2.3 Asymptotic Analysis

It has already been described qualitatively that if we intend to store nothing but objects, that this can be done quickly using a hash table; however, if we wish to store relationships and perform queries and data manipulations based on that relationship, it will require more time and memory. The words "quickly" and "more" are qualitative.

In order to make rational engineering decisions about implementations of data structures and algorithms, it is necessary to describe such properties quantitatively: "How much faster?" and "How much more memory?"

The best case to be made is that a new algorithm may be known to be *faster* than another algorithm, but that one word will not be able to allow any professional engineer to determine whether or not the newer algorithm is worth the time required to implement, integrate, document, and test the new algorithm. In some cases, it may be simply easier to buy a faster computer. Therefore, we will want to, instead, determine the run time of the algorithms using mathematics.

2.3.1 Variables of Analysis

In general, we need to describe how much time or memory will be required with respect to one or more variables. The most common variables include:

1. The number of objects n that are currently stored in a container,
2. The number of objects m that the container could possibly hold, or
3. The dimensions of a square $n \times n$ matrix.

In some cases, we may deal with multiple variables:

1. When dealing with n objects stored in a container with m memory locations,
2. Dealing with non-square $m \times n$ matrices, and
3. Dealing with sparse square $n \times n$ matrices where only m entries are non-zero.

We will use the following code as an example throughout the next two topics:

```
int find_max( int *array, int n ) {
    int max = array[0];

    for ( int i = 1; i < n; ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }

    return max;
}
```

Finding the maximum entry in an array requires that every entry of array to be inspected. We would expect that if we double the size of the array, we would expect it to take twice as long to find the maximum entry.

Aside: In C++, a primitive array is simply a pointer storing the address of the first entry of the array. Thus, the actual size of the array must be stored in a separate variable. In this case, the array pointer is passes as the first parameter, while the size is passed as a second. This breaks the object-oriented programming goal of encapsulation, where all information concerning a data structure is maintained together. The Standard Template Library (STL) has a class `vector` that more closely mimics the object-oriented characteristics of the Java and C# `Array` classes.

The previous algorithm implemented using the `vector` class would be:

```
#include <vector>

int find_max( std::vector<int> array ) {
    if ( array.size() == 0 ) {
        throw underflow();
    }

    int max = array[0];

    for ( int i = 1; i < array.size(); ++i ) {
        if ( array[i] > max ) {
            max = array[i];
        }
    }

    return max;
}
```

If we multiply an $n \times n$ matrix by an n-dimensional vector. Each entry of the matrix is multiplied by one entry in the vector. Therefore, if we multiply a $2n \times 2n$ matrix by a $2n$-dimensional vector, we would expect four times as many multiplications and therefore it should take about four times as long.

The question is, how can we express this mathematically?

2.3.2 Binary search versus linear search

As another example, consider searching an array for a value. If the array is sorted, we can do a binary search but if the array is not sorted, we must perform a linear search checking each entry of the array. Here are two implementations of these functions.

```
int linear_search( int value, int *array, int n ) {
    for ( int i = 0; i < n; ++i ) {
        if ( array[i] == value ) {
            return i;
        }
    }

    return -1;
}
```

```
int binary_search( int value, int *array, int n ) {
    int a = 0;
    int c = n - 1;

    while ( a <= c ) {
        int b = (a + c)/2;

        if ( array[b] == value ) {
            return b;
        } else if ( array[b] < value ) {
            a = b + 1;
        } else {
             c = b - 1;
        }
    }

    return -1;
}
```

In each case, there is a worst case as to how many entries of the array will be checked; however, there is also an average number of entries that will be checked. Figure 1 shows the worst-case and average number of comparisons required by both the linear and binary searches in searching an array of size *n* from 1 to 32.

