Chapter 1–Complexity Analysis,

Source: https://www.freecodecamp.org/news/asymptotic-analysis-explained-with-pokemon-a-deep-dive-into-complexity-analysis-8bf4396804e0/

Algorithms can be understood and studied in a language- and machine-independent manner

Tools:

1. RAM model of computation
2. Asymptotic analysis

RAM Model

Random Access Machine (RAM) model

- A hypothetical model
- All simple operations (call, if, $+$, $-$, $\times$, $=$), and memory-access take exactly one time step
- Instructions are executed one after another, one operation at a time

. . . .

→ suitable for desining and analyzing of **sequential algorithms**

→ Operation count converts naturally to the actual running time

Complaints about RAM: too simple

- $\times$ more time than $+$
- Not considering multi-threading → not suitable for **parallel algorithms**
- memory access times differ greatly depending on whether data sits in cache or on the disk → not suitable for **big data algorithms**

But, RAM proves an excellent model for understanding how an algorithm will perform on a real computer.

- Like the flat earth model

Time Complexity

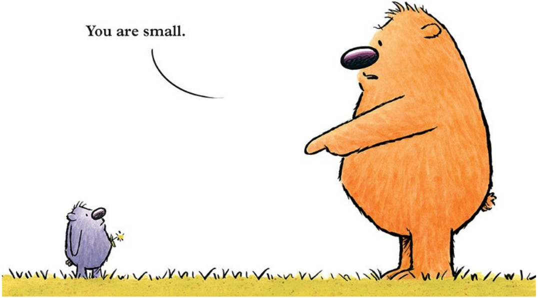
Asymptotic Analysis Approach

Time Complexity Analysis: Counting the number of times some basic operation is done as a function of the size of the input.

Basic Operation:

- Considering every operation: difficult, depends on the way the programmer writes the program, and $\underbrace{\text{usually unnecessary}}_{\textbf{order of growth is important}}$

→ identify the most important operation (or a set of operations) of the algorithm, called the *basic operation*, the operation contributing the most to the total running time.



Source https://www.freecodecamp.org/news/asymptotic-analysis-explained-with-pokemon-a-deep-dive-into-complexity-analysis-8bf4396804e0/

How to Find the Basic Operation?

- No hard-and-fast rule for choosing the basic operation. It is largely a matter of judgment and experience
- we ordinarily do not include the instructions that make up the control structure
- Usually the most time-consuming operation is in the algorithm's innermost loop. Example: comparison in sorting algorithms

At times we may want to consider two different basic operations

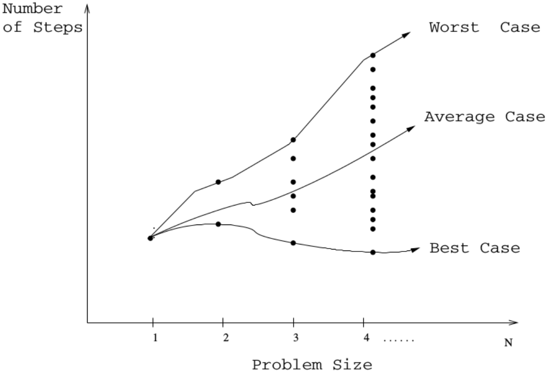
- comparison instruction and the assignment instruction in sorting algorithms
- Not meaningful in RAM model, but maybe of interest

Input Size:

- For many algorithms it is easy to find a reasonable measure of the size of the input, which we call the input size.
    - In sorting, searching, and adding all members of an array, the number of items in the array is the input size
- In some algorithms, it is more appropriate to measure the size of the input using two numbers.
    - For example, when the input to an algorithm is a graph, we often measure the size of the input in terms of both the number of vertices and the number of edges

What About the Input Value?

- In some cases the time complexity (number of operations) depends not only on the input size, but also on the input's values.



Time Complexity

□ Every-case

□ Worst-case

□ Best-case

□ Average

Every-Case Time Complexity; $T(n)$

the basic operation is always done the same number of times for every instance of size n.

