Algorithmen und Datenstrukturen

SoSe 2008 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Algorithmen und Datenstrukturen

Gesamtübersicht

- Organisatorisches / Einführung
- Grundlagen: RAM, $\mathcal{O}$-Notation, Rekursion, Datenstrukturen
- Sortieren
- Wörterbücher und Mengen
- Graphen und Graphalgorithmen

Sortieren:

- Elementare Sortierverfahren
- Sortieren durch Auswahl (selection sort) $\leadsto$ Heapsort
- Quicksort

Repititorium gefällig? Siehe `http://www.linux-related.de/index.html?/coding/sort/sort_main.htm`

Historisches

Wie Heapsort geht Quicksort auf Hoare zurück.
Es zählt “im Mittel” bis heute zu den schnellsten Sortierverfahren.
Im schlimmsten Fall zeigt es jedoch quadratische Laufzeit.
Man kann diesen Fall geeignet durch “rechtzeitiges Ausweichen” auf “richtige” $\mathcal{O}(n \log n)$-Verfahren vermeiden; dieses *Introsort* genannte Verfahren ist in vielen Bibliotheken implementiert.

Quicksort—Die Grundidee

Aufgabe: Sortiere Feld $A[1..n]$ nach dem Teile-und-Herrsche-Prinzip.

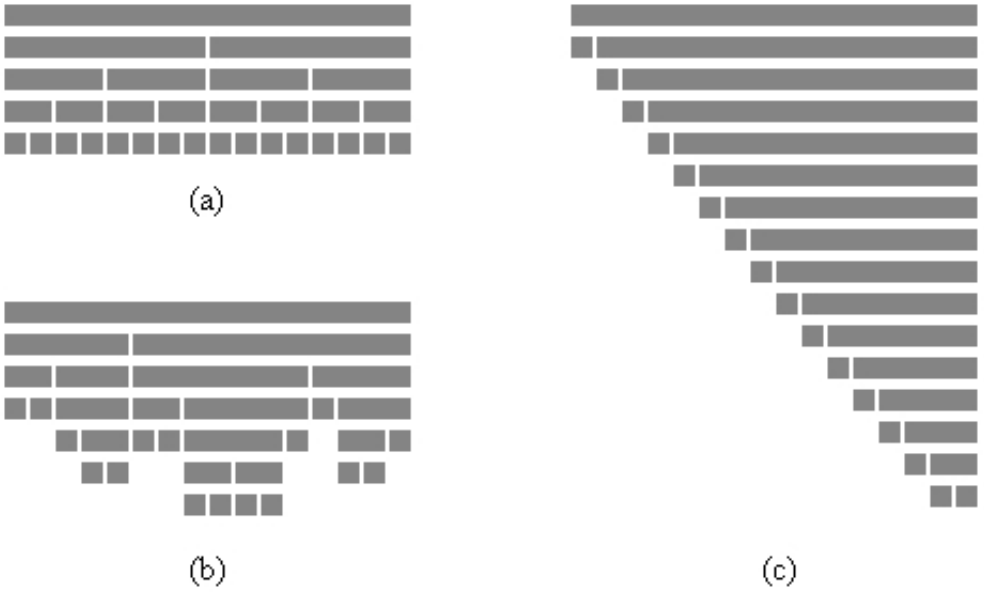
1) Teile A bzgl. $A[1]$ in zwei Teilmengen P, Q:
   $P = \{A[i] \mid 2 \leq i \leq n, A[i] < A[1]\}$
   $Q = \{A[i] \mid 2 \leq i \leq n, A[i] > A[1]\}$.
   Speichere P in $A[1..|P|]$ und Q in $A[n - |Q| + 1..n]$.
2) Wende das Verfahren rekursiv an auf den P- (Anfangs-)Teil von A und auf den Q- (Schluss-)Teil von A.
3) Hänge die Lösungsfelder zusammen

$\rightsquigarrow$ Prozedur quicksort(ℓ, r) (Näheres siehe z.B. matheprisma)

Quicksort—Der schlimmste Fall

Die Kosten eines Aufrufs von quicksort(ℓ, r) sind:
1) die unmittelbaren Aufteilungskosten: $\mathcal{O}(r - \ell + 1)$ Vergleiche,
2) die Kosten der rekursiven Aufrufe.
$QS(n)$ sei max. # Vergleiche, die Quicksort bei einem n-elem. Feld durchführt.
$\rightsquigarrow QS(n) = n + \max_{1 \le j \le n}(QS(j-1) + QS(n-j))$, $QS(0) = 0$.
Mit Induktion sieht man leicht: $QS(n) \le \frac{1}{2} \cdot n \cdot (n+1) = \mathcal{O}(n^2)$.
Hinweis: Bei der Eingabe $1, 2, 3, \ldots, n$ führt Quicksort $\mathcal{O}(n^2)$ Vergleiche durch.

