Find the number of statements executed by each algorithm in the worst case. Our rules for counting statements are on the following page.

1. Linear search: search an unsorted array for a key. If it’s in the list, return its subscript. If it’s not in the list, return -1. Also characterize the input that gives the worst case performance.

```c
// A: the list (input)
// n: the number of elements in the list (input)
// key: the value being searched for (input)
int Lin_search(int A[], int n, int key) {
    int i;
    for (i = 0; i < n; i++)
        if (A[i] == key) return i;
    return -1;
} /* Lin_search */
```

Note that in the worst case, key is not in the list. So the statement return i will never be executed, and the loop will be exited when i = n. (See below for further discussion of this.)

Also note that the rules for counting did not include the final test: either including it or omitting it is fine.

<table>
<thead>
<tr>
<th>Initialization i = 0:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>One iteration of for</td>
<td></td>
</tr>
<tr>
<td>Test i &lt; n (true):</td>
<td>1</td>
</tr>
<tr>
<td>Test A[i] == key (false):</td>
<td>1</td>
</tr>
<tr>
<td>Increment i++:</td>
<td>1</td>
</tr>
<tr>
<td>----------------------</td>
<td>---</td>
</tr>
<tr>
<td>Test i &lt; n (false):</td>
<td>1</td>
</tr>
<tr>
<td>Branch out of loop:</td>
<td>1</td>
</tr>
<tr>
<td>return -1:</td>
<td>1</td>
</tr>
</tbody>
</table>
If we write the `for` in the form

\[
\text{for (i = 0; i <= n-1; i++)}
\]

we can use the rule that

\[
\text{for (i = a; i <= b; i++)}
\]

executes \(b - a + 1\) iterations, and get that the `for` will execute

\[
(n - 1) - 0 + 1 = n \text{ iterations.}
\]

If we count the final test, this gives

\[
T(n) = 1 + 3n + 3 = 3n + 4 \text{ statements.}
\]

If we omit the final test (including it with branch), this gives

\[
T(n) = 1 + 3n + 2 = 3n + 3 \text{ statements.}
\]

Either is fine.

It might seem that worst case occurs when \(A[n-1] == \text{key}\), but in this case we get

\[
\begin{align*}
\text{Initialization i = 0:} & \quad 1 \\
\text{++-- One iteration of for} & \\
\text{|Test i < n (true):} & \quad 1 \\
\text{|Test A[i] == key (false):} & \quad 1 \\
\text{|Increment i++:} & \quad 1 \\
\text{++--} & \\
\text{Test i < n (true):} & \quad 1 \\
\text{Test A[n-1] == key (true):} & \quad 1 \\
\text{return n-1:} & \quad 1
\end{align*}
\]

Now we get \(n - 1\) iterations of the `for` statement when \(A[i] == \text{key}\) is false. So the total number of statements is

\[
T(n) = 1 + 3(n - 1) + 3 = 3n + 1,
\]

and worst case performance occurs when \(\text{key}\) is not in the list.

2. Vector dot product: find the dot-product of two \(n\)-element arrays: find the product of corresponding elements and return their sum.
// x:  the first n-element input array (input)
// y:  the second n-element input array (input)
// n:  the number of elements in x and y.
int Dot_product(int x[], int y[], int n) {
    int i, dot;
    dot = 0;
    for (i = 0; i < n; i++)
        dot = dot + x[i]*y[i]; // add, mult and assign = 3 statements
    return dot;
} /* Dot_prod */

Now we count

\[
\begin{align*}
\text{Initialization} & \quad \text{dot} = 0; & 1 \\
\text{Initialization} & \quad \text{i} = 0; & 1 \\
\text{--- One iteration of for} & \\
\text{|Test i < n (true):} & 1 \\
\text{|Add, mult, and assign:} & 3 \\
\text{|Increment i++:} & 1 \\
\text{---} & \\
\text{Test i < n (false):} & 1 \\
\text{Branch out of loop:} & 1 \\
\text{return dot:} & 1 \\
\end{align*}
\]

As before we get \((n - 1) - 0 + 1 = n\) iterations. So if we include the final test before exiting the loop, we get

\[
T(n) = 2 + 5n + 3 = 5n + 5\text{ statements.}
\]

If we omit the final test (including it with the branch), we get

\[
T(n) = 2 + 5n + 2 = 5n + 4\text{ statements.}
\]

**Some Rules for Counting Statements**

1. Declaring a variable requires no statements.

2. Assigning to a variable requires one statement.

3. A `for` statement *usually* requires
   (a) One initialization
(b) \( k \) increments
(c) \( k \) tests
(d) (Final test)
(e) One branch

4. An if statement requires

(a) At least one statement for the test.
(b) (Technically, we should allow a statement for a branch, but we’ll include this with the test.)