# Übung 7 Abgabe 13.6.2019

## Aufgabe 1 – Konstruieren von Adversarial Keys 8 Punkte

Hashtabellen werden ineffizient, wenn die verwendete Hashfunktion viele Kollisionen verursacht. Ein Hacker, der die Hashfunktion kennt, kann diese Tatsache ausnutzen, um absichtlich ineffiziente Anfragen zu konstruieren: Er wählt die in der Anfrage verwendeten Schlüssel gezielt so, dass alle auf den gleichen Hashwert abgebildet werden ("*adversarial keys*"). Werden viele ineffiziente Anfragen dieser Art gleichzeitig gestartet, kann die angegriffene Webseite (die die gegebene Hashfunktion intern benutzt) zusammenbrechen ("*denial of service*"). In der Praxis verhindert man dies durch die Verwendung von universellem Hashing, indem man die Hashfunktion per Zufall aus einem großen Pool erlaubter Hashfunktionen auswählt. Dann ist es nicht mehr möglich, im Vorhinein ein ungünstiges Set von Schlüsseln zu konstruieren. Für diese Übungsaufgabe wollen wir aber annehmen, dass stets die folgende einfache Hashfunktion verwendet wird:

```
def hhash(s): # s ist ein Schlüssel vom Typ string
    h = 0     # der Hashwert wird mit 0 initialisiert
    for k in s:
        h = 23*h + ord(k)  # Aktualisieren des Hashs mit dem Zeichencode
    return h
```

Dabei gibt `ord(k)` den Zeichencode des Zeichens `k` zurück. Dies ist eine Variante der Bernsteinfunktion aus der Vorlesung, wo der Multiplikator „33“ durch „23“ ersetzt wurde. Finden Sie mindestens 16 Schlüssel (Strings) der Länge 4, die alle den gleichen Hashwert haben. Geben Sie diese Schlüssel im File "`collisions.txt`" ab (ein String pro Zeile) und beschreiben Sie, wie Sie vorgegangen sind, um diese Schlüssel zu finden. Hinweis: Beginnen Sie damit, Kollisionen mit Schlüsseln der Länge 2 zu konstruieren und verwenden Sie diese Ergebnisse zur Konstruktion von Schlüsseln der Länge 4.

## Aufgabe 3 – BucketSort 14 Punkte

Gegeben sei eine Liste von Punkten, die im Einheitskreis gleichverteilt sind, d.h. jeder Punkt im Einheitskreis hat die gleiche Chance, in der Liste enthalten zu sein. Ihre Aufgabe besteht darin, die Punkte mittels BucketSort nach ihrem Abstand vom Koordinatenursprung $r = \sqrt{x^2 + y^2}$ (aufsteigend) zu sortieren. Dafür müssen Sie zunächst mit Hilfe des Python-Moduls `random` und dem sogenannten *rejection sampling* Testdaten erzeugen:

```
def createData(size):
    a = []
    while len(a) < size:
        x, y = random.uniform(-1, 1), random.uniform(-1, 1)
        r = math.sqrt(x**2 + y**2)
        if r < 1.0:
            a.append(r)
    return a
```

Die Funktion arbeitet folgendermaßen: Zunächst werden *x* und *y* als gleichverteilte Zufallszahlen im Intervall [-1, 1] gezogen. Damit sind die Punkte (*x,y*) gleichverteilt im Quadrat [-1,1]x[-1,1]. Durch den Test *r* < 1.0 wird geprüft, ob der Punkt (*x,y*) sogar im Einheitskreis (der ja eine Teilmenge des Quadrats ist) liegt. Nur dann wird er in die Liste *a* übernommen, andernfalls wird er ignoriert („rejection“). Der Einfachheit halber speichern wir nur die Abstände *r* in der Liste, weil die Koordinaten für die Aufgabe nicht mehr benötigt werden. Beachten Sie, dass die *r*-Werte im Intervall [0,1) nicht gleichverteilt sind.

a) Um BucketSort zu implementieren, brauchen Sie neben dem eigentlichen Algorithmus eine geeignete Funktion `bucketMap(r, M)`, die den Sortierschlüssel `r` auf einen Index im Bereich [0, *4 Punkte*

*M*) abbildet, wenn *M* Buckets verwendet werden sollen. Da *r* im Intervall [0,1) liegt, bietet sich als naive Implementierung `index = int(r*M)` an. Allerdings werden die Schlüssel dann nicht gleichmäßig auf die Buckets verteilt – kleine Indizes kommen viel seltener vor als große (probieren Sie das aus!). Die richtige `bucketMap` Funktion sieht so aus:

```
def bucketMap(r, M):
    return int(r**2 * M)
```

Begründen Sie, warum diese Funktion im Mittel zu einer Gleichverteilung der Indizes führt.

