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

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

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

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

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

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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, 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 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 $S =$ "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 $S =$ "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 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 $S[m+1..n-1]$.*

(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.)

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.

(a)

Algorithm 1

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

(b)

Algorithm 2

```
$i \leftarrow 1$
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 $n \geq 1$.

[1] For 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 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.

(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 $n = 2k + 1$.*

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.

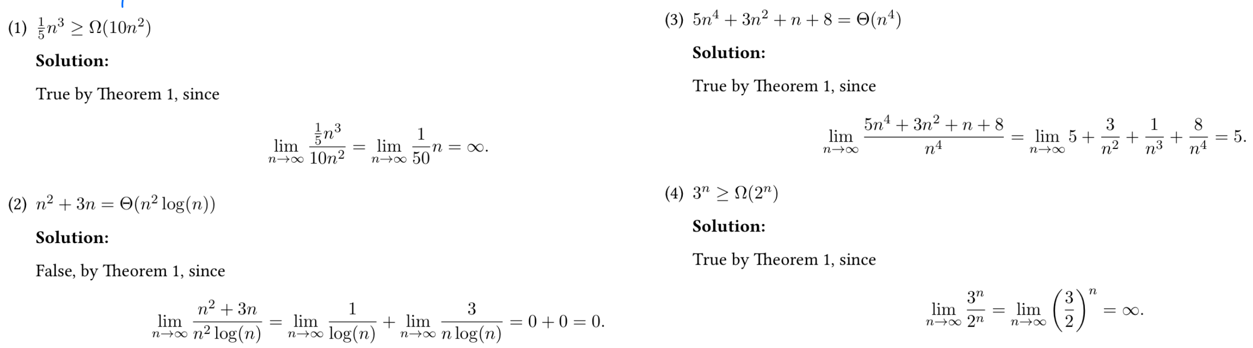
(1) $\frac{1}{5}n^3 \geq \Omega(10n^2)$

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

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

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

All of these subtesk just rely on Theorem 1.1.



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

As the first hint suggests, we should use the set definitions of O-notation.

Official solutions:

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

My solution using definition of $\Theta(f)$

$$\Theta(f) = \{ g : \mathbb{N} \to \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}\ \exists c_1, c_2 > 0 : c_1 g(n) \leq f(n) \leq c_2 g(n)\ \forall n \geq n_0 \}$$

Since $-1 \leq \sin(n) \leq 1$ we have $1 \leq \sin(n) + 2 \leq 3$ and thus $n \leq (\sin(n) + 2)n \leq 3n$.

Since $(\sin(n) + 2)n \leq 3n$ we also have $\frac{1}{3}(\sin(n) + 2)n \leq n$.

Thus $\frac{1}{3}(\sin(n) + 2)n \leq n \leq (\sin(n) + 2)n \quad \forall n \geq 1$.

We find $c_1 = \frac{1}{3}$, $c_2 = 1$ and $n_0 = 1$ which fulfil the definition of $\Theta(n)$, thus $(\sin(n) + 2)n = \Theta(n)$.

(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.*

In recent exercise sheets we have often tried to find lower / upper bounds for sums to determine their asymptotic complexity. The same applies for double, triple ... sums:

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

Thus $\sum_{i=1}^{n} \sum_{j=1}^{i} j \leqslant O(n^3)$.

To show the other direction we use the hint, i.e. 1.3 b

(b) Show that for all $n \in \mathbb{N}_0$, we have $\sum_{i=1}^{n} i^k \geq \frac{1}{2^{k+1}} \cdot n^{k+1}$.

$$\xRightarrow{1.3(b)} \quad \sum_{j=1}^{i} j \geqslant \frac{1}{4} i^2 \quad \text{and} \quad \sum_{i=1}^{n} i^2 \geqslant \frac{1}{8} n^3$$

Thus we have

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

Hence $n^3 \leqslant 32 \sum_{i=1}^{n} \sum_{j=1}^{i} j$, which implies $n^3 \leqslant O(\sum_{i=1}^{n} \sum_{j=1}^{i} j)$.

Both directions show $\sum_{i=1}^{n} \sum_{j=1}^{i} j = \Theta(n^3)$

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

Here we want to use the properties of the logarithm function. We say log is a monotone increasing function, thus for $x < y$ we have $\log(x) < \log(y)$.

Side note: This isn't true in general. It is only true for $\log_a$ where $a > 1$, but it doesn't matter since $\log_a n < 0$ for $n \geq 1$ and in computer science we restrict ourselves to functions $f: \mathbb{N} \to \mathbb{R}^+$ when using O-notation. Sorry for this side note.

Since log is monotone increasing and $n^4 + n^3 + n^2 \leq n^4 + n^4 + n^4 = 3n^4$ 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 $n^3 \leq n^3 + n^2 + n$ thus

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

and hence

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

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

we used $7/3$, <u>why?</u>

Notice: $\log(3) \leq \log(n^3+n^2+n)$ and we showed $4\log(n) \leq \frac{4}{3}\log(n^3+n^2+n)$. Now

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

<u>But how do I get there myself?</u>

Focus on what it is you have to prove: we want to prove $f \leq O(g)$ for $f(n) = \log(n^4+n^3+n^2)$ and $g(n) = \log(n^3+n^2+n)$. Thus we try to find a constant c sum that $f(n) \leq c\, g(n)$.

(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.*

$$\sum_{i=1}^{n} \sqrt{i} \leq \sum_{i=1}^{n} \sqrt{n} = n\sqrt{n} \implies \sum_{i=1}^{n} \sqrt{i} \leq O(n\sqrt{n})$$

For the other direction we use 1.3

((b) Show that for all $n \in \mathbb{N}_0$, we have $\sum_{i=1}^{n} i^k \geq \frac{1}{2^{k+1}} \cdot n^{k+1}$.)

Careful now, for 1.3 we said $k \in \mathbb{N}$, so we can't just say $\sum_{i=1}^{n} \sqrt{i} = \sum_{i=1}^{n} i^{1/2} \geq \frac{1}{2^{3/2}} \cdot n^{3/2}$. We have to prove it, using the same idea of 1.3 (b):

Notice that

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

where we used that the sum from $i = \lceil \frac{n}{2} \rceil$ to n has $n - \lceil \frac{n}{2} \rceil + 1 \geq n - (\frac{n}{2} + 1) + 1 = \frac{n}{2}$ terms and every term $\sqrt{i} \geq \sqrt{\frac{n}{2}}$ for $i = \{\lceil \frac{n}{2} \rceil, \dots, n\}$.

Thus

$$\frac{1}{2\sqrt{2}} n\sqrt{n} \leq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \sqrt{i} \iff n\sqrt{n} \leq 2\sqrt{2} \sum_{i=\lceil \frac{n}{2} \rceil}^{n} \sqrt{i}$$

which shows $n\sqrt{n} \leq O(\sum_{i=1}^{n} \sqrt{i})$.

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
    $f()$
    $j \leftarrow j + 1$
```