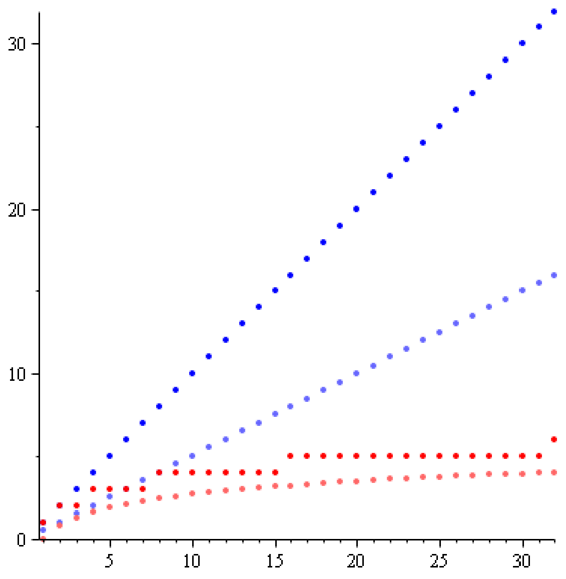

Figure 1. The average and worst-case number of comparisons required for using a linear search (blue) and a binary search (red) on an array if size *n* = 1, ..., 32.

It seems that even the worst-case for a binary search is significantly better than the average case for a linear search. Also, the number of comparisons required for a linear search appears to grow, well, linearly, whereas the number of searches required for a binary search appears to be growing at the same rate as either $\sqrt{n}$ or $\ln(n)$. Which one is it and why?

Another issue is what do we really care about? If Algorithm A runs twice as fast as Algorithm B, it is always possible to simply purchase a faster computer in which case Algorithm B will perform just as well, if not better than Algorithm A. An important question about the linear search versus the binary search is: can we purchase a computer fast enough so that linear search will always outperform a binary search implemented on the 1980s-era 68000 processor?

2.3.3 Rate of Growth of Polynomials

Consider any two polynomials of the same degree. If the coefficients of the leading term are the same, while the functions may be very different around zero, as you plot them on larger and larger intervals, they will appear to be essentially the same. Figure 2 shows two quadratic polynomials, n^2 and $n^2 - 3n + 2$, and while they appear different on the interval [0, 3], on [0, 1000], they are essentially identical.

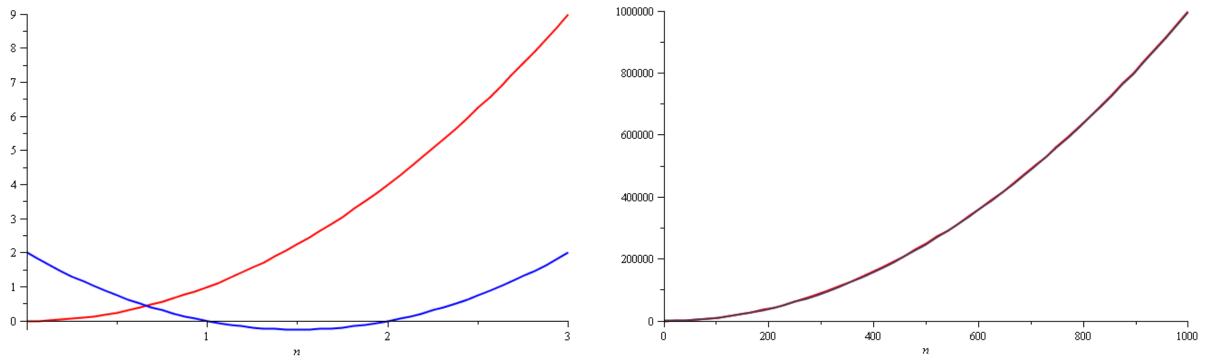

Figure 2. Two quadratic polynomials plotted over [0, 3] and [0, 1000].

Figure 3 shows two sextic policnomials: n^6 and $n^6 - 23n^5 + 193n^4 - 729n^3 + 12062^2 - 648n$ and while they appear to be very different on the interval [–2, 5], but again, on the interval [0, 1000], they appear to be essentially identical.

