Chapter 5

Algorithm Analysis and Asymptotic Notation

5.1 Correctness, running time of programs

So far we have been proving statements about databases, mathematics and arithmetic, or sequences of numbers. Though these types of statements are common in computer science, you'll probably encounter algorithms most of the time. Often we want to reason about algorithms and even prove things about them. Wouldn't it be nice to be able to *prove* that your program is correct? Especially if you're programming a heart monitor or a NASA spacecraft?

In this chapter we'll introduce a number of tools for dealing with computer algorithms, formalizing their expression, and techniques for analyzing properties of algorithms, so that we can prove correctness or prove bounds on the resources that are required.

5.2 Binary (base 2) notation

Let's first think about numbers. In our everyday life, we write numbers in decimal (base 10) notation (although I heard of one kid who learned to use the fingers of her left hand to count from 0 to 31 in base 2). In decimal, the sequence of digits 20395 represents (parsing from the right):

$$5 + 9(10) + 3(100) + 0(1000) + 2(10000) =$$
$$5(10^0) + 9(10^1) + 3(10^2) + 0(10^3) + 2(10^4)$$

Each position represents a power of 10, and 10 is called the BASE. Each position has a digit from $[0, 9]$ representing how many of that power to add. Why do we use 10? Perhaps due to having 10 fingers (however, humans at various times have used base 60, base 20, and mixed base 20,18 (Mayans)). In the last case there were $(105)_{20,18}$ days in the year. Any integer with absolute value greater than 1 will work (so experiment with base -2).

Consider using 2 as the base for our notation. What digits should we use?[1] We don't need digits 2 or higher, since they are expressed by choosing a different position for our digits (just as in base 10, where there is no single digit for numbers 10 and greater).
Here are some examples of binary numbers:

$$(10011)_2$$

represents

$$1(2^0) + 1(2^1) + 0(2^2) + 0(2^3) + 1(2^4) = (19)_{10}$$

We can extend the idea, and imitate the decimal point (with a "binary point"?) from base 10:

$$(1011.101)_2 = 19\frac{5}{8}$$

How did we do that?[2] Here are some questions:

- How do you multiply two base 10 numbers?[3] Work out 37×43.
- How do you multiply two binary numbers?[4]
- What does "right shifting" (eliminating the right-most digit) do in base 10?[5]
- What does "right shifting" do in binary?[6]
- What does the rightmost digit tell us in base 10? In binary?

Convert some numbers from decimal to binary notation. Try 57. We'd like to represent 57 by adding either 0 or 1 of each power of 2 that is no greater than 57. So $57 = 32 + 16 + 8 + 1 = (111001)_2$. We can also fill in the binary digits, systematically, from the bottom up, using the % operator from Python (the remainder after division operator, at least for positive arguments):

$$\begin{aligned}
57 \,\%\, 2 = 1 &\quad \text{so} \quad (?????1)_2 \\
(57-1)/2 = 28 \,\%\, 2 = 0 &\quad \text{so} \quad (????01)_2 \\
28/2 = 14 \,\%\, 2 = 0 &\quad \text{so} \quad (???001)_2 \\
14/2 = 7 \,\%\, 2 = 1 &\quad \text{so} \quad (??1001)_2 \\
(7-1)/2 = 3 \,\%\, 2 = 1 &\quad \text{so} \quad (?11001)_2 \\
(3-1)/2 = 1 \,\%\, 2 = 1 &\quad \text{so} \quad (111001)_2
\end{aligned}$$

Addition in binary is the same as (only different from...) addition in decimal. Just remember that $(1)_2 + (1)_2 = (10)_2$. If we add two binary numbers, this tells us when to "carry" 1:

$$\begin{array}{rr} & 1011 \\ + & 1011 \\ \hline & 10110 \end{array}$$

5.3 log$_2$

How many 5-digit binary numbers are there (including those with leading 0s)? These numbers run from $(00000)_2$ through $(11111)_2$, or 0 through 31 in decimal — 32 numbers. Another way to count them is to consider that there are two choices for each digit, hence 2^5 strings of digits. If we add one more digit we get twice as many numbers. Every digit doubles the range of numbers, so there are two 1-digit binary numbers (0 and 1), four 2-digit binary numbers (0 through 3), 8 3-digit binary numbers (0 through 7), and so on.

Reverse the question: how many digits are required to represent a given number. In other words, what is the smallest integer power of 2 needed to exceed a given number? $\log_2 x$ is the power of 2 that gives $2^{\log_2 x} = x$. You can think of it as how many times you must multiply 1 by 2 to get x, or roughly the number of digits in the binary representation of x. (The precise number of digits needed is $\lceil \log_2(x+1) \rceil$ — which happens to be equal to $\lfloor (\log_2 x) + 1 \rfloor$ for all positive values of x).

5.4 Loop invariant for base 2 multiplication

Integers are naturally represented on a computer in binary, since a gate can be in either an on or off (1 or 0) position. It is very easy to multiply or divide by 2, since all we need to do is perform a left or right shift (an easy hardware operation). Similarly, it is also very easy to determine whether an integer is even or odd. Putting these together, we can write a multiplication algorithm that uses these fast operations:

```
def mult(m,n):
    """ Multiply integers m and n. """
    # Precondition: m >= 0
    x = m
    y = n
    z = 0

    # loop invariant: z = mn - xy
    while x != 0:
        if x % 2 == 1:  z = z + y  # x is odd
        x = x >> 1  # x = x / 2 (right shift)
        y = y << 1  # y = y * 2 (left shift)

    # post condition: z = mn
    return z
```

After reading this algorithm, there is no reason you should believe it actually multiplies two integers: we'll need to prove it to you. Let's consider the precondition first. So long as m is a positive natural number, and n is an integer, the program claims to work. The postcondition states that z, the value that is returned, is equal to the product of m and n (that would be nice, but we're not convinced).