Algorithm 1.2

Problem: Add all the numbers in the array S of n numbers.

Inputs: positive integer n, array of numbers S indexed from 1 to n.

Outputs: sum, the sum of the numbers in S.

```
number sum (int n, const
number S[ ])
{ index i;
number result;
result = 0;
for (i = 1; i<= n;i++)
    result = result + S[i];
return result; }
```

Basic operation: the addition of an item in the array to `result`.

Input size: n, the number of items in the array.

- Regardless of the values of the numbers in the array, there are n passes through the for loop. Therefore, the basic operation is always done n times and

$$T(n) = n$$

Algorithm 1.3 (Exchange Sort)

Problem: Sort n keys in nondecreasing order.
Inputs: positive integer n, array of keys S indexed from 1 to n.
Outputs: the array S containing the keys in nondecreasing order.

```
void exchangesort (int n, keytype S[])
{ index i, j;
for(i = 1;i <= n - 1;i + +)
   for(j= i + 1;j<= n;j + +)
      if(S[j]<S[i])
         exchange S[i] and S[j];
}
```

Basic operation: the comparison of `S[j]` with `S[i]`.
Input size: n, the number of items to be sorted.

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 1 = \frac{(n-1)n}{2}$$

Algorithm 1.4 (Matrix Multiplication)

Problem: Determine the product of two n × n matrices.

Inputs: a positive integer n, two-dimensional arrays of numbers A and B, each of which has both its rows and columns indexed from 1 to n.

Outputs: a two-dimensional array of numbers C, which has both its rows and columns indexed from 1 to n, containing the product of A and B.

```
void matrixmult (int n, const number A[ ][ ], const number B[][], number C[][])
{ index i, j, k;
for (i = 1;i <= n;i++)
   for(j= 1;j <= n;j++){
      C[i][j] = 0;
      for (k = 1;k <= n;k++)
         C[i][j] = C[i][j] + A[i][k]*B[k][j];
   }
}
```

Basic operation: multiplication instruction in the innermost for loop.
Input size: n, the number of rows and columns.

$$T(n) = n \times n \times n = n^3$$

Worst-Case, Average-Case, and Best-Case Time Complexity; $W(n), A(n), B(n)$

- In some cases the number of times the basic operation is done depends not only on the input size, but also on the input's values.
- The basic operation is not done the same number of times for all instances of size n.
- Such an algorithm does not have an every-case time complexity
- However, this does not mean that we cannot analyze such algorithms, because there are three other analysis techniques that can be tried.
  1. Worst-Case: Maximum
  2. Average-Case: Average (expected value)
  3. Best-Case: Minimum

     number of times the algorithm will ever do its basic operation for an input size of n

Algorithm 1.1 (Sequential Search)

Problem: Is the key x in the array S of n keys?

Inputs: positive integer n, array of keys S indexed from 1 to n, and a key x.

Outputs: *location*, the location of x in S (0 if x is not in S).

```
void seqsearch (int n, const keytype S[ ], keytype x,
index& location)
{
location = 1;
while (location<=n && S[location] != x)
   location ++;
if (location> n)
   location = 0;
}
```

Basic operation: the comparison of an item in the array with x.

Input size: n, the number of items in the array.

$$B(n) = 1$$

$$W(n) = n$$

$$A(n) = ?$$

To calculate $A(n)$, we assume that x is in S with probability p, and all items of S are distinct and equi-probable to hit x. Then we can write

$$\begin{aligned}
A(n) &= E\{T(n)\} \\
&= E\{T(n)|x \in S\}Pr\{x \in S\} + E\{T(n)|x \notin S\}Pr\{x \notin S\} \\
&= E\{T(n)|x \in S\}p + n(1-p) \\
&= \left(\frac{1}{n} \times 1 + \frac{1}{n} \times 2 + \cdots \frac{1}{n} \times n\right) p + n(1-p) \\
&= \frac{p(n+1)}{2} + n(1-p) \\
&= \boxed{n(1 - \frac{p}{2}) + \frac{p}{2}}
\end{aligned}$$

Why Asymptotic Analysis?