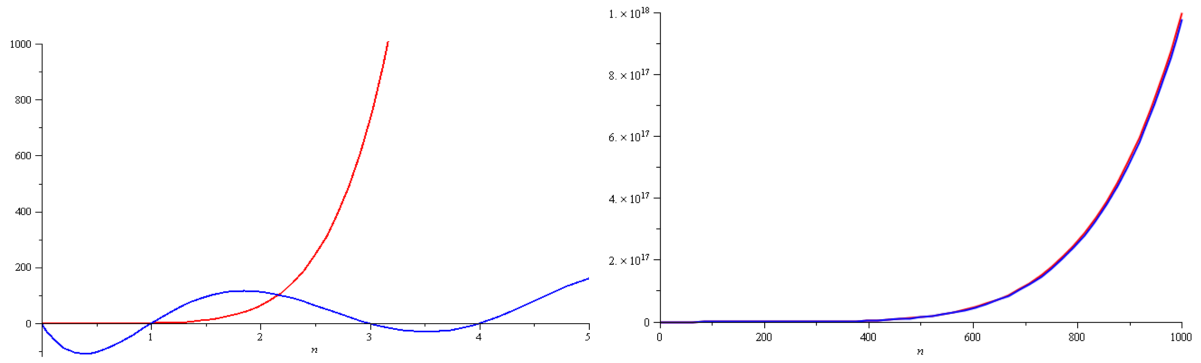

Figure3. Two sextic polynomials plotted on [0, 5] and again on [0, 1000].

If two polynomials have the same degree but the coefficients of the leading terms are different, again, while they may appear to be significantly different around the origin, on a larger scale, one will simply be a scalar multiple of the other.

2.3.4 Examples of algorithm analysis

We will look at two hypothetical examples: a comparison of selection sort and bubble sort and then quick sort and insertion sort, both with respect to run time.

2.3.4.1 Selection and bubble sorts

Consider selection sort which has a fixed run time and bubble sort which has best-case and worst-case run times:

Selection Sort		$4n^2 + 8.0n + 6$
Bubble Sort	best case	$4.7n^2 + 0.5n + 5$
	worst case	$3.8n^2 - 0.5n + 5$

These describe the number of instructions required to implement selection sort and bubble sort to sort a list of size *n*, respectively. You can look at appendix A to see the C++ source code and disassembled object code produced by g++. Initially, selection sort requires significantly even more instructions than the worst-case for bubble sort; however, for problems of size $n \geq 22$, selection sort performs better than the worst-case bubble sort. As the problem size becomes very large, selection sort falls approximately half way between the best and worst cases of bubble sort. This is shown in Figure 4.

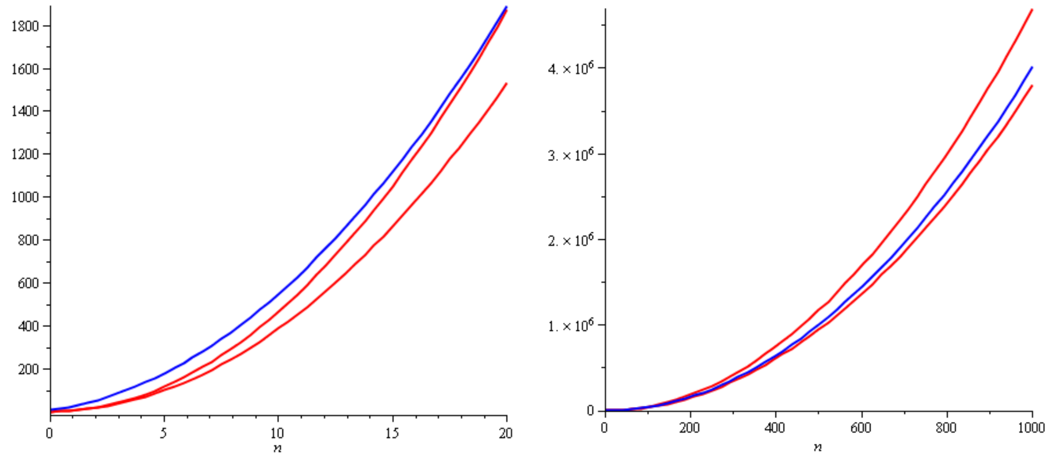

Figure 4. The instructions required to sort an array of size *n* for selection sort (blue) and the best and worst case scenarios for bubble sort (red).

