WS 2007/2008

Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

1. Landau Symbols

The definitions above offer us a very detailed look at an algorithm's resource usage. But as we will see, we will sometimes want to reduce the amount of detail.

- Uniform and logarithmic time and space complexities depend considerably on the machine model, i.e. the type of computer the algorithm is implemented and run on. Differences between machine models include the instruction set, the cost of individual instructions, etc.

  As a result, running times differ by a constant factor which is independent of n. We are not interested in the particularities of individual machines and therefore want to get rid of constant factors.

1. Landau Symbols

The definitions above offer us a very detailed look at an algorithm's resource usage. But as we will see, we will sometimes want to reduce the amount of detail.

- Uniform and logarithmic time and space complexities depend considerably on the machine model, i.e. the type of computer the algorithm is implemented and run on. Differences between machine models include the instruction set, the cost of individual instructions, etc.

  As a result, running times differ by a constant factor which is independent of n. We are not interested in the particularities of individual machines and therefore want to get rid of constant factors.

1. Landau Symbols

The definitions above offer us a very detailed look at an algorithm's resource usage. But as we will see, we will sometimes want to reduce the amount of detail.

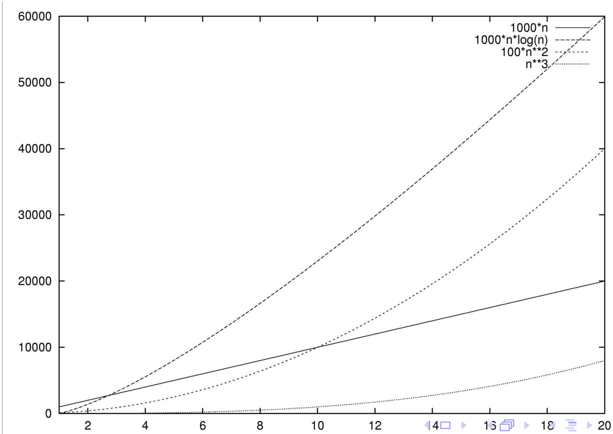
- Uniform and logarithmic time and space complexities depend considerably on the machine model, i.e. the type of computer the algorithm is implemented and run on. Differences between machine models include the instruction set, the cost of individual instructions, etc.
  As a result, running times differ by a constant factor which is independent of n. We are not interested in the particularities of individual machines and therefore want to get rid of constant factors.

- We are not interested in the time and space usage of an algorithm when executed for some specific input, but instead consider the worst case scenario, given input size n.
  Hence, we look at upper bounds of the time and space complexities for given input sizes.

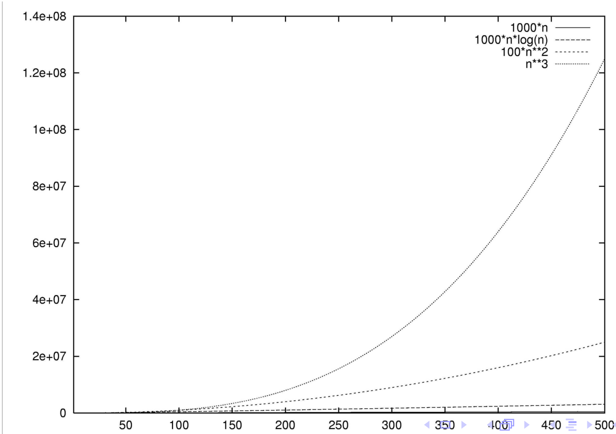
- We are not interested in the time and space usage of an algorithm when executed for some specific input, but instead consider the worst case scenario, given input size n.
  Hence, we look at upper bounds of the time and space complexities for given input sizes.

- We are not interested in the time and space usage of an algorithm when executed for some specific input, but instead consider the worst case scenario, given input size n. Hence, we look at upper bounds of the time and space complexities for given input sizes.

- Comparing the efficiency of algorithms by the functions $t(n)$ and $s(n)$ describing the upper bounds of time and space usage makes little sense for small values of n.
  One function $t_1(n)$ that clearly dominates another function $t_2(n)$ may yield smaller values for small input sizes n.
  Example: Remember our the functions seen in the first lecture.



- Comparing the efficiency of algorithms by the functions $t(n)$ and $s(n)$ describing the upper bounds of time and space usage makes little sense for small values of n.
  One function $t_1(n)$ that clearly dominates another function $t_2(n)$ may yield smaller values for small input sizes n.
  Example: Remember our the functions seen in the first lecture.



