



Complexity IBC028, Lecture 1

H. Geuvers

Institute for Computing and Information Sciences
Radboud University Nijmegen

Version: spring 2026





Outline

Organisation and Overview

Induction proofs

Substitution Method





About this course I

Lectures

- Teacher: Herman Geuvers
- Weekly, 2 hours, on Monday, 13:30–15:30 in HG00.307 (with an exception, like today)
- The lectures follow:
 - these slides, available via the webpage
 - some extra notes by Niels van der Weide (the webpage)
 - *Introduction to Algorithms*, “CLRS”, by Cormen, Leiserson, Rivest and Stein
OR *Algorithms Illuminated Omnibus Edition*, “Roughgarden”, by Tim Roughgarden.
 - older lecture notes by Hans Zantema (the webpage)
- Course URL:
www.cs.ru.nl/~herman/onderwijs/complexity2026/

Please check there first



About this course II

Exercises

- Weekly exercise classes, on Friday, 13:30-15:15
- You have been **automatically enrolled** for a **TA group** in Brightspace.
- Schedule:
 - Monday: “lecture n ” and “exercises n ” on the web
 - Next exercise class (Friday) you can work on “exercises n ”, ask questions, get answers for “exercises $n - 1$ ”.
 - Next **Tuesday**, before **11:00**: hand in “exercises n ” via Brightspace.
 - Before next exercise class: find your grade for “exercises n ” in Brightspace
- Your handed in exercises are graded by your TA.
- If e is the average grade of your **exercises**, $\frac{e}{10}$ is added to your exam grade as a **bonus**.



About this course III

Examination

- The final grade is composed of
 - the grade of your final (3hrs) exam, f ,
 - the average grade of your exercises, e ,
- The final grade is computed as follows.
 - If f is 5 or higher, the final grade is $\text{round}(\min(10, f + \frac{e}{10}))$
 - If f is below 5, the final grade is $\text{round}(f)$.
- The resit exam is a full 3hrs exam about the whole course. You keep the (average) grade of the exercises.
- If you fail again, you must start all over next year



Overview

Topics

- Techniques for computing the **complexity of algorithms**, especially recursive algorithms: substitution method, recursion tree method, the “master theorem”.
- Examples of algorithms and data structures and their complexity.
- **Complexity of problems**: the complexity classes **P** (polynomial complexity) and **NP**. The notion of **NP**-completeness and the open problem $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$.

Important:

⇒ Precise formal definitions and precise formal proofs



Complexity of algorithms

Time complexity of algorithm $A := \#$ steps it takes to execute A .

- what is a “step”?
- algorithm ... not “program”!
- $\#$ steps should be related to size of input

DEFINITION

Time complexity of algorithm A is f if for an input of size n , A takes $f(n)$ steps to compute the output.

Here, f is a function from \mathbb{N} to \mathbb{N} .

- We study **worst case complexity**: we want an upperbound that applies to all possible inputs.
- We study complexity “in the limit” and ignore a finite number of “outliers”: **asymptotic complexity**
- Constants and lower factors can be ignored: n^2 and $5n^2 + 3n + 7$ are “the same” complexity.



Asymptotic complexity

Complexity definitions: “big \mathcal{O} ”, “big Ω ”, “big Θ ” notation.

DEFINITION

For $f, g : \mathbb{N} \rightarrow \mathbb{N}$ functions, we define

- $f \in \mathcal{O}(g)$ if $\exists c \in \mathbb{R}_{>0} \exists N_0 \forall n > N_0 (f(n) \leq c g(n))$
- $f \in \Omega(g)$ if $\exists c \in \mathbb{R}_{>0} \exists N_0 \forall n > N_0 (c g(n) \leq f(n))$
- $f \in \Theta(g)$ if $f \in \mathcal{O}(g) \cap \Omega(g)$.

