§1. The problem
Consider a carpenter who is building you a porch or an addition to your house. You won’t think much of this carpenter if he or she couldn’t produce a reasonable estimate of how much lumber this job is going to require — especially since this lumber is a major part of the bill you will be paying.

Now consider a programmer who is putting together a little application program for you. You would not think much of this programmer if he or she couldn’t tell you roughly how much time various parts of the program will take on different size inputs. The following covers the beginnings of how to make such estimates.

§2. A beginning example
Let us start with looking a particular method that does something mildly interesting.

```java
public static int findMin(int a[]) {
    // PRECONDITION: a is not null.
    // RETURNS: the minimal value in
    // a[0], a[1], ..., a[a.length-1]
    int j, n, minVal;
    n = a.length; minVal = a[0];

    for (j=1; j < n; j++)
        if (a[j]<minVal) { minVal = a[j]; }
    return minVal;
}
```

Now let us ask our question

*How much time does this take?*

and notice that we are in trouble already.
Problem 1
Different arrays will produce different run times. For example, you would expect that an array of length 10000 will take longer to search than an array of length 3.

Problem 2
Different machines, different Java compilers, different Java runtime systems will generally different run times for findMin.

To deal with Problem 1, we can make our answer depend on the length of the array. That is, we can come up with a formula $F(n)$ such that $F(n)$ will be the runtime for an array of length $n$.

Finding $F$ exactly is usually hard, but it turns out that an estimate is almost always good enough.

To begin to deal with Problem 2, we make two assumptions about machines, compilers, and runtime systems.

Assumption 1
Given:
• a particular method $M$,
• a particular setup, $S_1$, of computer, compiler, and runtime,
• another particular setup, $S_2$, of computer, compiler, and runtime,
then there are positive (real) constants $c_{\text{lower}}$ and $c_{\text{upper}}$ such that

\[
 c_{\text{lower}} \leq \text{the } S_1 \text{ runtime of } M \text{ for an input of size } n \leq c_{\text{upper}}
\]

for sufficiently large values of $n$.

Assumption 1 is just a careful way of saying things of the sort:

If we run the same program on the WonderBox$^\text{TM}$ Mark V computer (which came out in 2001) and the WonderBox$^\text{TM}$ Mark XI computer (which came out in 2006), we expect the Mark XI to be between 2.5 and 3.8 times faster than the Mark V — on reasonable size problems.

What does Assumption 1 have to do with Problem 2? It says that if we figure out an estimate for one setup, then we can turn this in to an estimate for another setup by finding (or estimating) $c_{\text{lower}}$ and $c_{\text{upper}}$.

\[\text{1There are some oversimplifications here. We'll fix these later.}\]
Example: Suppose we know that \texttt{findMin} runs in $F(n)$ microseconds on a WonderBox\textsuperscript{TM} Mark XI, then we know that on a Mark V \texttt{findMin} will run between $2.5 \cdot F(n)$ and $3.8 \cdot F(n)$ microseconds — on reasonable size arrays.

Example: Suppose we know that on a Mark XI, we know only that \texttt{findMin} will run between $0.6 \cdot F(n)$ and $1.6 \cdot F(n)$ microseconds. Then we know that on a Mark V, \texttt{findMin} will run between $0.6 \cdot 2.5 \cdot F(n) = 1.5 \cdot F(n)$ and $1.6 \cdot 3.8 \cdot F(n) = 6.08 \cdot F(n)$ microseconds — on reasonable size arrays.

The second assumption helps us figure out $F(n)$ for a particular machine. This second assumption concerns straightline code. For our purposes straightline code is any section of Java code that does not contain a loop, recursion, or a call to method that somehow invokes a loop or a recursion. For example, in \texttt{findMin}

- $n = \texttt{a.length}$;
  $\texttt{minVal = a[0]}$;

- if ($\texttt{a[j]} < \texttt{minVal}$) {
  $\texttt{minVal = a[j]}$;
  }

are both examples straightline code, but

- for (j=1; j<n; j++)
  if ($\texttt{a[j]} < \texttt{minVal}$) {
    $\texttt{minVal = a[j]}$;
  }

is not straightline because of the for loop.

