1. Übung zur Vorlesung Algorithmentechnik am 23.10.2008

Übersicht

- Organisatorisches
- Landau-Symbole & Laufzeiten
- Amortisierte Analyse
- Rekursionsauflösung
- Heaps und Fibonacci Heaps

Organisatorisches

Algorithmentechnik-Homepage (mit Forum)
http://i11www.iti.uni-karlsruhe.de/teaching/WS_0809/algotech/

Skript

- auf der Homepage
- zusätzliche Kapitel werden gerade eingearbeitet

Übungsleiter

- Reinhard Bauer (rbauer@ira.uka.de)
- Marcus Krug (krug@ira.uka.de)

Vorlesungs/Übungsbetrieb

- kein regelmäßiger Rhythmus
- Übersicht auf der Homepage

	Di		Do
21.10	Vorlesung	23.10	Übung
28.10	Vorlesung	30.10	Vorlesung
4.11	Übung	6.11	Vorlesung
11.11	Vorlesung	13.11.	Vorlesung

Übungsblätter

- Insgesamt 6 oder 7 Übungsblätter
- Blätter werden jeden zweiten Dienstag online gestellt
- Bearbeitungszeit: 2 Wochen
- vorraussichtlicher Plan:

Ausgabe 1. Blatt	21.10.2008
Ausgabe 2. Blatt	04.11.2008
Ausgabe 3. Blatt	18.11.2008
Ausgabe 4. Blatt	02.12.2008
Ausgabe 5. Blatt	16.12.2008
Ausgabe 6. Blatt	13.01.2009
Ausgabe 7. Blatt ?	27.01.2009