This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n\rightarrow\infty} \frac{g(n)}{f(n)} = 0\}$
- $\omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) > c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} = 0\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0\}$
- $\omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) > c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0\}$
- $\omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) > c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0\}$

This leads us to the following definitions.

Definition 1

Let $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ be some real-valued function. Then we define the following notations:

- $O(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \leq c \cdot f(n))\}$
- $\Omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\exists c \in \mathbf{R}^+, n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) \geq c \cdot f(n))\}$
- $\Theta(f) := O(f) \cap \Omega(f)$
- $o(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) < c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0\}$
- $\omega(f) := \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : (\forall c \in \mathbf{R}^+ : \exists n_0 \in \mathbf{N} : \forall n \geq n_0 : g(n) > c \cdot f(n))\} = \{g : \mathbf{N} \longrightarrow \mathbf{R}^+ : \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0\}$

Note on Landau notation: Most everybody in the computer science world agrees that it is nicer to write $f(n) = O(g(n))$ or $f = O(g)$ instead of $\in$. This notation is somewhat sloppy, but most likely you will want to use it, too, once you're used to it.

Some intuition on Landau symbols:

Note on Landau notation: Most everybody in the computer science world agrees that it is nicer to write $f(n) = O(g(n))$ or $f = O(g)$ instead of $\in$. This notation is somewhat sloppy, but most likely you will want to use it, too, once you're used to it.

Some intuition on Landau symbols:

Note on Landau notation: Most everybody in the computer science world agrees that it is nicer to write $f(n) = O(g(n))$ or $f = O(g)$ instead of $\in$. This notation is somewhat sloppy, but most likely you will want to use it, too, once you're used to it.

Some intuition on Landau symbols:

- $f = O(g)$ means that, for almost all n, $g(n) \leq c \cdot f(n)$, where c is some existing positive constant. In other words, f grows at most as fast as g as n grows.
- $f = \Omega(g)$ conversely means that f grows at least as fast as g asymptotically.
- $f = \Theta(g)$ means that, up to some constant factor, f and g grow equally fast asymptotically.
- $f = o(g)$ means that f grows strictly less fast than g asymptotically.
- $f = \omega(g)$ menas that f grows strictly faster than g asymptotically.

Note on Landau notation: Most everybody in the computer science world agrees that it is nicer to write $f(n) = O(g(n))$ or $f = O(g)$ instead of $\in$. This notation is somewhat sloppy, but most likely you will want to use it, too, once you're used to it. Some intuition on Landau symbols:

- $f = O(g)$ means that, for almost all n, $g(n) \leq c \cdot f(n)$, where c is some existing positive constant. In other words, f grows at most as fast as g as n grows.
- $f = \Omega(g)$ conversely means that f grows at least as fast as g asymptotically.
- $f = \Theta(g)$ means that, up to some constant factor, f and g grow equally fast asymptotically.
- $f = o(g)$ means that f grows strictly less fast than g asymptotically.
- $f = \omega(g)$ menas that f grows strictly faster than g asymptotically.

Note on Landau notation: Most everybody in the computer science world agrees that it is nicer to write $f(n) = O(g(n))$ or $f = O(g)$ instead of $\in$. This notation is somewhat sloppy, but most likely you will want to use it, too, once you're used to it. Some intuition on Landau symbols:

- $f = O(g)$ means that, for almost all n, $g(n) \leq c \cdot f(n)$, where c is some existing positive constant. In other words, f grows at most as fast as g as n grows.
- $f = \Omega(g)$ conversely means that f grows at least as fast as g asymptotically.
- $f = \Theta(g)$ means that, up to some constant factor, f and g grow equally fast asymptotically.
- $f = o(g)$ means that f grows strictly less fast than g asymptotically.
- $f = \omega(g)$ menas that f grows strictly faster than g asymptotically.

Note on Landau notation: Most everybody in the computer science world agrees that it is nicer to write $f(n) = O(g(n))$ or $f = O(g)$ instead of $\in$. This notation is somewhat sloppy, but most likely you will want to use it, too, once you're used to it. Some intuition on Landau symbols:

- $f = O(g)$ means that, for almost all n, $g(n) \leq c \cdot f(n)$, where c is some existing positive constant. In other words, f grows at most as fast as g as n grows.
- $f = \Omega(g)$ conversely means that f grows at least as fast as g asymptotically.
- $f = \Theta(g)$ means that, up to some constant factor, f and g grow equally fast asymptotically.
- $f = o(g)$ means that f grows strictly less fast than g asymptotically.
- $f = \omega(g)$ menas that f grows strictly faster than g asymptotically.

