Worksheet 8: Searching and Asymptotic Analysis

In preparation: Read Chapter 4 to learn more about big-oh notation.

<table>
<thead>
<tr>
<th>Linear search</th>
<th>O(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary search</td>
<td>O(log n)</td>
</tr>
<tr>
<td>countOccurrences</td>
<td>O</td>
</tr>
<tr>
<td>isPrime</td>
<td>O</td>
</tr>
<tr>
<td>printPrimes</td>
<td>O</td>
</tr>
<tr>
<td>matMult</td>
<td>O</td>
</tr>
<tr>
<td>SelectionSort</td>
<td>O</td>
</tr>
</tbody>
</table>

In this worksheet we will be concentrating on algorithms involving loops. In this case the question to ask is how many times the inner statements in a loop are being executed, and how this quantity changes if the input size is changed. As you look at each example, fill in the values in the table at left.

Our first function counts the number of occurrences of a given value in an array of data. If he execution time is proportional to the size of the input array, it is O(__)?

```c
int countOccurrences (double data [], int n, double testValue)
{
    int count = 0; int i;
    for (i = 0; i < data.length; i++)
    {
        if (data[i] == testValue)
            count++;
    }
    return count;
}
```

Our second function determines if an integer variable is prime. A prime, you will recall, is a value that is divisible only by itself and 1. Note that the input here is not an array, but a single integer value. What can you say about the worst-case execution time for this function in relation to the value of this variable?

```c
int isPrime (int n) {
    int i;
    for (i = 2; i * i <= n; i++)
    {
        if (0 == n % i) return 0;
    }
    return 1; /* 1 is true */
}
```

When one function calls another inside of the loop, the execution time of the called function must be counted in determining the time for the loop. Here there is a simple O(n) loop that calls isPrime. So the total execution time is in the worst case O(__)?

```c
void printPrimes (int n) {
    int i;
    for (i = 2; i < n; i++)
    {
        if (isPrime(i))
            printf("Value %d is prime\n");
    }
}
```

The classic matrix multiplication routine is a good example of nested loops. Again the question to ask is how many times the statements in the innermost loop are executed as a function of the data size (in this case,

```c
void matMult (int [][] a, int [][] b, int [][] c) {
    int n = n; // assume all same size
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            c[i][j] = 0;
            for (k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```
the number of rows and columns of the input arrays).

```c
void selectionSort(double * storage, int n) {
    int p, i, indexLargest;
    for (p = n - 1; p > 0; p--) {
        indexLargest = 0;
        for (i = 1; i <= p; i++) {
            if (storage[i] > storage[indexLargest])
                indexLargest = i;
        }
        if (indexLargest != position)
            swap(storage, indexLargest, position);
    }
}
```

Finally, consider selectionSort. Here the inner loop is more subtle. How is it related to the outer loop? Can you identify the pattern? How many steps will it take the first time the outer loop executes? The last? What does this tell you about the number of times the innermost if statement will be executed?

In the following table, assume that the … represents constant time operations. Describe in big-Oh notation the running time of each of the following loops.

<table>
<thead>
<tr>
<th>Loop Description</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for (i = 0; i &lt; n; i++)</code></td>
<td><code>O(n)</code></td>
</tr>
<tr>
<td><code>for (i = n; i &gt; 0; i--)</code></td>
<td><code>O(n)</code></td>
</tr>
<tr>
<td><code>for (i = 0; i * i &lt; n; i++)</code></td>
<td><code>O(n)</code></td>
</tr>
<tr>
<td><code>for (i = n; i &gt; 0; i = i / 2)</code></td>
<td><code>O(log n)</code></td>
</tr>
<tr>
<td><code>for (i = n; i &gt; 0; i = i &gt;&gt; 2)</code></td>
<td><code>O(log n)</code></td>
</tr>
<tr>
<td><code>for (i = 0; i &lt; n; i++)</code></td>
<td><code>O(n)</code></td>
</tr>
<tr>
<td><code>for (j = 0; j &lt; n; j++)</code></td>
<td><code>O(n)</code></td>
</tr>
<tr>
<td><code>for (j = 0; j &lt; i; j++)</code></td>
<td><code>O(i)</code></td>
</tr>
<tr>
<td><code>for (i = 0; i &lt; n; i++)</code></td>
<td><code>O(n)</code></td>
</tr>
<tr>
<td><code>for (j = 0; j &lt; 13; j++)</code></td>
<td><code>O(1)</code></td>
</tr>
<tr>
<td><code>for (i = n; n &gt; 0; i = i / 2)</code></td>
<td><code>O(log n)</code></td>
</tr>
<tr>
<td><code>for (j = n; j &gt; 0; j = j / 2)</code></td>
<td><code>O(log n)</code></td>
</tr>
<tr>
<td><code>for (i = 0; i &lt; n; i++)</code></td>
<td><code>O(n)</code></td>
</tr>
<tr>
<td><code>for (j = n; j &gt; 0; j = j / 2)</code></td>
<td><code>O(log n)</code></td>
</tr>
</tbody>
</table>