This document is a work in progress. Please email me if you find any typos, or if you have any comments to add.

When given an algorithm, the process of worst case complexity analysis is that of counting how many computations will be performed. Since the details of how computations are actually performed depends on the hardware and programming language used, we resort to theoretically counting the computations in relation to the input size.

In this document, you will find some of the basic instructions on how to compute the worst case running time complexity of an algorithm. I will be basing all the instructions on the Linear Search example from lecture and assignment 1.

```plaintext
function LinearSearch(A,v)
    i ← 1
    while v ≠ A[i] and i < length[A] + 1 do
        i ← i + 1
    end while
    if i < length[A] + 1 then return i
    else return NIL
end if
end function
```

Before you even start, you need to identify the worst case of the algorithm you are analyzing. In LinearSearch, that is not finding the element.

Then, you go over the algorithm, and find the cost of each line of code. Most operations are O(1), constant number, unless they are functions calls. Be careful of function calls hidden as plain text!

Finally, you go over the algorithm again, and count the number of repetitions of each line of code.

The final complexity of the algorithm is equal to the summation for all lines of the cost of the line multiplied by its repetition number.

Example:

<table>
<thead>
<tr>
<th>Code</th>
<th>Cost</th>
<th>#repetitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>i ← 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>while v ≠ A[i] and i &lt; length[A] + 1 do</td>
<td>5</td>
<td>n</td>
</tr>
<tr>
<td>i ← i + 1</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>end while</td>
<td></td>
<td>doesn’t count</td>
</tr>
<tr>
<td>if i &lt; length[A] + 1 then return i</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>else return NIL</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>end if</td>
<td></td>
<td>doesn’t count</td>
</tr>
</tbody>
</table>

From the table above, we can compute that \( T(n) = 7n + 2 \). Please note that the exact values of the coefficients and the constants are not important. The important thing is that \( T(n) \) is a linear function, which belongs to \( O(n) \)
Analyzing recursive algorithms There is a formal method to analyzing recursive algorithms, which will be discussed later in the course. The straw-man’s approach is to treat the algorithm the same way you would an algorithm with an iterative loop.

In recursion, you know that when the function invokes itself, a new execution frame is created with its own set of local variables and operations. To analyze the algorithm, first analyze the complexity of the function while assuming that the recursive calls have a cost and repetition value of 1. Then, count how many execution frames could be created in the worst case scenario of the algorithm. Finally, multiply these two values.

Example:

```plaintext
function LinearSearch(A, v, i)
    if i > length[A] then return NIL
    else if v = A[i] then return i
    else return LinearSearch(A, v, i+1)
end if
end function
```