Problem 1.2-6: How can we modify almost any algorithm to have a good best-case running time?

To improve the best case, all we have to do it to be able to solve one instance of each size efficiently. We could modify our algorithm to first test whether the input is the special instance we know how to solve, and then output the canned answer.

For sorting, we can check if the values are already ordered, and if so output them. For the traveling salesman, we can check if the points lie on a line, and if so output the points in that order.

The supercomputer people pull this trick on the linpack benchmarks!

Because it is so easy to cheat with the best case running time, we usually don’t rely too much about it.

Because it is usually very hard to compute the average running time, since we must somehow average over all the instances, we usually strive to analyze the worst case running time.

The worst case is usually fairly easy to analyze and often close to the average or real running time.
Exact Analysis is Hard!

We have agreed that the best, worst, and average case complexity of an algorithm is a numerical function of the size of the instances.

However, it is difficult to work with exactly because it is typically very complicated!

Thus it is usually cleaner and easier to talk about *upper and lower bounds* of the function.

This is where the dreaded big O notation comes in!

Since running our algorithm on a machine which is twice as fast will effect the running times by a multiplicative constant of 2 - we are going to have to ignore constant factors anyway.
Names of Bounding Functions

Now that we have clearly defined the complexity functions we are talking about, we can talk about upper and lower bounds on it:

- \( g(n) = O(f(n)) \) means \( C \times f(n) \) is an upper bound on \( g(n) \).

- \( g(n) = \Omega(f(n)) \) means \( C \times f(n) \) is a lower bound on \( g(n) \).

- \( g(n) = \Theta(f(n)) \) means \( C_1 \times f(n) \) is an upper bound on \( g(n) \) and \( C_2 \times f(n) \) is a lower bound on \( g(n) \).

Got it? \( C, C_1, \) and \( C_2 \) are all constants independent of \( n \).

All of these definitions imply a constant \( n_0 \) beyond which they are satisfied. We do not care about small values of \( n \).
The value of \( n_0 \) shown is the minimum possible value; any greater value would also work.

(a) \( f(n) = \Theta(g(n)) \) if there exist positive constants \( n_0, c_1, \) and \( c_2 \) such that to the right of \( n_0 \), the value of \( f(n) \) always lies between \( c_1 \cdot g(n) \) and \( c_2 \cdot g(n) \) inclusive.

(b) \( f(n) = O(g(n)) \) if there are positive constants \( n_0 \) and \( c \) such that to the right of \( n_0 \), the value of \( f(n) \) always lies on or below \( c \cdot g(n) \).

(c) \( f(n) = \Omega(g(n)) \) if there are positive constants \( n_0 \) and \( c \) such that to the right of \( n_0 \), the value of \( f(n) \) always lies on or above \( c \cdot g(n) \).

Asymptotic notation \((O, \Theta, \Omega)\) are as well as we can practically deal with complexity functions.
What does all this mean?

\[ 3n^2 - 100n + 6 = O(n^2) \text{ because } 3n^2 > 3n^2 - 100n + 6 \]
\[ 3n^2 - 100n + 6 = O(n^3) \text{ because } .01n^3 > 3n^2 - 100n + 6 \]
\[ 3n^2 - 100n + 6 \neq O(n) \text{ because } c \cdot n < 3n^2 \text{ when } n > c \]

\[ 3n^2 - 100n + 6 = \Omega(n^2) \text{ because } 2.99n^2 < 3n^2 - 100n + 6 \]
\[ 3n^2 - 100n + 6 \neq \Omega(n^3) \text{ because } 3n^2 - 100n + 6 < n^3 \]
\[ 3n^2 - 100n + 6 = \Omega(n) \text{ because } 10^{10^{10}} n < 3n^2 - 100 + 6 \]

\[ 3n^2 - 100n + 6 = \Theta(n^2) \text{ because } O \text{ and } \Omega \]
\[ 3n^2 - 100n + 6 \neq \Theta(n^3) \text{ because } O \text{ only} \]
\[ 3n^2 - 100n + 6 \neq \Theta(n) \text{ because } \Omega \text{ only} \]

Think of the equality as meaning \textit{in the set of functions}.

Note that time complexity is every bit as well defined a function as \(\sin(x)\) or you bank account as a function of time.
Testing Dominance

$f(n)$ dominates $g(n)$ if $\lim_{n \to \infty} g(n)/f(n) = 0$, which is the same as saying $g(n) = o(f(n))$.

Note the little-o–it means “grows strictly slower than”.

Knowing the dominance relation between common functions is important because we want algorithms whose time complexity is as low as possible in the hierarchy. If $f(n)$ dominates $g(n)$, $f$ is much larger (ie. slower) than $g$.

- $n^a$ dominates $n^b$ if $a > b$ since
  \[
  \lim_{n \to \infty} \frac{n^b}{n^a} = n^{b-a} \to 0
  \]

- $n^a + o(n^a)$ doesn’t dominate $n^a$ since
  \[
  \lim_{n \to \infty} \frac{n^a}{n^a + o(n^a)} \to 1
  \]

<table>
<thead>
<tr>
<th>Complexity</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>0.00001 sec</td>
<td>0.00002 sec</td>
<td>0.00003 sec</td>
<td>0.00004 sec</td>
</tr>
<tr>
<td>$n^2$</td>
<td>0.001 sec</td>
<td>0.004 sec</td>
<td>0.009 sec</td>
<td>0.16 sec</td>
</tr>
<tr>
<td>$n^3$</td>
<td>0.001 sec</td>
<td>0.008 sec</td>
<td>0.027 sec</td>
<td>0.64 sec</td>
</tr>
<tr>
<td>$n^5$</td>
<td>0.1 sec</td>
<td>3.2 sec</td>
<td>24.3 sec</td>
<td>1.7 min</td>
</tr>
<tr>
<td>$2^n$</td>
<td>0.001 sec</td>
<td>1.0 sec</td>
<td>17.9 min</td>
<td>12.7 days</td>
</tr>
<tr>
<td>$3^n$</td>
<td>0.59 sec</td>
<td>58 min</td>
<td>6.5 years</td>
<td>3855 cent</td>
</tr>
</tbody>
</table>