Sortieren

Sortieren ist eine der häufigsten und wichtigsten Aufgaben, die von Algorithmen auszuführen sind.

Die üblichen Sortieralgorithmen sind grundsätzlich *vergleichsbasiert*, d.h. dass einerseits die Grundoperation, die zum Erfolg führt, der Vergleich zweier Elemente ist – aber auch, dass zur Messung des Aufwands nur die Anzahl der Vergleichsoperationen heranzuziehen ist.

Für vergleichsbasierte Sortieralgorithmen gibt es eine beweisbare untere Schranke für den Aufwand:

Jeder vergleichsbasierte Sortieralgorithmus, der n Elemente sortiert, braucht im worst case $\Omega(n \log n)$ Vergleiche.

Beweis der unteren Schranke

Die entscheidende Beobachtung ist folgende:

Zu Beginn eines Ablaufs jedes beliebigen Sortieralgorithmus, der eine Folge von n Elementen sortieren soll, gibt es $n!$ mögliche Vorsortierungen, die der Algorithmus unterscheiden können muss.

Jeder Vergleich teilt die Menge der möglichen Vorsortierungen in zwei Teile auf, von denen durch das Vergleichsergebnis einer – im worst case bei ungleicher Verteilung der größere – zur weiteren Verarbeitung ausgewählt wird.

Also kann der Algorithmus im worst case frühestens nach $\log(n!)$ vielen Vergleichen die komplette und korrekte Sortierung finden. Dass $\log(n!)$ in $\Omega(n \log n)$ ist ein bekannter Sachverhalt.

Wir werden das im 2. Teil der Vorlesung (Diskrete Strukturen) auch noch beweisen.

Quicksort

Den Quicksort-Algorithmus haben wir schon vorgestellt.
Die zentrale Idee: Wähle ein *Pivot-Element* aus dem zu sortierenden Array, teile in zwei Teile, die rekursiv sortiert werden.
Einer dieser Teile beinhaltet die Elemente, die kleiner als das Pivot-Element sind, der andere die, die größer sind.

Bei geschickter Implementierung landet das Pivot-Element jeweils direkt an seinem Platz, die übrigen $n - 1$ Elemente werden rekursiv weiter verarbeitet, jedes in einem der beiden Teile.

Dass die worst-case Laufzeit des Quicksort-Algorithmus in jedem Fall quadratisch ist, haben wir uns schon überlegt.
Hieran ändert auch eine geschickte Strategie zur Pivot-Wahl nichts – obwohl in der Praxis die „Median-aus-drei“ Strategie sehr gut funktioniert.

Quicksort-Varianten

Die Median-aus-drei Variante funktioniert so:

Wenn für den Teil $a[links, \ldots, rechts]$ ein Pivot-Element festzulegen ist, wählt man aus drei Elementen – etwa $a[links]$, $a[rechts]$ und $a[(links + rechts)/2]$ – das der Größe nach mittlere.

Es ist ein einfaches Rechenbeispiel, nachzuweisen, dass diese Strategie im worst case nach wie vor eine quadratische Anzahl Vergleiche benötigt.

Um ein „gutes“ Pivot-Element in jedem Schritt zu erzwingen, muss man im noch zu bearbeitenden Teil-Array $a[links, \ldots, rechts]$ eine komplette Medianberechnung durchführen. Diese ist allerdings vergleichsweise aufwändig. Daher findet diese Variante in der Praxis kaum Verwendung.

Quicksort: average case Analyse

Wie kann man den Quicksort-Algorithmus in seinem Durchschnittsverhalten analysieren?
Wir gehen von einer Standardvariante aus, die bei einem Aufruf mit dem Teil-Array $a[\ell, \ldots, r]$ (hier steht ℓ für *links* und r für *rechts*) ein zufälliges Pivot-Element auswählt, wobei Gleichverteilung unter den in Frage kommenden Elementen vorausgesetzt wird.

Bei jeder Aufteilung von m Elementen ist eines von ihnen das Pivot-Element, alle anderen (also $m-1$ Elemente) werden mit dem Pivot verglichen. Andere Vergleiche gibt es nicht.

Nun stellt sich die Frage, wie viele Vergleiche insgesamt im Ablauf der Quicksort-Prozedur auf dem Gesamtarray $a[1, \ldots, n]$ durchschnittlich ausgeführt werden müssen.

Wahrscheinlichkeit eines Vergleichs

Nehmen wir an, dass die sortierte Reihenfolge des vorliegenden zu sortierenden Feldes $a[1, \ldots, n]$ die Folge $s_1, \ldots, s_n$ ist. Also

$$s_1 < s_2 < \cdots < s_{n-1} < s_n$$

Wie wahrscheinlich ist es, dass für $i < j$ das Element s_i irgendwann im Ablauf unseres Algorithmus mit s_j verglichen wird?

Offensichtlich ist ein solcher Vergleich nur möglich, solange s_i und s_j zusammen in einem zu sortierenden Teil-Array liegen.

