Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Übungsblatt 05

Algorithmen I - Sommersemester 2024

Abgabe im ILIAS bis 31.05.2024, 18:00 Uhr

Die Abgabe erfolgt alleine oder zu zweit als *eine* PDF-Datei über das Übungsmodul in der Gruppe eures Tutoriums im ILIAS. Beschriftet die Abgabe deutlich mit Matrikelnummer(n) und Name(n). Bei Abgaben zu zweit reicht es, wenn eine Person die Abgabe im ILIAS hochlädt.

- Achtet bei handschriftlichen Abgaben auf Lesbarkeit.
- Achtet darauf, ob Algorithmen in Worten oder Pseudocode beschrieben werden sollen.
- Es werden immer asymptotisch möglichst effiziente Algorithmen erwartet, wenn nicht anders angegeben.
- Wenn Korrektheits- oder Laufzeitanalysen gefordert sind, behandelt diese separat von der Algorithmenbeschreibung.

Gesamtpunkte: 20 (+ 3 Bonus)

Aufgabe 1 - Total eindeutig (5 Punkte)

In dieser Aufgabe darfst du davon ausgehen, mithilfe einer universellen Familie von Hashfunktionen Hashmaps verwenden zu können, die erwartet konstante Zugriffszeiten haben.

In GBI hast du gelernt, dass eine Relation R auf Menge M eine Teilmenge von $M \times M$ ist. Des Weiteren hast du gelernt, dass eine linkstotale[1] und rechtseindeutige[2] Relation auch Funktion genannt wird. Eine andere Eigenschaft von Relationen, die du kennengelernt hast, ist die Symmetrie[3]. Sei nun $|M| = m$ und $|R| = n$.

Beim Forschen an verschiedenen Relationen stellst du fest, dass es für eine gegebene Menge von Tupeln gar nicht so einfach ist festzustellen, ob diese eine der obigen Eigenschaften erfüllt.

[1] $\forall x \in M : \exists y \in M : (x, y) \in R$
[2] $\forall x, y_1, y_2 \in M : ((x, y_1) \in R \wedge (x, y_2) \in R) \Rightarrow y_1 = y_2$
[3] $\forall x, y \in M : (x, y) \in R \Rightarrow (y, x) \in R$

a) Entwirf einen Algorithmus, der bei Eingabe Grundmenge M und Tupelmenge R in erwartet $\mathcal{O}(n+m)$ Zeit testet, ob R eine linkstotale Relation auf M darstellt. (1.5 Punkte)

b) Entwirf einen Algorithmus, der bei Eingabe von R in erwartet $\mathcal{O}(n)$ Zeit testet, ob R eine rechtseindeutige Relation darstellt. Begründe die Laufzeit und Korrektheit deines Algorithmus. (2 Punkte)

c) Entwirf einen Algorithmus, der bei Eingabe von R in erwartet $\mathcal{O}(n)$ Zeit testet, ob R eine symmetrische Relation darstellt. (1.5 Punkte)

Lösung 1

a) Wir fügen zuerst alle Tupel $(x, y) \in R$ mit Schlüssel x in eine Hashmap ein. Im nächsten Schritt iterieren wir über alle Elemente in M und fragen für jedes dieser Elemente ab, ob ein Tupel für dieses Element in der Hashmap gespeichert ist. Is für ein Element kein Tupel gespeichert, ist die Relation nicht linkstotal.

b) Wir nutzen eine Hashmap, die für ein Tupel (x, y) das linke Element x auf das rechte Element y abbildet. Wir iterieren nun über alle Tupel in R. Für jedes Tupel (x, y) fragen wir zuerst ab, ob sich bereits ein Tupel mit linkem Element x in der Hashmap befindet. Falls dies der Fall ist, testen wir, ob die rechten Elemente gleich sind. Sind diese nicht gleich ist die Relation nicht rechtseindeutig und wir können abbrechen. Anderenfalls fügen wir das Tupel in die Hashmap ein.

   Wir iterieren einmal über alle Tupel von R und müssen für jedes Tupel in der Hashmap abfragen, ob dieses enthalten ist, und es gegebenenfalls einfügen. Da diese Operationen in erwartet konstanter Zeit erfolgen, ergibt sich die geforderte Laufzeit.

