Chapter 3
Growth of Functions and Company

The notion of “the order of functions” captures the efficiency of algorithms, and also provides a way of comparing the relative performance of different algorithms for the same problem.

For example, when solving the sorting problem, once the input size, \( n \), becomes large enough, merge sort, with its \( \Theta(n \log n) \) running time, beats insertion sort, with its worst-case running time being \( \Theta(n^2) \).

**Question:** What does it mean?
Asymptotic behavior

When we look at the running time of those algorithms when input size grows larger, we are studying the asymptotic behavior of those algorithms.

In other words, we are concerned with how the running time of an algorithm goes up with the size of the input goes up without a bound. Usually, the most asymptotically efficient algorithm is the best choice among its peers.

The asymptotic behavior shows the long term behavior, but not what this algorithm does for a specific input size. This behavior is important when we consider the whole ninety feet.

For example, when we buy stock, we don’t care about its performance on a particular day, but that in a long term.
Asymptotic notations

To study the asymptotic behaviors of algorithms, we need some special notations, such as the ‘Θ’ notation that we have already seen.

These notations are defined in term of functions whose domains are the set of natural numbers, which represent the sizes of algorithm input.

For example, when we use $T(n)$ to describe the worst-case running time of the Insertion sort algorithm, we may say $T(n) = \Theta(n^2)$, where $n$ stands for the input size.

It is the time for us to define the notations for the the exact bound(Θ), the upper bound(O), and the lower bound(Ω) of a function.
The $\Theta$ notation

For a given function $g(n)$, we denote, by $\Theta(g(n))$, a set of functions.

$$\Theta(g(n)) = \{f(n) : \text{for some positive constant } c_1, c_2, \text{ and } n_0 \text{ such that for all } n \geq n_0, 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}.$$ 

Graphically speaking, $f(n) \in \Theta(g(n))$ if there exists positive integers $c_1$ and $c_2$ such that $f(n)$ can be sandwiched between $c_1g(n)$ and $c_2g(n)$, after a finite $n_0$. Sometimes, we simply write $f(n) = \Theta(g(n)).$

We say that $g(n)$ is an asymptotically tight bound for $f(n)$, since for large enough $n$, they can be identified with each other within a constant factor.
An example

To show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$, we have to find positive integers $c_1, c_2$ and $n_0$ such that for all $n \geq n_0 > 0$,

$$c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2.$$ 

We start by dividing both sides by $n^2$ to obtain

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2.$$ 

Since $n > 0$, $-\frac{3}{n} < 0$. If we let $c_2 \geq \frac{1}{2}$, we will have

$$\frac{1}{2} - \frac{3}{n} \leq \frac{1}{2} \leq c_2.$$
On the other hand, for \( n \geq 7 \), i.e., \( \frac{1}{n} \leq \frac{1}{7} \), we have

\[
\frac{1}{2} - \frac{3}{n} \geq \frac{1}{2} - \frac{3}{7} = \frac{1}{14}.
\]

Thus, if we let \( c_1 \leq \frac{1}{14} \), we will have, for \( n \geq 7 \),

\[
\frac{1}{2} - \frac{3}{n} \geq \frac{1}{14} \geq c_1.
\]

Combine the above together, we have that, for all \( n \geq 7 \), \( c_1 \leq \frac{1}{14} \), and \( c_2 \geq \frac{1}{2} \),

\[
c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2,
\]

namely,

\[
c_1 n^2 \leq \frac{1}{2} n^2 - 3n \leq c_2 n^2.
\]

In particular, setting \( n_0 = 7, c_1 = \frac{1}{14} \), and \( c_2 = \frac{1}{2} \), we have that \( \frac{1}{2} n^2 - 3n = \Theta(n^2) \), by definition.
An counter example

We can show that $6n^3 \neq \Theta(n^2)$, as follows. Just assume that is the case, i.e., for some positive integers $c_1, c_2,$ and $n_0$, for all $n \geq n_0 > 0$,

$$c_1n^2 \leq 6n^3 \leq c_2n^2.$$  

Then, we have that for all $n \geq n_0 > 0, n^3 \leq \frac{c_2}{6}n^2$, or $n \leq \frac{c_2}{6}$, which is to say, for all $n \geq n_0, n \leq \frac{c_2}{6}$.