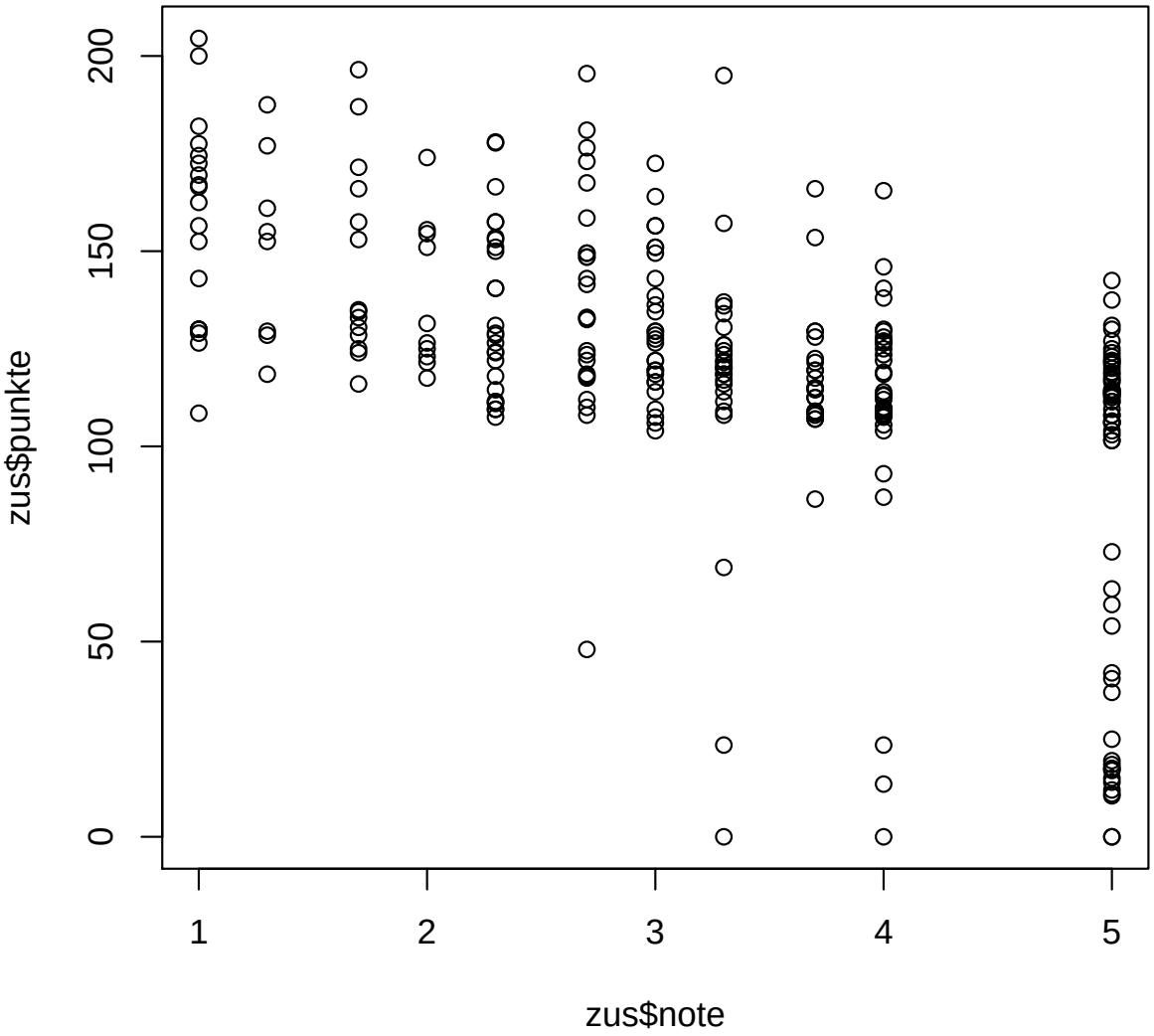
Übungsblätter

- Abgabe immer Dienstags in Kasten am Sekretariat Wagner (gegenüber von Raum 319 im Informatik-Hauptgebäude 50.34)
- Die korrigierten Blätter werden an diesem Kasten zur Abholung ausgelegt
- **Keine Matrikelnummer auf das Blatt schreiben**
- Korrekturzeitraum: 2 Wochen