It is very difficult to work precisely with time complexity functions

- Many bumps
- Many details needed
- After all, it may be meaningless! (At different levels of computational abstraction, the notion of an operation may grow or shrink by a constant factor)

On the other hand,

- Algorithms show their true efficiencies in large input sizes. And this is the **order of growth** which dictates the behavior of the function for large n (**asymptotic behavior**)
- We'd actually like to classify running times at a coarser level of granularity so that similarities among different algorithms, and among different problems, show up more clearly.

For all these reasons, we want to express the growth rate of running times and other functions in a way that is insensitive to constant factors and low-order terms.

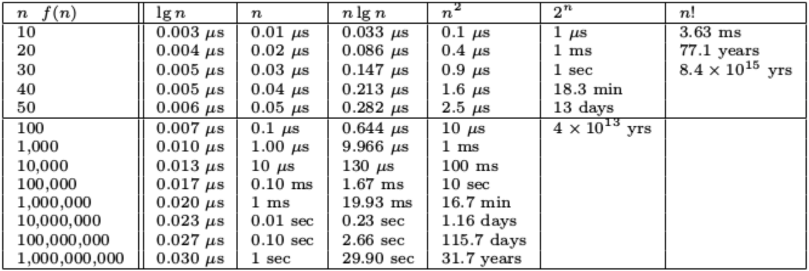
we'd like to be able to take a running time like $1.62n^2 + 3.5n + 8$, and say that it grows like n^2, up to constant factors.

Informal way of specifying the order of growth:

- Ignore the constant factors and keep the term with the highest order of growth using

$n! \gg b^n \gg a^n \gg n^j \gg n^k \gg n\log n \gg n \gg \log n \gg 1$

for any $(b > a > 1,\ j > k > 1)$



$n \quad f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Figure 1: Assuming a fast computer that takes 1 nanosecond (10^{-9} second) to process each operation

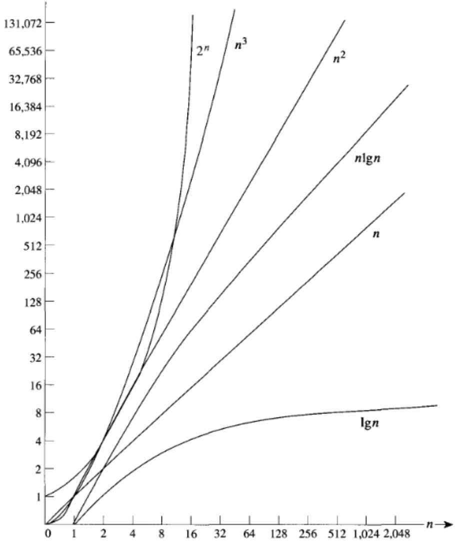
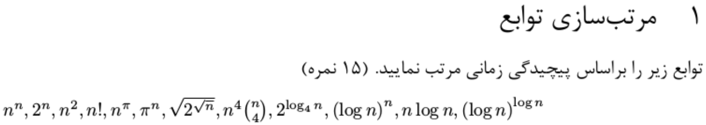


Figure 2: Growth rates of some common complexity functions.

Complexity Categories:
The set of all functions such as $g(n)$ which are reduced to the same reference function $f(n)$, is denoted by $\Theta(f(n))$ and we can say that the order of $g(n)$ is the same as the order of $f(n)$.

