Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Übungsblatt 04

Algorithmen I - Sommersemester 2024

Abgabe im ILIAS bis 17.05.2024, 18:00 Uhr

Die Abgabe erfolgt alleine oder zu zweit als eine PDF-Datei über das Übungsmodul in der Gruppe eures Tutoriums im ILIAS. Beschriftet die Abgabe deutlich mit Matrikelnummer und Name.

- Achtet bei handschriftlichen Abgaben auf Lesbarkeit.
- Achtet darauf, ob Algorithmen in Worten oder Pseudocode beschrieben werden sollen.
- Es werden immer asymptotisch möglichst effiziente Algorithmen erwartet, wenn nicht anders angegeben.
- Wenn Korrektheits- oder Laufzeitanalysen gefordert sind, behandelt diese separat von der Algorithmenbeschreibung.

Gesamtpunkte: 20 (+ 2 Bonus)

Aufgabe 1 - Von beiden Enten her (5 Punkte)

Die Biberklonaufsicht (BKA) hat mittlerweile Wind von Dr. Metas geklonten Bibern bekommen. Um Dr. Meta aufzuhalten möchte die BKA herausfinden was Dr. Meta plant. Durch Tarnung zweier Agenten als Mitarbiber in Mr. Metas Geheimlabor, konnte die BKA zwei brisante Dokumente entwenden. Das erste Dokument scheint die Operationen einer Datenstruktur zu beschreiben, während das zweite die Umsetzung derselben durch drei Stacks darstellt. Der Inhalt der beiden Dokumente ist unten angegeben:

Ein **Quack** ist eine Datenstruktur, welche die Eigenschaften von Stacks und Queues vereint. Ein Quack repräsentiert eine Folge von Elementen, die von links nach rechts betrachtet werden.

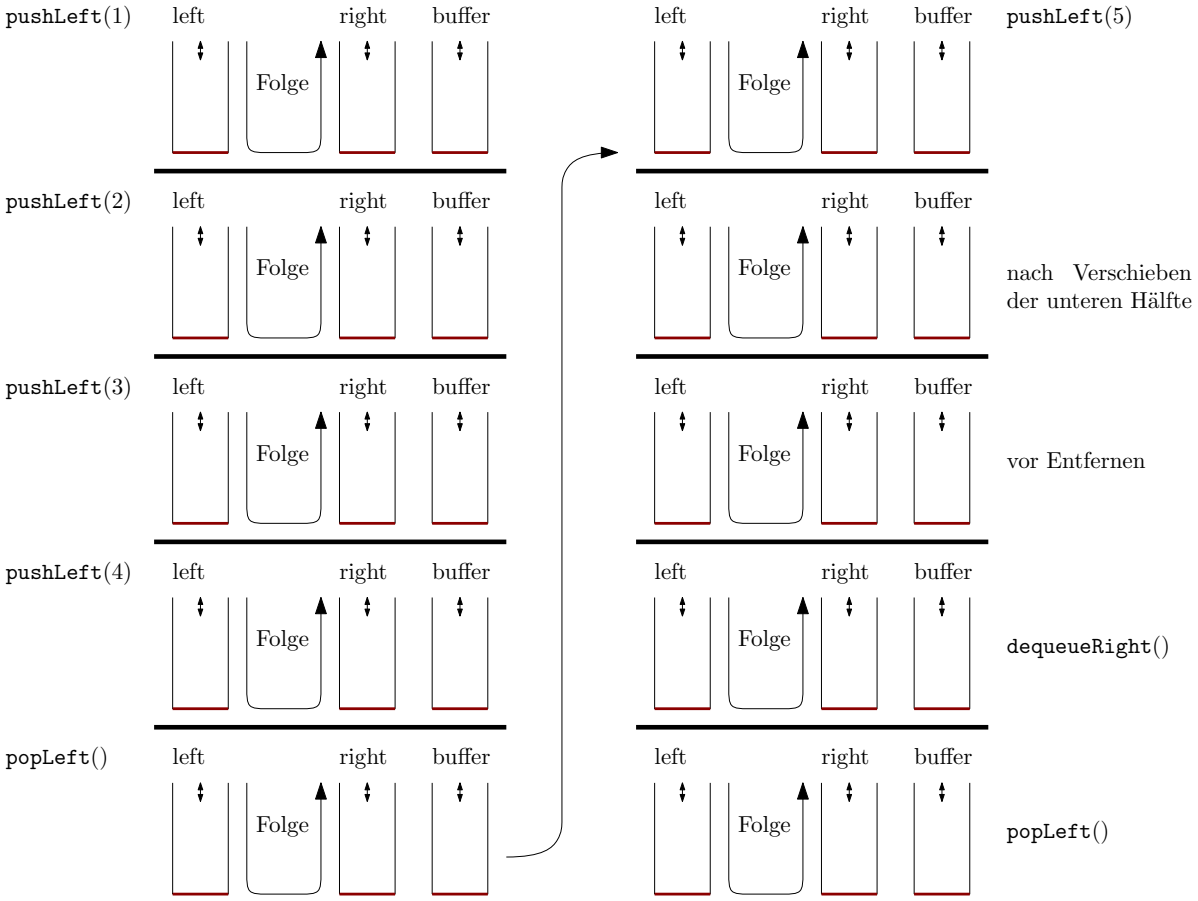
- `pushLeft`(x): Fügt das Element x am linken Ende der Folge ein.
- `popLeft`(): Entfernt das Element am linken Ende der Folge und gibt es zurück.
- `dequeueRight`(): Entfernt das Element am rechten Ende der Folge und gibt es zurück.

