

ALGORITHMEN UND DATENSTRUKTUREN

TEIL 2: SORTIEREN UND SUCHEN

Martin Hofmann

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

16. April 2016



- 1 HEAPSORT
 - Heaps
 - Operationen auf Heaps
- 2 PRIORITÄTSSCHLANGEN
- 3 QUICKSORT
 - Randomisiertes Quicksort
- 4 VERGLEICHSKOMPLEXITÄT
- 5 MEDIAN UND SELEKTION



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. haben alle Pfade 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 $\text{heap-size}[A]$ repräsentiert (und nicht als „verzeigerter“ Baum)

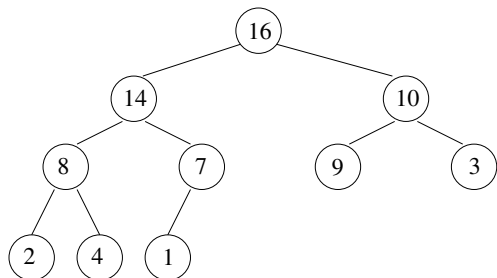
- Der Eintrag 1 bildet die Wurzel des **heaps**.
- Der Elternknoten des Eintrags i ist der Eintrag $\lfloor i/2 \rfloor$.
- Die linken und rechten Nachfolger des Eintrags i sind die Einträge $2i$ und $2i + 1$. Übersteigt dieser Wert die Größe $\text{heap-size}[A]$, so existiert der entsprechende Nachfolger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



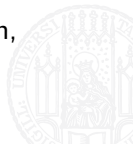
BEISPIEL

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

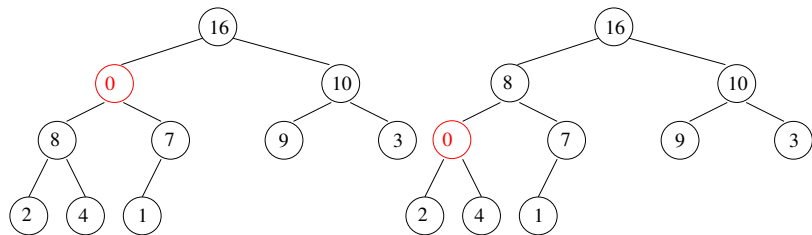
PROZEDUR Heapify

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



AUFRUF VON HEAPIFY(A,2)

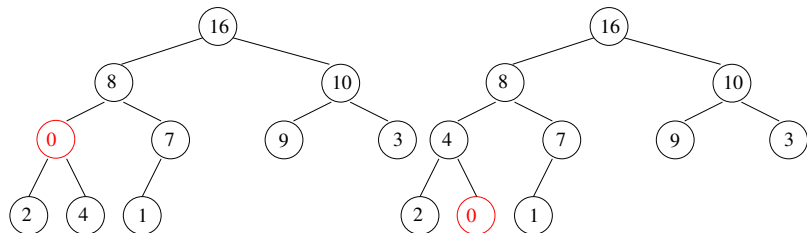
Achtung: anderes Array mit $A[2] = 0$.



Abarbeitung der Programmzeilen 1–9 von Heapify(A, 2).



BEISPIEL (FORTS.)



Abarbeitung der Programmzeile 10 von $\text{Heapify}(A, 2)$, also $\text{Heapify}(A, 4)$.

Erst jetzt ist der Aufruf von $\text{Heapify}(A, 2)$ vollständig abgearbeitet.



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 $\text{Heapify}(A, i)$ 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.

PROZEDUR BUILD-HEAP

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 Teilb. 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:

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

MERKE

$\text{Build-Heap}(A)$ läuft in $O(\text{length}[A])$.



KONVERGENZ UND WERT DER SUMME

Wir haben verwendet:

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x} \quad \left| \quad \frac{d}{dx} \right.$$

$$\sum_{h=0}^{\infty} hx^{h-1} = \frac{1}{(1-x)^2} \quad \left| \quad x \right.$$

$$\sum_{h=0}^{\infty} hx^h = \frac{x}{(1-x)^2}$$

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



PROZEDUR Heap-Sort

PROZEDUR HEAP-SORT

```
Heap-Sort(A)
1  Build-Heap(A)
2  for i ← length[A] downto 2 do
3      exchange A[1] ↔ A[i]
4      heap-size[A] ← heap-size[A] − 1
5      Heapify(A, 1)
```

Laufzeit von $\text{Heap-Sort}(A)$ 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

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 $O(\log n)$.



