Karlsruher Institut für Technologie

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

Übungsblatt 08

Algorithmen I - Sommersemester 2024

Abgabe im ILIAS bis 21.06.2024, 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 (+ 11 Bonus)

Aufgabe 1 - Heappy Potter und die Heiligtümer des Sortierens (11 Punkte)

Auf vergangenen Blättern bist du bereits verschiedene Algorithmen im Bereich des Sortierens begegnet. Da du in der Vorlesung Heaps und deren Verwendung in Heapsort kennengelernt hast, kommt dir der Gedanke, dass dir Heaps helfen könnten, um andere Sortierprobleme schneller zu lösen. Für heute hast du dir vorgenommen Sortieraufgaben der letzten Blätter, wie k-way-mergeSort, das Sortieren eines k-sortierten Arrays und die Auswahl des k-kleinsten Elements, noch einmal anzugehen. Hierbei gehen wir davon aus, dass der Datensatz paarweise verschiedene Zahlen enthält.

a) Verwende einen Heap, um den Algorithmus k-way-merge umzusetzen und gib Pseudocode für diese Variante des Algorithmus an. Der Algorithmus erhält k sortierte Arrays und gibt ein sortiertes Array, das die selben Elemente enthält zurück. Der in dieser Aufgabe entworfene k-way-merge Algorithmus soll dann in dem aus Blatt 4 Aufgabe 2 c) bekanntem k-way-mergeSort Algorithmus verwendet werden können. Begründe die Laufzeit, abhängig von k und Eingabearraygröße n, des resultierenden k-way-mergeSort.

Hinweis: verwende nicht die MERGE Methode aus der Vorlesung als Subroutine. (3.5 Punkte)

b) Für ein Array $A : [\mathbb{N}_0, n]$ und ein Element e ist $\text{sortiert}_A(e)$ die Position, die e in A hätte nachdem wir das Array sortieren. Für ein Array A und ein Element e ist $A(e)$ die Position, die e in A hat. Wir bezeichnen das Array A als k-sortiert, falls für jedes Element e aus A gilt, dass $|\text{sortiert}_A(e) - A(e)| \leq k$.

Beschreibe nun in Worten einen Algorithmus, der ein k-sortiertes Array sortiert. Begründe die Laufzeit und Korrektheit deines Algorithmus. (3.5 Punkte)

c) Für ein unsortiertes Array $A : [\mathbb{N}_0, n]$ wollen wir das k-kleinste Element in A finden. Beachte, dass nach dem Sortieren von A das k-kleinste Element an Index $k-1$ steht. Beschreibe einen Algorithmus, der das k-kleinste Element in A mit einem Zeitbedarf in $O(n + k \cdot \log(k))$ ausgibt. Begründe die Korrektheit und die Laufzeit deines Algorithmus. (4 Punkte)

Hinweis: In dieser Laufzeit kannst du zwar `buildHeap` auf A ausführen, aber du kannst auf dem aus A resultierenden Heap `popMin`, `push` und `decPrio` nicht für k Elemente aufrufen. Für k Aufrufe der obigen Funktionen benötigst du einen kleineren Heap der Größe $\mathcal{O}(k)$.

Lösung 1

a) Wir nutzen einen Min-Heap der Größe k, der je das erste Element jedes Teilarrays verwaltet.

```
1:  K-WAY-MERGE(A_1 : [ℕ_0; n_1], ..., A_k : [ℕ_0; n_k])
2:      i_1, ..., i_k : ℕ = 0
3:      result : [ℕ; n_1 + ··· + n_k] = [0, ..., 0]
4:      Q := MinHeap(k)    \\Tupel aus Element (genutzt als Priorität) und Arraynummer
5:      for j ∈ {1, ..., k} do
6:          Q.push((A_j[i_j], j))
7:      end
8:      while Q ≠ ∅ do
9:          (m, j) := Q.popMin()
10:         result[i_1 + ··· + i_k] := m
11:         i_j = i_j + 1
12:         if i_j < n_j then
13:             Q.push((A_j[i_j], j))
14:         end
15:     end
16:     return result
```

