Complexity IBC028, Lecture 1

H. Geuvers

Institute for Computing and Information Sciences
Radboud University Nijmegen

Version: spring 2020

Outline

About this course I

Lectures

- Teachers: Herman Geuvers and Hans Zantema
- Weekly, 2 hours, on Tuesday, 13:30-15:15 (except for the carnaval week, then on Friday, 8:30-10:15)
- Presence not compulsory . . .
  - but active, polite attitude expected, when present
- The lectures follow:
  - these slides, available via the web
  - additional lecture notes by Hans Zantema, available via the web
  - *Introduction to Algorithms* by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
- Course URL:

  www.cs.ru.nl/~herman/onderwijs/complexity2020/

  Check there first, before you dare to ask/mail a question!

About this course II

Exercises

- Weekly exercise classes, on Friday, 8:30-10:15
  Except for Friday February 28, then there is a lecture.
  - Presence not compulsory
  - Answers (for old exercises) & Questions (for new ones)
- Schedule:
  - New exercises on the web: Tuesday
  - Next exercise meeting (Friday) you can ask questions
- At 3 points in the course, homework can be handed in with the assistant at the exercise class. This will be graded.
- If a is the average grade of your homework assignments, $\frac{a}{10}$ is added to your exam grade as a bonus.

About this course III

Exercise Classes

- 6 Assistants:

Eline Bovy	HG00.062	snr $= 0 \pmod 6$
Bas Hofmans	HFML0220	snr $= 1 \pmod 6$
Thomas van Ouwerkerk	HG00.310	snr $= 2 \pmod 6$
Lars van Rhijn	HG01.028	snr $= 3 \pmod 6$
Gijs Hendriksen	HG00.086	snr $= 4 \pmod 6$
Deivid Rodrigues do Vale	HG00.068	snr $= 5 \pmod 6$

About this course IV

Examination

- The final grade is composed of
  - the grade of your final (2hrs) exam, **f**,
  - the average grade of your exercises, **a**,
- Your final grade is $\min(10, \mathbf{f} + \frac{\mathbf{a}}{10})$
  - The re-exam is a full 2hrs exam about the whole course. You keep the (average) grade of the exercises.
- If you fail again, you must start all over next year (including re-doing new exercises)

Overview

Topics

- Techniques for computing the complexity of algorithms, especially recursive algorithms; the "master theorem".
- Examples of algorithms and data structures and their complexity; geometric algorithms.
- Complexity classes: P (polynomial complexity), NP; NP-completeness and $P \stackrel{?}{=} NP$?

$\Longrightarrow$ 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

Time complexity of algorithm A is f if
for any 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 ignore constants: n^2 and $5n^2 + 7$ are “the same” complexity.
- we study complexity “in the limit” and ignore a finite number of “outliers”: asymptotic complexity

Space complexity

Apart from running time as a measure of complexity, one could also look at memory consumption. This is called space

complexity': 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.

Asymptotic complexity

Complexity definitions: "big $\mathcal{O}$", "big Ω", "big Θ" notation.
For $f, g : \mathbb{N} \to \mathbb{N}$ a functions,

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

Example of a recursive program and its complexity (I)

An naive (inefficient) recursive algorithm to compute 2^n: for n a natural number,

```
A(n) =   if n=0   then 1
                  else A(n-1) + A(n-1)
```

What is the complexity of A?

Define $T(n) := \#$ steps it takes to execute $A(n)$.

Assuming 1 step for addition and no steps for the case-distinction, we have

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

We want to find a closed expression for $T(n)$ so we can try some values.

Example of a recursive program and its complexity (I)

Educated guess: $T(n) = 2^{n+1} - 1$. We now prove that this is actually the case.

THEOREM. For all $n \in \mathbb{N}$, $T(n) = 2^{n+1} - 1$

Proof by induction on n

- base case, $n = 0$: $T(0) = 1 = 2^1 - 1$ ✓
- step case: suppose (IH) $T(n) = 2^{n+1} - 1$, we need to prove (TP) $T(n+1) = 2^{n+2} - 1$.

$$\begin{aligned} T(n+1) &= 1 + 2T(n) \\ &\overset{IH}{=} 1 + 2(2^{n+1} - 1) \\ &= 1 + 2^{n+2} - 2 \\ &= 2^{n+2} - 1 \end{aligned}$$

Strong induction (I)

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

$$\frac{P(0) \qquad \forall n \in \mathbb{N}\,(P(n) \to P(n+1))}{\forall n \in \mathbb{N}\,(P(n))}$$

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:

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

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

Strong induction (II)

Strong induction:

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

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.

