6. Sortieren

Vgl. Skript Plödereder, SS 01, Folien 203 bis 317. Dieses Kapitel geht vor allem auf eine Vorlesung aus dem Jahre 1983 zurück.

Sortierte Folgen:

Gegeben sei eine endliche oder unendliche Menge mit totaler Ordnung $A = \{a_1, ..., a_s\}$ oder $A = \{a_1, a_2, a_3, a_4, ...\}$ mit $a_1 < a_2 < a_3 < a_4 < ...$.

(1) Eine Folge $v = v_1v_2...v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt (aufsteigend) geordnet genau dann, wenn gilt $v_1 \leq v_2 \leq ... \leq v_n$.

(2) Eine Folge $v = v_1v_2...v_n$ mit $v_i \in A$ (d.h., $v \in A^*$) heißt invers oder absteigend geordnet $\Leftrightarrow$ $v_n \leq v_{n-1} \leq ... \leq v_1$.

Permutationen:

Es sei n eine natürliche Zahl.
Eine bijektive Abbildung π: $\{1, 2, ..., n-1\} \rightarrow \{1, 2, ..., n-1\}$ heißt Permutation der Ordnung n.

[bijektiv = injektiv und surjektiv, d.h.,
injektiv: zu je zwei Elementen $i \neq j$ gilt $\pi(i) \neq \pi(j)$ und
surjektiv: zu jedem j existiert ein i mit $\pi(i) = j$.
Eine Permutation ist also nur eine Umordnung der n Elemente.]

Sortieraufgabe:
Ordne eine Folge von n Elementen so um, dass sie sortiert ist.
Formal: Finde zu einer beliebigen Folge $v = v_1v_2...v_n \in A^*$ eine Permutation π der Ordnung n mit: $v_{\pi(1)}v_{\pi(2)}...v_{\pi(n)}$ ist sortiert.

Meist hängt ein Sortieralgorithmus ab von Fragen wie:

- Wie sind die Daten gegeben, zu welchen Mengen gehören sie?
- Wo liegen die Daten? (Hauptspeicher, Platten, Bänder, ...?)
- Zulässige Operationen? (Vertauschen ...?)
- Behandlung gleicher Elemente? ($\Rightarrow$ Stabilität.)
- Nur Zugriffsstruktur oder Gesamtdaten sortieren?
- Gibt es zusätzlichen Speicher oder nicht?
- Sequentielles, paralleles, verteiltes Sortieren?
- Effizienz im Mittel oder auch im worst case?
- Erkennen oder Beachten von Vorsortierungen?

6.0 Überblick, allgemeine Analyse

Jedes Element v_i der zu sortierenden Folge $v = v_1v_2...v_n$ ist in der Praxis meist ein umfangreicher Datensatz (record). Die Folge wird nach einem Ordnungskriterium sortiert.

Fall 1: Dieses Kriterium wird durch eine Funktion $g: A \rightarrow M$ beschrieben, wobei M eine geordnete Menge ist (A braucht in diesem Fall gar nicht sortierbar zu sein). $g(v_i) = k_i$ heißt dann der Schlüssel von v_i, der in der Regel im record v_i enthalten ist. (Wir können also stets Fall 2 annehmen.)

Fall 2: Das Ordnungskriterium bezieht sich auf eine oder mehrere Komponenten des records. Den Vektor dieser Komponenten, nach denen zu sortieren ist, bezeichnen wir als Schlüssel.

Wird eine Folge v nach mehreren Komponenten (Schlüsseln) nacheinander geordnet (z.B. die Bevölkerung eine Stadt zunächst nach dem Alter und dann - bei gleichem Alter - nach dem Namen), so kann man die Folge zunächst nach dem am wenigsten relevanten Kriterium (im Beispiel: nach dem Namen) und dann schrittweise nach dem nächstwichtigeren Kriterium sortieren. Hierfür muss ein Sortierverfahren "stabil" sein.

Definition:
Ein Sortierverfahren S heißt **stabil**, wenn S die Reihenfolge von Elementen mit gleichem Schlüssel nicht verändert, d.h.:
wenn $S(v_1 v_2 ... v_n) = v_{i_1} v_{i_2} ... v_{i_n}$ ist, dann gilt für alle $v_{i_j} = v_{i_k}$ mit $i_j < i_k$ stets $j<k$.
S heißt invers stabil, wenn die Reihenfolge der Elemente mit gleichem Schlüssel von S gespiegelt wird.

Sortieren erfordert in der Praxis viele Umspeicherungs-operationen. Sind die Elemente sehr groß, so kostet dies viel Zeit. Man zieht daher die Schlüssel und den Verweis auf den jeweiligen Datensatz heraus und sortiert nur diese Schlüssel-Verweis-Tabelle. Wir werden also unseren Sortierverfahren Elemente des Datentyps

```
record  key: <Typ des Schlüssels>;
        zeiger: <Typ der zu sortierenden Elemente>;
end record
```

zugrunde legen. Es genügt, sich nur auf die Sortierung der Schlüssel zu beschränken.