Note on Landau notation: Most everybody in the computer science world agrees that it is nicer to write $f(n) = O(g(n))$ or $f = O(g)$ instead of $\in$. This notation is somewhat sloppy, but most likely you will want to use it, too, once you're used to it.
Some intuition on Landau symbols:

- $f = O(g)$ means that, for almost all n, $g(n) \leq c \cdot f(n)$, where c is some existing positive constant. In other words, f grows at most as fast as g as n grows.
- $f = \Omega(g)$ conversely means that f grows at least as fast as g asymptotically.
- $f = \Theta(g)$ means that, up to some constant factor, f and g grow equally fast asymptotically.
- $f = o(g)$ means that f grows strictly less fast than g asymptotically.
- $f = \omega(g)$ menas that f grows strictly faster than g asymptotically.

Example 2

- $n^2 + 2n - 1 = O(n^2)$: select $n_0 = 1, c = 2$
- $n^2 + 2n - 1 = \Omega(n^2)$: select $n_0 = 1, c = 1$
- Hence: $n^2 + 2n - 1 = \Theta(n^2)$
- Equally easy to prove: $3n + 4 = o(n^2)$ and $n^2 + 2n - 1 = \omega(n \log n)$
- $n^2 \log + 4n(\log n)^2 = \Theta(n^2 \log n)$
- $\log n = o\left(\sqrt{n}\right)$ (Try to prove this by yourself)
- General polynomials: $\sum\limits_{i=0}^{k} a_i n^i = \Theta(n^k)$ if $a_k > 0$

Example 2

- $n^2 + 2n - 1 = O(n^2)$: select $n_0 = 1, c = 2$
- $n^2 + 2n - 1 = \Omega(n^2)$: select $n_0 = 1, c = 1$
- Hence: $n^2 + 2n - 1 = \Theta(n^2)$
- Equally easy to prove: $3n + 4 = o(n^2)$ and $n^2 + 2n - 1 = \omega(n \log n)$
- $n^2 \log + 4n(\log n)^2 = \Theta(n^2 \log n)$
- $\log n = o\left(\sqrt{n}\right)$ (Try to prove this by yourself)
- General polynomials: $\sum\limits_{i=0}^{k} a_i n^i = \Theta(n^k)$ if $a_k > 0$

Example 2

- $n^2 + 2n - 1 = O(n^2)$: select $n_0 = 1, c = 2$
- $n^2 + 2n - 1 = \Omega(n^2)$: select $n_0 = 1, c = 1$
- Hence: $n^2 + 2n - 1 = \Theta(n^2)$
- Equally easy to prove: $3n + 4 = o(n^2)$ and $n^2 + 2n - 1 = \omega(n \log n)$
- $n^2 \log + 4n(\log n)^2 = \Theta(n^2 \log n)$
- $\log n = o\left(\sqrt{n}\right)$ (Try to prove this by yourself)
- General polynomials: $\sum\limits_{i=0}^{k} a_i n^i = \Theta(n^k)$ if $a_k > 0$

Example 2

- $n^2 + 2n - 1 = O(n^2)$: select $n_0 = 1, c = 2$
- $n^2 + 2n - 1 = \Omega(n^2)$: select $n_0 = 1, c = 1$
- Hence: $n^2 + 2n - 1 = \Theta(n^2)$
- Equally easy to prove: $3n + 4 = o(n^2)$ and $n^2 + 2n - 1 = \omega(n \log n)$
- $n^2 \log +4n(\log n)^2 = \Theta(n^2 \log n)$
- $\log n = o\left(\sqrt{n}\right)$ (Try to prove this by yourself)
- General polynomials: $\sum\limits_{i=0}^{k} a_i n^i = \Theta(n^k)$ if $a_k > 0$

Example 2

- $n^2 + 2n - 1 = O(n^2)$: select $n_0 = 1, c = 2$
- $n^2 + 2n - 1 = \Omega(n^2)$: select $n_0 = 1, c = 1$
- Hence: $n^2 + 2n - 1 = \Theta(n^2)$
- Equally easy to prove: $3n + 4 = o(n^2)$ and $n^2 + 2n - 1 = \omega(n \log n)$
- $n^2 \log + 4n(\log n)^2 = \Theta(n^2 \log n)$
- $\log n = o\left(\sqrt{n}\right)$ (Try to prove this by yourself)
- General polynomials: $\sum_{i=0}^{k} a_i n^i = \Theta(n^k)$ if $a_k > 0$