Es kann davon ausgegangen werden, dass `popLeft` und `dequeueRight` werden nur aufgerufen, wenn die vom Quack repräsentierte Folge nicht leer ist.

Bauanleitung für einen **Quack**:
Nutze Stacks *left*, *right* und *buffer*. Hierbei repräsentiert *left* den linken Teil der Folge und *right* den rechten. Dabei sind die Elemente von *right* in gespiegelter Reihenfolge angeordnet, so dass das rechteste Element oben auf dem Stack *right* liegt. Jedes Element kommt zu jedem Zeitpunkt nur in einem der drei Stacks vor. Somit können `pushLeft` und `popLeft` am linken Ende der Folge arbeiten, indem sie *left* nutzen. Analog arbeitet `dequeueRight` auf *right* und am rechten Ende. Falls bei `popLeft` oder `dequeueRight` der entsprechende Stack leer ist obwohl die repräsentierte Folge nicht leer ist, wird vor dem Entfernen die Last ausgeglichen. Bei einem Lastausgleich werden abgerundet die Hälfte der Elemente vom noch vollen Stack auf *buffer* verschoben. Anschließend werden die verbleibenden Elemente auf den leeren Stack und die Elemente von *buffer* zurück an ihren Ursprungsplatz verschoben. Nun speichern *left* und *right* je die Hälfte der Elemente. Beachte, dass jedes Verschieben der Elemenente ihre Reihenfolge umkehrt und daher die obigen Vorgaben eingehalten werden.

Das BKA vermutet, dass Dr. Meta einen Beweis dafür hat, dass die amortisierte Laufzeit aller Operationen konstant ist.

a) Hilf der BKA dabei die Datenstruktur zu verstehen. Führe dafür die Operationssequenz `pushLeft`(1), `pushLeft`(2), `pushLeft`(3), `pushLeft`(4), `popLeft`(), `pushLeft`(5), `dequeueRight`(), `popLeft`() auf einem leeren Quack aus und gebe den Zustand der Datenstruktur nach jeder Operation an. Wenn es zu einem Lastausgleich kommt, gebe außerdem die Zustände nach Verschieben der unteren Hälfte der Elemente und direkt vor dem Entfernen des Elementes an. Du kannst deine Ergebnisse in die untenstehende Abbildung eintragen. (2 Punkte)