Definition:
Für eine Permutation π: {1, 2, ..., n-1} $\rightarrow$ {1, 2, ..., n-1} heißt

$$I(\pi) = |\{ (i,j) \mid i < j \text{ und } \pi(i) > \pi(j)\}|$$

die Inversionszahl (oder der Fehlstand) von π.

Analog: Für eine Folge $v_1 v_2 ... v_n$ heißt

$$I(v) = |\{ (i,j) \mid i < j \text{ und } v_i > v_j\}|$$

die Inversionszahl (oder der Fehlstand) von v.

Wegen $I(v_1 v_2 ... v_n) + I(v_n v_{n-1} ... v_1) = |\{ (i,j) \mid i < j\}|$ gilt der

Hilfssatz: $I(v_1 v_2 ... v_n) + I(v_n v_{n-1} ... v_1) = ! \; n \cdot (n-1)$.

Wegen I(1 2 3 ... n) = 0 folgt I(n n-1 ...2 1) = ! $n \cdot (n-1)$.

Definition:
Ein Sortierverfahren S heißt ordnungsverträglich $\Leftrightarrow$

Je geordneter die zu sortierende Folge bereits ist,
umso schneller arbeitet S.

Genauer:
Wenn S für zwei Folgen $v=v_1 v_2 ... v_n$ und $w=w_1 w_2 ... w_n$ die Permutationen π_1 und π_2 realisiert und wenn die Inversionszahl von π_1 kleiner als die von π_2 ist, dann ist auch die Zeit, die S zum Sortieren von v benötigt, kleiner als die Zeit zum Sortieren von w.

Wir betrachten die Operation "benachbartes Vertauschen". Diese überführt eine Folge

$$v_1\, v_2 \dots v_{i-1}\, v_i\, v_{i+1}\, v_{i+2} \dots v_n$$

in die Folge

$$v_1\, v_2 \dots v_{i-1}\, v_{i+1}\, v_i\, v_{i+2} \dots v_n \quad \text{(für ein } 0<i<n\text{)}.$$

Hierfür gilt:

$$\left| I(v_1 v_2 \dots v_{i-1} v_i v_{i+1} v_{i+2} \dots v_n) - I(v_1 v_2 \dots v_{i-1} v_{i+1} v_i v_{i+2} \dots v_n) \right| = 1.$$

Somit haben wir gezeigt:

Hilfssatz:
Ein Sortierverfahren, das ausschließlich mit der Operation "benachbartes Vertauschen" arbeitet, benötigt im worst case mindestens ! $n \cdot (n-1)$ Schritte.

Wir betrachten die Operation "Vertauschen". Diese überführt eine Folge (für $1 \leq i \leq j \leq n$)

$$v_1\, v_2 \dots v_{i-1}\, v_i\, v_{i+1} \dots v_{j-1}\, v_j\, v_{j+1} \dots v_n$$

in die Folge

$$v_1\, v_2 \dots v_{i-1}\, v_j\, v_{i+1} \dots v_{j-1}\, v_i\, v_{j+1} \dots v_n\, .$$

Hierfür gilt:

$$0 \leq \left| I(v_1 \dots v_i \dots v_j \dots v_n) - I(v_1 \dots v_j \dots v_i \dots v_n) \right| \leq n-1,$$

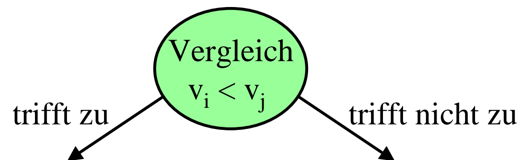
wobei der größte Wert n-1 nur erreicht wird, wenn das kleinste Element am Ende stand und durch die Vertauschung an den Anfang gebracht wurde bzw. eine hierzu symmetrische Situation vorliegt. Es folgt:

Hilfssatz: Ein Sortierverfahren, das ausschließlich mit der Operation "Vertauschen" arbeitet, benötigt im worst case mindestens ! n Schritte.