Assumption 2
Given:
- a particular method $M$,
- a particular section, $C$, of straightline code of method $M$,
- a particular set up, $S_1$, of computer, compiler, and runtime,
then there are positive (real) constants $c_{\text{lower}}$ and $c_{\text{upper}}$ such that

\[
c_{\text{lower}} \leq \frac{\text{the } S_1 \text{ runtime of } C \text{ for size } n \text{ input}}{1 \text{ microsecond}} \leq c_{\text{upper}}
\]

for any value of $n$. 

\textit{CIS 575 Autumn 2008} 26 August 2009
Assumption 2 says that under a particular setup, the time to run a particular piece of straightline code does not depend on the input size. Assumption 2 also says that it is the loops and the recursions that cause the dependence on $n$ — because it sure isn’t the straightline code. So here is what we do to get $F(n)$, or at least an estimate of $F(n)$.

1. We identify the pieces of straightline code that are buried deepest in the method’s loops.

   **Example:** In `findMin` the straightline code that is buried deepest is
   
   ```java
   if (a[j]<minVal) { minVal = a[j]; }
   ```
   
   since it occurs within the loop. Whereas

   ```java
   n = a.length; minVal = a[0];
   ```
   
   occurs outside the loop.

2. We count how many times these innermost pieces of the program are executed for a given value of $n$.

   **Example:** The loop of `findMin` is
   
   ```java
   for (j=1; j<n; j++)
     if (a[j]<minVal) { minVal = a[j]; }
   ```
   
   So, the first iteration has $j==1$, the last iteration has $j==n-1$, and $j$ increases by 1 with each iteration. So, there are $(n-1) - (1) +1 == n-1$ many iterations.

3. If the formula we get from step 2 is $G(n)$, then it follows from Assumption 2, it follows that there are positive constants $c_{lower}$ and $c_{upper}$ such that the run time of our method is between $c_{lower} \cdot G(n)$ and $c_{upper} \cdot G(n)$.

   **Example:** From step 2, our formula is $n - 1$. So, for the WonderBox™ Mark XI, there are positive constants $c_\ell$ and $c_u$ such that, for any array of length $n$,

   $$c_\ell \cdot (n - 1) \mu \text{ secs} \leq \text{the runtime of } findMin \leq c_u \cdot (n - 1) \mu \text{ secs}.$$ 

---

2Again, we are oversimplifying things.
For a slightly different choice of positive constants $c'_\ell$ and $c'_u$, we have

$$c'_\ell \cdot n \ \mu \text{secs} \leq \text{the runtime of } \text{findMin} \leq c'_u \cdot n \ \mu \text{secs}$$

for sufficiently large $n$. This last inequality can be restated as:

$$c'_\ell \leq \frac{\text{the runtime of } \text{findMin}}{n \ \mu \text{secs}} \leq c'_u.$$

So, the runtime of $\text{findMin}$ is bounded below and above by linear functions (in $n$) on the Mark XI. By Assumption 1, the same is going to hold true (with different constants) on any other machine on which we run $\text{findMin}$. Because of this, we say that $\text{findMin}$ has linear run time. If we want to know the constants for any particular machine, we can run some timing experiments on that machine.

§3. A second example

Let us consider a more complex (and more interesting) example. Here is a Java version of the selection sort algorithm.

```java
public static void selectionSort(int a[]) {
    // PRECONDITION: a is not null.
    // POSTCONDITION: a is sorted in increasing order
    int i, j, minJ, minVal, n;
    n = a.length;

    for (i=0; i < n-1; i++) {
        // find a minJ ∈ {i,...,n-1} such that
        // a[minJ] = min {a[j] : i ≤ j ≤ n-1}
        minJ = i; minVal = a[i];
        for (j=i+1; j<n; j++)
            if (a[j]<minVal) { minJ = j; minVal = a[j]; }

        // Swap the values of a[i] and a[minJ]
        a[minJ] = a[i]; a[i] = minVal;

        // Now we have a[0] ≤ a[1] ≤⋯≤ a[i]
        // ≤ min {a[j] : i < j ≤ n-1}
    }
}
```

CIS 575 Autumn 2008 26 August 2009
Selection sort works by first finding the smallest value in the array and making $a[0]$ equal to this value (by swapping the positions some elements in the array). Then it finds the next smallest element in $a$ and makes $a[1]$ equal to this value (again by swaps). It keeps on doing this with $a[2]$, $a[3]$, ... until it has a sorted array.

