Effiziente Algorithmen

Martin Hofmann

Institut für Informatik

LMU München

Sommersemester 2007

Literatur

- T. Cormen, C. Leiserson and R. Rivest, Introduction to Algorithms, MIT Press, 2. Aufl. 2001. Seit 2004 existiert dt. Üb. bei Oldenbourg. Standardwerk und Vorlesungstext.
- Weitere Lehrbücher: [Aho, Hopcroft, Ullman], [Sedgewick], [Mehlhorn], [Ottmann-Widmayer]. Können zur Vertiefung hinzugezogen werden.

Was sind Algorithmen, wieso Algorithmen?

Ein **Algorithmus** ist eine genau festgelegte **Berechnungsvorschrift**, die zu einer Eingabe oder Problemstellung eine wohldefinierte Ausgabe oder Antwort berechnet.

Ein Algorithmus kann durch ein Programm in einer Programmiersprache angegeben werden, aber typischerweise lässt sich ein und derselbe Algorithmus auf verschiedene Arten und in verschiedenen Programmiersprachen implementieren.

Neben Programmiermethodiken (wie Modularisierung, Objektorientierung, etc.) bilden Algorithmen den **Schlüssel zu erfolgreicher Softwareentwicklung**. Durch Verwendung professioneller Algorithmen lassen sich oft spektakuläre Laufzeitgewinne realisieren.

Zwar stellen Sprachen wie Java viele Algorithmen fertig implementiert in Form von **Bibliotheken** bereit; viele Projekte werden jedoch nach wie vor in C implementiert, außerdem ist es oft der Fall, dass eine Bibliotheksfunktion nicht genau passt.

Die **Algorithmik** ist eine lebendige Disziplin. **Neue Problemstellungen** (Bioinformatik, Internet, Mobilität) erfordern zum Teil neuartige Algorithmen. Für alte Problemstellungen werden bessere Algorithmen entwickelt (kombinatorische Optimierung, Planung). **Schnellere Hardware** rückt bisher unlösbare Probleme in den Bereich des Machbaren (automatische Programmverif., Grafik, Simulation).

Was sind Datenstrukturen, wieso Datenstrukturen?

Methoden, Daten so anzuordnen, dass Zugriffe möglichst schnell erfolgen.

Konkurrierende Anforderungen: Platzbedarf, verschiedene Arten von Zugriffen.

Beispiel: Stapel (Keller): Ist die Größe von vornherein beschränkt, so bietet sich ein Array an. Anderenfalls eine verkettete Liste. Diese verbraucht aber mehr Platz und Zeit. Will man nicht nur auf das oberste Element zugreifen, so benötigt man eine doppelt verkettete Liste oder gar einen Baum, ...

Datenstrukturen und Algorithmen gehen Hand in Hand. Oft bildet eine geeignete Datenstruktur den Kern eines Algorithmus. Beispiel: Heapsort (nächste Woche).

Zusammenfassung

- Kapitel I **Einführung**: Sortieren durch Einfügen, Sortieren durch Mischen, Laufzeitabschätzungen, Lösen von Rekurrenzen. Folie 6.
- Kapitel II **Sortieren und Suchen**: Heapsort, Quicksort, Median. Folie ??.
- Kapitel III **Professionelle Datenstrukturen**: Balancierte Binärbäume, Hashtabellen, *union find*, Prioritätsschlangen. Folie ??.
- Kapitel IV **Allgemeine Entwurfsmethoden**: Greedy, Backtracking, Branch & Bound, dynamische Programmierung. Amortisierte Komplexität. Folie ??.
- Kapitel V **Graphalgorithmen**: transitive Hülle, Spannbäume, kürzester Pfad, Fluss in Netzwerken. Folie ??.
- Kapitel VI[†] **Schnelle Fouriertransformation**: Grundlagen der FT, FFT Algorithmus, Anwendungen aus der Signal- und Bildverarbeitung. Folie ??.
- Kapitel VII[†] **Suche in Zeichenketten** mit Hashing, mit Automaten, mit dynamischer Programmierung (approximative Z.); Anwendung in der Bioinformatik. Folie ??.
- Kapitel VIII[†] Geometrische Algorithmen und Entwurfsverfahren. Folie ??.

Von den mit † gekennzeichneten Kapiteln wird nur eine Auswahl behandelt.

I. Einführung

- Sortieren durch Einfügen **2.1,2.2**
- Sortieren durch Mischen **2.3**
- Laufzeitabschätzungen **3.1**
- Lösen von Rekurrenzen **4**

Kapitelangaben wie **2.1,2.2** beziehen sich auf [Cormen].

Sortieren

Eingabe: Eine Folge von n Zahlen $\langle a_1, a_2, \dots, a_n \rangle$.

Ausgabe: Eine Umordnung $\langle a_{\pi_1}, a_{\pi_2}, \dots, a_{\pi_n} \rangle$, sodass $a_{\pi_1} \leq a_{\pi_2} \leq \dots \leq a_{\pi_n}$.
Hierbei muss $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ eine Bijektion sein.

Beispiel: $\langle 31, 41, 59, 26, 41, 58 \rangle \mapsto \langle 26, 31, 41, 41, 58, 59 \rangle$.

Allgemeiner: „Zahlen“ $\mapsto$ „Objekte“, „ $\leq$ “ $\mapsto$ „Ordnungsrelation“.

Beispiele:

- Zeichenketten, lexikographische Ordnung.
- Autos, Baujahr.
- Geometrische Objekte, Entfernung.

Sortieren durch Einfügen

INSERTION-SORT(A)

```
1  for  $j \leftarrow 2$  to  $length[A]$ 
2      do  $key \leftarrow A[j]$ 
3           $\triangleright$  Einfügen von  $A[j]$  in die sortierte Folge  $A[1..j-1]$ 
4           $i \leftarrow j - 1$ 
5          while  $i > 0$  und  $A[i] > key$ 
6              do  $A[i+1] \leftarrow A[i]$ 
7                   $i \leftarrow i - 1$ 
8           $A[i+1] \leftarrow key$ 
```

Pseudocode

Um von Implementierungsdetails zu abstrahieren, verwenden wir zur Angabe von Algorithmen **Pseudocode**: Eine PASCAL oder C ähnliche Sprache, welche

- Neben formellen Anweisungen und Kontrollstrukturen auch **Umgangssprache** enthalten kann,
- Verallgemeinerte Datentypen (wie Mengen, Graphen, etc.) bereitstellt,
- Blockstruktur auch durch **Einrückung** kennzeichnet (wie in Python)

Laufzeitanalyse

Wir wollen die **Laufzeit** eines Algorithmus als **Funktion der Eingabegröße** ausdrücken.

