Complexity of Algorithms
Andreas Klappenecker
Example
The sequence of Fibonacci numbers is defined as
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

\[ F_n = \begin{cases} 
F_{n-1} + F_{n-2} & \text{if } n > 1 \\
1 & \text{if } n = 1 \\
0 & \text{if } n = 0 
\end{cases} \]
Fibonacci Implementation

\[
F_n = \begin{cases} 
  F_{n-1} + F_{n-2} & \text{if } n > 1 \\
  1 & \text{if } n = 1 \\
  0 & \text{if } n = 0 
\end{cases}
\]

def fib(n)
    return 0 if n == 0
    return 1 if n == 1
    return fib(n-1) + fib(n-2)
end

For any algorithm, we always ask the following questions:
1) Is it correct?
2) How much time does it take as a function of its input?
3) Can we do better?
Correctness

\[ F_n = \begin{cases} 
F_{n-1} + F_{n-2} & \text{if } n > 1 \\
1 & \text{if } n = 1 \\
0 & \text{if } n = 0 
\end{cases} \]

Our implementation is obviously correct, since it is nothing but the definition of Fibonacci numbers.
def fib(n)
    return 0 if n == 0
    return 1 if n == 1
    return fib(n-1) + fib(n-2)
end

The running time is a bit more difficult to analyze. Let's try it out in ruby. Start the interactive ruby interpreter irb, type in the program, run fib(0), fib(1), ..., fib(10), fib(100)
def fib(n)
    return 0 if n == 0
    return 1 if n == 1
    return fib(n-1) + fib(n-2)
end

Let $T(n)$ denote the number of steps needed to compute $\text{fib}(n)$. Then:
$T(0) \leq 3$ if $n \leq 1$
$T(n) = T(n-1) + T(n-2) + 3$ for $n > 1$
that is, two recursive invocations of $\text{fib}$, two checks of values of $n$, plus one addition.

Slowpoke alert!
The running time $T(n)$ of this program exceeds the $n$th Fibonacci number!
Impractical, except for the smallest inputs of $n$. 
Motivating
Asymptotic Run Time Analysis
Running Time

The running time of a program depends
- on the input
- how the compiler translates the program
- the hardware

This can be quite complex, especially with modern architectures that might have long pipelines, speculative execution, and so on.
The execution of a high level instruction might take a short or a long time, depending the state of the pipeline, success of speculative execution, and many other factors.

We will have to estimate the time, no matter what kind of approach we will take, otherwise the analysis will become too complex.
In general, the size of the input has a big impact on the running time, as we have seen in the Fibonacci example.

In general, we might have many inputs of the same size (e.g. data to be sorted). To simplify matters, we would like to study the running time as a function of the input size.

Problem: Different inputs of the same size can lead to a different running time (think sorting!)
Worst Case Running Time

The largest possible running time of an algorithm over all possible inputs of a given size n is called the worst case running time for an input of size n.

We would like to know how the worst case running time scales with n. Often we are happy with good upper bounds on the worst case running time.
Best Case Running Time

The smallest possible running time of an algorithm over all possible inputs of a given size \( n \) is called the best case running time for an input of size \( n \).

We would like to know how the best case running time scales with \( n \). Often we are happy with good lower bounds on the best case running time.
Machine Independence

We would like to make the running time analysis independent of a particular choice of processor or compiler.

We do not want to redo an analysis if we replace a processor by a new processor that is twice faster and has a slightly different architecture.

==> Analyze running time up to a multiplicative constant.
Asymptotics

At the beginning of an execution, the behavior is quite different, since the pipeline might not be completely filled, the number of cache faults is large, and so on.

Therefore, it makes sense to consider the running time just for large input sizes.

As an added benefit, it will allow us to simplify our estimates of the best case and worst case running time even further.
Asymptotic Notations
Big Oh Notation

Let \( f, g : \mathbb{N} \to \mathbb{R} \) be functions from the natural numbers to the set of real numbers.

We write \( f(n) \in O(g(n)) \) [or \( f(n) = O(g(n)) \)] if and only if there exists some natural number \( n_0 \) and a positive real constant \( U \) such that

\[
|f(n)| \leq U|g(n)|
\]

for all \( n \) satisfying \( n \geq n_0 \).
Upper Bound

The big Oh notation provides an upper bound on a function $f$.

$f(n) = O(n)$ means that $f(n) \leq U n$
for some constant $U$ and for all $n \geq n_0$

The notation does not imply that $f$ has to grow that fast.
E.g. $f(n) = 1$ satisfies $f(n)=O(n)$.
so $f(n) = O(1)$ would have been a better bound.