Let us analyze `selectionSort`’s runtime the same way we did with `findMin`.

1. **Identify the pieces of straightline code that are buried deepest.**
   In `selectionSort` this is:
   ```java
   if (a[j]<minVal) { minJ = j; minVal = a[j];}
   ```
   since it occurs within both loops. Whereas both
   - $\text{minJ} = i; \text{minVal} = a[i];$
   - $a[\text{minJ}] = a[i]; a[i] = \text{minVal};$
   occur within only the outermost loop.

2. **Count how many times these innermost pieces of the program are executed for a given value of $n$.**
   We handle the two loops in `selectionSort` one at a time, starting with the innermost.
   
   **THE INNERMOST LOOP:** This is
   ```java
   for (j=i+1; j<n; j++)
   if (a[j]<minVal) {\text{minJ} = j; \text{minVal} = a[j];}
   ```
   So, the first iteration has $j=i+1$, the last iteration has $j=n-1$, and $j$ increases by 1 with each iteration. So, there are $(n-1) - (i+1) + 1 = n-i-1$ many iterations. So for particular values of $i$ and $n$ the innermost code is executed $n-i-1$ times every time the innermost for loop is executed.
THE OUTERMOST LOOP: This is

\[
\text{for } (i=0; i<n-1; i++) \{
\quad \text{minJ} = i; \text{minVal} = a[i];
\quad \text{— the innermost loop —}
\quad a[\text{minJ}] = a[i]; a[i] = \text{minVal};
\}
\]

The first iteration thus has \( i = 0 \), the last iteration has \( i = n - 2 \), and \( i \) increases by 1 with each iteration. So, the number of times the innermost code is executed is:

<table>
<thead>
<tr>
<th>iteration #</th>
<th>value of ( i )</th>
<th># of executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>( n-1 )</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>( n-2 )</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>( n-3 )</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>n-3</td>
<td>n-4</td>
<td>3</td>
</tr>
<tr>
<td>n-2</td>
<td>n-3</td>
<td>2</td>
</tr>
<tr>
<td>n-1</td>
<td>n-2</td>
<td>1</td>
</tr>
</tbody>
</table>

Therefore, the total number of executions is

\[
1 + 2 + 3 + \cdots + (n - 1).
\]

By a standard bit of maths this last sum equals

\[
\frac{(n - 1) \cdot n}{2} = \frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n.
\]

3. Take the count from step 2 and conclude the “order” of the runtime on any particular machine.

So for our faithful Mark XI, there are positive constants \( c_\ell \) and \( c_u \) such that, for any array of length \( n \),

\[
c_\ell \cdot (\frac{1}{2} n^2 - \frac{1}{2} n) \ \mu \text{secs}
\leq \text{the runtime of selectionSort} \leq c_u \cdot (\frac{1}{2} n^2 - \frac{1}{2} n) \ \mu \text{secs}.
\]
For a slightly different choice of positive constants $c'_\ell$ and $c'_u$, we have
\[ c'_\ell \cdot n^2 \text{ µ secs} \leq \text{the runtime of } \text{selectionSort} \leq c'_u \cdot n^2 \text{ µ secs} \]
for sufficiently large $n$.

So, the runtime of selectionSort is bounded below and above by quadratic functions (in $n$) on the Mark XI. By Assumption 1, the same is going to hold true (with different constants) on any other machine on which we run selectionSort. Because of this, say that selectionSort has quadratic run time.

§4. Proportional rates of growth

The examples show that we need a way of talking about run times (or functions in general) that have a linear or quadratic growth rate — without having to bring in the irritating constants all the time.

Convention: Let $f$ and $g$ range over functions from $\mathbb{N} = \{0, 1, 2, \ldots\}$ to $\mathbb{N}^+ = \{1, 2, 3, \ldots\}$.