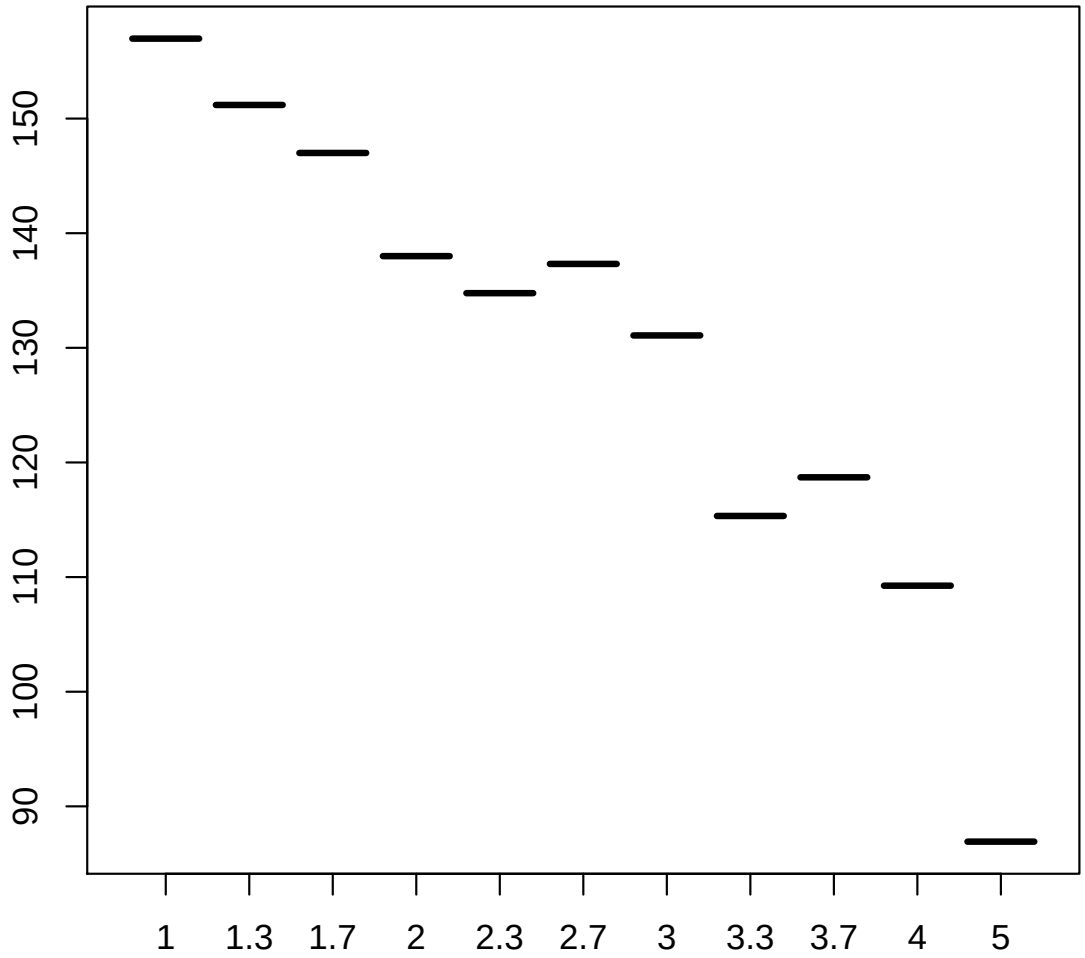
Übungsblätter

- Besprechung der Blätter in der Übung
- Lösungshinweise auf der Homepage
- Keinen Übungsschein

Wieso die Algorithmentechnik Übungsblätter?



Wieso die Algorithmentechnik Übungsblätter?



Klausurentermine (vorläufig)

- Haupttermin: 26.02.2009
- Nachschreibetermin: 09.04.2009

Frage: Ich kann mich an nichts erinnern. Wo kann ich den Stoff der heutigen Übung nachlesen?

Antwort:

- Algorithmentechnik-Skript
- Informatik II Vorlesungsfolien
- Cormen/Leiserson/Stein: Introduction to Algorithms

Wiederholung Landau-Symbole

$f \in O(g)$

- f wächst (asymptotisch) nicht wesentlich schneller als g
- Definition: $0 \leq \limsup_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| < \infty$

$f \in o(g)$

- f wächst (asymptotisch) langsamer als g
- Definition: $\lim_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| = 0$

Bemerkungen:

- Häufig auch äquivalente Definitionen mit Quantoren üblich
- Verallgemeinerung $x \to a$ möglich

$f \in \Omega(g)$

- f wächst (asymptotisch) mindestens so schnell wie g
- Definition: $0 < \liminf_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| \leq \infty$
- Äquivalente Definition: $g \in O(f)$

$f \in \omega(g)$

- f wächst (asymptotisch) schneller als g
- Definition: $\lim_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| = \infty$
- Äquivalente Definition: $g \in o(f)$

$f \in \Theta(g)$

- f wächst (asymptotisch) genauso schnell wie g
- Definition: $0 < \liminf_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| \leq \limsup_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| < \infty$
- Äquivalente Definition: $f \in O(g)$ und $g \in O(f)$

Wiederholung Laufzeiten

Sei $t_{\mathcal{A}}(x)$ die Anzahl Operationen, die ein Algorithmus $\mathcal{A}$ bei Eingabe x ausführt.

Die **Worst-case-Laufzeit** von $\mathcal{A}$ ist:

$$T_{\mathcal{A}}^{\text{worst}}(n) := \max_{x \text{ Eingabe}, |x|=n} t_{\mathcal{A}}(x)$$

Die **Average-case-Laufzeit** von $\mathcal{A}$ ist:

$$T_{\mathcal{A}}^{\text{av}}(n) := \frac{1}{|\{x : x \text{ ist Eingabe mit } |x| = n\}|} \sum_{x:|x|=n} t_{\mathcal{A}}(x)$$

Vergleichsbasiertes Sortieren

- **Gegeben:** Ein unsortierter Array $A = A[1, \ldots, n]$ von Objekten
- **Gesucht:** Ein Algorithmus der A sortiert und dabei die Schlüssel in A nur durch Vergleiche $\leq$ auswertet
- **Lösungen:** Zum Beispiel
  - Bubble Sort (Laufzeit $O(n^2)$)
  - Insertion Sort (Laufzeit $O(n^2)$)
  - Merge Sort (Laufzeit $O(n \log n)$)

Frage: Ist die Laufzeit $O(n \log n)$ schon optimal ?

