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.

The solutions are intended to help you understand how to solve the exercises and are thus more detailed than what would be expected at the exam. All parts that contain explanation that you would not need to include in an exam are in grey.

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

Solution:

False by Theorem 1, since

$$\lim_{n\to\infty} \frac{3n^5 + 5n^3}{4n^4} = \lim_{n\to\infty} \frac{3}{4}n + \lim_{n\to\infty} \frac{5}{4n} = \lim_{n\to\infty} \frac{3}{4}n + 0 = \infty.$$

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

Solution:

False, by Theorem 1, since

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

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

Solution:

True by Theorem 1, since

$$\lim_{n\to\infty} \frac{\frac{1}{6}n^6 + 10n^4 + 100n^3}{6n^6} = \lim_{n\to\infty} \frac{1}{36} + \frac{10}{6n^2} + \frac{100}{6n^3} = \frac{1}{36}.$$

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

Solution:

True by Theorem 1, since

$$\lim_{n\to\infty} \frac{3^n}{n^{3/\ln n}e^n} = \lim_{n\to\infty} \frac{e^{\ln(3)n}}{e^{3\ln(n)/\ln(n)}e^n} = \lim_{n\to\infty} e^{\ln(3)n-3-n} = \lim_{n\to\infty} e^{(\ln(3)-1)n-3} = \infty.$$

Since $\ln(3) - 1 > 0$ and so $\lim_{n\to\infty}(\ln(3) - 1)n - 3 = \infty$.

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

Solution:

We have $n^2 + n + 1 \leq n^2 + 2n + 1 = (n + 1)^2$ so

$$\sqrt{n^2 + n + 1} \leq n + 1 \leq O(n).$$

In the other direction, $n^2 + n + 1 \geq n^2$ so

$$\sqrt{n^2 + n + 1} \geq n = \Omega(n).$$

Together we get $\sqrt{n^2 + n + 1} = \Theta(n)$.

(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.*

Solution:

To compute the sum, we first simplify each term from $\log(i^i)$ to $i \log(i)$. Then, we lower bound the sum by only looking at the last $n/2$ terms, so

$$\sum_{i=1}^{n} \log\left(i^i\right) = \sum_{i=1}^{n} i \log(i) \geq \sum_{i=\lceil n/2 \rceil}^{n} i \log(i)$$

Since we only look at terms indexed from $i = \lceil n/2 \rceil$ to n, we have that $i \geq n/2$. Plugging this bound into each term we get

$$\sum_{i=\lceil n/2 \rceil}^{n} i \log(i) \geq \sum_{i=\lceil n/2 \rceil}^{n} \frac{n}{2} \log\left(\frac{n}{2}\right) = \left(n - \left\lceil \frac{n}{2} \right\rceil + 1\right) \frac{n}{2} \log\left(\frac{n}{2}\right) \geq \frac{n}{2}\frac{n}{2} \log\left(\frac{n}{2}\right) = \frac{n^2}{4} \log\left(\frac{n}{2}\right)$$

where the second inequality comes from $\lceil n/2 \rceil - 1 \leq n/2$. Now, we just use properties of $\log$ to further simplify

$$\frac{n^2}{4} \log\left(\frac{n}{2}\right) = \frac{n^2}{4} \left(\log(n) - \log(2)\right)$$

From here we can use Theorem 1 to argue that the right hand side is $\Omega(n^2 \log(n))$. Computing the limit we have

$$\lim_{n \to \infty} \frac{\frac{n^2}{4} \left(\log(n) - \log(2)\right)}{n^2 \log(n)} = \lim_{n \to \infty} \frac{1}{4} - \lim_{n \to \infty} \frac{\log(2)}{4 \log(n)} = \frac{1}{4} - 0 = \frac{1}{4}.$$

Using this fact alongside the earlier inequalities, we get

$$\sum_{i=1}^{n} \log\left(i^i\right) \geq \frac{n^2}{4} \left(\log(n) - \log(2)\right) \geq \Omega(n^2 \log(n)).$$

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

Solution:

First we factor $n^2 + n$ as $n(n + 1)$ getting

$$\log(n^2 + n) = \log(n(n + 1)) = \log(n) + \log(n + 1).$$

Then in one direction,

$$\log(n) + \log(n + 1) \geq \log(n + 1) \geq \Omega(\log(n + 1)).$$

As $\log$ is monotone, $\log(n) \leq \log(n + 1)$ so in the other direction,

$$\log(n) + \log(n + 1) \leq 2 \log(n + 1) \leq O(\log(n + 1)).$$

Together we get $\log(n^2 + n) = \Theta(\log(n + 1))$.

