# Warum Bottom-up Heapsort?

Die Idee hinter bottom-up Heapsort ist, die Feststellung von eben aufzugreifen und die Konstante bei $n \log n$ kleiner zu bekommen.

Bottom-up Heapsort funktioniert ganz genau so wie Standard-Heapsort, nur die reheap-Prozedur wird anders implementiert. Am einfachsten stellt man sich das so vor, dass das einsinkende Element zunächst bis zu einem Blatt sinkt, unabhängig von seinem eigenen Wert. Dazu benötigt man in jedem Schritt nur einen Vergleich!

Anschließend steigt das eingesunkene Element so lange auf, bis das über ihm liegende Element kleiner als dieses (oder gleich diesem) ist. Auch hier genügt ein Vergleich pro Schritt.

# Analyse von Bottom-up Heapsort

Da beim Absinken jeweils nur ein Vergleich nötig ist (nämlich der Vergleich $a[2i] < a[2i+1]$?), werden für das Absinken aller $n$ Elemente insgesamt nicht mehr als $n \log n$ Vergleichsoperationen benötigt.
Aber wieviel kostet das anschließende Aufsteigen?

Wir benutzen noch einmal das Ergebnis über die unendliche Summe der Terme $i/2^i$.

Offensichtlich werden weniger als die Hälfte aller $n$ Elemente um eine Ebene aufsteigen, weniger als ein Viertel um zwei, und so weiter – also weniger als $n/2^i$ um $i$ Ebenen.

Die Gesamtzahl der Aufstiege ist also kleiner als $n$ mal die unendliche Summe der $i/2^i$, und damit in $O(n)$.

# Analyse (Fortsetzung)

Auf der vorherigen Folie haben wir ein bisschen *geschummelt*...

Die Behauptung, die Zahl der um $i$ Ebenen steigenden Elemente wäre nicht größer als $n/2^i$, ist zwar im Durchschnitt plausibel.

Aber im worst case kann es durchaus sein, dass viele Elemente weit nach oben steigen, um dann wieder von anderen verdrängt und weiter nach unten geschoben zu werden!

Daher braucht man eine genauere Untersuchung, um die Komplexität von Bottom-up Heapsort im worst case bzw. im average case zu analysieren.

Die folgenden Ergebnisse geben wir hier ohne Beweis an:

Bottom-up Heapsort benötigt $1.5n \log n + o(n \log n)$ Vergleiche im worst case, bzw. $n \log n + o(n \log n)$ im average case.

# Ultimatives Heapsort

Wir machen zunächst ein Gedankenspiel:

Wenn wir zur Sortierung von $n$ Elementen $2n$ Speicherplätze verwenden könnten, würden wir zu den Elementen $a[1, \ldots, n]$ noch weitere $n$ Elemente mit Wert $\infty$ legen und dann Heapsort durchführen bis alle $n$ Elemente einsortiert sind.

Welchen Aufwand hätten wir dann?

Wir brauchen $O(n)$ Vergleiche für den Heapaufbau.

Aber dann genügen $n \log n$ Vergleiche für das Heraussortieren der Elemente $a[1]$ bis $a[n]$ an die $n$ hinzugefügten Plätze.

Warum nur $n \log n$, also $1 \cdot \log n$ pro Element?

> Weil die einsinkenden Elemente immer den Wert $\infty$ haben, d.h. wir lassen sie wie bei Bottom-up Heapsort ganz nach unten rutschen, aber ohne Aufstieg!

# Plan für Ultimate-Heapsort

Wir wollen aber ein In-place Sortierverfahren verwenden, d.h. den zusätzlichen Speicherplatz haben wir nicht.

Was können wir tun, um den Gedanken der vorigen Folie dennoch weiterzuverfolgen?

Wir müssen dafür sorgen, dass die erste Hälfte des Eingabe-Arrays die kleinen, und die zweite Hälfte die großen Elemente enthält, d.h.

$$i < \lceil n/2 \rceil,\ j \geq \lceil n/2 \rceil \quad \Longrightarrow \quad a[i] < a[j]$$

Wenn wir das geschafft haben, können wir die $a[j]$ mit $j \geq \lceil n/2 \rceil$ behandeln als wenn sie den Wert $a[j] = \infty$ hätten. Damit hätten wir mit Aufwand $\frac{n}{2} \log n$ die Hälfte des Arrays sortiert und müssten nur noch einen rekursiven Aufruf auf der halben Größe machen, um den Rest zu sortieren. Die Details folgen auf den nächsten Folien.

# Die ersten Aktionen

Wir müssen die Bedingung erfüllen, dass alle Elemente der ersten Array-Hälfte kleiner als alle Elementen der zweiten Hälfte sind. (OBdA seien alle Elemente verschieden!)

Dazu berechnen wir zuerst den Median der Elemente $a[1, \ldots, n]$.
Die Medianberechnung ist in Linearzeit möglich - das werden wir nachher zeigen.

