Karlsruher Institut für Technologie

Maximilian Katzmann
Hans-Peter Lehmann
Stefan Walzer

Übungsblatt 12

Randomisierte Algorithmik – Wintersemester 2023/2024

Aufgaben zur Vorlesung vom 25.1.2024
Abgabe im ILIAS bis 1.2.2024, 11:30 Uhr
Besprechung am 1.2.2024, 11:30 Uhr
Achte insbesondere bei handschriftlichen Abgaben auf Lesbarkeit. Die Abgabe erfolgt über das Übungsmodul im ILIAS. Gib Deine Ausarbeitungen in *einer* PDF-Datei ab.

Folgende Aufgabe hat nicht wirklich mit randomisierten Algorithmen zu tun, außer dass wir die Abschätzung in der Vorlesung gebraucht haben. Daher ist sie optional.

Aufgabe 1 – Bonus: Approximationen von e

Du weißt sicher, dass $e = \lim_{n\to\infty}(1+\frac{1}{n})^n$ gilt (das ist sogar eine gängige *Definition* von e). Leite daraus ab, dass die folgenden beiden Gleichungen für alle $n \in \mathbb{N}$ gelten:

$$(1+\tfrac{1}{n})^n \le e \le (1+\tfrac{1}{n})^{n+1}$$
$$(1-\tfrac{1}{n})^n \le e^{-1} \le (1-\tfrac{1}{n})^{n-1}.$$

Folgende Schritte bieten sich an.

(i) Zeige dass für alle $x \in \mathbb{R}$ gilt $1 + x \le e^x$.
Hinweis: Betrachte die Ableitungen bei $x = 0$.

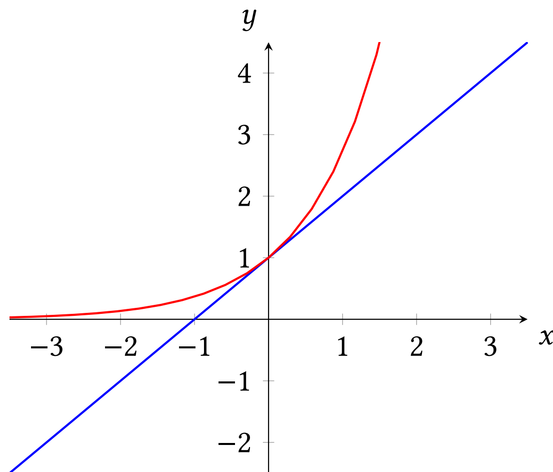
(ii) Folgere aus (i) die linken beiden Ungleichungen.

(iii) Zeige, dass die rechten Seiten monoton in n fallen.

(iv) Folgere aus (iii) und einer Grenzwertbetrachtung die rechten beiden Ungleichungen.

Lösung 1

(i) Sei $f(x) = 1 + x$ und $g(x) = e^x$. Es gilt $f(0) = g(0) = 1$ und $f'(0) = g'(0) = 1$. Daher ist der Funktionsgraph von f eine Tangente des Funktionsgraphen von g. Weil g linksgekrümmt ist (d.h. $g''(x) > 0$ für alle x) verläuft g stets oberhalb von f.



Etwas formaler kann man sich $d(x) = g(x) - f(x) = e^x - x - 1$ anschauen. Es gilt:

- $d(0) = 0$
- $d'(0) = 0$
- $d''(x) > 0$ für alle $x \in \mathbb{R}$.

Damit ist $d(0) = 0$ das globale Minimum von $d(x)$ und somit gilt $g(x) \geq f(x)$ für alle $x \in \mathbb{R}$.

(ii) Es gilt jeweils:

$$(1 + \tfrac{1}{n})^n \leq (e^{1/n})^n = e$$
$$(1 - \tfrac{1}{n})^n \leq (e^{-1/n})^n = e^{-1}$$

(iii) Zunächst erhalten wir durch Logarithmieren der Gleichung aus (i):

$$\forall x \in (-1, \infty) : \ln(1 + x) \leq x$$

Um zu zeigen, dass $f(n) = (1 + \frac{1}{n})^{n+1}$ monoton in $n \in \mathbb{N}$ fällt genügt es zu zeigen, dass $\ln(f(x)) = (x + 1)\ln(1 + \frac{1}{x})$ monoton in $x \in (0, \infty)$ fällt. Dazu betrachten wir die Ableitung und zeigen, dass diese überall kleinergleich 0 ist:

$$(\ln(f(x))' = \ln(1 + \tfrac{1}{x}) + \frac{x+1}{1 + \frac{1}{x}} \cdot (-\tfrac{1}{x^2}) \leq \tfrac{1}{x} - \frac{x+1}{(x+1)x} = 0.$$

