# Big-O-ology[1]

CIS 675: Algorithms

January 14, 2019

## 1. The problem

Consider a carpenter who is building you an addition to your house. You would not think much of this carpenter if she or he 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 your bill.

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.

```
public static int findMin(int[] a) {
   // PRECONDITION: a != null,
   // RETURNS: min{a[i] : i=0,..., 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?*

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 $n$.[2] Finding $F$ exactly is usually hard, but a good estimate almost always suffices.

[1]James S. Royer, Dept. of Elec. Engrg. & Computer Science Syracuse University, Syracuse, NY 13244, USA. January 2019. 

[2]There are some oversimplifications here. We'll fix these later.

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

$$c_{\text{lower}} \leq \frac{\text{the } S_1 \text{ runtime of } M \text{ for an input of size } n}{\text{the } S_2 \text{ runtime of } M \text{ for an input of size } n} \leq c_{\text{upper}}$$

for sufficiently large values of $n$.

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

> A new WonderBox™ Cloud 9 computer is between 2.5 and 3.8 times faster than an old WonderBox™ 4 machine.

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

**Example:** Suppose we know that

$$2.5 \leq \frac{\text{the WonderBox™ 4 runtime of } \texttt{findMin} \text{ for an input of size } n}{\text{the WonderBox™ Cloud 9 runtime of } \texttt{findMin} \text{ for an input of size } n} \leq 3.8$$

and that we have a formula, $F(n)$, for the run time of `findMin` on WonderBox™ 9 for arrays of length $n$. Then we know that on a WonderBox™ 4, given an array of length $n$, `findMin` will take time between $2.5 \cdot F(n)$ and $3.8 \cdot F(n)$.

Assumption 2 (below) 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 `findMin`

- `n = a.length;  minVal = a[0];`
- `if (a[j]<minVal) { minVal = a[j]; }`

are both examples straightline code, but

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

$$c_{\text{lower}} \leq \frac{\text{the } S_1 \text{ runtime of } C \text{ for size } n \text{ input}}{1 \text{ picosecond}} \leq c_{\text{upper}}$$

for any value of $n$.

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 $F(n)$.[3] For the moment we will not worry about recursions, just loops.

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:*

   ```
   if (a[j]<minVal) { minVal = a[j]; }
   ```

   *since it occurs within the loop. Whereas:*

   ```
   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:*

   ```
   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 that there are positive constants $c_{\text{lower}}$ and $c_{\text{upper}}$ such that the run time of our method is between $c_{\text{lower}} \cdot G(n)$ and $c_{\text{upper}} \cdot G(n)$.

   **Example:** *From step 2, our formula is $n - 1$. So, for the WonderBox™ Cloud 9, there are*

[3] Again, we are oversimplifying things.

> *positive constants $c_\ell$ and $c_u$ such that, for any array of length $n$,*
>
> $$c_\ell \cdot (n-1) \text{ picosecs} \leq \begin{pmatrix} \text{the runtime} \\ \text{of } \texttt{findMin} \end{pmatrix} \leq c_u \cdot (n-1) \text{ picosecs.}$$
>
> *For a slightly different choice of positive constants $c'_\ell$ and $c'_u$, we have*
>
> $$c'_\ell \cdot n \text{ picosecs} \leq \begin{pmatrix} \text{the runtime} \\ \text{of } \texttt{findMin} \end{pmatrix} \leq c'_u \cdot n \text{ picosecs}$$
>
> *for sufficiently large $n$. This last inequality can be restated as:*
>
> $$c'_\ell \leq \frac{\text{the runtime of } \texttt{findMin}}{n \text{ picosecs}} \leq c'_u.$$

So, the runtime of `findMin` is bounded below and above by linear functions (in $n$) on the Cloud 9. By Assumption 1, the same is going to hold true (with different constants) on any other machine on which we run `findMin`. Because of this, we say that `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.