- $\mathcal{O}(g)$ is a **set** of functions (and similarly for $\Omega(g)$ and $\Theta(g)$):

$$\mathcal{O}(g) = \{f \mid \exists c \in \mathbb{R}_{>0} \exists N_0 \forall n > N_0 (f(n) \leq c g(n))\}$$

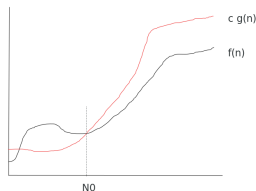
- Nevertheless, one always writes $f = \mathcal{O}(g)$, and we will follow that (abuse of) notation.
- Also: we follow the habit of writing $f(n)$ for the function $n \mapsto f(n)$, so we write $f(n) = \mathcal{O}(g(n))$ etc.



$$f(n) = \mathcal{O}(g(n))$$

$f(n) = \mathcal{O}(g(n))$ if

$$\exists c \in \mathbb{R}_{>0} \exists N_0 \forall n > N_0 (f(n) \leq c g(n))$$

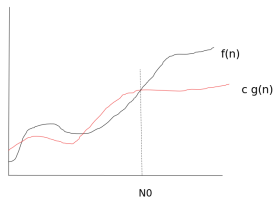




$$f(n) = \Omega(g(n))$$

$f(n) = \Omega(g(n))$ if

$$\exists c \in \mathbb{R}_{>0} \exists N_0 \forall n > N_0 (c g(n) \leq f(n))$$



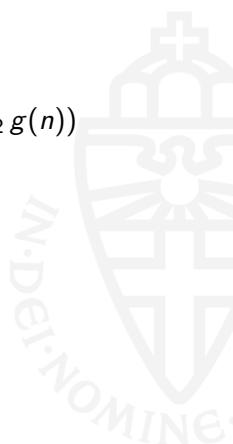
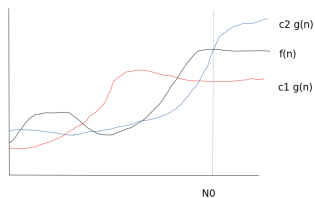


$$f(n) = \Theta(g(n))$$

$f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n)) \wedge f(n) = \Omega(g(n))$.

This is equivalent to saying:

$$\exists c_1, c_2 \in \mathbb{R}_{>0} \exists N_0 \forall n > N_0 (c_1 g(n) \leq f(n) \leq c_2 g(n))$$





Why can we ignore constants and lower factors

For $f(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$ with $a_k \neq 0$, we have

$$f(n) = \Theta(n^k)$$

We show this by an example: $7n^2 + 5n + 8 = \Theta(n^2)$





Space complexity

Apart from **running time** as a measure of complexity, one could also look at **memory consumption**.

This is called **space complexity**: the amount of **memory** it takes to execute an algorithm.

In the final lectures we will say something about space complexity, but for now we restrict to time complexity.

Just one observation:

space complexity \leq time complexity, because it takes at least n time steps to use n memory cells.



Strong induction (I)

The induction principle that we are used to is also called **structural induction**: it relies directly on the inductive structure of \mathbb{N} .

$$\frac{P(0) \quad \forall n \in \mathbb{N} (P(n) \rightarrow P(n+1))}{\forall n \in \mathbb{N} (P(n))}$$

We will often use **strong induction**, which relies on the fact that $<$ is well-founded on \mathbb{N} . (No infinite decreasing $<$ -sequences in \mathbb{N} .)

Strong induction:

$$\frac{\forall n \in \mathbb{N} (\forall k < n (P(k))) \rightarrow P(n)}{\forall n \in \mathbb{N} (P(n))}$$

Strong induction gives a stronger induction hypothesis: to prove $P(n)$ we may assume as (IH): $\forall k < n (P(k))$ (and not just $P(n-1)$).



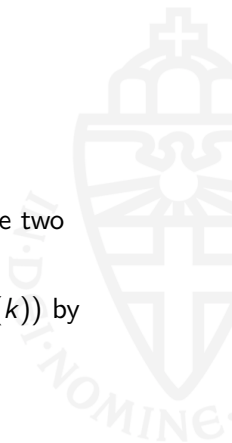
Strong induction (II)

Strong induction:

$$\frac{\forall n \in \mathbb{N} (\forall k < n (P(k))) \rightarrow P(n)}{\forall n \in \mathbb{N} (P(n))}$$

Strong induction is only seemingly stronger: in fact the two reasoning principles are equivalent.

Strong induction can be proved by proving $\forall k < n (P(k))$ by (structural) induction on n .