Since both $n_0$ and $c_2$ are constants, there must exist $n_1 > n_0$, such that $n_1 > \frac{c_2}{6}$. But, the previous inequality leads to the fact that $n_1 \leq \frac{c_2}{6}$.

As a result, such an $n_1$ would be both no more than and strictly larger than $\frac{c_2}{6}$, which is a contradiction. Thus, the original assumption must be false.

**Homework:** Exercises 3.1-1 and 3.1-2.
Why can they be ignored?

We use such notation as $\Theta$ to ignore lower order terms and the coefficients of the highest order term.

1. The lower-order term in $f(n)$ can be ignored since they are insignificant for larger $n$.

Let $f(n) = cg(n) + dh(n)$, where the order of $h(n)$ is strictly lower than that of $g(n)$. To show $f(n) = \Theta(g(n))$, we have to construct the following inequality, for all $n \geq n_0$,

$$c_1 g(n) \leq f(n) (= cg(n) + dh(n)) \leq c_2 g(n).$$

We only need to let $c_1 = c$ and $c_2 = c + d$, where $d \geq 0$; or $c_1 = c - 1$ and $c_2 = c$, otherwise.
2. The coefficient of the highest order in \( f(n) \) can also be ignored since it will be absorbed by the constants.

If we can find \( c_1, c_2 \) and \( n_0 \) for the following inequality, for all \( n \geq n_0 \),

\[
c_1 g(n) \leq cf(n) \leq c_2 g(n),
\]
then, we can surely find \( c'_1, c'_2 \) and \( n'_0 \) for the following inequality, for all \( n \geq n'_0(= n_0) \),

\[
c'_1 g(n) \leq f(n) \leq c'_2 g(n).
\]
It is clear that \( c'_1 = c_1/c \), and \( c'_2 = c_2/c \).

The other direction also goes through. Thus

\[
cf(n) = \Theta(g(n)) \equiv f(n) = \Theta(g(n)).
\]

Thus, we can drop all the lower-order terms and take off the coefficient of the highest term when applying the \( \Theta \) notation.
An example

Consider any quadratic function $f(n) = an^2 + bn + c$, where $a(> 0), b$ and $c$ are constants. Throwing away the lower-order terms, and ignoring the coefficient of the term $n^2$, we come to the conclusion that

$$an^2 + bn + c = \Theta(n^2).$$

More formally, we may prove that if we take $c_1 = a/4, c_2 = 7a/4$ and $n_0 = 2 \max\{|b|/a, \sqrt{|c|/a}\}$, then for all $n \geq n_0$,

$$0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2.$$

In general, for any polynomial, $p_d(n), d \geq 0$,

$$p_d(n) = \sum_{i=0}^{d} a_i n^i,$$

we have that $p_d(n) = \Theta(n^d)$. In particular, we have that any constant $c (= p_0(n)) = \Theta(n^0) = \Theta(1)$.
The $O$ notation

The $\Theta$ notation provides an asymptotic tight bound. When we only need the upper bound, we use the ‘$O$’ notation. For a given function $g(n)$, we denote, by $O(g(n))$, be a set of functions.

$$O(g(n)) = \{f(n) : \text{for some positive constant } c, \text{ and } n_0 \text{ such that for all } n \geq n_0, 0 \leq f(n) \leq cg(n)\}.$$  

We use this notation to give an upper bound of a collection of functions, within a constant factor, in the long term.

For example, when taking $c = 3$ and $n_0 = 0$, we have that

$$3n^2 - 2 = O(n^2).$$

Notice that, technically, we have $3n^2 - 2 \in O(n^2).$
Difference between $\Theta$ and $O$

We note that $\Theta$ is a stronger notation than $O$, particularly, $f(n) = \Theta(g(n))$ implies that $f(n) = O(g(n))$. Thus, since, $an^2 + bn + c = \Theta(n^2)$, we immediately have that $an^2 + bn + c = O(n^2)$.

The $O$ notation is not a tight one, e.g., any linear function $an + b = O(n^2)$, just taking $c = a + b$, and $n_0 = 1$, since for all $n \geq 1, (an + b) \leq (a + b)n^2$. But, $an + b \neq O(n^2)$.

Since the $O$– notation provides an upper bound, when we use it to describe the worst-case running time of an algorithm we have a bound for every input: *Nothing would be worse, but something could be better.*

For example, when we say the worst-case running time of the insertion sort is $O(n^2)$, we mean that, for every input, its worst running time would be $n^2$. But it could be better, e.g., when the list is pre-sorted, it takes just $n$. 

