Algorithmen & Wahrscheinlichkeit

Übungsstunde 9

Michelle Honegger - ML F34

Minitest

- Passwort: quick
- Zeit: 10 Minuten
- Anzahl Fragen: 10
- 0.2 Punkte pro richtige Antwort

Ablauf

- Minitest
- Theory Recap
- InClass Exercise
- Kahoot!

Minitest

Aufgaben 1

- Wenn ein probabilistischer Algorithmus mit einer Wahrscheinlichkeit von mindestens 2/3 eine korrekte JA/NEIN-Antwort liefert und wir ihn unabhängig n-mal ausführen, ist die Wahrscheinlichkeit, dass er mehr als n/2 Mal eine falsche Antwort gibt, kleiner als exp(-0.001n)

- Es gibt einen probabilistischen Algorithmus, der testet, ob n eine Primzahl ist, und zwar in einer Zeit, die polynominell in log(n) ist.

Minitest

Aufgaben 2

- Sei A ein probabilistischer Algorithmus, dessen Ausgabe immer entweder 0 oder 1 ist, und gelte E[A] = s > 0. Seien ausserdem 0 < ϵ < 1. Wenn wir A unabhängig m = $\lceil 100 \cdot log(1/\delta)/(s\epsilon^2) \rceil$ Mal ausführen und s' als Durchschnitt der Ausgaben definieren, dann gilt mit Wahrscheinlichkeit mindestens 1-δ: $s' \in \lceil (1-\epsilon)s, (1+\epsilon)s \rceil$.

- Wir können jeden Las-Vegas-Algorithmus mit bekannter erwarteter Laufzeit höchstens T in einen randomisierten Algorithmus umwandeln dessen Laufzeit höchstens 10T ist und der mit Wahrscheinlichkeit mindestens 0.9 erfolgreich ist.

Minitest

Aufgaben 3

- Jeder Monte Carlo Algorithmus kann in einen Las Vegas Algorithmus umgewandelt werden.
- Sarah besitzt zwei spezielle Münzen: eine zeigt immer ‘Kopf’, die andere immer ‘Zahl’. Sie wählt eine der beiden Münzen zufällig und wirft sie n = 1000 Mal. Sie bezeichnet mit X die Anzahl ‘Kopf’, die sie sieht. Dann gilt für $\delta := 1/4 : Pr[X \geq (1+\delta)n/2] \leq e^{-\delta^2 n/6}$
- Ein deterministischer Algorithmus kann immer als randomisierter Algorithmus gesehen werden.

Minitest

Aufgaben 4

- Seien X, Y, Z drei unabhängige Zufallsvariablen. Dann gilt immer E[X+Y*Z] = E[X] + E[Y]*E[Z].
- Die erwartete Laufzeit von Quickselect ist O(n).
- Quicksort ist ein Monte-Carlo-Algorithmus.

Quicksort

- Sortiert Array in erwarteter Laufzeit O(nlog(n))

```
QuickSort(A, ℓ, r)
1: if ℓ < r then
2:     p ← Uniform({ℓ, ℓ + 1, ..., r})        ▷ wähle Pivotelement zufällig
3:     t ← Partition(A, ℓ, r, p)
4:     QuickSort(A, ℓ, t − 1)
5:     QuickSort(A, t + 1, r)
```

Quickselect

- Gibt das k-kleinste Element aus in erwarteter Laufzeit O(n)

```
QuickSelect(A, ℓ, r, k)
1: p ← Uniform({ℓ, ℓ + 1, ..., r})          ▷ wähle Pivotelement zufällig
2: t ← Partition(A, ℓ, r, p)
3: if t = ℓ + k − 1 then
4:     return A[t]                           ▷ gesuchtes Element ist gefunden
5: else if t > ℓ + k − 1 then
6:     return QuickSelect(A, ℓ, t − 1, k)    ▷ gesuchtes Element ist links
7: else
8:     return QuickSelect(A, t + 1, r, k − t)  ▷ gesuchtes Element ist rechts
```

Duplikate finden

Gegeben ein Datensatz $S := \{s_1, \ldots, s_n\}$ finde Duplikate,
d.h. finde ein bzw. alle Paare $1 \leq i < j \leq n$ mit $s_i = s_j$.

