Karlsruher Institut für Technologie

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

Zusatzaufgaben 03

Algorithmen I – Sommersemester 2022

Gesamtpunkte: 35

Aufgabe 1 - Kleinaufgaben (4 Punkte)

1. Ordne die folgenden Algorithmen absteigend nach ihrer Worst-Case Laufzeitkomplexität: Quicksort, Bucketsort, Mergesort (1 Punkt)
2. Wie lautet die untere Schranke für die Worst-Case Laufzeitkomplexität bei vergleichsbasiertem Sortieren? Gilt diese untere Schranke auch für Sortieralgorithmen für ganze Zahlen? Begründe. (1 Punkt)
3. Was macht ein stabiles Sortierverfahren aus? Nenne ein Beispiel für ein stabiles Sortierverfahren. (1 Punkt)
4. Nenne die Heap-Eigenschaft bezüglich eines Arrays $A[0, \ldots, n-1]$, welches implizit einen binären Heap darstellt. (1 Punkt)

Lösung 1

1. Quicksort ($\mathcal{O}(n^2)$) $>$ Mergesort ($\mathcal{O}(n \log n)$) $>$ Bucketsort($\mathcal{O}(n)$).
2. Die untere Schranke für vergleichsbasiertes Sortieren ist $\mathcal{O}(n \log n)$. Diese gilt nicht beim ganzzahligen Sortieren. Dort kann man bei guter Eingabeverteilung ziffernweise in ($\mathcal{O}(n)$ sortieren.
3. Die relative Position gleich großer Elemente zueinander bleibt unverändert. Beispiel: Insertionsort, Mergesort, Bubblesort, LSD-Radixsort
4. $\forall i > 0 : A[\lfloor \frac{i-1}{2} \rfloor] \leq A[i]$

Aufgabe 2 - Pseudocode (8 Punkte)

```
FIRST(A: [ℕ; n], x: ℕ, y: ℕ, z: ℕ)
    if x < y then
        u: ℕ = SECOND(A, x, y, z)
        z_1: ℕ = x + (z mod (u − x))
        z_2: ℕ = u + 1 + (z mod (y − u − 1))
        FIRST(A, x, u − 1, z_1)
        if u < y then
            FIRST(A, u + 1, y, z_2)
        end
    end
```

```
SECOND(A: [ℕ; n], x: ℕ, y: ℕ, z: ℕ): ℕ
    u: ℕ = A[z]
    SWAP(A[y], A[z])
    i: ℕ = x
    j: ℕ = x
    while j < y do
        if A[j] ≤ u then
            SWAP(A[i], A[j])
            i := i + 1
        end
        j := j + 1
    end
    SWAP(A[i], A[y])
    return i
```

1. Seien $A = \langle 4, 8, 12, 34, 19, 26 \rangle$ und $n = 6$. Gib die den Zustand von A nach einem Aufruf von FIRST mit Eingabe $(A, 0, n-1, 1)$ an. (1 Punkt)

2. Was berechnet FIRST für eine beliebiges Array A, was berechnet SECOND? Haben wir diesen Algorithmus schon in der Vorlesung kennengelernt? Wie war sein Name? (2 Punkte)

3. Ändere den Pseudocode von FIRST bzw. SECOND so ab, dass er stabil wird, sich die Laufzeit aber nicht ändert. Du darfst dabei die Funktion

CONCAT$(A1, A2)$ benutzen um zwei Arrays in Linearzeit zu konkatenieren. (3 Punkte)

4. Der Algorithmus FIRST ist in-place, das heißt er benötigt nur konstant viel zusätzlichen Speicherplatz. Wie viel zusätzlichen Speicherplatz braucht dein Algorithmus? Gib im O-Kalkül an. (1 Punkt)

5. Nenne einen Algorithmus der das gleiche tut wie FIRST, aber stabil und in-place ist. Was ist seine Laufzeit in O-Kalkül? (1 Punkt)

Lösung 2

1. $A = \langle 4, 8, 12, 19, 26, 34 \rangle$

2. Der Algorithmus FIRST beschreibt QUICKSORT, sortiert also ein Array A, SECOND partitioniert dabei das Array in kleinere/größere Element