(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.*

Solution:

We first show $\sum_{i=1}^{n} \frac{1}{\sqrt{i}} \geq \Omega(\sqrt{n})$. For all $1 \leq i \leq n$ we have $\frac{1}{\sqrt{i}} \geq \frac{1}{\sqrt{n}}$ and thus

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

which shows that $\sum_{i=1}^{n} \frac{1}{\sqrt{i}} \geq \Omega(\sqrt{n})$. Second, we show that $\sum_{i=1}^{n} \frac{1}{\sqrt{i}} \leq O(\sqrt{n})$. We have that $\sqrt{i} \geq \sqrt{i-1}$ so $2\sqrt{i} \geq \sqrt{i} + \sqrt{i-1}$ and therefore we get the bound

$$\frac{1}{\sqrt{i}} \leq \frac{2}{\sqrt{i} + \sqrt{i-1}}.$$

Multiplying the top and bottom by $\sqrt{i} - \sqrt{i-1}$ we get

$$\frac{2(\sqrt{i} - \sqrt{i-1})}{(\sqrt{i} + \sqrt{i-1})(\sqrt{i} - \sqrt{i-1})} = \frac{2(\sqrt{i} - \sqrt{i-1})}{i + \sqrt{i}\sqrt{i-1} - \sqrt{i-1}\sqrt{i} - (i-1)} = \frac{2(\sqrt{i} - \sqrt{i-1})}{1}.$$

Combining this with the previous inequality we get

$$\frac{1}{\sqrt{i}} \leq 2(\sqrt{i} - \sqrt{i-1})$$

Using this inequality on each term we have

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

Notice that this sum telescopes, meaning that for each $k \in \mathbb{N}$ such that $1 \leq k < n$, a $\sqrt{k}$ comes from the k entry of the sum and a $-\sqrt{k}$ comes from the $k+1$ entry of the sum. Therefore the sum simply equals the terms which do not get cancelled out. This will be $\sqrt{n}$ from the nth entry and a $-\sqrt{0}$ from the 1st entry. So

$$2 \sum_{i=1}^{n} \sqrt{i} - \sqrt{i-1} = 2(\sqrt{n} - \sqrt{0}) = 2\sqrt{n} \leq O(\sqrt{n}),$$

getting us $\sum_{i=1}^{n} \frac{1}{\sqrt{i}} \leq O(\sqrt{n})$. Combining everything we have $\sum_{i=1}^{n} \frac{1}{\sqrt{i}} = \Theta(\sqrt{n})$.

Guidelines for correction:

The exercise consists of 7 items (excluding b.4). Award 1/2 point for the first correctly solved item. Then award 1/2 point for each two additional correctly solved items (so 3 gives 1 point, 5 gives 1.5 points, and 7 gives 2 points).

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).

Solution:

We can for example use the following algorithm:

The following algorithm works by creating every possible substring and checking if it fulfills the required condition. Therefore, it is considered to be "naive".

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
```

Runtime: The nested for-loops have three levels and each level has at most n iterations, leading to $O(n^3)$ iterations in total. Each iteration runs in $O(1)$ time. Thus the total running time is $O(n^3)$.

Correctness: Follows directly 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".

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).

Solution:

Algorithm 2

```
function PREFIXTABLE($S$)
    $T[0..n] \leftarrow$ a new array of size $(n+1)$                ▷ 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 idea of the algorithm is iterating over all prefixes and counting the 1's. For each prefix we increase the table's count for the current number of 1's. As this can only increase, we can create the table in $O(n)$.

Runtime: The for loop has n iterations and each iteration runs in $O(1)$ time, so the total runtime is $O(n)$.

Correctness: 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) Consider an integer $m \in \{0, 1, \dots, n-2\}$. 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 $i \le m < j$.