Let's look at the stated loop invariant. A loop invariant is a relationship between the variables that is always true at the start and at the end of a loop iteration (we'll need to prove this). It's sufficient to verify that the invariant is true at the start of the first iteration, and verify that if the invariant is true at the start of any iteration, it must be true at the end of the iteration. Before we start the loop, we set $x = m$, $y = n$ and $z = 0$, so it is clear that $z = mn - xy = mn - mn = 0$. Now we need to show that if $z = mn - xy$ before executing the body of the loop, and $x \neq 0$, then after executing the loop body, $z = mn - xy$ is still true (can you write this statement formally?). Here's a sketch of a proof:

Assume $x_i, y_i, z_i, x_{i+1}, y_{i+1}, z_{i+1}, m, n \in \mathbb{Z}$, where x_i represents the value of variable x at the beginning of the ith iteration of the loop, and similarly for the other variables and subscripts. (Note that there is no need to subscript m, n, since they aren't changed by the loop.)

Assume $z_i = mn - x_i y_i$.

Case 1: Assume x_i odd.

Then $z_{i+1} = z_i + y_i$, $x_{i+1} = (x_i - 1)/2$, and $y_{i+1} = 2y_i$.

So

$$\begin{aligned} mn - x_{i+1}y_{i+1} &= mn - (x_i - 1)/2 \cdot 2y_i && \text{(since } x_i \text{ is odd)} \\ &= mn - x_i y_i + y_i \\ &= z_i + y_i \\ &= z_{i+1} \end{aligned}$$

Case 2: Assume x_i even.

Then $z_{i+1} = z_i$, $x_{i+1} = x_i/2$, and $y_{i+1} = 2y_i$.

So

$$\begin{aligned} mn - x_{i+1}y_{i+1} &= mn - x_i/2 \cdot 2y_i \\ &= mn - x_i y_i \\ &= z_i \\ &= z_{i+1} \end{aligned}$$

Since x_i is either even or odd, in all cases $mn - x_{i+1}y_{i+1} = z_{i+1}$

Thus $mn - x_i y_i = z_i \Rightarrow mn - x_{i+1}y_{i+1} = z_{i+1}$.
Since $x_i, x_{i+1}, y_i, y_{i+1}, z_i, z_{i+1}, m, n$ are arbitrary elements,
$\forall x_i, x_{i+1}, y_i, y_{i+1}, z_i, z_{i+1}, m, n \in \mathbb{Z}, mn - x_i y_i = z_i \Rightarrow mn - x_{i+1}y_{i+1} = z_{i+1}$.

We should probably verify the postcondition to fully convince ourselves of the correctness of this algorithm. We've shown the loop invariant holds, so let's see what we can conclude when the loop terminates (*i.e.*, when $x = 0$). By the loop invariant, $z = mn - xy = mn - 0 = mn$, so we know we must get the right answer (assuming the loop eventually terminates).

We should now be fairly convinced that this algorithm is in fact correct. One might now wonder, how many iterations of the loop are completed before the answer is returned?

Also, why is it necessary for $m \geqslant 0$? What happens if it isn't?

5.5 Running time of programs

For any program P and any input x, let $t_P(x)$ denote the number of "steps" P takes on input x. We need to specify what we mean by a "step." A "step" typically corresponds to machine instructions being executed, or some indication of time or resources expended.
Consider the following (somewhat arbitrary) accounting for common program steps:

method call: 1 step + steps to evaluate each argument + steps to execute the method.

return statement: 1 step + steps to evaluate return value.

if statement: 1 step + steps to evaluate condition.

assignment statement: 1 step + steps to evaluate each side.

arithmetic, comparison, boolean operators: 1 step + steps to evaluate each operand.

array access: 1 step + steps to evaluate index.

member access: 2 steps.

constant, variable evaluation: 1 step.

Notice that none of these "steps" (except for method calls) depend on the size of the input (sometimes denoted with the symbol n). The smallest and largest steps above differ from each other by a constant of about 5, so we can make the additional simplifying assumption that they all have the same cost — 1.

5.6 Linear search

Let's use linear search as an example.

```
def LS(A,x):
    """ Return an index i such that x == L[i].  Otherwise, return -1. """
    i = 0                # (line 1)
    while i < len(A):    # (line 2)
        if A[i] == x:    # (line 3)
            return i     # (line 4)
        i = i + 1        # (line 5)
    return -1            # (line 6)
```

Let's trace a function call, LS([2,4,6,8],4):

Line 1: 1 step (i=0)

Line 2: 1 step (`0 < 4`)

Line 3: 1 step (`A[0] == 4`)

Line 5: 1 step (`i = 1`)

Line 2: 1 step (`1 < 4`)

Line 3: 1 step (`A[1] == 4`)

Line 4: 1 (`return 1`)

So $t_{LS}([2,4,6,8],4) = 7$. Notice that if the first index where x is found is j, then $t_{LS}(A,x)$ will count lines 2, 3, and 5 once for each index from 0 to $j-1$ (j indices), and then count lines 2, 3, 4 for index j, and so $t_{LS}(A,x)$ will be $1+3j+3$.

If x does not appear in A, then $t_{LS}(A,x) = 1 + 3\,\mathrm{len}(A) + 2$, because line 1 executes once, lines 2, 3, and 5 executes once for each index from 0 to $\mathrm{len}(A)-1$, and then lines 2 and 6 execute.

We want a measure that depends on the size of the input, not the particular input. There are three standard ways. Let P be a program, and let I be the set of all inputs for P. Then:

Average-case complexity: the weighted average over all possible inputs of size n.

In general

$$A_P(n) = \sum_{x \text{ of size } n} t_P(x) \cdot p(x)$$

where $p(x)$ is the probability that input x is encountered.

Assuming all the inputs are equally likely, this “simplifies” to

$$A_P(n) = \frac{\sum_{x \text{ of size } n} t_P(x)}{\text{number of inputs of size } n}$$

(Difficult to compute.)

Best-case complexity: $\min(t_P(x))$, where x is an input of size n.

In other words, $B_P(n) = \min\{t_P(x) \mid x \in I \wedge \mathsf{size}(x) = n\}$.

(Mostly useless.)

Worst-case complexity: $\max(t_P(x))$, where x is an input of size n.

In other words, $W_P(n) = \max\{t_P(x) \mid x \in I \wedge \mathsf{size}(x) = n\}$.

