CSE 101
Introduction to Data Structures and Algorithms
Algorithm Runtime and Efficiency

How should we measure the runtime of an algorithm? A standard approach is to identify some *basic operation* within the algorithm, then count the number of times that operation is performed on particular inputs. Runtime is thus measured not in seconds, but by our integer count of the number of executions of that basic operation. Care must therefore be taken in choosing which operation to count, so as not to obtain a runtime that is artificially low. We thus seek an operation that will be executed as many, or more, times than any other operation in the algorithm.

This is not as difficult as it may sound. For iterative algorithms, we choose an operation within the body of the innermost loop.

Example
A simple algorithm that performs some generic operations.

```
op1
for i=1 to 10
   op2
   for j=1 to 10
      op3
      for k=1 to 10
         op4
```

Examination of this pseudo-code reveals the number of executions of each operation to be those given in the following table.

operation	# executions
`op1`	1
`op2`	10
`op3`	100
`op4`	1000

Assuming that the costs of ops 1-4 are (at least approximately) equal, the better choice of basic operation is `op4`, since doing something 1000 times is worse than doing some (roughly equivalent) thing 1, or 10, or 100 times. Two obvious questions arise though.

(1) What if ops 1-4 are not of equal cost

and

(2) Why not count all operations?

To address (1), we would need to look inside ops 1-4. Consider the case for instance when ops 1-4 are themselves function calls. We would examine the code for each function and attempt to analyze its runtime in terms of some more basic operation(s). Ultimately then, we should deconstruct our pseudo-code to a point where each basic operation is of approximately equal constant cost, then measure runtime in those constant units. Let us assume, for convenience, that we have already done this, so that ops 1-4 might as well be the same operation.

This brings up question (2). The total number of executions of all operations (now assumed to have equivalent cost) in the above example is $1000 + 100 + 10 + 1 = 1{,}111$. We could justifiably take this as the cost, but very few iterative algorithms (that do anything interesting) iterate a fixed number of times, as above. A more realistic situation would be an algorithm taking an integer n as input, and having the number of iterations depend on that input.

Example
Another simple algorithm performing a single operation `op` inside nested loops, whose number of iterations depend on n.

```
op
for i=1 to n
   op
   for j=1 to n
      op
      for k=1 to n
         op
```

The number of executions of our basic operation is now a function $T(n) = n^3 + n^2 + n + 1$ of the input value n. The question is whether we should ignore the lower order terms in this function, and instead consider the runtime to be the simpler function n^3. Note that the point of our analysis is not to determine the runtime for a fixed value of n, but for *all* values of n, and especially for arbitrarily large such values. In other words, we wish to determine *how the runtime grows* with the parameter n. But both functions, the accurate $T(n)$, and the estimate n^3, grow without bound as n becomes arbitrarily large. To compare them, we consider their quotient $T(n)/n^3$, and analyze its behavior in the limit as n approaches ∞. We see that

$$\lim_{n\to\infty} \frac{T(n)}{n^3} = \lim_{n\to\infty} \frac{n^3 + n^2 + n + 1}{n^3} = \lim_{n\to\infty} \left(1 + \frac{1}{n} + \frac{1}{n^2} + \frac{1}{n^3}\right) = 1$$

The limit 1 indicates that the two functions $T(n)$ and n^3, although they both have limit ∞ as $n \to \infty$, approach ∞ *at the same rate*. We therefore deem the runtime, measured as the number of basic operations performed, to be n^3. Equivalently, we can simply refrain from counting those operations not within the body of the innermost loop, making the above algorithm equivalent, as far as runtime is concerned, to

```
for i=1 to n
   for j=1 to n
      for k=1 to n
         op
```

Now suppose we analyze a second algorithm that performs 2 executions of `op` within its innermost loop.

```
for i=1 to n
   for j=1 to n
      for k=1 to n
         op
         op
```

Let us also suppose that these two algorithms (somehow) accomplish the same overall task, so it makes sense to compare them. The runtime of the second algorithm is easily seen to be $2n^3$. It would appear that the first algorithm is twice as efficient as the second, since it produces the same result at half the cost.