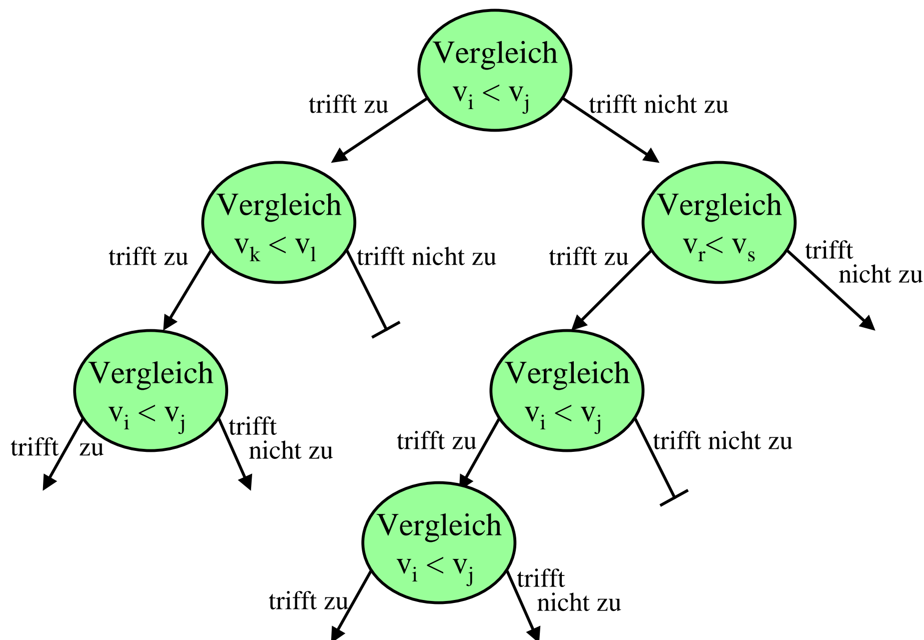
In der Regel weiß man nicht, ob man zwei Elemente einer Folge vertauschen soll. Diese Entscheidung wird durch einen Vergleich getroffen: Falls $v_i < v_j$ und $i > j$ ist, dann vertauscht man diese beiden Elemente in der Folge.

Wird das Vertauschen durch vorher gehende Vergleiche gesteuert, so dauert das Sortierverfahren mindestens so lange wie die Anzahl der hierfür erforderlichen Vergleiche.

Eine Folge von Vergleichen bildet einen binären Baum, der aus den Knoten und Kanten



aufgebaut ist.



Hat man alle Informationen zum Sortieren gewonnen, dann stellen die null-Zeiger in diesem Baum die Permutationen dar, die zur Sortierung gehören. Da es n! Permutationen der Ordnung n gibt, muss der "Baum der Vergleiche" daher mindestens n! null-Zeiger besitzen.

Ein binärer Baum mit n-1 Knoten besitzt genau n null-Zeiger. Also muss der Baum der Vergleiche

mindestens n!-1

Knoten besitzen.
Die Länge des längsten Weges, also die Tiefe dieses Baums gibt die Zahl der erforderlichen Vergleiche im worst case an. Die Tiefe eines binären Baums mit k Knoten ist aber mindestens log(k+1).

Hilfssatz: Ein Sortierverfahren, das ausschließlich auf Vergleichen zweier Elemente beruht, benötigt im worst case mindestens

$$\log(n!-1) \approx n \cdot \log(n) - 1{,}44\, n$$

Schritte.

Hinweis: Wende die Stirlingschen Formel an: Zu jedem n gibt es ein d mit 0<d<1, so dass gilt:

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\ \ e^{\frac{1}{12}d}$$

Durch Logarithmieren erhält man hieraus:
$\log(n!) \approx n \cdot \log(n) - n \cdot \log(e) \approx n \cdot \log(n) - 1{,}44\, n$

Überblick über die üblichen Sortiermethoden:

6.1: Aussuchen / Auswählen:
- a. Minimumsuche (minimum sort)
- b. Heapsort (normal, bottom up, ultimativ)

6.2: Einfügen:
- a. Einfügen in Listen (Insertion sort)
- b. Baumsortieren (mit binären Bäumen, AVL-Bäumen, ...)
- c. Fachverteilen (radix exchance)

6.3: Austauschen:
- a. Benachbartes Austauschen (bubble sort, shaker sort)
- b. Shellsort
- c. Quicksort

6.4: Mischen:
merge sort und diverse Varianten

6.5: Streuen und Sammeln (bucket sort)

Sortiermethoden	Zeitaufwand	zusätzl. Platz
6.1: Aussuchen / Auswählen		
a. Minimumsuche	! n^2	konstant
b. Heapsort	$\leq 2n \log(n)$	konstant
6.2: Einfügen		
a. Einfügen in Listen	! n^2	konstant
b. Baumsortieren	$\leq 1{,}44\ n \log(n)$	O(n)
c. Fachverteilen (im Mittel)	$O(n \log(n))$	O(n)
6.3: Austauschen		
a. Benachb. Austauschen	! n^2	konstant
b. Shellsort	$O(n^{\frac{3}{2}})$	$\leq \log(n)$
c. Quicksort (im Mittel)	$\leq 1{,}386\ n \log(n)$	$2 \log(n)$
6.4: Mischen		
Verschmelzen (merge sort)	$O(n \log(n))$	n
6.5: Streuen und Sammeln	O(n)	O(n)