# Diskrete Mathematik

**Univ.-Prof. Dr. Goulnara ARZHANTSEVA**

SS 2018

# Sortieralgorithmen

1. Sortieren durch Einfügen
2. Mergesort
3. Quicksort

# Sortieren durch Einfügen

Wir beginnen mit dem ersten Element der Liste, das eine geordnete Liste $(a_1)$ darstellt.

Wenn wir die ersten $i$ Elemente in die richtige Ordnung

$$b_1 < b_2 < \cdots < b_i$$

gebracht haben, dann fügen wir im folgenden Schritt $a_{i+1}$ mit Binary search in die richtige Stelle ein.

# Sortieren durch Einfügen: $B(n)$

Im schlechtesten Fall ist also die Gesamtzahl $B(n)$ der benötigten Vergleiche gemäß ‘Binary search’

$$B(n) = \sum_{i=2}^{n} \lceil \log_2 i \rceil .$$

Diese Summe kann man explizit ausrechnen:

# Sortieren durch Einfügen: $B(n)$

Im schlechtesten Fall ist also die Gesamtzahl $B(n)$ der benötigten Vergleiche gemäß 'Binary search'

$$B(n) = \sum_{i=2}^{n} \lceil \log_2 i \rceil .$$

Diese Summe kann man explizit ausrechnen:
Schreiben wir $n$ als $n = 2^m + r$ für natürliche Zahlen $m$ und $r$, sodaß $0 < r \leq 2^m$. Es gilt

$$\lceil \log_2 i \rceil = k \iff 2^{k-1} < i \leq 2^k .$$

Wir partitionieren also den Summationsbereich, den $i$ durchläuft, in die Blöcke $\{2\}$, $\{3, 4\}$, $\{5, 6, 7, 8\}$, ..., $\{2^m + 1, \dots, n\}$.

# Sortieren durch Einfügen: $B(n)$

Im schlechtesten Fall ist also die Gesamtzahl $B(n)$ der benötigten Vergleiche gemäß ‘Binary search’

$$B(n) = \sum_{i=2}^{n} \lceil \log_2 i \rceil \,.$$

Diese Summe kann man explizit ausrechnen:
Schreiben wir $n$ als $n = 2^m + r$ für natürliche Zahlen $m$ und $r$, sodaß $0 < r \leq 2^m$. Es gilt

$$\lceil \log_2 i \rceil = k \iff 2^{k-1} < i \leq 2^k.$$

Wir partitionieren also den Summationsbereich, den $i$ durchläuft, in die Blöcke $\{2\}$, $\{3,4\}$, $\{5,6,7,8\}$, ..., $\{2^m+1,\ldots,n\}$. Im Bereich $\{2^{k-1}+1,\ldots,2^k\}$ ist $\lceil \log_2 i \rceil$ konstant gleich $k$. Der Beitrag den dieser Bereich für die Summe liefert ist daher $2^{k-1} \cdot k$. Insgesamt erhalten wir

$$B(n) = \sum_{k=1}^{m} k \cdot 2^{k-1} + (n - 2^m)(m+1).$$

# Sortieren durch Einfügen: $B(n)$

Wenn wir die (abbrechende) geometrischen Reihe

$$\sum_{k=0}^{m} x^k = \frac{1 - x^{m+1}}{1 - x}$$

differenzieren, erhalten wir

$$\sum_{k=0}^{m} k \cdot x^{k-1} = \frac{1 - (m+1)x^m + mx^{m+1}}{(1-x)^2}.$$

Wenn wir in dieser Formel $x = 2$ setzen, erhalten wir also zunächst

$$\sum_{k=1}^{m} k \cdot 2^{k-1} = 1 + (m-1)\, 2^m,$$

und damit schließlich für $B(n)$

$$B(n) = n(m+1) - 2^{m+1} + 1$$

oder, wenn wir $m$ wieder durch $n$ ausdrücken,

$$B(n) = n \lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1.$$

# Sortieren durch Einfügen: $B(n)$

Noch einmal:

$$B(n) = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1.$$

Im Vergleich mit der Größenordnung der informationstheoretischen Schranke ist das nicht schlecht:

Der führende Term $n\lceil \log_2 n \rceil$ ist gleich, nur der nächste Term ist kleiner, denn $-2^{\lceil \log_2 n \rceil}$ ist ungefähr $-n$, während $-n \log_2 e$ ungefähr $-1.44\,n$ ist.