$\Rightarrow$ We need lower bounds as well.
We define \( f(n) = \Omega(g(n)) \) if and only if there exists a constant \( L \) and a natural number \( n_0 \) such that
\[
L \cdot |g(n)| \leq |f(n)|
\]
holds for all \( n \geq n_0 \).

In other words, \( f(n) = \Omega(g(n)) \) if and only if \( g(n) = O(f(n)) \).
Example

In Analysis of Algorithms, you will learn that any comparison based sorting algorithm needs at least $\Omega(n \log n)$ comparisons.

Mergesort needs $O(n \log n)$ comparisons, so this is essentially an optimal sorting algorithm.
We define $f(n) = \Theta(g(n))$ if and only if there exist constants $L$ and $U$ and a natural number $n_0$ such that

$$L|g(n)| \leq |f(n)| \leq U|g(n)|$$

holds for all $n \geq n_0$.

In other words, $f(n) = \Theta(g(n))$ if and only if $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$. 

Big $\Theta$
Informal Summary

\[ f(n) = O(g(n)) \] means that \(|f(n)|\) is upper bounded by a constant multiple of \(|g(n)|\) for all large \(n\).

\[ f(n) = \Omega(g(n)) \] means that \(|f(n)|\) is lower bounded by a constant multiple of \(|g(n)|\) for all large \(n\).

\[ f(n) = \Theta(g(n)) \] means that \(|f(n)|\) has the same asymptotic growth as \(|g(n)|\) up to multiplication by constants.
Convenience of the Asymptotic Notations
Rule 1

Suppose that \( f(n) \) and \( g(n) \) are functions such that \(|f(n)| \leq |g(n)|\) holds for all \( n \geq n_0 \). Then \( f(n) + g(n) = O(g(n)) \)

Indeed, \(|f(n)+g(n)| \leq |f(n)|+|g(n)| \leq 2|g(n)|\) holds for all \( n \geq n_0 \). Therefore, by definition, \( f(n)+g(n) = O(g(n)) \)

Examples:
\[
\begin{align*}
  n^2+2n &= O(n^2) \\
  n^2+n \log n + 2n + 1723 &= O(n^2)
\end{align*}
\]
Rule 2

Suppose that $c$ is a nonzero real number. Then

$$c \cdot f(n) = O(f(n))$$

Indeed, there exists a constant $U = |c|$ such that

$$|c f(n)| \leq |c| |f(n)| = U |f(n)|$$

holds for all $n$. Therefore, $c f(n) = O(f(n))$. 
The limit
\[ \lim_{n \to \infty} f(n) \]
exists and is equal to \( L \) if and only if for all \( \varepsilon > 0 \) there exists a natural number \( n_0 = n_0(\varepsilon) \) such that
\[ |f(n) - L| < \varepsilon \]
holds for all \( n \geq n_0 \).

Roughly, the sequence \( f(n) \) has a limit iff the values of \( f(n) \) approach \( L \) for large \( n \).
Rule 3

Let $f$ and $g$ be two functions such that

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

exists and is equal to $L$. Then $f(n) = O(g(n))$.

Indeed, since $\lim_{n \to \infty} \frac{|f(n)|}{|g(n)|} = L$, it follows that for all $\varepsilon > 0$ there exists a natural number $n_0$ such that

$$L - \varepsilon \leq \frac{|f(n)|}{|g(n)|} \leq L + \varepsilon$$

holds for all $n \geq n_0$.

Hence, $|f(n)| \leq U|g(n)|$ holds for all $n \geq n_0$ with $U = L + \varepsilon$. 

If $f$, $g$, $h$ are functions such that $f(n) = O(g(n))$ and $g(n) = O(h(n))$
then $f(n) = O(h(n))$
(1) Suppose that \( f(n) \) and \( g(n) \) are functions such that \(|f(n)| \leq |g(n)|\) holds for all \( n \geq n_0 \). Then \( f(n) + g(n) = \Omega(f(n)) \)

(2) \( cf(n) = \Omega(f(n)) \)

(3) If the limit \( \lim_{n \to \infty} |g(n)|/|f(n)| \) exists, then \( f(n) = \Omega(g(n)) \)

(4) If \( f(n) = \Omega(g(n)) \) and \( g(n) = \Omega(h(n)) \), then \( f(n) = \Omega(h(n)) \)
Let $f$ and $g$ be two functions such that

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

exists and is equal to $L > 0$. Then $f(n) = \Theta(g(n))$.

Indeed, since $\lim_{n \to \infty} \frac{|f(n)|}{|g(n)|} = L > 0$, for all $\varepsilon$ in the range $0 < \varepsilon < L$ there exists a natural number $n_0$ such that

$$L - \varepsilon \leq \frac{|f(n)|}{|g(n)|} \leq L + \varepsilon$$

holds for all $n \geq n_0$. Hence $(L - \varepsilon)|g(n)| \leq |f(n)| \leq (L + \varepsilon)|g(n)|$ holds for all $n \geq n_0$. Thus, $f(n) = \Theta(g(n))$. 
Different Interpretation
Set Interpretation

We have interpreted $f(n) = O(g(n))$ as a relation between the two functions $f(n)$ and $g(n)$.