As the problem size gets larger and larger, the rate of growth of all three functions is similar: the best case for bubble sort will require approximately 0.95 the number of instructions for selection sort and the worst case will require 1.175 times the number of instructions. Because the number of instructions can be calculated directly, we can also approximate the time it will take to execute this code on, for example, a computer running at 1 GHz: divide the number of instructions by 10^9. However, we can also, for example, run selection sort on a computer that runs at 2 GHz. Figure 5 shows the best and worst cases for bubble sort run on a 1 GHz computer; however, the left image shows the time required by selection sort run on a 1 GHz computer while the right shows it run on a 2 GHz computer.

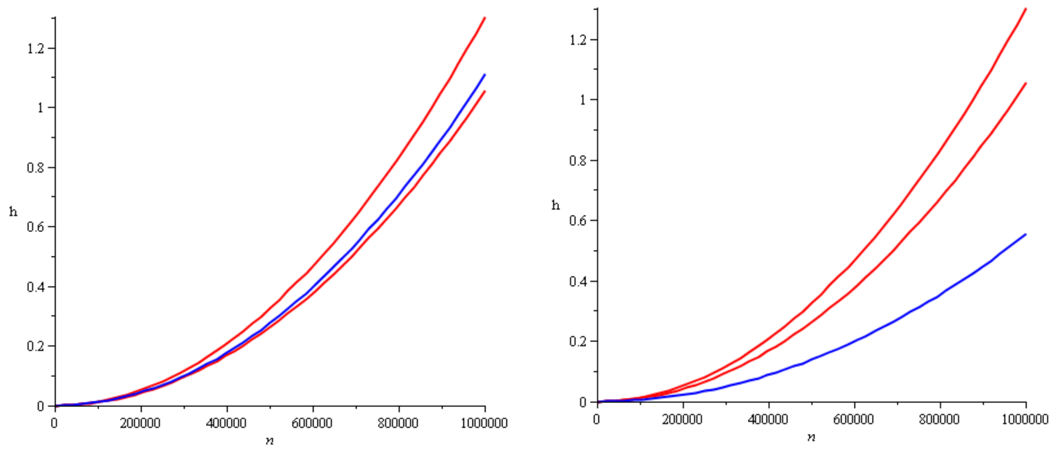

Figure 5. Time required to sort a problem of size up to one million for bubble sort run on a 1 GHz computer and selection sort run on 1 and 2 GHz computers, respectively.

The critical point here is that selection sort is not significantly better or worse than bubble sort. Any different in speed can be compensated by using better hardware.

The justification for the behavior of these to algorithms is that the run times are quadratic functions and thus we can always speed one up by a scalar multiple as compared to the other when running it on a faster computer.

2.3.4.2 Insertion sort and quicksort

Next, we will compare insertion sort and quicksort. Both functions expressing the run times are concave up, and for small problems, it seems that quicksort is slower.