Manchmal auch den Verbrauch an anderen Ressourcen wie Speicherplatz, Bandbreite, Prozessorzahl.

Laufzeit bezieht sich auf ein bestimmtes Maschinenmodell, hier RAM (*random access machine*).

Laufzeit kann neben der Größe der Eingabe auch von deren Art abhängen (*worst case, best case, average case*). Meistens *worst case*.

Laufzeit von INSERTION-SORT

INSERTION-SORT(A)	Zeit	Wie oft?
1 for $j \leftarrow 2$ to $length[A]$	c_1	n
2 do $key \leftarrow A[j]$	c_2	$n - 1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1..j - 1]$		
4 $i \leftarrow j - 1$	c_4	$n - 1$
5 while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^n t_j$
6 do $A[i + 1] \leftarrow A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
7 $i \leftarrow i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] \leftarrow key$	c_8	$n - 1$

$t_j =$ Anzahl der Durchläufe der *while*-Schleife im j -ten Durchgang.

$c_1 - c_8 =$ un spezifizierte Konstanten.

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

Bester Fall: Array bereits sortiert

Ist das Array bereits aufsteigend sortiert, so wird die *while*-Schleife jeweils nur einmal durchlaufen: $t_j = 1$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Also ist $T(n)$ eine **lineare Funktion** der Eingabegröße n .

Schlechtester Fall: Array absteigend sortiert

Ist das Array bereits absteigend sortiert, so wird die *while*-Schleife maximal oft durchlaufen: $t_j = j$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \sum_{j=2}^n t_j + (c_6 + c_7) \sum_{j=2}^n (t_j - 1)$$

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + (c_2 + c_4 + c_8)(n - 1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + (c_6 + c_7) \frac{n(n-1)}{2}$$

Also ist $T(n)$ eine **quadratische Funktion** der Eingabegröße n .

worst case und average case

Meistens geht man bei der Analyse von Algorithmen vom *worst case* aus.

- *worst case* Analyse liefert **obere Schranken**
- In vielen Fällen ist der *worst case* die Regel
- Der (gewichtete) Mittelwert der Laufzeit über alle Eingaben einer festen Länge (*average case*) ist oft bis auf eine multiplikative Konstante nicht besser als der *worst case*.
- Belastbare Annahmen über die mittlere Verteilung von Eingaben sind oft nicht verfügbar.
- Manchmal muss man aber eine *average case* Analyse durchführen (Beispiel: Quicksort)
- Manchmal zeigt sich auch, dass in der Praxis der *worst case* selten auftritt (Beispiel: Simplexverfahren, Typinferenz in ML). Abhilfe *parametrisierte Komplexität, smoothed analysis*.

Größenordnungen

Um Rechnungen zu vereinfachen und da Konstanten wie $c_1, \dots, c_8$ sowieso willkürlich sind, beschränkt man sich oft darauf, die **Größenordnung** der Laufzeit anzugeben:

$$an^2 + bn + c = O(n^2)$$

$$an + b = O(n)$$

$$a2^n + bn^{10000} = O(2^n)$$

$O(f(n))$ bezeichnet alle Funktionen der **Größenordnung** höchstens $f(n)$. Dies wird später formal definiert.

Laufzeit von INSERTION-SORT im schlechtesten Fall ist $O(n^2)$.

Es gibt auch gemischte Notationen wie $2n^2 + O(n)$, also $2n^2$ plus eine Funktion, die höchstens linear wächst.

Teile und herrsche

Das Entwurfsverfahren *divide-and-conquer* (Teile und Herrsche, *divide et impera*) dient dem Entwurf **rekursiver Algorithmen**.

Die Idee ist es, ein Problem der Größe n in mehrere gleichartige aber kleinere **Teilprobleme** zu zerlegen (*divide*).

Aus rekursiv gewonnenen Lösungen der Teilprobleme gilt es dann, eine Gesamtlösung für das ursprüngliche Problem zusammenzusetzen (*conquer*).

Beispiel: Sortieren durch Mischen (*merge sort*):

Teile n -elementige Folge in zwei Teilfolgen der Größe $n/2 (+1)$.

Sortiere die beiden Teilfolgen rekursiv.

Füge die nunmehr sortierten Teilfolgen zusammen durch Reißverschlussverfahren.

Sortieren durch Mischen

MERGE-SORT(A, p, r)

▷ Sortiere $A[p..r]$

```
1  if  $p < r$ 
2      then  $q \leftarrow (p + r)/2$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

MERGE(A, p, q, r)

▷ Sortiere $A[p..r]$ unter der Annahme, dass $A[p..q]$ und $A[q + 1..r]$ sortiert sind.

