COMP 203 DATA STRUCTURES AND ALGORITHMS

QUIZ 1 SOLUTION

1.
```
// Java program to create Special triangle.
import java.util.*;
import java.lang.*;

public class ConstructTriangle
{
	// Function to generate Special Triangle.
	public static void printTriangle(int[] A)
	{
		// Base case
		if (A.length < 1)
			return;

		// Creating new array which contains the
		// Sum of consecutive elements in
		// the array passes as parameter.
		int[] temp = new int[A.length - 1];
		for (int i = 0; i < A.length - 1; i++)
		{
			int x = A[i] + A[i + 1];
			temp[i] = x;
		}

		// Make a recursive call and pass
		// the newly created array
		printTriangle(temp);

		// Print current array in the end so
		// that smaller arrays are printed first
		System.out.println(Arrays.toString(A));
	}

	// Driver function
	public static void main(String[] args)
```

```
    {
        int[] A = { 1, 2, 3, 4, 5 };
        printTriangle(A);
    }
}
```

2.a.b.c

```
public class DoublyLinkedList<E> {
    private class Node {
        E value;
        Node prev;
        Node next;

        public Node(E value) {
            this.value = value;
            this.prev = null;
            this.next = null;
        }
    }

    private Node head;
    private Node tail;

    public void InsertBetween(E first, E second, E addedValue) {
        Node newNode = new Node(addedValue);

        Node current = head;

        // Traverse the list to find the node with the value 'first'
        while (current != null && !current.value.equals(first)) {
            current = current.next;
        }

        // If 'first' is not found, return or handle as needed
        if (current == null) {
            // Handle error or return
            return;
        }

        // Link the new node to the next node in the list
        newNode.next = current.next;
        newNode.prev = current;
        if (current.next != null) {
            current.next.prev = newNode;
        }
        current.next = newNode;

        // Move the 'current' pointer to the newly added node
        current = newNode;

        // Traverse the list to find the node with the value
'second'
        while (current != null && !current.value.equals(second)) {
            current = current.next;
```

```
        }

        // If 'second' is not found, return or handle as needed
        if (current == null) {
            // Handle error or return
            return;
        }

        // Update the links to insert the new node between 'first'
and 'second'
        newNode.next = current;
        newNode.prev = current.prev;
        current.prev = newNode;
        if (newNode.prev != null) {
            newNode.prev.next = newNode;
        } else {
            // If newNode is inserted at the beginning, update the
head pointer
            head = newNode;
        }
    }

}
```

d. O(n) where is n is the size of the DLL because it visits every node.

3. a.
```
public class ArraySum {
    public static int sumArrayElements(int[] array, int left, int
right) {
        if (left == right) {
            return array[left];
        } else {
            int mid = (left + right) / 2;
            int sumLeft = sumArrayElements(array, left, mid);
            int sumRight = sumArrayElements(array, mid + 1, right);
            return sumLeft + sumRight;
        }
    }

    public static void main(String[] args) {
        int[] myArray = {1, 2, 3, 4, 5};
        int result = sumArrayElements(myArray, 0, myArray.length -
1);
        System.out.println("Sum of array elements: " + result);
    }
}
```

b.c. The time complexity of this binary recursion function is O(n), where "n" is the number of elements in the array. Even though it uses a divide-and-conquer approach, it doesn't reduce the overall time complexity compared to the iterative approach. This is because each recursive call splits the array into two halves, but it still visits and processes every element in the array.
The function makes O(n) recursive calls, and in each call, it performs a constant amount of work to calculate the sum. Therefore, the overall time complexity is O(n), which is the same as the iterative (given function) approach.