# Mergesort

Eine andere Idee besteht darin, das Sortieren rekursiv aufzubauen:

Wir teilen die $n$ Elemente $a_1, a_2, \ldots, a_n$ in zwei ungefähr gleich große Hälften, sortieren beide Hälften (rekursiv) nach derselben Methode, und fügen am Schluß die beiden (dann geordneten) Listen zusammen.

Dieser Algorithmus heißt Sortieren durch Zusammenlegen (englisch: Merge-Sort).

# Mergesort

Eine andere Idee besteht darin, das Sortieren rekursiv aufzubauen:

Wir teilen die $n$ Elemente $a_1, a_2, \ldots, a_n$ in zwei ungefähr gleich große Hälften, sortieren beide Hälften (rekursiv) nach derselben Methode, und fügen am Schluß die beiden (dann geordneten) Listen zusammen.

Dieser Algorithmus heißt Sortieren durch Zusammenlegen (englisch: Merge-Sort).

Das Zusammenlegen erfordert aber einige Vergleiche:

Seien die Listen $b_1 < b_2 < \cdots < b_m$ und $c_1 < c_2 < \cdots < c_k$ gegeben, die wir in der richtigen Reihenfolge zusammenfügen sollen. Das Zusammenfügen können wir "reißverschlußartig" durchführen, also durch folgenden Algorithmus:

# Merge-Sort

{Initialisierung:}

$b \leftarrow (b_1, b_2, \ldots, b_m)$, $c \leftarrow (c_1, c_2, \ldots, c_k)$ und $l \leftarrow \Big(\Big)$ ($l$ ist die leere Liste).

{Schleife: Wird wiederholt, solange die Bedingung erfüllt ist.}

**while** (Bedingung: Beide Listen $b$ und $c$ sind nicht leer.) **do**

    Vergleiche die ersten Elemente $b'$ und $c'$ von $b$ und $c$; sei $x$ das kleinere der beiden.

    Füge $x$ hinten an die Liste $l$ an {Bemerke, daß $l$ stets richtig geordnet ist!}

    Entferne $x$ aus seiner "alten" Liste (d.h., wenn $x = b'$, dann setze $b \leftarrow b \setminus x$.

**end while**{Zum Schluß}

Falls eine der Listen $b$, $c$ nicht leer ist, füge sie an $l$ hinten an: $l$ ist dann die aus den ursprünglichen Listen $b$ und $c$ zusammengefügte geordnete Liste.

# Merge-Sort: Worst-Case Analyse

Wir brauchen für dieses Zusammenfügen im schlechtesten Fall $m + k - 1$ Vergleiche.

Sei $M(n)$ die Gesamtzahl der Vergleiche, die Mergesort im schlechtesten Fall für die Liste $(a_1, a_2, \ldots, a_n)$ benötigt.

Dann erhalten wir die Rekursion[1]

$$M(n) = M(\lfloor n/2 \rfloor) + M(\lceil n/2 \rceil) + n - 1.$$

Denn wir müssen zuerst die beiden Hälften mit $\lfloor n/2 \rfloor$ und $\lceil n/2 \rceil$ Elementen ordnen und brauchen dann schlimmstenfalls noch weitere $n - 1$ Vergleiche, um die beiden geordneten Hälften zusammenzufügen.

[1] Das ist nicht “nur” eine obere Schranke für den worst–case, sondern der “echte” worst–case!