Example 2

- $n^2 + 2n - 1 = O(n^2)$: select $n_0 = 1, c = 2$
- $n^2 + 2n - 1 = \Omega(n^2)$: select $n_0 = 1, c = 1$
- Hence: $n^2 + 2n - 1 = \Theta(n^2)$
- Equally easy to prove: $3n + 4 = o(n^2)$ and $n^2 + 2n - 1 = \omega(n \log n)$
- $n^2 \log + 4n(\log n)^2 = \Theta(n^2 \log n)$
- $\log n = o\left(\sqrt{n}\right)$ (Try to prove this by yourself)
- General polynomials: $\sum\limits_{i=0}^{k} a_i n^i = \Theta(n^k)$ if $a_k > 0$

Example 2

- $n^2 + 2n - 1 = O(n^2)$: select $n_0 = 1, c = 2$
- $n^2 + 2n - 1 = \Omega(n^2)$: select $n_0 = 1, c = 1$
- Hence: $n^2 + 2n - 1 = \Theta(n^2)$
- Equally easy to prove: $3n + 4 = o(n^2)$ and $n^2 + 2n - 1 = \omega(n \log n)$
- $n^2 \log + 4n(\log n)^2 = \Theta(n^2 \log n)$
- $\log n = o\left(\sqrt{n}\right)$ (Try to prove this by yourself)
- General polynomials: $\sum_{i=0}^{k} a_i n^i = \Theta(n^k)$ if $a_k > 0$

Lemma 3

It holds that

1. $g = o(f) \Rightarrow g = O(f)$
2. $g = o(f) \not\Leftarrow g = O(f)$
3. $g = \omega(f) \Rightarrow g = \Omega(f)$
4. $g = \omega(f) \not\Leftarrow g = \Omega(f)$
5. $g = \Omega(f) \Leftrightarrow f = O(g)$
6. $g = o(f) \Rightarrow g \neq \Omega(f)$
7. $g = \omega(f) \Rightarrow g \neq O(f)$

Lemma 3

It holds that

1. $g = o(f) \Rightarrow g = O(f)$
2. $g = o(f) \not\Leftarrow g = O(f)$
3. $g = \omega(f) \Rightarrow g = \Omega(f)$
4. $g = \omega(f) \not\Leftarrow g = \Omega(f)$
5. $g = \Omega(f) \Leftrightarrow f = O(g)$
6. $g = o(f) \Rightarrow g \neq \Omega(f)$
7. $g = \omega(f) \Rightarrow g \neq O(f)$

Lemma 3

It holds that

1. $g = o(f) \Rightarrow g = O(f)$
2. $g = o(f) \not\Leftarrow g = O(f)$
3. $g = \omega(f) \Rightarrow g = \Omega(f)$
4. $g = \omega(f) \not\Leftarrow g = \Omega(f)$
5. $g = \Omega(f) \Leftrightarrow f = O(g)$
6. $g = o(f) \Rightarrow g \neq \Omega(f)$
7. $g = \omega(f) \Rightarrow g \neq O(f)$

Lemma 3

It holds that

1. $g = o(f) \Rightarrow g = O(f)$
2. $g = o(f) \not\Leftarrow g = O(f)$
3. $g = \omega(f) \Rightarrow g = \Omega(f)$
4. $g = \omega(f) \not\Leftarrow g = \Omega(f)$
5. $g = \Omega(f) \Leftrightarrow f = O(g)$
6. $g = o(f) \Rightarrow g \neq \Omega(f)$
7. $g = \omega(f) \Rightarrow g \neq O(f)$

Lemma 3

It holds that

1. $g = o(f) \Rightarrow g = O(f)$
2. $g = o(f) \not\Leftarrow g = O(f)$
3. $g = \omega(f) \Rightarrow g = \Omega(f)$
4. $g = \omega(f) \not\Leftarrow g = \Omega(f)$
5. $g = \Omega(f) \Leftrightarrow f = O(g)$
6. $g = o(f) \Rightarrow g \neq \Omega(f)$
7. $g = \omega(f) \Rightarrow g \neq O(f)$

Lemma 3

It holds that

1. $g = o(f) \Rightarrow g = O(f)$
2. $g = o(f) \not\Leftarrow g = O(f)$
3. $g = \omega(f) \Rightarrow g = \Omega(f)$
4. $g = \omega(f) \not\Leftarrow g = \Omega(f)$
5. $g = \Omega(f) \Leftrightarrow f = O(g)$
6. $g = o(f) \Rightarrow g \neq \Omega(f)$
7. $g = \omega(f) \Rightarrow g \neq O(f)$