(Relatively easy to compute, and gives a performance guarantee.)

What is meant by “input size”? This depends on the algorithm. For linear search, the number of elements in the array is a reasonable parameter. Technically (in csc 463 h, for example), the size is the number of bits required to represent the input in binary. In practice we use the number of elements of input (length of array, number of nodes in a tree, *etc.*)

5.7 Run time and constant factors

When calculating the running time of a program, we may know how many basic “steps” it takes as a function of input size, but we may not know how long each step takes on a particular computer. We would like to estimate the overall running time of an algorithm while ignoring constant factors (like how fast the CPU is). So, for example, if we have 3 machines, where operations take 3μs, 8μs and 0.5μs, the three functions measuring the amount of time required, $t(n) = 3n^2$, $t(n) = 8n^2$, and $t(n) = n^2/2$ are considered the same, ignoring ("to within") constant factors (the time required always grows according to a quadratic function in terms of the size of the input n).

To view this another way, think back to the linear search example in the previous section. The worst-case running time for that algorithm was given by the function

$$W(n) = 3n + 3$$

But what exactly does the constant "3" in front of the "n" represent? Or the additive "$+3$"? Neither value corresponds to any intrinsic property of the algorithm itself; rather, the values are consequences of some arbitrary choices on our part, namely, how many "steps" to count for certain Python statements. Someone counting differently (*e.g.*, counting more than 1 step for statements that access list elements by index) would arrive at a different expression for the worst-case running time. Would their answer be "more" or "less" correct than ours? Neither: both answers are just as imprecise as one another! This is why we want to come up with a tool that allows us to work with functions while ignoring constant multipliers.

The nice thing is that this means that lower order terms can be ignored as well! So $f(n) = 3n^2$ and $g(n) = 3n^2 + 2$ are considered "the same," as are $h(n) = 3n^2 + 2n$ and $j(n) = 5n^2$. Notice that

$$\forall n \in \mathbb{N}, n \geqslant 1 \Rightarrow f(n) \leqslant g(n) \leqslant h(n) \leqslant j(n)$$

but there's always a constant factor that can reverse any of these inequalities.

Really what we want to measure is the growth rate of functions (and in computer science, the growth rate of functions that bound the running time of algorithms). You might be familiar with binary search and linear search (two algorithms for searching for a value in a sorted array). Suppose one computer runs binary search and one computer runs linear search. Which computer will give an answer first, assuming the two computers run at roughly the same CPU speed? What if one computer is much faster (in terms of CPU speed) than the other, does it affect your answer? What if the array is really, really big?

How large is "sufficiently large?"

Is binary search a better algorithm than linear search?[7] It depends on the size of the input. For example, suppose you established that linear search has complexity $L(n) = 3n$ and binary search has complexity $B(n) = 9 \log_2 n$. For the first few n, $L(n)$ is smaller than $B(n)$. However, certainly for $n > 10$, $B(n)$ is smaller, indicating less "work" for binary search.

When we say "large enough" n, we mean we are discussing the ASYMPTOTIC behaviour of the complexity function (*i.e.*, the behaviour as n grows toward infinity), and we are prepared to ignore the behaviour near the origin.

5.8 Asymptotic notation: Making Big-O precise

We define $\mathbb{R}^{\geqslant 0}$ as the set of nonnegative real numbers, and define $\mathbb{R}^+$ as the set of positive real numbers. Here is a precise definition of "The set of functions that are eventually no more than f, to within a constant factor":

DEFINITION: For any function $f : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$ (*i.e.*, any function mapping naturals to nonnegative reals), let

$$\mathcal{O}(f) = \{g : \mathbb{N} \to \mathbb{R}^{\geqslant 0} \mid \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow g(n) \leqslant cf(n)\}.$$

Saying $g \in \mathcal{O}(f)$ says that "g grows no faster than f" (or equivalently, "f is an upper bound for g"), so long as we modify our understanding of "growing no faster" and being an "upper bound" with the practice of ignoring constant factors. Now we can prove some theorems.

Suppose $g(n) = 3n^2 + 2$ and $f(n) = n^2$. Then $g \in \mathcal{O}(f)$. To be more precise, we need to prove the statement $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 3n^2 + 2 \leqslant cn^2$. It's enough to find some c and B that "work" in order to prove the theorem.

Finding c means finding a factor that will scale n^2 up to the size of $3n^2 + 2$. Setting $c = 3$ almost works, but there's that annoying additional term 2. Certainly $3n^2 + 2 < 4n^2$ so long as $n \geqslant 2$, since $n \geqslant 2 \Rightarrow n^2 > 2$. So pick $c = 4$ and $B = 2$ (other values also work, but we like the ones we thought of first). Now concoct a proof of

$$\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 3n^2 + 2 \leqslant cn^2.$$

```
Let $c' = 4$ and $B' = 2$.
Then $c' \in \mathbb{R}^+$ and $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$.    # direct proof for an arbitrary natural number
    Then $n^2 \geqslant {B'}^2 = 4$.    # squaring is monotonic on natural numbers
    Then $n^2 \geqslant 2$.
    Then $3n^2 + n^2 \geqslant 3n^2 + 2$.    # adding $3n^2$ to both sides of the inequality
    Then $3n^2 + 2 \leqslant 4n^2 = c'n^2$    # re-write
Then $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow 3n^2 + 2 \leqslant c'n^2$    # introduce $\forall$ and $\Rightarrow$
Then $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 3n^2 + 2 \leqslant cn^2$.    # introduce $\exists$ (twice)
```

So, by definition, $g \in \mathcal{O}(f)$.

