

Departement of Computer Science

5 October 2026

Johannes Lengler, Markus Püschel

Manuel Christalla, Paul Paschmanns, Andor Vári-Kakas, Manuel Wiedmer

Algorithms & Data Structures

Exercise sheet 3

HS 26

The solutions for this sheet are submitted on Moodle until 11 October 2026, 23:59.

Exercises that are marked with a * are more challenging. We recommend first focusing on the other exercises.

You can use results from previous parts without solving those parts.

Asymptotic Notation

The following two definitions are closely related to the O -notation and are also useful in the running time analysis of algorithms. Let N be again a set of possible inputs (e.g. $N = \{n_0, n_0 + 1, \dots\}$ for $n_0 \in \mathbb{N}$).

Definition 1 (Ω -Notation). For $f : N \rightarrow \mathbb{R}^+$,

$$\begin{aligned}\Omega(f) &:= \{g : N \rightarrow \mathbb{R}^+ \mid f \leq O(g)\} \\ &= \{g : N \rightarrow \mathbb{R}^+ \mid \text{there exists } C > 0 \text{ such that for all } n \in N \ g(n) \geq C \cdot f(n)\}.\end{aligned}$$

We write $g \geq \Omega(f)$ instead of $g \in \Omega(f)$.

Ω -notation is a way to lower bound the asymptotic behavior of a function, like the runtime of an algorithm.

Definition 2 (Θ -Notation). For $f : N \rightarrow \mathbb{R}^+$,

$$\begin{aligned}\Theta(f) &:= \{g : N \rightarrow \mathbb{R}^+ \mid g \leq O(f) \text{ and } f \leq O(g)\} \\ &= \{g : N \rightarrow \mathbb{R}^+ \mid \text{there exist } C_1, C_2 > 0 \text{ s.t. for all } n \in N \ C_1 \cdot f(n) \leq g(n) \leq C_2 \cdot f(n)\}.\end{aligned}$$

We write $g = \Theta(f)$ instead of $g \in \Theta(f)$.

Θ -notation is a way to simultaneously upper and lower bound the asymptotic behavior of a function.

In other words, for two functions $f, g : N \rightarrow \mathbb{R}^+$ we have

$$g \geq \Omega(f) \Leftrightarrow f \leq O(g)$$

and

$$g = \Theta(f) \Leftrightarrow g \leq O(f) \text{ and } f \leq O(g).$$

We can restate Theorem 1 from exercise sheet 2 as follows.

Theorem 1. Let N be an infinite subset of $\mathbb{N}$ and $f : N \rightarrow \mathbb{R}^+$ and $g : N \rightarrow \mathbb{R}^+$.

- If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$, then $f \leq O(g)$, but $f \neq \Theta(g)$.

- If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$, then $f = \Theta(g)$.
- If $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$, then $f \geq \Omega(g)$, but $f \neq \Theta(g)$.

Exercise 3.1 Asymptotic growth.

(a) Prove or disprove the following statements. Justify your answer.

- (1) For all $a, b > 1$, suppose $n \in \mathbb{N}$ and $n > \max\{a, b\}$, then $\log_a(n) = \Theta(\log_b(n))$.
- (2) For all $a, b \geq 1$, suppose $n \in \mathbb{N}$, then $a^n = \Theta(b^n)$

(b) (1) Prove that $\lim_{n \rightarrow \infty} \frac{n}{\log(n)} = \infty$.

Hint: Use L'Hôpital's rule.

(2) Prove that $\lim_{n \rightarrow \infty} n(e^{1/n} - 1) = 1$.

You may use the following variant of L'Hôpital's rule:

Theorem 2 (L'Hôpital's rule (going to 0)). Assume that functions $f : \mathbb{R}^+ \rightarrow \mathbb{R}$ and $g : \mathbb{R}^+ \rightarrow \mathbb{R}$ are differentiable, $\lim_{x \rightarrow \infty} f(x) = \lim_{x \rightarrow \infty} g(x) = 0$ and for all $x \in \mathbb{R}^+$, $g'(x) \neq 0$. If $\lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)} = C \in \mathbb{R}$ or $\lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)} = \infty$, then

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}.$$

(3) Prove that for $n \geq 3$

$$\frac{n^{1/n} - 1}{n} = \Theta\left(\frac{\ln(n)}{n^2}\right).$$

You may use results from the earlier exercises.

You may use the following fact:

Let $f, g : \mathbb{R}^+ \rightarrow \mathbb{R}$. Suppose that $\lim_{n \rightarrow \infty} g(n) = \infty$ and there exists some constant $C \in \mathbb{R}$ such that $\lim_{n \rightarrow \infty} f(n) = C$. Then $\lim_{n \rightarrow \infty} f(g(n)) = C$.

Exercise 3.2 Substring counting (2 bonus points).

Given an n -bit bitstring $S[0..n-1]$ (i.e. $S[i] \in \{0, 1\}$ for $i = 0, 1, \dots, n-1$), and an integer $k \geq 0$, we would like to count the number of nonempty contiguous substrings of S with exactly k ones. Assume $n \geq 2$.

For example, when $S = \text{"0110"}$ and $k = 2$, there are 4 such substrings: "011", "11", "110", and "0110".

- Design a "naive" algorithm that solves this problem with a runtime of $O(n^3)$. Describe the algorithm using pseudocode. Justify the runtime (you don't need to provide a formal proof, but you should state your reasoning).
- We say that a bitstring S' is a (*non-empty*) *prefix* of a bitstring S if S' is of the form $S[0..i]$ where $0 \leq i < \text{length}(S)$. For example, the prefixes of $S = \text{"0110"}$ are "0", "01", "011" and "0110".