Lemma 3

It holds that

1. $g = o(f) \Rightarrow g = O(f)$
2. $g = o(f) \not\Leftarrow g = O(f)$
3. $g = \omega(f) \Rightarrow g = \Omega(f)$
4. $g = \omega(f) \not\Leftarrow g = \Omega(f)$
5. $g = \Omega(f) \Leftrightarrow f = O(g)$
6. $g = o(f) \Rightarrow g \neq \Omega(f)$
7. $g = \omega(f) \Rightarrow g \neq O(f)$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Definition 4

A function f is called

- constant if $f(n) = \Theta(1)$
- logarithmic if $f(n) = O(\log n)$
- linear if $f(n) = O(n)$
- quadratic if $f(n) = O(n^2)$
- polynomial if $f(n) = O(n^k)$ for some $k \in \mathbf{N}$
- superpolynomial if $f(n) = \omega(n^k)$ for all $k \in \mathbf{N}$
- subexponential if $f(n) = o(2^{cn})$ for all $c \in \mathbf{R}^+$
- exponential if $f(n) = O(2^{cn})$ for some $c \in \mathbf{R}^+$

Chapter III Sorting Algorithms

1. Introduction

We are given a non-sorted sequence $D_1, D_2, \ldots, D_n$ consisting of n data elements. Each data element D_i consists of a key k_i and some data contents X_i.

Definition 5

The sorting problem consists in determining a permutation, i.e. a bijection $\pi : \{1, 2, \ldots, n\} \longrightarrow \{1, 2, \ldots, n\}$ such that

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(n)}.$$

Chapter III Sorting Algorithms

1. Introduction

We are given a non-sorted sequence $D_1, D_2, \ldots, D_n$ consisting of n data elements. Each data element D_i consists of a key k_i and some data contents X_i.

- The data contents can be of any form, e.g. a number, a string or some representation of a picture.
- The set of keys has a total ordering "$\leq$".
- We make the temporary assumption that the keys all have distinct values that are stored in an array.
- In particular, we assume that the set of keys is $\{1, 2, \ldots, n\}$.

Definition 5

The sorting problem consists in determining a permutation, i.e. a bijection $\pi : \{1, 2, \ldots, n\} \longrightarrow \{1, 2, \ldots, n\}$ such that

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(n)}.$$

Chapter III Sorting Algorithms

1. Introduction

We are given a non-sorted sequence $D_1, D_2, \dots, D_n$ consisting of n data elements. Each data element D_i consists of a key k_i and some data contents X_i.

- The data contents can be of any form, e.g. a number, a string or some representation of a picture.
- The set of keys has a total ordering "$\leq$".
- We make the temporary assumption that the keys all have distinct values that are stored in an array.
- In particular, we assume that the set of keys is $\{1, 2, \dots, n\}$.

Definition 5

The sorting problem consists in determining a permutation, i.e. a bijection $\pi : \{1, 2, \dots, n\} \longrightarrow \{1, 2, \dots, n\}$ such that

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(n)}.$$

Chapter III Sorting Algorithms

1. Introduction

We are given a non-sorted sequence $D_1, D_2, \ldots, D_n$ consisting of n data elements. Each data element D_i consists of a key k_i and some data contents X_i.

- The data contents can be of any form, e.g. a number, a string or some representation of a picture.
- The set of keys has a total ordering "$\leq$".
- We make the temporary assumption that the keys all have distinct values that are stored in an array.
- In particular, we assume that the set of keys is $\{1, 2, \ldots, n\}$.

Definition 5

The sorting problem consists in determining a permutation, i.e. a bijection $\pi : \{1, 2, \ldots, n\} \longrightarrow \{1, 2, \ldots, n\}$ such that

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(n)}.$$

Chapter III Sorting Algorithms

1. Introduction

We are given a non-sorted sequence $D_1, D_2, \ldots, D_n$ consisting of n data elements. Each data element D_i consists of a key k_i and some data contents X_i.

- The data contents can be of any form, e.g. a number, a string or some representation of a picture.
- The set of keys has a total ordering "$\leq$".
- We make the temporary assumption that the keys all have distinct values that are stored in an array.
- In particular, we assume that the set of keys is $\{1, 2, \ldots, n\}$.

Definition 5

The sorting problem consists in determining a permutation, i.e. a bijection $\pi : \{1, 2, \ldots, n\} \longrightarrow \{1, 2, \ldots, n\}$ such that

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(n)}.$$

Chapter III Sorting Algorithms

1. Introduction

We are given a non-sorted sequence $D_1, D_2, \dots, D_n$ consisting of n data elements. Each data element D_i consists of a key k_i and some data contents X_i.