```
1   $i \leftarrow p; j \leftarrow q + 1$ 
2  for  $k \leftarrow 1$  to  $r - p + 1$  do
3      if  $j > r$  or  $(i \leq q$  and  $A[i] \leq A[j])$ 
4      then  $B[k] \leftarrow A[i]; i \leftarrow i + 1$  else  $B[k] \leftarrow A[j]; j \leftarrow j + 1$ 
5  for  $k \leftarrow 1$  to  $r - p + 1$  do  $A[k + p - 1] \leftarrow B[k]$ 
```

Analyse von MERGE-SORT

Sei $T(n)$ die Laufzeit von MERGE-SORT.

Das **Aufteilen** braucht $O(1)$ Schritte.

Die **rekursiven Aufrufe** brauchen $2T(n/2)$ Schritte.

Das **Mischen** braucht $O(n)$ Schritte.

Also:

$$T(n) = 2T(n/2) + O(n), \text{ wenn } n > 1$$

NB $T(1)$ ist irgendein fester Wert.

Die Lösung dieser Rekurrenz ist $T(n) = O(n \log(n))$.

Für große n ist das besser als $O(n^2)$ trotz des Aufwandes für die Verwaltung der Rekursion.

NB: $\log(n)$ bezeichnet den Zweierlogarithmus ($\log(64) = 6$). Bei O -Notation spielt die Basis des Logarithmus keine Rolle, da alle Logarithmen proportional sind. Z.B.: $\log(n) = \ln(n) / \ln(2)$.

Motivation der Lösung

Intuitiv: Rekursionstiefe: $\log(n)$, auf dem Tiefenniveau k hat man 2^k Teilprobleme der Größe jeweils $g_k := n/2^k$, jedes verlangt einen Mischaufwand von $O(g_k) = O(n/2^k)$. Macht also $O(n)$ auf jedem Niveau: $O(n \log(n))$ insgesamt.

Durch Formalismus: Sei $T(n)$ für Zweierpotenzen n definiert durch

$$T(n) = \begin{cases} 0, & \text{wenn } n = 1 \\ 2T(n/2) + n, & \text{wenn } n = 2^k, k \geq 1 \end{cases}$$

Dann gilt $T(2^k) = k2^k$, also $T(n) = n \log(n)$ für $n = 2^k$.

Beweis durch Induktion über k (oder n).

Dass es dann für Nicht-Zweierpotenzen auch gilt, kann man beweisen.

Asymptotik: Definition von O

Seien $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ Funktionen. (NB Geht auch für andere Definitionsbereiche.)

$$O(g) = \{f \mid \text{es ex. } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass } f(n) \leq cg(n) \text{ für alle } n \geq n_0\}$$

$O(g)$ ist also eine **Menge von Funktionen**: die Funktionen der **Größenordnung** höchstens g .

In Ermangelung **guter Notation** für Funktionen schreibt man z.B. $O(n^2)$ und meint damit $O(g)$ wobei $g(n) = n^2$.

Bei $O(m + n^2)$ wird's schwierig. Da muss aus dem **Zusammenhang** klar werden, auf welche Variable sich das O bezieht.

Noch schlimmer sind Notationen wie $O(1 + \alpha)$, wobei $\alpha = n/m$, kommt aber vor...

Man schreibt $f = O(g)$ anstelle von $f \in O(g)$.

Asymptotik: Definition von O

Beispiele:

$$4n^2 + 10n + 3 = O(n^2)$$

$$an + b = O(n)$$

$$a2^n + bn^{10000} = O(2^n)$$

$$n = O(2^n)$$

$$3n^4 + 5n^3 + 7\log(n) \leq 3n^4 + 5n^4 + 7n^4 = 15n^4 = O(n^4)$$

Asymptotik: Ausdrücke mit O

Kommt $O(g)$ in einem Ausdruck vor, so bezeichnet dieser die Menge aller Funktionen, die man erhält, wenn man die Elemente von $O(g)$ für $O(g)$ einsetzt.

Z.B. ist $n^2 + 100n + 2 \log(n) = n^2 + O(n)$ aber $2n^2 \neq n^2 + O(n)$.

Manchmal kommt O sowohl links, als auch rechts des Gleichheitszeichens vor. Man meint dann eine Inklusion der entsprechenden Mengen. M.a.W. jede Funktion links kommt auch rechts vor.

Z.B.

$$2n^2 + O(n) = O(n^2)$$

Die Verwendung von „ $=$ “ statt „ $\in$ “ oder „ $\subseteq$ “ ist zugegebenermaßen etwas unbefriedigend, z.B. nicht symmetrisch, hat sich aber aus praktischen Gründen durchgesetzt.

Asymptotik: $\Theta, \Omega, o, \omega$

Seien wieder $f, g : \mathbb{N} \rightarrow \mathbb{R}$. (NB Geht auch für andere Definitionsbereiche.)

$$\Omega(g) = \{f \mid g \in O(f)\} = \{f \mid \text{es ex. } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass } cg(n) \leq f(n) \text{ für alle } n \geq n_0\}$$

$$\Theta(g) = O(g) \cap \Omega(g)$$

$f = \Omega(g)$ heißt: g ist eine asymptotische **untere Schranke** für f .

$f = \Theta(g)$ heißt: f ist von derselben Größenordnung wie g .

Oft wird O im Sinne von Θ gebraucht.

Kleines o und ω

$o(g) = \{f \mid \text{für alle } c > 0 \text{ gibt es } n_0 \in \mathbb{N} \text{ so dass } 0 \leq f(n) \leq cg(n) \text{ für alle } n \geq n_0\}$

$f = o(g)$ heisst: g ist eine asymptotische **obere Schranke** für f und f ist nicht asymptotisch proportional zu g . f ist gegenüber g verschwindend gering.

Beachte: Ist f nichtnegativ, so gilt $f = o(g) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

Man schreibt auch $f \in \omega(g)$ für $g \in o(f)$, also $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$.

Beispiele

$$2n^2 = O(n^2), \text{ aber } 2n^2 \neq o(n^2)$$

$$1000n = O(n^2) \text{ und sogar } 1000n = o(n^2)$$

Beispiele für Asymptotik

- $n^2 = o(n^3)$
- $10^{10}n^2 = O(10^{-10}n^2)$ aber nicht mit o statt O .
- $f = o(g)$ impliziert $f = O(g)$
- $\Theta(\log_b(n)) = \Theta(\log_c(n))$. NB CORMEN schreibt $\lg(x)$ für $\log_2(x) = \log(x)$. Allgemein schreibt man $\ln(x)$ für $\log_e(x)$.
- $f = \Theta(g)$ und $g = \Theta(h)$ impliziert $f = \Theta(h)$. Ähnliches gilt für O, Ω, o, ω .
- $\log n = o(n^\epsilon)$ für jedes $\epsilon > 0$.
- $n! = O(n^n)$
- $\log(n!) = \Omega(n \log n)$
- $\log(n!) = \Theta(n \log n)$
- $\Theta(f) + \Theta(g) = \Theta(f + g)$ und umgekehrt und auch für O, Ω, o, ω .

Asymptotik und Induktion

Will man eine asymptotische Beziehung durch Induktion beweisen, so muss man die Konstante c ein für alle Mal wählen und darf sie nicht während des Induktionsschritts abändern:

Sei $S(n) = \sum_{i=1}^n i$, also $S(n) = n(n+1)/2$.

Falscher Beweis von $S(n) = O(n)$:

$$S(1) = O(1).$$

Sei $S(n) = O(n)$, dann gilt $S(n+1) = S(n) + (n+1) = O(n) + O(n) = O(n)$.

Asymptotik und Induktion

Richtiger Beweis von $S(n) = \Omega(n^2)$:

Wir versuchen die Induktion mit einem noch zu bestimmenden c durchzukriegen und leiten Bedingungen an diese Konstante ab.

$S(1) = 1$, daraus $c \leq 1$.

Sei $S(n) \geq cn^2$. Es ist

$S(n+1) = S(n) + n + 1 \geq cn^2 + n + 1 \geq cn^2 + 2cn + c = c(n+1)^2$ falls $c \leq 1/2$.

Also funktioniert's mit $c \leq 1/2$, insbesondere $c = 1/2$.

Die Induktion fing bei $n = 1$ an, also können wir $n_0 = 1$ nehmen. $\square$

Bemerkung: Man kann im Prinzip immer $n_0 = 1$ wählen und eventuelle Sprünge der zu beschränkenden Funktion durch Vergrößern/Verkleinern von c auffangen. Die Definition mit n_0 ist aber in der Praxis flexibler und auf Funktionen mit negativen Werten verallgemeinerbar.

Lösen von Rekurrenzen bei *divide and conquer*

Bei der Analyse von *divide-and-conquer* Algorithmen stößt man auf Rekurrenzen der Form:

$$T(n) = aT(n/b) + f(n)$$

Das passiert dann, wenn ein Problem der Größe n in a Teilprobleme der Größe n/b zerlegt wird und der Aufwand für das Aufteilen und Zusammenfassen der Teilresultate Aufwand $f(n)$ erfordert.

Bemerkung: n/b steht hier für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$.

Mehr noch: $aT(n/b)$ kann man hier sogar als $a_1T(\lfloor n/b \rfloor) + a_2T(\lceil n/b \rceil)$ mit $a_1 + a_2 = a$ lesen.

Die *Master-Methode* liefert eine kochrezeptartige Lösung für derartige Rekurrenzen.

Hauptsatz der Master-Methode

Satz: Seien $a \geq 1, b > 1$ Konstanten, $f, T : \mathbb{N} \rightarrow \mathbb{R}$ Funktionen und gelte

$$T(n) = aT(n/b) + f(n)$$

Dann erfüllt T die folgenden Größenordnungsbeziehungen:

1. Wenn $f(n) = O(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$, so gilt $T(n) = \Theta(n^{\log_b a})$.
2. Wenn $f(n) = \Theta(n^{\log_b a})$, so gilt $T(n) = \Theta(n^{\log_b a} \log(n))$.
3. Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ und außerdem $af(n/b) \leq cf(n)$ für ein $c < 1$ und genügend großes n , so gilt $T(n) = \Theta(f(n))$.

Zum Beweis, siehe CORMEN Abschnitt 4.4.

Beachte: Ist $f(n) = n^{\log_b(a) + \epsilon}$, so gilt

$$af(n/b) = an^{\log_b(a) + \epsilon} / b^{\log_b(a) + \epsilon} = af(n)/a/b^\epsilon \leq cf(n) \text{ mit } c = b^{-\epsilon} < 1.$$

Beachte: $\log_b(a) = \log(a)/\log(b)$.

Wichtigster Spezialfall: Wenn $T(n) = aT(n/a) + O(n)$, dann $T(n) = O(n \log n)$.

Beispiele für die Master Methode

- Die Laufzeit $T(n)$ von MERGE-SORT genügt der Beziehung:
 $T(n) = 2T(n/2) + \Theta(n)$ somit $T(n) = \Theta(n^{\log_2(2)} \log(n)) = \Theta(n \log(n))$.
- Wenn $T(n) = 2T(n/3) + n$ dann $T(n) = \Theta(n)$.
- Wenn $T(n) = 2T(n/2) + n^2$ dann $T(n) = \Theta(n^2)$.
- Wenn $T(n) = 4T(n/2) + n$ dann $T(n) = \Theta(n^2)$.
- Wenn $T(n) = aT(n/a) + O(n)$ dann $T(n) = O(n \log n)$. Analog für Θ .
- Die Rekurrenz $2T(n/2) + n \log n$ kann man mit der Master-Methode nicht lösen. Die Lösung ist hier $T(n) = \Theta(n \log^2(n))$.

Matrizenmultiplikation

Seien $A = (a_{ik}), B = (b_{ik})$ zwei $n \times n$ Matrizen. Das Produkt $C = AB$ ist definiert durch

$$c_{ik} = \sum_{j=1}^n a_{ij}b_{jk}$$

Das macht $\Theta(n^3)$ Additionen und Multiplikationen.

Matrizen kann man **blockweise** multiplizieren:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix}$$

Die “kleinen” Produkte AB sind hier selbst wieder Matrizenprodukte.

Die blockweise Multiplikation führt auf dasselbe Ergebnis, wie die direkte Verwendung der Definition.

Matrizenmultiplikation mit *divide-and-conquer*

$T(n)$ = Anzahl der Operationen erforderlich zur Multiplikation zweier $n \times n$ Matrizen

Es ist mit *divide-and-conquer*:

$$T(n) = 8T(n/2) + \Theta(n^2)$$

Es ist $\log_2(8) = 3$ und $\Theta(n^2) = O(n^{3-\epsilon})$, z.B. mit $\epsilon = 1$, also $T(n) = \Theta(n^3)$.

Keine Verbesserung, da auch die direkte Verwendung der Definition $\Theta(n^3)$ Rechenoperationen ($\Theta(n^3)$ Multiplikationen, $\Theta(n^2)$ Additionen.)

STRASSENS erstaunlicher Algorithmus

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} R & S \\ T & U \end{pmatrix}$$

wobei (V. STRASSENS geniale Idee):

$$\begin{aligned} R &= P_5 + P_4 - P_2 + P_6 & S &= P_1 + P_2 \\ T &= P_3 + P_4 & U &= P_5 + P_1 - P_3 - P_7 \\ P_1 &= A(F - H) & P_2 &= (A + B)H \\ P_3 &= (C + D)E & P_4 &= D(G - E) \\ P_5 &= (A + D)(E + H) & P_6 &= (B - D)(G + H) \\ P_7 &= (A - C)(E + F) \end{aligned}$$

Nunmehr ist

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Also: $T(n) = \Theta(n^{\log_2(7)}) = O(n^{2,81})$.

II. Sortieren und Suchen

- Heapsort **6**
- Quicksort **7**
- Vergleichskomplexität **8.1**
- Maximum und Minimum **9.1**
- Median und Selektion **9.2, 9.3**

Heapsort

Heapsort ist ein Verfahren, welches ein Array der Größe n ohne zusätzlichen Speicherplatz in Zeit $O(n \log n)$ sortiert.

Dies geschieht unter gedanklicher Verwendung einer baumartigen Datenstruktur, dem *heap*.

Aus einem *heap* kann man in logarithmischer Zeit das größte Element entfernen. Sukzessives Entfernen der größten Elemente liefert die gewünschte Sortierung.

Man kann auch neue Elemente in logarithmischer Zeit einfügen, was alternativ eine Vorgehensweise wie bei INSERTION-SORT erlaubt.

Heaps

Ein *heap* (dt. „Halde“) ist ein binärer Baum mit den folgenden Eigenschaften

H1 Die **Knoten und die Blätter** des Baums sind **mit Objekten beschriftet** (hier Zahlen).

H2 Alle Schichten sind gefüllt **bis auf den rechten Teil der Untersten**. M.a.W. alle Pfade haben die Länge d oder $d - 1$; hat ein Pfad die Länge d , so auch alle Pfade zu weiter links liegenden Blättern.

H3 Die Beschriftungen der Nachfolger eines Knotens sind **kleiner oder gleich** den Beschriftungen des Knotens.

Repräsentation von Heaps

Ein *heap* A wird im Rechner als **Array** A zusammen mit einer Zahl $heap-size[A]$ repräsentiert (und nicht als verzeigertes Baum)

- Der Eintrag 1 bildet die Wurzel des *heaps*.
- Der Elternknoten des Eintrags i ist der Eintrag $\lfloor i/2 \rfloor$.
- Die linken und rechten Nachfolger des Eintrags i sind die Einträge $2i$ und $2i + 1$. Übersteigt dieser Wert die Größe $heap-size[A]$, so existiert der entsprechende Nachfolger nicht.

Die Eigenschaften H1 und H2 sind für ein Array von Objekten **automatisch gegeben**. H3 bedeutet, dass $A[\lfloor i/2 \rfloor] \geq A[i]$ für alle $i \leq heap-size[A]$.

Beispiel:

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Die **Höhe** eines *heaps* der Größe n ist $\Theta(\log(n))$.

NB: $\log_b =$ „Anzahl der Male, die man durch b dividieren kann, bevor man 1 erreicht.“

Prozedur HEAPIFY: Spezifikation

Spezifikation von $\text{HEAPIFY}(A, i)$:

- Wir sagen, der Teilbaum mit Wurzel i erfülle die **Heapeigenschaft** wenn gilt $A[j] \leq A[j/2]$ für alle von i aus erreichbaren Knoten j .
- Vor Aufruf mögen die Teilbäume mit Wurzeln $2i$ und $2i + 1$ die Heapeigenschaft erfüllen.
- Dann erfüllt i nach Aufruf von $\text{HEAPIFY}(A, i)$ die Heapeigenschaft.
- Die *Menge* der Knoten des Teilbaums mit Wurzel i ändert sich dabei nicht; die Knoten können aber umstrukturiert werden. Der Heap außerhalb des Teilbaums bleibt unverändert.

NB Erfüllt ein Knoten die Heapeigenschaft so auch alle seine „Kinder“. Verletzt ein Knoten die Heapeigenschaft so auch alle seine „Vorfahren“.

Prozedur HEAPIFY: Implementierung

