Präsenzblatt 7

Hinweis: Dieses Aufgabenblatt wurde von Tutor:innen erstellt. Die Aufgaben sind für die Klausur weder relevant noch irrelevant.

Aufgabe 7.1: Wiederholung: Running time

Erklären Sie was folgende Konstellationen bedeuten:

- Best-Case-Laufzeit $\mathcal{O}$
- Worst-Case-Laufzeit $\mathcal{O}$
- Best-Case-Laufzeit Ω
- Worst-Case-Laufzeit Ω

Zeigen Sie die Unterschiede der verschiedenen Begriffe anhand von Algorithmen auf.

Aufgabe 7.2: Korrekturarbeit

Stellen Sie sich vor, Sie sind AlgoDat-Tutor und müssen die Aufgabe 1 vom Übungsblatt 7 korrigieren. Da der größte Teil der Studierenden sich komischerweise dagegen entschieden hat, einfach die universelle Hashfamilie aus den Lösungsvorschlägen vom letzten Jahr abzuschreiben und Ihnen die händische Überprüfung zu mühselig ist, kommen Sie auf die Idee, einen Algorithmus zu entwickeln, der Ihnen die Arbeit erleichtert.

Implementieren Sie den entsprechenden Algorithmus in einer Programmiersprache Ihrer Wahl. Ihr Programm soll also überprüfen, ob eine Hashfamilie universell ist. Falls nicht, soll es auf sinnvolle Art und Weise aufzeigen, wo der Fehler liegt. Führen Sie (wie immer) eine Laufzeitanalyse durch und erläutern Sie kurz, warum Ihr Algorithmus korrekt ist. Sie dürfen vereinfachend annehmen, dass $\mathcal{U} = \{0, \ldots, n-1\}$ für $n \in \mathbb{N}$ und $\forall h \in \mathcal{H} : \forall a \in \mathcal{U} : h(a) \in \{0, \ldots, t-1\}$. Können Sie die gleiche Laufzeitkomplexität erzielen, wenn diese vereinfachenden Annahmen nicht gelten?

Aufgabe 7.3: Auf Kollisionskurs

Mit universellen Hashfamilien konnten wir garantieren, dass die erwartete Anzahl an Kollisionen eine gewisse Schranke nicht übersteigt. Um die Anzahl an Kollisionen niedrig zu halten, könnte man jedoch auch auf die Idee kommen, bei zu vielen Elementen die Hashtabelle zu vergrößern.

Welche Vorteile und welche Probleme gibt es bei dieser Methode? Was ist die erwartete armortisierte Laufzeit zum Suchen eines Elementes?

Aufgabe 7.4: Memoization

Wir haben gesehen, dass man einige Mengenoperationen, insbesondere $\in$ mit Hashfunktionen und Hashtabellen sehr effizient implementieren kann.

1. Schreiben Sie eine rekursive Funktion zur Berechnung der Fibonacci-Folge ($f_0 = 0, f_1 = 1, f_{n+2} = f_n + f_{n+1}$)
2. Wenn ihre Berechnung lineare Zeit oder weniger braucht, schreiben Sie eine naivere Variante.[1]
3. Wenn Sie die Funktion mit großen Zahlen aufrufen, dauert es recht lange. Erklären Sie das Problem.
4. Verbessern Sie den Algorithmus, indem Sie die Funktion so ändern, dass Hashtabellen zum Beheben des identifizierten Problemes verwendet werden. Welche Laufzeit können Sie erreichen?
5. Wie kann man das Vorgehen verallgemeinern? Wo ist ein solches Vorgehen nicht möglich?

[1] Ansonsten überlegen Sie sich, wie man $\mathcal{O}(n)$ oder $\mathcal{O}(\log n)$ Zeit erreichen kann.