Erläuterungen zur Laufzeitbetrachtung der Sortierprozeduren Quicksort und Heapsort

1 Quicksort

Die Laufzeit von Quicksort hängt essentiell vom Partitionierungsschritt und der Wahl des Pivotelements ab, da dies über die Anzahl der rekursiven Aufrufe entscheidet. Wir betrachten für Quicksort den günstigsten und den ungünstigsten Fall getrennt voneinander. Sei n die Länge des betrachteten Arrays.

Günstigster Fall Im besten Fall wird für jeden Partitionierungsschritt der Median des betrachteten Teil des Arrays, d.h. der Array-Einträge zwischen `ug` und `og`, als Pivotelement gewählt. Sei k die Größe des jeweiligen Teilarrays, d.h $k = og - ug$. In diesem Fall wird der Abschnitt bei der Partitionierung halbiert. Der rekursive Aufruf erfolgt dann auf Arrays halber Länge, genauer $\frac{k-1}{2}$. Für den Aufrufbaum ergibt sich daher ein balancierter Baum der Tiefe von $O(\log n)$.

Bei der Partitionierung wird mit Hilfe von `left` und `right` der Abschnitt von `ug` bis `og-1` abgelaufen. Im besten Fall sind keine Vertauschungen notwendig, allerdings wird jedes Element (genau) einmal mit dem Pivot verglichen. Damit benötigt man für diesen Schritt des Algorithmus $O(k)$ Operationen.

Auf jedem Level wird der Scan für (fast) alle Elemente durchgeführt (Ausnahme sind die bisher ermittelten Pivotelemente). Insgesamt ergibt sich als Größenordnung für Laufzeit im günstigsten Fall daher $O(n \log n)$.

Ungünstigster Fall Im ungünstigsten Fall wird das maximale (oder minimale) Element in jedem Partitionierungsschritt gewählt. Für den rekursiven Aufruf ergibt es sich dann, dass einmal `quicksort` auf einem Teilarray der Größe 0 und einmal auf einem Teilarray der Größe $k - 1$ aufgerufen wird. Damit ist der Aufrufbaum zu einer Liste degeneriert und hat eine Tiefe von $O(n)$.

Auch im ungünstigsten Fall wird beim Scan mit `left` und `right` der Abschnitt von `ug` bis `og-1` einmal abgelaufen. Dabei wird im ungünstigsten Fall jedes Element (genau einmal) mit dem Pivot verglichen und mit einem anderen Element vertauscht. Damit benötigt man auch im ungünstigsten Fall für diesen Schritt des Algorithmus $O(k)$ Operationen.

Auf jedem Level wird der Scan auch im ungünstigen Fall für (fast) alle Elemente durchgeführt. Insgesamt ergibt sich als Größenordnung für Laufzeit im ungünstigsten Fall daher $O(n * n)$, also $O(n^2)$.

2 Heapsort

Um die Laufzeit von Heapsort zu analysieren, betrachten wir die beiden Schritte getrennt voneinander

```
static void sort(DataSet[] f) {
  int size = f.length;

  //  Schritt 1: Herstellen der Heap-Bedingung
  for (int i = size/2 - 1; i >= 0; i--) {
    sink(f, size, i);
  }

  //  Schritt 2: Sortieren
  while (size > 0) {
    size--;
    swap(f, 0, size);
    sink(f, size, 0);
  }
}
```

Herstellen der Heap-Bedingung Um die Analyse zu vereinfachen, gehen wir davon aus, dass das Array $n = 2^{h+1} - 1$ viele Elemente enthält. In diesem Fall ist der Heap-Baum vollständig, d.h. das letzte Niveau des Baums ist "voll". Für die einzelnen Niveaus des Baums gilt dann, dass Niveau 0 genau einen Knoten enthält (das Wurzelelement des Baums), Niveau 1 hat 2 Knoten, Niveau 2 hat 4 Knoten, etc., Niveau h enthält schließlich 2^h Knoten.

Die Anzahl der Vergleiche und Vertauschungen in der `sink`-Prozedur ist abhängig von der Höhe des Baumes. Für einen Baum der Höhe h sind dies maximal $\log h$ viele Vergleiche und Vertauschungen.

Beim Herstellen der Heap-Bedigungen werden die Knoten auf dem untersten Niveau h nicht versickert, da es sich dabei schon um Blätter handelt. Ein Niveau darüber befinden sich 2^{h-1} Knoten, die jeweils ein Niveau tiefer versickern können. Ein weiteres Niveau darüber befinden sich 2^{h-1} Knoten, die jeweils zwei Niveau tiefer versickern können. Generell gilt, dass für Niveau j von unten gezählt 2^{h-j} Knoten sind, die (max.) j Niveau versickern können. Für den ungünstigsten Fall ergibt sich somit:

$$T_1(n) = \sum_{j=0}^{h} j 2^{h-j} = \sum_{j=0}^{h} j \frac{2^h}{2^j} = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h 2 = 2^{h+1}$$

Aus $n = 2^{h+1} - 1$ folgt damit

$$T_1(n) \leq n + 1 \in O(n)$$

Sortieren Beim zweiten Schritt wird das aktuelle Maximum des Heaps aus dem Heap entfernt, indem es mit dem letzten Element des Heaps vertauscht wird. Dieses Element wird anschliessend im Heap versickert.

Dabei werden alle n Elemente einmal an das Ende getauscht. Die anschliessende Versickerung umfasst jeweils maximal $h + 1$ Operationen, da die Elemente maximal bis auf die Blattebene versickern.

Damit ergibt sich für den zweiten Schritt

$$T_2 \in O(n \log n)$$

Zusammenfassung Bezüglich der Größenordnung dominiert die Laufzeit des zweiten Schritts die des ersten Schritts, da $O(n) \subset O(n \log n)$.

Es ergibt sich daher für die Laufzeit des gesamten Heapsort-Algorithmus, dass er in der Größenordnung $O(n \log n)$ ist.