Note that you are required to state both the exact number of calls to f and a maximally simplified bound in Θ notation.

For these type of exercises, there is a great little guide on my website, kindly provided to me by Maximilian Schlegel and two other co-authors. Find it under the heading "Week 3", link "Week 4".

The exact number of calls is:

$$\sum_{i=0}^{n} 2 + \sum_{j=0}^{2n} 1 = 2(n+1) + 2n+1 = 4n+3 = \Theta(n)$$

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.~~* 1.2

Notice how j is set to one every iteration of our outermost while loop and takes on values from 1 to i^3. $f()$ is only called once, thus:

$$\sum_{i=0}^{n} i^3 \overset{1.2}{=} \frac{n^2(n+1)^2}{4} = \Theta(n^4) \quad \text{calls to } f.$$

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 $S =$ "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.

As with a lot of naive algorithms, we just try out all possible substrings and count the 1's.
What are all possible substrings?

b_0	b_1	b_2	b_3	b_4	b_5	string

substrings starting from b_0: b_0, $b_0 b_1$, $\dots$, $b_0 \dots b_5$

substrings starting from b_1: b_1, b_{12}, $\dots$, $b_1 \dots b_5$

$\vdots$

this showcases how we process "move through" the string. Lastly when we have some substring $b_i \dots b_j$ we just loop through it, counting 1's.

Offical solutions:

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).

As you can see, the correctness and runtime argument is very short. Personally, I like to write a little bit more, just to be sure.

For example, what is the number of substrings? Why do we check all of them in the code?

(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 $S =$ "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.

Here it is crucial to carefully read the description as it tells you exactly what your algorithm should do!

For the algorithm, we use one important fact: If $S=$ "0110" $(b_0 b_1 b_2 b_3)$ and we look at the prefix "0", then the next prefix "01" is precisely the first one, but with b_1 appended. Thus we can reuse information instead of looping through it everytime.

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).

(c) Let S be a n-bit bitstring. Consider an integer $m \in \{0, \ldots, n \overset{n-2}{\cancel{-1}}\}$, 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 $S =$ "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 $\sum_{p=\max(0,k-(n-m-1))}^{\min(k,\cancel{m})\ m+1} (T_1[p] \cdot T_2[k-p])$
```

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

I'm sure this might confuse some students. My recommendation: work through an exampl. Step by step.

Two comments: First, the bounds of the sum $\min(k,m)$, $p = \max(0, k-(n-m-1))$ can be determined by looking at the size of $T_{1,2}$. We always want to be in their respective range! T_1 is of size $m+2$, thus the index range is $0 \leq i \leq m+1$ which explains the upper bound for the sum.

T_2 is of size $n-m$, thus the index range is $0 \leq i \leq n-m-1$.
If the difference $k-(n-m-1) \geq 1$ then $T_2(k-p)$ if $p=0$ would fail since $k \geq n-m-1$.
This explains the lower bound of the sum.

(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.)

Hint: *Use the recursive idea from the lecture.*

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($m, k, S$)
```

Again: Simulate this algorithm using pen and paper! You want to know what is going on!

This strategy of making smaller subtask from a bigger one and putting the results back together is often called "divide and conquer" (e.g. Merge sort).
I highly advise you to familiarize you with this topic! It takes time to do so!

Exercise 3.4 *Fibonacci numbers.*

I have nothing to add here, please refer to the official solutions.

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.

I have nothing to add to (a) and (b)

(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

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

This is what we call "telescoping":

$$T(n) \le T(\tfrac{1}{2}n) + 1 \le T(\tfrac{1}{2}(\tfrac{1}{2}n)) + 1 + 1$$

$$\le T(\tfrac{1}{2}(\tfrac{1}{2}(\tfrac{1}{2}n))) + 1 + 1 + 1$$

...

I have nothing to add to (d) and (e)