Lösungen der Hausaufgaben von Übungsblatt 3

Algorithmen und Datenstrukturen (WS 2013, Ulrike von Luxburg)

Lösungen zu Aufgabe 1
Schreiben $\mathbb{N} := \mathbb{N}_0$, und für $a, b \in \mathbb{N}$ auch $a\mathbb{N} + b := \{a \cdot t + b \mid t \in \mathbb{N}\}$, beispielsweise $5\mathbb{N} + 2 = \{2, 7, 12, 17, \ldots\}$. Jede der Hashfunktionen ist von der Form $h(k) =: g(k) \bmod 11$. An Indexposition 10 kollidieren genau die Keys mit $g(k) \bmod 11 \equiv 10$, also für $g(k) \in \{10, 21, 32, 43, \ldots\} = 11\mathbb{N}+10$.

(a) $g(k) = k$. Kollisionen falls $g(k) = k \in 11\mathbb{N} + 10$, also für jedes k das sich schreiben lässt als $k = 11 \cdot t + 10$ für ein $t \in \mathbb{N}$, also für jedes $k \in 11\mathbb{N} + 10$ selbst.

(b) $g(k) = 2k$. Kollisionen falls $g(k) = 2k = 11 \cdot t + 10$, also aller (natürlichen!) k der Form $k = 11 \cdot t/2 + 5$. Da $k \in \mathbb{N}$ nur für $t = 0, 2, 4, \ldots$ gilt, liefert dies $k = 5, 16, 27, 38, \ldots$. Also Kollision aller $k \in \{11 \cdot t + 5 \mid t \in \mathbb{N}\} = 11\mathbb{N} + 5$.

(c) $g(k) = k^2 + 10$. Da $k^2 + 10 \bmod 11 \equiv 10$ genau dann wenn $k^2 \bmod 11 \equiv 0$, müssen wir also alle solchen Quadratzahlen k^2 bestimmen $(1, 4, 9, 16, 25, \ldots)$ die durch 11 teilbar sind. Also muss 11 ein Primfaktor von k^2 sein. Außerdem tritt bei Quadratzahlen jeder Primfaktor eine gerade Anzahl oft auf. Also erhält man $k^2 = 11^2 \cdot a^2 = (11a)^2$ für beliebige $a \in \mathbb{N}$ als genau die gesuchten Lösungen, somit $k \in 11\mathbb{N}$.

(d) $g(k) = 3^k - 1$. Da $3^k - 1 \bmod 11 \equiv 10$ genau dann wenn $3^k \bmod 11 \equiv 0$, müssen wir also alle Potenzen von 3 bestimmen die durch 11 teilbar sind. Da 3^k nur 3 als Primfaktor enthält, ist dies nie der Fall. Also wird kein k auf Position 10 abgebildet, somit gibt es dort insbesondere keine Kollisionen.

Lösungen zu Aufgabe 2
An $\frac{n!}{n^n} = \frac{n(n-1)(n-2)\cdots 2\cdot 1}{n\cdot n\cdot n\cdots n\cdot n}$ kann man durch elementweisen Vergleich ablesen, dass $n! \leq n^n$. Somit folgt aufgrund der Monotonie des Logarithmus auf $\mathbb{R}_{>0}$, dass $\log(n!) \leq \log(n^n) = n\log(n) \in \Theta(n \log n)$, also $\log(n!) \in \mathcal{O}(n \log n)$.

Analog kann man an $\frac{(n/2)^{n/2}}{n!} = \frac{(n/2)(n/2)\cdots(n/2)}{n\cdot(n-1)\cdots(n/2)} \cdot \frac{1\cdots 1}{(n/2-1)\cdots 1}$ ablesen, dass $(n/2)^{n/2} \leq n!$ gilt. Somit $\log(n!) \geq \log((n/2)^{n/2}) = \frac{n}{2}\log(n/2) = \frac{n}{2}\log(n) - \frac{n}{2}\log(2) \in \Theta(n\log(n) + n) = \Theta(n\log(n))$, also $\log(n!) \in \Omega(n\log(n))$.

Zusammengefasst liefert $\log n! \in \Theta(n \log n)$.

Lösungen zu Aufgabe 3

(a) Das Finden des Medians einer Eingabesequenz der Länge ℓ benötigt nach Aufgabenstellung[1]. $\mathcal{O}(\ell)$ Zeitaufwand. Dieses Pivotelement halbiert die Eingabesequenz in zwei Teile der Länge $\ell/2$, auf welchen dann rekursiv fortgefahren wird. Im worst-case taucht das Pivotelement zudem nur einmal in der Eingabesequenz auf. Die Laufzeit auf Eingabegröße n ist somit $T(n) \leq 2T(\lceil n/2 \rceil) + \mathcal{O}(n)$, wobei in $\mathcal{O}(n)$ sowohl das Finden des Pivotelement, das Zerlegen in die Teilsequenzen, als auch das Zusammensetzen der Teillösungen zur Gesamtlösung enthalten ist. Mit $T(1) = \Theta(1)$ und dem Master-Theorem folgt, dass $T(n) = \mathcal{O}(n \log n)$ die worst-case Laufzeit dieser Quicksort-Variante ist. Asymptotisch im worst-case ist diese Variante also besser als der randomisierte Quicksort oder andere Quicksort-Varianten mit worst-case Laufzeit $\mathcal{O}(n^2)$. Zudem ist $\mathcal{O}(n \log n)$ scharf, da auch diese Quicksort-Variante vergleichsbasiert ist, und somit $\Omega(n \log n)$ nicht unterbieten kann.

