5.7 Sortieren in linearer Zeit

In Satz 5.4 haben wir gezeigt, dass das Sortieren von n Objekten nicht schneller als in $\Omega(n \log n)$ möglich ist - vorausgesetzt, man kann dafür keine Zusatzinformation einsetzen und darf Objekte nur paarweise vergleichen (und ggf. vertauschen).

In vielen Situationen hat man aber zusätzliche Informationen!
Das wollen wir hier analysieren.

5.7.1 Countingsort

Im einfachsten Fall sind die n Objekte Zahlen aus dem Bereich $1, \dots, k$.
Wenn nun k klein ist (insbesondere $k \in O(n)$), dann gibt es einen einfachen Ansatz, der ganz ohne Vergleiche auskommt:

(1) Lege einen „Zählarray" $C[0], \dots, C[k]$ an

(2) Gehe den zu sortierenden Array $A[1], \dots, A[n]$ durch; für jedes Objekt $A[j]$ erhöhe den entsprechenden Eintrag in C, d.h. $C[A[j]]$ um 1.

(3)

Für $i = 1 \dots k$:
Schreibe jeweils i in die nächsten $C[A[i]]$ Einträge von $B[i]$.

Am Ende liefert $B[1], \dots, B[n]$ eine sortierte Form von $A[1], \dots, A[n]$.

Im Detail:

ALGORITHMUS 5.14

INPUT : Array $A[1], \dots, A[n]$ mit Schlüsselwerten $\in \{1, \dots, k\}$

OUTPUT : Sortierte Kopie $B[1], \dots, B[n]$ von $A[1], \dots, A[n]$

```
COUNTING-SORT (A, B, k)
1.  FOR (i=1) TO k
    1.1 DO C[i] := 0          // Zählerarray initialisieren
2.  FOR (j=1) TO n
    2.1 DO C[A[j]] := C[A[j]]+1   // Anzahl Elemente mit Wert
                                  // A[j] um 1 erhöhen
3.  FOR (i=1) TO k
    3.1 DO C[i] := C[i] + C[i-1]  // Anzahl Elemente mit Wert
                                  // höchstens i zählen
4.  FOR (j=n) DOWNTO 1
    4.1 DO B[C[A[j]]] := A[j]     // Element A[j] an richtige
    4.2    C[A[j]] := C[A[j]]-1   // Stelle in B schreiben.
```

SATZ 5.15

Für n Objekte mit Schlüsseln $\in \{1, \dots, k\}$ benötigt COUNTING-SORT die Zeit $O(n+k)$. Insbesondere benötigt man für $k \in O(n)$ die Zeit $\Theta(n)$.

5.7.2 Radixsort

Idee: Sortiere nicht gleich nach den Schlüsselwerten, sondern nach den Ziffern der Schlüssel!

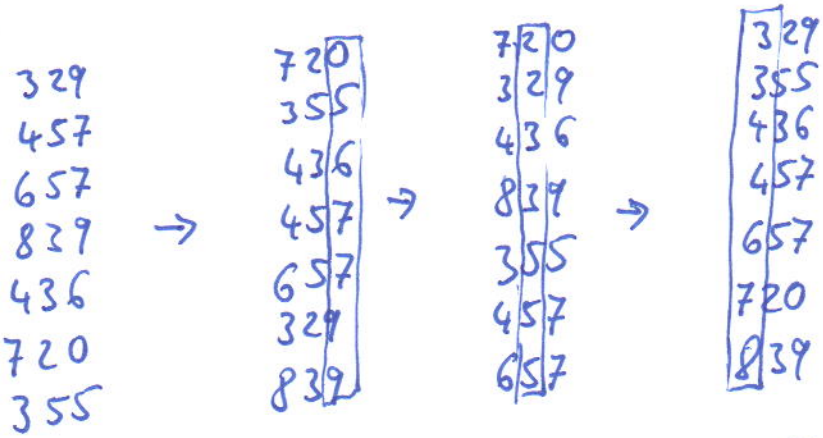
(Früher sehr häufig bei Lochkarten eingesetzt.)

Naiver Ansatz: Sortiere nach größter Ziffer, dann nach zweiter Ziffer, ...

Problem: Entweder benötigt man viele temporäre Zwischenmengen - oder die Vorsortierung geht kaputt!

Besserer Ansatz: Sortiere nach letzte Ziffer, dann nach vorletzter Ziffer, ...

Beispiel:



Wichtig: Jeweilige Sortierung darf Reihenfolge gleichwertiger Ziffern nicht verändern!

DEFINITION 5.16 (Stabilität)

Ein Sortierverfahren heißt stabil wenn gleiche Werte nach dem Algorithmusdurchlauf in der gleichen Reihenfolge sind wie vorher.

Satz 5.17 (Stabilität)

Mergesort und Countingsort sind stabil, Quicksort je nach Umsetzung der Pivotregel.

Beweis: Selbst!

Damit:

Algorithmus 5.18

Input: n Zahlen mit je d Ziffern, die k verschiedene Werte annehmen können, A[1], ..., A[n]

Output: Sortierter Array

Radik-Sort (A, d)

1 FOR (i=1) TO d

1.1 DO Countingsort für A auf Stelle i

Satz 5.19

Radixsort liefert ein korrektes Ergebnis in Zeit $\Theta(d(n+k))$.