Wird ein Pivot s_μ mit $\mu < i$ oder $\mu > j$ gewählt, ändert sich nichts an der Situation – s_i und s_j bleiben im gleichen Teil-Array.
Wird ein Pivot s_μ mit $i < \mu < j$ gewählt, wandern die beiden in verschiedene Teil-Arrays und werden nie miteinander verglichen.
Nur bei s_i oder s_j als Pivot werden die beiden verglichen.

Resultat für average case

Damit ist klar: die Wahrscheinlichkeit, dass s_i und s_j im Verlauf des Sortiervorgangs verglichen werden, ist $\frac{2}{j-i+1}$.

Hieraus ergibt sich die erwartete Anzahl von Vergleichen, indem man für alle in Frage kommenden Vergleiche – also für alle Paare (i,j) mit $1 \leq i < j \leq n$ – die Wahrscheinlichkeiten dieser Vergleiche aufsummiert.

Der Erwartungswert für die Anzahl Vergleiche $V(n)$ ergibt sich damit zu

$$V(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \leq \sum_{i=1}^{n-1} 2 \cdot (H_n - 1) \leq 2n \cdot (H_n - 1)$$

$H_n - 1$ kann durch $\ln n$ beschränkt werden, daher erhalten wir:

$$V(n) \leq 2n \ln n < 1.39\, n \log n.$$

Heapsort

Ein weiteres schnelles In-place Sortierverfahren ist *Heapsort.*
Dabei wird erst das Feld $a[1, \ldots, n]$ in einen Heap verwandelt, d.h. es erfüllt für alle i die Bedingungen

$$a[i] \leq a[2i] \text{ und } a[i] \leq a[2i+1]$$

(natürlich nur, falls $2i \leq n$ bzw. $2i+1 \leq n$)

Die bildliche Vorstellung ist hier ein Binärbaum mit Wurzel $a[1]$, als Nachfolgeknoten zur Wurzel: $a[2]$ und $a[3]$, etc.

Die Tiefe dieses Baums ist $\lceil \log n \rceil$, und im Baum ist jeder Knoten im Wert kleiner (oder gleich) beiden Nachfolgeknoten.

Also befindet sich das Minimum eines Heaps immer in $a[1]$.

Damit wird so sortiert:

Tausche $a[1]$ mit $a[n]$, betrachte ab jetzt nur noch $a[1, \ldots, n-1]$.
„Repariere“ den Heap bei $a[1]$ und fahre entsprechend fort.

Die Einsink-Prozedur reheap

Um einen Heap zu erstellen oder eine „gestörte“ Heapbedingung (bei $a[1]$) zu *reparieren*, benötigen wir die Einsink-Prozedur, die auch reheap genannt wird. Sie wird rekursiv so realisiert:

```
reheap(i):
  if 2i + 1 > n then                          ⟸ Sonderfall!
    if 2i = n then
      if a[i] > a[2i] then swap(a[i], a[2i])
  else                                        ⟸ Normalfall!
    if a[2i] ≤ a[2i + 1] then
      if a[i] > a[2i] then
          swap(a[i], a[2i]); reheap(2i)
    else
      if a[i] > a[2i + 1] then
          swap(a[i], a[2i + 1]); reheap(2i + 1)
```

Die Heap-Erstellung

Wir lösen zunächst eine kleine Rechenaufgabe:

Es gilt $\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$.

Das folgt aus $\sum_{i=0}^{k} \frac{i}{2^i} = 2 - \frac{k+2}{2^k}$.

Die rote Gleichung folgt mit Induktion – den Beweis hierzu machen wir an der Tafel.

Sie ist aber auch eine gute Übungsaufgabe...

Wir entnehmen dem Pseudo-Code der Einsink-Prozedur, dass beim Einsinken eines Elements von Höhe h maximal $2h$ Vergleichsoperationen ausgeführt werden müssen.

Die Blätter (Höhe 0) müssen nicht einsinken. Auf Höhe 1 gibt es maximal $n/2$ viele Elemente, auf Höhe 2 maximal $n/4$ viele, etc.

Also ist obere Schranke für Anzahl Vergleiche der Heaperstellung:

$$2 \cdot 1 \cdot \tfrac{n}{2} + 2 \cdot 2 \cdot \tfrac{n}{4} + 2 \cdot 3 \cdot \tfrac{n}{8} + \ldots \leq 2n \cdot \sum_{i=0}^{\infty} \frac{i}{2^i} = 4n \in O(n).$$

Aufwand für Heapsort

Im wesentlichen muss nach Heap-Erstellung jedes der n Elemente einmal von ganz oben nach ganz unten sinken, d.h. man braucht für den Sortier-Teil maximal $2n \log n$ Vergleiche.

Damit gilt für die Gesamtzahl $V(n)$ der Vergleiche bei Heapsort:

$$V(n) \leq 2n \log n + O(n)$$

Die gute Nachricht: Diese Schranke gilt für den worst case, d.h. anders als bei Quicksort erreichen wir hier *immer* die bis auf Konstanten optimale Laufzeit $O(n \log n)$.

Die schlechte Nachricht: Die Konstante 2 vor dem $n \log n$ gilt auch im average case, d.h. im Mittel ist dieses Verfahren dem Quicksort-Verfahren deutlich unterlegen, also praktisch nicht konkurrenzfähig.