$\Theta(1)$	Constant
$\Theta(\log n)$	Logarithmic
$\Theta(n)$	Linear
$\Theta(n \log n)$	Linearithmic
$\Theta(n^2)$	Quadratic
$\Theta(n^3)$	Cubic
$\Theta(2^n)$	Exponential
$\Theta(n!)$	Factorial



۱ مرتب‌سازی توابع

توابع زیر را براساس پیچیدگی زمانی مرتب نمایید. (۱۵ نمره)

$$n^n, 2^n, n^2, n!, n^\pi, \pi^n, \sqrt{2^{\sqrt{n}}}, n^4\binom{n}{4}, 2^{\log_4 n}, (\log n)^n, n\log n, (\log n)^{\log n}$$

O, Ω, and Θ notations

Informal Definitions

O-notation: characterizes an upper bound on the asymptotic behavior of a function based on the highest-order term

$f(n) = 7n^3 + 100n^2 - 20n + 6 \rightarrow$ highest order term: $7n^3 \rightarrow$
$f(n) \in O(n^3), O(n^4), O(n^c), c \geq 3$

Ω-notation: characterizes a lower bound on the asymptotic behavior of a function based on the highest-order term

$f(n) = 7n^3 + 100n^2 - 20n + 6 \rightarrow$ highest order term: $7n^3 \rightarrow$
$f(n) \in \Omega(n^3), \Omega(n^2), O(n^c), c \leq 3$

Θ-notation: characterizes a tight bound on the asymptotic behavior of a function based on the highest-order term

$f(n) = 7n^3 + 100n^2 - 20n + 6 \rightarrow$ highest order term: $7n^3 \rightarrow f(n) \in \Theta(n^3)$

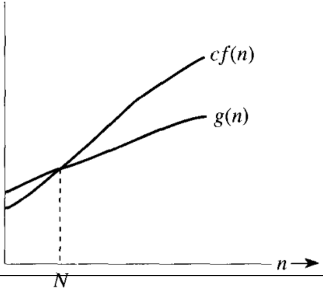
Formal Definitions

Definition (**Big O: Asymptotic Upper Bounds**)

For a given complexity function $f(n)$, $0(f(n))$ is the **set** of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$

$$g(n) \leq c \times f(n).$$

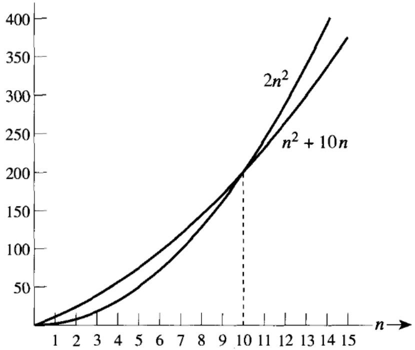
If $g(n) \in O(f(n))$, we say that $g(n)$ is big O of $f(n)$ or **asymptotically upper-bounded** by $f(n)$.



→ Although $g(n)$ starts out above $cf(n)$, eventually it falls beneath $cf(n)$ and stays there.

Example

$n^2 + 1On \le 2n^2, n \ge 10 \ \rightarrow n^2 + 1On \in O(n^2)$.



Note that c and N are not unique. In the above example, one can prove $n^2 + 1On \in O(n^2)$ by writing:

$n^2 + 1On \le n^2 + 1On^2 = 11n^2, n \ge 0$

Example

Consider $g(n) = pn^2 + qn + r$ with some positive p, q, and r. We can write

$$g(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = \underbrace{(p + q + r)}_{c} n^2, \text{ for } n \geq 0$$

$\rightarrow g(n) \in O(n^2)$

A function can have many upper bounds

- Note that $O(\cdot)$ expresses only an upper bound, not the exact growth rate of the function. For the example, above, we can continue as $g(n) \leq (p+q+r)n^2, n^2 \leq n^3 \rightarrow g(n) \leq (p+q+r)n^3 \rightarrow g(n) \in O(n^3)$.
- There are cases, where an algorithm has been proved to have running time $O(n^3)$; some years pass, people analyze the same algorithm more carefully, and they show that in fact its running time is $O(n^2)$.

Example

We show that $n^2 \in O(n^2 + 10n)$.
Because, for $n \geq 0$, $n^2 \leq 1 \times (n^2 + 10n)$, we can take $c = 1$ and $N = 0$ to obtain our result.

- The purpose of this example is to show that the function inside "big O" does not have to be one of the simple functions plotted in Figure 2. It can be any complexity function. Ordinarily, however, we take it to be a simple function like those plotted in Figure 2.