But what if we run the second algorithm on a machine that runs twice as fast? If we do that, we will have equalized the two algorithms, and we will have done so for any value of n. Our goal however, is to determine a procedure for analyzing runtime that is independent of the computing device on which the algorithm is running. In other words, we don't want to measure how fast the computer is. We want to measure how fast the algorithm is. You might think that one can always equalize two algorithms by running one on a faster machine, but that is not the case, provided we wish to take into account all possible n. Consider for instance a third algorithm that again performs the same overall task as the preceding two.

```
for i=1 to n
   for j=1 to n
      for k=1 to 10
         op
```

This algorithm runs in time $10n^2$. If we compare this to the runtime of the first, i.e. n^3, we find that the first is better (does fewer basic operations) for small n, and the third is better for large n. Specifically, the two algorithms perform the same number of operations only for n satisfying the equation $n^3 = 10n^2$, namely for $n = 10$. The better algorithm is the one that does fewer operations for the larger number of input sizes. Thus the third algorithm is best, since it wins for all infinitely many $n > 10$. Observe also that there is no way to equalize the two algorithms for infinitely many n. If we run the first algorithm on a machine that is 100 times faster than the device running the third, all that is accomplished is to move the crossover point from 10 to 1000, since the equation is now $n^3/100 = 10n^2$. This tells us that we should ignore the coefficient of the leading term, i.e. replace it by 1.

Our discussion has led us to the following informal procedure for analyzing iterative algorithms.

(1) Choose a basic operation that appears within the body of the innermost loop.
(2) Count the number of executions of this operation as a function of the input size n.
(3) Simplify the function found in (2) by dropping all lower order terms, and replacing the coefficient of the highest order term by 1.

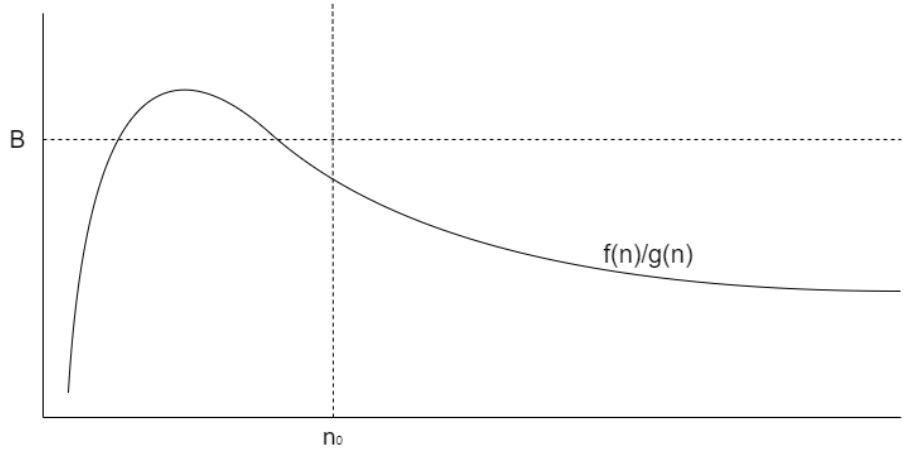
The simplified function found in (3), or rather its rate of growth, is then deemed to be an algorithm's runtime. Thus, if we wish to compare two algorithms that perform the same task, we must compare the growth rate rates of two functions.

Asymptotic Growth of Functions

In what follows, let $f(n)$ and $g(n)$ denote positive functions.

Definition 1

$f(n) = O(g(n))$ if and only if there exist $B > 0$ and $n_0 > 0$ such that $\frac{f(n)}{g(n)} \leq B$ for all $n \geq n_0$. We say $g(n)$ is an *asymptotic upper bound* for $f(n)$. Intuitively, $f(n) = O(g(n))$ means that the growth rate of $f(n)$ is *no greater than* that of $g(n)$.