```
HEAPIFY( $A, i$ )
1   $l \leftarrow 2i$ 
2   $r \leftarrow 2i + 1$ 
3  if  $l \leq \text{heap-size}[A]$  und  $A[l] > A[i]$ 
4      then  $largest \leftarrow l$ 
5      else  $largest \leftarrow i$ 
6  if  $r \leq \text{heap-size}[A]$  und  $A[r] > A[largest]$ 
7      then  $largest \leftarrow r$ 
8  if  $largest \neq i$ 
9      then  $\text{exchange}A[i] \leftrightarrow A[largest]$ 
10     HEAPIFY( $A, largest$ )
```

Prozedur HEAPIFY: Laufzeitanalyse

Sei $h(i)$ die **Höhe** des Knotens i , also die Länge des längsten Pfades von i zu einem Blatt.

NB: $h(i) = O(\log(\text{heap-size}[A]))$.

Sei $T(h)$ die maximale Laufzeit von HEAPIFY(A, i) wenn $h(i) = h$.

Es gilt $T(h) = O(h)$, also $T(h) = O(\log(\text{heap-size}[A]))$.

Prozedur BUILD-HEAP

Wir wollen die Einträge eines beliebigen Arrays so permutieren, dass ein *heap* entsteht.

BUILD-HEAP(A)

1 $heap\text{-}size[A] \leftarrow length[A]$

2 **for** $i \leftarrow heap\text{-}size[A]/2$ **downto** 1 **do**

3 HEAPIFY(A, i)

4 ▷ Alle Teilbäume mit Wurzel $\geq i$ erfüllen die Heapeigenschaft

Nach Aufruf von BUILD-HEAP(A) enthält A dieselben Einträge wie zuvor, aber nunmehr bildet A einen *heap* der Größe n .

Prozedur BUILD-HEAP: Laufzeitanalyse

Ein *heap* der Größe n enthält maximal $\lceil n/2^{h+1} \rceil$ Knoten der Höhe h .

Die Laufzeit von $\text{BUILD-HEAP}(A)$ ist somit:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \lceil n/2^{h+1} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

Hier haben wir verwendet:

$$\begin{array}{l} \sum_{h=0}^{\infty} x^h = \frac{1}{1-x} \quad \left| \quad \frac{d}{dx} \right. \\ \sum_{h=0}^{\infty} hx^{h-1} = \frac{1}{(1-x)^2} \quad \left| \quad x \right. \\ \sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2} \end{array}$$

Also gilt $\sum_{h=0}^{\infty} h/2^h = (1/2)/(1/2)^2 = 2$ mit $x = 1/2$.

Merke: $\text{BUILD-HEAP}(A)$ läuft in $O(\text{length}[A])$.

Prozedur HEAP-SORT