Analog argumentieren wir für $g(x) = (1 - \frac{1}{n})^{n-1}$ dass die Ableitung von $\ln(g(x))$ kleinergleich 0 ist:

$$(\ln(g(x))' = ((n-1)\ln(1 - \tfrac{1}{n}))' = \ln(1 - \tfrac{1}{x}) + \frac{x-1}{1 - \frac{1}{x}} \cdot \frac{1}{x^2} \leq -\tfrac{1}{x} + \frac{x-1}{(x-1)x} = 0.$$

(iv) Es gilt durch Separierung eines Faktors:

$$\lim_{n\to\infty}(1+\tfrac{1}{n})^{n+1} = \lim_{n\to\infty}(1+\tfrac{1}{n})^{n} \cdot \lim_{n\to\infty}(1+\tfrac{1}{n}) = e \cdot 1 = e.$$

Den Grenzwert von $(1-\frac{1}{n})^{n-1}$ können wir folgendermaßen bestimmen:

$$\lim_{n\to\infty}(1-\tfrac{1}{n})^{n-1} = \lim_{n\to\infty}(\tfrac{n-1}{n})^{n-1} = \lim_{n\to\infty}\left(\frac{1}{\frac{n}{n-1}}\right)^{n-1} = \frac{1}{\lim_{n\to\infty}(\frac{n}{n-1})^{n-1}}$$
$$= \frac{1}{\lim_{n\to\infty}(1+\frac{1}{n-1})^{n-1}} = \frac{1}{e} = e^{-1}.$$

Also konvergieren die rechten Seite gegen e bzw. e^{-1}. Das tun sie nach (iii) „von oben“. Also gelten die Ungleichungen wie behauptet.

Folgende Aufgabe ist diese Woche als einzige abzugeben.

Aufgabe 2 – Counting Bloomfilter (a.k.a.: Count-Min-Sketch)

Ein Counting Bloomfilter ist zunächst wie ein Bloomfilter: Er verwaltet n Schlüssel in einem Array der Größe m, der *load factor* ist $\alpha := \frac{n}{m}$ und es gibt k Hashfunktionen. Die Parameter seien wie in der Vorlesung gewählt, sodass ein (gewöhnlicher) Bloomfilter eine Falsch-Positiv-Wahrscheinlichkeit von ε hätte. Das Array enthält nun keine Bits mehr sondern natürliche Zahlen. Neben insert und query gibt es nun als weitere Operation delete.

Algorithm insert(x):
  for $i \in [k]$ **do**
    $A[h_i(x)]++$

Algorithm delete(x):
  for $i \in [k]$ **do**
    $A[h_i(x)]--$

Algorithm query(x):
  for $i \in [k]$ **do**
    if $A[h_i(x)] = 0$ **then**
      return false
  return true

Wir lassen zu, dass ein Schlüssel *mehrfach* in den Counting Bloomfilter eingefügt wird. Insofern ist das verwaltete Objekt S nun eine *Multimenge*, deren n Elemente $\{x_1, \dots, x_n\}$ jeweils mit einer Vielfachheit $a_1, \dots, a_n$ vorliegen. Die Operation insert(x) soll der Multimenge S eine Kopie von x hinzufügen, das heißt die Vielfachheit von x erhöhen, falls x bereits in S enthalten war oder ein weiteres Element mit Vielfachheit 1 in S aufnehmen, falls x noch nicht in S enthalten war. Die Bedeutung von delete ist analog.

Wie zuvor auch darf query falsch-positive, aber keine falsch-negativen Antworten liefern.

(a) Wir fordern, dass delete nur für solche Elemente aufgerufen wird, die auch wirklich in S enthalten sind (mit Vielfachheit mindestens 1). Was geht kaputt wenn wir das nicht fordern?

Mit Counting Bloomfiltern lassen sich noch zusätzliche nützliche Operationen implementieren.

(b) Implementiere eine Operation count, die für $x \in D$ einen Schätzwert count(x) für die Vielfachheit a von x in S angibt. Zeige, dass $\Pr[a \neq \text{count}(x)] \leq \varepsilon$ gilt.