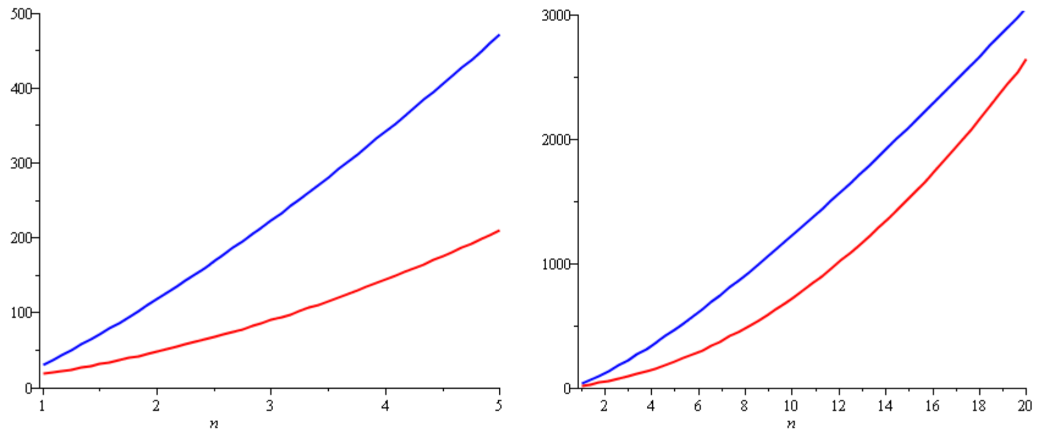

Figure 6. A comparison of runtimes for insertion sort (red) and quicksort (blue).

For larger lists, however, quicksort is always faster, as is shown in Figure 7.

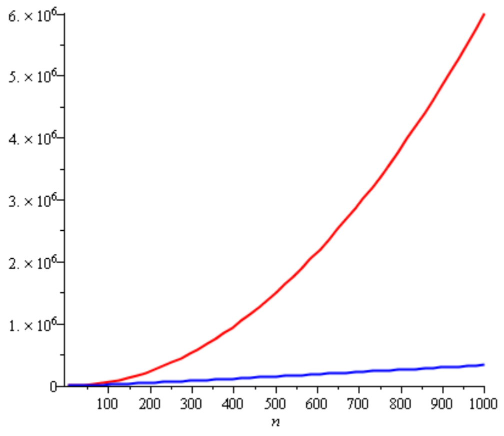

Figure 7. A comparison of runtimes for insertion sort (red) and quicksort (blue).

You may ask yourself whether or not purchasing a faster computer may make insertion sort faster than quick sort? The answer is: yes, up to a specific value of *n*, but ultimately, quicksort will always be significantly faster than insertion sort. This also suggests, however, that if the problem size is known to be small, perhaps it is better to use insertion sort.

2.3.5 Big-Oh and Big-Theta

What we need is a way of saying that two functions are growing at the same rate—that is, they are essentially growing at the same rate. To do this, we must recall a concept from first year: big-Oh notation. This is one of five *Landau symbols* that we will use in this class.

You will recall that in first year, you described $f(n) = \mathrm{O}(\mathrm{g}(n))$ if

$$\exists M > 0 \text{ and } \exists N > 0 \text{ such that } \left|f(n)\right| < M\left|g(n)\right| \text{ whenever } n > N.$$

If we are dealing with functions such as linear combinations of terms like n^r or $n^r\ln(n)$ where r is a positive real number where the final result is monotonically increasing and positive for $n > 0$ (behaviours we would expect from functions describing algorithms), then $f(n) = \mathrm{O}(\mathrm{g}(n))$ is equivalent to saying that

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty.$$

However, to say that two functions are growing at the same rate, we need a stronger restriction: two functions will be said to be growing at the same rate if

$$0 < \lim_{n\to\infty}\frac{f(n)}{g(n)} < \infty$$

and we will write that $f(n) = \Theta(\mathrm{g}(n))$.

For example, two polynomials of the same degree are big-theta of each other because the above limit will be the ratio of the coefficients of the leading terms, for example,

$$\lim_{n\to\infty}\frac{3n^2+5n+10}{8n^2+7n+98} = \frac{3}{8}.$$

For real values $p \geq 0$ and $q \geq 0$, it is true that

1. $n^p = \Theta(n^q)$ if and only if $p = q$, and
2. $n^p = \mathrm{O}(n^q)$ if and only if $p \leq q$.

2.3.5.1 Consequence

If $f(n)$ and $g(n)$ describes either the time required or the number of instructions required for two different algorithms to solve a problem of size n and $f(n) = \Theta(\mathrm{g}(n))$, we can always make one function run faster than the other by having sufficiently better hardware.

2.3.6 Little-oh

Consider the two functions $f(n) = \ln(n)$ and $g(n) = n$. It is already true that for all $n > 0$, $n > \ln(n)$. However, is there a sufficiently small value of $c > 0$ such that $\ln(n) > cn$ for all $n > N$ for some N?