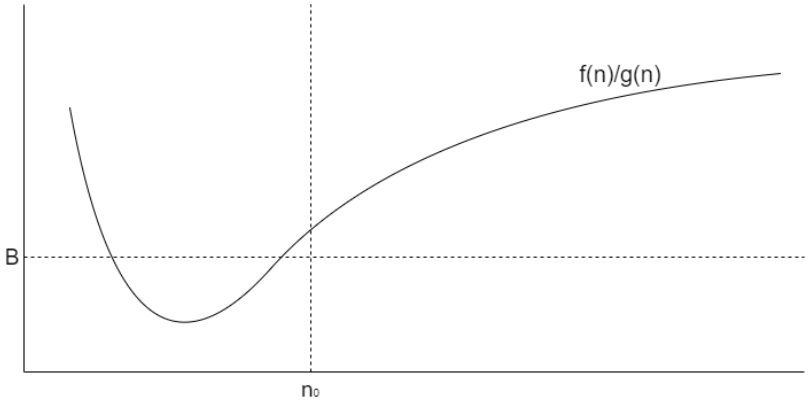
Example

Let $f(n) = 2n + 5$ and $g(n) = n$. In this case $\frac{f(n)}{g(n)} = 2 + \frac{5}{n} \leq 3$ for all $n \geq 5$, as one can easily verify. In this case, the constants in the definition are $B = 3$ and $n_0 = 5$. Observe the B and n_0 are not unique. For instance $B = 6$ and $n_0 = 2$ would also work. Thus $2n + 5 = O(n)$.

A moment's thought suggests that there is nothing special about the coefficients 2 and 5 in this example. As an exercise, verify that $an + b = O(n)$ for any real numbers a and b. (Hint: let $B = |a| + 1$ and let $n_0 = \lceil |b| \rceil$, where $\lceil x \rceil$ is the *ceiling function*, the least integer greater than or equal to x).

Definition 2

$f(n) = \Omega(g(n))$ if and only if there exist $B > 0$ and $n_0 > 0$ such that $B \leq \frac{f(n)}{g(n)}$ for all $n \geq n_0$. We say $g(n)$ is an *asymptotic lower bound* for $f(n)$. We understand $f(n) = \Omega(g(n))$ to mean that the growth rate of $f(n)$ is *no less than* that of $g(n)$.



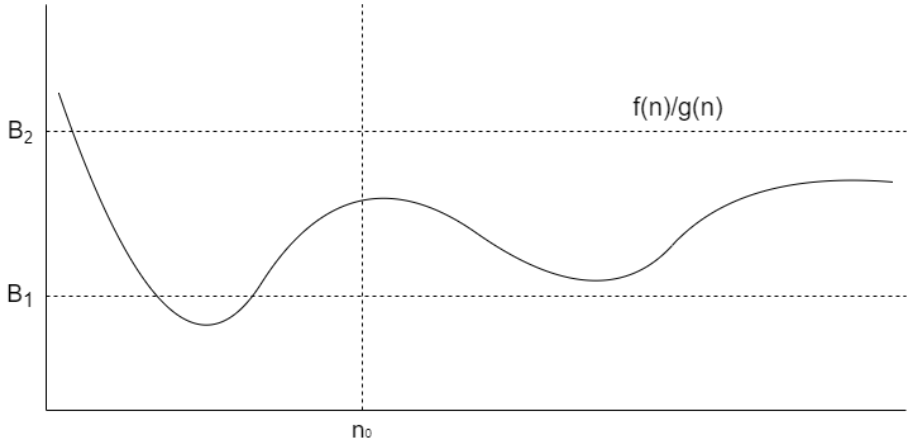
Example

Let $f(n) = 6n^3 + 4n$ and $g(n) = 2n^2$. One checks that $\frac{f(n)}{g(n)} = 3n + \frac{2}{n} \geq 7$ for all $n \geq 2$. Hence, with constants $B = 7$ and $n_0 = 2$, we have $6n^3 + 4n = \Omega(2n^2)$.

Observe that $\frac{f(n)}{g(n)} \leq B_1$ iff $\frac{g(n)}{f(n)} \geq B_2$, where $B_2 = \frac{1}{B_1}$. It follows that $f(n) = O\big(g(n)\big)$ iff $g(n) = \Omega\big(f(n)\big)$. Applying this fact to the last two examples we obtain two more: $n = \Omega(2n + 5)$ and $2n^2 = O(6n^3 + 4n)$.

Definition 3

$f(n) = \Theta(g(n))$ if and only if there exist positive B_1, B_2 and n_0 such that $B_1 \leq \frac{f(n)}{g(n)} \leq B_2$ for all $n \geq n_0$. In this case, we say $g(n)$ is a *tight asymptotic bound* for $f(n)$.



Obviously $f(n) = \Theta\big(g(n)\big)$ iff both $f(n) = O(g(n))$ and $f(n) = \Omega\big(g(n)\big)$. We thus interpret $f(n) = \Theta\big(g(n)\big)$ to mean that $f(n)$ and $g(n)$ *grow at the same rate*. Coupled with our previous observation that $f(n) = O(g(n))$ iff $g(n) = \Omega\big(f(n)\big)$, we see that $f(n) = \Theta\big(g(n)\big)$ iff $g(n) = \Theta\big(f(n)\big)$.