Fibonacci (I)

The Fibonacci function is defined as follows.

$$\begin{aligned} \text{fib}(0) &= 0 & \text{fib}(1) &= 1 \\ \text{fib}(n+2) &= \text{fib}(n+1) + \text{fib}(n) \end{aligned} \quad (1)$$

Claim: fib is exponential.

- So we are looking for an a such that $\text{fib}(n) = \Theta(a^n)$.
- Let's first try to find an a such that $\text{fib}(n) = a^n$.

Looking at equation (1), a should satisfy

$$a^{n+2} = a^{n+1} + a^n.$$

Knowing that $a \neq 0$, we obtain the quadratic equation $a^2 = a + 1$ that we can easily solve. Its solutions are called φ and $\hat{\varphi}$:

$$\varphi := \frac{1 + \sqrt{5}}{2} \approx 1.618 \qquad \hat{\varphi} := \frac{1 - \sqrt{5}}{2} \approx -0.618$$



Fibonacci (II)

$$\begin{aligned} \text{fib}(0) &= 0 & \text{fib}(1) &= 1 \\ \text{fib}(n+2) &= \text{fib}(n+1) + \text{fib}(n) \end{aligned} \quad (1)$$

$$\varphi := \frac{1 + \sqrt{5}}{2} \approx 1.618 \qquad \hat{\varphi} := \frac{1 - \sqrt{5}}{2} \approx -0.618$$

Neither φ^n nor $\hat{\varphi}^n$ provide solutions to the equations for fib, but

- the sum of two solutions to (1) is again a solution to (1)
- a solution to (1) multiplied with a c is again a solution to (1)

So we try to find c_1 and c_2 such that $\text{fib}(n) = c_1\varphi^n + c_2\hat{\varphi}^n$. This yields a unique solution and we obtain

$$\text{fib}(n) = \frac{1}{5}\sqrt{5}\varphi^n - \frac{1}{5}\sqrt{5}\hat{\varphi}^n.$$

As $\hat{\varphi}^n \rightarrow 0$, we can conclude that $\text{fib}(n) = \Theta(\varphi^n)$.



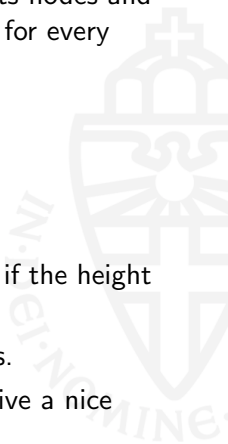
Binary search trees

A **binary search tree**, bst, is a binary tree that has, in its nodes and leaves, elements of an ordered structure (A, \sqsubseteq) , where for every node labeled a with left subtree ℓ and right subtree r ,

- for all labels x in ℓ : $x \sqsubseteq a$
- for all labels y in r : $a \sqsubseteq y$.

Often we have (\mathbb{N}, \leq) as ordered structure.

- A bst t is an efficient data-structure for searching if the height of t is $\mathcal{O}(\log k)$, for k the number of nodes in t .
- In a previous lecture you have seen red-black trees.
- We now introduce **AVL-trees**, also because they give a nice application of the fib function.





AVL trees

DEFINITION

An **AVL tree** is a binary search tree in which, for every node a , the difference between the height of the left and the right subtree of a is ≤ 1 .

We prove the following Theorem, that shows that AVL trees are efficient.

THEOREM

The height of an AVL tree t with k nodes is $\mathcal{O}(\log k)$.

The Theorem follows from our result that fib is exponential and a Lemma.

LEMMA

The number of nodes in an AVL tree of height n is $\geq \text{fib}(n)$.



The number of nodes in an AVL tree

LEMMA

The number of nodes in an AVL tree of height n is $\geq \text{fib}(n)$.

Proof. By (strong) induction on n .

IH: for all $p < n$: if t is an AVL tree of height p , then the number of nodes in t is $\geq \text{fib}(p)$.

To prove: if n is the height of an AVL tree s , then the number of nodes in s is $\geq \text{fib}(n)$.

Case distinction on n :

- $n = 0, 1$. Easy; check for yourself.
- $n \geq 2$. Then $n = 1 + \max(\text{height}(s_1), \text{height}(s_2))$, where s_1 and s_2 are the left and right subtrees of the top node of s . One of s_i has $\text{height}(s_i) = n - 1$, while the other has height $n - 1$ or $n - 2$. Using (IH) we derive that the number of nodes in s is $\geq 1 + \text{fib}(n - 1) + \text{fib}(n - 2)$, which is $\geq \text{fib}(n)$.