Nun benutzen wir den Median als Pivot-Element und machen wie bei Quicksort eine Aufteilung in zwei Teile, wobei im ersten Teil alle Elemente liegen werden, die kleiner oder gleich dem Median sind (also genau die Hälfte!), und im rechten Teil die, die größer als der Median sind.

Jetzt liegt die gewünschte Ausgangsposition vor. Wir können in den ersten $n/2$ Elementen zunächst einen Heap aufbauen und diese Elemente dann wie bei Standardheapsort eines nach dem anderen „herauspflücken“ und an das Ende des Arrays sortiert einfügen.

# Der Rest

Wie sieht unser Array $a[1, \ldots, n]$ nun aus?

Durch die beschriebene Prozedur war zunächst das kleinste der $a[i]$ auf Platz $a[n]$ getauscht worden, dann das Zweitkleinste auf Platz $a[n-1]$ usw.

Machen Sie sich bitte selbst klar, dass dabei nie eines der im Quicksort-Schritt in der zweiten Hälfte des Arrays gelandeten Elemente (also derer, die größer als der Median sind), an die Position $a[1]$ gelangen kann.

Es liegen nun also in $a[1, \ldots, \lfloor n/2 \rfloor]$ die noch unsortierten Array-Elemente, die allesamt größer als der Median sind. Der übrige Teil ist bereits (von groß nach klein) sortiert.

Nun rufen wir rekursiv das Programm wieder auf, und zwar mit der Eingabe $a[1, \ldots, \lfloor n/2 \rfloor]$.

## Analyse

Wir gehen von einem Algorithmus zur Medianberechnung aus, der mit einer linearen Anzahl Vergleiche auskommt. Der anschließende Quicksortschritt und der Heapaufbau auf der ersten Hälfte des Arrays brauchen ebenfalls maximal linear viele Vergleiche, so dass wir höchstens $c \cdot n$ (für geeignetes $c$) Vergleiche verbraucht haben, bis wir mit dem *Herauspflücken* beginnen.

Wir sortieren $n/2$ Elemente ein, wobei das getauschte Element ziemlich genau aus $\log n$ Höhe bis ganz nach unten einsinkt.

Man beachte: Dabei wird der Heap jedesmal um ein Element kleiner!

Zum Schluss ruft sich die Prozedur rekursiv mit halb so großem Eingabe-Array auf. Das bedeutet:

$$T(n) = c \cdot n + \tfrac{n}{2} \log n + T(\tfrac{n}{2})$$

Diese Rekursionsformel hat offenbar die Lösung

$$T(n) = n \log n + 2c \cdot n.$$

# Ein Beispiel

Wir wählen $n = 31$ und der Einfachheit halber seien die zu sortierenden Array-Elemente gerade die Zahlen 1 bis 31. Der Median ist dann 16, und die Aufteilung nach dem Quicksort-Schritt sei wie folgt (grün: $a[1]$ bis $a[16]$, blau: $a[17]$ bis $a[31]$)

5 15 10 1 4 8 16 11 7 2 6 3 13 12 9 14
25 26 28 17 31 30 23 22 18 29 19 20 27 24 21

Als nächstes wird der grüne Teil zum Heap gemacht. Ergebnis:

1 2 3 7 4 8 9 11 15 5 6 10 13 12 16 14
25 26 28 17 31 30 23 22 18 29 19 20 27 24 21

Nun 1 mit 21 tauschen und 21 ganz nach unten sinken lassen:

2 4 3 7 5 8 9 11 15 17 6 10 13 12 16 14
25 26 28 21 31 30 23 22 18 29 19 20 27 24 1

Jetzt 2 mit 24 tauschen, Einsinkelemente sind 3, 8, 10, 22.

# Beispiel (Forts.)

3 4 8 7 5 10 9 11 15 17 6 22 13 12 16 14
25 26 28 21 31 30 23 24 18 29 19 20 27 2 1

Jetzt 3 mit 27 tauschen, Einsinkelemente sind 4, 5, 6, 30.

4 5 8 7 6 10 9 11 15 17 30 22 13 12 16 14
25 26 28 21 31 27 23 24 18 29 19 20 3 2 1

Jetzt 4 mit 20 tauschen, Einsinkelemente sind 5, 6, 17, 21.

5 6 8 7 17 10 9 11 15 21 30 22 13 12 16 14
25 26 28 20 31 27 23 24 18 29 19 4 3 2 1

Nun machen wir einen Zeitsprung um 12 weitere Schritte. Die „blauen“Elemente (17 bis 31) liegen (unsortiert) in den ersten Komponenten, also als $a[1, \ldots, 15]$, der Rest des Arrays sieht genau so aus:

16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Den Rest erledigt der rekursive Aufruf.

# Medianberechnung in Linearzeit

Um den Median von $n$ Array-Elementen zu berechnen, d.h. den Index $r$ zu finden, für den $|\{i \mid a[i] \leq a[r]\}| \geq n/2$ und $|\{i \mid a[i] \geq a[r]\}| \geq n/2$ gilt, sollen wir nur $O(n)$ Vergleiche durchführen. Dazu müssen einige Tricks angewendet werden.