# Merge-Sort: Worst-Case Analyse vs $\lceil \log_2 n! \rceil$

Mit Induktion: $M(n) = B(n)$ für alle $n$. Wir vergleichen $M(n) = B(n)$ mit der informationstheoretischen Schranke $\lceil \log_2 n! \rceil$:

| $n$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lceil \log_2 n! \rceil$ | 1 | 3 | 5 | 7 | 10 | 13 | 16 | 19 | 22 | 26 | 29 |
| $B(n) = M(n)$ | 1 | 3 | 5 | 8 | 11 | 14 | 17 | 21 | 25 | 29 | 33 |

Für $n \leq 11$ kann man tatsächlich Algorithmen konstruieren, die mit $\lceil \log_2 n! \rceil$ Vergleichen auskommen.

Für $n = 12$ hat eine Computer–Suche ergeben, daß die Minimalzahl 30 ist, also um 1 größer als die informationstheoretische Schranke.

# Quicksort–Algorithmus

Wir teilen wir die $n$ Elemente wieder in zwei Teile, diesmal aber so, daß die Elemente in einem Teil alle kleiner sind als die Elemente im anderen Teil; ordnen jeden der Teile (rekursiv) nach derselben Methode; und fügen die Teile dann (ohne zusätzliche Arbeit wie bei Mergesort) wieder aneinander.

# Quicksort–Algorithmus

Wir teilen wir die $n$ Elemente wieder in zwei Teile, diesmal aber so, daß die Elemente in einem Teil alle kleiner sind als die Elemente im anderen Teil; ordnen jeden der Teile (rekursiv) nach derselben Methode; und fügen die Teile dann (ohne zusätzliche Arbeit wie bei Mergesort) wieder aneinander.

Wenn wir annehmen, daß die Liste im Computer als Vektor

$$a = (a_1, \ldots, a_n)$$

gespeichert ist (daß wir also insbesondere zwei Komponenten des Vektors vertauschen können und zu jeder Komponente des Vektors den linken bzw. rechten Nachbarn — sofern vorhanden — bestimmen können), dann können wir diese Aufteilung algorithmisch so vornehmen, daß kein zusätzlicher Speicherplatz für die zwei Teile benötigt wird:

# Quicksort–Algorithmus

{Initialisierung:}
Markiere die letzte Koordinate (i.e.: $a_n$) des Vektors.
{Schleife: Wird wiederholt, solange die Bedingung erfüllt ist.}
**while** (Bedingung: Die Koordinate $a_1$ ist nicht markiert.) **do**
  Sei $x$ das markierte Element.
  **if** ($a_1$ steht links von $x$ UND $a_1 > x$) ODER ($a_1$ steht rechts von $x$ UND $a_1 \leq x$ **then**
    vertausche $a_1$ und $x$ (die Markierung “wandert dabei mit”)
  **end if**
  Bewege die Markierung um eine Stelle in Richtung von Element $a_1$ (nach rechts, wenn $a_1$ rechts von $x$ steht, sonst nach links).
**end while**

# Quicksort: $n = 9$ (aus Skriptum)

$a_1 = 4$ wird durch einen Kreis gekennzeichnet, die Markierung durch ein kleines Dreieck:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 4 | 8 | 9 | 5 | 2 | 1 | 6 | 7 | 3 |
| 3 | 8 | 9 | 5 | 2 | 1 | 6 | 7 | 4 |
| 3 | 4 | 9 | 5 | 2 | 1 | 6 | 7 | 8 |
| 3 | 4 | 9 | 5 | 2 | 1 | 6 | 7 | 8 |
| 3 | 4 | 9 | 5 | 2 | 1 | 6 | 7 | 8 |
| 3 | 1 | 9 | 5 | 2 | 4 | 6 | 7 | 8 |
| 3 | 1 | 4 | 5 | 2 | 9 | 6 | 7 | 8 |
| 3 | 1 | 2 | 5 | 4 | 9 | 6 | 7 | 8 |
| 3 | 1 | 2 | 4 | 5 | 9 | 6 | 7 | 8 |

# Quicksort: Worst-Case Analyse

Es ist klar, daß dieser Algorithmus nach $n - 1$ Schritten abbricht.