We can give $O(g(n))$ a meaning by interpreting it as the set of all functions $f(n)$ such that $f(n) = O(g(n))$, that is,

$$O(g) = \{ f : \mathbb{N} \rightarrow \mathbb{R} \mid \text{there exists a constant } U \text{ and a natural number } n_0 \text{ such that } |f(n)| \leq U |g(n)| \text{ for all } n \geq n_0 \}$$
Example $O(n^2)$
Complexity
Elementary Operations

Elementary operations such as
- assignments \( a := \text{rhs} \)
- arithmetic with machine sized words (\( x+y, x-y, x*y, x/y \))
- boolean operations (\( a && b, a || b, a \& b, a | b, ... \))
- comparisons (\( a<b, a\leq b, a=b, a>b, a\geq b, ... \))
- array access \( arr[i] \)

have constant running time, that is, \( O(1) \) time.
Compound Statements

Suppose that we are given a sequence of statements:

Block1 ; Block 2; ... ; Block n

If Block k takes $T_k$ time, then a total time of

$T_1 + T_2 + ... + T_n$

is required to execute the sequence of the $n$ blocks.
Control Structures

if BoolExpr then
    Block 1
else
    Block 2
end

If evaluation of BoolExpr takes time $T_B$, and execution of the Block $k$ takes $T_k$ time, then the above if .. then .. else statement takes $O(T_B) + O(\max(T_1, T_2))$ time.
For Loop

for k in (a..b) do
    Block 1
end

If $T_1(v)$ is the time required for the execution of Block 1 when the loop variable $k == v$, then the for loop takes

$O(T_1(a)) + O(T_1(a+1)) + \ldots + O(T_1(b))$

time. In particular, if the running time $T_1$ of the block is independent from the loop variable, then $(b-a+1) O(T_1)$ time.
Function Calls

def f(params)
    Block 1
end

If $T_p$ is the time to assign the parameters and $T_1(params)$ is the time to execute Block 1 given the parameters params, then the running time of $f(params)$ is given by $O(T_p + T_1)$.
Example

def bubble_sort(list):
    list = list.dup  # make copy of input
    for i in 0..(list.length-2) do
        for j in 0..(list.length - i - 2) do
            list[j], list[j + 1] = list[j + 1], list[j] if list[j + 1] < list[j]
        end
    end
    return list
end
def bubble_sort(list)
    list = list.dup  # we should not modify original list
    for i in 0..(list.length-2) do
        for j in 0..(list.length - i - 2) do
            list[j], list[j + 1] = list[j + 1], list[j]  if list[j + 1] < list[j]
        end
    end
    return list
end

>> bubble_sort([6,5,4,3,2,1])
[5, 4, 3, 2, 1, 6]
[4, 3, 2, 1, 5, 6]
[3, 2, 1, 4, 5, 6]
[2, 1, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
list[j], list[j + 1] = list[j + 1], list[j] if list[j + 1] < list[j]

The evaluation of the boolean expression takes $O(1)$ time, and the parallel assignment realizing the swap operation takes $O(1)$ as well, since it corresponds to 3 assignments.

Therefore, the statement takes $O(1)$ time.
Inner For Loop

```plaintext
for j in 0..(list.length - i - 2) do
    list[j], list[j + 1] = list[j + 1], list[j] if list[j + 1] < list[j]
end

Let n = list.length
Then the loop takes \((n - i - 1)\) \(O(1)\) time.
```
Nested For Loops

```
for i in 0..(list.length-2) do  # number of iterations: n-1
    for j in 0..(list.length - i - 2) do  # number of items: n-1, n-2, ..., 1
        list[j], list[j + 1] = list[j + 1], list[j]  if list[j + 1] < list[j]
    end
end
```

Let \( n = \text{list.length} \). Then the two nested loops take

\[
(n-1 + n-2 + ... + 1) \, O(1) = \left(\frac{n(n-1)}{2}\right) \, O(1) = O(n^2) \text{ time.}
\]
Total Time Complexity

def bubble_sort(list) # O(1) for parameter assignment
    list = list.dup  # O(n)
    for i in 0..(list.length-1) do # O(n^2)
        for j in 0..(list.length - i - 2) do
            list[j], list[j + 1] = list[j + 1], list[j] if list[j + 1] < list[j]
        end
    end
    return list # O(1)
end # O(1) + O(n) + O(n^2) + O(1) = O(n^2)