Darius Dolha dolha@uni-bremen.de
Mustafa Kemal Kücük mustafak@uni-bremen.com
Vasu Yaduvanshi vasu@uni-bremen.de

Übungsgruppe 51

1 Übung 3.1

1.1 Übung 3.1.(a)

We are going to use the Master Theorem to find a closed form in Θ-notation for the recursion ecuation.

Let $a \geq 1$, $b \geq 1$ constants, $f : \mathbb{N} \to \mathbb{N}$ and $T(n) = 8 \cdot T(\frac{n}{2}) + 2^n$. We know that $a = 8$, $b = 2$ and $f(n) = 2^n$.

If $f(n) \in \Omega(n^{\log_b(a+\epsilon)})$ for an $\epsilon > 0$ (*) and $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ for a constant $c < 1$ and n large enough(**) then:

$$T(n) \in \Theta(f(n))$$

.

Fist we are going to prove (*). Let ϵ=8. Now we have to prove that $2^n \in \Omega(n^{\log_4(8+8)})$. $2^n \in \Omega(n^3)$.

$\exists c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}\ \forall n \geq n_0$, $2^n \geq cn^3$ for all $n \geq n_0$.
Let $c = 1$ and $n_0 = 10$.
The inequality $2^n \geq n^3$ will hold for all $n \geq 10$ if the derivative of $f(n) = 2^n - n^3$ is positive for $n \geq 10$

$$f'(n) = \frac{d}{dn}\left(2^n - n^3\right) = 2^n \ln(2) - 3n^2.$$

For $n \geq 10$, $2^n \ln(2) \geq 3n^2$: 2^n grows exponentially, while $3n^2$ grows polynomially. Thus, for large enough n, $2^n \ln(2) > 3n^2$.
Since $f'(n) > 0$ for $n \geq 10$, $f(n) = 2^n - n^3$ is increasing for $n \geq 10$.
Then we can conclude that $2^n \in \Omega(n^3)$

The next step involves proving (**). $8 \cdot f(\frac{n}{2}) \leq c \cdot f(n)$ for a constant $c < 1$ and n large enough.
$8 \cdot 2^{n/2} \leq c \cdot 2^n$. We need to prove that exists $c < 1$ for n large enough such as $c \geq \frac{8}{2^{n/2}}$.
As $n \to \infty$, $2^{n/2}$ grows exponentially, and thus $\frac{8}{2^{n/2}}$ approaches 0. Therefore, for sufficiently large n, we can make c as small as we want, specifically, for any constant $c < 1$, there will exist an n large enough such that the inequality holds.
Since we have proved (*) and (**) we can conclude that:

$$T(n) \in \Theta(2^n)$$

.

1.2 Übung 3.1.(b)

Let $a \geq 1$, $b \geq 1$ constants, $f : \mathbb{N} \to \mathbb{N}$ and $T(n) = 64 \cdot T(\frac{n}{4}) + (n^2 + 1)(n + 7)$. We know that $a = 64$, $b = 4$ and $f(n) = n^3 + 7n^2 + n + 7$.
If $f(n) \in \Theta(n^{\log_b a})$ then:

$$T(n) \in \Theta(n^{\log_b a} \log n)$$

.

We need to prove that $n^3 + 7n^2 + n + 7 \in \Theta(n^{\log_4 64})$. $n^3 + 7n^2 + n + 7 \in \Theta(n^3)$. It is also known that $\Theta(g) = O(g) \cap \Omega(g)$, with $g(n) = 2^n$.

To show that $n^3 + 7n^2 + n + 7 \in O(n^3)$, we are going to prove the following.
$\exists c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}\ \forall n \geq n_0$, $n^3 + 7n^2 + n + 7 \leq cn^3$ for all $n \geq n_0$.
Let $c = 2$ and $n_0 = 10$. We need to prove that $7n^2 + n + 7 \leq n^3$ for $n \geq 10$, which is true since n^3 increases faster than $7n^2 + n + 7$. This can be shown using derivates or by induction.

To show that $n^3 + 7n^2 + n + 7 \in \Omega(n^3)$, we are going to prove the following.
$\exists c \in \mathbb{R}_{>0}, \exists n_0 \in \mathbb{N}\ \forall n \geq n_0$, $n^3 + 7n^2 + n + 7 \geq cn^3$ for all $n \geq n_0$.

Let $c = 1$. We need to prove that $7n^2 + n + 7 \geq 0$ for $n \geq n_0$. This is true for $n_0 = 1$ for example.