```
HEAP-SORT( $A$ )
1  BUILD-HEAP( $A$ )
2  for  $i \leftarrow \text{length}[A]$  downto 2 do
3      exchange  $A[1] \leftrightarrow A[i]$ 
4       $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$ 
5      HEAPIFY( $A, 1$ )
```

Laufzeit von HEAP-SORT(A) mit $\text{length}[A] = n$ ist $O(n \log n)$.

Prioritätsschlangen

Eine **Prioritätsschlange** (*priority queue*) ist eine Datenstruktur zur Verwaltung einer Menge von Objekten, die linear geordnete Schlüssel als Attribute besitzen.

Eine Prioritätsschlange unterstützt die folgenden Operationen:

- $\text{INSERT}(S, x)$: Einfügen des Elements x in die Schlange S .
- $\text{MAXIMUM}(S)$ liefert das (ein) Element von S mit dem größten Schlüssel.
- $\text{EXTRACT-MAX}(S)$ liefert das (ein) Element von S mit dem größten Schlüssel und löscht es aus S .

Wird S als *heap* organisiert, so laufen alle drei Operationen jeweils in Zeit $O(\log(|S|))$, außerdem erfordert das einmalige Aufbauen eines *heaps* nur lineare Zeit.

Prozedur HEAP-INSERT

HEAP-INSERT(A, key)