3.

```
SECOND(A: [ℕ; n], x: ℕ, y: ℕ, z: ℕ): ℕ
    B, C: [ℕ; n]
    idxB, idxC: ℕ = 0, 0
    for i ∈ {x, ..., y} do
        if A[i] ≤ A[z] then
            B[idxB] := A[i]
            idxB := idxB + 1
        else
            C[idxC] := A[i]
            idxC := idxC + 1
        end
    end
    A[x..., y] := CONCAT(B[0...idxB], C[0...idxC])
    return x + idxB
```

4. Er braucht $\mathcal{O}(n)$ zusätzlichen Speicherplatz.

5. Der Algorithmus ist INSERTIONSORT und seine Laufzeit ist $\mathcal{O}(n^2)$.

Aufgabe 3 - Min-Heaps (11 Punkte)

1. Folgende Arrays stellen Max-Heaps dar. Falls die Heap-Eigenschaft verletzt wurde, gib an wo, wenn nicht, zeichne die Baumrepräsentation des Heaps. (2 Punkte)
   a) $\langle 56, 47, 56, 10, 20, 50, 51, 1, 2, 3, 18\rangle$
   b) $\langle 56, 56, 47, 10, 20, 50, 51, 1, 2, 3, 18\rangle$
   c) $\langle 56, 47, 56, 10, 51, 20, 50, 1, 2, 3, 18\rangle$
   d) $\langle 56, 47, 56, 10, 5, 20, 50, 1, 2, 3, 18\rangle$

2. Erstelle aus Array $A = \langle 7, 8, 1, 2, 6, 7, 5\rangle$ einen binären Min-Heap mit BUILD aus der Vorlesung. Gib dabei das Array an, sobald sich Einträge darin ändern. Du musst nur die vertauschten Zahlen angeben, bei allen anderen wird angenommen, dass sie gleich bleiben. (2 Punkte)

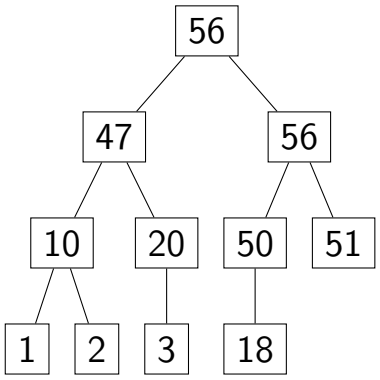
3. Füge in den Min-Heap$A = \langle 2, 7, 3, 9, 7, 5, 5\rangle$ mit Hilfe von PUSH zuerst ein Element mit Priorität 4 und anschließend ein Element mit Priorität 1 ein. Gib deinen Heap nach jeder Einfügeoperation an. (2 Punkte)

4. Lösche aus dem Min-Heap $A = \langle 2, 7, 3, 9, 7, 5, 5\rangle$ mit Hilfe von POPMIN zwei Mal das kleinste Element. Gib das Array nach jeder Operation an. (2 Punkte)

5. Man kann ein Array A sortieren, indem man mit BUILD einen Heap aus A ausbaut und mit POPMIN wiederholt das Minimum entfernt. Dieses Sortierverfahren heißt HEAPSORT und ist im Allgemeinen nicht stabil.
   a) Gib ein Beispiel an, bei dem die Stabilität eines Arrays verloren geht. (1 Punkt)
   b) Beschreibe, wie man ohne die Laufzeit oder Funktionsweise von HEAPSORT zu verändern Stabilität erzwingen kann. Du hast dazu $\mathcal{O}(n)$ zusätzlichen Speicher gegeben. (2 Punkte)

Lösung 3

1. a) ist ein Max-Heap:



   b) ist kein Max-Heap, da $47 < 50$ und $47 < 51$

   c) ist kein Max-Heap, da $37 < 51$

   d) ist kein Max-Heap, da $5 < 18$

2. - $\langle 7, 8, 1, 2, 6, 7, 5 \rangle$
   - $\langle 7, 2, 1, 8, 6, 7, 5 \rangle$
   - $\langle 1, 2, 7, 8, 6, 7, 5 \rangle$
   - $\langle 1, 2, 5, 8, 6, 7, 7 \rangle$

3. - Nach Einfügen von 4: $\langle 2, 4, 3, 7, 7, 5, 5, 9 \rangle$
   - Nach Einfügen von 1: $\langle 1, 2, 3, 4, 7, 5, 5, 9, 7 \rangle$