Example

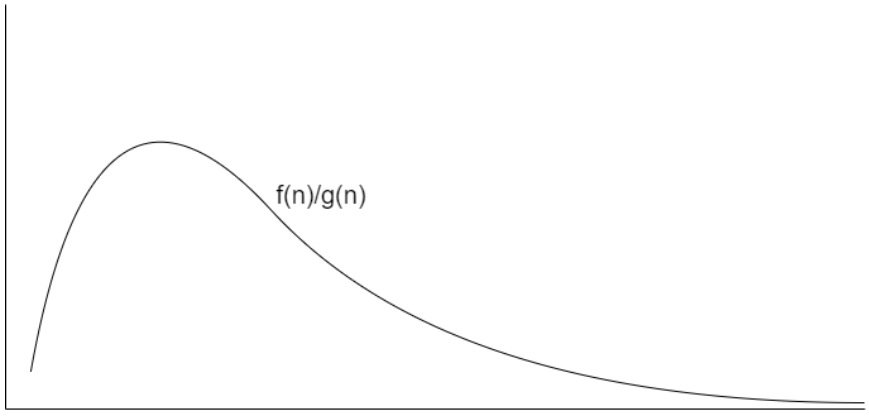
Let $f(n) = 5n^2 + 11n - 24$ and $g(n) = n^2$. In this case we take $B_1 = 4$, $B_2 = 6$ and $n_0 = 8$. The reader should check that

$$4 \leq \frac{5n^2 + 11n - 24}{n^2} \leq 6,$$

for all $n \geq 8$. Hence $5n^2 + 11n - 24 = \Theta(n^2)$. We leave it as an exercise to show for any real numbers a, b, c with $a > 0$, that $an^2 + bn + c = \Theta(n^2)$.

Definition 4

$f(n) = o\big(g(n)\big)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. We say $g(n)$ is a *strict asymptotic upper bound* for $f(n)$. If both $f(n)$ and $g(n)$ both approach ∞ as $n \to \infty$, $f(n)$ grows so much slower than $g(n)$, that the ratio approches 0. Thus we interpret $f(n) = o\big(g(n)\big)$ to mean that $f(n)$ *grows strictly slower than* $g(n)$.



Example

Let $f(n) = \ln(n)$ and $g(n) = n$. L'Hopital's rule yeilds $\lim_{n\to\infty} \frac{\ln n}{n} = \lim_{n\to\infty} \frac{1/n}{1} = \lim_{n\to\infty} \frac{1}{n} = 0$, showing that $\ln(n) = o(n)$. Observe that the calculation is basically the same for $f(n) = \log_b(n)$ and $g(n) = n^k$, with any $b > 1$ and $k > 0$ (and $\lceil k \rceil$ applications of L'Hopital's rule.) Thus $\log_b(n) = o(n^k)$.

Example

Let $f(n) = n^k$ (where $k \geq 0$ is an integer) and $g(n) = e^n$. Then after k applications of L'Hopital's rule

$$\lim_{n\to\infty} \frac{n^k}{e^n} = \lim_{n\to\infty} \frac{kn^{k-1}}{e^n} = \cdots = \lim_{n\to\infty} \frac{k(k-1)(k-2)\cdots 1}{e^n} = 0,$$

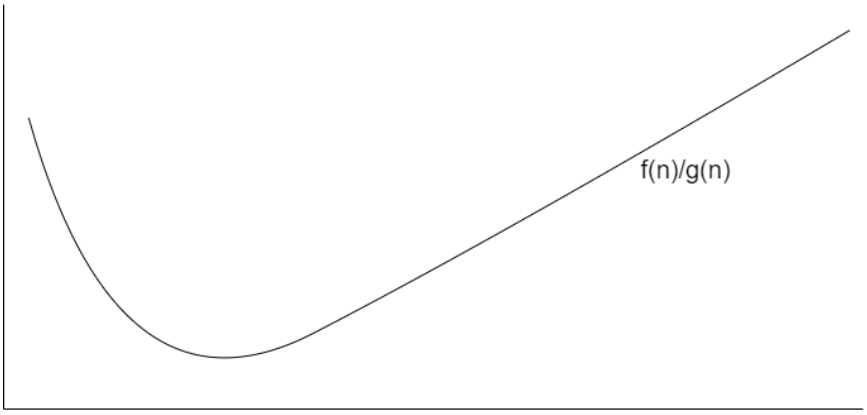
whence $n^k = o(e^n)$. Observe that if k is not an integer, then $\lceil k \rceil$ applications will suffice. As an exercise, check that $n^k = o(b^n)$ for any $b > 1$.

