Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Arbeitsgruppe Algorithmik

Wintersemester 2024/2025

Version 2024-11-08 12:03

Prof. Dr. Sebastian Wild
Dr. Nikolaus Glombiewski

Übungen zur Vorlesung
Effiziente Algorithmen

Abgabe: 15.11.2024,
bis **spätestens** 19:00 Uhr
über die ILIAS Plattform

Übungsblatt 3

Aufgabe 3.1: Anwendung von Sortieralgorithmen (1+2+1) (4 Punkte)

a) Sortieren Sie das Array $[3, 1, 4, 1, 5, 8, 9, 2, 6]$ mit dem Standard Mergesort Algorithmus. Zeichnen Sie den Zustand des Array für jede Höhe des Merge Baums und markieren Sie darin die Teile des Arrays, welche vom Algorithmus sortiert wurden.

b) In der Vorlesung wurde als ein vereinfachtes Kostenmaß für Mergesort die Größe der Output Runs verwendet. Geben Sie für das Array aus Teilaufgabe a) die Kosten für den Standard Mergesort, Natural Bottom-Up Mergesort und $\mathcal{H}$ an. Existiert eine Merge Reihenfolge mit geringeren Mergekosten? Begründen Sie Ihre Antwort.

c) Beschreiben Sie einen Algorithmus, der n ganze Zahlen im Bereich von $1, \ldots, n^2$ mit einer Laufzeit **und** einem Speicherplatzbedarf von $\mathcal{O}(n)$ sortiert. Begründen Sie die Laufzeit Ihres Algorithmus.

Aufgabe 3.2: Quicksort (2+1+1) (4 Punkte)

Angenommen der Quicksort Algorithmus aus der Vorlesung wählt stets das letzte Element als Pivot. Wie verhält sich die Θ-Asymptotik der erwarteten Laufzeit vom Quicksort Algorithmus aus der Vorlesung auf den folgenden Eingaben?

Geben Sie jeweils die Extremfälle an und analysieren Sie diese, indem Sie beschreiben wie Quicksort vorgeht (Swaps, Zustand pro Rekursion, ...) . Gehen Sie ferner darauf ein, inwiefern etwaige negative Effekte auch für weniger extreme Fälle auftreten. Sofern nötig, schlagen Sie Verbesserungen der Quicksort Implementierung vor, die den Effekten entgegenwirken.

a) Schlüssel können mehrfach vorkommen, d.h. es existieren Duplikate.

b) Die Eingabe ist teilweise schon sortiert.

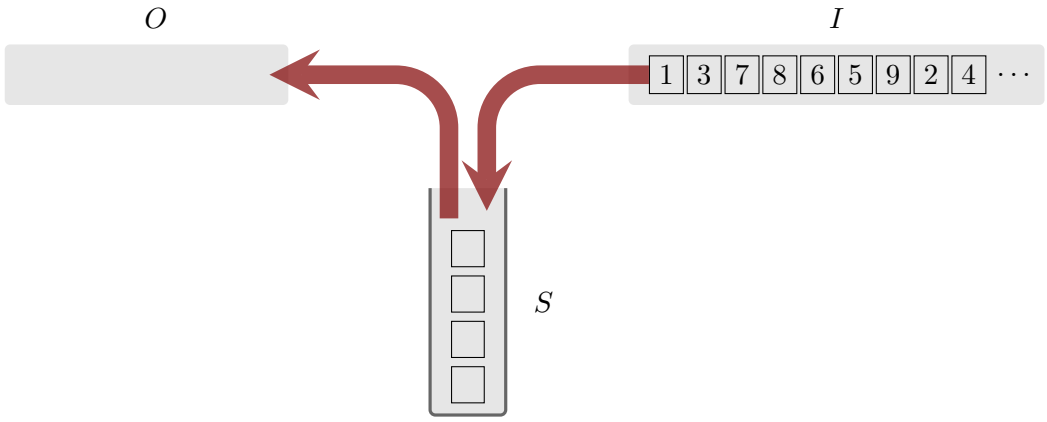
c) Die Eingabe ist teilweise vorsortiert, allerdings rückwärts.

Aufgabe 3.3: Sortieren mit Stacks (1+3+2) (6 Punkte)

Betrachten Sie das folgende Streaming Modell: Gegeben seien n (paarweise verschiedene) Elemente in einem Input Stream I. Sie können nacheinander jeweils auf ein Element von I zugreifen und müssen einen Output in den Output Stream O erstellen, indem Sie ebenfalls Elemente eins nach dem anderen einfügen. Es gibt keine andere Möglichkeit I oder O zu modifizieren oder

darauf zuzugreifen. Die Streams können Sie sich als zwei Queues vorstellen, wobei I nur *dequeue* und O nur *enqueue* Operationen erlaubt. Die Anzahl von Elementen n ist vorab bekannt.