The collection of functions that have linear growth rate is the set:
\[
\left\{ f : \text{for some } c_\ell, c_u > 0, \quad \left. \begin{array}{c}
\text{for all sufficiently large } n \\ 
\frac{c_\ell}{n} \leq f(n) \leq \frac{c_u}{n}
\end{array} \right\} \right.
\]
Let us give this set the funny name $\Theta(n)$. So we can now say that findMin has a run time in $\Theta(n)$.

The collection of functions that have quadratic growth rate is the set:
\[
\left\{ f : \text{for some } c_\ell, c_u > 0, \quad \left. \begin{array}{c}
\text{for all sufficiently large } n \\ 
\frac{c_\ell}{n^2} \leq f(n) \leq \frac{c_u}{n^2}
\end{array} \right\} \right.
\]
Let us give this set the funny name $\Theta(n^2)$. So we can now say that selectionSort has a run time in $\Theta(n^2)$.

In general, the collection of functions that have growth rate like some given function $g$ is the set:
\[
\left\{ f : \text{for some } c_\ell, c_u > 0, \quad \left. \begin{array}{c}
\text{for all sufficiently large } n \\ 
\frac{c_\ell}{g(n)} \leq f(n) \leq \frac{c_u}{g(n)}
\end{array} \right\} \right.
\]
We give this set the name $\Theta(g(n))$.

With this $\Theta$ notation we can talk in general about the growth rate of functions. Let us consider some examples.

**Example.** Suppose $f$ is given by:

$$f(n) = 100n^2 + 8000n + 97.$$  

We claim that $f$ is in $\Theta(n^2)$. We can do this by doing a bit of high-school algebra and show that

$$100 \leq \frac{f(n)}{n^2} \leq 200$$

for all $n > 81$. Hence, $f$ belongs to the $\Theta(n^2)$ club.

**Example.** Suppose $f$ is given by:

$$f(n) = 8000n + 97.$$  

We claim that $f$ is not in $\Theta(n^2)$. To see this, suppose that there are $c_\ell, c_u > 0$ such that

$$c_\ell \leq \frac{8000n + 97}{n^2} \leq c_u$$

for all sufficiently large $n$. But this can’t be since by choosing $n$ large enough we can make

$$\frac{8000}{n} + \frac{97}{n^2} \leq c_\ell.$$  

So no such $c_\ell$ can exist.

There is a fast way to do examples such as those above.

**The Limit Rule (Version 1)** Suppose that

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

where $0 \leq c \leq +\infty$.

(a) If $0 < c < +\infty$, then $f$ is in $\Theta(g(n))$.

(b) If $c = 0$ or $c = +\infty$, then $f$ is not in $\Theta(g(n))$. 

---

CIS 575 Autumn 2008 26 August 2009
For example,
\[ \lim_{n \to \infty} \frac{100n^2 + 8000n + 97}{n^2} = 100. \]
Therefore, \(100n^2 + 8000n + 97\) is in \(\Theta(n^2)\). Also
\[ \lim_{n \to \infty} \frac{8000n + 97}{n^2} = 0. \]
Therefore, \(8000n + 97\) is not in \(\Theta(n^2)\).

§5. A third example

Now let us deal with some of the oversimplifications we mentioned above. Another sorting algorithm helps show what the problem is.

```java
public static void insertionSort(int a[]) {
    // PRECONDITION: a is not null.
    // POSTCONDITION: a is sorted in increasing order
    int i, j, key, n;
    n = a.length;
    for (i=1; i < n; i++) {
        // We assume a[0],...,a[i-1] is already sorted.
        // We save the initial value of a[i] in key.
        // All the values in a[0], ..., a[i-1] that are
        // larger than key are shifted up one position.
        // This opens a hole between the values \(\leq\) key and
        // the values \(>\) key.
        // We place the value in key in this hole.
        key = a[i]; j = i-1;
        while (j>=0 && a[j]>key) {
            a[j+1] = a[j]; j = j -1;
        }
        a[j+1] = key;
        // Now a[0] \(\leq\) a[1] \(\leq\) ... \(\leq\) a[i].
    }
}
```

Insertion sort has the property that it is much faster on some inputs than others. Here are the two key examples.
1. Suppose the array to be sorted is already in increasing order. Then in every iteration of the for-loop, the while-loop test \(a[j] > \text{key}\) will fail the first time we try it. So, we go through no iterations of the while-loop at all and a little work shows that on already sorted inputs, \text{insertionSort}\ runs in \(\Theta(n)\) time.