Beweis:

Für je zwei Zahlen muss am Ende die Reihenfolge stimmen; diese entspricht der Ordnung der höchsten Ziffern, die verschieden sind. Die Sortierung nach diesen Ziffern liefert also die richtige Reihenfolge, die sich wegen der Stabilität nicht mehr ändert.

Die Laufzeit folgt aus Satz 5.15. □

5.7.3 Spaghettisort

Nicht ganz ernst gemeint!

Algorithmus 5.20

Input: n Zahlen $A[1], \ldots, A[n]$
Output: Sortierter Array

1. FOR $(i=1)$ TO n
   1.1 DO (schneide ein Spaghetto der Länge $A[i]$ zurecht)
2. Stoße alle Spaghetti auf den Tisch
3. FOR $(i=1)$ TO n
   3.1 DO (wähle die längste verbleibende Nudel)

„Analoger Algorithmus" – nicht digital!

„Satz" 5.21

Algorithmus 5.20 sortiert in Linearzeit!

Beweis: Scheint klar!

Problematisch: Rechnermodell, Rechenoperationen, Speicherplatz, Ausführbarkeit (und Genauigkeit!) für große n...

5.8 Parallelles Sortieren

Idee: Nicht nur einzelne Paare können verglichen werden, sondern viele (disjunkte) Paare zugleich!

Dazu zunächst sequentiell:

ALGORITHMUS 5.22 (Bubblesort)

INPUT: Array $A[1], \ldots, A[n]$

OUTPUT: Sortierter Array

```
1   FOR (i=1) TO n
  1.1  DO FOR (j=n) DOWNTO i+1
    1.1.1  DO IF (A[j] < A[j-1])
      1.1.1.1  Vertausche A[j] und A[j-1]
```

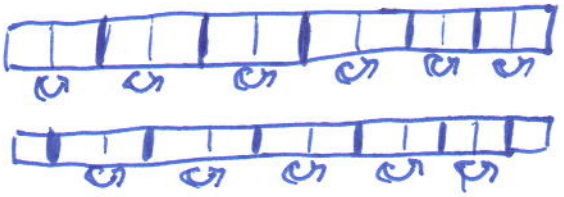
SATZ 5.23

Bubblesort ist ein korrektes, stabiles Sortierverfahren.
Die Laufzeit ist $\Theta(n^2)$.

Beweis: Einfach!

Also nicht so toll...

Aber: Lässt sich gut parallelisieren!



ALGORITHMUS 5.24 (Paralleles Bubblesort / odd-even-sort)

INPUT: n Zahlen $A[1], \dots, A[n]$

OUTPUT: Sortierter Array

1 FOR $(i=1)$ TO n

1.1 Vergleiche und ggf. tausche gerade Paare parallel

1.2 Vergleiche und ggf. tausche ungerade Paare parallel

SATZ 5.25

Paralleles Bubblesort sortiert die Zahlen in n parallelen Runden.
Die Gesamtzahl der Vergleiche ist $\Theta(n^2)$.

Beweis: Selbst!

Das funktioniert auch zwei- und mehrdimensional!

5.9 Unernste Sortierverfahren

ALGORITHMUS 5.26 (Bogosort)

INPUT: Array $A[1], \dots, A[n]$

OUTPUT: Sortierter Array

1 WHILE (Array unsortiert)

1.1 Permutiere zufällig

1.2 Überprüfe Sortiertheit

Satz 5.27

Bogosort hat erwartete Laufzeit $O(n \cdot n!)$.

Beweis: Überprüfen dauert $O(n)$, die erwartete Zahl der Runden ist $O(n!)$, weil die Wahrscheinlichkeit in jeder Runde $\frac{1}{n!}$ ist, die sortierte Reihenfolge zu erhalten.

Noch schlimmer:

Algorithmus 5.28 (Bogobogosort)

INPUT: n Zahlen $A[1], \ldots, A[n]$
OUTPUT: Sortierter Array

1. $i := 2$
2. WHILE $(i \leq n)$ DO
   - 2.1 wähle Zufallspermutation der Elemente $A[1], \ldots, A[i]$
   - 2.2 IF (Permutation ist sortiert)
     - 2.2.1 $i := i+1$
   - 2.3 ELSE
     - 2.3.1 $i := 2$

Satz 5.29

Bogobogosort hat erwartete Laufzeit $O\left(n \cdot \prod_{i=2}^{n} i!\right)$.

Beweis:
Jede Permutation hat Wahrscheinlichkeit $\frac{1}{i!}$, sortiert zu sein.
Man braucht eine Glückssträhne für alle $i = 2, \ldots, n$

In einem geeigneten Berechnungsmodell lässt sich das aber trotzdem parallel in Linearzeit implementieren!

ALGORITHMUS 5.30 (Quantum Bogosort)

INPUT: n Zahlen $A[1], \dots, A[n]$
OUTPUT: Sortierter Array in geeignetem Universum

1. Permutiere zufällig ← Quantenbits
2. IF (Permutation unsortiert)
   2.1 Zerstöre Universum

SATZ 5.31

Quantum Bogosort liefert ein Universum, in dem nach $O(n)$ alle Zahlen als sortiert verifiziert sind.

Beweis:
Erfordert:
- Quantenphysikalisches Weltmodell (→ Zufallsereignisse ergeben Parallelwelten)
- Mechanismus zur Zerstörung des Universums in $O(1)$

Rest selbst! (Bitte nicht experimentell...) □

KOROLLAR 5.32

Quantum Bogobogosort liefert ein Universum, in dem nach $O(n^2)$ alle Zahlen als sortiert verifiziert sind.