

# Übungsblatt 4 zur Vorlesung Effiziente Algorithmen (Winter 2025/26)

**Abgabe:** Bis 2025-11-14 18:00, on ILIAS.

## 1. Aufgabe

20 + 10 + 10 Punkte

Angenommen der Quicksort-Algorithmus aus der Vorlesung wählt stets das *letzte* Element als Pivot. Wie verhält sich die  $\Theta$ -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.

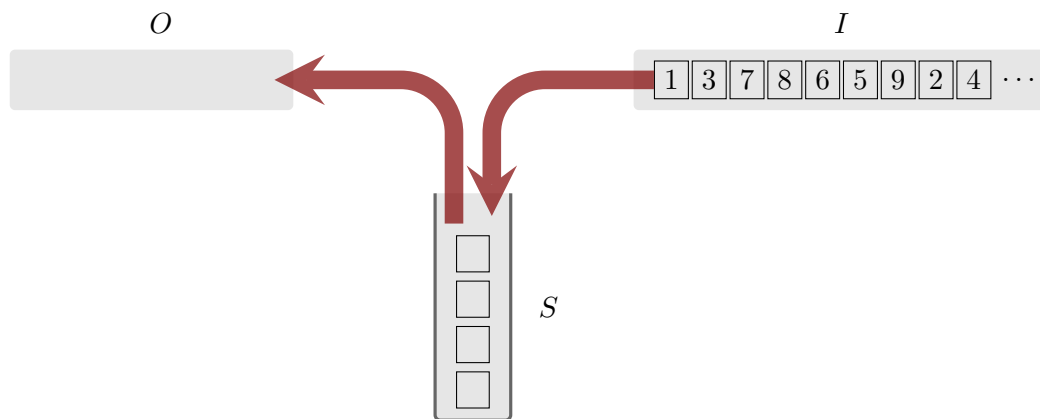
- a) Schlüssel können mehrfach vorkommen, d.h. es existieren Duplikate.
- b) Die Eingabe ist teilweise schon sortiert.
- c) Die Eingabe ist teilweise vorsortiert, allerdings rückwärts.

## 2. Aufgabe

10 + 30 + 20 Punkte

Betrachten Sie das folgende Streaming-Modell: Gegeben seien  $n$  (paarweise verschiedene) Elemente in einem Input-Stream  $I$ . Sie können nacheinander jeweils das nächste Element aus  $I$  konsumieren oder einen Output in den Output-Stream  $O$  schreiben. Es gibt keine andere Möglichkeit  $I$  oder  $O$  zu modifizieren oder auf Elemente zuzugreifen. Die Streams können Sie sich als zwei Queues vorstellen, wobei  $I$  nur *dequeue* und  $O$  nur *enqueue* Operationen unterstützt. Die Anzahl  $n$  an Elementen ist vorab bekannt.

Zusätzlich zu  $I$  und  $O$  (und potentiell einer konstanten Anzahl lokaler 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 “ $I \rightarrow S$ ” oder “ $S \rightarrow O$ ” verschoben werden.

- Zeigen Sie, dass es in dem obigen Modell *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.
- Nehmen Sie an,  $O$  kann als Input für einen weiteren Durchlauf genutzt werden, d.h.  $O$  und  $I$  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. Elemente können einander also nicht “übereunden” und jede Ausführung besitzt eine wohl-definierte Anzahl von  $k$  Runden.

Entwerfen Sie einen Sortier-Algorithmus für dieses Modell, d.h. eine Sequenz von “ $I \rightarrow S$ ” oder “ $S \rightarrow O$ ” Operationen, die jede mögliche Eingabe korrekt sortiert. Weitere Operationen zur Neuordnung von Daten steht nicht zur Verfügung, aber Ihr Algorithmus kann beliebig viel Zeit oder Speicherplatz verwenden, um die nächste Operation zu berechnen. Vergleiche von  $S.top()$  und  $I.front()$  sind 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](https://en.wikipedia.org/wiki/Merge_sort#Use_with_tape_drives).

- Geben Sie eine nicht-triviale untere Schranke für  $k$  für *beliebige* Sortieralgorithmen in diesem Modell an.

### 3. Aufgabe

30 + 10 + 20 Punkte

Ein Array  $A[0..n)$  wird als *d-lösch-sortierbar* bezeichnet, wenn es Positionen  $0 \leq i_1 < i_2 < \dots < i_d < n$  gibt, sodass nach dem Löschen der Elemente an den Position  $i_1, \dots, i_d$  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  $i_1, \dots, i_d$ , die  $A$  *d-lösch-sortierbar* machen.

Unter der Annahme dass  $d \ll n$  ( $d$  “viel kleiner” als  $n$ ) 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 (entweder textuell oder in Pseudocode) und analysieren Sie die Laufzeit (als  $\Theta$ -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?