Präsenzübung 9

Besprechung: 15.12.25 – 19.12.25

Auf diesem Blatt schauen wir uns verschiedene Sortieralgorithmen an. Dazu definieren wir noch zwei Eigenschaften, die zur Charakterisierung von Sortieralgorithmen verwendet werden:

Stabilität. Ein Sortieralgorithmus heißt **stabil**, wenn er die relative Reihenfolge von Elementen mit gleichem Schlüsselwert beibehält. Das bedeutet: Wenn zwei Elemente a und b denselben Sortierschlüssel haben und a in der Eingabe vor b steht, dann steht a auch in der sortierten Ausgabe vor b.

Beispiel: Gegeben sei eine Liste von Studierenden, die bereits nach Namen sortiert ist. Sortieren wir diese Liste nun stabil nach Matrikelnummer, so bleiben Studierende mit gleicher Matrikelnummer (falls vorhanden) weiterhin alphabetisch nach Namen geordnet. Stabilität ist besonders wichtig, wenn nach mehreren Kriterien sortiert werden soll.

In-place. Ein Sortieralgorithmus arbeitet **in-place** (dt. *vor Ort*), wenn er nur eine konstante Menge an zusätzlichem Speicherplatz benötigt, also $O(1)$ extra Speicher unabhängig von der Eingabegröße. Der Algorithmus sortiert die Daten direkt im gegebenen Array, ohne eine Kopie der gesamten Daten anzulegen.

Hinweis: Manche Algorithmen benötigen noch $O(\log N)$ zusätzlichen Speicher (z. B. für einen Rekursionsstack). Wir bezeichnen diese Algorithmen auch als in-place.

Aufgabe 1: Alle meine Sortieralgorithmen (keine Punkte)

Betrachten Sie die folgenden Sortieralgorithmen. Die Algorithmen sind hier jeweils in Ihrem Pseudocode gegeben, allerdings ohne Namen.

```
n := lengthA
B = empty array of length n
H = BuildMaxHeap(A)
for i = 1, ..., n
    B(n − i + 1) =ExtractMax(H)
```

```
n = length(A)
B = new array of length n
for it=1, ..., n
    Find the smallest element a and its index p in A
    B(it) = a # Insert the element in B
    A(p) = NaN # Replace the element in A by NaN
Return B
```

```
for i = 1 to A.length − 1
    for j = A.length downto i + 1
        if A[j] < A[j − 1]
            exchange A[j] with A[j − 1]
```

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 . . j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

```
if (|A| = 1), Return A(1), end if
Choose pivot element p ∈ A according to some strategy
A_− := {a ∈ A | a < p}
A_= := {a ∈ A | a = p}
A_+ := {a ∈ A | a > p}
Return [Quicksort(A_−), A_=, Quicksort(A_+)] # concatenation of the three arrays
```

```
Input:  An array of numbers a[1...n]
Output:  A sorted version of this array

if n > 1:
  return merge(mergesort(a[1...⌊n/2⌋]), mergesort(a[⌊n/2⌋ + 1...n]))
else:
  return a

function merge(x[1...k], y[1...l])
if k = 0:  return y[1...l]
if l = 0:  return x[1...k]
if x[1] ≤ y[1]:
  return x[1] ∘ merge(x[2...k], y[1...l])
else:
  return y[1] ∘ merge(x[1...k], y[2...l])
```

Ordnen Sie die jeweilgen Algorithmen ihrem Pseudocode zu. Sind die Algoithmen stabil, sortieren sie inplace? Was ist die Laufzeit der Algorithmen, und wie viel zusätzlichen Speicherplatz benötigen Sie? Um diese Fragen zu beantworten können Sie den gegebenen Pseudocode betrachten. Sie können sich aber auch fragen, ob es einfache Modifikationen am Psudeocode gibt, so dass die Algorithmen z.B. stabil werden. Tragen Sie Ihre Ergebnisse in die folgende Tabelle ein.

	Stable?	Inplace?	Running Time?	Space Complexity?	Parallelisierbar?
Selection Sort					
Insertion Sort					
Bubble Sort					
Merge Sort					
Heap Sort					
Quick Sort					

Aufgabe 2: Anwendungen des Sortierens (keine Punkte)

Sortieren ist hilfreich bei der Lösung verschiedenster algorithmischer Probleme. Welche Probleme fallen Ihnen ein, bei denen sortieren Teil einer Lösung ist? Betrachten Sie z.B. Algorithmen aus vorherigen Kapiteln der Vorlesung.

Aufgabe 3: Finde meinen Mode (keine Punkte)

Der Mode einer Sequenz von Zahlen ist diejenige Zahl, die am öftesten in der Sequenz vorkommt (`https://en.wikipedia.org/wiki/Mode_(statistics)`). Beschreiben Sie einen Algorithmus mit Laufzeit $O(N \log N)$, der den Mode eines Arrays mit N Elementen findet.