**Ludwig-Maximilians-Universität München**
**Institut für Informatik**
Prof. Dr. Thomas Seidl
Janina Sontheim, Florian Richter

München, 23.05.2019

# Algorithmen und Datenstrukturen

SS 2019

## Übungsblatt Global 3: Sortieren

**Aufgabe Global 3-1** *Knobelei: Pferderennen*

Gegeben sind 25 Pferde, von denen die drei Schnellsten ermittelt werden sollen. Dabei gelten folgende Annahmen:

- Jedes Pferd gallopiert stets konstant mit seiner maximalen Geschwindigkeit.
- Kein Pferd ist genauso schnell wie ein anderes.
- Es gibt eine Rennbahn mit genau 5 Spuren.
- Es gibt keine Stoppuhren oder ähnliche Hilfsmittel.

Wie viele Rennen sind nötig, um die drei schnellsten Pferde zu bestimmen?

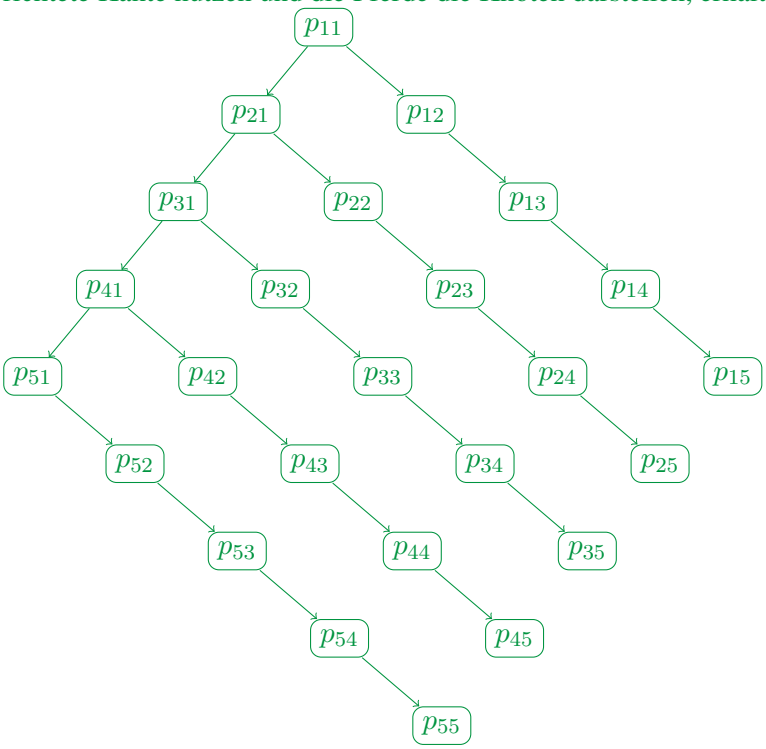
**Lösungsvorschlag:**

1. In den ersten 5 Rennen lassen wir jedes Pferd genau einmal starten. Wir sortieren nach jedem Rennen die 5 teilgenommenen Pferde nach ihrer Platzierung in einer Liste. Das schnellste Pferd links, das langsamste rechts.

2. Im sechsten Rennen treten die 5 schnellsten Pferde jeder der 5 Gruppen gegeneinander an. Wir sortieren die Pferde als Matrix, sodass die Gruppenlisten die Zeilen bilden und die Reihenfolge der Zeilen der Platzierung des Gruppenbesten entspricht. Die Gruppe mit dem schnellsten Pferd ist Zeile 1, die Gruppe mit dem langsamsten Pferd ist Zeile 5.

   | | | | | |
   |---|---|---|---|---|
   | $p_{11}$ | $p_{12}$ | $p_{13}$ | $p_{14}$ | $p_{15}$ |
   | $p_{21}$ | $p_{22}$ | $p_{23}$ | $p_{24}$ | $p_{25}$ |
   | $p_{31}$ | $p_{32}$ | $p_{33}$ | $p_{34}$ | $p_{35}$ |
   | $p_{41}$ | $p_{42}$ | $p_{43}$ | $p_{44}$ | $p_{45}$ |
   | $p_{51}$ | $p_{52}$ | $p_{53}$ | $p_{54}$ | $p_{55}$ |

   Wir können die Matrix auch als Binärbaum schreiben. Wenn wir die „ist schneller“-Relation als gerichtete Kante nutzen und die Pferde die Knoten darstellen, erhalten wir:

   Das ist im wesentlichen ein Max-Heap. Wir haben schon einige Informationen zur Reihenfolge eingebettet, aber es ist nicht vollständig, was aber auch nicht nötig ist.

3. Wir wissen bereits, dass Pferd $p_{11}$ das schnellste Pferd ist. Wir suchen nun Kandidaten für Platz 2 und 3.

4. Kandidat Platz 2: Entweder $p_{12}$ oder $p_{21}$. Alle anderen Pferde sind langsamer als diese beiden wegen Transitivität.

5. Kandidat Platz 3: Hierfür in Frage kommen: Das langsamere von $p_{12}$, $p_{21}$. Zusätzlich $p_{31}$, $p_{22}$ und $p_{13}$. Alle Pferde, die weiter von der Wurzel im MaxHeap entfernt sind als 3 können ignoriert werden.