12
On the other hand, when we use the $\Theta$ notation to describe the worst-case running time of an algorithm, it only applies to the worst case, but not to every case. Thus, it does not cover every input.

When we say the worst-case running time of the insertion sort is $\Theta(n^2)$, we only mean that for the worst case, e.g., for a completely reversed input, it will take that long, which is not true for the other cases.

For example, for an already sorted list, it takes $\Theta(n)$ time.
The $\Omega$ notation

Similarly, when we only need the lower bound, we use the $\Omega$ notation. For a given function $g(n)$, we denote, by $\Omega(g(n))$, be a set of functions.

$$\Omega(g(n)) = \{f(n) : \text{for some positive constant } c, \text{ and } n_0 \text{ such that for all } n \geq n_0, f(n) \geq cg(n) \geq 0\}.$$  

We use this notation to give a lower bound on a function, within a constant factor, in the long term: for all values $n$ to the right of $n_0$, the value of $f(n)$ is on or above $cg(n)$.

**Homework:** Exercises 3.1-3 and 3.1-4.
Relating $\Theta$, $O$ and $\Omega$

Based on the definitions of the three notations, $\Theta$, $O$ and $\Omega$, we immediately have the following result.

**Theorem 3.1** For any two functions $f(n)$ and $g(n)$, we have that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

For example, since, $an^2 + bn + c = \Theta(n^2)$, we immediately have that $an^2 + bn + c = O(n^2)$, and $an^2 + bn + c = \Omega(n^2)$.; and vice versa.

**Homework:** Exercises 3.1-5 and 3.1-6.
Tight bounds of insertion sort

Since the $\Omega$–notation describes a lower bound, when we use it to describe the best-case running time of an algorithm, it takes care of every input.

For example, the best-case running time of insertion sort is $\Omega(n)$ implies that the running time of insertion sort, when applied to any input, is $\Omega(n)$.

To summarize, the running time of insertion sort is between $\Omega(n)$ and $O(n^2)$. 
An important usage of...

When we have asymptotic notations in the middle of an expression, we interpret it as something we really don’t care that much about its precision. For example, the expression

\[ 2n^2 + 3n + 1 = 2n^2 + \Theta(n), \]

really means

\[ 2n^2 + 3n + 1 = 2n^2 + f(n), \]

and \( f(n) \) is some function in the set of \( \Theta(n) \).
Why this usage?

It helps us to clean up things and get rid of inessential details.

For example, when we analyzed the behavior of merge sort, we come up with the following recurrence equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).$$

In fact, if we are only interested in the asymptotic behavior of merge sort, we need not list all the lower-order terms exactly, we just need to collect them in the expression $\Theta(n)$. 
Two “little” notations

As we saw, the asymptotic upper bound and lower bound may or may not be tight. For example, $O(n^2)$ is a tight upper bound for $2n^2$, but not for $2n$. We use the $o$–notation to indicate an upper bound that is not tight.

Formally, we have that, for a given function $g(n)$, we denote, by $o(g(n))$, be a set of functions with their orders strictly less than that of $g(n)$.

$$o(g(n)) = \{ f(n) : \text{for any positive constant } c, \text{ and } n_0 \text{ such that for all } n \geq n_0, 0 \leq f(n) < cg(n) \}.$$  

Thus, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$. 
**O vs. o**

They are similar, with the difference being, when \( f(n) = O(g(n)) \), the inequality \( 0 \leq f(n) \leq cg(n) \) holds for some constant \( c \); while when \( f(n) = o(g(n)) \), the inequality \( 0 \leq f(n) < cg(n) \) holds for all constants \( c \).

Thus, when \( n \) approaches infinity, \( f(n) \) becomes insignificant, namely,

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.
\]

Since the \( o \)-notation is stronger in the sense that if \( f(n) = o(g(n)) \) (\( a < b \Rightarrow a \leq b \)) then immediately \( f(n) = O(g(n)) \), we can use the above test to find out the \( O \)-notation of a function.
What is the upper bound ‘O’?

To show that \( f(n) = O(g(n)) \), we go through the following process:

1. 
\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0,
\]

2. The above means \( f(n) = o(g(n)) \).

3. The above result implies \( f(n) = O(g(n)) \).

For example, to show \( 3n - 2 = O(n^2) \), we only need to show that
\[
\lim_{n \to \infty} \frac{3n - 2}{n^2} = \lim_{n \to \infty} \frac{3}{2n} = 0.
\]

Thus, \( 3n - 2 = o(n^2) \), i.e., \( 3n - 2 = O(n^2) \).