In Class Question 1

If someone says

$$\mathit{The\ running\ time\ of\ Algorithm\ is\ at\ least\ O(n^2)}$$

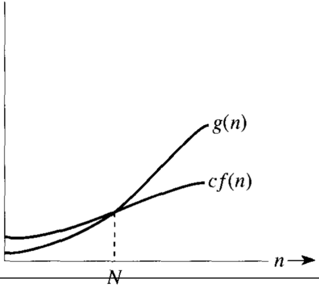
Then what do you say?

Definition (Ω: **Asymptotic Lower Bounds**)

For a given complexity function $f(n)$, $\Omega(f(n))$ is the **set** of complexity functions $g(n)$ for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$

$$g(n) \geq c \times f(n).$$

If $g(n) \in \Omega(f(n))$, we say that $g(n)$ is big omega of $f(n)$ or **asymptotically lower-bounded** by $f(n)$.



→ Although $g(n)$ starts out below $cf(n)$, eventually it goes above $cf(n)$ and stays there.

Example

Again consider $g(n) = pn^2 + qn + r$ with some positive p, q, and r. We can claim that $g(n) \in \Omega(n^2)$, because
$g(n) = pn^2 + qn + r \geq pn^2 \rightarrow g(n) \in \Omega(n^2)$

- Just as we discussed the notion of "tighter" and "weaker" upper bounds, the same issue arises for lower bounds. For example, it is correct to say that our function $g(n) = pn^2 + qn + r \in \Omega(n)$, since $g(n) \geq pn^2 \geq pn$.

Example

We show that n is not in $\Omega(n^2)$
Proof by contradiction:
Suppose $n \in \Omega(n^2)$, this means that there exist constants c and N such that
$\forall n \geq N : n \geq c \times n^2$
but $n \geq c \times n^2 \rightarrow n \leq \frac{1}{c}$ which contradicts with the $\forall n \geq N$ assumption $\Rightarrow\!\Leftarrow$

Definition (Θ: **Asymptotic Tight Bounds**)

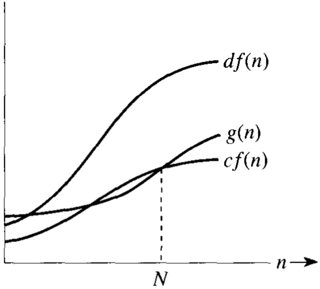
For a given complexity function $f(n)$,

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

This means that $\Theta(f(n))$ is the **set** of complexity functions $g(n)$ for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$,

$$c \times f(n) \leq g(n) \leq d \times f(n).$$

If $g(n) \in \Theta(f(n))$, we say that $g(n)$ is order of $f(n)$ or **asymptotically tight-bounded** by $f(n)$.

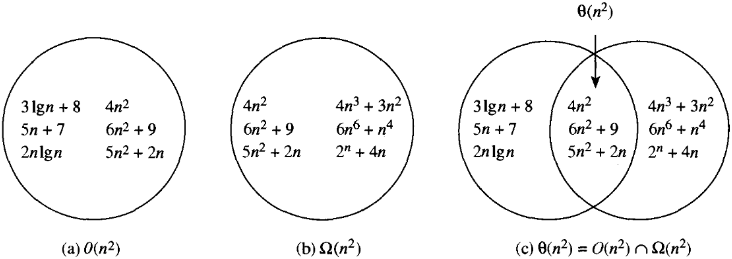


→ Although $g(n)$ starts not fitting between $cf(n)$ and $df(n)$, eventually it lies between $cf(n)$ and $df(n)$, and stays there.

Example

Consider $g(n) = pn^2 + qn + r$, We showed that $g(n) \in O(n^2)$ and $g(n) \in \Omega(n^2) \rightarrow g(n) \in \Theta(n^2)$

Example



Theorem

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c,\ c > 0 \Rightarrow g(n) \in \Theta(f(n))$$

Proof.