Da wir jedes Element genau einmal in Q einfügen und entfernen, durchlaufen wir die Schleife $n := \sum_{j=1}^{k} n_j$ mal. Die Laufzeit eines Schleifendurchlaufs wird durch die Heap-Operationen dominiert. Da Q die Größe k hat, benötigen diese $\mathcal{O}(\log(k))$ Zeit und die Gesamtlaufzeit des Algorithmus ist in $\mathcal{O}(n \cdot \log(k))$. Verwenden wir in der Rekurrenz und Analyse für k-WAY-MERGESORT die Laufzeit von k-WAY-MERGE, erhalten wir $\Theta(n \cdot \log_k(n) \cdot \log(k)) = \Theta(n \cdot \log(n))$. Achtung hier meint n nun die Länge des zu sortierenden Arrays.

b) Die grundlegende Idee des Algorithmus ist, dass das Element, das an der aktuell betrachteten Position stehen sollte, sich nur in einem kleinen Bereich um diese Position befinden kann. Wir können nun diesen Bereich in einem Heap verwalten. Wir nutzen einen Min-Heap H der Größe $k+1$. In diesen fügen wir die $k+1$ ersten Elemente von A ein. Anschließend gehen wir über das Array und fügen in jedem Schritt das Minimum des Heaps in die Ausgabe ein. Außerdem fügen wir in jedem Schritt das nächste Element, das noch nicht im Heap war, in H ein.

   Die Korrektheit kann über die folgende Invariante eingesehen werden. Sobald in Schritt i das i-te Element aus dem Heap in die Ausgabe eingefügt wird und das $i+k+1$-te Element zum Heap hinzugefügt wird, enthält die Ausgabe die i kleinsten Elemente aus A in sortierter Reihenfolge und alle Element in A zwischen Index 0 und $i + k + 1$ befinden sich im Heap oder in der Ausgabe. Die Invariante gilt zu beginn (Schritt 0), da das kleinste Element in A, maximal an stelle $k + 1$ steht. Wenn die Invariante vor Schritt i gilt, gilt sie auch für den folgenden Schritt: Da, das i kleinste Element maximal an Stelle $i + k + 1$ steht, befindet es sich nach Invariante im Heap (es ist nicht in A) und wird nach `popMin()` an die richtige Stelle in der Ausgabe geschrieben. Anschließend wird das $i + k + 2$-te Element aus A in den Heap eingefügt, un die Invariante gilt für den nächsten Schritt.

   Da wir jedes der n Elemente des Arrays genau einmal in H einfügen und genau einmal entfernen und H die Größe $k + 1$ hat, ist die Gesamtlaufzeit $\mathcal{O}(n \cdot \log(k))$.

c) Zunächst führen wir auf A einmal `buildHeap` aus. Nun legen wir uns einen Heap H der Größe $\mathcal{O}(k)$ an, in welchen Elemente aus A einfügen werden. Zu jedem Element merken wir uns dabei den Index, an dem es in A steht.
   Vorbereitend fügen wir $A[0]$ in H ein. Dann wiederholen wir $k-1$ Mal die folgenden Schritte: wir rufen `popMin` auf H auf und fügen die beiden Kind-Elemente des entsprechenden Elements in A in H ein. Anschließend rufen wir ein Mal `popMin` auf H auf, um das k-kleinste Element zu erhalten.
   Zu Anfang enthält H nur das minimale Element aus A, d.h. der erste `popMin`-Aufruf liefert das kleinste Element in A. In der i-ten Iteration wird das i-kleinste Element von A entfernt und dessen Kind-Elemente in H eingefügt. Dann befindet sich in H auch das $(i + 1)$-kleinste Element von A in H, denn es wurden bereits die Kinder der i kleinsten Elemente und damit auch alle Elemente kleiner gleich des $(i + 1)$-kleinsten Elementes eingefügt. Darüber hinaus musste das i-kleinste Element in H an der Wurzel stehen, denn es wurde bereits $i - 1$ Mal `popMin`

aufgerufen.
Die Methode `buildHeap` auf einem bereits bestehenden Array der Kapazität n hat eine Laufzeit in $\Theta(n)$. Sowohl `popMin` als auch `push` haben auf H eine Laufzeit in $\Theta(\log(k))$. Wir führen insgesamt k-Mal `popMin` und $(2k-2)$-mal `push` auf H aus. Damit ergibt sich eine Gesamtlaufzeit in $O(n + k\log(k))$.