A more complex example

Let's prove that $2n^3 - 5n^4 + 7n^6$ is in $\mathcal{O}(n^2 - 4n^5 + 6n^8)$. We begin with:

```
Let $c' =$ ____. Then $c' \in \mathbb{R}^+$.
Let $B' =$ ____. Then $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$.    # arbitrary natural number and antecedent
    Then $2n^3 - 5n^4 + 7n^6 \leqslant \ldots \leqslant c'(n^2 - 4n^5 + 6n^8)$.
Then $\forall n \in \mathbb{N}, n \geqslant Bi' \Rightarrow 2n^3 - 5n^4 + 7n^6 \leqslant c'(n^2 - 4n^5 + 6n^8)$.    # introduce $\Rightarrow$ and $\forall$
Hence, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 2n^3 - 5n^4 + 7n^6 \leqslant c(n^2 - 4n^5 + 6n^8)$.    # introduce $\exists$
```

To fill in the $\cdots$ we try to form a chain of inequalities, working from both ends, simplifying the expressions:

$$\begin{aligned}
2n^3 - 5n^4 + 7n^6 &\leqslant 2n^3 + 7n^6 && \text{(drop $-5n^4$ because it doesn't help us in an important way)}\\
&\leqslant 2n^6 + 7n^6 && \text{(increase n^3 to n^6 because we have to handle n^6 anyway)}\\
&= 9n^6\\
&\leqslant 9n^8 && \text{(simpler to compare)}\\
&= 2(9/2)n^8 && \text{(get as close to form of the simplified end result: now choose $c' = 9/2$)}\\
&= 2cn^8\\
&= c'(-4n^8 + 6n^8) && \text{(reading bottom up: decrease $-4n^5$ to $-4n^8$ because we have to handle n^8 anyway)}\\
&\leqslant c'(-4n^5 + 6n^8) && \text{(reading bottom up: drop n^2 because it doesn't help us in an important way)}\\
&\leqslant c'(n^2 - 4n^5 + 6n^8)
\end{aligned}$$

We never needed to restrict n in any way beyond $n \in \mathbb{N}$ (which includes $n \geqslant 0$), so we can fill in $c' = 9/2$, $B' = 0$, and complete the proof.

Let's use this approach to prove $n^4 \notin \mathcal{O}(3n^2)$. More precisely, we have to prove the negation of the statement $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in N, n \geqslant B \Rightarrow n^4 \leqslant c3n^2$.

Assume $c \in \mathbb{R}^+$ and $B \in \mathbb{N}$. # arbitrary positive real number and natural number
  Let $n_0 =$ ____.
  $\vdots$
  So $n_0 \in \mathbb{N}$.
  $\vdots$
  So $n_0 \geqslant B$.
  $\vdots$
  So $n_0^4 > c3n_0^2$.
Then $\forall c \in \mathbb{R}^+, \forall B \in \mathbb{N}, \exists n \in \mathbb{N}, n \geqslant B \wedge n^4 > c3n^2$.

Here's our chain of inequalities (the third $\vdots$):

$$\begin{aligned} \text{And } n_0^4 &\geqslant n_0^3 && \text{(don't need full power of } n_0^4\text{)} \\ &= n_0 \cdot n_0^2 && \text{(make form as close as possible)} \\ &> c \cdot 3n_0^2 && \text{(if we make } n_0 > 3c \text{ and } n_0 > 0\text{)} \end{aligned}$$

Now pick $n_0 = \max(B, \lceil 3c+1 \rceil)$.

The first $\vdots$ is:

Since $c > 0$, $3c+1 > 0$, so $\lceil 3c+1 \rceil \in \mathbb{N}$.
Since $B \in \mathbb{N}$, $\max(B, \lceil 3c+1 \rceil) \in \mathbb{N}$.

The second $\vdots$ is:

$\max(B, \lceil 3c+1 \rceil) \geqslant B$.

We also note just before the chain of inequalities:

$n_0 = \max(B, \lceil 3c+1 \rceil) \geqslant \lceil 3c+1 \rceil \geqslant 3c+1 > 3c$.

Some points to note are:

- Don't "solve" for n until you've made the form of the two sides as close as possible.
- You're not exactly solving for n: you are finding a condition of the form $n >$ __ that makes the desired inequality true. You might find yourself using the "max" function a lot.

OTHER BOUNDS

In analogy with $\mathcal{O}(f)$, consider two other definitions:

DEFINITION: For any function $f : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, let

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^{\geqslant 0} \mid \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow g(n) \geqslant cf(n)\}.$$

To say "$g \in \Omega(f)$" expresses the concept that "g grows at least as fast as f" (f is a lower bound on g).

DEFINITION: For any function $f : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, let

$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}^{\geqslant 0} \mid \exists c_1 \in \mathbb{R}^+, \exists c_2 \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)\}.$$

To say "$g \in \Theta(f)$" expresses the concept that "g grows at the same rate as f" (f is a tight bound for g, or f is both an upper bound and a lower bound on g).

Induction interlude

Suppose $P(n)$ is some predicate of the natural numbers, and:

$$(*) \quad P(0) \wedge (\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)).$$

You should certainly be able to show that $(*)$ implies $P(0)$, $P(1)$, $P(2)$, in fact $P(n)$ where n is any natural number you have the patience to follow the chain of results to obtain. In fact, we feel that we can "turn the crank" enough times to show that $(*)$ implies $P(n)$ for any natural number n. This is called the Principle of Simple Induction (PSI). It isn't proved, it is an axiom that we assume to be true.
Here's an application of the PSI that will be useful for some big-Oh problems.

$P(n)$: $2^n \geqslant 2n$.

I'd like to prove that $\forall n, P(n)$, using the PSI. Here's what I do:

Prove $P(0)$: $P(0)$ states that $2^0 = 1 \geqslant 2(0) = 0$, which is true.

```
Prove $\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)$:
    Assume $n \in \mathbb{N}$.    # arbitrary natural number
        Assume $P(n)$, that is $2^n \geqslant 2n$.    # antecedent
            Then $n = 0 \vee n > 0$.    # natural numbers are non-negative
            Case 1 (assume $n = 0$): Then $2^{n+1} = 2^1 = 2 \geqslant 2(n+1) = 2$.
            Case 2 (assume $n > 0$): Then $n \geqslant 1$.    # $n$ is an integer greater than 0
                Then $2^n \geqslant 2$.    # since $n \geqslant 1$, and $2^n$ is monotone increasing
                Then $2^{n+1} = 2^n + 2^n \geqslant 2n + 2 = 2(n+1)$.    # by previous line and IH $P(n)$
            Then $2^{n+1} \geqslant 2(n+1)$, which is $P(n+1)$.    # true in both possible cases
        Then $P(n) \Rightarrow P(n+1)$.    # introduce $\Rightarrow$
    Then $\forall n \in \mathbb{N}, P(n) \Rightarrow P(n+1)$.    # introduce $\forall$
```

I now conclude, by the PSI, $\forall n \in \mathbb{N}, P(n)$, that is $2^n \geqslant 2n$.