```
1   $heap-size[A] \leftarrow heap-size[A] + 1$ 
2   $i \leftarrow heap-size[A]$ 
3  while  $i > 1$  and  $A[i/2] < key$  do
4       $A[i] \leftarrow A[i/2]$ 
5       $i \leftarrow i/2$ 
5   $A[i] \leftarrow key$ 
```

Verfolgt den Pfad vom ersten freien Blatt ($heap-size + 1$) zur Wurzel bis der Platz für key gefunden ist. Laufzeit $O(\log n)$.

Quicksort

```
QUICKSORT( $A, p, r$ )  
  ▷ Sortiere  $A[p..r]$   
1  if  $p < r$  then  
2     $q \leftarrow$  PARTITION( $A, p, r$ )  
3    QUICKSORT( $A, p, q - 1$ )  
4    QUICKSORT( $A, q + 1, r$ )
```

Die Prozedur PARTITION(A, p, r) arbeitet wie folgt: Man gruppiert die Elemente von $A[p..r]$ um und bestimmt einen Index $q \in \{p, \dots, r\}$ sodass nach der Umgruppierung gilt: $A[p..q - 1] \leq A[q] \leq A[q + 1..r]$.

Prozedur PARTITION

PARTITION(A, p, r)

1 $x \leftarrow A[r]$

2 $i \leftarrow p - 1$

3 **for** $j \leftarrow p$ **to** $r - 1$

4 **do if** $A[j] \leq x$

5 **then** $i \leftarrow i + 1$

6 exchange $A[i] \leftrightarrow A[j]$

7 exchange $A[i + 1] \leftrightarrow A[r]$

8 **return** $i + 1$

In Zeile 4 gilt die **Invariante**:

$$A[p..i] \leq x < A[i + 1..j - 1]$$

Laufzeit von QUICKSORT

Sei $n = r - p + 1$ die Größe des zu bearbeitenden Arrays.

Der Aufruf $\text{PARTITION}(A, p, r)$ hat Laufzeit $\Theta(n)$.

Sei $T(n)$ die Laufzeit von $\text{QUICKSORT}(A, p, r)$.

Es gilt $T(n) = T(n_1) + T(n_2) + \Theta(n)$ wobei $n_1 + n_2 = n - 1$.

Bester Fall: $n_1, n_2 = O(n)$, z.B., $= n/2$. Dann ist $T(n) = \Theta(n \log n)$.

Schlechtester Fall: $n_1 = O(n), n_2 = O(1)$ oder umgekehrt. Dann ist $T(n) = \Theta(n^2)$.

Randomisiertes Quicksort

Der schlechteste Fall tritt tatsächlich auf wenn das Array schon mehr oder weniger sortiert ist.

Beispiel: Buchungen sind nach Eingangsdatum sortiert, sollen nach Buchungsdatum sortiert werden.

Um diesen Effekt zu vermeiden, wählt man das Pivotelement **zufällig**:

RANDOMIZED-PARTITION(A, p, r)

- 1 $i \leftarrow \text{RANDOM}(p, r)$
- 2 $\text{exchange}A[r] \leftrightarrow A[i]$
- 3 **return** **PARTITION**(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)

- 1 **if** $p < r$ **then**
- 2 $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$
- 3 **RANDOMIZED-QUICKSORT**($A, p, q - 1$)
- 4 **RANDOMIZED-QUICKSORT**($A, q + 1, r$)

Erwartete Laufzeit

Die Laufzeit von RANDOMIZED-QUICKSORT ist nunmehr eine **Zufallsvariable** ohne festen Wert.

Wir bestimmen ihren **Erwartungswert**. Wir nehmen **vereinfachend** an, dass alle Elemente verschieden sind.

Wie groß wird der Teil $n_1 = q - p$ der Partition?

Das hängt vom **Rang** des Pivotelements ab.

Ist das Pivotelement das **Kleinste**, so ist $n_1 = 0$.

Ist das Pivotelement das **Zweitkleinste**, so ist $n_1 = 1$.

Ist das Pivotelement das **Drittkleinste**, so ist $n_1 = 2$.

Ist das Pivotelement das **$n - 1$ kleinste**, so ist $n_1 = n - 2$.

Ist das Pivotelement das **Größte**, so ist $n_1 = n - 1$.

Also gilt für den **Erwartungswert** der Laufzeit $T(n)$:

$$T(n) = \frac{1}{n} \left(\sum_{q=0}^{n-1} T(q) + T(n - q - 1) \right) + \Theta(n)$$

Explizite Bestimmung der erwarteten Laufzeit

Wir wissen bereits, dass $T(n) = O(n^2)$ somit $T(n)/n = O(n)$ und somit kann der erste und der letzte Summand durch $\Theta(n)$ absorbiert werden:

$$T(n) = \frac{1}{n} \left(\sum_{q=1}^{n-2} T(q) + T(n-q-1) \right) + \Theta(n) = \frac{2}{n} \sum_{q=1}^{n-2} T(q) + \Theta(n)$$

Wir **raten** $T(n) = O(n \log n)$ und probieren durch Induktion über n zu zeigen $T(n) \leq cn \ln n$ für ein **noch zu bestimmendes** $c > 0$.

Sei n groß genug und fest gewählt. Es gelte $T(q) \leq cq \ln q$ für alle $q < n$.

$$T(n) \leq \frac{2c}{n} \sum_{q=1}^{n-2} q \ln q + dn \quad (\text{das "d" kommt vom } \Theta(n))$$

Es ist $\sum_{q=1}^{n-2} q \ln q \leq \int_{q=1}^n q \ln q \, dq = \left[\frac{1}{2} q^2 \ln q - \frac{1}{4} q^2 \right]_1^n \leq \frac{1}{2} n^2 \ln n - \frac{1}{4} n^2$.

Mit $c \geq 2d$ bekommen wir also $T(n) \leq cn \ln n$ somit ist $T(n) = O(n \log n)$ erwiesen.

Untere Schranke für vergleichsbasiertes Sortieren

Sei $\mathcal{A}$ irgendein Algorithmus, welcher n Objekte sortiert, indem auf die Objekte nur über binäre Vergleiche der Form “ $o_1 \leq o_2?$ ” zugegriffen wird.

Also nicht durch explizites Lesen der Objekte.

Beispiel aus `stdlib.h`