2. Suppose the array to be sorted is already in decreasing order (that is, from biggest to smallest). Then in every iteration of the for-loop, the while-loop has to go through \(i\)-many iterations since initially all of the values in \(a[0], \ldots, a[i-1]\) are larger than \text{key}. So an analysis similar to that for \text{selectionSort}\ shows that for inputs that are sorted in decreasing order, \text{insertionSort}\ runs in \(\Theta(n^2)\) time.

Hence, for a particular algorithm, inputs of the same size can result in quite different run times.

**Terminology:** The best case run time of an algorithm (or program or method) on an input of size \(n\) is the best (i.e., smallest) run time of the algorithm on size-\(n\) inputs. The worst case run time of an algorithm (or program or method) on an input of size \(n\) is the worst (i.e., biggest) run time of the algorithm on size-\(n\) inputs. For example,

- \text{findMin}\ has linear best and worst case run times.
- \text{selectionSort}\ has quadratic best and worst case run times.
- \text{insertionSort}\ has linear best case and quadratic worst case run times.

To deal with the fact that the order of the best and worse case run times may not match, we introduce ways of taking about upper and lower bounded on growth rates.

**§6. Upper and lower bounds on rates of growth**

Recall that

\[
\Theta(g(n)) = \left\{ f : \begin{array}{l}
\text{for some } c_l, c_u > 0, \\
\text{for all sufficiently large } n \\
c_l \leq f(n)/g(n) \leq c_u
\end{array} \right\}.
\]
To talk about upper and lower bounds on rates of growth, we break $\Theta$ in two parts. For a given $g$, define

$$O(g(n)) = \left\{ f : \begin{array}{l} \text{for some } c_u > 0, \\ f(n)/g(n) \leq c_u \\ \text{for all sufficiently large } n \end{array} \right\}. $$

$$\Omega(g(n)) = \left\{ f : \begin{array}{l} \text{for some } c_\ell > 0, \\ c_\ell \leq f(n)/g(n) \\ \text{for all sufficiently large } n \end{array} \right\}. $$

Intuitively:

- $\Theta(g(n))$ is the collection of functions that have growth rates proportional to $g(n)$.
- $O(g(n))$ is the collection of functions $f(n)$ such that $f(n)/g(n)$ is no worse than some constant for large $n$.
- $\Omega(g(n))$ is the collection of functions $f(n)$ such that $f(n)/g(n)$ is no smaller than some positive constant for large $n$.

**Theorem.** $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$. That is, $f$ is in $\Theta(g(n))$ if and only if $f$ is in $O(g(n))$ and $f$ is in $\Omega(g(n))$.

**Example.** Suppose $f$ is given by:

$$f(n) = 8000n + 97.$$  

We claim that $f$ is in $O(n^2)$. To see this, use some high school math to show that

$$8000n + 97 \leq 8000 \cdot n^2$$

for all $n > 3$.

**Example.** Suppose $f$ is given by:

$$f(n) = 100n^2 + 8000n + 97.$$
We claim that $f$ is in $\Omega(n)$. To see this, use some high school math to show that
\[ n \leq 100n^2 + 8000n + 97 \]
for all $n > 0$.

We can extend the limit rule to help with these $O$ and $\Omega$ problems.

**The Limit Rule (Version 2)** Suppose that
\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \]
where $c$ is such that $0 \leq c \leq +\infty$.

(a) If $0 < c < +\infty$, then $f$ is in $\Theta(g(n))$, $O(g(n))$, and $\Theta(g(n))$.
(b) If $c = 0$, then $f$ is in $O(g(n))$, but not in either $\Theta(g(n))$ or $\Omega(g(n))$.
(c) If $c = \infty$, then $f$ is in $\Omega(g(n))$, but not in $\Theta(g(n))$ or $O(g(n))$.