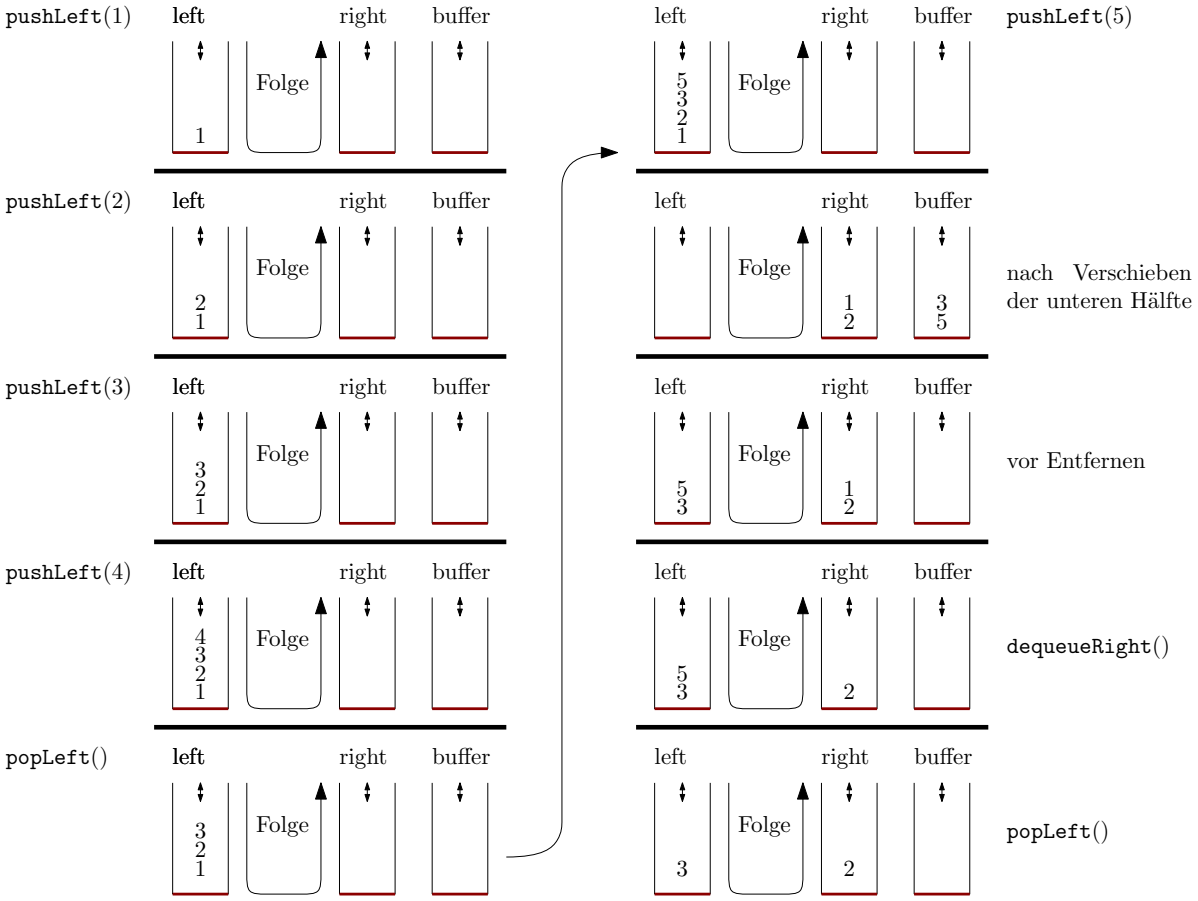
Darstellung eines leeren Quacks, inklusive freie Felder für Zwischenzustände. Neue Elemente werden stets von oben (kleine Pfeile) auf den Stack gelegt und stapeln sich oberhalb der roten Kante. Zusätzlich ist die Sortierung der Folge über die beiden Stacks hinweg visualisiert.

b) Hilf der BKA den Beweis zu rekonstruieren. Zeige hierfür, dass die Laufzeit für eine beliebige Folge von Operationen auf einem zu Beginn leeren Quack bei $\Theta(1)$ pro Operation liegt. Nutze dafür eine Methode deiner Wahl. (3 Punkte)

Hinweis: Du darfst annehmen, dass die Kosten von `push` und `pop` auf den zugrunde liegenden **Stacks** die Kosten von genau 1 haben.

Lösung 1

a) Die untenstehende Grafik zeigt die Lösung.



Lösung zur Teilaufgabe a).

b) Im folgenden geben wir Lösungen mit Konto-, Charging-, und Potentialmethode an.

Wir beginnen zunächst mit allgemeinen Beobachtungen. Die Kosten für `pushLeft` sind unabhängig vom Zustand des Quack immer konstant. Bei `popLeft` sowie `dequeueRight` sind die Kosten ebenfalls konstant, falls kein Lastausgleich erfolgen muss. Andernfalls hängen die Kosten davon ab, wie viele Elemente vor dem Lastausgleich auf dem vollen Stack liegen. Sei diese Zahl im folgenden n_{voll}. Während dem Lastausgleich werden $\lfloor n_{\text{voll}}/2 \rfloor$ Elemente auf den Buffer geschoben, $\lceil n_{\text{voll}}/2 \rceil$ Elemente auf den vorher leeren Stapel geschoben und anschließend alle Elemente von Buffer zurück auf den vorher vollen Stapel geschoben. Insgesamt werden also weniger als $4 \cdot n_{\text{voll}}$ Einfüge und Entfernen Operationen auf den Stacks ausgeführt.

