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 $f^2 \leq O(g^2)$.

Solution:

Assume that $f \leq O(g)$. Then we can find $T, C \in \mathbb{R}^+$ such that for all $x \geq T$, we have $f(x) \leq C \cdot g(x)$. For all $x \geq T$, we get $f^2(x) = f(x) \cdot f(x) \leq (C \cdot g(x)) \cdot (C \cdot g(x)) = C^2 \cdot g^2(x)$, hence $f^2 \leq O(g^2)$.

(b) Does $f \leq O(g)$ imply $2^f \leq O(2^g)$? Prove it or provide a counterexample.

Solution:

The implication does **not** hold.

Consider $f(n) = 2n$, $g(n) = n$. Obviously, $f \leq O(g)$. However,

$$\lim_{n\to\infty} \frac{2^{f(n)}}{2^{g(n)}} = \lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} 2^n = \infty\,,$$

hence by Theorem 1 of Exercise sheet 1, $2^f \not\leq O(2^g)$.

Another important example is $f(n) = \log_2 n$ and $g(n) = \log_4 n$. As we already showed, $f \leq O(g)$. However, $2^{f(n)} = n$ and $2^{g(n)} = \sqrt{n}$, so by Theorem 1 of Exercise sheet 1, $2^f \not\leq O(2^g)$.

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.

Solution:

We can for example use the following algorithm:

Algorithm 1 Naive substring counting

```
$c \leftarrow 0$                                   ▷ Initialize counter of substrings with $k$ ones
for $i \leftarrow 0, \dots, n-1$ do                ▷ Enumerate all nonempty substrings $S[i..j]$
    for $j \leftarrow i, \dots, n-1$ do
        $x \leftarrow 0$                           ▷ Initialize counter of ones
        for $\ell \leftarrow i, \dots, j$ do       ▷ Count ones in substring
            if $S[\ell] = 1$ then
                $x \leftarrow x + 1$
        if $x = k$ then                            ▷ If there are $k$ ones in substring, increment $c$
            $c \leftarrow c + 1$
return $c$                                         ▷ Return number of substrings with $k$ ones
```

We perform at most n iterations of each loop, leading to a total runtime in $O(n^3)$. The correctness directly follows from the description of the algorithm (see comments above).

(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.

Solution:

Algorithm 2

```
function PREFIXTABLE($S$)
    $T \leftarrow \texttt{int}[n+1]$
    $s \leftarrow 0$
    for $i \leftarrow 0, \dots, n-1$ do
        $s \leftarrow s + S[i]$
        $T[s] \leftarrow T[s] + 1$
    return $T$
```

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 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 $S[m+1..n-1]$.*

Solution:

Each substring $S[i..j]$ with $i \leq m < j$ is 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 $S[m+1..n-1]$, such that the numbers of “1” in $S[i..m]$ and $S[m+1..j]$ sum up to k. Moreover, from each $S[i..m]$ that contains $p \leq k$ ones, we can build as many different sequences $S[i..j]$ with k ones as there are substrings $S[m+1..j]$ with $k-p$ ones. We obtain the following algorithm:

Algorithm 3

```
function SPANNING($m, k, S$)
    $T_1 \leftarrow$ SUFFIXTABLE($S[0..m]$)
    $T_2 \leftarrow$ PREFIXTABLE($S[m+1..n-1]$)
    return $\sum_{p=\max(0,k-(n-m-1))}^{\min(k,m)} (T_1[p] \cdot T_2[k-p])$
```

The complexity of this algorithm is $O(n)$.

*(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.*

Solution:

Whenever $n \geq 2$, we can distinguish between:

- Substrings with k ones located entirely in the first half of the bitstring, which we compute recursively;
- Substrings with k ones located entirely in the second half of the bitstring, which we also compute recursively;
- Substrings with k ones that span the two halves, which we can count using (c).

We obtain the following algorithm:

Algorithm 4 Clever substring counting

```
function COUNTSUBSTR($S, k, i = 0, j = n-1$)
    if $i = j$ then
        if $k = 1$ and $S[i] = 1$ then
            return 1
        else if $k = 0$ and $S[i] = 0$ then
            return 1
        else
            return 0
    else
        $m \leftarrow \lfloor (i+j)/2 \rfloor$
        return COUNTSUBSTR($S, k, i, m$) + COUNTSUBSTR($S, k, m+1, j$) + SPANNING($m, k, S$)
```

The complexity of this algorithm is given by a recursive expression of the form $A(n) = 2A(\frac{n}{2}) + O(n)$, which, as in the lecture, yields a total complexity of $O(n \log n)$.

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 5

(a)
```
$f()$
$i \leftarrow 0$
while $i \leq n$ do
    $f()$
    $i \leftarrow i + 1$
```

Solution:

This algorithm performs $1 + \sum_{i=0}^{n} 1 = 1 + (n+1) = n + 2 = O(n)$ calls to f.