Wie so oft, benötigen wir auch hier zunächst ein nachweislich gutes Pivot-Element, um eine Aufteilung in zwei annähernd gleich große Teile zu erreichen.

Nach der Aufteilung müssen wir nicht in beiden Teilen weiter suchen, denn der Median kann bei einer Aufteilung, bei der der erste Teil größer als der zweite ist, nur im ersten Teil liegen, und ebenso im anderen Fall nur im zweiten Teil.

Allerdings brauchen wir jetzt nicht mehr den Median, sondern das $k$-kleinste Element für ein bestimmtes $k \in \{1, \ldots, \lceil n/2 \rceil\}$.

# Erster Schritt: Median aus 5

Zunächst suchen wir ein gutes Pivot-Element.

Wir berechnen aus je 5 Elementen einen *lokalen* Median.
Dazu braucht man 6 Vergleiche:

Die Elemente seien $a, b, c, d, e$.

IF $a > b$ THEN vertausche$(a, b)$ ENDIF;

IF $c > d$ THEN vertausche$(c, d)$ ENDIF;

IF $a > c$ THEN vertausche$(a, c)$; vertausche$(b, d)$ ENDIF;

Nun ist $a$ kleiner als $b$, $c$ und $d$, daher nicht der Median.

IF $b > e$ THEN vertausche$(b, e)$ ENDIF;

IF $b > c$ THEN vertausche$(b, c)$; vertausche$(e, d)$ ENDIF;

Nun ist $b$ kleiner als $c$, $d$ und $e$, daher nicht der Median.
Außerdem ist $d$ größer als $a$, $b$ und $c$, also auch nicht der Median.
Ein Vergleich zwischen $c$ und $e$ genügt nun zur Ermittlung des gesuchten Medians. (Das kleinere von beiden ist der Median!)

## Zweiter Schritt: Median der Mediane

Bisher haben wir bei $n/5$ Blöcken jeweils 6 Vergleiche verwendet, um die Blockmediane zu finden.
Der Gesamtaufwand ist hier also $\frac{6}{5}n$ Vergleiche.

Wenn wir aus jedem Fünferblock des Eingabe-Arrays jeweils einen Blockmedian ermittelt haben, können wir von diesen $\lceil \frac{n}{5} \rceil$ Elementen rekursiv den Median berechnen.

Welchen Aufwand brauchen wir hierfür?

Die Anzahl Vergleiche der Gesamtprozedur sei wie üblich durch $T(n)$ (bei Eingabe eines Arrays $a[1, \ldots, n]$) gegeben.

Der Aufwand dieser Rekursion ist also $T(\frac{n}{5})$.

Es sollte klar sein, dass dieser Median der Blockmediane nicht der wirkliche Median sein muss – aber er ist ein gutes Pivotelement.

## Dritter Schritt: Feld aufteilen

Nun teilen wir mit dem ermittelten Median der Blockmediane als Pivotelement den gesamten Array auf in zwei Teile, von denen der erste, $a[1, \ldots, m]$, die Elemente enthält, die kleiner als das Pivotelement sind, der zweite, $a[m+1, \ldots, n]$, die übrigen.

Die folgende Feststellung ist entscheidend für diese Methode.

Es gilt: $\frac{3}{10}n \leq m \leq \frac{7}{10}n.$

Denn: Mit jedem Blockmedian, der kleiner als das Pivot-Element ist, sind insgesamt drei Elemente aus diesem Block im linken Teil. Das trifft bei der Hälfte der $n/5$ Blöcke, also bei $n/10$ Blöcken zu!

Auf der rechten Seite gilt das gleiche analog.

Damit ist klar, dass wir beim rekursiven Aufruf zum Ermitteln des gesuchten Elements nur mit einer Feldgröße bis $\frac{7}{10}n$ rechnen müssen.

## Vierter Schritt: Rekursion

Nun wollen wir rekursiv das gesuchte Element in einem der beiden Teile finden. Wenn wir also das $k$-kleinste Element in $a[1,\ldots,n]$ suchen, dann unterscheiden wir so:

Ist $k \leq m$? Wenn ja: Suche das $k$-kleinste Element in $a[1,\ldots,m]$.
Wenn nein: Suche das $(k-m)$-kleinste Element in $a[m+1,\ldots,n]$.

Dass damit das korrekte Element ermittelt wird, ist klar.
Aber welchen Gesamtaufwand haben wir nun?

Zunächst haben wir $\frac{6}{5}n$ Vergleiche für die Blockmediane verbraucht, dann $T(\frac{n}{5})$ für die erste Rekursion. Danach maximal $n$ Vergleiche für den Quicksort-Schritt (tatsächlich genügen weniger), und zum Schluss noch einmal rekursiv $T(\frac{7}{10}n)$. Es folgt

$$T(n) \leq \tfrac{6}{5}n + T(\tfrac{n}{5}) + n + T(\tfrac{7}{10}n).$$

Mit dem Master-Theorem II folgt daraus lineare Anzahl Vergleiche.