- **Kontomethode.** Wir lassen die Operation `pushLeft` immer 5 Einheiten auf das Konto einzahlen. Ebenso zahlen `popLeft` und `dequeueRight` 5 Einheiten auf das Konto ein, falls kein Lastausgleich erfolgt. Falls ein Lastausgleich mit $n_{\text{voll}} \leq 5$ erfolgt, zahlen `popLeft` und `dequeueRight` diese Kosten direkt und nutzen das Konto nicht. Im Falle eines teuren Lastausgleichs mit $n_{\text{voll}} > 5$ heben `popLeft` und `dequeueRight` $4 \cdot n_{\text{voll}}$ Guthaben vom Konto ab, wobei

n_{voll} die Anzahl Elemente auf dem nicht-leeren Stack ist. Jede Operation hat also entweder konstante tatsächliche Kosten und zahlt einen konstanten Betrag aufs Konto ein, oder sie gleicht ihre nicht-konstanten Kosten über das Konto aus.

Es verbleibt zu zeigen, dass das Konto nie negativ wird. Hierfür zeigen wir, dass nach jedem Lastausgleich das Konto nicht negativ ist. Wir betrachten zunächst den ersten Lastausgleich. Hier wird `dequeueRight` ausgeführt obwohl der rechte Stack leer ist, während der linke Stack n_{voll} Elemente enthält.[1] Dann müssen vorher mindestens n_{voll} Aufrufe von `pushLeft` erfolgt sein und somit ein Konstostand von mindestenz $4 \cdot n_{\text{voll}}$ vorliegen. Folglich ist das Konto nach dem ersten Lastausgleich nicht negativ.

Wir betrachten nun den i-ten Lastausgleich unter Annahme, dass nach dem $i-1$-ten Lastausgleich das Konto nicht negativ war. Nun verhalten sich `popLeft` und `dequeueRight` symmetrisch und wir nehmen ohne Beschränkung der Allgemeinheit an, dass der rechte Stack leer ist und der i-te Lastausgleich wegen `dequeueRight` erfolgt. Nach jedem Lastausgleich unterscheidet sich die Anzahl Elemente auf *left* und *right* nur um 1. Sei daher n_l und n_r mit $|n_r - n_\ell| \leq 1$ die Anzahl Elemente des linken und rechten Stacks nach dem $i-1$-ten Lastausgleich. Sei weiterhin n_{voll} die Anzahl Elemente des vollen linken Stacks unmittelbar vor dem i-ten Lastausgleich. Seit dem $i-1$-ten Lastausgleich müssen mindestenz n_r `dequeueRight` Operationen aufgerufen worden sein und diese haben $5n_r$ Guthaben aufs Konto eingezahlt. Falls $n_{\text{voll}} \leq n_r$, reicht das Guthaben also um $4n_{\text{voll}}$ abzuheben ohne negativ zu werden. Falls $n_{\text{voll}} > n_r$, wurden mindestenz $n_{\text{voll}} - n_\ell$ `pushLeft` Operationen aufgerufen. Insgesamt ergibt das mindestenz $n_{\text{voll}} - 1$ Einzahlungen von jeweils 5 Guthaben. Für $n_{\text{voll}} > 5$ gilt dann $4n_{\text{voll}} \leq 5(n_{\text{voll}} - 1)$ und somit ist genügend Guthaben vorhanden um die Abbuchung für den Lastausgleich zu bezahlen. Falls andernfalls $n_{\text{voll}} \leq 5$, sind die Kosten des Lastausgleichs selbst konstant und das Konto wird nicht belastet. Es folgt also dass auch nach dem i-te Lastausgleich das Konto nicht negativ bleibt.

- **Charging.** Tritt bei `popLeft` oder `dequeueRight` ein Lastausgleich auf, teilen wir die Token der Operation gleichmäßig auf die günstigen vorhergegangenen `popLeft`, `dequeueRight`, sowie `pushLeft` Operationen auf, welche seit dem letzten Lastausgleich erfolgt sind. Falls es noch keinen vorhergehenden Lastausgleich gab, dann werden die Token auf alle vorherigen Operationen aufgeteilt. Es ist klar, dass jede günstige Operation nur von einer einzigen teuren Operation zusätzliche Token bekommt. Es bleibt also zu zeigen, dass die Kosten des Lastausgleichs auf genügend günstige Operationen aufgeteilt werden, sodass keine Operation zu viele Token bekommt.

  Wie zuvor sei n_{voll} die Anzahl Elemente auf dem nicht-leeren Stapel, womit

[1] Der erste Lastausgleich kann nicht bei `popLeft` auftreten, da der rechte Stack zu Beginn leer ist und ein leerer linker Stack somit eine leere Datenstruktur impliziert, auf den kein Lastausgleich erfolgt.

die Kosten des Lastausgleichs höchstens $4 \cdot n_{\text{voll}}$ sind. Falls noch kein anderer Lastausgleich statt fand, müssen also mindestens n_{voll} `pushLeft` Operationen ausgeführt worden sein. Da die Kosten gleichmäßig aufgeteilt werden, bekommt jede dieser `pushLeft` Operationen also 4 zusätzliche Token.