A little thought will probably convince you that no such positive number exists, for no matter what number we choose,

$$\lim_{n\to\infty}\frac{\ln(n)}{cn} = \lim_{n\to\infty}\frac{1/n}{c} = \frac{1}{c}\lim_{n\to\infty}\frac{1}{n} = 0\,.$$

Therefore, the logarithm grows significantly slower than the function n. We would also like to describe when one function $f(n)$ grows significantly slower than another function $g(n)$, and therefore we will say that $f(n) = \mathrm{o}(\mathrm{g}(n))$ (little-oh) if

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0\,.$$

Thus, we could write that $\ln(n) = \mathrm{o}(n)$. A nice way of remembering this is that the little "o" looks like a zero and the limit is zero.

We may continue the analogy: for real values $p \geq 0$ and $q \geq 0$, it is true that

1. $n^p = \Theta(n^q)$ if and only if $p = q$,
2. $n^p = \mathrm{O}(n^q)$ if and only if $p \leq q$, and
3. $n^p = \mathrm{o}(n^q)$ if and only if $p < q$.

2.3.7 Little-omega and Big-Omega

Current, we have the following table:

Landau Symbol	**Limit**	**Description**	**Analogous Relational Operator**
$f(n) = \Theta(g(n))$	$0 < c < \infty$	f grows at the same rate as g	$=$
$f(n) = \mathrm{O}(g(n))$	$c < \infty$	f grows at the same rate as or slower than g	$\leq$
$f(n) = \mathrm{o}(g(n))$	0	f grows significantly slower than g	$<$

What we are missing are means of describing if one functions grows *faster* than another function. We will add two more Landau symbols: ω and Ω. We will say that $f(n)$ grows significantly faster than $g(n)$ if

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

and we will write this as $f(n) = \omega(\mathrm{g}(n))$. Note that little-omega looks like the infinity symbol.

We will also say that $f(n)$ grows either at the same rate or faster than $g(n)$ if

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} > 0$$

and we will write this as $f(n) = \Omega(g(n))$.

We may continue the analogy: for real values $p \geq 0$ and $q \geq 0$, it is true that

1. $n^p = \omega(n^q)$ if and only if $p > q$, and
2. $n^p = \Omega(n^q)$ if and only if $p \geq q$.

Thus, we have the full table:

Landau Symbol	**Limit**	**Description**	**Analogous Relational Operator**
$f(n) = \omega(g(n))$	∞	f grows significantly faster than g	$>$
$f(n) = \Omega(g(n))$	$0 < c$	f grows at the same rate as or faster than g	$\geq$
$f(n) = \Theta(g(n))$	$0 < c < \infty$	f grows at the same rate as g	$=$
$f(n) = O(g(n))$	$c < \infty$	f grows at the same rate as or slower than g	$\leq$
$f(n) = o(g(n))$	0	f grows significantly slower than g	$<$

2.3.8 Big-Theta as an Equivalence Relation

There are some interesting characteristics of Landau symbols with respect to the functions that we are interested in:

1. $f(n) = \Theta(f(n))$,
2. $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$, and
3. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, it follows that $f(n) = \Theta(h(n))$.

Therefore, big-Theta defines an equivalence relation on functions. One of the properties of equivalence relations is that you can create equivalence classes of all functions that are big-Theta of each other.

For example, all of the functions

n^2	$32n^2 + 54n + 7$	$n^2 + n + \ln(n) + 1$
$739n^2$	$n^2 + n\ln(n)$	$n^2 + n^{1.5} + n + n^{0.5} + 1$
$5.2n^2 + 4.7n$	$0.00523n^2$	$n^2 + 2n + 3$

grow at the same rate as each other. Therefore, to describe this entire equivalence class of functions that grow at the same rate as a quadratic monomial, we will select on member to represent the entire class and while we could chose $n^2 + n^{1.5} + n + n^{0.5} + 1$, it would make more sense to choose n^2 and call this class of functions *quadraticly growing functions*.

There are specific equivalence classes that appear so often in the discussion of algorithm analysis that we given them special names. These are listed in Table 1.