Quicksort—in verschiedenen Fällen



(a) Der Idealfall, (b) ein mittlerer Fall, (c) der schlechteste Fall

Quicksort—Der mittlere Fall

Wir machen jetzt ein paar weitere Annahmen über $A[1..n]$:

1. Alle Elemente sind paarweise verschieden.
2. Jeder der $n!$ vielen möglichen Permutationen ist gleichwahrscheinlich.

Lemma: Die rekursiv zu lösenden Teilprobleme der Größe $k-1$ und $n-k$ erfüllen wiederum die beiden Annahmen.

(ohne Beweis: lästiges Rechnen mit Wahrscheinlichkeiten.)

O.E. im Folgenden: zu sortierende Zahlen aus $\{1, \ldots, n\}$.

Quicksort—Der mittlere Fall

$\overline{QS}(n)$: mittlere Anzahl Vergleiche, die Quicksort bei einem Feld der Größe n durchführt.
$\overline{QS}(n)$ ist Erwartungswert der Zufallsvariablen # Vergleiche
Es gilt (mit gewissen Anfangsbedingungen):
$\overline{QS}(n) = n + \frac{1}{n}\sum_{k=1}^{n}(\overline{QS}(k-1) + \overline{QS}(n-k)) = n + \frac{2}{n}\sum_{k=0}^{n-1}\overline{QS}(k)$.
Multiplizieren mit n und Ersetzen von n durch $(n+1)$ führt auf:
$n \cdot \overline{QS}(n) = n^2 + 2\sum_{k=0}^{n-1}\overline{QS}(k)$ bzw.
$(n+1) \cdot \overline{QS}(n+1) = (n+1)^2 + 2\sum_{k=0}^{n}\overline{QS}(k)$. Der Unterschied:
$(n+1) \cdot \overline{QS}(n+1) - n \cdot \overline{QS}(n) = 2n + 1 + 2\overline{QS}(n)$. $\rightsquigarrow$
$\overline{QS}(n+1) \leq 2 + \frac{n+2}{n+1}\overline{QS}(n) \leq 2 + \frac{n+2}{n+1}(2 + \frac{n+1}{n}(2 + \frac{n}{n-1}\dots)$
$\overline{QS}(n+1) \leq (n+2)(\frac{2}{n+2} + \frac{2}{n+1} + \frac{2}{n}\dots) = \mathcal{O}(n \log n)$,
denn für $H_n = \sum_{k=1}^{n}\frac{1}{k}$ gilt: $H_n = \Theta(\log n)$.(*Harmonische Zahl* VL3)

Abschließende Folgerungen

Wir sollten versuchen, den schlimmsten Fall zu vermeiden.
Eine theoretisch befriedigende Idee ist es, das Vertauschungselement (Pivot) stets zu würfeln (statt immer das Erstbeste zu nehmen).
Für die als CleverQuicksort bekannte Variante, bei der zufällig drei Elemente der Folge gewählt werden und dann das mittlere als Pivot herangezogen wird, konnte man zeigen, dass im mittleren Fall $1.188n\log(n)+\mathcal{O}(n)$ viele Vergleiche benötigt werden, was die entsprechende Schranke des “normalen” Quicksort $(1.286n\log(n)+\mathcal{O}(n))$ deutlich verbessert.

Diskussion der Sortierverfahren

Wir haben bis jetzt zwei Verfahren kennengelernt, die eine Laufzeit von $\mathcal{O}(n \cdot \log(n))$ versprechen: Mergesort (ganz zu Anfang) und Heapsort.

