RWTH Aachen **SS 2018**
Lehrgebiet Theoretische Informatik Blatt 5
Rossmanith–Dreier–Kuinke 28.5.2018

Übung zur Vorlesung Algorithmen und Datenstrukturen

Aufgabe T10

Betrachten Sie dieses Programm:

```
public class WasMachtDas {
    public static void main(String[] args) {
        int n = 1000;
        boolean notp[] = new boolean[n];
        for(int k = 2; k < n; k++) {
            for(int i = 2 * k; i < n; i += k) notp[i] = true;
        }
        for(int i = 2; i < n; i++) {
            if(!notp[i]) System.out.println(i);
        }
    }
}
```

a) Was macht dieses Programm?

b) Wie groß ist die Laufzeit abhängig von n?

c) Wenn in der äußeren Schleife die Abbruchbedingung durch $i \leq \sqrt{n}$ ersetzt würde, würde sich die Laufzeit um mehr als einen konstanten Faktor verringern?

Hinweis: $\sum_{k=1}^{n} 1/k = H_n = \ln n + \gamma + O(1/n)$ mit $\gamma \approx 0.5772156649$.

Lösungsvorschlag:

a)
Das Programm gibt alle Primzahlen um Interval $[2, 999]$ aus und geht dabei wie folgt vor:

1. Anfangs werden alle Zahlen als Primzahl angenommen.
2. Für jede Zahl werden alle ihre vielfachen auf nicht prim gesetzt.
3. Am Ende bleiben nur die Zahlen prim die nicht vielfaches einer vorangegangen Zahl sind.

b)
Die äußere Schleife benötigt $O(n)$ und die innere Schleife benötigt $O(n/k)$ Durchläufe. Insgesamt ergibt sich $O(\sum_{k=2}^{n} n/k) = O(n \log n)$. Die letzte Schleife benötigt wieder $O(n)$ Durchläufe, allerdings als additiver Term der vernachlässigt werden kann. Die gesamt Laufzeit beträgt also $O(n \log n)$.

c)
Die Laufzeit würde sich ungefähr um den Faktor $H_n/H_{\sqrt{n}}$ verbessern. Wir erhalten

$$\frac{\ln n + \gamma + O(1/n)}{\ln \sqrt{n} + \gamma + O(1/\sqrt{n})} = O(1)$$

Aufgabe T11
Entwerfen Sie eine gute Hashfunktion für binäre Suchbäume.

Lösungsvorschlag:
Wir starten mit einer Hashfunktion die drei Zahlen miteinander hasht:
`hash(a,b,c):=3*a+6b+c mod m`
Um einen Knoten zu hashen wollen wir seinen Key, den Hashwert seines linken und den Hashwert seines rechten Teilbaumes zusammenhashen. Diese Funktion rufen wir dann auf der Wurzel aus. Das Ganze könnte etwa so aussehen:
`hashTree(Node):=hash(hashTree(Node.left), hashTree(Node.right), Node.key)`.

Aufgabe H8 (8 Punkte)
Wir betrachten die Hashfunktion

$$h\colon \Sigma^* \to \{0, \ldots, m-1\},\ c_1 c_2 \ldots c_k \mapsto \Big(\sum_{i=1}^{k} c_i\Big) \bmod m,$$

welche die Menge Σ^* aller Wörter (im ASCII-Alphabet) auf eine Zahl zwischen 0 und $m-1$ abbildet.

a) Ist h ein gute Hashfunktion? Überlegen Sie sich realistische Anwendungsbeispiele, für welche h sehr viele Kollisionen erzeugt.

b) Wie könnte eine vernünftige Hashfunktion für Zeichenketten aussehen, für welche Sie unter normalen Umständen ein gutes Verhalten erwarten würden? Wie gehen Sie mit dem Fall um, daß m sehr groß ist?

Lösungsvorschlag:

a) Wenn wir Wörter hashen, die alle Permutationen voneinander sind, dann werden *alle* auf denselben Wert abgebildet – schlechter geht es wirklich nicht. Außerdem werden ja alle kurzen Wörter auf relativ kleine Zahlen abgebildet, so daß nur ein verschwindend kleiner Teil einer sehr großen Hashtabelle verwendet würde. Dort gäbe es dann sehr viele Kollisionen.

b) Eine einfache Möglichkeit ist eine Linearkombination der c_i mit geeigneten Koeffizienten, die mit steigendem i auch größer werden.

Aufgabe H9 (16 Punkte)

Am Ende dieses Blatts und auf der Webseite finden Sie ein Programm, das eine Datei lesen kann, welche je Zeile ein Wort enthalten soll. Das Programm zählt, wieviele verschiedene Wörter es in der Datei gibt, wobei es eine Hashtabelle verwendet.

a) Wenn Sie das Programm auf einer typischen Datei laufen lassen, wie lang ist die Laufzeit auf Ihrem Computer typischerweise für eine Datei mit n Bytes?

b) Erzeugen Sie eine Eingabedatei, deren Größe 1MB nicht überschreitet und deren Bearbeitung länger als 5 Sekunden dauert. Machen Sie die Datei Ihrem Tutor zugänglich. Erklären Sie, wie sie die Datei erstellt haben.

c) Könnten Sie solch eine furchtbare Datei auch dann erstellen, wenn im Programm h nicht mit x sondern mit y multipliziert würde? (Eine sehr kurze Antwort reicht hier.)

```
public class WordCount {
    static int x = 37;
    static int y = Math.abs((new Random()).nextInt());

    public static int hash(String w) {
        int h = 0;
        for(int i = 0; i < w.length(); i++) {
            h = h * x + w.charAt(i);
        }
        return Math.abs(h);
    }

    public static void main(String args[]) throws IOException {
        String filename = args[0];
        List<String> words = Files.readAllLines(Paths.get(filename));
        int m = 100000;
        long count = 0;
        ArrayList<LinkedList<String>> slot = new ArrayList<>();
        for(int i = 0; i < m; i++) {
            slot.add(new LinkedList<String>());
        }
        for(String w : words) {
            LinkedList<String> list = slot.get(hash(w)%m);
            if(!list.contains(w)) {
                count = count + 1;
                list.add(w);
            }
        }
        System.out.println(count + " different words in " + filename + ".");
    }
}
```

Lösungsvorschlag:

a)

Falls wir das Programm auf zufällig generierten Eingaben laufen lassen, können wir beobachten, dass es sehr schnell ist.

b)

Um eine lange Luafzeit zu erzwingen, wollen wir viele Wörter mit dem gleichen Hash einfügen, welche dann alle in einer einfachen Liste gespeichert werden, um eine quadratische Laufzeit zu bekommen. Dies können wir erreichen indem wir mithilfe eines Programm zufällig Wörter erzeugen die und dann jene in einer Datei speichern, welche den selben Hashwert haben bis wir eine Größe von 1MB erreicht haben. Diese Eingabedatei wird auf den meisten Computern zu einer Laufzeit von über 5 Sekunden führen.

c)

Nein können wir nicht, da wir dann keine Chance haben vorher zu wissen auf welchen Hash ein Wort abgebildet wird.