Antwort: Ja, denn Sortieren liegt in $\Omega(n \log n)$.

Anmerkung: Es gibt auch Sortieralgorithmen, die nicht in dieses Schema passen, etwa Bucket-Sort (entspricht Counting Sort aus Informatik II). Diese können mitunter mit stärkeren Annahmen bessere Laufzeiten erzielen.

Man kann einen deterministischen, vergleichsbasierten Sortieralgorithmus auf folgende Art als binären Entscheidungsbaum auffassen:

- Der Algorithmus entscheidet deterministisch (nach der Eingabelänge), welcher Vergleich zuerst durchgeführt wird (ergibt Wurzelknoten)
- jedes, der beiden möglichen Ergebnisse ergibt eine ausgehende Kante
- abhängig von dem Ergebnis wird wieder deterministisch der nächste Vergleich bestimmt (ergibt Knoten der dazu passenden ausgehenden Kante)
- usw
- fertige Sortierungen werden als Blätter an den Baum angehängt

Beweisskizze für die untere Schranke

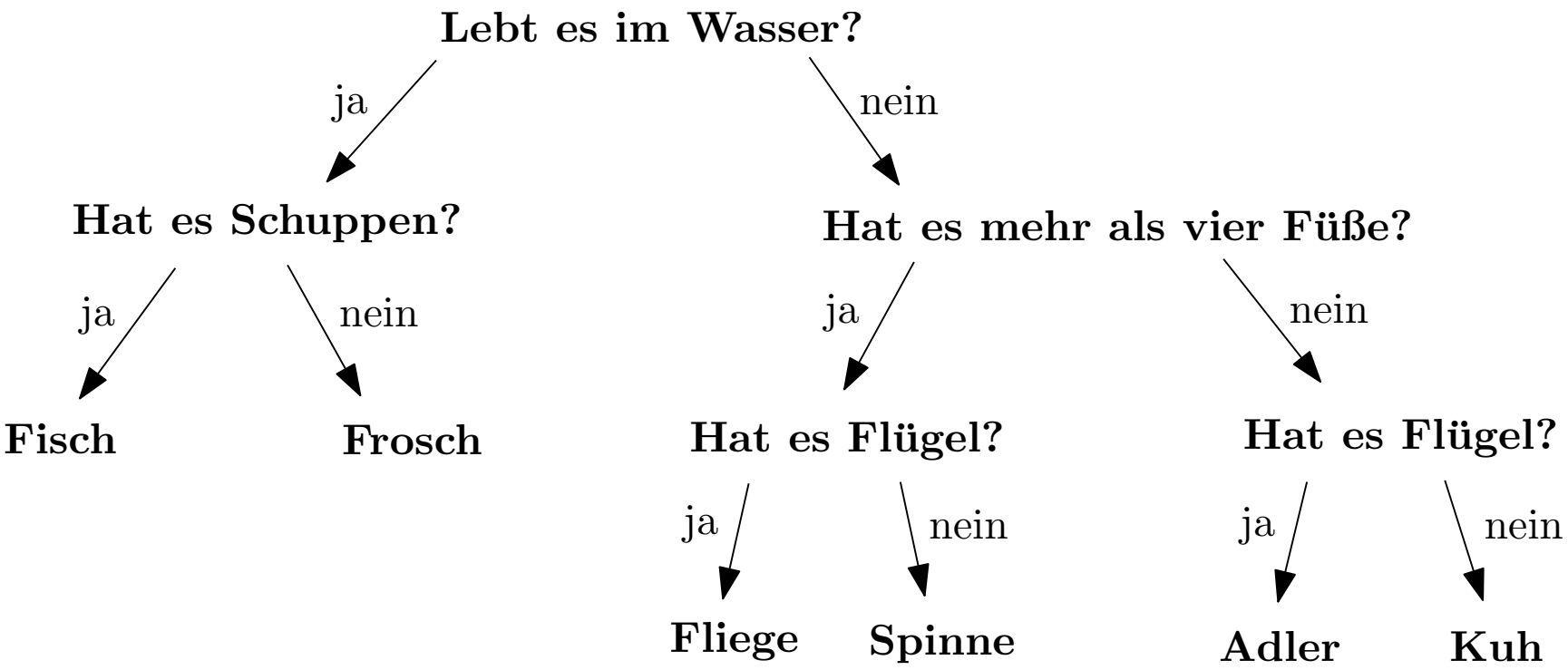
- Die *Tiefe* eines Baumes ist die Anzahl der Kanten von der Wurzel bis zum am weitesten entfernten Blatt.
- Damit entspricht die Tiefe des Entscheidungsbaums der worst-case Vergleichsanzahl

