all appropriate source code in a zip file.

11. Devise an algorithm that detects whether a given array is sorted into ascending order. Write a Java method that implements your algorithm. You can use your method to test whether a sort method has executed correctly.