Since $n^3 + 7n^2 + n + 7 \in O(n^3)$ and $n^3 + 7n^2 + n + 7 \in \Omega(n^3)$ we can conclude that.

$$T(n) \in \Theta(n^3 \log n)$$

.

2 Übung 3.2

We are initially going to use the master theorem in order to find a closed form in Θ-notation for the recursion ecuation.
Let $a \geq 1$, $b \geq 1$ constants, $f : \mathbb{N} \to \mathbb{N}$ and $T(n) = 2 \cdot T(\frac{n}{2}) + \sqrt{n}$. We know that $a = 2$, $b = 2$ and $f(n) = \sqrt{n}$.
If $f(n) \in O(n^{\log_b(a+\epsilon)})$ for an $\epsilon > 0$ then:

$$T(n) \in \Theta(n^{\log_b a})$$

This will be true but we need to prove by induction. We need to prove that:

$$T(n) \in \Theta(n)$$

2.1 First $T(n) \in O(n)$

Induction of n. Induction hypotesses:
$T(m) \leq cm$ for all $m < n$
Induction step: $T(n) = 2T(\frac{n}{2}) + \sqrt{n} \leq 2c\frac{n}{2} + \sqrt{n} = cn + \sqrt{n}$.
We observe that $\sqrt{n}$ grows slower than n, so for large enough n, we can make the $\sqrt{n}$ term negligible compared to n. Specifically, we can find a constant c' such that:

$$T(n) \leq cn + \sqrt{n} \leq c'n \quad \text{for large enough } n.$$

For example, if we take $c' = c + 1$, we ensure that $\sqrt{n}$ becomes negligible for large n.
Base case: $n = 2$ we assume $T(1) = c_1$, $T(2) = 2c_1 + \sqrt{2} \leq 2c$. With c being a constant large enough.

2.2 Second $T(n) \in \Omega(n)$

Induction hypotheses: $T(m) > m$, $\forall m < n$
$T(n) = 2T(\frac{n}{2}) + \sqrt{n} > 2\frac{n}{2} + \sqrt{n} = n + \sqrt{n} \geq n$
Base case $T(2) = 2T(1) + \sqrt{2} = 2 + \sqrt{2} \geq 2$

2.3 Conclusion

Since we also known that $\Theta(g) = O(g) \cap \Omega(g)$, it can be concluded that:

$$T(n) \in \Theta(n)$$

3 Übung 3.3

4 Übung 3.3(a)

We are going to use the greedy algorithm in order to determine the position of the transmiters in refrence with the houses position. First let's consider an array A which determines the position of the houses along the straight street of size M. The algorithm is going to do the following:

- Sort A in non-decreasing order.
- Use a pointer i to iterate over houses:

- Place a tower at $A[i]+5$, covering the current house and all houses within the next 10 km. $[A[i], A[i]+10]$.
- Move i to the first house not covered by this tower (more than 5 km away).
- Repeat until all houses are covered.
- Return the tower positions and their count.

Pseudocode

Algorithm 1

Data: A: array of house positions, M
Result: T(tower positions), j(minimum number of towers)

Algorithm Towerfind(A, M):

```
Sort(A)
i = 0
j = 0
initialize T
while i < M do
    T[j] = A[i] + 5
    j = j + 1
    i = i + 1
    while i < M and A[i] ≤ T[j − 1] + 5 do
        i = i + 1
    end while
end while
return T and j
```

Runtime:
Sorting A: O(nlogn)
Placing towers: O(n).
Total runtime: O(nlogn).

Correctness
The algorithm is correct because it places each tower at $H[i]+5$, maximizing the coverage range from the current house. By skipping all houses already coverd, it ensures efficiency while satisfying the condition that every house is within 5 km of a tower. This greedy strategy guarantees the minimum number of towers by always covering the largest possible range before moving to the next uncovered house. Thus, it produces an optimal and valid solution.

5 Übung 3.3(b)

To solve the problem of dtermining the fewest drinking breaks Mika's horse needs while riding along the road, we can use a greedy algorithm. The horse can travel at most 20 kilometers without drinking, so the strategy involves choosing water sources strategically to minimize stops while ensuring the horse never exceeds this distance limit. The road is equiped with sufficient water sources, and their positions are provided in a list W.

First, the algorithm sorts W in non-decreasing order to process the water sources sequentally along the road. Mika starts at the beginning of the road (currentPos=0), and at each step, she selects the furthest water source that can be reached without exceeding the horse's 20-kilometer limit (currentPos+20). This ensures that the drinking breaks are spaced as far apart as possible, reducing the total number of stops.

