Prof. Dr. Ir. Dr. h. c. Joost-Pieter Katoen   Stefan Dollase, Ira Fesefeldt, Tim Quatmann, Jip Spel, Tobias Winkler

Übung 4

Hinweise:

- Die Übungsblätter sollen in 3er Gruppen bearbeitet werden.
- Die Bearbeitung der Übungsblätter ist nicht zwingend notwendig um die Klausurzulassung zu erhalten. Es werden zwar Punkte vergeben, diese dienen jedoch nur zur eignen Einschätzung.
- Zur Vorbereitung auf Präsenzübung und Klausur empfehlen wir trotzdem alle Übungsblätter semesterbegleitend zu bearbeiten. Besonders relevant hierfür sind Aufgaben, die mit einem ★ markiert sind.
- Die Lösung des Übungsblattes wird in RWTHmoodle veröffentlicht. Solange die Tutorien nicht stattfinden können, werden außerdem Videos zu jeder Aufgabe angefertigt.
- Wir haben in RWTHmoodle ein Forum eingerichtet, welches als erste Anlaufstelle für **Fragen** dienen soll.
- Relevante Vorlesungen für dieses Übungsblatt: 1, 2, 3, 4, 5

Aufgabe 1 (Analyse eines rekursiven Algorithmus): 3+3+8+3+8 = 25 Punkte

In dieser Aufgabe soll der folgende rekursive Algorithmus analysiert werden:

```
int func_main(List input) {
    return func(input, 0, input.length())
}

int func(List input, int i, int n) {
    int p = input.first()
    List left, List right = partition(input, p) // Laufzeit: input.length()
    int k = left.length() + i
    if k = ⌊ n / 2 ⌋:
        return p
    else if k < ⌊ n / 2 ⌋:
        return func(right, k+1, n) // rekursiver Aufruf
    else k > ⌊ n / 2 ⌋:
        return func(left, i, n) // rekursiver Aufruf
}
```

Treffen Sie dazu folgende Annahmen:

- Listen enthalten nur Integer (ganze Zahlen).
- `list.first()` gibt das erste Element (Index 0) einer nicht-leeren Liste zurück.
- Die Funktion `partition(list, p)` erhält als Eingabe eine Liste `list` und eine Zahl `p`, die in `list` enthalten sein muss. Die Funktion entfernt zunächst `p` aus der Liste und gibt dann zwei neue Listen `left` und `right` zurück, sodass `left` alle verbliebenen Elemente enthält, die kleiner oder gleich `p` sind und `right` alle anderen Elemente. Die Reihenfolge der Elemente wird dabei beibehalten. Sollte `p` mehrmals vorkommen, so wird dasjenige mit dem kleinsten Index entfernt. Zum Beispiel erzeugt der Aufruf
  - `partition([3,1,7,1,4,5,4], 4)` die Listen `left = [3,1,1,4]` und `right = [7,5]`
  - `partition([6,6,6], 6)` die Listen `left = [6,6]` und `right = []`

  Gehen Sie davon aus, dass `partition` in jedem Fall exakt Laufzeit n, also die Länge der Eingabeliste hat.

a) Bestimmen Sie die Ergebnisse von

(i) `func_main([8,3,1,3,6,2,0])` und

(ii) `func_main([4,3,1,6,9,5,0,8,7,2])`.

Geben Sie dabei auch Zwischenschritte mit an.

b) Beschreiben Sie in Worten ohne Beweis, was `func_main` auf einer gegebenen Eingabeliste berechnet.

c) Zählen Sie bei allen folgenden Laufzeit-Analysen in dieser Aufgabe nur den Zeitaufwand der Aufrufe von `partition`. Bestimmen Sie:

- die asymptotische Worst-Case Laufzeit von `func_main`, d.h. eine Funktion g, sodass $W(n) \in \Theta(g(n))$.
- die asymptotische Best-Case Laufzeit von `func_main`, d.h. eine Funktion g, sodass $B(n) \in \Theta(g(n))$.

d) „Bei geeigneten (realistischen) Annahmen über die Eingabeverteilung gilt für die asymptotische Average-Case Laufzeit $A(n)$ von `func_main` die Rekursionsgleichung $A(n) = A(n/2) + n$ für $n > 1$“.

Begründen Sie die Plausibilität dieser Aussage (ein formaler Beweis ist nicht nötig).

e) Lösen Sie die Rekursionsgleichung aus Teil d) indem Sie eine möglichst genaue asymptotische obere Schranke angeben (also $A(n) \in O(...)$). Gehen Sie davon aus, dass das Funktionsargument immer auf eine ganze Zahl abgerundet wird und nehmen Sie als Startwert $A(1) = 1$. Vergleichen Sie die Lösung mit den Ergebnissen aus Teil c).

Aufgabe 2 (Lösen von Rekursionsgleichungen): 8+10+8 = 26 Punkte

Wenden Sie die Substitutionsmethode auf folgende Gleichungen an.