- The data contents can be of any form, e.g. a number, a string or some representation of a picture.
- The set of keys has a total ordering "$\leq$".
- We make the temporary assumption that the keys all have distinct values that are stored in an array.
- In particular, we assume that the set of keys is $\{1, 2, \dots, n\}$.

Definition 5

The sorting problem consists in determining a permutation, i.e. a bijection $\pi : \{1, 2, \dots, n\} \longrightarrow \{1, 2, \dots, n\}$ such that

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(n)}.$$

Chapter III Sorting Algorithms

1. Introduction

We are given a non-sorted sequence $D_1, D_2, \ldots, D_n$ consisting of n data elements. Each data element D_i consists of a key k_i and some data contents X_i.

- The data contents can be of any form, e.g. a number, a string or some representation of a picture.
- The set of keys has a total ordering "$\leq$".
- We make the temporary assumption that the keys all have distinct values that are stored in an array.
- In particular, we assume that the set of keys is $\{1, 2, \ldots, n\}$.

Definition 5

The sorting problem consists in determining a permutation, i.e. a bijection $\pi : \{1, 2, \ldots, n\} \longrightarrow \{1, 2, \ldots, n\}$ such that

$$k_{\pi(1)} \leq k_{\pi(2)} \leq \cdots \leq k_{\pi(n)}.$$

2. Algorithm Design by Induction

We now introduce a methodology that allows us to derive first algorithmic approaches to many types of problems in which the sizes of problem instances depend on a parameter n. We are interested in an algorithm that can solve arbitrary problem instances of size n, for all $n \in \mathbf{N}$.

Basic Approach:

2. Algorithm Design by Induction

We now introduce a methodology that allows us to derive first algorithmic approaches to many types of problems in which the sizes of problem instances depend on a parameter n. We are interested in an algorithm that can solve arbitrary problem instances of size n, for all $n \in \mathbf{N}$.

Basic Approach:

2. Algorithm Design by Induction

We now introduce a methodology that allows us to derive first algorithmic approaches to many types of problems in which the sizes of problem instances depend on a parameter n. We are interested in an algorithm that can solve arbitrary problem instances of size n, for all $n \in \mathbf{N}$.

Basic Approach:

I. Base Case: We give solutions to "small" problem instances, where n is at most some fixed constant, e.g. $n = 1$.

II. Inductive Step:

2. Algorithm Design by Induction

We now introduce a methodology that allows us to derive first algorithmic approaches to many types of problems in which the sizes of problem instances depend on a parameter n. We are interested in an algorithm that can solve arbitrary problem instances of size n, for all $n \in \mathbf{N}$.

Basic Approach:

I. Base Case: We give solutions to "small" problem instances, where n is at most some fixed constant, e.g. $n = 1$.

II. Inductive Step:
  - We pretend to have a method that solves problem instances of size $< n$. This is often called the induction hypothesis.
  - For instances of size n, we devise a method that decomposes the given problem instance into smaller instances of size $< n$, e.g. $n - 1$ or $\lfloor n/2 \rfloor$. It is essential that this be instances of the same problem, just smaller.

2. Algorithm Design by Induction

We now introduce a methodology that allows us to derive first algorithmic approaches to many types of problems in which the sizes of problem instances depend on a parameter n. We are interested in an algorithm that can solve arbitrary problem instances of size n, for all $n \in \mathbf{N}$.

Basic Approach:

I. Base Case: We give solutions to "small" problem instances, where n is at most some fixed constant, e.g. $n = 1$.

II. Inductive Step:
  - We pretend to have a method that solves problem instances of size $< n$. This is often called the induction hypothesis.
  - For instances of size n, we devise a method that decomposes the given problem instance into smaller instances of size $< n$, e.g. $n - 1$ or $\lfloor n/2 \rfloor$. It is essential that this be instances of the same problem, just smaller.

2. Algorithm Design by Induction

We now introduce a methodology that allows us to derive first algorithmic approaches to many types of problems in which the sizes of problem instances depend on a parameter n. We are interested in an algorithm that can solve arbitrary problem instances of size n, for all $n \in \mathbf{N}$.

