



Departement of Computer Science
Johannes Lengler, David Steurer
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

9 October 2023

Algorithms & Data Structures

Exercise sheet 3

HS 23

The solutions for this sheet are submitted at the beginning of the exercise class on 16 October 2023.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

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.

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

$$\Omega(f) := \{g : N \rightarrow \mathbb{R}^+ \mid f \leq O(g)\}.$$

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

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

$$\Theta(f) := \{g : N \rightarrow \mathbb{R}^+ \mid g \leq O(f) \text{ and } f \leq O(g)\}.$$

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

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 (Theorem 1.1 from the script). 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 (2 points).

For all the following functions the variable n ranges over \mathbb{N} .

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

(1) $\frac{1}{5}n^3 \geq \Omega(10n^2)$

(2) $n^2 + 3n = \Theta(n^2 \log(n))$

(3) $5n^4 + 3n^2 + n + 8 = \Theta(n^4)$

(4) $3^n \geq \Omega(2^n)$

(b) Prove the following statements.

Hint: For these examples, computing the limits as in Theorem 1 is hard or the limits do not even exist. Try to prove the statements directly with inequalities as in the definition of the O -notation.

(1) $(\sin(n) + 2)n = \Theta(n)$

Hint: For any $x \in \mathbb{R}$ we have $-1 \leq \sin(x) \leq 1$.

(2) $\sum_{i=1}^n \sum_{j=1}^i j = \Theta(n^3)$

Hint: In order to show $n^3 \leq O(\sum_{i=1}^n \sum_{j=1}^i j)$, you can use exercise 1.3.

(3) $\log(n^4 + n^3 + n^2) \leq O(\log(n^3 + n^2 + n))$

(4)* $\sum_{i=1}^n \sqrt{i} = \Theta(n\sqrt{n})$

Hint: Recall again exercise 1.3 and try to do an analogous computation here.

Exercise 3.2 Substring counting.

Given a n -bit bitstring S (an array over $\{0, 1\}$ of size $n \in \mathbb{N}$), and an integer $k \geq 0$, we would like to count the number of nonempty substrings of S with exactly k ones. For example, when $S = \text{"0110"}$ and $k = 2$, there are 4 such substrings: "011", "11", "110", and "0110".

(a) Design a "naive" algorithm that solves this problem with a runtime of $O(n^3)$. Justify its runtime and correctness.

(b) 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 a 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 prefix contains three "1" or four "1".

Describe an algorithm PREFIXTABLE that computes T from S in time $O(n)$, assuming S has size n .

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) Let S be a n -bit bitstring. Consider an integer $m \in \{0, \dots, n-1\}$, and divide the bitstring S into two substrings $S[0..m]$ and $S[m+1..n-1]$. Using PREFIXTABLE and SUFFIXTABLE, describe 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 $i \leq m < j$. What is its complexity?

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

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 a n -bit bitstring S with exactly k ones. (You can assume that n is a power of two.)

Hint: Use the recursive idea from the lecture.

Exercise 3.3 Counting function calls in loops (1 point).

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**
 $f()$
 $j \leftarrow j + 1$
-

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: See Exercise 1.4.

Exercise 3.4 Fibonacci numbers.

There are a lot of neat properties of the Fibonacci numbers that can be proved by induction. Recall that the Fibonacci numbers are defined by $f_0 = 0$, $f_1 = 1$ and the recursion relation $f_{n+1} = f_n + f_{n-1}$ for all $n \geq 1$. For example, $f_2 = 1$, $f_5 = 5$, $f_{10} = 55$, $f_{15} = 610$.

- (a) Prove that $f_n \geq \frac{1}{3} \cdot 1.5^n$ for $n \geq 1$.

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

- (b) Write an $O(n)$ algorithm that computes the n th Fibonacci number f_n for $n \in \mathbb{N}$.

Remark: As shown in part (a), f_n grows exponentially (e.g., at least as fast as $\Omega(1.5^n)$). On a physical computer, working with these numbers often causes overflow issues as they exceed variables' value limits. However, for this exercise, you can freely ignore any such issue and assume we can safely do arithmetic on these numbers.

- (c) Given an integer $k \geq 2$, design an algorithm that computes the largest Fibonacci number f_n such that $f_n \leq k$. The algorithm should have complexity $O(\log k)$. Prove this.

Remark: Typically we express runtime in terms of the size of the input n . In this exercise, the runtime will be expressed in terms of the input value k .

Hint: Use the bound proved in part (a).

Exercise 3.5 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. For this exercise, we will treat multiplication of two integers as a single elementary operation, i.e., for $a, b \in \mathbb{Z}$ you can compute $a \cdot b$ using one operation.

- (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 .
- (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.
- (c) Determine the number of elementary operations (i.e., 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 $k \in \mathbb{N}_0$.

- (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.

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