6. Da $p_{11}$ nicht mehr antreten muss, können im siebten Rennen die Plätze 2 und 3 ermittelt werden, indem die Pferde $p_{31}$, $p_{21}$, $p_{22}$, $p_{12}$, $p_{13}$ antreten.

**Aufgabe Global 3-2** *Vergleich von Sortierverfahren*

Diskutieren Sie die Unterschiede zwischen den in der Vorlesung vorgekommenen Sortierverfahren (z.B. bezüglich theoretischer Laufzeiten, Worst-Case Instanzen, Stabilität und sinnvoller Anwendungsgebiete). Geben Sie insbesondere die Schwächen der Algorithmen an und machen Sie einen Vorschlag, wie potentielle Verbesserungen aussehen könnten.

**Lösungsvorschlag:**

| | BubbleSort | SelectionSort | InsertionSort | MergeSort | QuickSort | HeapSort |
|---|---|---|---|---|---|---|
| Best-Case | $O(n)$ | $O(n^2)$ | $O(n)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Avg-Case | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ |
| Worst-Case | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n \log n)$ | $O(n^2)$ | $O(n \log n)$ |
| Stabilität | ja | nein | ja | ja | nein | nein |

- BubbleSort: Bester Fall - und da auch relativ effizient - ist eine vorsortierte Liste. Der ungünstigste Fall tritt daher ein, wenn die Liste absteigend sortiert ist. Sinnvolle Anwendungsgebiete sind kurze Listen, die mit höherer Wahrscheinlichkeit (partiell) vorsortiert sind. Eine große Schwäche ist das langsame Tempo, mit dem kleine Schlüssel von hinten nach vorne bewegt werden. Zwei Ansätze funktionieren gegen diesen Mechanismus: Die Iterationsrichtung kann nach jedem inneren Schleifendurchlauf alterniert werden. Damit bubblen kleine und große Schlüssel gleichmäßig zu den Rändern (CocktailSort/ShakerSort). Die zweite Möglichkeit ist der Vergleich und Tausch weit auseinanderliegender Paare, was ebenfalls langsame Schlüssel beschleunigt (CombSort).

- SelectionSort: Kommt mit sehr wenig Schlüsselbewegungen aus, da $n$ mal das Minimum gesucht und dann getauscht wird. Daher ist dieser Algorithmus genau dann effizient, wenn Vertauschungen teuer und Vergleiche günstig sind. Die größte Schwäche liegt damit in der hohen Anzahl der Vergleiche, auch im Best-Case. Eine mögliche Verbesserung kann in einer Vorbereitung der Liste liegen, die Elemente grob vorsortieren. Damit sind nur noch kleinere Teillisten zu betrachten. Das Verfahren lässt sich auch gut parallel durchführen, in dem von beiden Enden beginnend sortiert wird.

- InsertionSort: Ist theoretisch genauso gut wie BubbleSort und ist optimal im Best-Case. Auch hier ist der Worst-Case eine umgekehrt sortierte Liste. In einem Array werden die Verschiebungen sehr zahlreich, daher kann eine andere Datenstruktur sinnvoller sein. Ist gut für kleine Schlüsselfolgen geeignet, ebenso für unvollständige Folgen, die schon während einer Übertragung sortiert werden sollen. Um die Anzahl der Schlüsselvergleiche weiter zu reduzieren, kennen wir bereits die Variante mit einer Wächterposition am Anfang des Arrays.

- MergeSort: Best-Case identisch zu Worst-Case. Gut geeignet für verkettete Listen. Eine direkte Sortierung von Arrays ist kompliziert zu implementieren. Wie andere rekursive Verfahren (Quick- und HeapSort) ist der Overhead durch Funktionsaufrufe für kleine n nachteilig. Daher ist eine sinnvolle Verbesserung, die Rekursion ab einer bestimmten Schwelle durch ein einfaches Verfahren zu ersetzen.

- QuickSort: Hochgradig abhängig von der Pivot-Strategie. Im Worst-Case degeneriert der Aufruf-Baum zu einer Liste, dann wie BubbleSort. Durch randomisierte Pivots kann der Worst-Case in der Regel sehr zuverlässig vermieden werden (Alternativ: Einmaliges Mischen der Liste zu Beginn). Das nutzen mehrerer Pivots führt nicht zu einer Verbesserung der theoretischen Laufzeit, kann aber für Praxisszenarien eine Beschleunigung liefern (Dual-Pivot). QuickSort hat ebenfalls eine schlechtere Laufzeit bei Listen mit vielen Duplikaten. Mit QuickSelect lässt sich der Median eines Arrays in O(n) finden. Damit wird die Komplexität für Quicksort zwar nicht besser, da immer noch log n viele Aufrufe benötigt werden, allerdings können viele Worst-Cases eliminiert werden.

- HeapSort: Orientiert an SelectionSort. Der Speicherplatz von HeapSort ist konstant. QuickSort ist zwar meist schneller, benötigt aber für seine Operationen eine zusätzliche Struktur (z.B. Stack). Für günstige Vergleiche für die Schlüssel ist HeapSort QuickSort überlegen. HeapSort ist mit vorsortierten Feldern etwas ineffizienter, da die schon sortierten Elemente in die Wurzel des Heaps bubblen und anschließend wieder an das Ende vertauscht werden. Eine Variante dreht den Heap herum (SmoothSort).