AVL trees are efficient

THEOREM

The height of an AVL tree t with k nodes is $\mathcal{O}(\log k)$.

Proof

Let $h(k) :=$ the largest height of an AVL tree with k nodes. So for every k there is an AVL tree with k nodes that has height $h(k)$.

Following the Lemma and our earlier result on fib: there is a $c > 0$ such that: $k \geq c\varphi^{h(k)}$ for all k (larger than some fixed N_0).

Therefore: $\log k \geq \log(c\varphi^{h(k)}) = \log c + h(k) \log \varphi$ and so

$$h(k) \leq \frac{\log k - \log c}{\log \varphi} = \mathcal{O}(\log k)$$



Divide and Conquer algorithms: Mergesort

For A an array p, r numbers with $p \leq r$, MergeSort(A, p, r) sorts the part $A[p], \dots, A[r]$ and leaves the rest of A unchanged.

```
MergeSort( $A, p, r$ ) = if  $|r - p| > 5$  then   $q := \lfloor \frac{p+r}{2} \rfloor$  ;
                                     MergeSort( $A, p, q$ );
                                     MergeSort( $A, q + 1, r$ );
                                     Merge( $A, p, q, r$ )
                               else InsertionSort( $A, p, r$ )
```

- Merge(A, p, q, r) merges the parts $A[p], \dots, A[q]$ and $A[q + 1], \dots, A[r]$. It is linear (in the length of A) and produces a sorted array (if the input arrays are sorted).
- We write a recurrence relation for $T(n)$, the time it takes to compute MergeSort(A, p, r), with $n = r - p$



Mergesort

For A an array p, r numbers, $\text{MergeSort}(A, p, r)$ sorts the part $A[p], \dots, A[r]$ and leaves the rest of A unchanged.

```

MergeSort(A, p, r) = if  $|r - p| > 5$  then
     $q := \lfloor \frac{p+r}{2} \rfloor$ ;
    MergeSort(A, p, q);
    MergeSort(A, q + 1, r);
    Merge(A, p, q, r)
else
    InsertionSort(A, p, r)
    
```

Recurrence equation for T of MergeSort

$$\begin{aligned}
 T(n) &\leq t_{\text{base}} && \text{for } n \leq 5 \\
 T(n) &= T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n) && \text{for } n > 5
 \end{aligned}$$

How can we solve this and compute T ?



The complexity of Mergesort(I)

THEOREM

If $T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$, then $T(n) = \mathcal{O}(n \log n)$.

Proof (by strong induction)

We have $c_0 > 0$ and N_0 such that $\forall n > N_0 (T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + c_0 n)$.
 Need to find: $c_1 > 0$ and N_1 such that $\forall n > N_1 (T(n) \leq c_1 n \log n)$.
 Take $c_1 \geq c_0$ large enough so that $T(n) \leq c_1 n \log n$ for $n = 1, \dots, N_0$.
 Let $N_1 > 3, N_0$. Then for $n > N_1$ we have $\lfloor \frac{n}{2} \rfloor < n$, so we can apply strong induction.

$$\begin{aligned}
 T(n) \leq 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c_0 n & \stackrel{\text{IH}}{\leq} 2c_1 \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor + c_0 n \\
 & \leq 2c_1 \frac{n}{2} \log \frac{n}{2} + c_1 n \\
 & \leq c_1 n (\log n - 1) + c_1 n \\
 & = c_1 n \log n \quad \square
 \end{aligned}$$



Back to Mergesort I

For MergeSort, we had $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n)$.

What if, in fact, we “round up” and have

$$T(n) = 2T(\lceil \frac{n}{2} \rceil) + \Theta(n)?$$

We show that it doesn't matter: If $T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + D + cn$, for fixed D and c , then $T(n) = \mathcal{O}(n \log n)$.