Wir betrachten nun den i-ten Lastausgleich für $i > 1$. Wir nehmen o.B.d.A. an, dass nun der rechte Stapel leer ist und der Lastausgleich durch ein `dequeueRight` ausgelöst wurde. Sei n_ℓ und n_r die Anzahl Elemente auf *left* und *right* direkt nach dem $i-1$-ten Lastausgleich. Dann können wir erneut argumentieren, dass seit dem mindestens n_r `dequeueRight` Operationen und $n_{\text{voll}} - n_\ell$ `pushLeft` Operationen aufgerufen wurden. Insgesamt ergibt das mindestens $n_{\text{voll}} - 1$ Operationen, auf die noch keine Token geschoben wurde. Somit können wir bis zu vier Token auf jede der betrachteten Operationen verschieben und 4 Token auf der Operation mit dem Lastausgleich behalten. Damit verbleiben bei jeder Operation nur konstant viele Token, womit sie also jeweils amortisiert konstante Laufzeit hat.

- **Potentialmethode.** Für Quack Q bezeichen wir mit Q_{vor} den Zustand vor und mit Q_{nach} den Zustand nach einer Operation. Wir nutzen außerdem $a = left.length$ und $b = right.length$.
  Wir nutzen die Potentialfunktion $\Phi(M) = 4 \cdot |a - b|$. Durch den Betrag ist das Potential trivialerweise nie negativ. Damit zeigen wir, dass die amortisierten Kosten jeder Operation konstant sind, indem wir die tatsächlichen Kosten einer Operation zusammen mit ihrer Potentialänderung $\Delta\Phi = \Phi(Q_{\text{nach}}) - \Phi(Q_{\text{vor}})$ betrachten.
  - `pushLeft` hat tatsächliche Kosten von 1. Wir legen ein Element auf *left* und haben damit eine Potentialänderung von $\Phi(Q_{\text{nach}}) - \Phi(Q_{\text{vor}}) = 4 \cdot |a + 1 - b| - 4 \cdot |a - b| = \pm 4$. Somit liegen die amortisierten Kosten bei $1 \pm 4 \in \Theta(1)$.

  - `popLeft` und `dequeueRight` haben tatsächliche Kosten von 1 ohne Lastausgleich und $4 \cdot a$ bzw. $4 \cdot b$ mit Lastausgleich. Wenn kein Lastausgleich statt findet sind die Kosten analog zu `push` konstant. Findet ein Lastausgleich statt, dann gilt vor der Operation $a = 0$ oder $b = 0$. Sei o.B.d.A. $b = 0$. Dann verschieben wir effektiv $\frac{a}{2}$ Elemente von *left* nach *right* und entfernen eines. Dadurch haben wir eine Potentialänderung von $\Phi(Q_{\text{nach}}) - \Phi(Q_{\text{vor}}) = 4 \cdot |\frac{a}{2} - (\frac{a}{2} - 1)| - 4 \cdot |a - 0| = 4 - 4 \cdot a$. Somit liegen die amortisierten Kosten bei $4 \cdot a + 4 - 4 \cdot a = 4 \in \Theta(1)$. Wenn die Anzahl der Elemente nicht durch zwei teilbar ist, dann ist die Potentialänderung um vier kleiner und somit sind die Kosten um vier größer und weiterhin konstant.

Aufgabe 2 - I did it ~~my~~ k-way (9 Punkte)

Du hast in der Vorlesung den Algorithmus MERGESORT kennen gelernt, der die Methode MERGE nutzt. Wir wollen nun eine Variante des Algorithmus entwickeln, die das Array in jedem Schritt in mehr als zwei Teile zerteilt.

a) Gib Pseudocode für eine Variante der bekannten MERGE-Methode an, die drei sortierte Arrays als Eingabe erhält und ein sortiertes Array, dass die selben Elemente enthält, zurückgibt.
Gib außerdem die Laufzeit deines Algorithmus in Abhängigkeit von seinen Eingabeparametern an und begründe diese. Verwende dazu nicht die MERGE-Methode aus der Vorlesung als Subroutine.
(3 Punkte)

b) Gib ebenfalls Pseudocode für den Sortieralgorithmus 3-WAY-MERGESORT an, der analog zu MERGESORT funktioniert, aber das Array in drei Abschnitte unterteilt. Welche Laufzeit hat dein Algorithmus (mit Begründung)? (1.5 Punkte)

c) Wir wollen nun für ein festes k in jedem Schritt k sortierte Arrays betrachten. Beschreibe in Worten wie die Algorithmen k-WAY-MERGE und k-WAY-MERGESORT umgesetzt werden können. Gib außerdem die Laufzeit von k-WAY-MERGE und k-WAY-MERGESORT in Abhängigkeit von k und ihrer Eingabe an und begründe diese Laufzeiten. (3 Punkte)

d) Wann und für welche k ist es sinnvoll k-WAY-MERGESORT anstelle des Algorithmus aus der Vorlesung zu nutzen? (1.5 Punkte)

