Marburg University

Department of Computer Science
Prof. Dr. Sebastian Wild

Date: 2026-06-05
Version: 2026-06-10 17:53

Exercise Sheet 8 zur Vorlesung Algorithmen und Datenstrukturen (Sommer 2026)

Abgabe: *Bis 2026-06-13 18:00, on ILIAS.*

1. Aufgabe

20 Punkte

Geben Sie eine formalen Beweis für das Lemma aus der Vorlesung:

Jeder Binärbaum mit n Knoten besitzt $n+1$ Null-Pointer und eine Höhe von mindestens $\lceil \lg(n+1) \rceil - 1$.

2. Aufgabe

40 Punkte

In dieser Aufgabe werden Sie zeigen, dass Quicksort sogar im Worst-Case mit höchstens logarithmisch viel zusätzlichem Speicher implementiert werden kann. Nebenbei werden wir den Speicherbedarf von Quicksort im Allgemeinen genauer untersuchen.

Obwohl das Partitionieren (wie wir es auf den Folien implementiert haben) kein Puffer-Array benötigt, müssen wir uns für Quicksort die Grenzen rekursiver Aufrufe auf einem Aufruf-Stack merken. In unserer rekursiven Formulierung übernimmt zwar die JVM bzw. Betriebssystem die Verwaltung des dafür nötigen Zusatzspeichers, aber es ist dennoch wichtig, zu wissen, wie viel Platz dafür gebraucht wird. Eine alternative Implementierung mit explizit verwaltetem Stack macht den Speicherbedarf besser sichtbar, daher ist eine solche im folgenden Pseudocode gegeben:

QUICKSORT($A[0..n)$):

```
    // Eingabe: Array A[0..n) der Länge n
1   Stack = ein leerer Stack von zu sortierenden Eingabebereichen (Paare von Indices).
2   Stack.push([0, n))
3   while ¬Stack.empty()
4       [ℓ, r) = Stack.pop()
5       if ℓ < r
6           m = Partition(A[ℓ..r), A[ℓ]).
7           Stack.push([ℓ, m))
8           Stack.push([m + 1, r))
9   end while
```

a) Simulieren Sie diesen Pseudocode auf einem bereits sortierten Array. Da wir stets das erste Element als Pivot wählen, führt dies zu einer Worst-Case-Pivotwahl. Wie verhält sich die Stackgröße?

b) Der Trick zur Reduzierung des Speicherbedarfs von Quicksort besteht darin, die Reihenfolge, in der die beiden Teilprobleme nach der Partitionierung bearbeitet werden, geschickter zu wählen. Hinsichtlich der Korrektheit ist jede Reihenfolge zulässig, da die beiden Teilprobleme unabhängig sind; die Stackhöhe ist jedoch unterschiedlich!

Die einzige notwendige Änderung, um einen deutlich kleineren Stack zu erhalten, besteht darin, das größere Teilproblem zuerst auf den Stack zu legen:

QuicksortSmart($A[0..n)$):

```
    // Eingabe: Array A[0..n) der Länge n
1   Stack = ein leerer Stack von zu sortierenden Eingabebereichen.
2   Stack.push([0, n))
3   while ¬Stack.empty()
4       [ℓ, r) = Stack.pop()
5       if ℓ < r
6           m = Partition(A[ℓ..r), A[ℓ]).
7           if m − ℓ < r − m
8               Stack.push([m + 1, r))
9               Stack.push([ℓ, m))
10          else
11              Stack.push([ℓ, m))
12              Stack.push([m + 1, r))
13  end while
```

Beweisen Sie, dass QuicksortSmart($A[0..n)$) für $n \geq 1$ höchstens $\lfloor \log_2(n) \rfloor + 1$ Stack-Einträge benötigt.

Hinweis: Verwenden Sie vollständige Induktion.
Sei h_n die tatsächliche maximale Stackhöhe, die zum Sortieren einer Teilliste der Größe n benötigt wird; wir kennen h_n nicht (und es könnte eine komplizierte Funktion in n sein), aber es gibt mit Sicherheit eine solche Zahl. Beweisen Sie dann per Induktion, dass $h_n \leq \lfloor \log_2(n) + 1 \rfloor$ für alle $n \geq 1$ gilt.

3. Aufgabe (intervals)

40 Punkte

Lösen Sie `marburg.kilonova.ro/problems/10` (intervals).

ILIAS-Abgabe: Beschreiben Sie in Ihrer ILIAS-Abgabe Ihren verwendeten Algorithmus entsprechend des Templates: *Idee*, *Pseudocode*, *Korrektheit*, *Analyse*. Der Pseudocode-Teil kann hier eine informelle Zusammenfassung Ihres eingereichten Codes auf Kilonova sein.