Section Exercises 6: Running Times and ADTs

Problem 1: Big-O Notation

a) (Chapter 7, Review Question 12) What is the computational complexity of the following function:

```
int Mystery1 (int n)
{
       int i, j, sum;

       sum = 0;
       for (i = 0;i < n; i++) {
              for (j = 0; j < i; j++){
                     sum += i * j;
              }
       }
       return (sum);
}
```

b) (Chapter 7, Review Question 12) What is the computational complexity of this function:

```
int Mystery2 (int n)
{
       int i, j, sum;

       sum = 0;
       for (i = 0;i < 10; i++) {
              for (j = 0; j < i; j++){
                     sum += j * n;
              }
       }
       return (sum);
}
```

Problem 2: Searching and Sorting

You have a data set of size *n* that's currently unsorted. You're trying to decide whether it will be worthwhile to sort the data before performing repeated searches, to take advantage of binary search. (Look at pp. 182-183 of the text for a refresher on binary search.) If you use SelectionSort to sort the data, how many binary searches would you need to perform to "buy back" the cost that went into sorting your data if:

a) n is equal to 16 (2^4)
b) n is equal to 1024 (2^{10})

What if you use MergeSort instead of SelectionSort?

Problem 3: The Philosophy of ADTs

a) What is meant by "breaking the wall" of an ADT, and why is it a bad thing?

b) What do we gain by ensuring that the client cannot depend on the details of an ADT's implementation?

Problem 4: `StackADT`

Imagine that the `StackDepth` function had been mistakenly left out of the interface for the `stackADT`. *Without* breaking the wall, write a client-side version of a stack depth function that will return the numbers of elements currently on a stack without losing the stack's contents. (Hint: Think recursively.)

Problem 5: Advanced Stack ADT client (16 points)
This problem was taken from a previous midterm exam

Write the **`IsBalanced`** function that uses a character stack to determine whether the bracketing operations (parenthesis, square brackets, and curly braces) in a string are properly matched. As an example of proper matching, consider the string:

```
{ s = 1 * (array[1] + 3); x = (1 * (2 + 9)); }
```

If you go through the string, you can see that the bracketing operators are correctly nested, with each open parenthesis matched by a close parenthesis, each open bracket matched by a close bracket, and so on. In contrast, the following strings are all unbalanced for the reasons indicated:

`(([z])`	Missing a close parenthesis
`AB}{$`	The closing brace comes before open brace
`{[}]`	The bracketing operators are improperly nested

All characters other than the bracketing operators are ignored in determining whether a string is balanced. The empty string, by definition, is balanced.

You must use a stackADT. You have access to the standard stack implemented in Chapter 8, as described in the interface on pages 336-338, and **`stackElementT`** has been typedef-d to **`char`**. You do not have to be concerned about freeing memory when writing this function. Assume the following helpers are given to assist with the bracketing operators:

static bool IsOpener(char ch); // returns TRUE if ch is ({ or [, else FALSE
static bool IsCloser(char ch); // returns TRUE if ch is) } or], else FALSE
static char MatchingChar(char ch); // returns matching char, i.e. returns { for },] for [, and so on. Returns '\0' if ch isn't an opener or closer

bool IsBalanced(string str)