Based on the definition of a limit, there is some n_0 beyond which the ratio is always between $c/2$ and $2c$. Thus, $g(n) \leq 2cf(n)$ for all $n \geq n_0$, which implies that $g(n) = O(f(n))$; and $g(n) \geq cf(n)/2$ for all $n \geq n_0$, which implies that $g(n) = \Omega(f(n))$. □

Definition (small o)

For a given complexity function $f(n)$, $o(f(n))$ is the **set** of all complexity functions $g(n)$ satisfying the following: For every positive real constant c there exists a nonnegative integer N such that, for all $n \geq N$,

$$g(n) < c \times f(n)$$

If $g(n) \in o(f(n))$, we say that $g(n)$ is small o of $f(n)$. We use o-notation to denote an upper bound that is **not asymptotically tight**.

- Recall that "big O" means there must be **some** real positive constant c for which the bound holds. This definition says that the bound must hold for **every** real positive constant c.

Example

We show that $n \in o(n^2)$.
Let $c > 0$ be given. We need to find an N such that, for $n \geq N$,

$$n \leq cn^2 \rightarrow \frac{1}{c} \leq n$$

Therefore, it suffices to choose any $N \geq 1/c$.

Notice that the value of N depends on the constant c. For example, if $c = 0.00001$, we must take N equal to at least $100,000$. That is, for $n \geq 100,000$, $n \leq 0.00001\, n^2$.

Theorem

If $g(n) \in o(f(n))$ *then*

$$g(n) \in O(f(n)) - \Omega(f(n))$$

Proof.

$g(n) \in o(f(n)) \rightarrow \forall c, \exists N, g(n) \leq cf(n) \rightarrow g(n) \in O(f(n))$
On the other hand, We will show that $g(n)$ is not in $\Omega(f(n))$ using proof by contradiction.
$g(n) \in \Omega(f(n) \rightarrow, \exists c, N_1 : g(n) \geq cf(n), \forall n \geq N_1 \;\; \star$
$g(n) \in o(f(n) \rightarrow, \exists N_2 : g(n) < c/2f(n), \forall n \geq N_2 \;\; \star\star$
$\star$ contradicts with $\star\star$ □

- The reverse is not true. Why?

Theorem

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \Rightarrow g(n) \in o(f(n))$$
$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty \Rightarrow f(n) \in o(g(n))$$

Proof.

Exercise! □

Example

$\frac{n^2}{2} \in o(n^3)$

because $\lim_{n\to\infty} \frac{n^2/2}{n^3} = \lim_{n\to\infty} \frac{1}{2n} = 0$

Example

for $b > a > 0$, $a^n \in o(b^n)$

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

Example

for $a > 0$, $a^n \in o(n!)$

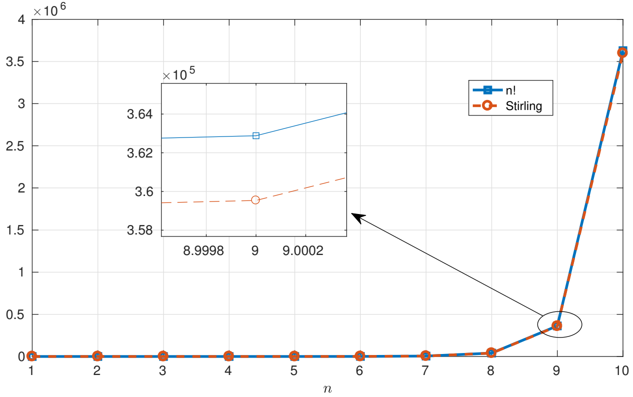
because

$$\lim_{n\to\infty} \frac{a^n}{n!} \le \lim_{n\to\infty} \frac{a^n}{\underbrace{a^4 a^4 \cdots a^4}_{\lceil n/2 \rceil}} \le \lim_{n\to\infty} \frac{a^n}{(a^4)^{n/2}} = \lim_{n\to\infty} \left(\frac{1}{a}\right)^n = 0$$

Example (Stirling's formula)