§7. Problems

**Part I.** For each pair of $f$’s and $g$’s below, fill true or false in each box on that line. Justify your answers!!!! Use the “Math Facts” on page 16 below.

<table>
<thead>
<tr>
<th></th>
<th>$f(n)$</th>
<th>$g(n)$</th>
<th>$f(n)$ in $O(g(n))$</th>
<th>$f(n)$ in $\Theta(g(n))$</th>
<th>$f(n)$ in $\Omega(g(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1.</td>
<td>$8000n + 97$</td>
<td>$n^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.</td>
<td>$(1 + (\cos n)^2) \cdot n$</td>
<td>$n^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>$\frac{1}{3\pi} n^3$</td>
<td>$n^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>$n$</td>
<td>$(n^2 - 1)/(n + 1)$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>$\log_2(n^n)$</td>
<td>$n \sqrt{n}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>$2^{2n}$</td>
<td>$2^n$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>$2^{\log_2 n}$</td>
<td>$n^2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>$(\log_2 n)^{100}$</td>
<td>$n^{1/100}$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example: Problem -1.** We use the limit rule.
\[ \lim_{n \to \infty} \frac{8000n + 97}{n^2} = \lim_{n \to \infty} \frac{8000}{n} + \frac{97}{n^2} = 0. \]

Therefore, $f$ is in $O(g(n))$ but not in either $\Theta(g(n))$ or $\Omega(g(n))$. 
Example: Problem 0. We use the limit rule.

\[ \lim_{n \to \infty} \frac{2 + \sin n}{n^2} \leq \lim_{n \to \infty} \frac{3}{n^2} \quad \text{(since } (2 + \sin n) \leq 3, \text{ for all } n) \]

\[ = 0. \]

Therefore, \( f \) is in \( O(g(n)) \) but not in either \( \Theta(g(n)) \) or \( \Omega(g(n)) \).

Part II. For each program fragment below, give a \( \Theta(\ ) \) expression (in parameter \( n \)) for the value of \( z \) after the program fragment is finished. Justify your answers!!!!

7.

```c
int z = 1;
for (int i = 1 ; i < n-2 ; i++)
    z = z + 1;
```

8.

```c
int z = 1;
for (int k = 1 ; k <= n ; k++)
    for (int j = n ; j > 0 ; j--)
        z = z + 1;
```

9.

```c
int z = 1;
for (int k = 1 ; k <= n ; k++)
    for (int j = 0 ; j < n ; j++)
        z = z + 1;
```

10.

```c
int z = 1;
int m = n;
while (m>1) {
    z = z + 1;
    m = m/2;
}
```
11.

```c
int z = 1;
for (int i=0; i < n-1; i++)
    for (int j=i+1; j<n; j++)
        z = z + 1;
```

**A Chatty Example: Problem 11.** Since there are nested loops, we work from the inside-out.

**The Innermost Loop:** This is

```c
for (int j=i+1; j<n; j++)
    z = z + 1
```

So, the first iteration has $j=i+1$, the last iteration has $j=n-1$, and $j$ increases by 1 with each iteration. So, there are $(n-1) - (i+1) + 1 = n-i-1$ many iterations. So for particular values of $i$ and $n$ the innermost code is executed $n-i-1$ times every time the innermost for loop is executed.

**The Outermost Loop:** This is

```c
for (int i=0; i<n-1; i++) {
    — the innermost loop —
}
```

The first iteration thus has $i=0$, the last iteration has $i=n-2$, and $i$ increases by 1 with each iteration. So, the number of times the innermost code is executed is:

<table>
<thead>
<tr>
<th>iteration #</th>
<th>value of $i$</th>
<th># of executions $= (n - i - 1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$n-1$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>$n-2$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$n-3$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$n-3$</td>
<td>$n-4$</td>
<td>3</td>
</tr>
<tr>
<td>$n-2$</td>
<td>$n-3$</td>
<td>2</td>
</tr>
<tr>
<td>$n-1$</td>
<td>$n-2$</td>
<td>1</td>
</tr>
</tbody>
</table>
Therefore, the total number of executions is

\[ 1 + 2 + 3 + \cdots + (n - 1) = \frac{1}{2} \cdot (n - 1) \cdot n \quad \text{is in } \Theta(n^2). \]

12. Show that \((2 + \sin n) \cdot n\) is in \(\Theta(n)\). Also explain why we can’t use the limit rule to do this problem.