```
void qsort(void A[], int size, int compare(void *,void *))
```

Anderes Beispiel: das Java Interface `Comparable`.

Wir behaupten: $\mathcal{A}$ erfordert $\Omega(n \log n)$ solche Vergleiche.

Beweis der unteren Schranke

Nehmen wir an, $\mathcal{A}$ führe V Vergleiche durch.

Welche Elemente hier verglichen werden, hängt i.a. vom Ausgang vorhergehender Vergleiche ab!

Die möglichen Ausgänge dieser Vergleiche partitionieren die möglichen Eingaben in 2^V Klassen.

Eingaben, die in die gleiche Klasse fallen, werden gleich behandelt.

Jede der $n!$ Permutationen von n verschiedenen Objekten erfordert unterschiedliche Behandlung.

Also muss gelten $2^V \geq n!$ oder $V \geq \log_2 n! = \Omega(n \log n)$.

[Beweis, dass $\log_2 n! = \Omega(n \log n)$.

$$\log n! = \sum_{i=1}^n \log_2 i \geq \sum_{i=n/2}^n \log_2 i \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \frac{n}{2} (\log_2(n) - 1) = \Omega(n \log n).]$$

Bestimmung des Maximums

Das Array A enthalte n **verschiedene** Zahlen.

Folgender Algorithmus bestimmt das Maximum der $A[i]$.

MAXIMUM(A, n)

▷ Bestimmt das größte Element von A , wenn $n \geq 0$

1 *kandidat* $\leftarrow A[1]$

2 **for** $i \leftarrow 2$ **to** n **do**

▷ *kandidat* $\geq A[1..i - 1]$

3 **if** $A[i] > \textit{kandidat}$ **then** *kandidat* $\leftarrow A[i]$

4 **return** *kandidat*

Die **Vergleichskomplexität** dieses Verfahrens beträgt $V(n) = n - 1$.

Soll heißen, $n - 1$ Größenvergleiche werden durchgeführt.

Ganz analog haben wir ein Verfahren MINIMUM, das das kleinste Element bestimmt.

Vergleichskomplexität des Maximums

Die **Vergleichskomplexität** des Problems „Maximumbestimmung“ ist die minimale Zahl von Vergleichen, die im **schlechtesten Fall** erforderlich sind, um das Maximum zu bestimmen.

Die Existenz des Verfahrens MAXIMUM belegt $V(n) \leq n - 1$.

Es gilt tatsächlich $V(n) = n - 1$.

Vergleichskomplexität des Maximums

- Sei M die Menge der Positionen im Array, an denen aufgrund der bis dato gemachten Vergleiche noch das Maximum stehen könnte.
- Am Anfang ist $M = \{1, \dots, n\}$. Am Ende muss $|M| = 1$ sein.
- Aus M entfernen können wir eine Position i nur dann, wenn ein Vergleich stattgefunden hat, in dem $A[i]$ das kleinere Element ist.
- Ein Vergleich entfernt also höchstens ein Element aus M .
- $n - 1$ Vergleiche sind erforderlich.
- Das gilt ganz gleich wie die Vergleiche ausgehen, also auch im **besten Fall**.

Maximum und Minimum gleichzeitig

Es gelte, simultan das größte und das kleinste Element in einem Array zu bestimmen.

Anwendung: **Skalierung** von Messwerten.

Durch Aufruf von MAXIMUM und dann MINIMUM erhalten wir einen Algorithmus für dieses Problem mit Vergleichskomplexität $2n - 2$.

Somit gilt für die Vergleichskomplexität $V(n)$ des Problems „Maximum und Minimum“

$$V(n) \leq 2n - 2$$

Ist das optimal?

Maximum und Minimum gleichzeitig

MAXIMUM-MINIMUM(A, n)

▷ Bestimmt das Maximum und das Minimum in $A[1..n]$