Lösung 2

a) Wir geben folgenden Pseudocode an, wobei wir davon ausgehen, dass $\min\{M\}$ das Minimum der Menge M zurückgibt.

```
1:  3-WAY-MERGE(A : [ℕ₀; n_A], B : [ℕ₀; n_B], C : [ℕ₀; n_C])
2:      i_A, i_B, i_C : ℕ = 0
3:      result : [ℕ; n_A + n_B + n_C] = [0, ..., 0]
4:      while i_A < n_A or i_B < n_B or i_C < n_C do
5:          next_A = A[i_A], falls i_A < n_A, sonst ∞
6:          next_B = B[i_B], falls i_B < n_B, sonst ∞
7:          next_C = C[i_C], falls i_C < n_C, sonst ∞
8:          smallest = min{next_A, next_B, next_C}
9:          result[i_A + i_B + i_C] := smallest
10:         if next_A = smallest then
11:             i_A = i_A + 1
12:         else if next_B = smallest then
13:             i_B = i_B + 1
14:         else                                   // next_C = smallest
15:             i_C = i_C + 1
16:         end
17:     end
18:     return result
```

Analog zu der Variante, die aus der Vorlesung bekannt ist, wird hier in jedem Schleifendurchlauf genau einer der drei Counter um genau eins erhöht. Somit benötigt der Algorithmus $n_A + n_B + n_C$ Schleifendurchläufe. Da jeder Durchlauf konstante Laufzeit hat, ist die Gesamtlaufzeit $\Theta(n_A + n_B + n_C)$.

b) Wir geben folgenden Pseudocode für 3-WAY-MERGESORT an:

```
1:  3-WAY-MERGESORT(Array : [ℕ₀; n])
2:      if n ≤ 1 then
3:          return Array
4:      end
5:      A_1, A_2, A_3 := equally sized parts of Array
6:      A_1 = 3-WAY-MERGESORT(A_1)
7:      A_2 = 3-WAY-MERGESORT(A_2)
8:      A_3 = 3-WAY-MERGESORT(A_3)
9:      return 3-WAY-MERGE(A_1, A_2, A_3)
```

Wir können die Rekurrenz aus der Vorlesung auf unseren Algorithmus anpassen und erhalten somit

$$T(n) = \begin{cases} 1 & |n \leq 1 \\ 3 \cdot T(\frac{n}{3}) + \Theta(n) & |\text{sonst} \end{cases}.$$

Wir argumentieren wie in der Vorlesung anhand des Rekursionsbaumes. Dieser hat Tiefe $\log_3(n)$ und auf Ebene i eine Knotenzahl von 3^i. Mit linearen Kosten pro Ebene erhalten wir eine Gesamtlaufzeit von $\Theta(n \cdot \log(n))$.

c) Der Algorithmus k-WAY-MERGE soll k sortierte Arrays als Eingabe erhalten und deren Elemente in ein neues sortiertes Array schreiben und dieses Ausgeben. Hierfür muss der Algorithmus sich für jedes Eingabe-Array A_k merken, welches der kleinste Index ist i_k, dessen Element x_k noch nicht in das Ausgabe-Array geschrieben wurde. Solange noch solche Elemente vorhanden sind, kann zuerst das kleinste solche Element über alle nicht vollständig abgearbeiteten Eingabe-Arrays bestimmt werden und anschließend ins Ausgabe-Array geschrieben werden.

Sei n_i die Länge des i-ten Eingabe-Arrays. Dann müssen insgesamt $n = \sum_{i=1}^{k} n_i$ Elemente in das Ausgabe-Array geschrieben werden. Für jedes dieser Elemente wird ein Vergleich mit den (höchstens) k nächsten Elementen aller Eingabe-Arrays gemacht. Somit ergibt sich eine Laufzeit von $\Theta(k \cdot n)$.