Table 1. Equivalence classes of equivalent rates of growth.

Equivalence Class	Representative
Constant	1
Logarithmic	$\ln(n)$
Linear	N
n-log-n	$n \ln(n)$
Quadratic	n^2
Cubic	n^3
Exponentials	2^n, e^n, 3^n, *etc*.

Note that with the exponential functions, there is not one single representative, for if $a < b$ then a^n grows slower than b^n:

$$\lim_{n\to\infty}\frac{a^n}{b^n} = \lim_{n\to\infty}\left(\frac{a}{b}\right)^n = 0 \text{ because } \frac{a}{b} < 1.$$

2.3.9 Little-oh as a Weak Ordering

We have already noted that for any real $0 \le p < q$, it follows that $n^p = \mathrm{o}(n^q)$ and therefore, we can order the equivalence classes. In addition to these classes, we also note that

$$1 = \mathrm{o}(\ln(n)),$$

however, $\ln(n) = \mathrm{o}(n^p)$ for any $p > 0$. This can be seen by seeing that

$$\lim_{n\to\infty}\frac{\ln(n)}{n^p} = \lim_{n\to\infty}\frac{1/n}{pn^{p-1}} = \frac{1}{p}\lim_{n\to\infty}\frac{1}{n^p} = 0 \text{ as } p > 0.$$

Similarly, $n^p = \mathrm{o}(n^q \ln(n))$ for any $0 \le p \le q$ and $n^q \ln(n) = \mathrm{o}(n^r)$ for any $0 \le q < r$.

Graphically, we can show this weak ordering as shown in Figure 8.

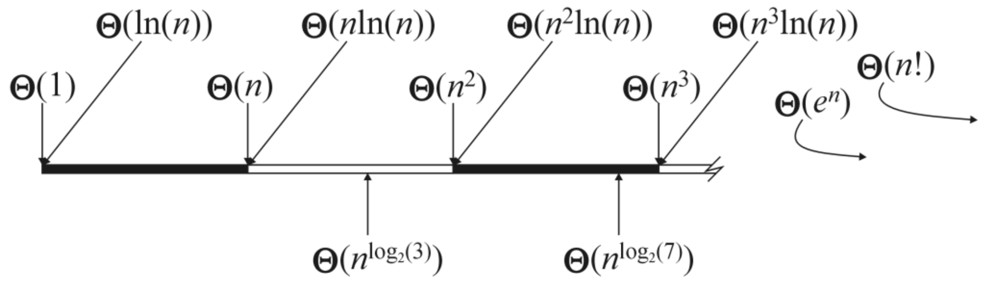

Figure 8. The ordering of the equivalence classes of functions growing at the same rate.

Notes (beyond the scope of this class)

We could also consider functions that grow according to $n^p \ln^q(n)$ where $p, q \geq 0$ as well as functions such as $\ln(\ln(n))$. There are searching algorithms that can run as fast as $\Theta(\ln(\ln(n)))$ meaning that a problem of size $n = 10^{24}$ requires only four times longer to solve than a problem of size $n = 15$.

We are restricting our definitions to using limits. To be more correct, we should use the *limit supremum* and *limit infimum*. Technically, we should say that $f(n) = \Theta(g(n))$ if

$$0 < \liminf_{n\to\infty} \frac{f(n)}{g(n)} \leq \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

and $f(n) = \mathrm{O}(g(n))$ if

$$\limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty .$$

2.3.10 What's Next?

We will use Landau symbols to describe the run-times and memory usage of various data structures and algorithms.

An algorithm will be said to have *polynomial time complexity* if its run time may be described by $\mathrm{O}(n^p)$ for some $p \geq 0$. In a general sense, problems that can be solved with known polynomial time algorithms are said to be *efficiently solvable* or *tractable*.

Problems for which there are no algorithms that can solve the problem in polynomial time are said to be *intractable*. For example, the travelling salesman problem (find the shortest path that visits each of n cities exactly once) can only be solved by an algorithm that is requires $\Theta(n^2\, 2^n)$ time: add one more city and the algorithm takes more than twice as long to run. Add 10 cities and the algorithm takes more than $2^{10} = 1024$ times to run.