Wenn wir $a_1$ und das markierte Element $x$ als “Grenzen” (also als erstes/letztes Element) eines “Intervalls” (oder Teilvektors) $I$ von $a$ betrachten, dann nach jedem Schritt des Algorithmus links von $I$ nur Elemente kleiner $a_1$ stehen und rechts von $I$ nur Elemente größergleich $a_1$ (denn zu Beginn ist dies leererweise richtig, und in jedem Wiederholungsschritt wird dieser Zustand aufrechterhalten).

# Quicksort: Worst-Case Analyse

Es ist klar, daß dieser Algorithmus nach $n - 1$ Schritten abbricht.

Wenn wir $a_1$ und das markierte Element $x$ als "Grenzen" (also als erstes/letztes Element) eines "Intervalls" (oder Teilvektors) $I$ von $a$ betrachten, dann nach jedem Schritt des Algorithmus links von $I$ nur Elemente kleiner $a_1$ stehen und rechts von $I$ nur Elemente größergleich $a_1$ (denn zu Beginn ist dies leererweise richtig, und in jedem Wiederholungsschritt wird dieser Zustand aufrechterhalten).

Damit ist weiters klar, daß nach Abbruch des Algorithmus

- alle Elemente, die links von $a_1$ stehen, kleiner sind als $a_1$ — diese bilden also den einen Teil der Liste,
- alle Elemente, die rechts von $a_1$ stehen, größergleich sind als $a_1$ — diese bilden also den anderen Teil der Liste).

# Quicksort: Worst-Case Analyse

Auf diese beiden Teil–Listen wird dann dasselbe Verfahren rekursiv angewendet (wenn sie mehr als ein Element beinhalten), bis die ganze Liste richtig geordnet ist.

# Quicksort: Worst-Case Analyse

Auf diese beiden Teil–Listen wird dann dasselbe Verfahren rekursiv angewendet (wenn sie mehr als ein Element beinhalten), bis die ganze Liste richtig geordnet ist.

Die Worst-Case Analyse von Quicksort fällt sehr schlecht aus:
Wenn die Liste $a$ zufälligerweise bereits total geordnet sein sollte, also

$$a_1 < a_2 < \cdots < a_n,$$

dann wird die Liste in jedem rekursiven Schritt stets

- in die leere Liste (die Teil–Liste der Elemente kleiner als das erste Element),
- und in die ursprüngliche Liste ohne ihr erstes Element (die Teil–Liste der Elemente größergleich dem ersten Element, ohne das erste Element selbst)

zerlegt. In diesem Fall benötigen wir also $(n-1)+(n-2)+\cdots+1 = \binom{n}{2}$ Vergleiche — das sind alle Vergleiche von 2 Elementen aus $a$!

# Quicksort: Average-Case Analyse

Sei $Q(n)$ die durchschnittliche Anzahl von Vergleichen, die Quicksort für eine Liste $a$ der Länge $n$ benötigt.

Das erste Element $a_1$ ist jeweils mit Wahrscheinlichkeit $1/n$ das kleinste, das zweitkleinste, $\ldots$, oder das größte Element.

Wenn $a_1$ das $s$–kleinste Element in $a$ ist, dann erhalten wir eine Aufteilung in $s-1$ (die kleineren) und $n-s$ (die größeren) Elemente; für jeden der Teile wiederholen wir rekursiv die Prozedur.