The process that finds out the lower bound is similar.
\Omega \text{ vs. } \omega

Similarly, we use the $\omega$–notation to indicate a lower bound that is not tight.

Formally, we have that, for a given function $g(n)$, we denote, by $\omega(g(n))$, be a set of functions.

$$
\omega(g(n)) = \{ f(n) : \text{for any positive constant } c, \text{ and } n_0 \text{ such that for all } n \geq n_0, 0 \leq cg(n) < f(n) \}.
$$

Thus, $\frac{n^2}{2} = \omega(n)$, but $2n^2 \neq \omega(n^2)$.

Still remember L'Hospital’s rule?

When $f(n), g(n) \to 0(\infty)$ as $n \to \infty$,

$$
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}.
$$
Going back to Calculus...

We can define the $\omega$-notation in term of the $o$–notation as follows: $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

We also have that $f(n) \in \omega(g(n))$ if

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

We can use it to confirm that $f(n) = \Omega(g(n))$.

For example, since

$$
\lim_{n \to \infty} \frac{\log n}{n^2} = \lim_{n \to \infty} \frac{1}{2n^2} = 0,
$$

we immediately have that

$$
\log n = o(n^2), \text{ i.e., } \log n = O(n^2)
$$

and equivalently,

$$
n^2 = \omega(\log n), \text{ i.e., } n^2 = \Omega(\log n).
$$

**Homework:** Exercise 3.1-7.
Mergesort is better

We know in the previous chapter that, to sort a list of size $n$, insertion sort takes $\Theta(n^2)$, and merge sort takes $\Theta(n \log n)$.

Since

$$
\lim_{n \to \infty} \frac{n \log n}{n^2} = \lim_{n \to \infty} \frac{\log n + 1}{2n} = \lim_{n \to \infty} \frac{1}{2n} = 0,
$$

thus,

$$n \log n = o(n^2).$$

We thus may conclude that merge sort is faster than insertion sort in the long run.
Practical speaking...

When predicting the long term behaviors of various programs, we go through the following steps:

1. Find out the running time, as functions of $n$, of various algorithms (loops, recurrence relation, etc.).

2. Simplify the running time to one of those simple functions, using mainly the $O$, and $\Theta$, notations (The tricks that we just went through)

3. Compare the growth rates of those simple functions, using the following order: $c$, $\log^k(n)$ ($k \geq 1$) (Cf. the forthcoming pp 35), $n$, $n \log(n)$, $n^2$, $n^3$ and $2^n$.

4. The one with a less rate wins.
Why the order in step 3?

The following picture shows why we put the time complexities of various program that way.
Why is $2^n$ bad?

From a practical point of view, for reasonably large $n$, only programs of small complexity, such as $n, n \log n, n^2$ and $n^3$, are feasible, even if we have a computer that can execute $10^{12}$ instructions per second.

For example, when $n = 50$, it would take 3.17 years to execute $n^{10}$ instructions, and $4 \times 10^{10}$ years to execute $2^n$ instructions.

Below shows how long it takes such a computer to solve a problem with $n$ inputs, when an algorithm has to execute $f(n)$ instructions.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^{10}$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10ps</td>
<td>.03μs</td>
<td>.1ns</td>
<td>10s</td>
<td>1μs</td>
</tr>
<tr>
<td>20</td>
<td>20ps</td>
<td>.09μs</td>
<td>40ns</td>
<td>2.84h</td>
<td>1ms</td>
</tr>
<tr>
<td>50</td>
<td>50ps</td>
<td>.28ns</td>
<td>2.5ns</td>
<td>3.17y</td>
<td>13d</td>
</tr>
<tr>
<td>100</td>
<td>.1ns</td>
<td>.66ns</td>
<td>10ns</td>
<td>3171y</td>
<td>$4 \times 10^{13}$y</td>
</tr>
<tr>
<td>$10^4$</td>
<td>10ns</td>
<td>0.13 ns</td>
<td>100ms</td>
<td>$3 \times 10^{23}$y</td>
<td>N/A</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1ms</td>
<td>20ms</td>
<td>17m</td>
<td>$3 \times 10^{43}$y</td>
<td>N/A</td>
</tr>
</tbody>
</table>
About the stop watch...