Alle übrigen Verfahren (Sortieren durch Einfügen oder Auswahl, Bubblesort bzw. Quicksort) haben Laufzeit $\mathcal{O}(n^2)$ im schlimmsten Fall, wobei Quicksort heraussticht, da es im mittleren Fall selbst die oben genannten Verfahren schlägt. Weitere Verbesserungen versprechen CleverQuicksort sowie Introsort.

Ist mit $\mathcal{O}(n \log(n))$ das "Ende der Fahnenstange" erreicht ?
Sicher müssen sich Sortierverfahren alle vorliegenden Daten anschauen dürfen, was sofort eine triviale untere Schranke von $\Omega(n)$ liefert.
Wären aber Sortierverfahren denkbar, die z.B. in $\mathcal{O}(n \log(\log(n)))$ laufen ?

Was heißt eigentlich “Sortieren” ?

Ausgangslage: Folge A von paarweise verschiedenen “Gegenständen” von einem (abstrakten) Datentyp D, d.h.: $A : D[1..n]$.
D soll uns Vergleichsoperatoren $<$, $\leq$ und $=$ zur Verfügung stellen, mit dessen Hilfe wir Vergleiche der Art $A[i] < A[j]$ in $\Theta(1)$ Zeit durchführen können.
Auf der dem Datentyp zugrundeliegenden Menge soll $\leq$ eine lineare Ordnung (manchmal auch totale Ordnung genannt) darstellen, d.h. insbesondere, dass stets $x \leq y$ oder $y \leq x$ für zwei beliebige Elemente gilt.
Ziel: Finde Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, sodass

$$A[\pi(1)] < A[\pi(2)] < \cdots < A[\pi(n)].$$

Frage: Wie viele Vergleiche muss ein Sortierverfahren S mindestens im schlimmsten Falle durchführen, wenn der Ablauf von S für festes n nur von den Ergebnissen der durchgeführten Vergleiche abhängen darf ?

Exkurs: nochmal Binärbäume

Wir hatten bereits gesehen:
Lemma: Die minimale Höhe eines Binärbaumes mit m Knoten ist
$h = \lceil \log_2(m+1) \rceil - 1$.

Leicht per Induktion einzusehen ist:
Lemma: Ein Binärbaum mit b Blättern hat mindestens $2b-1$ Knoten insgesamt.

Daraus ergibt sich:
Folgerung: Die minimale Höhe eines Binärbaumes mit b Blättern ist $h = \lceil \log_2(b) \rceil$.

Beweis: b Blätter $\rightsquigarrow$ mindestens $2b-1$ Knoten; um eine kleinstmögliche Höhe zu erzielen, wird diese Anzahl auch angenommen, wodurch für die kleinstmögliche Höhe h folgt:
$h = \lceil \log_2(2b) \rceil - 1 = \lceil \log_2(b) \rceil$.

Untere Schranke—ein Beweis

Vergleichsabhängiger Algorithmenverlauf und Notwendigkeit, $n!$ viele Anordnungen herstellen zu müssen, liefert *binären Entscheidungsbaum* mit $n!$ vielen Blättern für jeden Sortieralgorithmus.
Die Anzahl der schlimmstenfalls notwendigen Vergleiche entspricht der Länge eines Pfades von der Wurzel zu irgendeinem Blatt, also der Höhe h_n des Entscheidungsbaumes.
Mit dem folgenden (leicht zu zeigenden) Lemma folgt die behauptete untere Schranke mit obiger Folgerung sofort, denn:

$$h_n = \Omega(n!) = \Omega(n \log n)$$

Lemma: $n! \geq (n/2)^{n/2}$. Genaueres würde die *Stirlingsche Formel* liefern.

Sortieren in Linearzeit ?

Das sollte doch “eigentlich” nicht gehen. . .
Unsere Überlegungen waren aber mit verschiedenen Einschränkungen versehen, z.B.: es dürfen keine zwei gleichen “Schlüssel” auftreten, und das Sortieren muss durch ausschließliche Benutzung der Vergleichsoperatoren durchgeführt werden.
Sortiert man eine große Adressliste nach Postleitzahlen, so werden sehr viele Personen unter derselben Postleitzahl geführt werden.
Dieses kann essentiell dadurch ausgenutzt werden, dass man für jede Postleitzahl ein eigenes *Fach* zur Verfügung stellt; die eigentliche Sortieraufgabe wird durch Mitzählen der Gegenstände je Fach unterstützt (als weitere Operation).
$\leadsto$ Bucketsort (einfaches Sortieren durch Fachverteilen)

Fachverteilen

Bucketsort zählt die Häufigkeit jedes Schlüsselwertes in einer Liste. Daraus errechnet es die korrekte Position jedes Elements und fügt es in einer zweiten Liste dort ein.