b) Um experimentell zu prüfen, ob eine gegebene Funktion `bucketMap` zu einer gleichmäßigen Verteilung der Schlüssel auf die Buckets führt, kann man den $\chi^2$-Test (sprich „Chi-Quadrat-Test“) verwenden. Angenommen, es sollen *N* Schlüssel auf *M* Buckets verteilt werden. Erfolgt die Aufteilung wirklich gleichmäßig, sollte jeder Bucket im Durchschnitt $c = N/M$ Schlüssel enthalten. Da die Schlüssel jedoch Zufallszahlen sind, wird das nur selten ganz exakt stimmen, und der $\chi^2$-Test prüft, ob die Schwankungen innerhalb der erlaubten Grenzen bleiben. Man berechnet dazu die gewichtete Summe der quadratischen Abweichungen vom erwarteten Wert *c*, also die Größe *5 Punkte*

$$\chi^2 = \sum_{k=0}^{M-1} \frac{(n_k - c)^2}{c}$$

wobei $n_k$ die tatsächliche Anzahl der Schlüssel im Bucket *k* sind. Wenn die Hypothese „Schwankungen liegen in den erwarteten Grenzen“ zutrifft, folgt $\chi^2$ einer sogenannten Chi-Quadrat-Verteilung mit (*M*-1) Freiheitsgraden. Für genügend große *N* und *M* kann man dies weiter vereinfachen, weil dann die Größe $\tau = \sqrt{2\chi^2} - \sqrt{2M-3}$ näherungsweise wie eine Gaußsche Glockenkurve mit Mittelwert 0 und Standardabweichung 1 verteilt ist. Es gilt dann einfach: die Hypothese trifft mit 99.7%-iger Wahrscheinlichkeit *nicht* zu, die Daten sind also *nicht* gleichmäßig verteilt, wenn $|\tau| > 3$ ist.

Implementieren Sie diesen vereinfachten Test im File „`bucket_sort.py`“ mit einer Funktion `chi2Test(bucket_lens,N)`, der Bucketlängen eines bereits gefüllten Bucketarray übergeben wird, und die `True` zurückgibt, wenn der Test bestanden wurde. Testen Sie für verschiedene Zufallsarrays und verschiedene *M*, ob Ihre Funktion `bucketMap` den Test besteht. Zeigen Sie außerdem, dass die naive Formel `int(r*M)` den Test nicht besteht. Erzeugen Sie die Zufallsarrays mit der oben angegebenen Funktion `createData`, die ebenfalls in `bucket_sort.py` enthalten sein soll.

c) Implementieren Sie `bucketSort` (siehe Vorlesung – vergessen Sie nicht, die Korrektheit Ihrer Implementation zu testen!) und zeigen Sie, dass die Laufzeit tatsächlich nur linear mit der Länge des Eingabearrays wächst, wenn $c = N/M$ genügend klein gewählt wird (Werte zwischen 2 und 6 sollten gute Ergebnisse liefern). Verwenden Sie sowohl die naive Formel `int(r*M)` als auch Ihre Funktion `bucketMap` und vergleichen Sie die Laufzeiten. Sie werden feststellen, dass die optimale Wahl der Funktion `bucketMap` nicht sehr kritisch ist – der $\chi^2$-Test ist hier etwas zu pingelig. Die Lösung soll ebenfalls in `bucket_sort.py` enthalten sein. *5 Punkte*

Die Laufzeit in Abhängigkeit der Arraygröße können wir mit Hilfe des timeit-Moduls ermitteln. Dabei wird zehnmal die Zeit für jeweils einen Durchlauf gemessen und anschließend das Minium als Laufzeit verwendet. Dadurch kann man verhindern, dass Prozesse im Hintergrund die Zeitmessung stören. Mit Hilfe von `matplotlib` können wir die Werte in einem Diagramm darstellen. In diesem Beispiel ist `c = 3`. Der entsprechende Code sieht folgendermaßen aus:

```
import timeit
import matplotlib.pyplot as plt
```

```
t = []
t_naive = []
size = []
for n in range(1000,10001,1000):
    timer_1 = timeit.Timer(stmt= 'bucketSort(a,bucketMap)',
                           setup = 'from __main__ import insertionSort,'+
                           'bucketSort, bucketMap, createData \n' +
                           'a = createData('+ str(n) + ')\n')
    timer_2 = timeit.Timer(stmt= 'bucketSort(a,naiveBucketMap)',
                           setup = 'from __main__ import insertionSort,'+
                           'bucketSort, naiveBucketMap, createData \n' +
                           'a = createData('+ str(n) + ')\n')
    time_1 = timer_1.repeat(repeat = 10, number = 1)
    time_2 = timer_2.repeat(repeat = 10, number = 1)
    t.append(min(time_1))
    t_naive.append(min(time_2))
    size.append(n)

plt.xlabel('Anzahl der Elemente')
plt.ylabel('Laufzeit [s]')
plt.title('Laufzeit bucketSort')
plt.axis([0,10100,0,0.008])
plt.plot(size,t,'ro', label='bucketMap')
plt.plot(size,t_naive,'b*', label='naiveBucketMap')
plt.legend(loc='upper left')
plt.show()
```

Bitte laden Sie Ihre Lösung bis zum 13.6.2019 um 12:00 Uhr auf Moodle hoch.