Example

Let $f(n) = n^\alpha$ and $g(n) = n^\beta$ where $0 \leq \alpha < \beta$ are real numbers. Then $\lim_{n\to\infty} \frac{n^\alpha}{n^\beta} = \lim_{n\to\infty} \frac{1}{n^{\beta-\alpha}} = 0$, since $\beta - \alpha > 0$, and hence $n^\alpha = o(n^\beta)$.

Definition 5

$f(n) = \omega(g(n))$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. We say $g(n)$ is a *strict asymptotic lower bound* for $f(n)$. If $f(n)$ and $g(n)$ both approach ∞ as $n \to \infty$, then $f(n)$ grows so much faster than $g(n)$, that the ratio approaches ∞. Accordingly, we interpret $f(n) = \omega(g(n))$ to mean $f(n)$ *grows strictly faster than* $g(n)$.



Observe that the relationship between o and ω is similar to that between O and Ω, since the limit of a reciprocal is the reciprocal of the limit (considering $1/\infty = 0$ and $1/0 = \infty$.) Thus

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \text{ if and only if } \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty,$$

and therefore

$$f(n) = o(g(n)) \text{ if and only if } g(n) = \omega(f(n)).$$

The preceding three examples now give rise to three more examples pertaining to ω: $n^k = \omega(\log_b(n))$, $b^n = \omega(n^k)$ and $n^\beta = \omega(n^\alpha)$.

Example

Let $f(n) = b^n$ and $g(n) = a^n$, where $1 \le a < b$. Then $\frac{b^n}{a^n} = \left(\frac{b}{a}\right)^n \to \infty$ since $\frac{b}{a} > 1$, and $b^n = \omega(a^n)$. The comments in the paragraph above now imply $a^n = o(b^n)$.

Analogy with Numbers

The comparision of positive functions $f(n)$ and $g(n)$ as to their asymptotic growth rates is analogous to the comparison of real numbers x and y as to their size.

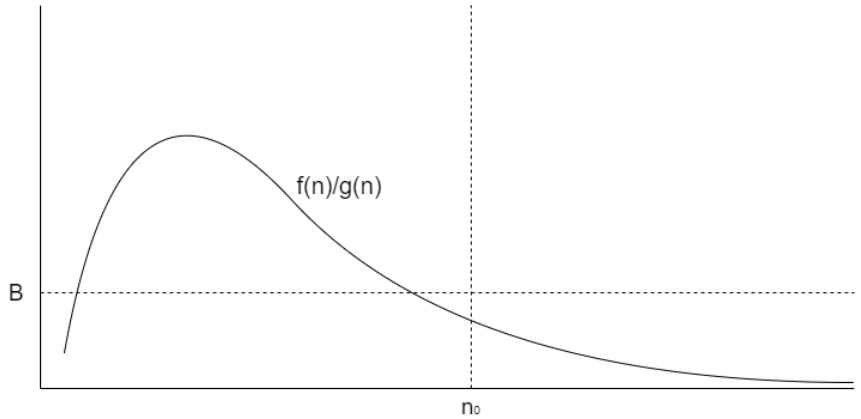
$f(n) = O\big(g(n)\big)$	*is analogous to*	$x \le y$
$f(n) = \Omega\big(g(n)\big)$	$\leftrightarrow$	$x \ge y$
$f(n) = \Theta\big(g(n)\big)$	$\leftrightarrow$	$x = y$
$f(n) = o\big(g(n)\big)$	$\leftrightarrow$	$x < y$
$f(n) = \omega\big(g(n)\big)$	$\leftrightarrow$	$x > y$

The analogy is quite deep. For instance, the transitive law for $\le$ ($x \le y$ and $y \le z$ implies $x \le z$) corresponds to the transitive law for O. Indeed

$$\frac{f(n)}{g(n)} \le B_1 \text{ and } \frac{g(n)}{h(n)} \le B_2 \text{ implies } \frac{f(n)}{h(n)} = \frac{f(n)}{g(n)} \cdot \frac{g(n)}{h(n)} \le B_1 \cdot B_2,$$

and therefore $f(n) = O\big(g(n)\big)$ and $g(n) = O\big(h(n)\big)$ implies $f(n) = O\big(h(n)\big)$. As an exercise, the reader should prove that Ω, Θ, o and ω also satisfy their respective transitive laws.