[1] der ausgeteilte Aufgabenzettel enthielt diese Information noch nicht, daher werden auch die aus der Vorlesung bekannten worst-case Lösungen zum Ermitteln des Medians akzeptiert

(b) Die Laufzeit-Konstanten zum Finden des Medians in jedem Zwischenschritt sind schlichtweg zu groß, so dass in der Praxis auf den meisten Eingaben der randomisierte Quicksort derart schneller ist, dass die worst-case Zusicherung des Median-Pivots irrelevant ist. Wenn man wirklich $\mathcal{O}(n \log n)$ zusichern will, kann man ggf. besser Merge-Sort wählen.

(c) Nein, da das arithmetische Mittel nicht wie der Median unabhängig von einer starken Unbalancierung der Eingabedaten ist. Betrachte beispielsweise die Eingabe $(n, n^2, n^3, \ldots, n^n)$ der Länge n. Das arithmetische Mittel all dieser Zahlen ist $\overline{a}_n = \frac{1}{n}\sum_{i=1}^{n} n^i = \sum_{i=0}^{n-1} n^i$ und ist offensichtlich größer als n^{n-1}. Damit wird bestenfalls das vorletzte Element des Arrays als Pivot gewählt. Dies teilt die Eingabe in

$$\underbrace{(n, n^2, n^3, \ldots, n^n - 2)}_{a}\underbrace{(n^{n-1})}_{b}\underbrace{(n^n)}_{c},$$

also insbesondere eine Sequenz a der Länge mindestens $n-2$.

Nun wird rekursiv auf Sequenz a der Länge k fortgefahren, also auf der Eingabesequenz $(n, n^2, n^3, \ldots, n^k)$. Dessen arithmetisches Mittel ist ganz analog zu oben $\overline{a}_k = \frac{1}{n}\sum_{i=1}^{k} n^i = \sum_{i=0}^{k-1} n^i$ und offensichtlich größer als n^{k-1}. Wieder teilt sich die Eingabe bestenfalls in eine Sequenz der Länge $k-2$, die Pivotsequenz b der Länge 1 und die Sequenz c der Länge 1.

Insgesamt wird also mindestens $n/2$ Mal rekursiv abgestiegen. Da die Ermittlung des arithmetischen Mittels einer Eingabesequenz der Länge k mindestens Zeit $c{\cdot}k$ für eine Konstante $c \geq 1$ benötigt, fällt somit mindestens Aufwand $cn + c(n-2) + c(n-4) + \ldots + 2c = c\sum_{i=1}^{n/2} 2i \in \Omega(n^2)$ an. Damit erhalten wir die Gesamtlaufzeit $\Omega(n^2)$.

Lösungen zu Aufgabe 4

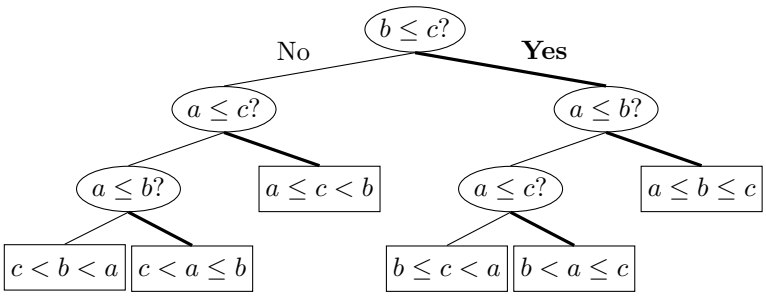
(a) Jedes der $n = 2^k$ Elemente kann eindeutig über einen Array-Index $0, \ldots 2^k - 1$ identifiziert werden, also anhand einer k-Bit-Zahl. Wir nutzen k Münzwürfe, um Bit für Bit eine zufällige k-Bit-Zahl zu konstruieren. Das zugehörige Element wählen wir aus. Die Wahrscheinlichkeit ist für jedes Element exakt gleich $(\frac{1}{2})^k = \frac{1}{n}$. Die Anzahl Münzwürfe ist $k = \log_2 n \in \mathcal{O}(\log n)$.