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 $i \le m < j$ 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 $i \le m < j$ 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 $p \le k$ 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 $\sum_{p=\max(0,k-(n-m-1))}^{\min(k,m+1)} (T_1[p] \cdot T_2[k-p])$
```

Runtime: $O(n)$.

(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.*

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($m, k, S[i..j]$)
```

Runtime: By subpart (c), the spanning subroutine needs time $O(n)$. Thus, if we denote the complexity of the algorithm by $A(n)$, then it is given by the recursive expression $A(n) \leq 2A(\frac{n}{2}) + O(n)$ (with the base case being $A(2) \leq O(1)$). Simplifying this, we get

$$\begin{aligned} A(n) &\leq 2A\left(\frac{n}{2}\right) + O(n) \leq 2 \cdot \left(2A\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)\right) + O(n) \\ &\leq 4A\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right) + O(n) \leq 4A\left(\frac{n}{4}\right) + O(n) + O(n) \\ &\leq \ldots \leq O\left(n \log(n)\right). \end{aligned}$$

The reason this is $O(n \log(n))$ is that for halving n, we get a factor of $O(n)$. Thus, to get n down to a constant, we need $\log(n)$ steps and thus overall $A(n)$ is bounded by $O\left(n \log(n)\right)$.

Note that we cannot absorb the factor "2" in $2O(n/2)$ in the O-notation and then conclude that the runtime is $O(n) + O(n/2) + O(n/4) + \ldots$, which would be $O(n)$. The reason is that when we write the full expression (assuming for simplicity that n is a power of 2), we get $A(n) \leq \sum_{i=0}^{\log(n)} 2^i \cdot O(n/2^i)$ and thus in general the factor 2^i is not a constant factor.

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

To prove the above formally, we first show the statement for $n = 2^k$ for $k \in \mathbb{N}$. Let C by a constant such that $A(2^k) \leq 2A(2^{k-1}) + C \cdot 2^k$ for all $k \in \mathbb{N}$ and $A(2) \leq C$. Then, we can prove the identity $A(n) = A(2^k) \leq C \cdot 2^k \cdot k = C \cdot n \log(n)$ for any $n = 2^k$ by induction over $k \in \mathbb{N}$:

Base case. Let $k = 1$, then $A(2^1) \leq C \leq C \cdot 2 \log(2)$ by assumption.

Induction hypothesis. Assume $A(2^\ell) \leq C \cdot 2^\ell \cdot \ell$ holds for some $\ell \in \mathbb{N}$.

Induction step. We now want to show $A(2^{\ell+1}) \leq C \cdot 2^{\ell+1} \cdot (\ell + 1)$. Using the recursive formula from above and then the induction hypothesis we get

$$A(2^{\ell+1}) \leq 2A(2^\ell) + C \cdot 2^{\ell+1} \leq 2 \cdot (C \cdot 2^\ell \cdot \ell) + C \cdot 2^{\ell+1} = C \cdot 2^{\ell+1}(\ell + 1).$$

By the principle of mathematical induction, we get $A(n) \leq C \cdot n \log(n)$ for all n of the form $n = 2^k$ for $k \in \mathbb{N}$. To prove the general case for any $n \geq 2$, we use that $A(n) \leq A(2^{\lceil \log(n) \rceil})$ to get

$$A(n) \leq A(2^{\lceil \log(n) \rceil}) \leq C \cdot 2^{\lceil \log(n) \rceil} \cdot \lceil \log(n) \rceil \leq 8C \cdot n \log(n),$$

where the last step used that $\lceil \log(n) \rceil \leq 2 \log(n)$ and $2^{\lceil \log(n) \rceil} \leq 2^{\log(n)+1} = 2n$. Thus, we get $A(n) \leq 8C \cdot n \log(n) \leq O(n \log(n))$ for every $n \in \mathbb{N}_{\geq 2}$ as wanted.

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 5

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

Solution:

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

Algorithm 6

(b)
```
$i \leftarrow 1$
while $i \leq 2n$ do
    $j \leftarrow 1$
    while $j \leq i^3$ do
        $k \leftarrow n$
        while $k \geq 1$ do
            $f()$
            $k \leftarrow k - 1$
        $j \leftarrow j + 1$
    $i \leftarrow i + 1$
```

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

Solution:

This algorithm performs $\sum_{i=1}^{2n} \sum_{j=1}^{i^3} \sum_{k=1}^{n} 1 = \sum_{i=1}^{2n} \sum_{j=1}^{i^3} n = n \sum_{i=1}^{2n} \sum_{j=1}^{i^3} 1 = n \sum_{i=1}^{2n} i^3 = \Theta(n \cdot (2n)^4) = \Theta(n \cdot 2^4 \cdot n^4) = \Theta(n^5)$ calls to f.