Besides theoretically analyzing a program, we can also measure the performance via experiments. Below shows an example in C++:

```c++
#include <time.h>
#include "insort.h"

void main(void){
    int a[1000], step = 10;
    clock_t start, finish;
    for (int n = 0; n <= 1000; n += step) {
        for (int i = 0; i < n; i++)
            a[i] = n - i; // initialize
        start = clock();
        InsertionSort(a, n);
        finish = clock();
        cout<<n<<' '<<(finish-start)/float(CLK_TCK)<<endl;
    }
}

Question: Ready to work on Project 3?
Relationship between functions

Many of the relational properties of real numbers apply to asymptotic comparisons as well. Assume that \( f(n) \) and \( g(n) \) are asymptotically positive, we have that

1. (**Transitivity**) If \( f(n) = \Theta(g(n)) \) and \( g(n) = \Theta(h(n)) \), then \( f(n) = \Theta(h(n)) \), where \( \Theta \) can be any of the five notations.

2. (**Reflexivity**) \( f(n) = \Theta(f(n)) \), where \( \Theta \) can be either \( \Theta \), \( O \) or \( \Omega \).

3. (**Symmetry**) \( f(n) = \Theta(g(n)) \) if and only if \( g(n) = \Theta(f(n)) \).

4. (**Transpose symmetry**) \( f(n) = O(g(n)) \) if and only if \( g(n) = \Omega(f(n)) \). The same thing exists between \( o \) and \( \omega \).
What can we talk about them?

We can also draw an analog between the asymptotic notations and the arithmetic inequalities: $O, \Omega, \Theta, o$ and $\omega$ can be interpreted as $\leq, \geq, =, <, >$.

For example,

$$f(n) = O(g(n)) \approx f(n) \leq g(n).$$

We thus say that $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = O(g(n))$.

However, the nice property of trichotomy among real numbers, i.e., between any two numbers $a$ and $b$, either $a < b, a = b$, or $a > b$ does not hold for asymptotic notations. For example, the pair of functions $n$ and $n^{1+\sin(n)}$ can’t be compared, since for an arbitrary $n$, $1 + \sin(n)$ could take any value in between 0 and 2.
Monotonicity

A function $f$ is called \textit{monotonically increasing} if

$$m \leq n \text{ implies } f(m) \leq f(n),$$

\textit{monotonically decreasing} if

$$m \leq n \text{ implies } f(m) \geq f(n).$$

A function $f$ is called \textit{strictly increasing} if

$$m < n \text{ implies } f(m) < f(n).$$

Finally, A function $f$ is called \textit{strictly decreasing} if

$$m \leq n \text{ implies } f(m) > f(n).$$

For example, the constant function $f(n) = c$ is both monotonically increasing and decreasing, but neither strictly increasing nor decreasing.

On the other hand, $f(n) = n^2$ is strictly increasing, while the function $f(n) = 100 - n$ is strictly decreasing.
Floor and ceiling

For any real number $x$, we denote the greatest integer less than or equal to $x$ by $\lfloor x \rfloor$, the floor of $x$; and the least integer that is greater than or equal to $x$ by $\lceil x \rceil$, the ceiling of $x$.

Thus, for all $x$ the following is true.

$$x - 1 < \lfloor x \rfloor \leq x \leq \lfloor x \rfloor < x + 1.$$ 

For example, $\lfloor 3.4 \rfloor = 3$, but $\lfloor -3.4 \rfloor = -4$.

For any integer $n$, $\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n$.

Finally, the floor function is monotonically increasing, as is the ceiling function.

Question: Are they strictly increasing?

Homework: 3.2-1. Think about relations 3.4–3.7 as given in page 54.
Modular arithmetic

For any integer $a$ and any positive integer $n$, the value $a \mod n$ is the remainder of the quotient $a/n$:

$$a \mod n = a \% b \text{ (in Java)} = a - \left\lfloor \frac{a}{n} \right\rfloor n.$$

For example,

$$8 \mod 3 = 8 - \left\lfloor \frac{8}{3} \right\rfloor 3 = 8 - 6 = 2.$$

