Algorithms and Data Structures

Exercise Session 4

Prove or disprove: $\Omega(n^2) \cap \mathcal{O}(n^3) = \Theta(n^2) \cup \Theta(n^3)$

Prove or disprove: $\Omega(n^2) \cap \mathcal{O}(n^3) = \Theta(n^2) \cup \Theta(n^3)$

Solution: We disprove with the following counterexample. Let $f = n^2 \log n$. Then $f \in \Omega(n^2)$, $f \in \mathcal{O}(n^3)$, but $f \notin \Theta(n^2)$, and $f \notin \Theta(n^3)$.

Prove or disprove: Let $f, g \in \Theta(h)$, then $|f - g| \in \mathcal{O}(1)$.

Prove or disprove: Let $f, g \in \Theta(h)$, then $|f - g| \in \mathcal{O}(1)$.

Solution: We disprove with the following counterexample: $f(n) = n^2 + n$, $g(n) = n^2$, $h(n) = n^2$.

Is there a function $f : \mathbb{N} \to \mathbb{R}^+$ that is neither in $\mathcal{O}(n^2)$ nor in $\Omega(n^2)$? If no, prove, if yes, give an example.

Is there a function $f : \mathbb{N} \to \mathbb{R}^+$ that is neither in $\mathcal{O}(n^2)$ nor in $\Omega(n^2)$? If no, prove, if yes, give an example.

$$f(n) = \begin{cases} n & \text{if } n \text{ is even,} \\ n^3 & \text{if } n \text{ is odd.} \end{cases}$$

EXERCISE SHEET 3

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

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

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

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

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

(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.*

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

(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.*

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 1

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

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

Algorithm 2

(b)

```
i ← 1
while i ≤ 2n do
    j ← 1
    while j ≤ i^3 do
        k ← n
        while k ≥ 1 do
            f()
            k ← k − 1
        j ← j + 1
    i ← i + 1
```

Hint: *See Exercise 1.2. You are allowed to use any statement from that exercise without 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 $k \geq 0$, 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 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 3.4 *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.

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 $F[n]$

(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).*

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 $F[i-1]$

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.

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.

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.

(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.

(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 ← Power(a, (n − 1)/2)
        return x · x · a
    else
        x ← Power(a, n/2)
        return x · x
```

EXTRA TASKS

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

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

Algorithm 1

(a)

```
i ← 0
while i ≤ n do
    f()
    f()
    i ← i + 1
j ← 0
while j ≤ 2n do
    f()
    j ← j + 1
```

Algorithm 2

(b)

```
i ← 1
while i ≤ n do
    j ← 1
    while j ≤ i^3 do
        f()
        j ← j + 1
    i ← i + 1
```

Peer Grading

Exercise 3.1b