Algorithms that require an exponential amount of time are, for the most part, undesirable. Be sure to never describe a function with quadratic growth as *exponential*.

Appendix A

Source Code

```
void bubble_sort( int *array, int n ) {
        for ( int i = (n - 1); i >= 1; --i ) {
                for ( int j = 0; j < i; ++j ) {
                        if ( array[j] > array[j + 1] ) {
                                int tmp = array[j];
                                array[j] = array[j + 1];
                                array[j + 1] = tmp;
                        }
                }
        }
}
```

Output of

```
% g++ -O -c bubble_sort.cpp
% objdump –d bubble_sort.o
```

```
   0:   83 ee 01                sub    $0x1,%esi
   3:   85 f6                   test   %esi,%esi
   5:   7f 21                   jg     28
   7:   f3 c3                   repz retq
   9:   8b 0c 87                    mov    (%rdi,%rax,4),%ecx
   c:   8b 54 87 04                 mov    0x4(%rdi,%rax,4),%edx
  10:   39 d1                       cmp    %edx,%ecx
  12:   7e 07                       jle    1b
  14:   89 14 87                      mov    %edx,(%rdi,%rax,4)
  17:   89 4c 87 04                   mov    %ecx,0x4(%rdi,%rax,4)
  1b:   48 83 c0 01                 add    $0x1,%rax
  1f:   39 c6                       cmp    %eax,%esi
  21:   7f e6                       jg     9
  23:   83 ee 01                  sub    $0x1,%esi
  26:   74 07                     je     2f
  28:   b8 00 00 00 00            mov    $0x0,%eax
  2d:   eb da                     jmp    9
  2f:   f3 c3                   repz retq
```

Soucre Code

```
void selection_sort( int *array, int const n ) {
        for ( int i = (n - 1); i >= 1; --i ) {
                int posn = 0;

                for ( int j = 1; j <= i; ++j ) {
                        if ( array[j] > array[posn] ) {
                                posn = j;
                        }
                }

                int tmp = array[i];
                array[i] = array[posn];
                array[posn] = tmp;
        }
}
```

Output of

```
% g++ -O -c selection_sort.cpp
% objdump –d selection_sort.o
```

```
   0:   83 ee 01                sub    $0x1,%esi
   3:   48 63 c6                movslq %esi,%rax
   6:   85 f6                   test   %esi,%esi
   8:   4c 8d 14 87             lea    (%rdi,%rax,4),%r10
   c:   7e 45                   jle    53
   e:   49 89 f9                  mov    %rdi,%r9
  11:   b9 01 00 00 00            mov    $0x1,%ecx
  16:   45 31 c0                  xor    %r8d,%r8d
  19:   0f 1f 80 00 00 00 00      nopl   0x0(%rax)
  20:   41 8b 41 04                 mov    0x4(%r9),%eax
  24:   42 3b 04 87                 cmp    (%rdi,%r8,4),%eax
  28:   48 63 d1                    movslq %ecx,%rdx
  2b:   4c 0f 4f c2                 cmovg  %rdx,%r8
  2f:   83 c1 01                    add    $0x1,%ecx
  32:   49 83 c1 04                 add    $0x4,%r9
  36:   39 f1                       cmp    %esi,%ecx
  38:   7e e6                       jle    20
  3a:   4a 8d 14 87               lea    (%rdi,%r8,4),%rdx
  3e:   41 8b 0a                  mov    (%r10),%ecx
  41:   8b 02                     mov    (%rdx),%eax
  43:   41 89 02                  mov    %eax,(%r10)
  46:   49 83 ea 04               sub    $0x4,%r10
  4a:   83 ee 01                  sub    $0x1,%esi
  4d:   89 0a                     mov    %ecx,(%rdx)
  4f:   75 bd                     jne    e
  51:   f3 c3                   repz retq
  53:   f3 c3                   repz retq
```