If $a \mod n = b \mod n$, we write $a \equiv b \pmod{n}$ and say that $a$ is equivalent to $b$, modular $n$, which essentially means that when dividing them using $n$, they end up with the same remainder, e.g., $8 \equiv 14 \pmod{3}$. 

33
Given a non-negative integer $d$, a polynomial in $n$ of degree $d$ is a function $p_d(n)$ of the form

$$p_d(n) = \sum_{i=0}^{d} a_in^i,$$

where $a_i$'s, $i \in [0, d]$, $a_d \neq 0$, are the coefficients of $p$.

A polynomial $p_d(n)$ is asymptotically positive if and only if $a_d > 0$, when $p_d(n) = \Theta(n^d)$.

For any real constant $a \leq 0$, the function $n^a$ is monotonically decreasing.

Finally, we say that $f(n)$ is polynomially bounded if for some constant $k$,

$$f(n) = O(n^k).$$
For all real \( a > 0, m, \) and \( n, \) the following identities always hold:

\[
\begin{align*}
a^0 &= 1, \\
a^1 &= a, \\
a^{-1} &= \frac{1}{a}, \\
(a^m)^n &= a^{mn}, \\
a^{mn} &= a^{nm}, \\
a^m a^n &= a^{m+n}.
\end{align*}
\]

For all \( n \) and \( a \geq 1, \) the function \( a^n \) is monotonically increasing in \( n. \)

Since for all real constants \( a(>1) \) and \( b, \)

\[
\lim_{n \to \infty} \frac{n^b}{a^n} = \lim_{n \to \infty} \frac{b!}{a^n \log^b a} = 0,
\]

we have that \( n^b = O(a^n). \) Thus, any exponential function with a base strictly larger than 1 grows faster than any polynomial function.
exponential function

Using $e = (2.71828\cdots)$ we have that for all real $x$,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} \frac{x^i}{i!}.$$  

In particular, we have that for all real $x$,

$$e^x \geq 1 + x,$$

where the equality holds when $x = 0$.

We also have that, for all real $x$,

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right)^n = e^x.$$
Logarithms

For all real $a > 0, b > 0, c > 0$ and $n$, the following identities always hold:

\[
\begin{align*}
    a &= b^{\log_b a}, \\
    \log_c(ab) &= \log_c a + \log_c b, \\
    \log_b(a^n) &= n \log_b a, \\
    \log_b a &= \frac{\log_c a}{\log_c b}, \text{ (Who cares about base?)} \\
    \log_b \left( \frac{1}{a} \right) &= -\log_b a, \\
    \log_b a &= \frac{1}{\log_a b}, \\
    a^{\log_b c} &= c^{\log_b a}.
\end{align*}
\]

**Homework:** Exercise 3.2-2.
We just saw that
\[ a^{\log_b c} = c^{\log_b a}. \]

We can prove this equality in at least two ways:

1. Let \( x = \log_b c \) and let \( y = \log_b a \), we have that \( c = b^x \), and \( a = b^y \). Then
\[
a^{\log_b c} = a^x = (b^y)^x = b^{yx} = b^{xy} = (b^x)^y = c^y = c^{\log_b a}.
\]

2. Let \( x = \log_b c \). Thus, \( c = b^x \). Thus
\[
c^{\log_b a} = (b^x)^{\log_b a} = b^{x \log_b a} = b^{\log_b a^x} = a^x = a^{\log_b c}.
\]
Logarithm is really slow.

We say that $f(n)$ is polylogarithmically bounded if for some constant $k$,

$$f(n) = O(\ln^k n),$$

where $\ln n \equiv \log_e(n)$. (Cf. Page 56 of the textbook)

Since for all real constants $a(>1)$ and $b$,

$$\lim_{n \to \infty} \frac{\ln^b n}{n^a} = \lim_{n \to \infty} \frac{b \ln^{b-1} n}{an^a} = \cdots = \lim_{n \to \infty} \frac{b!}{a^n n^a} = 0,$$

we have that $\ln^b n = o(n^a)$. By the same token, for any $c$, $\log_c^b n = o(n^a)$.

Thus, any positive polynomial function grows faster than any polylogarithmic function.

If we come up with an algorithm with a logarithmic run time, we should be really happy.
An example: Binary search

Given an integer \( x \) and a list of integers \( a_1, a_2, \ldots, a_n \), which are pre-sorted and already in memory, find \( i \) such that \( a_i = x \), or return \( i = 0 \) if \( x \) is not in the input.

This problem can be solved by using binary search.