```
public static void selectionSort(int[] a) {
   // PRECONDITION: a != 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}
   }
}
```

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:

   ```
   if (a[j]<minVal) { minJ = j; minVal = a[j];}
   ```

   since it occurs within both loops. Whereas both of:

   - `minJ = i; minVal = a[i];`
   - `a[minJ] = a[i]; a[i] = 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:

   ```
   for (j=i+1; j<n; j++)
     if (a[j]<minVal) {minJ = j; 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:

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

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

| iteration # | value of `i` | # of executions $= (n - i - 1)$ |
|---|---|---|
| 1 | 0 | n-1 |
| 2 | 1 | n-2 |
| 3 | 2 | n-3 |
| ⋮ | ⋮ | ⋮ |
| n-3 | n-4 | 3 |
| n-2 | n-3 | 2 |
| n-1 | n-2 | 1 |

Therefore, the total number of executions is

$$1+2+3+\cdots+(n-1).$$

By a standard bit of math

$$1+2+3+\cdots+(n-1) \;=\; \frac{(n-1)\cdot n}{2} \;=\; \frac{n^2-n}{2}.$$

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

   So for our faithful Cloud 9, there are positive constants $c_\ell$ and $c_u$ such that, for any array of length $n$,

$$c_\ell \cdot \frac{n^2-n}{2} \text{ picosecs} \;\le\; \left(\begin{array}{c}\text{the runtime of}\\ \texttt{selectionSort}\end{array}\right) \;\le\; c_u \cdot \frac{n^2-n}{2} \text{ picosecs.}$$

   For a slightly different choice of positive constants $c'_\ell$ and $c'_u$, we have

$$c'_\ell \cdot n^2 \text{ picosecs} \;\le\; \left(\begin{array}{c}\text{the runtime of}\\ \texttt{selectionSort}\end{array}\right) \;\le\; c'_u \cdot n^2 \text{ picosecs}$$

   for sufficiently large $n$.

So, the runtime of `selectionSort` is bounded below and above by quadratic functions (in $n$) on the Cloud 9. 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 $\mathbb{N} = \{0,1,2,\dots\}$ to $\mathbb{N}^+ = \{1,2,3,\dots\}$.

The collection of functions that have *linear growth rate* is the set:

$$\Theta(n) \;=_{\text{def}}\; \left\{ f \;\middle|\; \begin{array}{c}\text{for some } c_\ell,\, c_u > 0,\\ \text{for all sufficiently large } n\\ c_\ell \;\le\; f(n)/n \;\le\; c_u\end{array} \right\}.$$

($\Theta(n)$ is the funny name for this set.) 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:

$$\Theta(n^2) \;=_{\text{def}}\; \left\{ f \;\middle|\; \begin{array}{c}\text{for some } c_\ell,\, c_u > 0,\\ \text{for all sufficiently large } n\\ c_\ell \;\le\; f(n)/n^2 \;\le\; c_u\end{array} \right\}.$$

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 proportional* $g$ ($g$ is some given function) is the set:

$$\Theta(g(n)) \;=_{\text{def}}\; \left\{ f \;\middle|\; \begin{array}{c}\text{for some } c_\ell,\, c_u > 0,\\ \text{for all sufficiently large } n\\ c_\ell \;\le\; f(n)/g(n) \;\le\; c_u\end{array} \right\}.$$

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

***Aside:*** *Why the "for all sufficiently large n" in the above instead of "for all n"? The same class of functions is defined with "for all n" replacing "for all sufficiently large n", but if you actually have to find concrete values for $c_\ell$ and $c_u$, it is usually easier to ignore some noisy initial values and find $c_\ell$ and $c_u$ that work for $n \geq$ some convenient $n_0$ as in the example below.*

**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

$$100 \;\leq\; f(n)/n^2 \;\leq\; 200$$

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* $0 \leq c \leq +\infty$.
*Then:*
*(a) If* $0 < c < +\infty$, *then* $f$ *is in* $\Theta(g(n))$.
*(b) If* $c = 0$ *or* $c = \infty$, *then* $f$ *is* ***not*** *in* $\Theta(g(n))$.

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*

Here is another sorting algorithm thats analysis reveals some of the oversimplifications mentioned above.

```
public static void insertionSort(int[] a) {
   // PRECONDITION: a != 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 ≤ 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] ≤ a[1] ≤ ··· ≤ a[i].
   }
}
```

Insertion sort is much faster on some inputs than others. Here are the two key examples.

1. Suppose the array to be sorted starts in increasing order (i.e., $a[0] \leq a[1] \leq \cdots \leq a[n-1]$). Then in every iteration of the for-loop, the while-loop test `a[j]>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, `insertionSort` runs in $\Theta(n)$ time.

2. Suppose the array to be sorted starts in strictly decreasing order (i.e., $a[0] > a[1] > \cdots > a[n-1]$). 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],...,a[i-1]` are larger than `key`. So an analysis similar to that for `selectionSort` shows that for inputs that are sorted in decreasing order, `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:**

(a) 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.

