Algorithmen und
Berechnungskomplexität I
Wintersemester 2019/20

Abgabe: 7.11.2018 bis 12.15 Uhr
Besprechung: KW 46

Prof. Dr. T. Kesselheim
A. Braun, D. Kübel, L. Oettershagen
Institut für Informatik

Übungsblatt 4

Hinweis: Achten Sie bitte stets darauf, Ihren Pseudocode ausreichend zu kommentieren!

Aufgabe 4.1: Quicksort (5 Punkte)

Rekursive Funktionen können iterativ implementiert werden, wenn man den Rekursionsstack selbst verwaltet. Da bei *Quicksort* alle rekursiven Aufrufe am Ende des Funktionsrumpfes erfolgen, können wir weitere Einsparungen vornehmen. Dazu stellen wir uns vor, dass auf dem Stack Indexpaare (ℓ, r) liegen, die die Teilfelder beschreiben, die noch sortiert werden müssen. Diese können wir uns als Jobs vorstellen. Um einen Job (ℓ, r) abzuarbeiten, führen wir PARTITION auf dem entsprechenden Teilfeld aus und müssen anschließend zwei Jobs $(\ell, q-1)$ und $(q+1, r)$ abarbeiten. Bevor wir diese Paare jedoch auf den Stack legen, können wir aufgrund der Endrekursivität bereits den Job (ℓ, r) vom Stack entfernen. Trotz dieser Ersparnis hat die Reihenfolge, in der wir die Jobs $(\ell, q-1)$ und $(q+1, r)$ auf den Stack legen, einen großen Einfluss auf die maximale Stackgröße.

(a) Als Grundlage verwenden wir die PARTITION-Funktion aus der Vorlesung. Geben Sie eine Klasse von Instanzen und eine Strategie, die Jobs auf den Stack zu legen, an, bei der die maximale Stackgröße linear in n wächst.

(b) Geben Sie eine Strategie an, bei der die maximale Stackgröße durch $O(\log n)$ beschränkt ist, unabhängig davon, welche PARTITION-Funktion verwendet wird und auf welcher Instanz *Quicksort* ausgeführt wird. Beweisen Sie, dass Ihre Strategie die Anforderungen erfüllt.

Aufgabe 4.2: Inversionen in Arrays (5 Punkte)

Sei A ein Feld mit n Einträgen $A[1], \ldots, A[n]$. Die Anzahl $\chi(A)$ der *Inversionen* oder *Fehlstellungen* von A ist definiert als

$$\chi(A) = |\{(i,j) : 1 \leq i < j \leq n \text{ und } A[i] > A[j]\}|.$$

Geben Sie einen Algorithmus an, der für ein Feld A mit n Einträgen die Anzahl $\chi(A)$ der Inversionen in Zeit $O(n \log n)$ bestimmt.

Aufgabe 4.3: Greedy-Algorithmus für ein Bin-Packing-Problem (5 Punkte)

Ein Unternehmen will n Objekte unterschiedlicher Größen befördern lassen. Dazu kann das Unternehmen beliebig viele Seefracht-Container nutzen, die alle dasselbe Volumen V haben. Die Volumina der Objekte seien durch $v_1, \ldots, v_n$ gegeben und es gelte $v_i \leq V$ für alle $i = 1, \ldots, n$. Im Kontext dieser Aufgabe werden die genauen Abmessungen der Objekte vernachlässigt; d.h. ein Objekt passt in einen Container, wenn das Restvolumen des Containers größer oder gleich dem Volumen des Objektes ist.

Um Transportkosten einzusparen, ist das Unternehmen an einer Aufteilung der Objekte auf möglichst wenige Container interessiert.

(a) Beschreiben Sie einen Greedy-Algorithmus durch Pseudocode, der die Objekte folgendermaßen auf Container aufteilt: In jedem Schritt wird das Objekt, das unter den verbliebenen Objekten das größte Volumen hat, in den ersten passenden Container platziert (d.h., bereits angebrochene Container werden stets in derselben Reihenfolge betrachtet). Sie dürfen zum Sortieren der Eingabe die Methode *sortiere*(A) verwenden, die ein gegebenes Array absteigend sortiert.

(b) Vollziehen Sie Ihren Algorithmus anhand des folgenden Beispiels nach: Die Objekte mit den Volumina $\vec{v} = (2, 3, 6, 1, 4, 2, 5, 2)^T$ sollen in Container mit Volumen $V = 6$ einsortiert werden. Begründen Sie, warum die durch Ihren Algorithmus gefundene Lösung in diesem Beispiel optimal ist.

(c) Zeigen Sie anhand eines Gegenbeispiels, dass der Algorithmus nicht für jede Eingabe eine optimale Lösung liefert. Geben Sie einerseits eine optimale Lösung für das beschriebene Beispiel an und andererseits die von ihrem Algorithmus berechnete Lösung.

