Example 4. Linear Search.

```java
public boolean linearSearch(int[] list, int key)
{
    int n = list.length;
    for(int i = 0; i < n; i++)
        if(key == list[i])
            return true;
    return false;
}
```

Here \( n \) is the number of slots in the array \( list \) that we wish to search. It therefore represents the input size. Since the code inside the loop runs in constant time, the running time of linear search is linear in \( n \) in the worst case. Note that the running time is not always linear in \( n \); \( key \) may be in the first slot all the time and in such cases the running time is just constant. Therefore, it is important to add the qualifier “in the worst case.”

Example 5. Binary Search.

We discussed binary search a few classes ago. Here is that code:

```java
public static boolean binarySearch(int[] list, int n, int key)
{
    int first = 0; int last = n-1;
    int mid;
    while(first <= last)
    {
        mid = (first + last)/2;
        if(list[mid] == key)
            return true;
        else if(list[mid] < key)
            last = mid - 1;
        else if(list[mid] > key)
            first = mid + 1;
    }
    return false;
}
```

We will do a worst case analysis of the code. In other words, we will assume that \( key \) is not to be found and the while-loop terminates only when \( (first > last) \). The code fragment that performs initializations (before the start of the while-loop) runs in constant time. Also, the code fragment inside the loop runs in constant time. Therefore the worst case running time of the code fragment is

\[
A + B \cdot t,
\]

where \( A \) and \( B \) are constants independent of \( n \) and \( t \) is the number of times the while-loop executes, in the worst case.
Number of times the while loop has executed

<table>
<thead>
<tr>
<th>Size of array to be examined $(last - first + 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>$i$</td>
</tr>
</tbody>
</table>

After $t$ iterations of the while-loop, the size of the “yet-to-searched” array becomes 0. This means that after $t - 1$ iterations, this size must be 1. Note that by consulting the above table, we see that the after $t - 1$ iterations, the size of the “yet-to-searched” array is $n/2^{t-1}$. For this to be 1, it must be the case that $2^{t-1} = n$, and this happens when $t - 1 = \log_2(n)$. Therefore, $t = \log_2(n) + 1$ and the overall running worst case running time of binary search is $(A + B) + B \cdot \log_2(n)$. Later we will see that it is not necessary to explicitly specify the base of the logarithm and in general, we say that the running time of binary search is logarithmic.

**Logarithmic functions.** If $a^b = x$, then $b = \log_a(x)$. In other words, $\log_a(x)$ is the quantity to which $a$ has to be raised to get to $x$. So, as in the previous example, if $2^i = n$, then $i = \log_2(n)$. The function $\log_2(n)$ grows very slowly as compared to the linear function, $n$. For illustration, consider this table.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>2048</td>
<td>11</td>
</tr>
<tr>
<td>4096</td>
<td>12</td>
</tr>
<tr>
<td>8192</td>
<td>13</td>
</tr>
<tr>
<td>16384</td>
<td>14</td>
</tr>
<tr>
<td>32768</td>
<td>15</td>
</tr>
<tr>
<td>65536</td>
<td>16</td>
</tr>
<tr>
<td>131072</td>
<td>17</td>
</tr>
<tr>
<td>262144</td>
<td>18</td>
</tr>
<tr>
<td>524288</td>
<td>19</td>
</tr>
<tr>
<td>1048576</td>
<td>20</td>
</tr>
</tbody>
</table>

Even when $n$ exceeds a million, $\log_2(n)$ is still at 20. This means that even for a million element array, binary search examines (in the worst case) about 20 elements!

One formula concerning logarithms that you should know is called the change of base formula. This is

$$\log_b n = \frac{\log_a n}{\log_a b}.$$
This shows how logarithms to different bases relate to each other. For example, this formula tells us that
\[ \log_3 n = \frac{\log_2 n}{\log_2 3} = \frac{\log_2 n}{1.58496}, \]
implying that \( \log_3 n \) is about two-thirds of \( \log_2 n \). So logarithms to different fixed bases are within a constant times each other. This is why, in running time analysis, we tend to write \( \log_2 n \) as \( \log n \), since it does not matter what fixed base we use for the logarithm.

“Big Oh” notation
Our run-time analysis aims to ignore machine-dependent aspects of the running time. For example, when we showed that the running time of a code fragment was \( A \cdot n + B \), we don’t care about the constants \( A \) or \( B \) because these depend on the machine. We simply focus on the fact that the shape of the function \( A \cdot n + B \) is linear. In other words, we “approximate” \( A \cdot n + B \) by \( n \). The “Big Oh” notation permits a mathematically precise way of doing this.

**Definition:** Let \( f(n) \) and \( g(n) \) be functions defined on the set of natural numbers. A function \( f(n) \) is said to be \( O(g(n)) \) if there exist positive constants \( c \) and \( n_0 \) such that for all \( n > = n_0 \), \( f(n) < = c \cdot g(n) \).

Informally speaking, \( f(n) \) is \( O(g(n)) \) if there is a multiple of \( g(n) \) that eventually overtakes \( f(n) \).

![Figure 1: \( f(n) = O(g(n)) \) because there are positive constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).](image)

**Example 1.** Show that \( 5n + 20 = O(n) \).
To see this, let \( c = 6 \). \( 6n \) is a linear function with a greater slope than \( 5n + 20 \). So it is clear that eventually \( 6n \) will overtake \( 5n + 20 \). At what point does this happen? Solving \( 6n = 5n + 20 \), we see that \( 6n \) and \( 5n + 20 \) meet at \( n = 20 \). So for all \( n \geq 20 \), \( 6n \geq 5n + 20 \).

**Example 2.** Let \( A \) and \( B \) be arbitrary constants, with \( A > 0 \). Show that \( An + B = O(n) \).
Let \( c = A + 1 \). Then, we observe that \( (A + 1) \cdot n \geq An + B \), for all \( n \geq B \). This example is telling us that whenever the running time of an algorithm has the form \( An + B \), we can simply say that the running time is \( O(n) \).

**Example 3.** Show that \( 8n^2 + 10n + 25 = O(n^2) \).
As in the previous examples, let us select \( c = 9 \). We need to ask, when does \( 9n^2 \) start overtaking \( 8n^2 + 10n + 25 \)?

\[
\frac{9n^2}{8n^2 + 10n + 25} \geq 1 \Rightarrow n^2 \geq \frac{10n + 25}{9} \Rightarrow (n - 10) \cdot n \geq 25
\]
Now note that at $n = 12$, the left hand side (LHS) = 12 and the above inequality is not satisfied. However, at $n = 13$, the LHS = 39 and the inequality is satisfied. Furthermore, LHS is an increasing function of $n$ and therefore the inequality continues to be satisfied for all larger $n$ as well. In summary, we can set $c = 9$ and $n_0 = 13$.

**Example 4.** Show that $8n^2 + 10n + 25$ is not $O(n)$.

To obtain a contradiction suppose there are constants $c$ and $n_0$ such that

$$8n^2 + 10n + 25 \leq cn \text{ for all } n \geq n_0.$$  

Clearly, $c$ has to be larger than 10. So let us assume this. Then, the above inequality implies

$$8n^2 \leq (c - 10)n - 25 \text{ for all } n \geq n_0.$$  

Now pick $n = k(c - 10)$ where $k$ is a natural number such that $k(c - 10) \geq n_0$. Then the $LHS = 8k^2(c - 10)^2$ and the $RHS = k(c - 10)^2 - 25$. Clearly, the LHS is larger than the RHS - a contradiction.