$2^n \in o(n!)$

because, $\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty$



Example (We can use also the Hopital's rule)

$\log n \in o(n)$
because $\lim_{x\to\infty} \frac{\log x}{x} = \lim_{x\to\infty} \frac{d \log x / dx}{dx/dx} = \lim_{x\to\infty} \frac{1/(x \ln 2)}{1} = 0$

Definition (small omega)

For a given complexity function $f(n)$, $\omega(f(n))$ is the set of all complexity functions $g(n)$ satisfying the following: For every positive real constant c there exists a nonnegative integer N such that, for all $n \geq N$,

$$g(n) > c \times f(n)$$

If $g(n) \in \omega(f(n))$, we say that $g(n)$ is small omega of $f(n)$. We use ω-notation to denote a lower bound that is **not asymptotically tight**.

- Recall that "big Ω" means there must be **some** real positive constant c for which the bound holds. This definition says that the bound must hold for **every** real positive constant c.

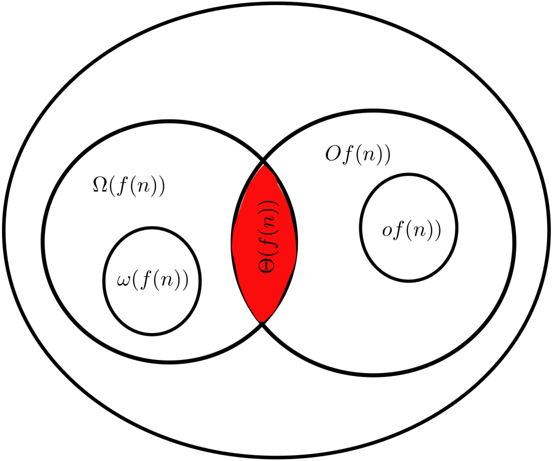
Example

$n^2 \in \omega(n)$ but $n^2/2 \notin \omega(n^2)$

The relationship $g(n) \in \omega(f(n))$ implies that

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

if the limit exists.


$Of(n))$
$\Omega(f(n))$
$\Theta(f(n))$
$of(n))$
$\omega(f(n))$

In Class Question 2

True or False?

> *An $O(n \lg n)$-time algorithm $\underline{\textit{always}}$ runs faster than an $O(n^2)$-time algorithm.*

Properties of Order:

- If $b > 1$ and $a > 1$, then

$$\log_a n \in \Theta(\log_b n) \quad \text{(since } \log_a n = \frac{\log_b n}{\log_b a}\text{)}$$

This implies that all logarithmic complexity functions are in the same complexity category. We will represent this category by $O(\log n)$.

- If $b > a > 0$ then

$$a^n \in o(b^n)$$

This implies that all exponential complexity functions are not in the same complexity category.

- for all $a > 0$

$$a^n \in o(n!)$$

This implies that n! is worse than any exponential complexity function.

- $\Theta(n!) \gg \Theta(b^n) \gg \Theta(a^n) \gg \Theta(n^j) \gg \Theta(n^k) \gg \Theta(n \log n) \gg \Theta(n) \gg \Theta(\log^j n) \gg \Theta(\log^k n) \gg \Theta(1)$
  for any $(a > b,\ j > k)$
- If $c \geq 0, d > 0, g(n) \in O(f(n)), h(n) \in \Theta(f(n))$, then
  $$c \times g(n) + d \times h(n) \in \Theta(f(n)).$$
- If $f(n) = O(h(n))$ and $g(n) = O(h(n))$, then $f(n) + g(n) = O(h(n))$.