   Wir nutzen für die Korrektheit die Tatsache, dass wir alle bisher gesehenen Tupel bereits in die Hashmap eingefügt haben. Wenn also zwei Tupel das selbe erste Element haben, finden wir bei Betrachtung des zweiten Tupels das erste in der Hashmap. Der getätigte Vergleich stellt nun sicher, dass die rechten Elemente übereinstimmen oder wir, wenn nötig, abbrechen.

c) Wir fügen zuerst alle Tupel $(x, y) \in R$ mit Schlüssel (x, y) in eine Hashmap ein. Im nächsten Schritt iterieren wir über alle Tupel (x, y) in R und fragen für jedes dieser Tupel ab, ob ein Tupel (y, x) in der Hashmap gespeichert ist. Ist für ein Tupel seine Umkehrung nicht gespeichert, ist die Relation nicht symmetrisch.

Aufgabe 2 - Konstantes Chaos (8 Punkte)

In dieser Aufgabe darfst du davon ausgehen, mithilfe einer universellen Familie von Hashfunktionen Hashmaps verwenden zu können, die erwartet konstante Zugriffszeiten haben. Außerdem darfst du Arrays und Listen verwenden.

In Dr. Metas Geheimlabor herrscht Chaos. Alle Biber watscheln wild durch die Gegend und stören wichtige Experimente. Nachdem er wieder etwas für Ordnung gesorgt hat, möchte Dr. Meta sicherstellen, dass noch alle benötigten Materialien für seine Experimente vorhanden sind. Glücklicherweise hat er alle Sorten von Materialien von 1 bis m durchnummeriert. Das heißt jede Box mit Materialien ist mit der Nummer seiner Sorte versehen. Dafür benötigt er verschiedene Datenstrukturen.

a) Als erstes möchte Dr. Meta wissen, ob bestimmte Sorten noch vorhanden sind. Entwirf daher eine Datenstruktur, die es dir erlaubt Zahlen von 1 bis m in konstanter Zeit einzufügen und zu entfernen, sowie in konstanter Zeit zu prüfen, ob die Zahl mindestens einmal in der Datenstruktur enthalten ist. Hierbei soll die Datenstruktur auch damit umgehen können, dass die selbe Zahl mehrmals eingefügt wird. Erkläre außerdem, wie die drei Operationen umgesetzt werden. (2 Punkte)

Nachdem Dr. Meta deine Datenstruktur genutzt hat, muss er mit Schrecken feststellen, dass nur von n der ursprünglichen m Sorten überhaupt noch Materialien vorhanden sind. Hierbei ist $n \ll m$ und die noch vorhandenen Sorten können beliebig auf die Nummern 1 bis m verteilt sein. Er möchte sich nun einen Überblick darüber verschaffen, wie viele Boxen von jeder Sorte übriggeblieben sind.

b) Beschreibe nun eine Datenstruktur, die es dir erlaubt die Nummern der noch vorhandenen Boxen in erwartet konstanter Zeit einzufügen und zu entfernen, sowie in erwartet konstanter Zeit die Anzahl der in der Datenstruktur verwalteten Boxen einer Sorte abzufragen. Die Datenstruktur soll nur $\mathcal{O}(n)$ Platz verwenden. Beschreibe wie zuvor auch die Umsetzung der drei Operationen. (2 Punkte)

Die Ergebnisse der Untersuchung sind erschreckend. Von jeder Sorte sind nur noch weniger als n Boxen übrig geblieben.

c) Dir steht nun ein Array zur Verfügung, das für jede noch vorhandene Sorte deren Nummer und Anzahl an Boxen als Tupel speichert. Du sollst nun eine Datenstruktur entwerfen, die die Anzahl der Boxen verwaltet. Diese Datenstruktur soll in erwartet $O(n)$ Zeit erstellt werden können und dir erlauben in konstanter Zeit, für gegebenes k, die k-häufigste Sorte[4] abzufragen. Des Weiteren sollen Änderungen am Datenbestand in erwartet konstanter Zeit möglich sein. Wenn Dr. Meta eine weitere Box in den Untiefen seines Labors findet, soll die Anzahl der entsprechenden Sorte um eins erhöht werden. Wenn die Biber eine Box zerstören, soll die Anzahl der entsprechenden Sorte um eins gesenkt werden. Beschreibe die Datenstruktur und ihre Operationen. Begründe außerdem die zum Erstellen benötigte Zeit und die Laufzeit der Operationen. (4 Punkte)

Hinweis 1: Gehe für diese Aufgabe davon aus, dass zu keinen Zeitpunkt mehr als drei Sorten, die selbe Anzahl an Boxen haben, existieren.

[4]Eine Sorte, die in einer absteigend sortierten Reihenfolge an k-ter Stelle steht, ist eine zulässige k-häufigste Sorte.