Aufgabe 2 - Heappy Potter und der Gefangene von ask a banality (5 Punkte)

Du darfst in dieser Aufgabe Min- und Max-Heaps, wie in der Vorlesung eingeführt, nutzen. Max-Heaps sind dabei analog zu Min-Heaps definiert. Zeige oder wiederlege die folgenden Aussagen zu binären Heaps der Größe n.

a) Es ist möglich eine Priority-Queue so zu implementieren, dass `push` auf dieser Datenstruktur Laufzeit $o(\log(n))$ hat. (1 Punkt)

b) Es gibt eine Permutation der Zahlen $\{1, \ldots, n\}$, so dass das Einfügen der Zahlen in einen Min-Heap in dieser Reihenfolge insgesamt Zeit $\Theta(n)$ benötigt. (1 Punkt)

c) Es gibt eine Permutation der Zahlen $\{1, \ldots, n\}$, so dass das Einfügen der Zahlen in einen Min-Heap in dieser Reihenfolge insgesamt Zeit $\Theta(n \cdot \log(n))$ benötigt. (1 Punkt)

d) Sei A die Umsetzung eines Min-Heaps als Array. Wenn wir nun die Reihenfolge der Elemente in A umkehren, ist das Ergebnisarray die Umsetzung eines Max-Heaps. (1 Punkt)

e) Wir können einen Max-Heap in Linearzeit in einem Min-Heap umbauen. (1 Punkt)

Lösung 2

a) Die Aussage ist **korrekt**. Wir können das Einfügen auf Kosten von `popMin` beschleunigen. So können wir z.B. eine unsortierte Liste nutzen und in konstanter Zeit die Elemente vorne an diese anhängen. Dann müssen wir das Minimum allerdings durch lineare Suche bestimmen. Die Schranke aus der Vorlesung besagt nur, dass nicht beide Operationen sublogarithmisch sein können.

b) Die Aussage ist **korrekt**. Wir können die Elemente in aufsteigend sortierter Reihenfolge einfügen. Dann fügen wir in jedem Schritt ein Element ein den Heap ein, der größer als alle aktuell enthaltenen ist. Somit werden keine Vertauschungsoperationen nötig. Da wir also jedes Element in konstanter Zeit einfügen ist die Gesamtlaufzeit linear.

c) Die Aussage ist **korrekt**. Wir können die Elemente in absteigend sortierter Reihenfolge einfügen. Dann fügen wir in jedem Schritt ein Element ein den Heap ein, der kleiner als alle aktuell enthaltenen ist. Somit muss das Element von einem Blatt

bis an die Wurzel getauscht werden. Da wir also jedes Element in logarithmischer Zeit einfügen ist die Gesamtlaufzeit $\Theta(n \cdot \log(n))$.

d) Die Aussage ist **falsch**. Betrachte das Array $[1, 2, 4, 3]$. Dies ist ein gültiger Min-Heap, aber die Umkehrung des Arrays ist kein Max-Heap, da das Maximum nicht an erster Stelle steht.

e) Die Aussage ist **korrekt**. Wir können `buildHeap` aus der Vorlesung verwenden. Dabei ignorieren wir, dass das gegebene Array bereits ein Max-Heap ist.