Another obvious fact relates $<$ to $\le$, namely $x < y$ implies $x \le y$. Correspondingly $f(n) = o\big(g(n)\big)$ implies $f(n) = O\big(g(n)\big)$. To see this, observe that if $f(n)/g(n) \to 0$, then any $B > 0$ satisfies $f(n)/g(n) \le B$ for sufficiently large n.



We leave it as an exercise for the reader to show that $f(n) = \omega\big(g(n)\big)$ implies $f(n) = \Omega\big(g(n)\big)$.

Similarly, it is obvious that $x < y$ implies $x \not\geq y$. The analogous statement is that $f(n) = o\big(g(n)\big)$ implies $f(n) \neq \Omega\big(g(n)\big)$. To see this, note that $f(n)/g(n) \to 0$ implies that $f(n)/g(n)$ has no positive lower bound. Show as an exercise that $f(n) = \omega\big(g(n)\big)$ implies $f(n) \neq \mathrm{O}\big(g(n)\big)$.

Although the analogy is good, it is not perfect. The set of real numbers satisfy the trichotomy law: given any $x, y \in \mathbb{R}$, either $x < y$, or $x > y$, or $x = y$. Roughly stated, this says that any two real numbers are comparable. However, there are asymptotic growth rates that are not comparable in any way, i.e. one can find functions $f(n)$, $g(n)$ for which none of the five relations O, Ω, Θ, o or ω hold. We leave it as a (somethat difficult) exercise to find such examples. Some of the following exercises have already been proved above, but the reader should prove them now directly.

Exercises

(1) For any $\alpha, \beta \in \mathbb{R}$: $n^{\alpha} = \begin{cases} o\big(n^{\beta}\big) & (\text{if } \alpha < \beta) \\ \Theta\big(n^{\beta}\big) & (\text{if } \alpha = \beta) \\ \omega\big(n^{\beta}\big) & (\text{if } \alpha > \beta) \end{cases}$

(2) For any $a, b \in \mathbb{R}$ with $a > 1, b > 1$: $a^n = \begin{cases} o(b^n) & (\text{if } a < b) \\ \Theta(b^n) & (\text{if } a = b) \\ \omega(b^n) & (\text{if } a > b) \end{cases}$

(3) For any functions $f(n)$, $g(n)$ and $c > 0$: $\begin{cases} cf(n) = O\big(f(n)\big) \\ cf(n) = \Omega\big(f(n)\big) \\ cf(n) = \Theta\big(f(n)\big) \\ f(n) = o\big(g(n)\big) \implies cf(n) = o\big(g(n)\big) \\ f(n) = \omega\big(g(n)\big) \implies cf(n) = \omega\big(g(n)\big) \end{cases}$

(4) For any $a, b \in \mathbb{R}$ with $a > 1, b > 1$: $\log_b(n) = \Theta(\log_a(n))$

(5) Let $L = \lim\limits_{n\to\infty} \frac{f(n)}{g(n)}$ (if the limit exists). Show that

a. $0 \leq L < \infty \implies f(n) = O\big(g(n)\big)$
b. $0 < L < \infty \implies f(n) = \Theta\big(g(n)\big)$
c. $0 < L \leq \infty \implies f(n) = \Omega\big(g(n)\big)$
d. $L = 0 \iff f(n) = o\big(g(n)\big)$
e. $L = \infty \iff f(n) = \omega\big(g(n)\big)$

Note: the converses ($\Longleftarrow$) to (a), (b) and (c) are false.

(6) For any function $f(n)$: $f(n) + o\big(f(n)\big) = \Theta\big(f(n)\big)$

(7) Rank the following functions from lowest to highest asymptotic growth rate.

n^2, $\ln(n)$, $(\ln(n))^2$, $\ln(n^2)$, $n\ln(n)$, $\sqrt{n}$, $n\sqrt{n}$, $\ln\big(ln\big(\sqrt{n}\big)\big)$, $2^{\ln(n)}$, 2^n, 2^{3^n}, 3^{2^n}