Basic Approach:

I. Base Case: We give solutions to ”small” problem instances, where n is at most some fixed constant, e.g. $n = 1$.

II. Inductive Step:
  - We pretend to have a method that solves problem instances of size $< n$. This is often called the induction hypothesis.
  - For instances of size n, we devise a method that decomposes the given problem instance into smaller instances of size $< n$, e.g. $n - 1$ or $\lfloor n/2 \rfloor$. It is essential that this be instances of the same problem, just smaller.

II. Cont.
   - We use the induction hypothesis to solve the smaller instances. This can be translated into recursive calls.
   - Finally, we give a method to recombine the smaller solutions into a solution of the original size-n problem.

Examples of this will be seen in the follwing sections (and in most of the remainder of this course).

II. Cont.

- We use the induction hypothesis to solve the smaller instances. This can be translated into recursive calls.
- Finally, we give a method to recombine the smaller solutions into a solution of the original size-n problem.

Examples of this will be seen in the follwing sections (and in most of the remainder of this course).

3. Selection Sort

We now directly apply the aforementioned framework to derive a relatively simple algorithm for the sorting problem.

3. Selection Sort

We now directly apply the aforementioned framework to derive a relatively simple algorithm for the sorting problem.

I. Base case for $n = 1$: Sorting one key value is easy, just do nothing. $\surd$

II. Inductive Step: Suppose we are given an array containing n keys.

3. Selection Sort

We now directly apply the aforementioned framework to derive a relatively simple algorithm for the sorting problem.

I. Base case for $n = 1$: Sorting one key value is easy, just do nothing. $\surd$

II. Inductive Step: Suppose we are given an array containing n keys.
   - Let us assume we know how to sort $(n-1)$ keys.
   - To reduce the problem size from n down to $(n-1)$, we remove one key from the array. Here we select the smallest key. (Hence "Selection Sort"). We need to make sure that the form of the problem does not change. The remaining keys must be stored in an array, at consecutive positions. To avoid a gap, we exchange the smallest key with the one at the first position.
   - We sort the remaining keys by induction, applying our hypothesis to the array, excluding the first position.
   - The smallest key is already in its correct position and, by induction, so are all other keys. Recombination is trivial.

3. Selection Sort

We now directly apply the aforementioned framework to derive a relatively simple algorithm for the sorting problem.

I. Base case for $n = 1$: Sorting one key value is easy, just do nothing. $\surd$

II. Inductive Step: Suppose we are given an array containing n keys.
  - Let us assume we know how to sort $(n-1)$ keys.
  - To reduce the problem size from n down to $(n-1)$, we remove one key from the array. Here we select the smallest key. (Hence "Selection Sort"). We need to make sure that the form of the problem does not change. The remaining keys must be stored in an array, at consecutive positions. To avoid a gap, we exchange the smallest key with the one at the first position.
  - We sort the remaining keys by induction, applying our hypothesis to the array, excluding the first position.
  - The smallest key is already in its correct position and, by induction, so are all other keys. Recombination is trivial.

3. Selection Sort

We now directly apply the aforementioned framework to derive a relatively simple algorithm for the sorting problem.

I. Base case for $n = 1$: Sorting one key value is easy, just do nothing. $\surd$

II. Inductive Step: Suppose we are given an array containing n keys.
  - Let us assume we know how to sort $(n-1)$ keys.
  - To reduce the problem size from n down to $(n-1)$, we remove one key from the array. Here we select the smallest key. (Hence "Selection Sort"). We need to make sure that the form of the problem does not change. The remaining keys must be stored in an array, at consecutive positions. To avoid a gap, we exchange the smallest key with the one at the first position.
  - We sort the remaining keys by induction, applying our hypothesis to the array, excluding the first position.
  - The smallest key is already in its correct position and, by induction, so are all other keys. Recombination is trivial.

3. Selection Sort

We now directly apply the aforementioned framework to derive a relatively simple algorithm for the sorting problem.

I. Base case for $n = 1$: Sorting one key value is easy, just do nothing. $\surd$

II. Inductive Step: Suppose we are given an array containing n keys.

  - Let us assume we know how to sort $(n-1)$ keys.
  - To reduce the problem size from n down to $(n-1)$, we remove one key from the array. Here we select the smallest key. (Hence ”Selection Sort”). We need to make sure that the form of the problem does not change. The remaining keys must be stored in an array, at consecutive positions. To avoid a gap, we exchange the smallest key with the one at the first position.
  - We sort the remaining keys by induction, applying our hypothesis to the array, excluding the first position.
  - The smallest key is already in its correct position and, by induction, so are all other keys. Recombination is trivial.

3. Selection Sort

We now directly apply the aforementioned framework to derive a relatively simple algorithm for the sorting problem.