Here's a big-Oh problem where we can use $P(n)$. Let $g(n) = 2^n$ and $f(n) = n$. I want to show that $g \notin \mathcal{O}(f)$.

```
Assume $c \in \mathbb{R}^+$, assume $B \in \mathbb{N}$.    # arbitrary values
    Let $k = \max(0, \lceil \log_2(c) \rceil) + 1 + B$, and let $n_0 = 2k$.
    Then $n_0 \in \mathbb{N}$.    # $\lceil x \rceil, 1, 2, B \in \mathbb{N}$, $\mathbb{N}$ closed under max, $+$, $*$
    Then $n_0 \geqslant B$.    # at least twice $B$
    Then $2^k > c$.    # choice of $k$, $2^x$ is increasing function
    Then
```

$$\begin{aligned} g(n_0) = 2^{n_0} &= 2^k \times 2^k \quad \text{\# by choice of } k \\ &\geqslant 2^k \times 2k \quad \text{\# by } P(2k) \\ &= 2^k \times n_0 > cn_0 \quad \text{\# since } n_0 = 2k \text{ and } 2^k > c \\ &= cf(n_0). \end{aligned}$$

```
    Then $n_0 \geqslant B \wedge g(n_0) \geqslant cf(n_0)$.    # introduce $\wedge$
    Then $\exists n \in \mathbb{N}, n \geqslant B \wedge g(n) \geqslant cf(n)$.    # introduce $\exists$
Then $\forall c \in \mathbb{R}, \forall B \in \mathbb{N}, \exists n \in \mathbb{N}, n \geqslant B \wedge g(n) > cf(n)$.    # introduce $\forall$
```

So, I can conclude that $g \notin \mathcal{O}(f)$.

What happens to induction for predicates that are true for all natural numbers after a certain point, but untrue for the first few natural numbers? For example, 2^n grows much more quickly than n^2, but 2^3 is not larger than 3^2. Choose n big enough, though, and it is true that:

$$P(n) : 2^n > n^2.$$

You can't prove this for all n, when it is false for $n = 2, n = 3$, and $n = 4$, so you'll need to restrict the domain and prove that for all natural numbers greater than 4, $P(n)$ is true. We don't have a slick way to restrict domains in our symbolic notation. Let's consider three ways to restrict the natural numbers to just those greater than 4, and then use induction.

RESTRICT BY SET DIFFERENCE: One way to restrict the domain is by set difference:

$$\forall n \in \mathbb{N} \setminus \{0, 1, 2, 3, 4\}, P(n)$$

Again, we'll need to prove $P(5)$, and then that $\forall n \in \mathbb{N} \setminus \{0, 1, 2, 3, 4\}, P(n) \Rightarrow P(n + 1)$.

RESTRICT BY TRANSLATION: We can also restrict the domain by translating our predicate, by letting $Q(n) = P(n + 5)$, that is:

$$Q(n) : 2^{n+5} > (n + 5)^2$$

Now our task is to prove $Q(0)$ is true and that for all $n \in \mathbb{N}$, $Q(n) \Rightarrow Q(n+1)$. This is simple induction.

RESTRICT USING IMPLICATION: Another method of restriction uses implication to restrict the domain where we claim $P(n)$ is true — in the same way as for sentences:

$$\forall n \in \mathbb{N}, n \geqslant 5 \Rightarrow P(n).$$

The expanded predicate $Q(n) : n \geqslant 5 \Rightarrow P(n)$ now fits our pattern for simple induction, and all we need to do is prove:

1. $Q(0)$ is true (it is vacuously true, since $0 \geqslant 5$ is false).
2. $\forall n \in \mathbb{N}, Q(n) \Rightarrow Q(n + 1)$. This breaks into cases.
   - If $n < 4$, then $Q(n)$ and $Q(n + 1)$ are both vacuously true (the antecedents of the implication are false, since n and $n + 1$ are not greater than, nor equal to, 5), so there is nothing to prove.
   - If $n = 4$, then $Q(n)$ is vacuously true, but $Q(n + 1)$ has a true antecedent ($5 \geqslant 5$), so we need to prove $Q(5)$ directly: $2^5 > 5^2$ is true, since $32 > 25$.
   - If $n > 4$, we can depend on the assumption of the consequent of $Q(n - 1)$ being true to prove $Q(n)$:

$$\begin{aligned} 2^n &= 2^{n-1} + 2^{n-1} \quad (\text{definition of } 2^n) \\ &> 2(n-1)^2 \quad (\text{antecedent of } Q(n-1)) \\ &= 2n^2 - 2n + 2 = n^2 + n(n-2) + 2 \geqslant n^2 + 2 > n^2 \quad (\text{since } n > 4 \geqslant 2) \end{aligned}$$

After all that work, it turns out that we need prove just two things:

1. $P(5)$
2. $\forall n \in \mathbb{N}$, If $n > 4$, then $P(n) \Rightarrow P(n + 1)$.

This is the same as before, except now our base case is $P(5)$ rather than $P(0)$, and we get to use the fact that $n \geqslant 5$ in our induction step (if we need it).

Whichever argument you're comfortable with, notice that simple induction is basically the same: you prove the base case (which may now be greater than 0), and you prove the induction step.

Some theorems

Here are some general results that we now have the tools to prove.

- $f \in \mathcal{O}(f)$.
- $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$.
- $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$.
- $g \in \Theta(f) \Leftrightarrow g \in \mathcal{O}(f) \wedge g \in \Omega(f)$.

