Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Kasper Lindberg, Lucas Slot, Hongjie Chen, Manuel Wiedmer

7 October 2024

Algorithms & Data Structures — Exercise sheet 3 — HS 24

The solutions for this sheet are submitted on Moodle until 13 October 2024, 23:59.

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 $f : N \to \mathbb{R}_+$,

$$\Omega(f) \coloneqq \{g : N \to \mathbb{R}_+ \mid f \leq O(g)\}.$$

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

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

$$\Theta(f) \coloneqq \{g : N \to \mathbb{R}_+ \mid g \leq O(f) \text{ and } f \leq O(g)\}.$$

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

In other words, for two functions $f, g : N \to \mathbb{R}_+$ 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 $f : N \to \mathbb{R}_+$ and $g : N \to \mathbb{R}_+$.*

- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f \leq O(g)$, but $f \neq \Theta(g)$.*
- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}_+$, then $f = \Theta(g)$.*
- *If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, then $f \geq \Omega(g)$, but $f \neq \Theta(g)$.*

Exercise 3.1 *Asymptotic growth* **(2 points)**.

For all the following functions $n \in \mathbb{N}$ and $n \geq 2$.

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

(1) $3n^5 + 5n^3 = \Theta(4n^4)$

(2) $n^2 + n\log(n) \geq \Omega(n^2 \log(n))$

(3) $\frac{1}{6}n^6 + 10n^4 + 100n^3 = \Theta(6n^6)$

(4) $3^n \geq \Omega(n^{3/\ln(n)} e^n)$

(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) $\sqrt{n^2 + n + 1} = \Theta(n)$

(2) $\sum_{i=1}^{n} \log(i^i) \geq \Omega(n^2 \log n)$

Hint: *Recall exercise 1.2 and try to do an analogous computation here.*

(3) $\log(n^2 + n) = \Theta(\log(n + 1))$

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

Hint: *For the lower bound, recall exercise 1.2 and try to do an analogous computation here. For the upper bound, first prove the following inequality $\frac{1}{\sqrt{i}} \leq 2(\sqrt{i} - \sqrt{i-1})$ for all $i \in \mathbb{N}$ with $i \geq 1$. Then analyze the new sum with these bounds.*

Exercise 3.2 *Substring counting.*

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

For example, when $S =$ "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 $O(n^3)$. Describe the algorithm using pseudocode. Justify the runtime (you don't need to provide a formal proof, but you should state your reasoning).

(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 $0 \leq i < \mathsf{length}(S)$. For example, the prefixes of $S =$ "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 $S =$ "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".

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 $m \in \{0, 1, \ldots, n-2\}$. 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 $S =$ "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 runtime[1] 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.)

Justify its runtime. You don't need to provide a formal proof, but you should state your reasoning.

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. In your expression for the exact number of calls, you are allowed to use summation signs. For example, '$\sum_{k=1}^{2n-1}(k+2) + 32n$' is a valid expression. In your simplified Θ notation this is not allowed. Furthermore, it should not have any unnecessary terms or factors. For example, '$\Theta(3n+2)$' is not valid.

Algorithm 1

(a)
```
i ← 0
while i ≤ n do
    i ← i + 1
    f()
    j ← 1
    while j ≤ n do
        f()
        f()
        j ← j + 1
```

[1]For 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 ← 1
while i ≤ 2n do
    j ← 1
    while j ≤ i^3 do
        k ← n
        while k ≥ 1 do
            f()
            k ← k − 1
        j ← j + 1
    i ← i + 1
```

Hint: *See Exercise 1.2. You are allowed to use any statement from that exercise without proof.*

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 $n \geq 1$.

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

(b) Design an $O(n)$ algorithm that computes the nth Fibonacci number f_n for $n \in \mathbb{N}$. Describe the algorithm using pseudocode. Justify the runtime (you don't need to provide a formal proof, but you should state your reasoning).

Remark: 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 $O(\log k)$. Describe the algorithm using pseudocode and formally prove its runtime is $O(\log k)$.

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.

(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 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 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. 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 $k \in \mathbb{N}$ such that $n = 2k + 1$.*