4. - Nach dem ersten Mal popMin: $\langle 3, 7, 5, 9, 7, 5 \rangle$
   - Nach dem zweiten Mal popMin : $\langle 5, 7, 5, 9, 7 \rangle$

5. a) Ein Gegenbeispiel ist $\langle 1, 2, 2 \rangle$

   b) Vor dem Start des Algorithmus bilden wir jeden Eintrag $A[i]$ auf ein Tupel $(A[i], i)$ ab. Das hat einen Zeitverbrauch von $\mathcal{O}(n)$, da man nur das Array von vorne nach hinten durchlaufen muss. Zusätzlich speichern wir nur $\mathcal{O}(n)$ zusätzliche Zahlen. Falls während des Sortierens zwei Elemente, also Tupel, die gleiche erste Komponente haben, dann sei das Element mit kleinerer zweiten Komponente, das also zuerst im Array stand, kleiner.

Aufgabe 4 - Weltraum-Scanner $(9\ \text{Punkte})$

Das ALMA (Atacama Large Millimeter Array) ist eines der größten Radioteleskope der Erde. Das Teleskop besteht aus 66 Antennen, welche variabel angeordnet werden könne, um verschiedene Zoomgrößen und Ausschnitte des Himmels zu erreichen. Durch geschickte mathematische und algorithmische Berechnungen erzeugt das Teleskop so ein Bild der Größe $N \times N$ wobei N von der aktuellen Konfiguration des Teleskops abhängt.
Im Folgenden wollen wir neu gefundene Planetensysteme in einer Galaxie indexieren. Pro aufgenommenem Bild erhalten wir dazu maximal G Koordinaten $(x_i, y_i) \in [0, N-1] \times [0, N-1]$, an denen Planetensysteme erkannt wurden.

1. Da sich die Erde dreht und stets nur ein Teil des Himmels gleichzeitig aufgenommen werden kann, gibt es Überschneidungen der Bilder. Wir wollen nun auf die Planetensysteme aus einem gegebenen Abschnitt der Form $[w_1, w_2] \times [0, N-1]$ oder $[0, N-1] \times [h_1, h_2]$ zugreifen. Beschreibe, wie du die gegebenen Daten so abspeichern kannst, dass in Zeit $\mathcal{O}(\log(G) + k)$ auf einen Abschnitt mit k Planetensystemen zugegriffen werden kann. Begründe deine Antwort. (3 Punkte)

2. Aufgrund bisheriger Erfahrungswerte gehen wir davon aus, dass pro Bild $\log(G)$ neue Planetensysteme entdeckt werden. Wir können in $\mathcal{O}(\log(G))$ überprüfen, ob eine Planetensystem schon entdeckt wurde. Um ein neues Planetensystem zu registrieren und zusätzliche Daten zu speichern benötigen wir $\mathcal{O}(G^3)$ Zeit, da eine Spektralanalyse gemacht werden muss. Zeige, dass ein beliebiges Planetensystem in amortisiert $\mathcal{O}(G^2 \log(G))$ Zeit abgearbeitet wird. (2 Punkte)

3. Für jedes Planetensystem erstellt die Spektralanalyse einen (a,b)-Baum, in welchem die Planeten des Planetensystems nach dem Abstand zu ihrem Stern sortiert sind.
   Im Folgenden betrachten wir einen Himmelsabschnitt mit k Planetensystemen. Beschreibe einen Algorithmus, der in $\mathcal{O}(G^3 \log(G) + k)$ alle Planeten aus der Habitablen Zone des Himmelsabschnitts zurückgibt. Du darfst dabei annehmen, dass ein Planetensystem niemals mehr als P Planeten hat, da wir noch kein Planetensystem mit mehr Planeten gefunden haben. Die Habitable Zone H_l ist für jedes Planetensystem

G_l unterschiedlich und wurde ebenfalls während der Spektralanalyse bestimmt. Die Habitable Zone ist der Bereich, in dem Leben existieren kann. Sie wird repräsentiert durch ein Tupel aus minimalem und maximalem Abstand zum Stern des Planetensystems. (4 Punkte)

Lösung 4

1. Speichere zwei (a,b)-Bäume T_x und T_y. In T_x sind die Planetensysteme nach X-Koordinate sortiert, in T_y nach Y-Koordinate. Bereichsanfragen in X-Richtung ($[w_1, w_2] \times [0, N-1]$) werden mit $T_x.\texttt{find}(w_1)$ bis $T_x.\texttt{find}(w_2)$ beantwortet. Bereichsanfragen in Y-Richtung($[0, N-1] \times [h_1, h_2]$) werden mit $T_y.\texttt{find}(h_1)$ bis $T_y.\texttt{find}(h_2)$ beantwortet.