- Annahme: Vergleich und Speicherzugriffe teuer, $s_1, \ldots, s_n$ sind grosse Elemente
- erste Lösung: Hashfunktion $h : \mathbf{U} \to [m]$
  - effizient berechenbar, zufällig verteilt, unabhängig:
    $\forall u \in U, \forall i \in [m] : Pr[h(u) = i] = \frac{1}{m}$
  - m << |U|

Duplikate finden

- Hashfunktion
  - unabhängig aber $s_i = s_j \Rightarrow h(s_i) = h(s_j)$
  - $m = n^2$

Duplikate finden

- Neues Problem: Finde alle Wiederholungen
  Bloom Filter: verwende k Hashfunktionen

- Bloom-Filter liefern Liste L, sodass
  - jede Wiederholung ist in L
  - für jedes andere i $\in$[n] ist i $\in$L sehr unwahrscheinlich.

Hase Igel Algorithmus

- Haben Array mit n Positionen, jede davon enthält Wert zwischen 1 und n-1
- Konstruiere Graph mit n Knoten und n Kanten, Kanten immer von A[i] zu i
- Von n gibt es Pfad von Länge $k \geq 1$ mit Kreis von Länge $l \geq 3$
- Wir definieren $s \geq 1$ und $0 \leq r < l$
  - Wenn $k \leq l$ -> $s = 1$, sonst $s = k/l$ (aufgerundet)

Hase - Igel Algorithmus

- $x_{k+r} = x_{2(k+r)}$ -> finde diese Position mit Algo
- Berechne k mit $x_k = x_{i+k}$

```
igel = a[n]; hase := a[a[n]]; i := 1
while (igel ≠ hase)
    igel = a[igel]; hase := a[a[hase]]; i := i + 1
```

So far, we mainly used randomization to construct efficient algorithms. However, randomization can also be used to prove statements that don't involve randomness themselves. In fact, such proves can usually be phrased by describing a Monte-Carlo algorithm. Answer the two following questions. If you want to, you can describe your solution as a Monte-Carlo algorithm.

(a) You are given 1000 subsets $A_1, \ldots, A_{1000}$ of $\{1, \ldots, n\}$. Each subset has size at least 11. Can the numbers $1, \ldots, n$ be colored 'red' and 'blue', such that each set A_i contain at least one 'red' and at least one 'blue' number?

(b) You are given 10 points in the plane (i.e., $\mathbb{R}^2$). Can you place (filled) disks of radius 1 in the plane, such that (i) each of the 10 points is contained in a disk, and (ii) all disks are disjoint (i.e., their centers have distance at least 2)?
Remark: You may cover multiple points with the same disk. In particular, the task would be trivial if all points are very close to each other (then you only need one disk to cover all of them). Similarly, if all points are very far apart, you could use one disk per point without having to worry about disjointness.
Remark: a formal solution to this exercise would be very technical. Focus more on the ideas than on technical details

Aufgabe

Das folgende Problem ist als ‘Element-Uniqueness-Problem’ bekannt: Gegeben ist ein Array von Zahlen $a_1, \ldots, a_n$, zum Beispiel vom Typ `double`. Wir wollen herausfinden, ob zwei der Elemente des Arrays identisch sind, also ob es $1 \leq i < j \leq n$ gibt, so dass $a_i = a_j$.

Deterministisch lässt sich dieses Problem mithilfe von zwei for-loops in Zeit $O(n^2)$ lösen. Ein etwas geschickterer Ansatz, der einen schnellen Sortieralgorithmus verwendet, benötigt Zeit $O(n \ln n)$. In der Praxis lässt sich aber mit Hash-Tables ein noch schnellerer Ansatz finden.

Wir nehmen an, dass eine Hashfunktion h gegeben ist, die als Input Zahlen vom Typ `double` verwendet und als Ausgabe eine ganze Zahl zwischen (einschliesslich) 1 und n generiert. Die Idee ist, zunächst n Listen $L_1, \ldots, L_n$ zu generieren, wobei die Liste L_k alle Indizes $i \in \{1, \ldots n\}$ enthält sodass $h(a_i) = k$, und dann jede Liste L_k genauer zu untersuchen.

(a) Seien n, $a_1, \ldots, a_n$ und die Hashfunktion h gegeben. Sei

$$C = \{(i, j) : a_i \neq a_j \text{ und } h(a_i) = h(a_j)\}.$$

Beschreiben Sie einen Algorithmus der das Element-Uniqueness-Problem in Zeit $O(n + |C|)$ löst und $O(n)$ Speicherplatz benötigt.

Wir nehmen hierbei an, dass $h(x)$ in Zeit $O(1)$ berechnet werden kann.

(b) Angenommen, h wird zufällig gewählt, sodass für alle $a \neq b$ gilt

$$\Pr[h(a) = h(b)] = \frac{1}{n}.$$

Zeigen Sie, dass die erwartete Laufzeit des in (a) konstruierten Algorithmus $O(n)$ ist.