Der Algorithmus für k-WAY-MERGESORT teilt das Array in k möglichst gleichgroße[2] Bereiche. Jeder Bereich wird durch einen rekursiven Aufruf sortiert. Anschließend nutzen wir k-WAY-MERGE, um die sortieren Bereiche zusammenzufügen und das sortierte Array zu erhalten. Im Basisfall können wir das Array direkt zurückgeben. Wir können die folgende Rekurrenz für die Laufzeit von k-WAY-MERGESORT aufstellen.

$$T(n) = \begin{cases} 1 & |n \leq 1 \\ k \cdot T(\frac{n}{k}) + \Theta(k \cdot n) & |\text{sonst} \end{cases}$$

Wir analysieren diese wie auf Blatt 2 gelernt. Wir haben $\log_k(n)$ Lagen mit k^i Knoten auf Lage i und Kosten von $k \cdot \frac{n}{k^i}$ pro Knoten auf Lage i. Wir erhalten die Summe $\sum_{i=0}^{\log_k(n)} k^i \cdot k \cdot \frac{n}{k^i} = \sum_{i=0}^{\log_k(n)} k \cdot n = n \cdot k \cdot \log_k(n)$. Somit ist die Gesamtlaufzeit in $\Theta(n \cdot k \cdot \log_k(n))$.

d) Der Vorteil von k-WAY-MERGESORT ist, dass wir in jedem Schritt in mehrere Teilarrays unterteilen und somit weniger Rekursionsschritte benötigen. Dafür wird allerdings jeder Rekursionsschritt teurer. Asymptotisch macht dies für konstante k keinen Unterschied. Je nach Anwendungsgebiet und Implementierung kann k-WAY-MERGESORT für bestimmte k schneller sein, in der Praxis wird k jedoch meist klein gewählt.

Weitergehende Information (nicht gefordert):
Wählen wir k asymptotisch wachsend (z.B. $k = \log n$) sorgt dies dafür, dass k-WAY-MERGESORT asymptotisch langsamer als $O(n \log n)$ wird.

Aufgabe 3 - Qual der Wahl oder Wahl der Qual (6 Punkte)

Gegeben sei ein unsortiertes Array $A : [\mathbb{N}_0;\ n]$, welches n paarweise verschiedene natürliche Zahlen enthält. Wir betrachten nur ungerade n, um die Definition des Median einfach

[2] Falls die Anzahl der Elemente nicht durch k teilbar ist, haben die letzten Bereiche jeweils ein Element weniger.

zu halten. Wir wollen das k-kleinste Element in A bestimmen. Hierbei ist k Teil der Eingabe und darf somit für die Abschätzung der Laufzeit nicht als konstant angenommen werden, d.h. insbesondere $O(kn) \neq O(n)$.

a) Zur Vorbereitung wollen wir das folgende Problem lösen: An welcher Position würde ein Element, das in A an Index $i \in \{0, \ldots, n-1\}$ steht, in dem Array stehen, dass wir aus A durch Sortieren erhalten? Beschreibe einen Algorithmus, der die gesuchte Position bestimmt, und begründe die Laufzeit deines Algorithmus. (1 Punkt)

b) Beschreibe einen Algorithmus, welcher in $O(n)$ das k-kleinste Element in A bestimmt. Dein Algorithmus darf dabei die Reihenfolge der Einträge in A vertauschen. Du darfst annehmen, dass du den Median eines unsortierten Arrays in $\Theta(n)$ bestimmen kannst. Begründe, warum dein Algorithmus die geforderte Laufzeit hat.

Hinweis: Die folgende Rekurrenz könnte nützlich sein: $T(n) = T(\frac{n}{b}) + \Theta(n)$, für ein geschickt gewähltes b. (5 Punkte)

Lösung 3

a) Wir können die Anzahl k an Elementen in A bestimmen, die kleiner als $A[i]$ sind, indem wir alle anderen Elemente in A mit $A[i]$ vergleichen. Damit wissen wir, dass $A[i]$ am Index k stehen würde, wäre A sortiert.
Dieser Algorithmus führt einen Vergleich pro Element in A durch und hat damit eine Laufzeit in $\Theta(n)$.

b) Der Algorithmus erhält als Eingabe ein Array A und die Zahl k. Zunächst bestimmen wir den Median m von A und partitionieren A mit m als Pivotelement. Dazu wird die gleiche Methode verwendet wir bei QUICKSORT. Dadurch finden wir auch die Anzahl der Elemente kleiner/größer als m und somit den Index von m, falls A sortiert wäre. Wurde m durch die Partitionierung an den Index $k-1$ geschrieben, so ist m das gesuchte Element. Ansonsten vergleichen wir k mit dem Index i, an den m in A eingeordnet wurde. Gilt $i < k-1$, so ruft sich der Algorithmus rekursiv auf dem Teilarray rechts von m auf, ansonsten auf dem Teilarray links von m.
Die Laufzeit dieses Algorithmus liegt in $O(n)$: Wir nehmen an, dass wir den Median eines (Teil-)Arrays in $\Theta(n)$ bestimmen können. Anschließend partitionieren wir ein Array, was ebenfalls in Linearzeit abläuft, da jedes Element des Arrays genau ein Mal mit dem Pivotelement verglichen und maximal ein Mal mit einem anderen vertauscht wird. Somit ergibt sich pro rekursivem Aufruf zusätzliche Arbeit mit linearem Zeitbedarf. Da wir anhand des Medians partitionieren, erfolgt der nächste Aufruf des Algorithmus mit einem Array von ungefähr halber Größe. Damit ergibt sich die folgende Rekurrenz:

