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

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

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 (Theorem 1.1 from the script). *Let N be an infinite subset of $\mathbb{N}$ and $f : N \to \mathbb{R}^+$ and $g : N \to \mathbb{R}^+$.*

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

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

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

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

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

Solution:

True by Theorem 1, since

$$\lim_{n\to\infty} \frac{\frac{1}{5}n^3}{10n^2} = \lim_{n\to\infty} \frac{1}{50}n = \infty.$$

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

Solution:

False, by Theorem 1, since

$$\lim_{n\to\infty} \frac{n^2+3n}{n^2\log(n)} = \lim_{n\to\infty} \frac{1}{\log(n)} + \lim_{n\to\infty} \frac{3}{n\log(n)} = 0 + 0 = 0.$$

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

Solution:

True by Theorem 1, since

$$\lim_{n\to\infty} \frac{5n^4+3n^2+n+8}{n^4} = \lim_{n\to\infty} 5 + \frac{3}{n^2} + \frac{1}{n^3} + \frac{8}{n^4} = 5.$$

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

Solution:

True by Theorem 1, since

$$\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty.$$

(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 $-1 \leq \sin(x) \leq 1$.*

Solution:

Using the hint we get that $1 \leq \sin(n) + 2 \leq 3$ and thus $n \leq (\sin(n) + 2)n \leq 3n$. The first inequality shows that $n \leq O((\sin(n) + 2)n)$ whereas the second one shows $(\sin(n) + 2)n \leq O(n)$. Together we get $(\sin(n) + 2)n = \Theta(n)$.

(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.*

Solution:

To show that $\sum_{i=1}^{n}\sum_{j=1}^{i} j \leq O(n^3)$ we can compute

$$\sum_{i=1}^{n}\sum_{j=1}^{i} j \leq \sum_{i=1}^{n}\sum_{j=1}^{i} n = \sum_{i=1}^{n} i \cdot n \leq \sum_{i=1}^{n} n^2 = n^3.$$

To show that $n^3 \leq O(\sum_{i=1}^{n}\sum_{j=1}^{i} j)$ we use exercise 1.3b, which states that

$$\sum_{j=1}^{i} j \geq \frac{1}{4} i^2$$

and

$$\sum_{i=1}^{n} i^2 \geq \frac{1}{8} n^3.$$

Combining these we get

$$\sum_{i=1}^{n}\sum_{j=1}^{i} j \geq \sum_{i=1}^{n} \frac{1}{4} i^2 \geq \frac{1}{4} \cdot \frac{1}{8} n^3,$$

which is equivalent to

$$n^3 \leq 32 \sum_{i=1}^{n}\sum_{j=1}^{i} j.$$

This shows that indeed $n^3 \leq O(\sum_{i=1}^{n}\sum_{j=1}^{i} j)$ and we get that $\sum_{i=1}^{n}\sum_{j=1}^{i} j = \Theta(n^3)$.

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

Solution:

As log is monotone and $n^4 + n^3 + n^2 \leq 3n^4$, we have

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

Now, $3 \leq n^3 + n^2 + n$ and $n^3 \leq n^3 + n^2 + n$, so we get

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

and

$$4\log(n) = \frac{4}{3}\log(n^3) \leq \frac{4}{3}\log(n^3 + n^2 + n).$$

So we can conclude that

$$\log(n^4 + n^3 + n^2) \leq \log(3) + 4\log(n) \leq \frac{7}{3}\log(n^3 + n^2 + n),$$

which means that $\log(n^4 + n^3 + n^2) \leq O(\log(n^3 + n^2 + n))$.

(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.*

Solution:

We first show $\sum_{i=1}^{n} \sqrt{i} \leq O(n\sqrt{n})$. For all $i \in \mathbb{N}$ with $1 \leq i \leq n$ we have $\sqrt{i} \leq \sqrt{n}$ and thus

$$\sum_{i=1}^{n} \sqrt{i} \leq \sum_{i=1}^{n} \sqrt{n} = n\sqrt{n},$$

which shows that $\sum_{i=1}^{n} \sqrt{i} \le O(n\sqrt{n})$.
Second, we show that $n\sqrt{n} \le O(\sum_{i=1}^{n} \sqrt{i})$. We have, similar to exercise 1.3,

$$\sum_{i=1}^{n} \sqrt{i} \ge \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \sqrt{i} \ge \frac{n}{2}\sqrt{\frac{n}{2}},$$

since the sum has $n - \lceil \frac{n}{2} \rceil + 1 \ge n - (\frac{n}{2} + 1) + 1 = \frac{n}{2}$ terms and every term is at least $\sqrt{\frac{n}{2}}$ because $i \ge \lceil \frac{n}{2} \rceil \ge \frac{n}{2}$. Thus,

$$\sum_{i=1}^{n} \sqrt{i} \ge \frac{1}{2\sqrt{2}} n\sqrt{n}$$

or equivalently

$$n\sqrt{n} \le 2\sqrt{2} \sum_{i=1}^{n} \sqrt{i},$$

which shows $n\sqrt{n} \le O(\sum_{i=1}^{n} \sqrt{i})$ and thus completes the proof.