(b) 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,

- `findMin` has linear best and worse case run times.

- `selectionSort` has quadratic best and worse case run times.
- `insertionSort` has linear best case and quadratic worse 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

$$\Theta(g(n)) \;=\; \left\{ f \;\middle|\; \begin{array}{c} \text{for some } c_\ell, c_u > 0, \\ \text{for all sufficiently large } n \\ c_\ell \leq f(n)/g(n) \leq c_u \end{array} \right\}.$$

To talk about upper and lower bounds on rates of growth, we break $\Theta$ in two parts. For a given $g$, define

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

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

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.** $\Theta(g(n)) = O(g(n)) \bigcap \Omega(g(n))$. *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 elementary math to show that, for all $n > 3$:

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

**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 elementary math to show that, for all $n > 0$:

$$n \leq 100n^2 + 8000n + 97.$$

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* $0 \leq c \leq +\infty$.
*Then:*
*(a) If* $0 < c < +\infty$, *then* $f$ *is in* $\Theta(g(n))$, $O(g(n))$, *and* $\Omega(g(n))$.
*(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* $O(g(n))$.

# Appendices

## *A. Some basic math facts*

To do growth rates problems, it is helpful to use a few basic facts from mathematics. Here are a few of these. The stuff about exponentiation and logarithms is included for very good reasons. (See tables 1.1 and 1.2 in *Open Data Structures*, §1.7.)

**(a)** $a^m \cdot a^n = a^{m+n}$ **(b)** $(a^m)^n = a^{m\cdot n}$ **(c)** $\log_a(x \cdot y) = \log_a x + \log_a y$

**(d)** $c \cdot \log_a x = \log_a(x^c)$ **(e)** $\log_a(a^n) = n$ **(f)** $c \cdot \log_a x = \log_a(x^c)$

**(g)** $a^{\log_a n} = n$ **(h)** $\sum_{k=1}^{n} k = \frac{n\cdot(n+1)}{2}$ **(i)** $\sum_{k=1}^{n} k^2 = \frac{n\cdot(n+1)\cdot(2n+1)}{6}$

**(j)** $\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$ **(k)** $\sum_{k=0}^{n} c^k = \frac{c^{n+1}-1}{c-1}$ for $c \neq 1$

**(ℓ)** For $a > 1$ and $b, c > 0$:

**(ℓ.1)** $\lim_{n\to\infty} \frac{(\log n)^b}{n^c} = 0.$ **(ℓ.2)** $\lim_{n\to\infty} \frac{n^b}{a^{c\cdot n}} = 0.$

**(ℓ.3)** $\lim_{n\to\infty} \frac{n^c}{(\log n)^b} = \infty.$ **(ℓ.4)** $\lim_{n\to\infty} \frac{a^{c\cdot n}}{n^b} = \infty.$

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

| | $c = 0$ | $0 < c < \infty$ | $c = \infty$ |
|---|---|---|---|
| $f(n) \in O(g(n))$ | True | True | False |
| $f(n) \in \Theta(g(n))$ | False | True | False |
| $f(n) \in \Omega(g(n))$ | False | True | True |

**Important!** You ***cannot*** use the limit rule when $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is undefined.

## *B. Problems*

**Part I.** For each pair of $f$'s and $g$'s in problems -1 through 6 below, state whether $f(n) \in \Theta(g(n))$, and if not, whether $f(n) \in O(g(n))$ or $f(n) \in \Omega(g(n))$ **and** *justify your answers with appropriate math!!!! Use the "Math Facts" on page 11.*

-1. $f(n) = 8000n^2 + 97$ $g(n) = n^2$.

0. $f(n) = 3 + \sin n$ $g(n) = n^2$.

1. $f(n) = n^3$ $g(n) = 10000000n^2$.

2. $f(n) = (\log_2 n)^{100}$ $g(n) = \log_2(n^{100})$.

3. $f(n) = 49n + (\log_2 n)^4$ $g(n) = 5(\log_2 n)^{76}$.

4. $f(n) = 6^{n/2}$ $g(n) = 6^n$.

5. $f(n) = n^{100}$ $g(n) = 2^{\sqrt{n}}$.

6. $f(n) = 17^{\log_2 n}$ $g(n) = n^{\log_2 17}$.