Hinweis 2: In dem Fall, dass mehrere Sorten gleich häufig vorkommen, ist die k-häufigste Sorte nicht mehr eindeutig. In diesem Fall darf innerhalb der gleichhäufig vorkommenden Elemente eine beliebige Reihenfolge gewählt werden. Beispiel: Wenn die Sorten a und b jeweils 4 mal vorkommen und die Sorte c 3 mal, wäre a und b jeweils eine korrekte Antwort für die häufigste und auch zweithäufigste Sorte. Für die dritthäufigste Sorte kommt nur c in Frage.

Lösung 2

a) Die Datenstruktur besteht aus einem Array mit m nicht-negativen, ganzzahligen Einträgen, wobei der Eintrag an Index $i \in \{1, \ldots, m\}$ angibt, wie oft das Element i in die Datenstruktur eingefügt wurde. Die Operationen werden dann wie folgt ausgeführt:

- `insert`: Erhöhe den Arrayeintrag an Stelle i um 1.
- `delete`: Verringere den Arrayeintrag an Stelle i um 1, sofern er größer als 0 ist.
- `find`: Gib aus, ob der Arrayeintrag an Stelle i größer als 0 ist oder nicht.

b) Wir verwenden eine Hashmap, die Sorten auf die Anzahl ihrer Boxen abbildet. Wenn wir eine Box der Sorte a einfügen wollen, überprüfen wir, ob zum Schlüssel a bereits ein Eintrag in der Hashmap existiert. Wenn nicht, fügen wir den Wert 1 zum Schlüssel a in die Hashmap ein, ansonsten wird der zugehörige Wert um 1 erhöht. Das Entfernen erfolgt analog, nur dass wir den Wert um 1 senken und den Eintrag entfernen, wenn wir 0 erreichen. Die Abfrage erfolgt durch bestimmen des Wertes, der für die gewünschte Sorte in der Hashmap gespeichert ist.

c) Unsere Datenstruktur besteht aus dem gegebene Array, das wir absteigend nach der Anzahl sortieren und einer Hashmap, die jede Sorte auf den Index, an dem diese gespeichert ist, abbildet.

Die k-häufigste Sorte ist dann die Sorte, die im Array an Index $k-1$ gespeichert ist. Diese können wir somit durch Nachschlagen im Array abfragen. Zur Anpassung der Anzahl einer Sorte nutzen wir zuerst die Hashmap, um den Index der Sorte im Array zu bestimmen. Wir erhöhen bzw. senken dann die im Array gespeicherte Anzahl. Wir müssen nun die Sorte im Array verschieben, falls dieses nicht mehr sortiert ist. Dafür tauschen wir die betrachtete Sorte nach links, beim Erhöhen, bzw. rechts, beim Senken, bis das Array wieder sortiert ist. Durch den Hinweis ist gegeben, dass wir nur konstant viele Stellen in eine Richtung tauschen müssen und danach abbrechen können. Abschließend müssen wir die Hashmap anpassen. Wir entfernen den alten Eintrag für die Sorte in der Hashmap und fügen einen Eintrag mit dem neuen Index ein. Wir können die Datenstruktur erstellen, indem wir das Array mit Bucketsort sortieren und dann über das Array iterieren und die Sorten-Index-Paare in die Hashmap einfügen.

Da wir das Element im Array ans Ende des Bereiches mit gleicher Anzahl verschieben, bleibt dieses auch nach der Anpassung sortiert. Die Abfragezeit ist konstant, da wir nur einen Arraywert auslesen. Die Laufzeit für eine Anpassung ist erwartet konstant, da wir konstant viele Operationen auf einer Hashmap ausführen und im Array nur Daten an bekannten Indizes bearbeiten. Die Laufzeit des Erstellens ist erwartet linear, da die Anzahl einer Sorte durch n beschränkt ist und wir Bucketsort nutzen können. Das Iterieren über die Daten und Einfügen sowie Entfernen in je erwartet konstanter Zeit in die Hashmaps gibt uns die Laufzeitschranke.