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 $k \ge 0$, 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 is $O(n^3)$. 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 $0 \le i < \mathsf{length}(S)$. 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($S$)
    $T \leftarrow$ int$[n+1]$                          ▷ Initialize array
    $s \leftarrow 0$
    for $i \leftarrow 0, \dots, n-1$ do                 ▷ Enumerate all prefixes $S[0..i]$
        $s \leftarrow s + S[i]$                         ▷ $s$ saves the number of "1" in $S[0..i]$
        $T[s] \leftarrow T[s] + 1$                      ▷ $S[0..i]$ is a prefix with $s$ "1"
    return $T$
```

The for loop has n iterations, so the total runtime is $O(n)$. The correctness directly follows from the description of the algorithm (see comments above).

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 $m \in \{0, \dots, n-2\}$, 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 $S =$ "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 $\sum_{p=\max(0,k-(n-m-1))}^{\min(k,m+1)} (T_1[p] \cdot T_2[k-p])$
```

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

(d)* Using SPANNING, design an algorithm with a runtime[1] of at most $O(n \log n)$ that counts the number

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

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 $n \in \mathbb{N}$. Provide **both** the exact number of calls and a maximally simplified asymptotic bound in Θ notation.

Algorithm 5

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

Solution:

This algorithm performs $\sum_{i=0}^{n} 2 + \sum_{j=0}^{2n} 1 = 2(n+1) + (2n+1) = 4n+3 = \Theta(n)$ calls to f.

(b)

Algorithm 6

```
$i \leftarrow 1$
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.*

Solution:

This algorithm performs $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4} = \Theta(n^4)$ calls to f.

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 $n \geq 1$.

Solution:

- **Base Case.** We prove that the inequality holds for $n = 1$ and $n = 2$.
  For $n = 1$: $f_1 = 1 \geq 0.5 = \frac{1}{3} \cdot 1.5$, which is true.
  For $n = 2$: $f_2 = 1 \geq 0.75 = \frac{1}{3} \cdot 1.5^2$, which is true.

- **Induction Hypothesis.** We assume that it is true for $n = k$ and $n = k + 1$, i.e.,

$$f_k \geq \frac{1}{3} 1.5^k$$
$$f_{k+1} \geq \frac{1}{3} 1.5^{k+1}$$

- **Inductive Step.** We must show that the property holds for $n = k + 2$, $k \geq 1$. We have:

$$\begin{aligned} f_{k+2} &= f_{k+1} + f_k \\ &\geq \frac{1}{3}1.5^{k+1} + \frac{1}{3}1.5^k \\ &= \frac{1}{3}1.5^k \cdot (1.5 + 1) \\ &= \frac{1}{3}1.5^k \cdot 2.5 \\ &\geq \frac{1}{3}1.5^k \cdot 2.25 \\ &= \frac{1}{3}1.5^k \cdot 1.5^2 \\ &= \frac{1}{3}1.5^{k+2} \end{aligned}$$

  By the principle of mathematical induction, this is true for every integer $n \geq 1$.

Remark: In a similar way to the proof above, one can also show that $f_{n+1} \leq 1.75^n$ for $n \geq 0$.

(b) Write an $O(n)$ algorithm that computes the nth 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.

Solution:

Algorithm 7

```
$F \leftarrow$ int$[n+1]$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2, \dots, n$ 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 $O(n)$.

(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).*

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, $F[i-1] = f_{i-1}$ is the largest Fibonacci number smaller or equal to k. Using part (a), 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 $i \leq \log_{1.5}(3k) = \frac{\ln 3 + \ln k}{\ln 1.5} \leq 3(2 + \ln k) = O(\log k)$. 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 $K = 3(2 + \ln k)$. Since every iteration of the while-loop has complexity $O(1)$, we get an overall complexity of $O(\log k)$.

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 9 $A_n(a)$

```
$x \leftarrow A_{n/2}(a)$
**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 10 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

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

(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 11 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 $x \cdot x$
```

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