# Quicksort: Average-Case Analyse

Zusammen mit den $n-1$ Vergleichen mit $a_1$ erhalten wir also die Rekursion

$$Q(n) = n - 1 + \frac{1}{n} \sum_{s=1}^{n} \left( Q(s-1) + Q(n-s) \right)$$

mit dem Anfangswert $Q(0) = 0$. Die rechte Seite können wir vereinfachen:

$$Q(n) = n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} Q(k) .$$

# Quicksort: Average-Case Analyse

Zusammen mit den $n-1$ Vergleichen mit $a_1$ erhalten wir also die Rekursion

$$Q(n) = n - 1 + \frac{1}{n}\sum_{s=1}^{n}\left(Q(s-1) + Q(n-s)\right)$$

mit dem Anfangswert $Q(0) = 0$. Die rechte Seite können wir vereinfachen:

$$Q(n) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} Q(k)\,.$$

Wir multiplizieren beide Seiten mit $n$ ...

$$nQ(n) = n(n-1) + 2\sum_{k=0}^{n-1} Q(k)$$

... und schreiben dieselbe Gleichung mit $n-1$ statt $n$ nochmals an:

$$(n-1)\,Q(n-1) = (n-1)(n-2) + 2\sum_{k=0}^{n-2} Q(k)\,.$$

# Quicksort: Average-Case Analyse

Nun subtrahieren wir die beiden obigen Gleichungen und erhalten

$$nQ(n) - (n-1)\,Q(n-1) = 2(n-1) + 2Q(n-1)\,,$$

oder vereinfacht

$$Q(n) = \frac{n+1}{n}Q(n-1) + 2\frac{n-1}{n}.$$

Das ist eine lineare Rekursion, die aber keine konstanten Koeffizienten hat.

Durch Iteration erraten wir eine Summendarstellung für $Q(n)$,

$$Q(n) = 2(n+1)\sum_{k=0}^{n-1}\frac{k}{(k+1)(k+2)},$$

die man mit Induktion nach $n$ leicht nachprüfen kann.

# Quicksort: Average-Case Analyse

Wenn wir die folgende “Partialbruchzerlegung”

$$\begin{aligned} \frac{k}{(k+1)(k+2)} &= \frac{k+2-2}{(k+1)(k+2)} \\ &= \frac{1}{k+1} - \frac{2}{(k+1)(k+2)} \\ &= \frac{1}{k+1} - \left( \frac{2}{k+1} - \frac{2}{k+2} \right) \end{aligned}$$

verwenden, dann vereinfacht sich die obige Summe (Teleskopsumme!) zu

$$Q(n) = 2(n+1) \left( \sum_{k=1}^{n} \frac{1}{k} - 2 + \frac{2}{n+1} \right) .$$

# Quicksort: Average-Case Analyse

Die Summe auf der rechten Seite ist die harmonische Zahl $H_n := \sum_{k=1}^{n} 1/k$; wir erhalten also

$$Q(n) = 2(n+1)H_n - 4n.$$

Da $H_n \sim \log n$, gelangen wir schließlich zu

$$Q(n) \sim 2n \log n = 2n \log_2 n / \log_2 e \approx 1.38\, n \log_2 n.$$

Wenn wir dieses Resultat wieder mit der informationstheoretischen Schranke $\lceil \log_2 n! \rceil$ vergleichen, dann sehen wir aus $\lceil \log_2 n! \rceil \sim n \log_2 n - n \log_2 e$:

Die Größenordnung $n \log_2 n$ ist "optimal", nur haben wir hier noch die multiplikative Konstante von $\approx 1.38$.

# Sortieralgorithmen: Bemerkung

Die “Effizienz” eines Algorithmus in der Praxis der Computerprogrammierung nicht allein mit der Anzahl der benötigten (abstrakten) Schritte (Tests) gemessen wird.

Es ist z.B. ein Nachteil des Sortierens durch Einfügen, daß man jedes Mal, wenn man den richtigen Platz für das neue Element $a_{i+1}$ in der bereits geordneten Liste gefunden hat, alle größeren Elemente verschieben muß, um für $a_{i+1}$ Platz zu schaffen.

Das Sortieren durch Zusammenlegen hat einen anderen Nachteil: Die Teillisten müssen der rekursiv aufgerufenen Funktion immer als Argument übergeben werden, dafür muß also stets neuer Speicherplatz verwendet werden. Insgesamt entsteht dadurch ein sehr großer Speicherbedarf.