- Es gibt $n!$ mögliche Sortierungen, also mindestens $n!$ Blätter im Entscheidungsbaum.
- Ein Binärbaum mit $n!$ Blättern besitzt mindestens Tiefe $\log n!$
- Es ergibt sich folgende Abschätzung:

$$T_{sort}^{worst}(n) \geq \log_2(n!) \geq \log_2\left[\left(\frac{n}{2}\right)^{\frac{n}{2}}\right] = \frac{n}{2}(log_2 n - 1) \in \Omega(n \log n)$$

Verallgemeinerung

Anstatt einfachen Vergleichen werden beliebige Orakel-Fragen zugelassen



Verallgemeinerung

- Entscheidungsbaum mit Orakel-Fragen als Model für einen Algorithmus
- Länge des Weges im Baum entspricht Laufzeit des Algorithmus
- Blatt entspricht Ausgabe des Algorithmus
- Sogenanntes *nicht-uniformes Model*: Für verschiedene Eingabegrößen verschiedene Bäume

Amortisierte Analyse

- Gewöhnliche Laufzeitanalyse: Jede Operation wird einzeln betrachtet
- Amortisierte Laufzeitanalyse: Die Kosten eines Programmabschnittes werden über alle auszuführenden Operationen gemittelt.
- Vorteil: Machmal bessere Schranken möglich
- Wichtig: Keine Verteilungsannahme der Eingabedaten, keine average-case Analyse

Beispiel Binärzähler

- **Gegeben:** k-bit Binärzähler als Array $A[0 \ldots k-1]$ von bits
- **Aufgabe:** Zähle den Binärzähler schrittweise hoch, beginne bei 0
- **Operationen:** bit-flips: $0 \longrightarrow 1$ und $1 \longrightarrow 0$

A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	0	0	1	1
0	0	1	0	0
0	0	1	0	1
0	0	1	1	0
0	0	1	1	1
0	1	0	0	0
0	1	0	0	1
0	1	0	1	0
0	1	0	1	1
0	1	1	0	0
0	1	1	0	1
0	1	1	1	0
0	1	1	1	1

- Für einen Hochzählvorgang sind im worst-case k Operationen notwendig: $01111 \rightarrow 10000$
- Bei n Hochzählvorgängen gibt das eine Laufzeitabschätzung von $O(kn)$
- Aber: Meistens werden viel weniger bits umgeklappt
- Mit amortisierter Analyse kann man zeigen, dass die Laufzeit in $O(n)$ liegt

A[4]	A[3]	A[2]	A[1]	A[0]
0	0	0	0	0
0	0	0	0	**1**
0	0	0	**1**	**0**
0	0	0	1	**1**
0	0	**1**	**0**	**0**
0	0	1	0	**1**
0	0	1	**1**	**0**
0	0	1	1	**1**
0	**1**	**0**	**0**	**0**
0	1	0	0	**1**
0	1	0	**1**	**0**
0	1	0	1	**1**
0	1	**1**	**0**	**0**
0	1	1	0	**1**
0	1	1	**1**	**0**
0	1	1	1	**1**

Ganzheitsmethode / Aggregat-Analyse

- Bestimme eine obere Schranke $T(n)$ für die Gesamtkosten von n Operationen.
- Die amortisierten Kosten für eine Operation sind dann $\frac{T(n)}{n}$

A[0] klappt bei	jedem	Aufruf um	also insgesamt	n-mal
A[1] klappt bei	jedem 2.	Aufruf um	also insgesamt	$\lfloor n/2 \rfloor$-mal
A[2] klappt bei	jedem 4.	Aufruf um	also insgesamt	$\lfloor n/4 \rfloor$-mal
...	...	...	...	...
A[i-1] klappt bei	jedem i.	Aufruf um	also insgesamt	$\lfloor n/2^i \rfloor$-mal

Die Gesamtzahl umgeklappter bits ist also

$$\sum_{i=0}^{n-1} \lfloor \frac{n}{2^i} \rfloor \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = O(n)$$

Damit liegt die Laufzeit in $O(n)$.