For the fourth equality, we have used the fact that there are constants $C_1, C_2 > 0$ such that $C_1 \cdot m^4 \leq \sum_{i=1}^{m} i^3 \leq C_2 \cdot m^4$ for any $m \in \mathbb{N}$, plugging in $m = 2n$. This was shown in Exercise 1.2.

Guidelines for correction:

The exercise consists of four items, two per part: the exact and the simplified Θ expression. Award 1 point if all for items are correct. Award 1/2 point if at least two items are correct. Do not subtract points for computational mistakes if the final simplified expression is correct.

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.

Solution:

Base Case. We prove that the inequality holds for $n = 1$ and $n = 2$.
For $n = 1$: $f_n = 1$ and $\frac{1}{3} \cdot 1.5^n = 0.5$, so $f_n \geq \frac{1}{3} \cdot 1.5^n$.
For $n = 2$: $f_n = 1$ and $\frac{1}{3} \cdot 1.5^n = 0.75$, so $f_n \geq \frac{1}{3} \cdot 1.5^n$.

As the recursive definition of the Fibonacci Numbers relies on two previous instances, they require two base cases and two induction hypothesis.

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:

$$\begin{aligned}
f_{k+2} &\overset{\text{by def.}}{=} f_{k+1} + f_k \\
&\overset{\text{I.H.}}{\geq} \frac{1}{3}1.5^{k+1} + \frac{1}{3}1.5^k \\
&= \frac{1}{3}1.5^k \cdot (1.5 + 1) \\
&= \frac{1}{3}1.5^k \cdot 2.5 \\
&\geq \frac{1}{3}1.5^k \cdot 2.25 \\
&= \frac{1}{3}1.5^k \cdot 1.5^2 \\
&= \frac{1}{3}1.5^{k+2}
\end{aligned}$$

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) 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.

Solution:

Algorithm 7

```
$F[0..n]$ ← an array of $(n+1)$ integers
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2, \ldots, n$ do
    $F[i] \leftarrow F[i-2] + F[i-1]$
return $F[n]$
```

This algorithm is a simple iterative implementation the Fibonacci Numbers, defining the non-recursive cases and then iterating over the recursive ones.

Runtime: Each of the n iterations has complexity $O(1)$, yielding a total complexity of $O(n)$.

Correctness: 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.

(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).*

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. We will decide the value of K later.

Algorithm 8

```
$F[0..K] \leftarrow$ an array of $(K+1)$ integers
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
$i = 1$
while $F[i] \leq k$ do
    $i \leftarrow i + 1$
    $F[i] \leftarrow F[i-2] + F[i-1]$
return $F[i-1]$
```

Runtime: After the ***i*th 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

$$i \leq \log_{1.5}(3k) = \frac{\log 3 + \log k}{\log 1.5} \leq 3(2 + \log k) \leq O(\log k).$$

Although we use base 2 for logarithms, it's not necessary to specify the base of logarithms within O-notation in this case, since different bases are equivalent up to constants, which are irrelevant for the O-Notation.

Therefore, the while loop can only execute $O(\log k)$ iterations. Also, we can choose $K = \lceil 3(2 + \log k) \rceil$ to always create an array of sufficient size. Since every iteration of the while-loop has complexity $O(1)$, we get an overall complexity of $O(\log k)$.

The algorithm is the same as the one in (b), except that we check if the Fibonacci number we just compute is at most k after each iteration. We know from (b) that it takes $O(n)$ time to compute f_n. Here, since we only need to compute f_n for $n \leq O(\log k)$, the runtime is $O(\log k)$.

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.

(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.)

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. (You don't need to argue correctness or runtime.)

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 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 integer multiplications that the algorithm from part (b) performs on input a, n. Then

$$\begin{aligned} T(n) &\leq T(n/2) + 1 \\ &\leq T(n/4) + 2 \\ &\leq T(n/8) + 3 \\ &\leq \dots \\ &\leq T(1) + \log n \\ &\leq O(\log n)\,.^2 \end{aligned}$$

The $\log n$ can be deduced by the fact that n halves in every step and recursion stops when a non-recursive definition is reached.

(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.

Solution:

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

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. 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 $n = 2k + 1$.*

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 $x \cdot x$
```