

1. Stacks and Queues

Part (a): Fill in the blanks in the code below so that the tests pass.

Part (b): Show the contents of each intermediate stack or queue (after processing each character) for the following two inputs:

```
"[()] {}{[()]}"
```

```
"[()]"
```

For the unbalanced input (the second one), which character is on top of the stack or at the end of the queue just before your method returns false?

Part (c): Explain why your code is correct (why it returns the right result on any string, not just the ones you tested.)

Extra credit: For unbalanced strings, make your method print out the index within the string (where the first character is 0) of the opening character that causes the string not to be balanced. For example, for "[()]", it should print out 1, because when the ']' is encountered, it's '(' that is unmatched.

```
import edu.princeton.cs.algs4.Stack;
import edu.princeton.cs.algs4.Queue;
public class Problem1 {

    // Return true if and only if all parentheses (), brackets [], and braces {} in s are balanced
    // Ignore any other characters in s
    public static boolean checkBalanced(String s) {
        // First, create either a stack or queue of characters.
        // Code goes here

        /* Then: For each character, either:
           add it to the stack or queue, or
           check something about the stack or queue.
           You may (or may not) use early return statements in this part of the code.
        */
        // Code goes here.

        /* Once all the characters are processed:
           return true if (the proper conditions are true about the stack or queue).
        */
        // Code (return statement) goes here.
    }

    // Helper method
    public static boolean match(char opener, char closer) {
        return(opener == '(' && closer == ')'
            || opener == '{' && closer == '}'
            || opener == '[' && closer == ']');
    }

    public static void test(String s, boolean expected) {
        boolean result = checkBalanced(s);
        String message = (result == expected ? "Test succeeded! " : "Test failed: ") + s;
        System.out.println(message);
    }

    public static void main(String[] args) {
        String balanced = "[()] {}{[()]}" ;
        String unbalanced = "[()]" ;
        test("", true);
        test("[()]", true);
        test(balanced, true);
        test(unbalanced, false);
        test("(", false);
    }
}
```

2. Analysis of Algorithms

Consider the following class that implements a sorting algorithm:

```
public class Problem2 {

    private static class Two {
        int x;
        int y;

        public Two(int xx, int yy) {
            x = xx;
            y = yy;
        }
    }

    public static void sort(Comparable[] a) {
        sort(a, 0, a.length - 1);
        assert isSorted(a);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        Stack<Two> todo = new Stack();
        todo.push(new Two(lo, hi));
        while(!todo.isEmpty()) {

            Two t = todo.pop();

            if (t.y > t.x) {
                int j = split(a, t.x, t.y);

                todo.push(new Two(t.x, j - 1));
                todo.push(new Two(j+1, t.y));
            }
        }
    }

    private static int split(Comparable[] a, int lo, int hi) {

        int i = lo;
        int j = hi + 1;
        Comparable v = a[lo];

        while (less(a[++i], v)) {
            if (i == hi) { exch(a, lo, hi); return hi; }
        }

        while (less(v, a[--j])) {
            if (j == lo + 1) return lo;
        }

        while (i < j) {
            exch(a, i, j);
            while (less(a[++i], v)) ;
            while (less(v, a[--j])) ;
        }

        exch(a, lo, j);
        return j;
    }
}
```

The definitions of `less()` and `exch()` are as in the book.

Suppose `a.length == N`.

Part(a): In terms of N , what is the running time of this algorithm? Explain why. **Use tilde notation.**

Part (b): In terms of N , how much memory does this algorithm use? (Not counting the size of a itself.) Explain why. **Use tilde notation.**

3. Mergesort

Modify the implementation of top-down mergesort on p. 273 of the book to use cutoff to insertion sort, just as you did for QuickSort on lab 2. (See p. 296 if you need to refresh your memory on what "cutoff to insertion sort" means.) Then give a trace of what your modified version of mergesort does on the array "MERGESORTEXAMPLE", with cutoff 4. Use the style of the trace on p. 273 of the book.

ALGORITHM 2.4 Top-down mergesort

```
public class Merge
{
    private static Comparable[] aux;    // auxiliary array for merges
    public static void sort(Comparable[] a)
    {
        aux = new Comparable[a.length]; // Allocate space just once.
        sort(a, 0, a.length - 1);
    }
    private static void sort(Comparable[] a, int lo, int hi)
    {
        // Sort a[lo..hi].
        if (hi <= lo) return;
        int mid = lo + (hi - lo)/2;
        sort(a, lo, mid);    // Sort left half.
        sort(a, mid+1, hi);  // Sort right half.
        merge(a, lo, mid, hi); // Merge results (code on page 271).
    }
}
```

To sort a subarray $a[lo..hi]$ we divide it into two parts: $a[lo..mid]$ and $a[mid+1..hi]$, sort them independently (via recursive calls), and merge the resulting ordered subarrays to produce the result.

	lo	hi	a[]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15		
merge(a, 0, 0, 1)																		
merge(a, 2, 2, 3)																		
merge(a, 0, 1, 3)																		
merge(a, 4, 4, 5)																		
merge(a, 6, 6, 7)																		
merge(a, 4, 5, 7)																		
merge(a, 0, 3, 7)																		
merge(a, 8, 8, 9)																		
merge(a, 10, 10, 11)																		
merge(a, 8, 9, 11)																		
merge(a, 12, 12, 13)																		
merge(a, 14, 14, 15)																		
merge(a, 12, 13, 15)																		
merge(a, 8, 11, 15)																		
merge(a, 0, 7, 15)																		

Trace of merge results for top-down mergesort

4. Queues and Linked Lists

Part (a): Write a Queue implementation that uses a circular linked list. Fill in the class below. Note that there is only one Node instance variable. You should not add more. Be sure that the implementations of isEmpty() and size() return the right results (you have to write code to make sure N is correct after each operation). Part (b): For the following input:

```
to be or not to - be - - that - - - to
```

show your program's output (from running the code, noticing what the test client in main() does) **and** show each enqueue() and dequeue() operation that is called, in the correct order, **and** draw a picture of the queue before each operation. Follow the example on p. 152 to draw the pictures of the queue at each step. Note that instead of null links in the "next" instance variable, instead your queue will have a pointer back to the front of the queue.

```
public class Queue {
    private Node last;
    private int N;

    private class Node {
        Item item;
        Node next;
    }

    public boolean isEmpty() { return last == null; }
    public int size() { return N; }

    public void enqueue(Item item) {
        // create a new node

        // Two cases: enqueue into empty queue, and non-empty queue

        Node newLast = new Node();
        newLast.item = item;
        if (isEmpty()) {
            // fill in
        } else {
            // fill in
        }
    }

    public Item dequeue() throws Exception {
        if (isEmpty()) {
            throw new Exception("dequeue of empty queue");
        }
        // Two more cases: dequeue from queue of size 1, or
        // dequeue from queue of size n >= 1
        if (N == 1) {
            // Fill in code
        } else {
            // Fill in code
        }
    }
}

// test client, after p. 150 of the book
public static void main(String[] args) {
    Queue queue = new Queue();
    while (!StdIn.isEmpty()) {
        String item = StdIn.readString();
        if (!item.equals("-"))
            queue.enqueue(item);
        else if (!queue.isEmpty()) {
            try {
                StdOut.print(queue.dequeue());
            } catch (Exception e) {
                StdOut.print(e.toString());
                break;
            }
        }
    }
    StdOut.println("(" + queue.size() + " left on queue)");
}
```