```
1  for  $i \leftarrow 1$  to  $\lfloor n/2 \rfloor$  do
2      if  $A[2i - 1] < A[2i]$ 
3          then  $B[i] \leftarrow A[2i - 1]; C[i] \leftarrow A[2i]$ 
4          else  $C[i] \leftarrow A[2i - 1]; B[i] \leftarrow A[2i]$ 
5  if  $n$  ungerade
6      then  $B[\lfloor n/2 \rfloor + 1] \leftarrow A[n]; C[\lfloor n/2 \rfloor + 1] \leftarrow A[n]$ 
7  return (MINIMUM( $B, \lfloor n/2 \rfloor$ ), MAXIMUM( $C, \lfloor n/2 \rfloor$ ))
```

- Die Elemente werden zunächst in $\lfloor n/2 \rfloor$ verschiedenen Paaren verglichen. Das letzte Paar besteht aus zwei identischen Elementen, falls n ungerade.
- Das Maximum ist unter den $\lfloor n/2 \rfloor$ „Siegern“; diese befinden sich in C .
- Das Minimum ist unter den $\lfloor n/2 \rfloor$ „Verlierern“; diese befinden sich in B .

Es gilt also $V(n) \leq \lceil \frac{3n}{2} \rceil - 2$. Das ist optimal.

Untere Schranke für Maximum und Minimum

Satz: Jeder Algorithmus, der simultan Minimum und Maximum eines Arrays A mit $\text{length}[A] = n$ bestimmt, benötigt mindestens $\lceil \frac{3n}{2} \rceil - 2$ Vergleiche.

Seien $K_{\max}$ und $K_{\min}$ die Mengen der Indizes i , für die $A[i]$ noch als Maximum bzw. Minimum in Frage kommen.

$K_{\max}$ enthält diejenigen i , für die $A[i]$ noch bei keinem Vergleich “verloren” hat.

Bei jedem Vergleich werden $K_{\min}$ und $K_{\max}$ jeweils höchstens um ein Element kleiner. Ein Vergleich ist ein

- **Volltreffer**, falls $K_{\min}$ und $K_{\max}$ kleiner werden,
- **Treffer**, falls nur eines von $K_{\min}$ und $K_{\max}$ kleiner wird,
- **Fehlschuss** sonst.

Untere Schranke für Maximum und Minimum

Wir werden zeigen:

Lemma: Für jeden Algorithmus gibt es eine Eingabe, bei der er nur $\lfloor \frac{n}{2} \rfloor$ Volltreffer landet.

Beweis des Satzes:

Am Anfang ist $|K_{\max}| = |K_{\min}| = n$, am Ende soll $|K_{\max}| = |K_{\min}| = 1$ sein.

Es sind also $2n - 2$ Elemente aus $K_{\min}$ und $K_{\max}$ zu entfernen.

Werden nur $\lfloor \frac{n}{2} \rfloor$ Volltreffer gelandet, muss es also noch

$$2n - 2 - 2\lfloor \frac{n}{2} \rfloor = 2\lceil \frac{n}{2} \rceil - 2$$

Treffer geben, also ist die Zahl der Vergleiche mindestens

$$\lfloor \frac{n}{2} \rfloor + 2\lceil \frac{n}{2} \rceil - 2 = \lceil \frac{3n}{2} \rceil - 2 .$$

Beweis des Lemmas über Volltreffer

Idee: Eingabe wird von einem Gegenspieler $\mathbb{A}$ (engl. **adversary**) während des Ablaufs konstruiert.

$\mathbb{A}$ merkt sich seine Antworten zu den Vergleichsanfragen “ $A[i] \leq A[j]$?”, und antwortet stets so, dass

- (1) die Antwort mit der erzeugten partiellen Ordnung konsistent ist.
- (2) möglichst kein Volltreffer erzielt wird.

Falls $i, j \in K_{\max} \cap K_{\min}$, sind i und j noch völlig frei.

$\leadsto$ $\mathbb{A}$ kann beliebig antworten, in jedem Fall ein Volltreffer.

In **jedem anderen** Fall kann $\mathbb{A}$ so antworten, dass nur ein Treffer oder Fehlschuss erzielt wird.

Ist z.B. $i \in K_{\max}$ und $j \in K_{\min}$, aber $i \notin K_{\min}$, so antwortet $\mathbb{A}$ mit $A[j] < A[i]$

$\leadsto$ Treffer, falls $j \in K_{\max}$, sonst Fehlschuss.

Also: Volltreffer nur, falls $i, j \in K_{\max} \cap K_{\min}$, das kann nur $\lfloor \frac{n}{2} \rfloor$ mal vorkommen.

Die Selektionsaufgabe

Die **Selektionsaufgabe** besteht darin, von n **verschiedenen** Elementen das i -**kleinste** (**sprich**: [ihstkleinste]) zu ermitteln.

Das i -kleinste Element ist dasjenige, welches nach aufsteigender Sortierung an i -ter Stelle steht.

Englisch: i kleinstes Element = ***ith order statistic***.

Das 1-kleinste Element ist das Minimum.

Das n -kleinste Element ist das Maximum.

Das $\lfloor \frac{n+1}{2} \rfloor$ -kleinste und das $\lceil \frac{n+1}{2} \rceil$ -kleinste Element bezeichnet man als **Median**.

Ist n gerade, so gibt es **zwei Mediane**, ist n ungerade so gibt es **nur einen**.

Anwendung des Medians

Fakt: Sei $x_1, \dots, x_n$ eine Folge von Zahlen. Der Ausdruck $S(x) = \sum_{i=1}^n |x - x_i|$ nimmt sein Minimum am Median der x_i an.

Beispiele

- n Messwerte x_i seien so zu interpolieren, dass die Summe der absoluten Fehler minimiert wird. Lösung: Median der x_i .
- n Städte liegen auf einer Geraden an den Positionen x_i . Ein Zentrallager sollte am Median der x_i errichtet werden um die mittlere Wegstrecke zu minimieren (unter der Annahme, dass jede Stadt gleich oft angefahren wird.)
- Analoges gilt auch in 2D bei Zugrundelegung der **Manhattandistanz**.

Vergleichskomplexität der Selektionsaufgabe

- Durch **Sortieren** kann die Selektionsaufgabe mit Vergleichskomplexität $\Theta(n \log n)$ gelöst werden, somit gilt für die Vergleichskomplexität $V(n)$ der Selektionsaufgabe: $V(n) = O(n \log n)$.
- $V(n) = \Omega(n)$ ergibt sich wie beim Maximum. Mit weniger als $n - 1$ Vergleichen kann ein Element nicht als das i -kleinste bestätigt werden.
- Tatsächlich hat man $V(n) = \Theta(n)$.

Selektion mit mittlerer Laufzeit $\Theta(n)$

RANDOMIZED-SELECT(A, p, r, i)

▷ Ordnet $A[p..r]$ irgendwie um und bestimmt den Index des i -kleinsten Elements in $A[p..r]$

1 **if** $p = r$ **then return** p

2 $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

3 $k \leftarrow q - p + 1$

4 **if** $i \leq k$

5 **then return** RANDOMIZED-SELECT(A, p, q, i)

6 **else return** RANDOMIZED-SELECT($A, q + 1, r, i - k$)

Laufzeit (und Vergleichskomplexität) im schlechtesten Falle: $\Theta(n^2)$.

Mittlere Laufzeit von RANDOMIZED-SELECT

Für den Erwartungswert $V(n)$ der Laufzeit von $\text{RANDOMIZED-SELECT}(A, p, r, i)$, wobei $n = r - p + 1$, gilt die Rekurrenz:

$$T(n) \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

Diese Rekurrenz hat die Lösung $T(n) = O(n)$ wie man durch Einsetzen und Induktion bestätigt.

Lineare Laufzeit im schlechtesten Fall

SELECT(A, p, r, i)

▷ Bestimmt den Index des i -kleinsten Elements in $A[p..r]$

1 **if** $p = r$ **then return** p

2 Teile die $A[p..r]$ in Fünfergruppen auf (plus eventuell eine kleinere Gruppe)

3 Bestimme den Median jeder Gruppe durch festverdrahtete Vergleiche

4 Bestimme durch rekursiven Aufruf von SELECT den Median dieser Mediane.

5 Vertausche in A diesen Median mit $A[r]$

6 $q \leftarrow$ PARTITION(A, p, r)

7 $k \leftarrow q - p + 1$

8 **if** $i \leq k$

9 **then return** SELECT(A, p, q, i)

10 **else return** SELECT($A, q + 1, r, i - k$)

Worst-case Laufzeit von SELECT

Sei $T(n)$ die *worst case* Laufzeit von SELECT.

- Gruppenbildung und individuelle Mediane: $O(n)$.
- Bestimmung des Medians der Mediane: $T(n/5)$.
- Der Median der Mediane liegt oberhalb und unterhalb von jeweils mindestens $\frac{3n}{10}$ Elementen.
- Die größere der beiden „Partitionen“ hat also weniger als $\frac{7}{10}$ Elemente.
- Der rekursive Aufruf auf einer der beiden „Partitionen“ erfordert also $T(\frac{7n}{10})$.

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

Die Lösung ist $T(n) = O(n)$ wie man durch Einsetzen bestätigt.

NB Die Lösung von $T(n) = T(n/5) + T(8n/10) + O(n)$ ist $O(n \log n)$.