Algorithm 6

(b)
```
$i \leftarrow 0$
while $i^2 \leq n$ do
    $f()$
    $f()$
    for $j \leftarrow 1, \ldots, n$ do
        $f()$
    $i \leftarrow i + 1$
```

Solution:

This algorithm performs $\sum_{i=0}^{\lfloor\sqrt{n}\rfloor}(2 + n) = (2 + n)(\lfloor\sqrt{n}\rfloor + 1) = O(n^{1.5})$ calls to f.

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.

Solution:

Algorithm 7

```
$F \leftarrow \texttt{int}[n + 1]$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2, \ldots, n$ do
    $F[i] \leftarrow F[i - 2] + F[i - 1]$
return $F[n]$
```

At the end of iteration i of this algorithm, we have $F[j] = f_j$ for all $0 \leq j \leq i$. Hence, at the end of the last iteration, $F[n]$ contains f_n. Each of the n iterations has complexity $O(1)$, yielding a total complexity in $O(n)$.

(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).*

Solution:

Consider the following algorithm, where we can just assume for now that K is 'large enough' so that no access outside of the valid index range of the array is performed.

Algorithm 8

```
$F \leftarrow \texttt{int}[K]$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
$i = 1$
while $F[i] \leq k$ do
    $i \leftarrow i + 1$
    $F[i] \leftarrow F[i-2] + F[i-1]$
return $F[i-1]$
```

After the ith iteration, we have $F[j] = f_j$ for all $0 \leq j \leq i$. The loop exists when the condition $F[i] = f_i > k$ is satisfied for the first time, and, in this case, $F[i-1] = f_i$ is the largest Fibonacci number smaller or equal to k. Using 2.2(b), we have $k \geq f_i \geq \frac{1}{3} \cdot 1.5^i$. We can rewrite $k \geq \frac{1}{3} \cdot 1.5^i$ as $i \leq \ln_{1.5}(3k) = \frac{\ln 3 + \ln k}{\ln 1.5} \leq 3(2 + \ln k) = O(\log k)$. Note that $\ln x$ denotes the natural logarithm; we do not need to specify the base of the logarithm within O-notation since different bases are equivalent up to constants (and get hidden in the O-notation). Therefore, the inner while loop can only execute $O(\log k)$ iterations. We can choose $K = 3(2 + \ln k)$.

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

Algorithm 9

```
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.*

Solution:

We have that $f_k = f_{k-1} + f_{k-2}$ for all $k \geq 2$. Using $f_{k-2} \leq f_{k-1}$ (for $k \geq 2$) we have:

$$\begin{aligned} f_k &= f_{k-1} + f_{k-2} \\ &\leq f_{k-1} + f_{k-1} \\ &\leq 2 \cdot f_{k-1}. \end{aligned}$$

The last inequality one can be rewritten as $f_{k-1} \geq \frac{1}{2} f_k$.

After any single iteration of the outer while loop, the variable k is at least halved. Hence, by straightforward induction, it must be 0 after at most $\lfloor \log_2 n \rfloor = O(\log n)$ steps.

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.

Solution:

Algorithm 10 $A_n(a)$

```
$x \leftarrow A_{n/2}(a)$
**return** $x \cdot x$
```

(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.

Solution:

Algorithm 11 Power(a, n)

```
**if** $n = 1$ **then**
    **return** a
**else**
    $x \leftarrow$ Power$(a, n/2)$
    **return** $x \cdot x$
```

(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.

Solution:

Let $T(n)$ be the number of elementary operations that the algorithm from part b) performs on input a, n. Then

$$T(n) \leq T(n/2) + 1 \leq T(n/4) + 2 \leq T(n/8) + 3 \leq \ldots \leq T(1) + \log_2 n \leq O(\log n)\,.$$

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

Solution:

- **Base Case.**
  Let $k = 0$. Then $n = 1$ and Power$(a, n) = a = a^1$.
- **Induction Hypothesis.**
  Assume that the property holds for some positive integer k. That is, Power$(a, 2^k) = a^{2^k}$.
- **Inductive Step.**
  We must show that the property holds for $k + 1$.

$$\text{Power}(a, 2^{k+1}) = \text{Power}(a, 2^k) \cdot \text{Power}(a, 2^k) \overset{\text{I.H.}}{=} a^{2^k} \cdot a^{2^k} = a^{2^{k+1}}.$$

By the principle of mathematical induction, this is true for any integer $k \geq 0$ and $n = 2^k$.

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

Solution:

Algorithm 12 Power(a, n)

```
if $n = 1$ then
    return a
else
    if $n$ is odd then
        $x \leftarrow$ Power$(a, (n-1)/2)$
        return $x \cdot x \cdot a$
    else
        $x \leftarrow$ Power$(a, n/2)$
        return $x \cdot x$
```