Aufgabe 3 - Heappy Potter und Major Tom Riddle (4 Punkte)

Dr. Meta hat sich bei seinen Bibern mit EM-Fieber angesteckt. Als verwöhnter Erfolgsfan möchte er natürlich nur die Mannschaft unterstützen, die am Ende auch den Titel gewinnt. Daher hat er sich das folgende Vorgehen für seine Prognose überlegt. Zuerst repräsentiert er die m teilnehmenden Mannschaften jeweils durch eine Gruppe von Bibern. Die Mannschaften werden dann in einer Tabelle basierend auf ihrer Punktzahl angeordnet. Bei Punktgleichheit steht das Team mit kleineren Namen (Sortierung analog zum Duden) in der Tabelle weiter oben. Die Punkte werden durch Duelle, die stets mit einem Sieger enden, zwischen den Bibern, welche die jeweiligen Mannschaften repräsentieren, ermittelt. Nach jedem Duell erhält der Sieger zwei Punkte und dem Verlierer wird ein Punkt abgezogen. Zu Beginn haben alle Teams 0 Punkte. Solange zwei Teams nicht gespielt haben, bleibt diese initiale Ordnung erhalten. Dr. Meta wählt nun so lange zwei Teams aus und lässt diese gegeneinander antreten, bis ein Team n Punkte erreicht hat. Die Mannschaft, die dies erreicht, kann mit Dr. Metas Unterstützung rechnen. Eines der beiden ausgewählten Teams ist stets der aktuelle Tabellenführer, das andere wird durch das Oktopus-Orakel Paul aus den restlichen Teams ausgewählt. Dies erfolgt durch Aufruf der Funktion `paulChoose`() in $\Theta(1)$. Die Funktion gibt hierbei einen Zeiger auf die ausgewählte Mannschaft zurück.

Wir betrachten zur Veranschaulichung das folgende Beispiel mit $n = m = 5$ und Testnationen aus aller Welt. Es treten die Mannschaften Aserbaumdschan, Bauhamas, Costa Rinda, Dämmemark und Eibenbaumküste an.

Start

Team	Punkte
A	0
B	0
C	0
D	0
E	0

Nach A > D

Team	Punkte
A	2
B	0
C	0
E	0
D	-1

Nach E > A

Team	Punkte
E	2
A	1
B	0
C	0
D	-1

Nach E > A

Team	Punkte
E	4
A	0
B	0
C	0
D	-1

Nach E < B

Team	Punkte
E	3
B	2
A	0
C	0
D	-1

Nach E > D

Team	Punkte
E	5
B	2
A	0
C	0
D	-2

Die Graphik zeigt den Verlauf der Testrunden. Es ist jeweils das aktuelle Duell, so wie die daraus resultierende Tabelle angegeben. Die Kleiner- bzw. Größerzeichen geben den Sieger des Duells an. Der aktuelle Tabellenführer steht stets links und das durch Paul gewählte Team rechts. Alle Teams sind durch ihren Anfangsbuchstaben repräsentiert.

a) Zeige, dass für beliebige $m \geq 2$, eine Mannschaft, die zu keinem Zeitpunkt (bevor n Punkte von einem Team erreicht wurden) der aktuelle Tabellenführer ist, trotzdem

als erstes n Punkte erreichen kann. (1 Punkt)

Hinweis: Für einen Existenzbeweis dieser Art, darfst du stets das Ergebnis des Orakels selbst wählen.

b) Beschreibe nun in Worten, wie du Dr. Metas Vorhersage mit Min- oder Max-Heaps möglichst effizient umsetzen kannst. Begründe die Laufzeit abhängig von m und der Anzahl der Duelle d, die benötigt werden bis der Sieger gefunden wurde. (3 Punkte)