WEITERE OPERATIONEN

HERABSETZEN EINES EINTRAGS (DECREASE-KEY)

Will man einen Eintrag verkleinern (sinnvoll etwa, wenn die Einträge Objekte mit numerischen Schlüsseln sind (z.B.: Artikel mit Preis)), so führt man die Ersetzung durch und ruft dann Heapify auf.

LÖSCHEN EINES EINTRAGS (HEAP-DELETE)

Man ersetzt den zu löschenden Eintrag durch den rechts unten stehenden $A[\text{heap-size}[A]]$ und korrigiert dann die Heapeigenschaft sowohl nach oben, als auch nach unten hin. Details als Übung.

SCHLÜSSEL ERHÖHEN

Will man einen Schlüssel nach oben hin verändern, so bietet sich Löschen gefolgt von erneuten Einsetzen an.

QUICKSORT

PROZEDUR 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

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**:

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

Zeile 6 tut nichts, wenn $A[i..j - 1]$ leer ist.



ILLUSTRATION VON PARTITION

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

[]()3	14	4	7	1	11	6	8	10	12	5	<u>7</u>
[3]()	14	4	7	1	11	6	8	10	12	5	<u>7</u>
[3](14)	4	7	1	11	6	8	10	12	5	<u>7</u>
[3	4](14)	7	1	11	6	8	10	12	5	<u>7</u>
[3	4	7](14)	1	11	6	8	10	12	5	<u>7</u>
[3	4	7	1](14)	11	6	8	10	12	5	<u>7</u>
[3	4	7	1](14	11)	6	8	10	12	5	<u>7</u>
[3	4	7	1	6](11	14)	8	10	12	5	<u>7</u>
[3	4	7	1	6](11	14	8	10	12)	5	<u>7</u>
[3	4	7	1	6	5](14	8	10	12	11)	<u>7</u>
[3	4	7	1	6	5](<u>7</u>	8	10	12	11)	14

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: $n_1, n_2 = n/2 \pm 1$. Dann ist $T(n) = \Theta(n \log n)$.

SCHLECHTESTER FALL: $n_1 = n - 1, n_2 = 0$ oder umgekehrt.

Passiert das immer wieder, so wird $T(n) = \Theta(n^2)$.



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*, indem man vorab das letzte Element mit einem zufälligen vertauscht.

Wir werden zeigen, dass dadurch der Erwartungswert der Laufzeit $O(n \log(n))$ wird.

NB: Hierbei wird über die (gleichverteilten) Zufallsvertauschungen bei fester Eingabe gemittelt.

Nicht verwechseln mit *average case*!



RANDOMIZED QUICKSORT

PROZEDUR RANDOMIZED-PARTITION

```
Randomized-Partition( $A, p, r$ )  
1  $i \leftarrow \text{Random}(p, r)$   
2  $\text{exchange} A[r] \leftrightarrow A[i]$   
3 return Partition( $A, p, r$ )
```