Test your intuition about Big-O by doing the "scratch work" to answer the following questions:

- Are there functions f, g such that $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(f)$ but $f \neq g$?[8]
- Are there functions f, g such that $f \notin \mathcal{O}(g)$, and $g \notin \mathcal{O}(f)$?[9]

To show that $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$, we need to find a constant $c \in \mathbb{R}^+$ and a constant $B \in \mathbb{N}$, that satisfy:

$$\forall n \in \mathbb{N}, n \geqslant B \Rightarrow f(n) \leqslant ch(n).$$

Since we have constants that scale h to g and then g to f, it seems clear that we need their product to scale g to f. And if we take the maximum of the two starting points, we can't go wrong. Making this precise:

Theorem 5.1: For any functions $f, g, h : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$.

Proof:

```
Assume $f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)$.
    So $f \in \mathcal{O}(g)$.
    So $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n > B \Rightarrow f(n) \leqslant cg(n)$.    # by def'n of $f \in \mathcal{O}(g)$
    Let $c_g \in \mathbb{R}^+, B_g \in \mathbb{N}$ be such that $\forall n \in \mathbb{N}, n \geqslant B_g \Rightarrow f(n) \leqslant c_g g(n)$.
    So $g \in \mathcal{O}(h)$.
    So $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow g(n) \leqslant ch(n)$.    # by def'n of $g \in \mathcal{O}(h)$
    Let $c_h \in \mathbb{R}^+, B_h \in \mathbb{N}$ be such that $\forall n \in \mathbb{N}, n \geqslant B_h \Rightarrow g(n) \leqslant c_h h(n)$.
    Let $c' = c_g c_h$. Let $B' = \max(B_g, B_h)$.
    Then, $c' \in \mathbb{R}^+$ (because $c_g, c_h \in \mathbb{R}^+$) and $B' \in \mathbb{N}$ (because $B_g, B_h \in \mathbb{N}$).
    Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
        Then $n \geqslant B_h$ (by definition of max), so $g(n) \leqslant c_h h(n)$.
        Then $n \geqslant B_g$ (by definition of max), so $f(n) \leqslant c_g g(n) \leqslant c_g c_h h(n)$.
        So $f(n) \leqslant c' h(n)$.
    Hence, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow f(n) \leqslant c' h(n)$.
    Therefore, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow f(n) \leqslant ch(n)$.
    So $f \in \mathcal{O}(g)$, by definition.
So $(f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h)) \Rightarrow f \in \mathcal{O}(h)$.
```

To show that $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$, it is enough to note that the constant, c, for one direction is positive, so its reciprocal will work for the other direction.[10]

Theorem 5.2: For any functions $f, g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $g \in \Omega(f) \Leftrightarrow f \in \mathcal{O}(g)$.

Proof:

$$\begin{aligned}
& g \in \Omega(f) \\
\iff & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow g(n) \geqslant cf(n) && \text{(by definition)} \\
\iff & \exists c' \in \mathbb{R}^+, \exists B' \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B' \Rightarrow f(n) \leqslant c' g(n) && \text{(letting } c' = 1/c \text{ and } B' = B) \\
\iff & f \in \mathcal{O}(g) && \text{(by definition)}
\end{aligned}$$

To show $g \in \Theta(f) \Leftrightarrow g \in \mathcal{O}(f) \wedge g \in \Omega(f)$, it's really just a matter of unwrapping the definitions.

THEOREM 5.3: For any functions $f, g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $g \in \Theta(f) \Leftrightarrow g \in \mathcal{O}(f) \wedge g \in \Omega(f)$.

PROOF:

$g \in \Theta(f)$
$\Leftrightarrow$ (by definition)
$\exists c_1 \in \mathbb{R}^+, \exists c_2 \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow c_1 f(n) \leqslant g(n) \leqslant c_2 f(n)$.
$\Leftrightarrow$ (combined inequality, and $B = \max(B_1, B_2)$)
$\left(\exists c_1 \in \mathbb{R}^+, \exists B_1 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B_1 \Rightarrow g(n) \geqslant c_1 f(n)\right) \wedge$
$\left(\exists c_2 \in \mathbb{R}^+, \exists B_2 \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B_2 \Rightarrow g(n) \leqslant c_2 f(n)\right)$
$\Leftrightarrow$ (by definition)
$g \in \Omega(f) \wedge g \in \mathcal{O}(f)$

Here's an example of a corollary that recycles some of the theorems we've already proven (so we don't have to do the grubby work). To show $g \in \Theta(f) \Leftrightarrow f \in \Theta(g)$, I re-use theorems proved above and the commutativity of $\wedge$:

COROLLARY: For any functions $f, g : \mathbb{N} \to \mathbb{R}^{\geqslant 0}$, we have $g \in \Theta(f) \Leftrightarrow f \in \Theta(g)$.

PROOF:

$$\begin{aligned}
& g \in \Theta(f) \\
\iff\ & g \in \mathcal{O}(f) \wedge g \in \Omega(f) && \text{(by 5.3)} \\
\iff\ & g \in \mathcal{O}(f) \wedge f \in \mathcal{O}(g) && \text{(by 5.2)} \\
\iff\ & f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(f) && \text{(by commutativity of } \wedge\text{)} \\
\iff\ & f \in \mathcal{O}(g) \wedge f \in \Omega(g) && \text{(by 5.2)} \\
\iff\ & f \in \Theta(g) && \text{(by 5.3)}
\end{aligned}$$

A VERY IMPORTANT NOTE

Note that asymptotic notation (the Big-$\mathcal{O}$, Big-Ω, and Big-Θ definitions) bound the asymptotic growth rates of *functions*, as n approaches infinity. Often in computer science we use this asymptotic notation to bound functions that express the running times of algorithms, perhaps in best case or in worst case. Asymptotic notation *does not* express or bound the worst case or best case running time, only the *functions* expressing these values.

