A Algorithmendesign (2 Punkte)

Gegeben sei eine sortierte Liste mit n Elementen. Nun haben Sie $k \leq n$ neue unsortierte Elemente in einem Array, die in die sortierte Liste einfügt werden sollen, so dass die Liste nach dem Einfügen weiter sortiert ist.

1. *Warmup:* Geben Sie einen Algorithmus an, der Laufzeit $\mathcal{O}(k \cdot n)$ hat und das Problem löst.
2. Geben Sie nun einen Algorithmus an, der Laufzeit $\mathcal{O}(k \cdot \mathsf{log}(k) + n)$ hat und das Problem löst.

A.1 Musterlösung

1. Man füge die Elemente nacheinander in die sortierte Liste ein. Um ein Element einzufügen macht man lineare Suche: man geht die Liste Stück für Stück durch und vergleicht den Wert des aktuellen Elements mit dem Wert des einzufügenden Elements bis man die richtige Stelle gefunden hat.
   Die lineare Suche hat pro einzufügendem Element Aufwand $\mathcal{O}(n + k)$, da im schlimmsten Fall die ganze Liste durchlaufen werden muss. Da wir k Elemente einfügen, ergibt sich somit ein Gesamtaufwand von $\mathcal{O}(k(n + k))$ und da $k \leq n$ die Abschätzung des Gesamtaufwands $\mathcal{O}(kn)$.
2. Nun gehen wir ein wenig anders vor. Zuerst sortieren wir das Array der neuen Elemente in Zeit $\mathcal{O}(k \log k)$ zum Beispiel mit dem Algorithmus Mergesort aus der Vorlesung. Das sortierte Array wird dann in eine Liste konvertiert. Dann haben wir zwei sortierte Listen: in der einen sortieren Liste befinden sich die k neuen Elemente und in der anderen befinden sich die n alten sortierten Elemente. Nun verwenden wir die Funktion *merge* aus der Vorlesung und erhalten so in Zeit $\mathcal{O}(n)$ eine sortierte Liste, die die Elemente beider Listen enthält. Insgesamt ergibt sich also ein Aufwand von $\mathcal{O}(k \log k + n)$

B Sortieren

Gegeben sei eine Liste A = ⟨3,4,5,1,17,19,13,14⟩

B.1 Quantifizierung (1 Punkt)

Bestimmen Sie die Anzahl der Inversionen und Runs in A.

Musterlösung:
(3,1) , (4,1) , (5,1) , (17,13) , (17,14) , (19,13) und (19,14) sind die 7 Inversionen in A.
⟨3,4,5⟩ , ⟨1,17,19⟩ und ⟨13,14⟩ sind die 3 Runs in A.

B.2 Sortieren mit Insertionsort (1 Punkt)

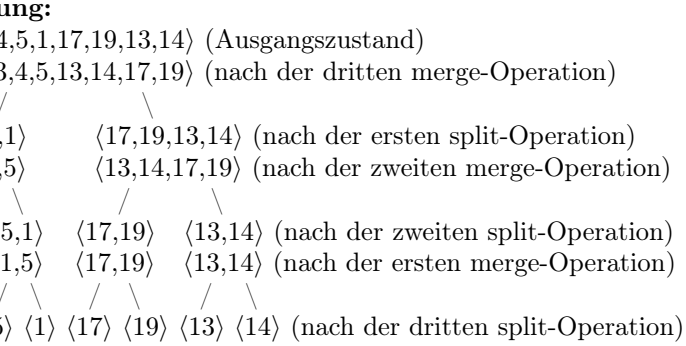
Sortieren Sie A mit der In-Place Variante von Insertionsort. Geben Sie den Zustand von A nach jedem Insert-Schritt an.

Musterlösung:
0: ⟨|3,4,5,1,17,19,13,14⟩
1: ⟨3|4,5,1,17,19,13,14⟩
2: ⟨3,4|5,1,17,19,13,14⟩
3: ⟨3,4,5|1,17,19,13,14⟩
4: ⟨1,3,4,5|17,19,13,14⟩
5: ⟨1,3,4,5,17|19,13,14⟩
6: ⟨1,3,4,5,17,19|13,14⟩
7: ⟨1,3,4,5,13,17,19|14⟩
8: ⟨1,3,4,5,13,14,17,19|⟩

B.3 Sortieren mit Mergesort (1 Punkt)

Sortieren Sie A mit der Variante von Mergesort, welche die n-elementigen Teillisten zwischen Element $\lfloor \frac{n}{2} \rfloor$ und $\lfloor \frac{n}{2} \rfloor + 1$ auftrennt. Zeichnen Sie einen Rekursionsbaum und annotieren Sie jeden Knoten und jedes Blatt mit dem Ergebnis dieser Auftrennung. Annotieren Sie anschließend jeden inneren Knoten des Rekursionsbaums zusätzlich mit dem Ergebnis der merge-Operation.