```c
int BinarySearch(int A, int x, int n)
  // Return position if found; -1 otherwise.
  int left = 1; int right = n;
  while (left <= right) {
    int middle = (left + right)/2;
    if (x == A[middle]) return middle;
    if (x > A[middle]) left = middle + 1;
    else right = middle - 1;
  }
  return -1; // x not found
}
```
How does it work?

The gist is that if $x$ is in $A[\text{left}..\text{right}]$ and $x \neq A[\text{middle}]$, then if $x > A[\text{middle}]$, $x$ must be in $A[\text{middle}+1, \text{right}]$; otherwise, it must be in $A[\text{left}, \text{middle}-1]$.

After repetitive cuts, either we find $x$ or we will end up with an empty segment. In the latter case, $x$ could not be in the original list, and we declare failure and return -1.
Algorithm analysis

We first show that what’s left after each repetition is strictly less than half of the size that it starts with.

Let $l = m$, $h = n$. The size of the original list is $n - m + 1$, and $mid = (l + h)/2$.

Case (i): $m + n = 2k$, i.e., $mid = k$.

- size of the lower part:

$$size([l, mid - 1]) = mid - 1 - l + 1 = k - m.$$  

As, $2(k - m) = m + n - 2m = n - m < n - m + 1,$

$$|l, mid - 1| < \frac{n - m + 1}{2} = \frac{1}{2}|l, h|.$$
• size of the upper part: Similarly, we can find out that

$$|\text{mid} + 1, h| \leq \frac{1}{2}|l, h|.$$  

Case (ii): $m + n = 2k + 1$, so $\text{mid} = k$. Similar analysis shows the same result.

Therefore, what’s left is always no more than half of the original list.

Thus, the maximal sizes of a list that we have to work on after each loop are as follows: $\frac{n}{2}$, $\frac{n}{4}$, $\ldots$, $\frac{n}{2^k}$, $\ldots$, $1$.

Since $\frac{n}{2^k} \geq 1$, $k$, the number of loops, cannot be more than $\log_2(n)$.

As the computation within each loop can be done with $O(1)$ time, the total time of the binary search is in $O(\log_2(n))$. 

43
Factorials

The well-known factorial function is defined as follows:

\[
1! = 1, \\
\frac{n!}{n} = n \times (n - 1)!.
\]

Thus, \(n! = 1 \times 2 \times \cdots n\).

It is immediate that \(n! \leq n^n\). (7) The following Stirling's approximation provides a much tighter bound:

\[
n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right).
\]
A little justification

\[ \log n! = \sum_{j=1}^{n} \log j \approx \int_{1}^{n} \log x \, dx \]
\[ = [x \log x - x]|_{1}^{n} \approx n \log \left( \frac{n}{e} \right) \]
\[ = \log \left( \frac{n}{e} \right)^{n} = \log n^{n}e^{-n}. \]

Thus, we have \( n! \approx n^{n}e^{-n} \). Indeed

\[
\begin{array}{|c|c|c|c|c|}
\hline
n & 1 & 10 & 100 & 1000 \\
\hline
\frac{n!}{n^{e^{-n}\sqrt{2\pi n}}} & 1.084437 & 1.008365 & 1.000833 & 1.000083 \\
\hline
\end{array}
\]

Based on this result, we can prove that

\[ n! = o(n^n), \]
\[ n! = \omega(2^n), \]
\[ \log(n!) = \Theta(n \log n). \]

Hence, permutation grows up pretty fast.
Fibonacci numbers

The Fibonacci numbers are defined with the following recurrence:

\[ F(0) = 0, \]
\[ F(1) = 1, \]
\[ F(n) = F(n - 1) + F(n - 2), \quad n \geq 2. \]

Thus, it leads to the sequence 0, 1, 1, 2, 3, 5, 8, 13, \ldots.

Let \( \phi \) and \( \phi' \) denote \( \frac{1 + \sqrt{5}}{2} \) and \( \frac{1 - \sqrt{5}}{2} \), respectively, we have that

\[ F(n) = \frac{\phi^n - \phi'^n}{\sqrt{5}}. \]

Since \( |\phi'| < 1 \), we have that

\[ \frac{|\phi'|^n}{\sqrt{5}} < \frac{1}{\sqrt{5}} < \frac{1}{2}. \]

Hence, we have that \( F(n) = \frac{\phi^n}{\sqrt{5}} \), rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially.

**Homework:** Exercises 3.2-6 and 3.2-7.