Zusätzlich zu I und O (und potentiell einer konstanten Anzahl von lokalen Variablen) können Elemente in *einem Stack* S abgelegt werden. Ihnen stehen somit zu jedem Zeitpunkt folgende Operationen zur Verfügung: Ein Element von I oder S entfernen; ein Element in O oder S einfügen.



Anmerkung: Vergleiche sind nur möglich zwischen dem obersten Element auf S und dem nächsten Element von I. Somit müssen zwischen jedem (nicht redundanten) Vergleich Elemente mittels einer Operation "$I \to S$" oder "$S \to O$" verschoben werden.

a) Zeigen Sie, dass es in dem obigen Model *nicht* immer möglich ist, einen sortierten Output Stream zu generieren. D.h. für eine Permutation von n Elementen in I ist es nicht möglich Elemente in O in aufsteigend sortierter Reihenfolge einzufügen.

   Sie können ein "genügend großes n" für diesen Beweis annehmen.

b) Nehmen Sie an O kann als Input für einen weiteren Durchlauf genutzt werden, d.h. O and I sind verbunden und bilden eine große Queue.

   Wir nehmen der Einfachheit halber an, dass ein Durchlauf immer beendet werden muss bevor der nächste gestartet wird, d.h. bevor ein Element das zweite Mal aus I entfernt werden kann, müssen alle n Elemente in O ausgegeben werden. D.h. Elemente können einander nicht überrunden und jede Ausführung besitzt eine wohl-definierte Anzahl von k Runden.

   Entwerfen Sie einen Sortier-Algorithmus für dieses Modell, d.h. eine Sequenz von "$I \to S$" oder "$S \to O$" Operationen. Weitere Operationen zur Neuanordnung von Daten steht nicht zur Verfügung, aber Ihr Algorithmus kann beliebig viel Zeit oder Speicherplatz verwenden, um die nächste Operation zu berechnen und ein Vergleich von $S.top()$ und $I.front()$ ist kostenlos.

   Analysieren Sie den Worst Case für die Anzahl von Durchläufen für Ihren Algorithmus, um n Elemente zu sortieren. Um volle Punktzahl zu erhalten, muss Ihr Algorithmus eine Laufzeit $k \in O(\log n)$ erreichen.

   Hinweis: Sie können sich Inspiration vom Sortieren auf Bandspeicher holen: `https://en.wikipedia.org/wiki/Merge_sort#Use_with_tape_drives`.

c) Geben Sie eine nicht-triviale untere Schranke für k für einen *beliebigen* Sortieralgorithmus in diesem Modell an.

Aufgabe 3.4: Adaptives Sortieren (3+1+2) (6 Punkte)

Ein Array $A[0..n)$ wird als *d-lösch-sortierbar* bezeichnet, wenn es Positionen $0 \leq i_1 < i_2 < \cdots < i_d < n$ gibt, sodass nach dem Löschen der Elemente an den Position $i_1, \ldots, i_d$ aus A die resultierte Sequenz sortiert ist. Zum Beispiel ist

$$2, 4, 1, 6, 7, 5, 8, 12, 0$$

3-lösch-sortierbar (indem die Elemente $1, 5, 0$ entfernt werden), aber nicht 2-lösch-sortierbar.

Im Folgenden wird angenommen dass ein gegebenes Array $A[0..n)$ d-lösch-sortierbar ist. Sie dürfen annehmen, dass die Elemente A paarweise verschieden sind.

a) Entwerfen Sie einen adaptiven Sortieralgorithmus für A, welcher als Input A und ein sortiertes Array $D[0..d)$ von Positionen erhält $i_1, \ldots, i_d$, die A *d-lösch-sortierbar* machen.

   Unter der Annahme dass $d \ll n$ sollte Ihr Algorithmus eine Laufzeit von $o(n \log n)$ besitzen; eine vollständige Lösung würde ein $\sqrt{n}$-lösch-sortierbares $A[0..n)$ in $O(n)$ Laufzeit sortieren.

   Beschreiben Sie Ihren Algorithmus (entwender textuell oder in Pseudocdeo) und analysieren Sie die Laufzeit (als Θ-Klasse).

b) Ab welcher Größe von d benötigt Ihre Lösung $\omega(n)$ Laufzeit?
   Ab welcher Größe von d benötigt Ihre Lösung $\Omega(n \log n)$ Laufzeit?

c) Entwerfen Sie einen Algorithmus wie in a) ohne d oder die Menge an Positionen als Input zu bekommen.

   Hinweis: Können Sie eine Menge von Positionen I finden, sodass A nach Entfernen der Elemente sortiert ist ohne dass I zu groß wird?