PROZEDUR RANDOMIZED-QUICKSORT

```
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 $q < n$.

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



FORTSETZUNG

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

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

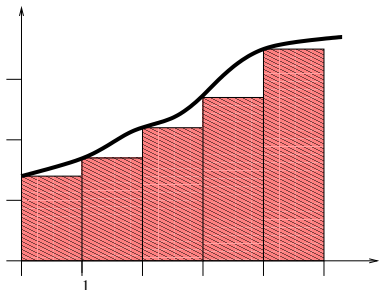
$T(n) = O(n \log n)$ erwiesen.



ABSCHÄTZUNG EINER SUMME DURCH EIN INTEGRAL

SATZ

Ist f monoton wachsend, so gilt $\sum_{k=0}^n f(k) \leq \int_{x=0}^{n+1} f(x) dx$



Die rosa schraffierte Fläche entspricht dem Wert der Summe $\sum_{k=0}^n f(k)$ für $n = 4$.

Durch ähnliche Bilder kann man auch andere Abschätzungen begründen, z.B.: $\sum_{k=1}^{n+1} f(k) \geq \int_{x=0}^n f(x) dx$.



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 $V \geq \log n! = \Omega(n \log n)$.



BESTIMMUNG DES MAXIMUMS

Das Array A enthalte n *verschiedene* Zahlen.
 Folgender Algorithmus bestimmt das Maximum der $A[i]$.

PROZEDUR MAXIMUM

Maximum(A, n)

- ▷ Bestimmt das größte Element von $A[1..n]$, wenn $n \geq 1$
- 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.



MAXIMUMBESTIMMUNG MIT KO-SYSTEM

PROZEDUR MAXIMUM-KO

Maximum-KO(A, p, r)

- ▷ Bestimmt das größte Element von $A[p..r]$, wenn $r - p \geq 0$
- 1 **if** $p = r$ **then return** $A[p]$ **else**
- 2 $q \leftarrow (p + r)/2$
- 3 $l \leftarrow$ Maximum-KO(A, p, q)
- 4 $r \leftarrow$ Maximum-KO($A, q + 1, r$)
- 5 **return** $\max(l, r)$ ▷ Erfordert 1 Vergleich

Auch Maximum-KO tätigt $n - 1$ Vergleiche, wobei $n = r - p + 1$.

GRUND: Ein binärer Baum mit n Blättern hat $n - 1$ innere Knoten.



VERGLEICHSKOMPLEXITÄT DES MAXIMUMS

Die **Vergleichskomplexität** des Problems „Maximumbestimmung“ ist die minimale Zahl von Vergleichen $V(n)$, die im **schlechtesten Fall** erforderlich sind, um das Maximum zu bestimmen:

$$V(n) = \min_{\text{Alg}} \max_A V(\text{Alg}, A)$$

wobei Alg über alle Algorithmen für die Maximum-Aufgabe rangiert und A über alle Eingaben der Größe n .

Hier ist $V(\text{Alg}, A)$ die Zahl der Vergleiche, die Alg bei Eingabe A tätigt.

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

Wir zeigen jetzt $V(n) = n - 1$.



BEWEIS DER UNTEREN SCHRANKE

- 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?



EIN BESSERER ALGORITHMUS

PROZEDUR MAXIMUM-MINIMUM

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($B, \lceil n/2 \rceil$), Maximum($C, \lceil n/2 \rceil$)))



ERLÄUTERUNG

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

$$\left\lceil \frac{3n}{2} \right\rceil - 2 \text{ Vergleiche.}$$

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

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.



BEWEIS, FORTSETZUNG

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

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

Treffer geben, also ist die Zahl der Vergleiche mindestens

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



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 $\left\lfloor \frac{n}{2} \right\rfloor$ mal vorkommen.



DIE SELEKTIONSAUFGABE

Die **Selektionsaufgabe** besteht darin, von n **verschiedenen** Elementen das **i -kleinste** (sprich: [ihtkleinste]) zu ermitteln. Das i -kleinste Element ist dasjenige, welches nach aufsteigender Sortierung an i -ter Stelle steht.

ENGLISCH: i kleinstes Element = **i th 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

SATZ

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.

BEGRÜNDUNG: Liegen n_1 Werte oberhalb von x und n_2 Werte unterhalb von x (wobei $n_1 + n_2 = n$, so gilt für kleines $d > 0$, dass $S(x - d) = S(x) + n_1 d - n_2 d \dots$

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 **Manhattan**distanz.



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 $\Theta(n)$

PROZEDUR RANDOMIZED-SELECT

Randomized-Select(A, p, r, i)

▷ Ordnet $A[p..r]$ um und best. den Ind. des i -kl. El. 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 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

PROZEDUR SELECT

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 $A[i]$ in Fünfergruppen auf

3 Bestimme den Median jeder Gruppe

4 Bestimme den Median dieser Mediane
durch rekursiven Aufruf von Select

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 $O(n \log n)$.