§A. Some basic math facts

To do growth rates problems, it is helpful to use a few basic facts from mathematics. Here are some useful ones.

1. \(\log_2 2^n = n. \quad 2^{\log_2 n} = n.\)
2. \(a^{m \cdot n} = (a^m)^n = (a^n)^m. \quad a^m \cdot a^n = a^{m+n}.\)
3. \(\log_2 (a \cdot b) = (\log_2 a) + (\log_2 b). \quad c \cdot \log_2 a = \log_2 a^c.\)
4. \(\sum_{k=1}^n k = \frac{n \cdot (n+1)}{2}. \quad \sum_{k=1}^n k^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}.\)
5. \(\sum_{k=0}^n c^k = \frac{c^{n+1}-1}{c-1}, \text{ when } c \neq 1. \quad \sum_{k=0}^n 2^k = 2^{n+1} - 1. \quad \sum_{k=1}^\infty \frac{1}{2^k} = 1.\)
6. Suppose \(a > 1\) and \(b, c > 0\). Then:
   
   (a) \(\lim_{n \to \infty} \frac{(\log n)^b}{n^c} = 0.\)
   
   (b) \(\lim_{n \to \infty} \frac{n^b}{\log n^c} = 0.\)

7. **The Limit Rule**: If \(\lim_{n \to \infty} \frac{f(n)}{g(n)} = c\), then

<table>
<thead>
<tr>
<th>(f(n)) is in (O(g(n)))</th>
<th>(c = 0)</th>
<th>(0 &lt; c &lt; \infty)</th>
<th>(c = \infty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

§B. Proof of version 2 of the limit rule

Suppose that

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \quad (1) \]
where \( c \) is a real number such that \( 0 \leq c \leq +\infty \).

**To show part (a).** Suppose that \( 0 < c < +\infty \). In this case, (1) means that:

If I gave you a positive real \( \epsilon \),
there is a smallest whole number \( n_{\epsilon} \) you could give me
so that if I pick any number \( \geq n_{\epsilon} \), I will find
\[
    c - \epsilon < \frac{f(n)}{g(n)} < c + \epsilon.
\]
So I pick \( \epsilon = c/2 \). Then for each \( n > n_{c/2} \),
\[
    \frac{c}{2} = c - c/2 < \frac{f(n)}{g(n)} < c + c/2 = 3c/2.
\]
So with \( c_\ell = c/2 \) and \( c_u = 3c/2 \), we see that \( f \) is in \( \Theta(g(n)) \).

**To show part (b).** If \( 0 < c < +\infty \), the argument for part (a) shows that \( f \) is in \( O(g(n)) \). Now suppose that \( c = 0 \). In this case, (1) means that:

If I gave you a positive real \( \epsilon \),
there is a smallest whole number \( n_{\epsilon} \) you could give me
so that if I pick any number \( \geq n_{\epsilon} \), I will find
\[
    0 \leq \frac{f(n)}{g(n)} < \epsilon.
\]
So I pick \( \epsilon = 1/2 \). Then for each \( n > n_{1/2} \), \( 0 \leq \frac{f(n)}{g(n)} < 1/2 \). So with \( c_u = 1/2 \), we see that \( f \) is in \( O(g(n)) \).

**To show part (c).** If \( 0 < c < +\infty \), the argument for part (a) shows that \( f \) is in \( \Omega(g(n)) \). Now suppose that \( c = +\infty \). In this case, (1) means that:

If I gave you a positive real \( \epsilon \),
there is a smallest whole number \( n_{\epsilon} \) you could give me
so that if I pick any number \( \geq n_{\epsilon} \), I will find
\[
    \epsilon < \frac{f(n)}{g(n)}.
\]
So I pick \( \epsilon = 1643 \). Then for each \( n > n_{1643} \), \( 1643 < \frac{f(n)}{g(n)} \). So with \( c_\ell = 1643 \), we see that \( f \) is in \( \Omega(g(n)) \).

James S. Royer, August 2008. • This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License.