★a) Raten Sie, durch wiederholtes Einsetzen, eine möglichst geringe obere Schranke für die folgende Rekursionsgleichung und zeigen Sie deren Korrektheit.

$$T(n) = \begin{cases} 4 \cdot T(\frac{n}{2}) + n^2, & \text{falls } n > 0 \\ 14, & \text{sonst} \end{cases}$$

★b) Raten Sie, mithilfe eines Rekursionsbaums, eine möglichst geringe obere Schranke für die folgende Rekursionsgleichung und zeigen Sie deren Korrektheit.

$$T(n) = \begin{cases} 2 \cdot T(\frac{n}{2}) + \sqrt{n}, & \text{falls } n > 0 \\ 1, & \text{sonst} \end{cases}$$

c) Eine Funktion $T\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ wächst *mindestent doppelt exponentiell*, falls es reelle Zahlen $a > 1$, $b > 1$ gibt, sodass $T(n) \in \Omega(a^{b^n})$. Zeigen Sie: Die Funktion $T(n)$, welche durch die Rekursionsgleichung

$$T(n) = \begin{cases} T(n-1)^2 + 1, & \text{falls } n > 0 \\ 2, & \text{falls } n = 0 \end{cases}$$

definiert ist, wächst mindestens doppelt exponentiell.

Aufgabe 3 (Korrektheitsbeweis Prominentensuche Problem): 17+7 = 24 Punkte

Eine prominente Person ist eine Person die niemanden kennt aber von allen gekannt wird (siehe Vorlesung). Betrachten Sie folgenden Algorithmus für die Prominentensuche:
Eingabe:

- $n \in \mathbb{N}$ Personen nummeriert $0, \ldots, n-1$
- Mindestens eine Person ist prominent
- $n \times n$ boolean Matrix K, sodass für alle $0 \leq i, j < n$ mit $i \neq j$ gilt: Person i kennt Person j gdw. $K[i,j] =$ `true`

Ausgabe: Nummer einer prominenten Person.

```
int celebritySearch(boolean[,] K, int n) {
  int min = 0, max = n - 1;
  while (min != max) {
    if (K[min, max]) {
      min = min + 1;
    } else {
      max = max - 1;
    }
  }
  return min;
}
```

a) Beweisen Sie die Korrektheit des gegebenen Algorithmus.

b) Beweisen Sie die Terminierung des gegebenen Algorithmus.

Aufgabe 4 (Suchen in “fast sortierten” Arrays): 9+4+12 = 25 Punkte

★a) Ein (1-basiert indiziertes) Array $a[1], \dots, a[n]$ von ganzen Zahlen der Länge $n \geq 3$ ist in *Up-Down-Ordnung* falls es einen (unbekannten) Index $1 < k < n$ gibt, sodass $a[1] < a[2] < \dots a[k]$ und $a[k] > a[k+1] > \dots a[n]$ gilt. So sind zum Beispiel die Arrays

- $[1, 2, 3, 2, 1]$ und
- $[0, 5, 4, 2]$

jeweils in Up-Down-Ordnung. Beschreiben Sie einen Algorithmus, der das Maximum eines Up-Down-geordneten Arrays in Worst-Case Zeit $W(n) \in O(\log(n))$ bestimmt. Zählen Sie nur für Array-Zugriffe eine Zeiteinheit und vernachlässigen Sie alle anderen Operationen. Sollte die Eingabe nicht Up-Down-geordnet sein, so darf Ihr Algorithmus sich beliebig verhalten.

★b) Geben Sie nun einen möglichst effizienten Algorithmus an, der für ein gegebenes Up-Down-sortiertes Array a und eine ganze Zahl z bestimmt, ob z in a enthalten ist. Sie brauchen wieder nur Array-Zugriffe zu zählen.

c) Ein Array $a[1], \dots, a[n]$ von ganzen Zahlen der Länge $n \geq 4$ ist in *Up-Down-Up-Ordnung* falls es Indizes $1 < k < \ell < n$ gibt, sodass $a[1] < a[2] < \dots a[k]$, $a[k] > a[k+1] > \dots a[\ell]$ und $a[\ell] < a[\ell+1] < \dots a[n]$ gilt. Sei $\mathcal{A}$ ein beliebiger *deterministischer* Algorithmus, der als Eingabe ein Array a von ganzen Zahlen erhält und folgendes berechnet

$$\mathcal{A}(a) = \begin{cases} \text{Das Maximum von } a, & \text{falls } a \text{ Up-Down-Up-geordnet ist} \\ \text{Eine beliebiege ganze Zahl,} & \text{sonst.} \end{cases}$$

Die beliebige ganze Zahl kann für jede nicht Up-Down-Up-geordnete Eingabe verschieden sein. Zeigen Sie: Für die Worst-Case Laufzeit (nur Array-Zugriffe) von $\mathcal{A}$ gilt $W(n) \in \Omega(n)$.

Hinweise:

- Ein deterministischer Algorithmus liefert auf gleicher Eingabe immer die gleiche Ausgabe.
- Es bietet sich ein Widerspruchsbeweis an.