This distinction is subtle, but crucial to understanding both running times and asymptotic notation. Suppose U is an upper bound on the worst-case running time of some program P, denoted $T_P(n)$:

$$\begin{aligned}
& T_P \in \mathcal{O}(U) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T_P(n) \leqslant cU(n) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \max\{t_P(x) \mid x \in I \wedge \text{size}(x) = n\} \leqslant cU(n) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \forall x \in I, \text{size}(x) = n \Rightarrow t_P(x) \leqslant cU(n) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall x \in I, \text{size}(x) \geqslant B \Rightarrow t_P(x) \leqslant cU(\text{size}(x))
\end{aligned}$$

So to show that $T_P \in \mathcal{O}(U(n))$, you need to find constants c and B and show that for an arbitrary input x of size n, P takes at most $c \cdot U(n)$ steps.
In the other direction, suppose L is a lower bound on the worst-case running time of algorithm P:

$$\begin{aligned}
& T_P \in \Omega(L) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \max\{t_P(x) \mid x \in I \wedge \text{size}(x) = n\} \geqslant cL(n) \\
\iff\ & \exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow \exists x \in I, \text{size}(x) = n \wedge t_P(x) \geqslant cL(n)
\end{aligned}$$

So, to prove that $T_p \in \Omega(L)$, we have to find constants c, B and for arbitrary n, find an input x of size n, for which we can show that P takes at least $cL(n)$ steps on input x

5.9 Insertion sort example

Here is an intuitive[11] sorting algorithm:

```
def IS(A):
""" Sort the elements of A in non-decreasing order. """
  i = 1                              # (line 1)
  while i < len(A):                  # (line 2)
      t = A[i]                       # (line 3)
      j = i                          # (line 4)
      while j > 0 and A[j-1] > t:    # (line 5)
          A[j] = A[j-1]              # (line 6)
          j = j-1                    # (line 7)
      A[j] = t                       # (line 8)
      i = i+1                        # (line 9)
```

Let's find an upper bound for $T_{IS}(n)$, the maximum number of steps to Insertion Sort an array of size n. We'll use the proof format to prove and find the bound simultaneously — during the course of the proof we can fill in the necessary values for c and B.
We show that $T_{IS}(n) \in \mathcal{O}(n^2)$ (where $n = \text{len}(A)$):

> Let $c' =$ ____. Let $B' =$ ____.
> Then $c' \in \mathbb{R}^+$ and $B' \in \mathbb{N}$.
> Assume $n \in \mathbb{N}$, A is an array of length n, and $n \geqslant B'$.
>
> > Then lines 5–7 execute at most $n-1$ times, which we can overestimate at $3n$ steps, plus 1 step for the last loop test.
> > Then lines 2–9 take no more than $n(5+3n)+1 = 5n+3n^2+1$ steps.
> > So $3n^2+5n+1 \leqslant c'n^2$ (fill in the values of c' and B' that makes this so — setting $c' = 9, B' = 1$ should do).
>
> Since n is the length of an arbitrary array A, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow T_{IS}(n) \leqslant c'n^2$ (so long as $B' \geqslant 1$).
> Since c' is a positive real number and B' is a natural number,
> $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T_{IS}(n) \leqslant cn^2$.
> So $T_{IS} \in \mathcal{O}(n^2)$ (by definition of $\mathcal{O}(n^2)$).

Similarly, we prove a lower bound. Specifically, $T_{IS} \in \Omega(n^2)$:

> Let $c' =$ ____. Let $B' =$ ____.
> Then $c' \in \mathbb{R}^+$ and $B' \in \mathbb{N}$.
> Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
>
> > Let $A' = [n-1, \ldots, 1, 0]$ (notice that this means $n \geqslant 1$).
> > Then at any point during the outside loop, $A'[0..(i-1)]$ contains the same elements as before but sorted (*i.e.*, no element from $A'[(i+1)..(n-1)]$ has been examined yet).
> > Then the inner while loop makes i iterations, at a cost of 3 steps per iteration, plus 1 for the final loop check, since the value $A'[i]$ is less than all the values $A'[0..(i-1)]$, by construction of the array.
> > Then the inner loop makes strictly greater than $2i+1$, or greater than or equal to $2i+2$., steps.
> > Then (since the outer loop varies from $i = 1$ to $i = n-1$ and we have $n-1$ iterations of lines 3 and 4, plus one iteration of line 1), we have that $t_{IS}(n) \geqslant 1+3+5+\cdots+(2n-1)+(2n+1) = n^2$ (the sum of the first n odd numbers), so long as n is at least 4.

So there is some array A of size n such that $t_{IS}(A) \geqslant c'n^2$.
This means $T_{IS}(n) \geqslant c'n^2$ (setting $B' = 4, c' = 1$ will do).
Since n was an arbitrary natural number, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow T_{IS}(n) \geqslant c'n^2$.
Since $c' \in \mathbb{R}^+$ and B' is a natural number, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow T_{IS}(n) \geqslant cn^2$.
So $T_{IS} \in \Omega(n^2)$ (by definition of $\Omega(n^2)$).

From these proofs, we conclude that $T_{IS} \in \Theta(n^2)$.

Exercises for asymptotic notation

1. Prove or disprove the following claims:
   (a) $7n^3 + 11n^2 + n \in \mathcal{O}(n^3)$[12]
   (b) $n^2 + 165 \in \Omega(n^4)$
   (c) $n! \in \mathcal{O}(n^n)$
   (d) $n \in \mathcal{O}(n \log_2 n)$
   (e) $\forall k \in \mathbb{N}, k > 1 \Rightarrow \log_k n \in \Theta(\log_2 n)$

2. Define $g(n) = \begin{cases} n^3/165, & n < 165 \\ \lceil \sqrt{6n^5} \rceil, & n \geqslant 165 \end{cases}$. Note that $\forall x \in \mathbb{R}, x \leqslant \lceil x \rceil < x + 1$.

   Prove that $g \in \mathcal{O}(n^{2.5})$.

3. Let $\mathcal{F}$ be the set of functions from $\mathbb{N}$ to $\mathbb{R}^{\geqslant 0}$. Prove the following theorems:
   (a) For $f, g \in \mathcal{F}$, if $g \in \Omega(f)$ then $g^2 \in \Omega(f^2)$.
   (b) $\forall k \in \mathbb{N}, k > 1 \Rightarrow \forall d \in \mathbb{R}^+, d \log_k n \in \Theta(\log_2 n)$.[13]

   Notice that (b) means that all logarithms eventually grow at the same rate (up to a multiplicative constant), so the base doesn't matter (and can be omitted inside the asymptotic notation).