Buchungsmethode (engl. accounting)

- Wähle amortisierten Kosten für die Operationen.
- Verschiedene Operationen können verschiedene Kosten haben.
- Es werden frühe Operationen in der betrachteten Folge höher veranschlagt.
- Die zu hoch veranschlagten Kosten werden jeweils als Kredit für festgelegte Objekte gespeichert und später verbraucht.

- **Beobachtung:** Damit ein bit auf 0 umklappen kann, muss es erst auf 1 umgeklappt sein.
- Schiebe die Kosten für das Umklappen auf Umklappen nach 1

Operation	tatsächliche Kosten	zugewiesene Kosten
Bit auf 1 setzen	1	0
Bit auf 0 setzen	1	2

- Beobachtung: Jeder Hochzählvorgang setzt höchstens ein bit auf 1
- Gesamtkosten für n Hochzählvorgänge also $2n$
- Damit liegt die Laufzeit in $O(n)$.

Potentialmethode

- Sei D_i die zugrundeliegende Datenstruktur im i-ten Schritt
- Seien c_i die tatsächliche Kosten von Schritt i
- Ordne den D_i's ein Potential $\Phi(D_i)$ zu
- Ordne dem i-ten Schritt amortisierte Kosten zu:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- Es gilt

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- Also ist $\sum_{i=1}^{n} \hat{c}_i$ eine obere Schranke für die tatsächlichen Kosten **falls** $\Phi(D_n) \geq \Phi(D_0)$.

- Sei $\Phi(D_i) := b_i$ die Anzahl der 1en nach dem i-ten Hochzählvorgang
- Sei t_i die Anzahl der bits die im i-ten Hochzählvorgang auf 0 gesetzt werden
- Dann gilt

$$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$$

und damit

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$

- Die Laufzeit ist also in $O(n)$.

Divide-and-Conquer Verfahren - Das Prinzip der Rekursion

- Divide-and-Conquer Verfahren zerlegen Probleme in kleinere Teilprobleme (häufig desselben Typs). Aus den Lösungen dieser Hilfsprobleme wird eine Lösung für das Ausgangsproblem konstruiert.
- Häufiger Spezialfall: Rekursive Algorithmen (z. B. Mergesort aus Informatik II)

Frage: Wie kann man worst-case Laufzeiten von solchen Algorithmen ermitteln?

Die folgenden Methoden stehen zur Verfügung:

1. **Substitutionsmethode:** Wir vermuten eine Lösung und beweisen deren Korrektheit induktiv.
2. **Iterationsmethode:** Die Rekursionsabschätzung wird in eine Summe umgewandelt und dann mittels Techniken zur Abschätzung von Summen aufgelöst.
3. **Meistermethode:** Man beweist einen allgemeinen Satz zur Abschätzung von rekursiven Ausdrücken der Form

$$T(n) = a \cdot T(n/b) + f(n), \quad \text{wobei} \quad a \geq 1 \quad \text{und} \quad b > 1.$$

Theorem (Master-Theorem)

Seien $a \geq 1$ und $b > 1$ Konstanten, $f(n)$ eine Funktion in n und $T(n)$ über nichtnegative ganze Zahlen definiert durch

$$T(n) = a \cdot T(n/b) + f(n),$$

wobei n/b für $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ steht.

(i) $T(n) \in \Theta(f(n))$ *falls* $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ *und* $af(\frac{n}{b}) \leq cf(n)$ *für eine Konstante* $c < 1$ *und* $n \geq n_0$,

(ii) $T(n) \in \Theta(n^{\log_b a} \log n)$ *falls* $f(n) \in \Theta(n^{\log_b a})$,

(iii) $T(n) \in \Theta(n^{\log_b a})$ *falls* $f(n) \in O(n^{\log_b a - \varepsilon})$.

