Script generated by TTT

Title: TÃ¤ubig: GAD (06.06.2013)

Date: Thu Jun 06 12:01:00 CEST 2013

Duration: 49:32 min

Pages: 31

Übersicht

Sortieren schneller als $O(n \log n)$

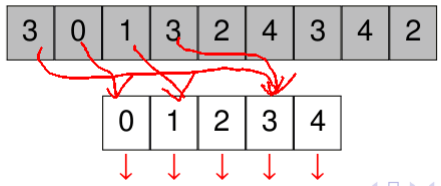
Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
- z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer bzw. das erste Zeichen. Nur bei Gleichheit des Anfangs kommt es auf weitere Ziffern / Zeichen an.

Sortieren schneller als $O(n \log n)$

Buckets

- mit paarweisen Schlüsselvergleichen: nie besser als $O(n \log n)$
- Was aber, wenn die Schlüsselmenge mehr Struktur hat?
- z.B. Zahlen / Strings bestehend aus mehreren Ziffern / Zeichen
- Um zwei Zahlen / Strings zu vergleichen reicht oft schon die erste Ziffer bzw. das erste Zeichen. Nur bei Gleichheit des Anfangs kommt es auf weitere Ziffern / Zeichen an.
- Annahme: Elemente sind Zahlen im Bereich $\{0, \ldots, K-1\}$
- Strategie: verwende Feld von K Buckets (z.B. Listen)



Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```


Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n+K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n+K)$

Sortieren schneller als $O(n \log n)$

Buckets

```
Sequence<Elem> kSort(Sequence<Elem> s) {
  Sequence<Elem>[ ] b = new Sequence<Elem>[K];
  foreach (e ∈ s)
    b[key(e)].pushBack(e);
  return concatenate(b);   // Aneinanderreihung von b[0],...,b[k-1]
}
```

Laufzeit: $\Theta(n+K)$ Problem: nur gut für $K \in o(n \log n)$
Speicher: $\Theta(n+K)$

- wichtig: kSort ist stabil, d.h. Elemente mit dem gleichen Schlüssel behalten ihre relative Reihenfolge

$\Rightarrow$ Elemente müssen im jeweiligen Bucket *hinten* angehängt werden

RadixSort

- verwende K-adische Darstellung der Schlüssel
- Annahme:
  Schlüssel sind Zahlen aus $\{0, \ldots, K^d - 1\}$
  repräsentiert durch d Ziffern aus $\{0, \ldots, K-1\}$
- sortiere zunächst entsprechend der niedrigstwertigen Ziffer mit kSort und dann nacheinander für immer höherwertigere Stellen
- behalte Ordnung der Teillisten bei

RadixSort

```
radixSort(Sequence<Elem> s) {
  for (int i = 0; i < d; i++)
    kSort(s,i);   // sortiere gemäß key_i(x)
    // mit key_i(x) = (key(x)/K^i) mod K
}
```


RadixSort

```
radixSort(Sequence<Elem> s) {
  for (int i = 0; i < d; i++)
    kSort(s,i);   // sortiere gemäß key_i(x)
    // mit key_i(x) = (key(x)/K^i) mod K
}
```

Verfahren funktioniert, weil kSort stabil ist:
Elemente mit gleicher i-ter Ziffer bleiben sortiert bezüglich der Ziffern $i-1 \ldots 0$ während der Sortierung nach Ziffer i