Aufgabe 3 - Es ist nicht alles eine gute Hashfunktion, was Chaos stiftet (3 Punkte)

Sei $M \in \mathbb{N}_+$, $U = \{0, 1, \ldots, M-1\}$ ein Universum an Schlüsseln und $m \ll M$ die Größe einer Hashtabelle. Gib für jede der folgenden „Hashfunktionen“ an, welche Nachteile bei der Verwendung auftreten können.

a) $h_1 : x \mapsto (4 \cdot x + 3) \mod m$ für gerades m (1 Punkt)

b) $h_2 : x \mapsto ((5 \cdot x + 4) \mod m) + 1$ (1 Punkt)

c) $h_3 : x \mapsto (x + \text{DICE}(1, 6)) \mod m$
wobei DICE bei jedem Aufruf jeweils mit Wahrscheinlichkeit $1/6$ die Zahlen $1, 2, 3, 4, 5, 6$ ausgibt. (1 Punkt)

Lösung 3

a) Da m gerade und $4 \cdot x + 3$ ungerade ist, bildet h ausschließlich auf ungerade Werte ab und damit nur auf maximal die Hälfte der Elemente von $\{0, \ldots, m-1\}$. Ein Nachteil ist also, dass nur die Hälfte des verfügbaren Speichers genutzt wird.

b) Für alle x mit $(5 \cdot x + 4) \mod m = m - 1$ gilt $h(x) = m$. Doch die Hashtabelle hat nur m Einträge mit Indizes $0, \ldots, m-1$. Ein Nachteil ist also, dass die Funktion auf ungültige Indizes abbildet.

c) Die Ausgabe von h für einen Wert x ist nicht reproduzierbar. Es handelt sich nicht einmal um eine Funktion und ist damit nicht geeignet, um Elemente in einer Hashtabelle zu indizieren.

Aufgabe 4 - Fast keine Kollisionen bei Familiengeburtstagen (4 Punkte)

Sei $M \in \mathbb{N}_+$, $U = \{0, 1, \ldots, M-1\}$ ein Universum an Schlüsseln und $m \ll M$ die Größe einer Hashtabelle.

a) Sei $\mathcal{H} = \{h : x \mapsto (42x + a) \mod m \mid a \in U\}$.
Zeige, dass $\mathcal{H}$ **keine** universelle Familie ist. (2 Punkte)

b) Sei $\mathcal{H} = \{h : x \mapsto (a \cdot x^3) \mod m \mid a \in U\}$.

Zeige, dass $\mathcal{H}$ **keine** universelle Familie ist. (2 Punkte)

Lösung 4

a) Es gilt zu zeigen, dass es $k_1, k_2 \in U$ mit $k_1 \neq k_2$ so gibt, dass für ein zufälliges $h \in \mathcal{H}$ gilt, dass $\Pr[h(k_1) = h(k_2)] > 1/m$. Dazu wählen wir $k_1 \in U$ beliebig und $k_2 = k_1 \pm m$. Sei $h \in \mathcal{H}$ beliebig. Dann:

$$\begin{aligned} h(k_2) &= (42 \cdot (k_1 \pm m) + a) \mod m \\ &\equiv (42 \cdot k_1 + a \pm 42 \cdot m) \mod m \\ &\equiv (42 \cdot k_1 + a) \mod m \\ &\equiv h(k_1) \end{aligned}$$

Also ist die Kollisionswahrscheinlichkeit für k_1 und k_2 gleich $1 > 1/m$.

b) Auch hier suchen wir $k_1, k_2 \in U$ mit $k_1 \neq k_2$ so, dass für ein zufälliges $h \in \mathcal{H}$ gilt, dass $\Pr[h(k_1) = h(k_2)] > 1/m$. Dazu wählen wir $k_1 \in U$ beliebig aber ausreichend klein und $k_2 = k_1 + i \cdot m$ für ein $i \in \mathbb{Z}$ (das ist möglich, wenn $|U| \gg m$ wie angenommen). Sei $h \in \mathcal{H}$ beliebig. Dann:

$$\begin{aligned} h(k_2) &= a \cdot k_2^3 \mod m \\ &\equiv a \cdot (k_1 + i \cdot m)^3 \mod m \\ &\equiv a \cdot (k_1^3 + 3 \cdot (i \cdot m)^2 \cdot k_1 + 3 \cdot i \cdot m \cdot k_1^2 + (im)^3) \mod m \\ &\equiv a k_1^3 \mod m \\ &\equiv h(k_1) \end{aligned}$$

Also ist die Kollisionswahrscheinlichkeit für k_1 und k_2 gleich $1 > 1/m$.