Aufgabe 4.4: Dynamische Programmierung (5 Punkte)

Gegeben seien zwei verschiedene Typen von *Spielsteinen* mit Maßen bezogen auf ein beliebiges Standardmaß:

(I) Rechteckige Spielsteine der Größe 2×1 sowie

(II) L-förmige Spielsteine der Größe 2×2 mit einer Aussparung von einem 1×1-Feld

Spielsteine dürfen im Kontext der Aufgabe beliebig gedreht oder gespiegelt werden. Die beiden Spielsteintypen werden nachfolgend abgebildet.

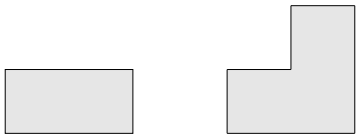

Abbildung 1: Typ (I) ist links und Typ (II) rechts abgebildet.

Eine *Pflasterung* eines Rechteckes mit den Maßen $m \times n$ ist eine Anordnung von Spielsteinen, sodass das Rechteck vollständig durch die Spielsteine überdeckt wird und zusätzlich kein Stein über das Rechteck hinausragt. Zwei Pflasterungen, die nicht deckungsgleich sind, werden als verschieden angesehen. Außerdem werden zwei Pflasterungen, bei denen sich die eine durch Spiegelung bzw. Rotation aus der anderen ergibt, als verschieden angesehen. Wir sind an der Anzahl verschiedener Pflasterungen für $n \times 2$-Rechtecke für $n \in \mathbb{N}$ interessiert.

(a) Finden Sie eine rekursive Form, um die Anzahl verschiedener Pflasterungen für ein $n \times 2$-Rechteck zu bestimmen, falls lediglich Spielsteine vom Typ (I) genutzt werden dürfen.

Tipp: Überlegen Sie sich zunächst, welche Möglichkeiten es gibt, den ersten Spielstein an den linken Rand des Rechtecks zu legen, welche Konsequenzen dieses Auslegen hat und wie Sie dadurch die Pflasterung auf jeweils kleinere Pflasterungen zurückführen können.

(b) Finden Sie nun eine rekursive Form, um die Anzahl verschiedener Pflasterungen für ein $n \times 2$-Rechteck zu bestimmen, falls Spielsteine vom Typ (I) und (II) genutzt werden dürfen.

Tipp: Nutzen Sie die folgenden Funktionen f, g und h, um die rekursive Form aufzustellen:

- $f(n)$: Anzahl verschiedener Pflasterungen für ein $n \times 2$-Rechteck
- $g(n)$: Anzahl verschiedener Pflasterungen für ein $n \times 2$-Rechteck mit einem zusätzlichen Feld in der unteren Reihe
- $h(n)$: Anzahl verschiedener Pflasterungen für ein $n \times 2$-Rechteck mit einem zusätzlichen Feld in der oberen Reihe

(c) Entwickeln Sie mithilfe von Teilaufgabe (b) einen Algorithmus, der auf dynamischer Programmierung beruht und für ein gegebenes $n \times 2$-Rechteck die Anzahl verschiedener Pflasterungen berechnet.

Hinweis: Falls Sie Teilaufgabe (b) nicht bearbeitet haben, nutzen Sie Ihr Ergebnis aus Teilaufgabe (a).

(d) Führen Sie nun eine asymptotische Worst-Case-Laufzeitanalyse für Ihren Algorithmus durch.

(e) Implementieren Sie ihren Algorithmus in einer Ihnen vertrauten Programmiersprache und berechnen Sie damit die Anzahl verschiedener Pflasterungen für $n = 1, \ldots, 20$. Für Ihre Lösungsabgabe genügt es, lediglich die Ausgabe der Programms einzureichen.

Aufgabe 4.5: RadixSort (4 Zusatzpunkte)

Wir wollen die Laufzeiten von *Quicksort*, *Mergesort* und einer optimierten Variante von *Radixsort* experimentell miteinander vergleichen. Erzeugen Sie dazu für $n = 2, \ldots, 500$ n-mal hintereinander ein Feld mit n zufälligen Einträgen aus der Menge $\{0, \ldots, 2^{32} - 1\}$ und führen Sie *Quicksort*, *Mergesort* und *Radixsort* auf diesem Feld aus. Bestimmen Sie für jedes n die durchschnittliche Laufzeit der drei Sortierverfahren und stellen Sie diese als Graphen in Abhängigkeit von n dar. Verwenden sie dabei für *Quicksort* die PARTITION-Funktion aus der Vorlesung und für *Radixsort* die folgende Variante: Zerlegen Sie die Binärdarstellung jeder der n 32-Bit-Zahlen von rechts nach links in Blöcke der Länge $r := \lceil \log_2(n) \rceil$ und interpretieren Sie diese Blöcke als Zahlen zwischen 0 und $M - 1$ mit $M := 2^r$, auf die Sie MSORT anwenden.