Musterlösung:



Übungsblatt 5, Aufgabe B.4: Algorithmendesign

Gegeben sei die Liste $A = \langle 3, 4, 5, 1, 17, 19, 13, 14 \rangle$. Gegeben Sei zusätzlich die Liste $R = \langle 2, 5, 7 \rangle$, welche die geordneten Indizes all derjenigen Elemente in A enthält, welche sich am Ende eines Runs befinden.

Entwurf (2 Punkte)

Implementieren Sie eine Variante von Mergesort $\mathsf{merges0rt}(A, R)$, dessen Rekursionstiefe in $\mathcal{O}(|R|)$ liegt. Implementieren Sie den Algorithmus so, dass unter den gegebenen Voraussetzungen nach jedem Rekursionsschritt gilt: "A ist sortiert" g.d.w. $|R| = 1$.

Algorithm 1: merges0rt(A,R)

```
Data: List A, List R
Result: Sorted List
1  if |R| = 1 then
2      return A
3  else
4      R.popBack()
5      splitIndex ← R.last
6      (firstHalf, secondHalf) ← A.splitAfterIndex(splitIndex)
7      return merge(merges0rt(firstHalf, R), secondHalf)
```

Algorithm 2: merges0rt(A,R,offset)

```
Data: List A, List R
Result: Sorted List
1  if |R| = 1 then
2      return A
3  else
4      mid ← ⌊|R|/2⌋
5      splitIndex ← R[mid]
6      R_0 ← subList(R, 0, mid − 1)
7      R_1 ← subList(R, mid + 1, |R| − 1)
8      for e ∈ R_1 do
9          e ← e - splitIndex
10     (A_0, A_1) ← A.splitAfterIndex(splitIndex)
11     return merge(merges0rt(A_0, R_0), merges0rt(A_1, R_1))
```

Zusatzfrage (1 Punkt)

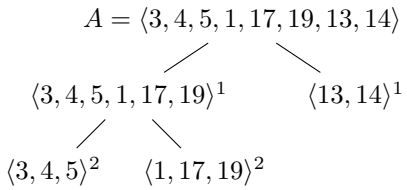
Würde es die asymptotische worst-case Laufzeit Ihres Algorithmus verändern, falls die Laufzeit des Initialisierungsschritts, in dem R berechnet wird, mit berücksichtigt wird? Begründen Sie Ihre Antwort.

Die Worst-Case-Laufzeit ändert sich nicht, da $|R|$ als Faktor in der Laufzeit des Algorithmus ohne Mitberechnung von R schon vorkommt.

Ausführen (1 Punkt)

Geben Sie einen Rekursionbaum mit Annotationen an.

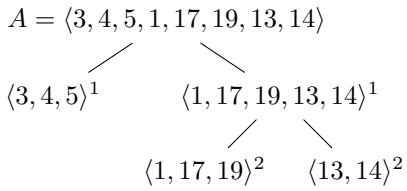
Algorithmus 1



[1] merged: $\langle 1, 3, 4, 5, 13, 14, 17, 19 \rangle$

[2] merged: $\langle 1, 3, 4, 5, 17, 19 \rangle$

Algorithmus 2



[1] merged: $\langle 1, 3, 4, 5, 13, 14, 17, 19 \rangle$

[2] merged: $\langle 1, 13, 14, 17, 19 \rangle$

C Hashing: Nachschlag (1 Punkt)

Ein Mitarbeiter des Instituts für abgewandte Informatik hat eine neue Theorie zum Hashing entwickelt:

Es sei U ein Universum von Eingaben, $\mathcal{H} \subseteq \{0, \dots m-1\}^U$ eine Familie von Hashfunktionen, und T eine Hashtabelle der Größe m. Die Familie der Hashfunktionen $\mathcal{H}$ sei so gewählt, dass für jedes $u \in U$ und $j \in \{0, \dots m-1\}$ bei zufälliger gleichverteilter Wahl von $h \in \mathcal{H}$ die Wahrscheinlichkeit $\mathbb{P}[h(u) = j] = \frac{1}{m}$ ist.

Die Behauptung ist nun, dass falls diese Bedingungen erfüllt sind, die erwartete Anzahl von Kollisionen für ein Element u beim Hashen von m Elementen in $\mathcal{O}(1)$ liegt.

Hat dieser Mitarbeiter recht oder liegt er falsch? Liefern Sie zu der Antwort einen Beweis oder ein Gegenbeispiel.

Musterlösung:
Die Behauptung ist falsch.
Gegenbeispiel:
Es sei $U = \{1, \dots, m\}$ und $\mathcal{H} = \{h_j \mid j \in \{0, \dots, m-1\},\ h_j(u) = j\ \forall u \in U\}$. Dann ist für jedes feste u und für jede Wahl $h_j \in \mathcal{H}$ dann $h_j(u) = j$. Da aber eine der m Hashfunktionen zufällig gewählt wird ist $\mathbb{P}[h(u) = j] = \frac{1}{m}$. Unabhängig von der Wahl von h kollidiert u aber immer mit allen anderen $m-1$ Elementen der Eingabe. Damit liegt der Erwartungswert der Anzahl der Kollisionen für ein Element u in $\Theta(m)$.