Beispiel: Matrix-Multiplikation

Problem Matrix-Multiplikation

- **Gegeben:** $(n \times n)$-Matrizen A, B
- **Gesucht:** $A \cdot B$

Herkömmlicher Ansatz mit Laufzeit $O(n^3)$

- $C = A \cdot B = (a_{ij}) \cdot (b_{ij}) = (c_{ij})$ mit $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$

Ein Divide-and-Conquer Ansatz für die Matrixmultiplikation ist:

$$A \cdot B = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} c_1 & c_2 \\ c_3 & c_4 \end{pmatrix} = C,$$

wobei die a_i, b_i, c_i $(n/2 \times n/2)$-Matrizen sind. Dann berechnet sich C durch

$$\left.\begin{aligned} c_1 &= a_1 b_1 + a_2 b_3 \\ c_2 &= a_1 b_2 + a_2 b_4 \\ c_3 &= a_3 b_1 + a_4 b_3 \\ c_4 &= a_3 b_2 + a_4 b_4 \end{aligned}\right\}$$

8 Matrixmultiplikationen
von $(n/2 \times n/2)$-Matrizen und
4 Additionen von $(n/2 \times n/2)$-Matrizen

Aufwand:

- 8 Matrixmultiplikationen von $(n/2 \times n/2)$-Matrizen
- 4 Additionen von $(n/2 \times n/2)$-Matrizen.

Die Addition zweier $(n \times n)$-Matrizen ist in $\Theta(n^2)$. Damit ergibt sich

$$\begin{aligned} T(n) &= 8 \cdot T(n/2) + c \cdot n^2, \quad c > 0 \text{ Konstante}, \\ c \cdot n^2 &\in O(n^{\log_2 8 - \varepsilon}) = O(n^{3-\varepsilon}) \text{ mit } 0 < \varepsilon = 1 \\ \Longrightarrow \quad T(n) &\in \Theta(n^{\log_2 8}) = \Theta(n^3) \quad \text{(leider!)} \end{aligned}$$

Es gibt einen Divide-and-Conquer Ansatz von Strassen, der

- $\gg$ 7 Multiplikationen
- $\gg$ 18 Additionen

verwendet. Als Laufzeit ergibt sich dann

$$\begin{aligned} & T(n) &=& \ 7 \cdot T(n/2) + c \cdot n^2 \\ \Longrightarrow\ & T(n) &\in& \ \Theta(n^{\log_2 7}). \end{aligned}$$

Wiederholung Priority Queue

Priority Queue: Datenstruktur H um eine geordnete Menge M zu verwalten, die folgende Operationen unterstützt:

- GetMax: gibt den maximalen Wert an, der in H abgelegt ist
- DeleteMax(H): entfernt das Element an der Stelle 1 in H
- Insert(H, x): fügt einen neuen Wert x in H ein