(c) Das wichtigste Argument dafür (Counting-) Bloomfilter zu verwenden, ist der geringe Speicherplatz im Vergleich zu einer exakten Datenstruktur. Wir sollten daher besser nicht große Integer Datentypen (etwa 64 Bit) für die Zähler verwenden. Stellen wir uns einen Anwendungsfall vor, in dem die „meisten“ Zähler niemals 8 Bit überschreiten. Wir nutzen daher ein Array A von 8-Bit Zählern. Diskutiere (kurz) die folgenden Vorschläge zum Umgang mit Zählerüberläufen. Was sind die Vorteile und welche Kompromisse werden eingegangen?

- Alice verhindert lediglich Zählerüberläufe, passt also die $A[h_i(x)]++$ Operation in insert und die $A[h_i(x)]--$ Operation in delete so an, dass der Zähler nur inkrementiert bzw. dekrementiert wird, wenn der maximale darstellbare Wert bzw. der minimale darstellbare Wert noch nicht erreicht ist.
- Bob schlägt vor, einen Zähler, der den Wert $(11111111)_2 = 255$ erreicht hat „einzufrieren“, das heißt zukünftige insert oder delete Operationen werden den Zähler nie mehr anpassen.
- Carol schlägt vor, die Bitfolge $(11111111)_2 = 255$ als Markierung dafür zu verwenden, dass der wahre Zählerwert Größer als 254 ist. Der wahre Zählerwert wird in diesem Sonderfall in einer Hashtabelle gespeichert.

Lösung 2

(a) Ein delete(y) für ein y, das niemals eingefügt reduziert unkontrolliert k Zähler im Counting Bloomfilter. So können Zähler 0 werden. Dadurch könnte es sein, dass für ein $x \in S$ anschließend query(x) = false gilt. Also wird für x eine falsch-negative Antwort gegeben, was wir nicht wollen.

(b) Wir geben das Minimum der mit x assozierten Zähler zurück:

```
Algorithm count(x):
    r ← ∞
    for i ∈ [k] do
        r ← min(r, A[h_i(x)])
    return r
```

Zunächst fällt auf, dass $\text{count}(x) < a$ unmöglich ist, denn jedes Vorkommen von x wird von jedem mit x assozierten Zähler gezählt. Es könnte allerdings sein, dass $\text{count}(x) > a$ gilt, wenn alle k Zähler zusätzlich von anderen Schlüsseln verwendet werden.

Um dies zu verstehen sei $A'[1..m] \in \{0,1\}^m$ der gewöhnliche Bloomfilter, in den die Elemente aus S mit Ausnahme von x eingefügt wurden (mit den selben Hashfunktionen, wie für den Counting Bloomfilter). Dann gilt:

$$
\begin{aligned}
\text{count}(x) \neq a &\Leftrightarrow \text{count}(x) > a \\
&\Leftrightarrow \min_{i \in [k]} A[h_i(x)] > a \\
&\Leftrightarrow \forall i \in [k] : A[h_i(x)] > a \\
&\Leftrightarrow \forall i \in [k] : \text{es gibt einen Schlüssel in } S \text{ außer } x \text{ der } A[h_i(x)] \text{ verwendet} \\
&\Leftrightarrow \forall i \in [k] : A'[h_i(x)] = 1 \\
&\Leftrightarrow x \text{ ist falsch-positives Element für } A'
\end{aligned}
$$

Das letzte Ereignis hat nach Wahl der Konfigurationsparameter höchstens Wahrscheinlichkeit ε. Also gilt dies auch für das dazu äquivalente erste Ereignis.

(c) Zu den Vorschlägen ist folgendes zu sagen:

- Der Vorschlag von Alice ist einfach zu implementieren, allerdings werden falsch-negative Antworten möglich. Am einfachsten sieht man das an der Operationsfolge, die 256 mal den selben Schlüssel x einfügt und diesen anschließend 255 mal wieder löscht. Die letzte Einfügung geht verloren und die Löschungen machen alle relevanten Zähler wieder zu 0. Nun würde query(x) als Ergebnis falsch zurückgeben, obwohl der Schlüssel noch einmal in der Datenstruktur sein müsste. Doof.
- Bei Bobs Vorschlag kann ein Zähler niemals fälschlich zurück auf Null fallen, also kann es keinen falsch-negativen Antworten geben. Ein Nachteil ist, dass man eingefrorene Zähler niemals wieder los wird. Langfristig könnte daher die falsch-positiv Wahrscheinlichkeit steigen.
- Carols Vorschlag macht keine Kompromisse bei der Funktionalität. Die zusätzliche Hashtabelle kostet aber selbstverständlich Platz und Zugriffe darauf brauchen Zeit.