After choosing a water source for a break, Mika updates her position to that source and continues to the next interval. This process repeats until the end of the road is reached. If no water source is found within the alowable range at any step, the task is deemed impossible. However, in this scenario, we assume sufficient water sources are available to guarantee feasibility.

The algorithm is efficient because it iterates through the sorted list of water sources and selects stops in O(n), with an initial sorting step taking O(nlogn). Thus, the total runtime is O(nlogn), which is polynomial.

This approach is optimal because the greedy strategy ensures the minimum number of breaks by always covering the maximum possible distance with each stop. Any alternative that reduces the number of breaks would leave gaps exceeding 20 kilometers, which violates the horse's travel limit.

The optimal solution can be achieved using a greedy approach:

- Sort the positions of the water sources (W) in non-decreasing order.
- Start from the beginning of the road.
- At each step: Select the furthest water source within 20 kilometers of the current position. Stop at this source for a drinking break.
- Move to the next position where the horse can reach without exceeding 20 kilometers.
- Repeat until the end of the road is reached.

Pseudocode

Algorithm 2 Optimal Drinking Break

```
1: Sort(W)                                          ▷ Sort the water source positions
2: i ← 1                                            ▷ Index for water sources (1-based)
3: currentPos ← 0                                   ▷ Current position of the horse
4: j ← 1                                            ▷ Index for drinking breaks
5: P                                                ▷ List of drinking break positions
6: while currentPos < M do
7:     furthest ← −1
8:     while i ≤ M and W[i] ≤ currentPos + 20 do
9:         furthest ← W[i]
10:        i ← i + 1
11:    end while
12:    if furthest = −1 then
13:        return "Impossible"
14:    end if
15:    P[j] ← furthest
16:    j ← j + 1
17:    currentPos ← furthest
18: end while
19: return P, j − 1
```

6 Übung 3.4

1) Problemanalyse:

Gegeben sind:

- n Kinder und m Kekse ($m \geq n$)
- Jedes Kind i hat einen Greedfaktor g_i
- Jeder Keks j hat eine Größe s_j
- Ein Kind ist zufrieden, wenn es einen Keks bekommt, der mindestens so groß wie sein Greedfaktor ist
- Ziel: Maximierung der Anzahl zufriedener Kinder

Algorithm 3

Data: g: Array der Greedfaktoren, s: Array der Keksgrößen
Result: Total nuber of satisfied children

Algorithm CookieAssignment(g[1..n], s[1..m]):

Sortiere beide Arrays aufsteigend
Sort(g)
Sort(s)
$satisfied = 0$ // Zähler für zufriedene Kinder
$i = 1$ // Index für Kinder
$j = 1$ // Index für Kekse
while $i \leq n$ and $j \leq m$ **do**
    if $s[j] \geq g[i]$ **then**: // Kind i bekommt Keks j
        $satisfied = satisfied + 1$
        $i = i + 1$
        $j = j + 1$
    else
        $j = j + 1$
    end if
end while
return $satisfied$

2) Greedy-Algorithmus:

3) Laufzeitanalyse:

- Sortieren beider Arrays: $O(n \log n + m \log m)$
- Durchauf durch die Arrays: $O(n + m)$
- Gesamtlaufzeit: $O(n \log n + m \log m)$

4) Beweis der Korrektheit:

a) Zuläsigkeit:

- Der Algorithmus weist jedem Kind maximal einen Keks zu
- Ein Kind bekommt nur dann enen Keks, wenn dieser groß genug ist ($s[j] \geq g[i]$)
- Da die Arrays sortiert sind, werden keine Zuweisungen übersprugen

b) Optimalität:

Beweis durch Widerspruch:

1. Annahme: Es gibt eine bessere Lösung mit mehr zufriedenen Kindern
2. Sei k die Anzahl zufriedener Kinder in unserer Lösung
3. In der angeblich besseren Lösung müssten $k + 1$ Kinder zufrieden sein
4. Da wir die Greedfaktoren aufstigend sortiert haben und immer den kleinsten möglichen Keks zuweisen, der ein Kind zufriedenstellt, und trozdem nur k Kinder zufriedenstellen konnten, kann es keine Zuweisung geben, die mehr Kinder zufriedenstellt
5. Widerspruch zur Annahme

Die Optimalität ergibt sich daraus, dass wir:

- Die Kinder mit den kleinsten Ansprüchen zuerst bedienen
- Jedem Kind den kleisten passenden Keks geben
- Dadurch die größeren Kekse für anspruchsvolere Kinder aufsparen