Eidgenössische
Technische Hochschule
Zürich

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

Departement of Computer Science
Markus Püschel, David Steurer
François Hublet, Goran Zuzic, Tommaso d'Orsi, Jingqiu Ding

10 October 2022

Algorithms & Data Structures — Exercise sheet 3 — HS 22

The solutions for this sheet are submitted at the beginning of the exercise class on 17 October 2022.

Exercises/questions marked by * are "challenge exercises". They do not count towards bonus points.

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

Exercise 3.1 *Some properties of O-Notation.*

Let $f : \mathbb{R}^+ \to \mathbb{R}^+$ and $g : \mathbb{R}^+ \to \mathbb{R}^+$.

(a) Show that if $f \leq O(g)$, then $f^2 \leq O(g^2)$.

(b) Does $f \leq O(g)$ imply $2^f \not\leq O(2^g)$? Prove it or provide a counterexample.

Exercise 3.2 *Substring counting* **(1 point)**.

Given a n-bit bitstring S (an array over $\{0, 1\}$ of size 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 $S =$ "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 $0 \leq i < \mathsf{length}(S)$. 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".

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 $m \in \{0, \ldots, n-1\}$, and divide 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 $S =$ "0110", $k = 2$, and $m = 0$, there exist exactly two such strings: "011" and "0110". Hence, 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 runtime 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. Provide **both** the exact number of calls and a maximally simplified, tight asymptotic bound in big-O notation.

Algorithm 1

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

Algorithm 2

(b)
```
i ← 0
while i² ≤ n do
    f()
    f()
    for j ← 1, ..., n do
        f()
    i ← i + 1
```

Exercise 3.4 *Fibonacci Revisited* **(1 point)**.

In this exercise we continue playing with the Fibonacci sequence.

(a) Write an $O(n)$ algorithm that computes the nth Fibonacci number. As a reminder, Fibonacci numbers are a sequence defined as $f_0 = 0$, $f_1 = 1$, and $f_{n+2} = f_{n+1} + f_n$ for all integers $n \geq 0$.

Remark: As shown in the last week's exercise sheet, 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.

(b) 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 2.2.(b).*

*(c) Given an integer $k \geq 2$, consider the following algorithm:

Algorithm 3

while $k > 0$ **do**
  find the largest n such that $f_n \leq k$
  $k \leftarrow k - f_n$

Prove that the loop body is executed at most $O(\log k)$ times.

Hint: *First, prove that* $f_{n-1} \geq \frac{1}{2} \cdot f_n$ *for all* n.

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 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 $k \in \mathbb{N}_0$.

*(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* $n = 2k + 1$.