Given an n -bit bitstring S , we would like to compute a table T indexed by $0..n$ such that for all i , $T[i]$ contains the number of prefixes of S with exactly i ones.

For example, for $S = \text{"0110"}$, the desired table is $T = [1, 1, 2, 0, 0]$, since, of the 4 prefixes of S , 1 prefix contains zero "1", 1 prefix contains one "1", 2 prefixes contain two "1", and 0 prefixes contain three "1" or four "1".

Design an algorithm `PREFIXTABLE` that computes T from S in time $O(n)$, assuming S has size n . Describe the algorithm using pseudocode. Justify the runtime (you don't need to provide a formal proof, but you should state your reasoning).

Remark: This algorithm can also be applied on a reversed bitstring to compute the same table for all suffixes of S . In the following, you can assume an algorithm `SUFFIXTABLE` that does exactly this.

- (c) Consider an integer $m \in \{0, 1, \dots, n - 2\}$. Using `PREFIXTABLE` and `SUFFIXTABLE`, design an algorithm `SPANNING`(m, k, S) that returns the number of substrings $S[i..j]$ of S that have exactly k ones and such that $i \leq m < j$.

For example, if $S = \text{"0110"}$, $k = 2$, and $m = 0$, there exist exactly two such strings: "011" and "0110". Hence, `SPANNING`(m, k, S) = 2.

Describe the algorithm using pseudocode. Mention and justify the runtime of your algorithm (you don't need to provide a formal proof, but you should state your reasoning).

Hint: Each substring $S[i..j]$ with $i \leq m < j$ can be obtained by concatenating a string $S[i..m]$ that is a suffix of $S[0..m]$ and a string $S[m + 1..j]$ that is a prefix of $S[m + 1..n - 1]$.

- (d)* Using `SPANNING`, design an algorithm with a runtime¹ of at most $O(n \log n)$ that counts the number of nonempty substrings of an n -bit bitstring S with exactly k ones. (You can assume that n is a power of two.)

Justify its runtime. You don't need to provide a formal proof, but you should state your reasoning. (This question does not count toward the bonus points.)

Hint: Use the recursive idea from the lecture.

Exercise 3.3 Counting function calls in loops (2 bonus points).

For each of the following code snippets, compute the number of calls to f as a function of $n \in \mathbb{N}$. Provide **both** the exact number of calls and a maximally simplified asymptotic bound in Θ notation.

Algorithm 1

- (a) $i \leftarrow 0$
while $i \leq n$ **do**
 $f()$
 $f()$
 $i \leftarrow i + 1$
 $j \leftarrow 0$
while $j \leq 2n$ **do**
 $f()$
 $j \leftarrow j + 1$
-

¹For this running time bound, we let n range over natural numbers that are at least 2 so that $n \log(n) > 0$.

Algorithm 2

(b) $i \leftarrow 1$
 while $i \leq n$ **do**
 $j \leftarrow 1$
 while $j \leq i^3$ **do**
 $f()$
 $j \leftarrow j + 1$
 $i \leftarrow i + 1$

Hint: You may use $\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$.

Exercise 3.4 *Iterative squaring.*

In this exercise you are going to develop an algorithm to compute powers a^n , with $a \in \mathbb{Z}$ and $n \in \mathbb{N}$, efficiently.

- (a) Assume that n is even, and that you already know an algorithm $A_{n/2}(a)$ that efficiently computes $a^{n/2}$, i.e., $A_{n/2}(a) = a^{n/2}$.

Given the algorithm $A_{n/2}$, design an efficient algorithm $A_n(a)$ that computes a^n . (You don't need to argue correctness or runtime.)

- (b) Let $n = 2^k$, for $k \in \mathbb{N}_0$. Find an algorithm that computes a^n efficiently. Describe your algorithm using pseudo-code. (You don't need to argue correctness or runtime.)
- (c) Determine the number of integer multiplications required by your algorithm for part (b) in O -notation. You may assume that bookkeeping operations don't cost anything. This includes handling of counters, computing $n/2$ from n , etc.
- (d) Let $\text{Power}(a, n)$ denote your algorithm for the computation of a^n from part b). Prove the correctness of your algorithm via mathematical induction for all $n \in \mathbb{N}$ that are powers of two.

In other words: show that $\text{Power}(a, n) = a^n$ for all $n \in \mathbb{N}$ of the form $n = 2^k$ for some $k \in \mathbb{N}_0$.

In your solution, you should address the base case, the induction hypothesis and the induction step.

- (e)* Design an algorithm that can compute a^n for a general $n \in \mathbb{N}$, i.e., n does not need to be a power of two. The algorithm should use $O(\log n)$ integer multiplications for $n \geq 2$. You don't need to argue about correctness or runtime.

Hint: Generalize the idea from part (a) to the case where n is odd, i.e., there exists $k \in \mathbb{N}_0$ such that $n = 2k + 1$.

Exercise 3.5 * *Finding the maximum in constant time.*

Consider the recursive maximum algorithm from Exercise 2.2 and the following argument.

Claim. The algorithm determines the maximum of n elements in $O(1)$ time.

Base Case. For $n = 1$, the algorithm returns the only element in $O(1)$ time.

Induction Hypothesis. Assume that the algorithm determines the maximum of k elements in $O(1)$ time for some $k \geq 1$.

Inductive Step. For $k + 1$ elements, we find the maximum of the first k elements in $O(1)$ time by the induction hypothesis. One additional comparison takes constant time, so the total time is $O(1)$. This proves the induction step.

Identify the flaw in this proof. What does the induction argument actually establish?