Complexity of Algorithms

Andreas Klappenecker
Let $f, g: \mathbb{N} \rightarrow \mathbb{R}$ be functions from the natural numbers to the set of real numbers.

We write $f \in O(g)$ if and only if there exists some real number $n_0$ and a positive real constant $U$ such that
\[
|f(n)| \leq U|g(n)|
\]
for all $n$ satisfying $n \geq n_0$.
The big Oh notation provides an upper bound on a function $f$.  

$f(n) = O(n)$ means that $f(n) \leq Un$ 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

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

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 $f(n) = O(g(n))$. 
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.
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 $\text{arr}[i]$

have constant running time, that is, $\Theta(1)$ 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

$O(T_1(a)) + O(T_1(a+1)) + ... + O(T_1(b))$

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.
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  # 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
```ruby
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
```

```ruby
>> 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]
```
Loop Body

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

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 itns: 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 = list.length. Then the two nested loops take

(n-1 + n-2 + ... + 1 ) \( O(1) = (n(n-1)/2) \) \( O(1) = O(n^2) \) 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)