4. Let $\mathcal{F}$ be the set of functions from $\mathbb{N}$ to $\mathbb{R}^{\geqslant 0}$. Prove or disprove the following claims:
   (a) $\forall f \in \mathcal{F}, \forall g \in \mathcal{F}, f \in \mathcal{O}(g) \Rightarrow (f + g) \in \Theta(g)$
   (b) $\forall f \in \mathcal{F}, \forall f' \in \mathcal{F}, \forall g \in \mathcal{F}, (f \in \mathcal{O}(g) \wedge f' \in \mathcal{O}(g)) \Rightarrow (f + f') \in \mathcal{O}(g)$

5. For each function f in the left column, choose one expression $\mathcal{O}(g)$ from the right column such that $f \in \mathcal{O}(g)$. Use each expression exactly once.

   (i) $3 \cdot 2^n \in$ ________
   (ii) $\frac{2n^4+1}{n^3+2n-1} \in$ ________
   (iii) $(n^5 + 7)(n^5 - 7) \in$ ________
   (iv) $\frac{n^4 - n\log_2 n}{n^2+1} \in$ ________
   (v) $\frac{n \log_2 n}{n-5} \in$ ________
   (vi) $8 + \frac{1}{n^2} \in$ ________
   (vii) $2^{3n+1} \in$ ________
   (viii) $n! \in$ ________
   (ix) $\frac{5\log_2(n+1)}{1+n\log_2 3n} \in$ ________
   (x) $(n-2)\log_2(n^3+4) \in$ ________

   (a) $\mathcal{O}(\frac{1}{n})$
   (b) $\mathcal{O}(1)$
   (c) $\mathcal{O}(\log_2 n)$
   (d) $\mathcal{O}(n)$
   (e) $\mathcal{O}(n \log_2 n)$
   (f) $\mathcal{O}(n^2)$
   (g) $\mathcal{O}(n^{10})$
   (h) $\mathcal{O}(2^n)$
   (i) $\mathcal{O}(10^n)$
   (j) $\mathcal{O}(n^n)$

Chapter 5 Notes

[1]From 0 to $(2-1)$, if we work in analogy with base 10.

[2]To parse the 0.101 part, calculate $0.101 = 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3})$.

[3]You should be able to look up this algorithm in an elementary school textbook.

[4]Same as the previous exercise, but only write numbers that have 0's and 1's, and do binary addition.

[5]Integer divide by 10.

[6]Integer divide by 2.

[7]Better in the sense of time complexity.

[8]Sure, $f = n^2$, $g = 3n^2 + 2$.

[9]Sure. f and g don't need to both be monotonic, so let $f(n) = n^2$ and

$$g(n) = \begin{cases} n, & n \text{ even} \\ n^3, & n \text{ odd} \end{cases}$$

So not every pair of functions from $\mathbb{N} \to \mathbb{R}^{\geqslant 0}$ can be compared using Big-O.

[10]Let's try the symmetrical presentation of bi-implication.

[11]but not particularly efficient...

[12]The claim is true.

Let $c' = 8$. Then $c' \in \mathbb{R}^+$.
Let $B' = 12$. Then $B' \in \mathbb{N}$.
Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
  Then $n^3 = n \cdot n^2 \geqslant 12 \cdot n^2 = 11n^2 + n^2 \geqslant 11n^2 + n$.   # since $n \geqslant B' = 12$
  Thus $c'n^3 = 8n^3 = 7n^3 + n^3 \geqslant 7n^3 + 11n^2 + n$.
So $\forall n \in \mathbb{N},\ n \geqslant B' \Rightarrow 7n^3 + 11n^2 + n \leqslant c'n^3$.
Since B' is a natural number, $\exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 7n^3 + 11n^2 + n \leqslant c'n^3$.
Since c' is a real positive number, $\exists c \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow 7n^3 + 11n^2 + n \leqslant cn^3$.
By definition, $7n^3 + 11n^2 + n \in \mathcal{O}(n^3)$.

[13] Assume $k \in \mathbb{N}$ and $k > 1$.
  Assume $d \in \mathbb{R}^+$.
    It suffices to argue that $d \log_k n \in \Theta(\log_2 n)$.
    Let $c_1' = \frac{d}{\log_2 k}$. Since $k > 1$, $\log_2 k \neq 0$ and so $c_1' \in \mathbb{R}^+$.
    Let $c_2' = \frac{d}{\log_2 k}$. By the same reasoning, $c_2' \in \mathbb{R}^+$.
    Let $B' = 1$. Then $B' \in \mathbb{N}$.
    Assume $n \in \mathbb{N}$ and $n \geqslant B'$.
      Then $c_1' \log_2 n = \frac{d}{\log_2 k} \log_2 n = d\frac{\log_2 n}{\log_2 k} = d \log_k n \leqslant d \log_k n$.
      Moreover, $d \log_k n \leqslant d\frac{\log_2 n}{\log_2 k} = \frac{d}{\log_2 k} \log_2 n = c_2' \log_2 n$.
    Hence, $\forall n \in \mathbb{N}, n \geqslant B' \Rightarrow c_1' \log_2 n \leqslant d \log_k n \leqslant c_2' \log_2 n$.
    Thus $\exists c_1 \in \mathbb{R}^+, \exists c_2 \in \mathbb{R}^+, \exists B \in \mathbb{N}, \forall n \in \mathbb{N}, n \geqslant B \Rightarrow c_1 \log_2 n \leqslant d \log_k n \leqslant c_2 \log_2 n$.
    By definition, $d \log_k n \in \Theta(\log_2 n)$.

Thus, $\forall d \in \mathbb{R}^+, d \log_k n \in \Theta(\log_2 n)$.

Hence $\forall k \in \mathbb{N}, k > 1 \Rightarrow \forall d \in \mathbb{R}^+, d \log_k n \in \Theta(\log_2 n)$.