I. Base case for $n = 1$: Sorting one key value is easy, just do nothing. $\surd$

II. Inductive Step: Suppose we are given an array containing n keys.
   - Let us assume we know how to sort $(n-1)$ keys.
   - To reduce the problem size from n down to $(n-1)$, we remove one key from the array. Here we select the smallest key. (Hence "Selection Sort"). We need to make sure that the form of the problem does not change. The remaining keys must be stored in an array, at consecutive positions. To avoid a gap, we exchange the smallest key with the one at the first position.
   - We sort the remaining keys by induction, applying our hypothesis to the array, excluding the first position.
   - The smallest key is already in its correct position and, by induction, so are all other keys. Recombination is trivial.

Let us suppose that the array can be accessed through a global variable $A[]$.

Algorithm:

```
void SelectionSort_recursive(unsigned $l, r$){
    if $(l = r)$ then return
    else let $k_s := \min\{k_l, k_{l+1}, \ldots, k_r\}$
        swap $A[k_l]$ and $A[k_s]$
        SelectionSort_recursive($l + 1, r$)
    fi
}
```

This function is initially called by "SelectionSort_recursive(1,n)".

Let us suppose that the array can be accessed through a global variable $A[]$.

Algorithm:

```
void SelectionSort_recursive(unsigned $l, r$){
  if $(l = r)$ then return
  else let $k_s := \min\{k_l, k_{l+1}, \ldots, k_r\}$
    swap $A[k_l]$ and $A[k_s]$
    SelectionSort_recursive($l + 1, r$)
  fi
}
```

This function is initially called by "SelectionSort_recursive(1,n)".

For the sake of efficiency, and sometimes code legibility, we often try to avoid recursion. An equivalent algorithm can be formulated using iteration rather than recursive calls. The strategy applied here is only slightly different.

Algorithm:

```
void SelectionSort_iterative(key $A[]$, unsigned $n$){
  for $i := 1$ to $n - 1$ do
    for $j := i + 1$ to $n$ do
      if $(A[j] < A[i])$ then
        swap $A[i]$ and $A[j]$
      fi
    od
  od
}
```

For the sake of efficiency, and sometimes code legibility, we often try to avoid recursion. An equivalent algorithm can be formulated using iteration rather than recursive calls. The strategy applied here is only slightly different.

Algorithm:

```
void SelectionSort_iterative(key $A[]$, unsigned $n$){
  for $i := 1$ to $n - 1$ do
    for $j := i + 1$ to $n$ do
      if $(A[j] < A[i])$ then
        swap $A[i]$ and $A[j]$
      fi
    od
  od
}
```

Complexity:

We measure the efficiency of the above algorithm by counting the number $c(n)$ of key comparisons needed to sort n keys. (This is the traditional way of comparing the complexities of sorting algorithms.)

Let us consider the iterative algorithm. In the i-th iteration of the outer for-loop, we have $(n-i)$ comparisons. ($A[i+1]$ vs. $A[i]$, $A[i+2]$ vs. $A[i]$, $\cdots$, $A[n]$ vs. $A[i]$). Hence:

$$c(n)=\sum_{i=1}^{n-1}(n-i)=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}=\frac{1}{2}(n^2-n)=\Theta(n^2).$$

This means we have just derived a quadratic-time sorting algorithm.

Complexity:

We measure the efficiency of the above algorithm by counting the number $c(n)$ of key comparisons needed to sort n keys. (This is the traditional way of comparing the complexities of sorting algorithms.)
Let us consider the iterative algorithm. In the i-th iteration of the outer for-loop, we have $(n-i)$ comparisons. ($A[i+1]$ vs. $A[i]$, $A[i+2]$ vs. $A[i]$, $\cdots$, $A[n]$ vs. $A[i]$). Hence:

$$c(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}(n^2-n) = \Theta(n^2).$$

This means we have just derived a quadratic-time sorting algorithm.

Complexity:

We measure the efficiency of the above algorithm by counting the number $c(n)$ of key comparisons needed to sort n keys. (This is the traditional way of comparing the complexities of sorting algorithms.)
Let us consider the iterative algorithm. In the i-th iteration of the outer for-loop, we have $(n-i)$ comparisons. ($A[i+1]$ vs. $A[i]$, $A[i+2]$ vs. $A[i]$, $\cdots$, $A[n]$ vs. $A[i]$). Hence:

$$c(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{1}{2}(n^2-n) = \Theta(n^2).$$

This means we have just derived a quadratic-time sorting algorithm.