Comparison of Functions

- Symmetry:
  $g(n) \in \Theta(f(n))$ if and only if $f(n) \in \Theta(g(n))$.
- Transpose Symmetry:
  $g(n) \in O(f(n))$ if and only if $f(n) \in \Omega(g(n))$.
  $g(n) \in o(f(n))$ if and only if $f(n) \in \omega(g(n))$.
- Transitivity:
  If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
  If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$
  If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$
  If $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$
  If $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$
- Reflexivity: $f(n) \in \Theta(f(n))$, $f(n) \in O(f(n))$, $f(n) \in \Omega(f(n))$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions g and f and the comparison of two real numbers a and b:

$$\begin{aligned}
g(n) \in O(f(n)) &\quad \approx \quad a \leq b \\
g(n) \in \Omega(f(n)) &\quad \approx \quad a \geq b \\
g(n) \in \Theta(f(n)) &\quad \approx \quad a = b \\
g(n) \in o(f(n)) &\quad \approx \quad a < b \\
g(n) \in \omega(f(n)) &\quad \approx \quad a > b
\end{aligned}$$

We say that $g(n)$ *is* **asymptotically smaller** *than* $f(n)$ *if* $g(n) \in o(f(n))$, *and* $g(n)$ *is* **asymptotically larger** *than* $f(n)$ *if* $g(n) \in \omega(f(n))$

But Not All Functions Are Asymptotically Comparable

Let's proof one of the above properties, namely the transitivity of Θ:
If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$

Proof.

$$f(n) = \Theta(g(n)) \rightarrow \exists c_1, d_1, N_1 : c_1 g(n) \leq f(n) \leq d_1 g(n) \quad \forall n \geq N_1$$
$$g(n) = \Theta(h(n)) \rightarrow \exists c_2, d_2, N_2 : c_2 h(n) \leq g(n) \leq d_2 h(n) \quad \forall n \geq N_2$$
$$\Longrightarrow c_1 c_2 h(n) \leq f(n) \leq d_1 d_2 h(n) \quad \forall n \geq \max(N_1, N_2)$$

□

Exercise: Prove other properties.

- A relation on a set S is called an **equivalence relation** if it is reflexive, symmetric, and transitive. Equivalence relations are important throughout mathematics and computer science. One reason for this is that **in an equivalence relation, when two elements are related it makes sense to say they are equivalent**.
- Let R be an equivalence relation on a set S . The set of all elements that are related to an element s of S is called the **equivalence class** of s. The equivalence classes of R form a **partition** of S.

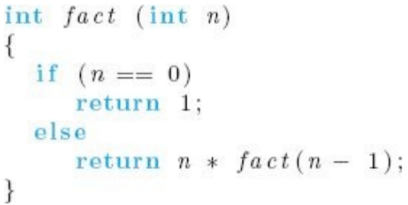
Θ defines an equivalence relation on the set of complexity functions.

Solving Recurrence Equations

The analysis of recursive algorithms is not as straightforward as it is for iterative algorithms.

- Ordinarily, however, **it is not difficult to represent the time complexity of a recursive algorithm by a recurrence equation.**
- The recurrence equation must then be solved to determine the time complexity.

Example



```
int fact (int n)
{
  if (n == 0)
      return 1;
  else
      return n * fact(n - 1);
}
```

For a given n, the number of multiplications, we can wright

$$t_n = t_{n-1} + 1$$
$$t_0 = 0$$

One method to solve the recurrence is **guess and prove by induction**:

$$t_1 = 0 + 1 = 1$$
$$t_2 = 1 + 1 = 2$$
$$t_3 = 2 + 1 = 3$$

So we guess $t_n = n$. The next step is proof by induction.
Induction base: for $n = 0$

$$t_0 = 0$$

Induction hypothesis: Assume, for an arbitrary positive integer n, that

$$t_n = n$$

Induction step: We need to show that

$$t_{n+1} = n + 1.$$

But we can write $t_{n+1} = t_n + 1 = n + 1$, and the proof is complete.

Exercise: What is the solution for the following recurrence equation?

$$t_n = t_{n/2} + 1 \quad n > 1 \text{ and } n \text{ a power of } 2$$
$$t_1 = 1$$

Exercise

```
void func(int m)
{
    cout « "A";
    if (!m)
        break;
    func(m - 1);
    func(m - 1);
}
int main()
{
    int m = 0;
    cin » m;
    func(m);
    return 0;
}
```

- We will see other techniques in the next chapter.