Laufzeit: $O(d(n+K))$ für n Schlüssel aus $\{0,\ldots,K^d-1\}$

RadixSort

Beispiel

012	203	003	074	024	017	112

RadixSort

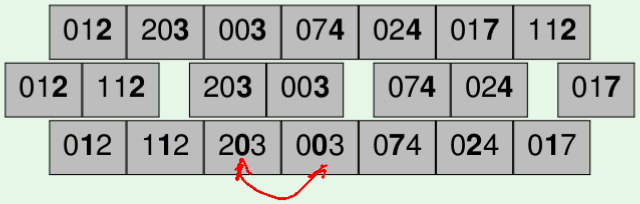
Beispiel

012	203	003	074	024	017	112

012	112	203	003	074	024	017

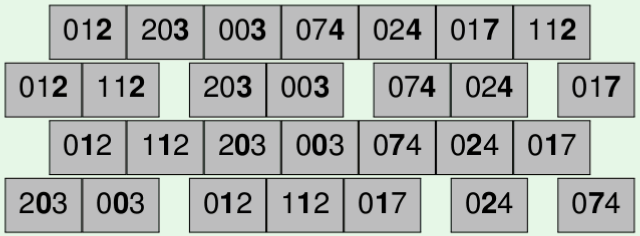
RadixSort

Beispiel



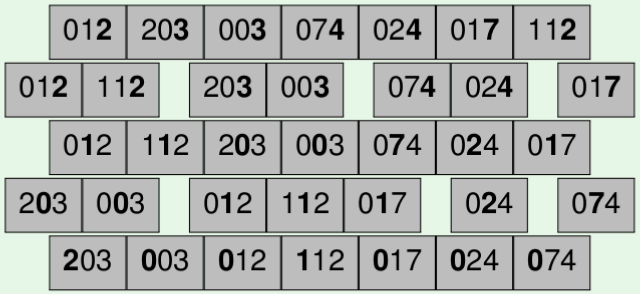
RadixSort

Beispiel



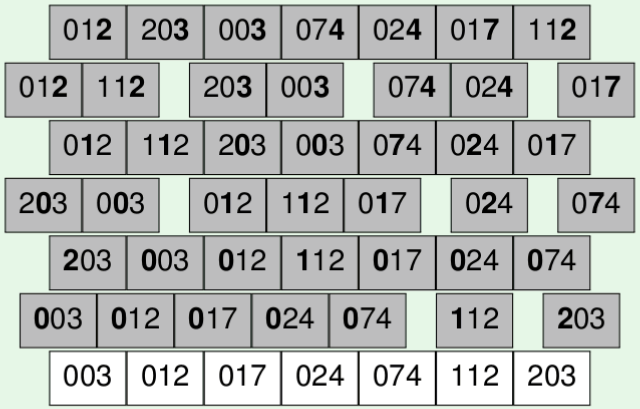
RadixSort

Beispiel



RadixSort

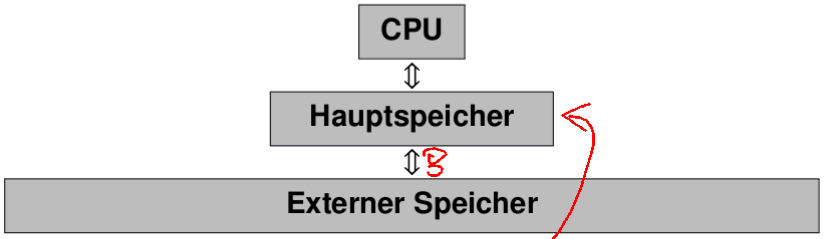
Beispiel



Übersicht

Externes Sortieren

Heutige Computer:



- Hauptspeicher hat Größe M
- Transfer zwischen Hauptspeicher und externem Speicher mit Blockgröße B

Externes Sortieren

Problem:
Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:
Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Externes Sortieren

Problem:
Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:
Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Lösung: Verwende MergeSort

Vorteil: MergeSort verwendet oft konsekutive Elemente (Scanning)
(geht auf Festplatte schneller als Random Access-Zugriffe)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar (sonst z.B. Auffüllen mit maximalem Schlüssel)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade Teilfelder der Größe M in den Speicher (Annahme: $B \mid M$),
- sortiere sie mit einem in-place-Sortierverfahren,
- schreibe das sortierte Teilfeld wieder zurück auf die Festplatte

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar (sonst z.B. Auffüllen mit maximalem Schlüssel)

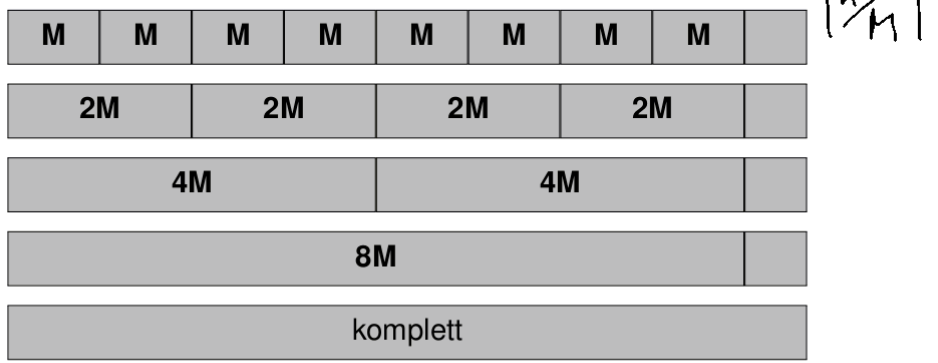
Run Formation Phase:

- Lade Teilfelder der Größe M in den Speicher (Annahme: $B \mid M$),
- sortiere sie mit einem in-place-Sortierverfahren,
- schreibe das sortierte Teilfeld wieder zurück auf die Festplatte
- benötigt n/B Blocklese- und n/B Blockschreiboperationen
- Laufzeit: 2n/B Transfers
- ergibt sortierte Bereiche (Runs) der Größe M

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $O(\lceil \log(n/M) \rceil)$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



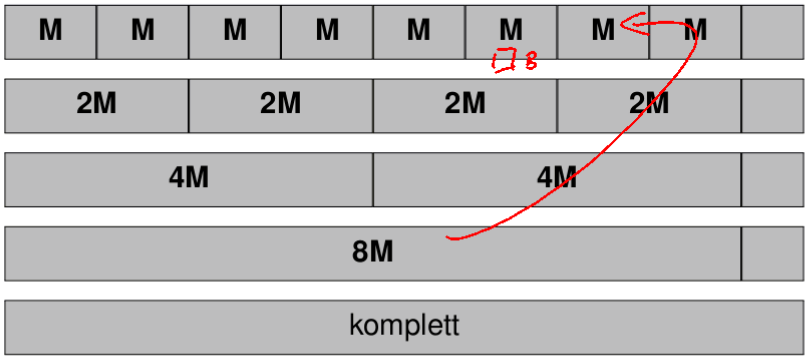
Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer ⇒ neuen Block laden
- Wenn Ausgabepuffer voll ⇒ Block auf Festplatte schreiben und Ausgabepuffer leeren

Externes Sortieren

Merge Phasen

- Merge von jeweils 2 Teilfolgen in $O(\lceil \log(n/M) \rceil)$ Phasen
- dabei jeweils Verdopplung der Größe der sortierten Teile



Externes Sortieren

Problem:
Minimiere Anzahl Blocktransfers zwischen internem und externem Speicher

Anmerkung:
Gleiches Problem trifft auch auf anderen Stufen der Hierarchie zu (Cache)

Externes Sortieren

- Eingabe: großes Feld auf der Festplatte
- Annahme: Anzahl der Elemente n ist durch B teilbar (sonst z.B. Auffüllen mit maximalem Schlüssel)

Run Formation Phase:

- Lade Teilfelder der Größe M in den Speicher (Annahme: $B \mid M$),
- sortiere sie mit einem in-place-Sortierverfahren,
- schreibe das sortierte Teilfeld wieder zurück auf die Festplatte
- benötigt n/B Blocklese- und n/B Blockschreiboperationen
- Laufzeit: 2n/B Transfers
- ergibt sortierte Bereiche (Runs) der Größe M

Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

Merge von zwei Runs

- von jedem der beiden Runs und von der Ausgabesequenz bleibt ein Block im Hauptspeicher (3 Puffer: 2× Eingabe, 1× Ausgabe)
- Anfang: beide Eingabepuffer mit B Elementen (1 Block) laden, Ausgabepuffer leer
- Dann: jeweils führende Elemente der beiden Eingabepuffer vergleichen und das kleinere in den Ausgabepuffer schreiben
- Wenn Eingabepuffer leer $\Rightarrow$ neuen Block laden
- Wenn Ausgabepuffer voll $\Rightarrow$ Block auf Festplatte schreiben und Ausgabepuffer leeren

- In jeder Merge-Phase wird das ganze Feld einmal gelesen und geschrieben

$\Rightarrow (2n/B)(1 + \lceil \log_2(n/M) \rceil)$ Block-Transfers

Multiway-MergeSort

- Verfahren funktioniert, wenn 3 Blöcke in den Speicher passen
- Wenn mehr Blöcke in den Speicher passen, kann man gleich mehr als zwei Runs (k) mergen.
- Benutze Prioritätswarteschlange (Priority Queue) zur Minimumermittlung, wobei die Operationen $O(\log k)$ Zeit kosten
- $(k+1)$ Blocks und die PQ müssen in den Speicher passen

$\Rightarrow (k+1)B + O(k) \leq M$, also $k \in O(M/B)$

- Anzahl Merge-Phasen reduziert auf $\lceil \log_k(n/M) \rceil$

$\Rightarrow (2n/B)\left(1 + \left\lceil \log_{M/B}(n/M) \right\rceil\right)$ Block-Transfers

- In der Praxis: Anzahl Merge-Phasen gering
- Wenn $n \leq M^2/B$: nur eine einzige Merge-Phase (erst M/B Runs der Größe M, dann einmal Merge)

Übersicht