Lösung 3

a) Wir betrachten eine Instanz, die die Mannschaften A und B, so wie $m-2$ weitere Teams mit lexikographisch größeren Namen, enthält. Wir sorgen nun dafür, dass A stets der aktuelle Favorit ist. Hierfür nutzen wir die Sortierung der Namen und erzwingen, dass nur A und B spielen, wodurch die beiden stets mehr Punkte als alle anderen Teams aufweisen. Außerdem wählen wir B immer als Rückgabe des Orakes. Somit treten in jedem Duell A und B gegeneinander an. Wenn nun beginnend bei A die beiden Mannschaften immer abwechselnd gewinnen, liegt Mannschaft A stets vor Mannschaft B. Zu Beginn steht A aufgrund des Namens an der Spitze, danach hat A immer entweder mehr Punkte oder ist im Namensvorteil. Nach je einem Sieg der Mannschaft A und einem darauffolgendem von B haben beide Mannschaften einen Punkt hinzugewonnen. Wir wiederholen dies bis beide Mannschaften $n-2$ Punkte erreicht haben. Wenn nun B das letzte Duell gewinnt, erfüllt diese Mannschaft die Anforderung.

b) Wir nutzen einen Max-Heap, um die aktuelle Tabelle zu verwalten. Als Priorität nutzen wir ein Tupel aus aktueller Punktzahl und Name, welches wir lexikographisch vergleichen. In jeder Runde können wir somit den aktuellen Tabellenführer durch `popMax` ermitteln. Wir nutzen dann `paulChoose`, um den Gegner zu ermitteln. Über `incPrio` können wir dann die Punktzahl beider Teams entsprechend der Regeln anpassen. Dies wiederholen wir nun bis das Team, welches wir mit `popMax` entfernen, n Punkte erzielt hat.

Da wir in jeder Runde konstant viele Operationen auf einem Heap der Größe m aufrufen und `paulChoose` konstante Laufzeit hat, benötigen wir Zeit $\mathcal{O}(\log(m))$ pro Duell. Somit ergibt sich die Gesamtlaufzeit von $\mathcal{O}(d \cdot \log(m))$.

Aufgabe 4 - Zum Knobeln (0 Punkte + 11 Bonus)

Eine Legende besagt, dass der Student, der diese Aufgabe lösen kann, sich den ewigen Respekt Dr. Metas verdient hat:
Starte mit einer Zahl $n \in \mathbb{N}$ und stelle diese in 2er-Superpotenz dar. Das bedeutet, dass wir die Zahl als Summe von 2er-Potenzen darstellen und dann rekursiv wiederum alle Exponenten als 2er-Potenz darstellen. Dies erfolgt so lange, bis wir nur Zahlen kleiner gleich 2 nutzen.

Im nächsten Schritt ersetzen wir jede 2 durch eine 3. Danach ziehen wir 1 von dem Ergebnis ab.

Dieses Vorgehen wiederholen wir. Also als 3er-Superpotenz darstellen, 3 durch 4 ersetzen, 1 abziehen, usw.

Beispiel:

- Zahl: 25
- 2er-Superpotenz: $25 = 2^4 + 2^3 + 1 = 2^{2^2} + 2^{2^1+1} + 1$
- 3er-Superpotenz: $3^{3^3} + 3^{3^1+1} + 1 \geq 7 \cdot 10^{12}$
- 1 abziehen: $3^{3^3} + 3^{3^1+1}$

a) Gib für die Zahl 4 die ersten sechs Folgenglieder an. (1 Punkt)

b) Zeige, dass wir so für jedes n irgendwann bei 0 ankommen. (10 Punkte)

Hinweis: Diese Aufgabe ist schwerer als bisherige Knobelaufgaben. Verzweifle also nicht, falls du nicht auf die Lösung kommst. Es kann trotzdem interessant sein, etwas herum zu überlegen.

Lösung 4

a) Die Folgenglieder sind:
   - $4 = 2^2$
   - $3^3 - 1 = 26 = 2 \cdot 3^2 + 2 \cdot 3^1 + 2$
   - $2 \cdot 4^2 + 2 \cdot 4^1 + 2 - 1 = 41$
   - $2 \cdot 5^2 + 2 \cdot 5^1 + 1 - 1 = 60$
   - $2 \cdot 6^2 + 2 \cdot 6^1 - 1 = 83 = 2 \cdot 6^2 + 6^1 + 5$
   - $2 \cdot 7^2 + 7^1 + 5 - 1 = 109$

b) Dr. Metas Respekt kann man sich nicht durch Nutzen einer Musterlösung erschleichen!