$$T(n) = \begin{cases} \Theta(1) & \mid n = 1 \\ T(\frac{n}{2}) + \Theta(n) & \mid \text{sonst} \end{cases}$$

T hat die Laufzeit

$$\sum_{i=0}^{\log_b(n)} \Theta\left(\frac{n}{b^i}\right) = n \cdot \Theta\left(\sum_{i=0}^{\log_b(n)} \frac{1}{b^i}\right)$$

Da die Summe eine geometrische Reihe ist und der Reihenwert somit in $\Theta(1)$ liegt, ist die Laufzeit in $\Theta(n)$.

Aufgabe 4 - Zum Knobeln (0 Punkte + 2 Bonus)

Wir betrachten erneut das Szenario aus Aufgabe 3. Wir nehmen nun nicht mehr an, den Median eines unsortierten Arrays in $\Theta(n)$ bestimmen zu können. Gib einen randomisierten Algorithmus an, welcher das k-kleinste Element von A in erwartet $O(n)$ findet. Begründe die Korrektheit des Algorithmus und der Laufzeit. (2 Punkte)

Lösung 4

Wir betrachten den in Aufgabe 3 c) beschriebenen Algorithmus mit der folgenden Anpassung: Statt den Median des übergebenen Arrays zu bestimmen, wählt er ein Pivotelement zufällig aus und partitioniert anhand dessen das Array.
Es bleibt zu zeigen, dass die Laufzeit dieses Algorithmus in erwartet $O(n)$ liegt. Dazu schätzen wir die erwartete Laufzeit in Abhängigkeit von den gewählten Pivots ab. Wir betrachten hierbei immer ein Teilarray $A[j], A[j+1], \ldots, A[j+m-1]$ mit m Elementen, welches partitioniert werden soll. Wie schon bei der Analyse von QUICKSORT unterscheiden wir zwischen guten und schlechten Pivots. Bei ersteren liegen in beiden Teilarrays nach dem Partitionieren maximal $\lfloor \frac{2m}{3} \rfloor$ Elemente. Das bedeutet, dass das Pivot durch das Partitionieren an einem Index i mit $j + \lceil \frac{m}{3} \rceil \leq i \leq j + \lfloor \frac{2m}{3} \rfloor$ stehen muss. Die Wahrscheinlichkeit dafür ist $\frac{1}{3}$. Wurde kein gutes Pivotelement gewählt, so kann im schlechtesten Fall ein Teilarray $m-1$ Elemente erhalten. Zusätzlich werden pro rekursivem Aufruf alle Elemente mit dem Pivot verglichen, was zusätzlichen linearen Zeitaufwand bedeutet. Also gibt es eine Konstante $c \in \mathbb{R}$ so, dass cn eine obere Schranke für den zusätzlichen Aufwand ist. Damit gilt für die erwartete Laufzeit $T(n)$ des Algorithmus:

$$T(n) \leq \frac{1}{3} \cdot T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + \left(1 - \frac{1}{3}\right) \cdot T(n) + cn$$

also

$$\frac{1}{3} \cdot T(n) \leq \frac{1}{3} \cdot T\left(\left\lfloor \frac{2n}{3} \right\rfloor\right) + cn$$

und damit

$$\begin{aligned}
T(n) &\leq T\Big(\Big\lfloor \frac{2n}{3} \Big\rfloor\Big) + 3cn \\
&\leq T\Big(\Big\lfloor \frac{4n}{9} \Big\rfloor\Big) + \frac{6cn}{3} + 3cn \\
&\leq T\Big(\Big\lfloor \frac{8n}{27} \Big\rfloor\Big) + \frac{12cn}{9} + \frac{6cn}{3} + 3cn \\
&\leq \dots \\
&\leq 3cn \sum_{i=0}^{\infty} \Big(\frac{2}{3}\Big)^i \\
&\leq 3cn \cdot 3 = 9cn \in O(n)
\end{aligned}$$

Also liegt die Laufzeit des randomisierten Algorithmus in erwartet $O(n)$.