Define $U(n) := T(n + 2D)$. Then

$$\begin{aligned} U(n) = T(n + 2D) &\leq 2T\left(\left\lfloor \frac{n + 2D}{2} \right\rfloor + D\right) + c(n + 2D) \\ &\leq 2T\left(\left\lfloor \frac{n}{2} \right\rfloor + 2D\right) + c(n + 2D) \\ &\leq 2U\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 2cn \quad (\text{for } n \geq 2D) \end{aligned}$$

Earlier Theorem: $U(n) = \mathcal{O}(n \log n)$. So we also have $T(n) = \mathcal{O}(n \log n)$. □



Back to Mergesort II

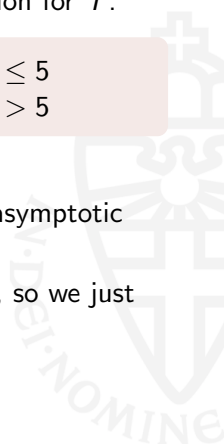
For MergeSort, we had the following recurrence equation for T .

$$\begin{aligned} T(n) &\leq t_{\text{base}} && \text{for } n \leq 5 \\ T(n) &= T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + \Theta(n) && \text{for } n > 5 \end{aligned}$$

We observe

- The values for small inputs don't matter for the asymptotic complexity.
- In computing complexity, we can ignore rounding, so we just consider $n \geq 2$ and

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





Sometimes we need “Induction loading”

Given: $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 1$, for $n \geq 2$, and $T(1) = b$

We guess that $T(n) = \mathcal{O}(n)$ and we try to show that $T(n) \leq c n$ for some appropriately chosen c (and $n > N$ for some chosen N).

$$\begin{aligned} T(n) &\leq c \frac{n}{2} + c \frac{n}{2} + 1 \\ &= cn + 1 \stackrel{??}{\leq} cn \quad \dots \text{no!} \end{aligned}$$

The trick is to add some constant: $T(n) \leq c n + d$.
 Try the proof again and figure out what c and d could be.

$$\begin{aligned} T(n) &\leq c \frac{n}{2} + d + c \frac{n}{2} + d + 1 \\ &= cn + 2d + 1 \\ &\leq cn + d \quad \text{for } d = -1 \text{ and any } c. \end{aligned}$$

For the base case: $T(1) = b \leq c - 1$, so take $c := b + 1$.
 We have $T(n) \leq (b + 1)n - 1$ for all $n \geq 1$, so $T(n) \in \mathcal{O}(n)$.



Substitution method: Changing variables

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$$

We **rename variables** and put $n = 2^m$ (and so $m = \log n$). Ignoring rounding off errors, we have

$$T(2^m) = 2T(2^{m/2}) + m$$

Consider this as a function in m : $S(m) = T(2^m)$ and we have

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

This is well-known and we have $S(m) = \mathcal{O}(m \log m)$.

We conclude that

$$T(n) = T(2^m) = S(m) \leq c(m \log m) = c(\log n \log \log n)$$

for some c .

So $T(n) = \mathcal{O}(\log n \log \log n)$.



Pitfalls in proving complexity

Suppose $T(1) = 1$ and $T(n) = T(n - 1) + n$ for $n > 1$.

Claim: then $T(n) = \mathcal{O}(n)$

Proof: By induction on n :

$$\begin{aligned} T(n) &= T(n - 1) + n \\ &\stackrel{\text{IH}}{=} \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n) \end{aligned}$$

\implies This is **WRONG!** We need to be **precise about functions and constants in induction proofs**:

$T(n) = \mathcal{O}(n)$ means: $\exists c \exists N_0 \forall n > N_0 (T(n) \leq c n)$

Correct reasoning:

$$\begin{aligned} T(n) &= T(n - 1) + n \\ &\leq c(n - 1) + n && \text{(for } n > N_0) \\ &= c n + n - c && \not\leq c n \end{aligned}$$

and the induction proof doesn't go through.



Substitution method: Example

Given $T(n) = 9T(\frac{n}{2}) + \Theta(n^3)$, prove that $T(n) = \mathcal{O}(n^3\sqrt{n})$.





Some final advice

- Make sure you can do induction proofs. See the exercises.
- Make sure you know how to compute with log, exponents etcetera. That means: you don't have to look up the "rules" but you know them by heart and you can apply them swiftly and correctly. (See e.g. Section 3.2 of the CLRS book.)
- Make sure you know how to compute with summations. (See e.g. Appendix A.1 of the CLRS book.)