Die Häufigkeit der Schlüsselwerte wird in einem so genannten *Histogramm* gespeichert. Dies wird meist als Array implementiert, das so lang ist, wie es mögliche Schlüsselwerte gibt; als Indizes werden dabei die Schlüsselwerte bzw. die ihnen zugeordneten ganzen Zahlen gewählt. Elemente mit gleichem Sortierschlüssel werden dabei in Gruppen, so genannten Fächern (Buckets), zusammengefasst.

Das Histogramm wird zunächst mit Nullen initialisiert. Dann wird die zu sortierende Liste durchlaufen und bei jedem Listenelement der entsprechende Histogrammeintrag um eins erhöht.

In einem zweiten Array, das ebenso lang ist wie das Histogramm-Array und ebenfalls mit Nullen initialisiert wird, werden nun die aus dem Histogramm errechneten Einfügepositionen gespeichert.

Schließlich werden in eine Liste, die ebenso lang ist wie die zu sortierende, die Elemente der zu sortierenden Liste nacheinander an den berechneten Positionen eingefügt.

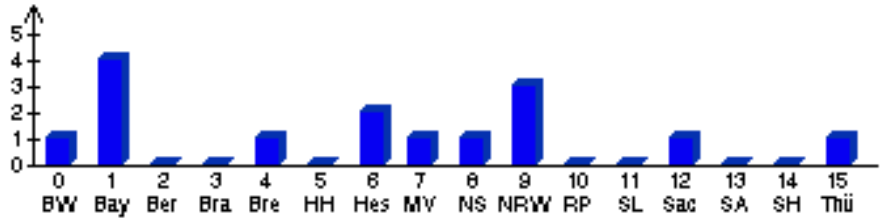
Fachverteilen am Beispiel

Es liegt eine alphabetisch nach Namen geordnete Liste von Städten vor:

1. Aachen (Nordrhein-Westfalen)
2. Augsburg (Bayern)
3. Bonn (Nordrhein-Westfalen)
4. Bremerhaven (Bremen)
5. Cuxhaven (Niedersachsen)
6. Dresden (Sachsen)
7. Erfurt (Thüringen)
8. Essen (Nordrhein-Westfalen)
9. Frankfurt am Main (Hessen)
10. München (Bayern)
11. Nürnberg (Bayern)
12. Rosenheim (Bayern)
13. Rostock (Mecklenburg-Vorpommern)
14. Stuttgart (Baden-Württemberg)
15. Wiesbaden (Hessen)

Diese Liste soll mit Bucketsort alphabetisch nach Bundesländern geordnet werden.

Fachverteilen am Beispiel (Forts.)



Bucket	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bundesland	BW	Bay	Ber	Bra	Bre	HH	Hes	MV	NS	NRW	RP	SL	SAC	SA	SH	THÜ
Einfügestelle	0	1	5	5	5	6	6	8	9	10	13	13	13	14	14	14

Nun wird das zu sortierende Feld durchlaufen: zunächst wird Aachen in der Zielliste an Position 10 gespeichert, da die Einfügeposition für den Bucket 9 (Nordrhein-Westfalen) 10 ist. Diese Einfügeposition wird daraufhin auf 11 erhöht, damit der nächste nordrhein-westfälische Ort hinter Aachen eingefügt wird.

Dann wird Augsburg an Position 1 der Zielliste gespeichert, denn dies ist die aktuelle Einfügeposition für Bucket 1 (Bayern). Diese Einfügeposition wird ebenfalls um eins erhöht.

Anschließend wird Bonn hinter Aachen an Position 11 gespeichert usw.