Aufgabe 5 - Zum Knobeln (0 Punkte + 3 Bonus)

Wir betrachten nun erneut das Szenario aus Aufgabe 2 c). Wir wollen die Anzahl der Duplikate in der Datenstruktur nicht mehr durch drei beschränken. Das heißt, dass es nun möglich ist, dass beliebig viele Sorten die selbe Anzahl an Boxen haben.

a*) Erkläre kurz, warum eine Datenstruktur aus Aufgabe 2 c) (die den Hinweis nutzt), bei beliebig vielen Duplikaten falsche Ergebnisse bei der Abfrage der k-häufigsten Sorte liefern kann oder die Laufzeit nicht einhält. (1 Punkt)

b*) Beschreibe nun eine Datenstruktur, die die selben Eingaben erhält und die selben Operationen umsetzt, wie in Aufgabe 2 c). Begründe außerdem auch, warum die Laufzeiten der Operationen und des Erstellens weiterhin die obigen Schranken erfüllen. (2 Punkte)

Hinweis: Eine mögliche Lösung ist es die Datenstruktur aus Aufgabe 2 c) zu erweitern, so dass diese mit Duplikaten umgehen kann.

Lösung 5

a) Die Probleme treten beim Wiederherstellen der Sortierung auf. Da die Datenstruktur nicht weiß, wie viele Duplikate es gibt, muss sie linear den Bereich der Duplikate absuchen. Somit ist entweder die Laufzeit zu langsam oder es wird zu früh abgebrochen und das Array ist nicht mehr sortiert. Sobald das Array nicht sortiert ist, sind auch die Ergebnisse von Anfragen falsch.

b) Im Vergleich zu Aufgabe 2 c) nutzen wir eine weitere Hashmap, die uns angibt, wie weit wir im Array suchen müssen, bis wir eine andere Anzahl finden. Im folgenden beschreiben wir die Datenstruktur erneut vollständig.

Unsere Datenstruktur besteht aus dem gegebene Array, das wir absteigend nach der Anzahl sortieren und einer Hashmap, die jede Sorte auf den Index, an dem diese gespeichert ist, abbildet. Außerdem speichern wir uns eine Hashmap, die für jede Anzahl an Boxen den Bereich, also Start- und Endindex, im Array angibt, in dem sich die Sorten mit dieser Anzahl befinden.

Die k-häufigste Sorte ist dann die Sorte, die im Array an Index $k-1$ gespeichert ist. Diese können wir somit durch Nachschlagen im Array abfragen. Zur Anpassung der Anzahl einer Sorte nutzen wir zuerst die erste Hashmap, um den Index der Sorte im Array zu bestimmen. Wir erhöhen bzw. senken dann die im Array gespeicherte Anzahl. Wir müssen nun die Sorte im Array verschieben, falls dieses nicht mehr sortiert ist. Dafür finden wir mit der zweiten Hashmap den Bereich in dem alle Sorten die alte Anzahl besitzen und tauschen im Array die geänderte Sorte mit der linkesten, beim Erhöhen, bzw. rechtesten, beim Senken, Sorte des Bereiches. Abschließend müssen wir die beiden Hashmaps anpassen. Wir entfernen den alten Eintrag für die Sorte in der ersten Hashmap und fügen einen Eintrag mit dem neuen Index ein. Durch Entfernen, falls vorhanden, und Einfügen passen wir auch die Bereiche für die alte und neue Anzahl der Sorte in der zweiten Hashmap an. Wir können die Datenstruktur erstellen, indem wir das Array mit Bucketsort sortieren und dann über das Array iterieren und die Sorten-Index-Paare in die erste Hashmap einfügen. Beim Iterieren merken wir uns ebenfalls den Startindex des aktuellen Bereichs und können auch so die Daten in die zweite Hashmap einfügen.

Da wir das Element im Array ans Ende des Bereiches mit gleicher Anzahl verschieben, bleibt dieses auch nach der Anpassung sortiert. Die Abfragezeit ist konstant, da wir nur einen Arraywert auslesen. Die Laufzeit für eine Anpassung ist erwartet konstant, da wir konstant viele Operationen auf Hashmaps ausführen und im Array nur Daten an bekannten Indizes bearbeiten. Die Laufzeit des Erstellens ist erwartet linear, da die Anzahl einer Sorte durch n beschränkt ist und wir Bucketsort nutzen können. Das Iterieren über die Daten und Einfügen sowie Entfernen in je erwartet konstanter Zeit in die Hashmaps gibt uns die Laufzeitschranke.

Dr. Meta hat sich bereits an einen geheimen Urlaubsdamm zurückgezogen, um sich von dem beruflichen Stress der letzten Wochen zu erholen.
Er wünscht eine erholsame Pfingst-Pause!