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

Claim: fib is exponential.

So we are looking for an a such that $\mathrm{fib}(n) \approx a^n$ for all n.

Looking at equation (1), we need to find an a that satisfies

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

Knowing that $a \neq 0$, we obtain the quadratic equation $a^2 = a + 1$ that we can easily solve.It's 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)

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

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

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 $\hat{\varphi}^n \to 0$, we can conclude that $\text{fib} \in \Theta(n \mapsto \varphi^n)$.

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 rightsubtree 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 is an efficient data-structure for storing search data if the tree is balanced: searching in a tree t is efficient if the height 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.

The following Theorem shows that AVL trees are efficient.

Theorem

The height of an AVL tree t is $\mathcal{O}(\log k)$, where k is the number of nodes in t.

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 $d(k) :=$ 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 $d(k)$.
Following the Lemma and our earlier result on fib: there is a $c > 0$ such that: $k \geq c\varphi^{d(k)}$ for all k.
Therefore: $\log k \geq \log(c\varphi^{d(k)}) = \log c + d(k) \log \varphi$ and so

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

Divide and Conquer algorithms: Mergesort

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

$$\text{MergeSort}(A, p, r) = \texttt{if } p < r \texttt{ then} \quad \begin{array}{l} q := \left\lfloor \dfrac{p+r}{2} \right\rfloor; \\ \text{MergeSort}(A, p, q); \\ \text{MergeSort}(A, q+1, r); \\ \text{Merge}(A, p, q, r) \end{array}$$

- Merge(A, p, q, r) takes A and merges the parts $A[p], \ldots A[q]$ and $A[q+1], \ldots A[r]$. It is linear and produces a sorted array (if the input arrays are sorted). See the book.
- 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 $A[p], \ldots A[r]$ and leaves the rest of A unchanged.

$$\text{MergeSort}(A, p, r) = \texttt{if } p < r \texttt{ then} \quad \begin{aligned} & q := \left\lfloor \frac{p+r}{2} \right\rfloor; \\ & \text{MergeSort}(A, p, q); \\ & \text{MergeSort}(A, q+1, r); \\ & \text{Merge}(A, p, q, r) \end{aligned}$$

Recurrence equation for T of MergeSort

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(\frac{n}{2}) + \Theta(n) \end{aligned}$$

The complexity of Mergesort (I)

$$\text{MergeSort}(A, p, r) = \texttt{if } p < r \quad \texttt{then} \quad q := \left\lfloor \frac{p+r}{2} \right\rfloor; \text{MergeSort}(A, p, q);$$
$$\text{MergeSort}(A, q+1, r); \text{Merge}(A, p, q, r)$$

We find that

- $T(1) = 1$
- $T(n) = 2T(\frac{n}{2}) + \Theta(n)$ (for $n \geq 2$)

Theorem

If $T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$, then

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

The complexity of Mergesort(II)

Theorem

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

Proof (by strong induction)

Suppose $T(n) \leq 2T(\lfloor \frac{n}{2} \rfloor) + cn$ for some constant c.
Take $c' \geq c$ large enough so that $T(n) \leq c' n \log n$ for $n = 2, 3$.
Let $n > 3$. Then $\lfloor \frac{n}{2} \rfloor < n$, so we can apply strong induction.

$$
\begin{aligned}
T(n) \leq 2T(\left\lfloor \frac{n}{2} \right\rfloor) + cn \quad & \overset{\text{IH}}{\leq} \quad 2c' \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor + c' n \\
& \leq \quad 2c' \frac{n}{2} \log \frac{n}{2} + c' n \\
& \leq \quad c' n(\log n - 1) + c' n \\
& = \quad c' n \log n \qquad \square
\end{aligned}
$$

Back to Mergesort

For MergeSort, we had $T(n) = 2T(\frac{n}{2}) + \Theta(n)$.
What if, in fact, we need to "round up" and have

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

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

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

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

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

Pitfalls in proving complexity

Suppose $T(1) = 1$ and $T(n) = T(n-1) + n$ for $n > 1$.
Claim: then $T(n) \in \mathcal{O}(n)$
Proof: By induction on n:

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

$\Longrightarrow$ This is WRONG! We need to be precise about functions and constants in induction proofs:
$T(n) \in \mathcal{O}(n)$ means: $\exists c \exists N \forall n > N (T(n) \leq cn)$
Correct reasoning:

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

and the induction proof doesn't go through.