Fachverteilen am Beispiel (Schluss)

Man erhält die folgende Zielliste:

1. Stuttgart (Baden-Württemberg)
2. Augsburg (Bayern)
3. München (Bayern)
4. Nürnberg (Bayern)
5. Rosenheim (Bayern)
6. Bremerhaven (Bremen)
7. Frankfurt am Main (Hessen)
8. Wiesbaden (Hessen)
9. Rostock (Mecklenburg-Vorpommern)
10. Cuxhaven (Niedersachsen)
11. Aachen (Nordrhein-Westfalen)
12. Bonn (Nordrhein-Westfalen)
13. Essen (Nordrhein-Westfalen)
14. Dresden (Sachsen)
15. Erfurt (Thüringen)

Wie man sieht, sind Städte, die im gleichen Bundesland liegen, untereinander alphabetisch nach den Städtenamen sortiert.

Radixsort kann als iterierte Variante von Bucketsort aufgefasst werden.

Z.B. kann man bei natürlichen Zahlen als “Fächer” die zehn möglichen Werte einer Dezimalstelle ansehen. Sortiert man nun zunächst mit Bucketsort gemäß der kleinsten Dezimalstelle, dann nach der zweitkleinsten usf. erhält man am Ende eine vollständig geordnete Zahlenliste.

Formaler unterscheidet man zwischen sich abwechselnden *Partitionsphasen* und *Sammelphasen*.

Sind so jedoch n verschiedene Zahlen zwischen 1 und $\mathcal{O}(n)$ zu sortieren, degeneriert das Verfahren zu $\mathcal{O}(n \log(n))$, vgl.
`http://www-lehre.inf.uos.de/~ainf/2004/skript/node65.html`

Aufgabe: Sortieren der Zahlen 124, 523, 483, 128, 923, 584.

```
 |0|  |1|  |2|  |3|  |4|  |5|  |6|  |7|  |8|  |9|
                 |    |                    |
               523  124                  128
               483  584
               923
--> 523, 483, 923, 124, 584, 128
|0|  |1|  |2|  |3|  |4|  |5|  |6|  |7|  |8|  |9|
           |                              |
         523                            483
         923                            584
         124
         128
--> 523, 923, 124, 128, 483, 584
|0|  |1|  |2|  |3|  |4|  |5|  |6|  |7|  |8|  |9|
      |              |    |                    |
     124            483  523                 923
     128                 584
--> 124, 128, 483, 523, 584, 923  OK!
```

Sortieren: Rückblick

- Elementare Sortierverfahren
- Sortieren durch Auswahl (selection sort) $\leadsto$ Heapsort
- Quicksort

Repititorium gefällig? Siehe `http://www.linux-related.de/index.html?/coding/sort/sort_main.htm`

Sortieren: Auswahl-Kriterien

Ein *stabiles Sortierverfahren* ist ein Sortieralgorithmus, der die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, bewahrt.

Wenn beispielsweise eine Liste alphabetisch sortierter Personendateien nach dem Geburtsdatum neu sortiert wird, dann bleiben unter einem stabilen Sortierverfahren alle Personen mit gleichem Geburtsdatum alphabetisch sortiert.

Beispiele für stabile Sortierverfahren:
Bubblesort, Insertionsort, Mergesort, Radixsort

Beispiele für instabile Sortierverfahren:
Heapsort, Introsort, Quicksort

Frage: Wie sieht das bei anderen betrachteten Sortierverfahren aus ?

Sortieren: Auswahl-Kriterien

Ein Algorithmus arbeitet *in-place* bzw. *in situ*, wenn er außer dem für die Speicherung der zu bearbeitenden Daten benötigten Speicher nur eine konstante, also von der zu bearbeitenden Datenmenge unabhängige, Menge von Speicher (explizit) benötigt (wobei der Rekursionskeller unberücksichtigt bleibt).
Der Algorithmus überschreibt die Eingabedaten mit den Ausgabedaten.
Hierbei ist auch von *Ortsfestigkeit* die Rede.

So arbeitet etwa der Bubblesort-Algorithmus in-place, während Bucketsort out-of-place arbeitet, weil die Ausgabedaten in einer zweiten Liste gespeichert werden müssen.

Sortieren: Auswahl-Kriterien

Benötige ich möglichst früh eine “anfangs” sortierte Liste (z.B. als Eingabe für einen parallel arbeitenden Prozess) ?

Muss ich zwischen unterschiedlichen Geschwindigkeiten der Speichermedien unterscheiden ?
Im Extremfall liegen manche Daten auf Magnetbändern vor, was zur Betrachtung *externer Sortierverfahren* führt (z.B.: Varianten von Sortieren durch Mischen).