(b) Sei $k_n := \lceil \log_2(n) \rceil$. Somit gilt $2^{k_n} \geq n \geq 2^{k_n - 1}$. Wählen nun irgendein $c \geq 1$ und setzen $k := \lceil c k_n \rceil$. Nun bestimmen wir wie zuvor eine uniform zufällige k-Bit-Zahl $x \in \{0, \ldots, 2^k - 1\}$ durch k Münzwürfe. Dann berechnen wir $i := \lfloor n \cdot x / 2^k \rfloor \in [0, n-1]$ und wählen das Element $A[i]$ aus. Die ungefähre Gleichverteilung rührt daher, dass in der Berechnung von i der Zahlenbereich $D := \{0, \ldots, 2^k - 1\}$ gleichmäßig auf das Intervall $[0, n - \frac{1}{2^k}]$ abgebildet wird. Daher fallen in jedes Teilintervall $[0,1)$, $[1,2)$, $[2,3)$, …, $[n-1, n)$ ungefähr gleichviele (± 1) Zahlen aus D, so dass jedes Teilintervall $[i, i+1)$, und somit jedes Element i, mit etwa derselben Wahrscheinlichkeit ausgewählt wird. Die Genauigkeit kann durch größere Werte für c beliebig verbessert werden. Die Anzahl Münzwürfe beträgt $ck_n \in \mathcal{O}(\log n)$.

(c) Wähle k_n wie in (b), und eine zufällige k_n-Bit-Zahl $x \in \{0, \ldots, 2^{k_n} - 1\}$ mit k_n Münzwürfen. Falls $x < n$, so geben wir $A[x]$ aus. Die Wahrscheinlichkeit ist für jedes Element exakt gleich $(\frac{1}{2})^{k_n}$. Falls $i \geq n$, so wählen wir mittels weiteren k_n Münzwürfen eine neue zufällige k_n-Bit-Zahl x und prüfen erneut auf $x < n$. Dies wiederholen wir solange, bis der Fall $x < n$ erreicht wird. Die Wahrscheinlichkeit für den Fall $x < n$ ist in jedem Durchgang $\geq \frac{1}{2}$, weil $n \geq 2^{k_n - 1} = \frac{1}{2} 2^{k_n}$. Damit ist die erwartete Anzahl Versuche mit dem Tipp aus der Aufgabenstellung höchstens 2. Die erwartete Anzahl Münzwürfe also höchstens $2k_n \in \mathcal{O}(\log n)$. Wie beschrieben ist die Wahrscheinlichkeit für jedes $A[x]$ exakt gleich, also $1/n$.

Lösungen zu Aufgabe 5

(a) Den folgenden Baum erhält man, indem man Merge-Sort Schritt für Schritt durchspielt und immer dann einen neuen Knoten hinzufügt, wenn irgendwo im Algorithmus eine Vergleichsoperation durchgeführt wird. Je nach Ergebnis des Vergleichs gibt es dann zwei mögliche Wege (hier immer rechts=yes, links=no), wie der Algorithmus bis zum nächsten Vergleich fortfährt, bis er letztlich irgendeine Anordnung der Elemente ausgibt. Anstelle der reinen Ausgabe des Algorithmus (z.B. $c\,a\,b$) ist in dieser Lösung zusätzlich das erlangte Wissen um deren (strikte) Ordnung eingezeichnet (z.B. $c < a \leq b$).



(b) Jede mögliche Ordnung der Eingabeelemente muss vom Algorithmus korrekt behandelt werden und somit irgendwo als Blatt auftauchen. Da n Elemente auf $n!$ verschiedene Weisen angeordnet werden können, erhalten wir $4! = 24$ Blätter für $[a, b, c, d]$ und $26! \approx 4 \cdot 10^{26}$ Blätter für $[a, b, \ldots, y, z]$.

Lösungen zu Aufgabe 6
Wir führen folgenden Algorithmus auf Array $A = (A[1], \ldots, A[n])$ für ein $k < n$ aus:

```
1:  function GetSortedMins(A, k)
2:      H ← CreateMaxHeap                          ▷ Erstelle neuen leeren Max-Heap
3:      for i = 1...k do
4:          Insert(H, A[i])                        ▷ Füge die ersten k Elemente in den Heap ein
5:      end for
6:      for i = k+1...n do
7:          if A[i] < ReadMax(H) then
8:              Decrease(H, root, A[i])            ▷ Verringere den Wurzelknotenwert auf A[i]
9:          end if
10:     end for
11:     M ← CreateArray(k)                         ▷ Erstelle neues leeres Array der Größe k
12:     for i = k...1 step −1 do
13:         M[i] ← ExtractMax(H)                   ▷ Extrahiere das Maximum der verbleibenden Werte
14:     end for
15:     return M
16: end function
```

(1) Die Zeilen 2-5 bauen einen neuen separaten Max-Heap aus den ersten k Elementen von A. Dies dauert Zeit $\mathcal{O}(k \log k)$.

(2) Die Zeilen 6-10 parsen den Rest des Arrays und fügen immer dann ein Element in den Max-Heap ein, wenn es kleiner als dessen bisheriges Maximum ist. Dies dauert Zeit $\mathcal{O}(n \log k)$. Offensichtlich befinden sich abschließend die k kleinsten Elemente von A im Max-Heap.

(3) Die Zeilen 11-14 lesen den MaxHeap komplett aus und schreiben seine Elemente aufsteigend sortiert in das Ausgabearray. Dies dauert Zeit $\mathcal{O}(k \log k)$.

Insgesamt erhalten wir also Laufzeit $\mathcal{O}(k \log k + n \log k + k \log k) = \mathcal{O}(n \log k)$.