7. Show that $f(n) \in \Theta(n^2)$ where $f(n) = (9 + 2\cos n) \cdot n^2 + 12n$.
   Also explain why we *cannot* use the limit rule to do this problem.
   Would we have the same problem with $f(n) = (9 + 2\cos n) \cdot n + 12n^2$.?

   **Example: Problem -1.** We use the limit rule.

$$\lim_{n\to\infty} \frac{8000n^2 + 97}{n^2} \quad = \quad \lim_{n\to\infty} \left(8000 + \frac{97}{n^2}\right) \quad = \quad 8000.$$

   Therefore, $f$ is in $\Theta(g(n))$.

   **Example: Problem 0.** We use the limit rule.

$$\lim_{n\to\infty} \frac{3 + \sin n}{n^2} \leq \lim_{n\to\infty} \frac{4}{n^2} \qquad \text{(since } (3 + \sin n) \leq 4\text{, for all } n\text{)}$$
$$= 0.$$

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

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

8.
```
int z = 1;
for (int i = 0 ; i <= n ; i++)
  z = z + 1;
```

9.
```
int z = 1;
for (int k = 0 ; k <= n ; k++) {
  int j = n+1;
  while (j>0) {
    z = z + 1;
    j = j - 1;
  }
}
```

10.
```
int z = 1;
for (int k = 1 ; k <= n ; k++)
  for (int j = 0 ; j < k ; j++)
    z = z + 1;
```

11.
```
int z = 1;   // Hint: a log figures in the answer.
int m = 1;
while (m<n) {
  z = z + 1;
  m = 2*m;
}
```

12.
```
int z = 1;
for (int i=0 ; i < n-1 ; i++)
  for (int j=i+1 ; j<n ; j++)
    z = z + 1;
```

**Example: Problem 12.** *(Your answers don't have to be this chatty.)*
Since there are nested loops, we work from the inside-out.

The innermost loop: This is

```
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

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

The first iteration 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:

| iteration # | value of $i$ | # of executions $= (n-i-1)$ |
|---|---|---|
| 1 | 0 | n-1 |
| 2 | 1 | n-2 |
| 3 | 2 | n-3 |
| ⋮ | ⋮ | ⋮ |
| n-3 | n-4 | 3 |
| n-2 | n-3 | 2 |
| n-1 | n-2 | 1 |

Therefore, the total number of executions is

$$1+2+3+\cdots+(n-1) \;=\; \frac{1}{2}\cdot(n-1)\cdot n \;\in\; \Theta(n^2).$$

### *C. Proof of version 2 of the limit rule*

Suppose that

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} \;=\; c \tag{1}$$

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

**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 $\ge n_\epsilon$, I will find
>
> $$c-\epsilon \;<\; f(n)/g(n) \;<\; c+\epsilon.$$
>
> So I pick $\epsilon = \frac{1}{2}c$. Then for each $n \ge n_{\frac{c}{2}}$:
>
> $$\frac{c}{2} \;=\; c-\frac{c}{2} \;<\; f(n)/g(n) \;<\; c+\frac{c}{2} \;=\; \frac{3c}{2}.$$
>
> So with $c_\ell = \frac{c}{2}$ and $c_u = \frac{3c}{2}$, 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 $\ge n_\epsilon$, I will find
>
> $$0 \;\le\; f(n)/g(n) \;<\; \epsilon.$$
>
> So I pick $\epsilon = \frac{1}{2}$. Then for each $n \ge n_{\frac{1}{2}}$: $0 \;\le\; f(n)/g(n) \;<\; \frac{1}{2}$. So with $c_u = \frac{1}{2}$, 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
>
> $$\epsilon \ < \ f(n)/g(n).$$
>
> So I pick $\epsilon = 1643$. Then for each $n > n_{1643}$: $1643 \ < \ f(n)/g(n)$. So with $c_\ell = 1643$, we see that $f$ is in $\Omega(g(n))$.

## *D. Challenge Problems*

Challenge 1. Show that $f \in O(g(n)) \iff g(n) \in \Omega(f(n))$.

Challenge 2. Show $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

Challenge 3. Define $o(g(n)) = \{ f \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \}$.
Show $O(g(n)) = \Theta(g(n)) \cup o(g(n))$ and $\Theta(g(n)) \cap o(g(n)) = \emptyset$.

Challenge 4. Prove or disprove: If $f \in O(g(n))$ and $f_\star(n) = 2^{f(n)}$, then $f_\star \in O(2^{g(n)})$.