2. Wir amortisieren mit der Aggregationsmethode. Dabei stellen wir zunächst fest, dass wir für jede der bis zu G Koordinaten überprüfen müssen, ob wir das dortige Planetensystem bereits entdeckt haben. Zusätzlich müssen $\log(G)$ Planetensysteme in jeweils $\Theta(G^3)$ Zeit registrieren. Insgesamt erhalten wir

$$\frac{G \cdot \Theta(\log(G)) + \log(G) \cdot \Theta(G^3)}{G} = \Theta(G^2 \log(G))$$

   amortisierten Aufwand pro Planetensystem.

3. Nach Teilaufgabe 1 können wir in $\mathcal{O}(\log(G))$ alle k Planetensysteme bestimmen, die im gegebenen Himmelsabschnitt liegen. (Beachte: Es gilt $k \leq G$.) Anschließend iterieren wir über diese k Planetensysteme, und prüfen für jedes Planetensystem, ob es bereits registriert ist. Wenn nicht, dann führen wir eine Spektralanalyse durch.
   Nachdem wir wissen, dass alle Planetensysteme aus dem Anfragebereich im System registriert sind und analysiert wurden, bestimmen wir für jedes Planetensystem G_l die Menge der bis zu P Planeten in H_l. Dazu machen wir analog zu Teilaufgabe 1 eine Bereichsanfrage auf dem (a,b)-Baum zu G_l. Die gesuchte Planetenmenge ist die Vereinigung der zu den Planetensystemen gesammelten Planeten.
   Wie oben erwähnt finden wir die relevanten Planetensysteme in $\mathcal{O}(\log(G))$ Zeit. Nach Teilaufgabe 2 benötigt das Finden und ggf. Registrieren von Planetensystemen insgesamt Zeit in $\mathcal{O}(G^3 \log(G))$. Das Bestimmen der

gesuchten Planetenmenge benötigt Zeit in $\mathcal{O}(k \cdot P) = \mathcal{O}(k)$, da für jedes Planetensystem bis zu P Planeten hinzugefügt werden und P eine Konstante ist.
Insgesamt ergibt sich also eine Laufzeit in $\mathcal{O}(G^3 \log(G) + k)$.
Hinweis: In der Laufzeitbetrachtung für das Finden und ggf. Registrieren von neuen Planetensystemen fällt die Amortisierung weg, denn wir betrachten k Operationen aggregiert, womit die Laufzeiten pro Planetensystem zu einer deterministischen Laufzeit „verschmelzen“.

Aufgabe 5 - To negative infinity and beyond! (3 Punkte)

In der Vorlesung wurde der ∞-Trick für (a,b)-Bäume vorgestellt. Beim $-\infty$-Trick hat das Dummy-Element den Wert $-\infty$. Wie unterscheidet sich ein (a, b)-Baum, bei dem der $-\infty$-Trick angewendet wird von einem, bei dem der ∞-Trick angewendet wird? Beachte insbesondere Veränderungen in den Operationen, die wir in der Vorlesung kennengelernt haben. (3 Punkte)

Lösung 5

Seien im folgenden $x_\perp$ das kleinste und $x_\top$ das größte Element im (a,b)-Baum T.

Das Dummy-Element ist nun nicht das Element, welches am weitesten „rechts“, sondern das Element, welches am weitesten „links“ in der Liste von T steht.

- `find`(k):
  Bei Anwendung des ∞-Tricks gibt `find` das kleinste Element größer / gleich k in T zurück, wenn $k > x_\top$, dann also `find`$(k) = \infty$.
  Bei Anwendung des $-\infty$-Tricks gibt `find` das größte Element kleiner / gleich k in T zurück,wenn $k < x_\perp$, dann also `find`$(k) = -\infty$.

- `insert`(k):
  Bei Anwendung des ∞-Tricks wird das neue Element k **vor** dem Element, welches das durch `find`(k) gefunden wird, eingefügt.
  Bei Anwendung des $-\infty$-Tricks wird das neue Element k **nach** dem Element, welches das durch `find`(k) gefunden wird, eingefügt.