Manchmal werden auch folgende Operationen unterstützt

- IncreaseKey
- DecreaseKey
- Delete(H, i)
- Merge

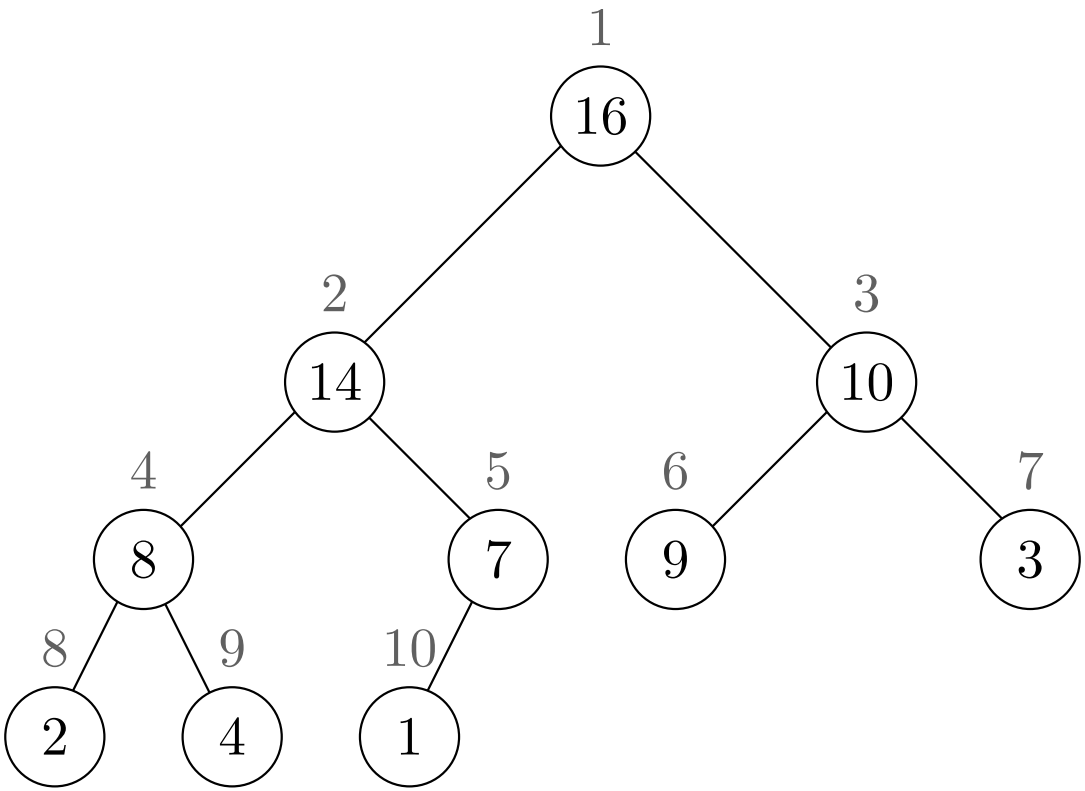
Häufige Variante: Suche das Element mit minimalem Wert anstatt dem mit maximalem Wert.

Wiederholung Heaps

Ein **(Binär)-Heap** ist ein voller binärer Baum, der mit einem Array A realisiert wird.

Die Indizierung von A wird entsprechend der Indizierung der Knoten von der Wurzel nach unten und im gleichen Level von links nach rechts angelegt. Der Heap erfüllt zusätzlich die HEAP-Eigenschaft, d.h.:

$$\forall i: \quad A[\text{Vorgänger}[i]] \geq A[i]$$



$$A = \begin{cases} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 16 & 14 & 10 & 8 & 7 & 9 & 3 & 2 & 4 & 1 \end{cases}$$

Operationen auf Heaps

- Heapify (alternativ auch Bottom-Up-Heapify)
- Makeheap
- Delete
- Sift-Up
- Insert

Mit diesen Operationen kann eine Priority Queue realisiert werden.

Laufzeiten einer Priority-Queue bei Realisierung durch einen Binär-Heap

Operation	Binär-Heap
Findmax	$O(1)$
Insert	$O(\log n)$
IncreaseKey	$O(\log n)$
DeleteMax	$O(\log n)$
Delete	$O(\log n)$

Fibonacci-Heaps (Skizze)

- Datenstruktur zur Realisierung der Operationen Insert, GetMax, DeleteMax, Merge, IncreaseKey, Delete
- Beispiel für eine Vorgehensweise die mit dem Gedanken an eine amortisierte Analyse entworfen wurde (wird nicht in der Übung vorgestellt)
- Theoretische (amortisierte) Laufzeiten sind besser als bei Binär-Heaps
- Wegen großen Konstanten allerdings praktisch weniger relevant

Vergleich der amortisierten Laufzeiten einer Priority-Queue bei Realisierung durch Binär-Heap und Fibonacci-Heap

Operation	Binär-Heap	Fibonacci-Heap
Findmax	$O(1)$	$O(1)$
Insert	$O(\log n)$	$O(1)$
IncreaseKey	$O(\log n)$	$O(1)$
DeleteMax	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$

Ungeordnete Binomialbäume

Rekursiv definiert:

- Der ungeordnete Binomialbaum U_0 besteht aus einem Knoten
- Ein ungeordneter Binomialbaum U_k besteht aus zwei ungeordneten Binomialbäumen
  - T_1 mit Wurzel r_1
  - T_2 mit Wurzel r_2

  und der Kante (r_1, r_2)

Datenstruktur des Fibonacci-Heaps

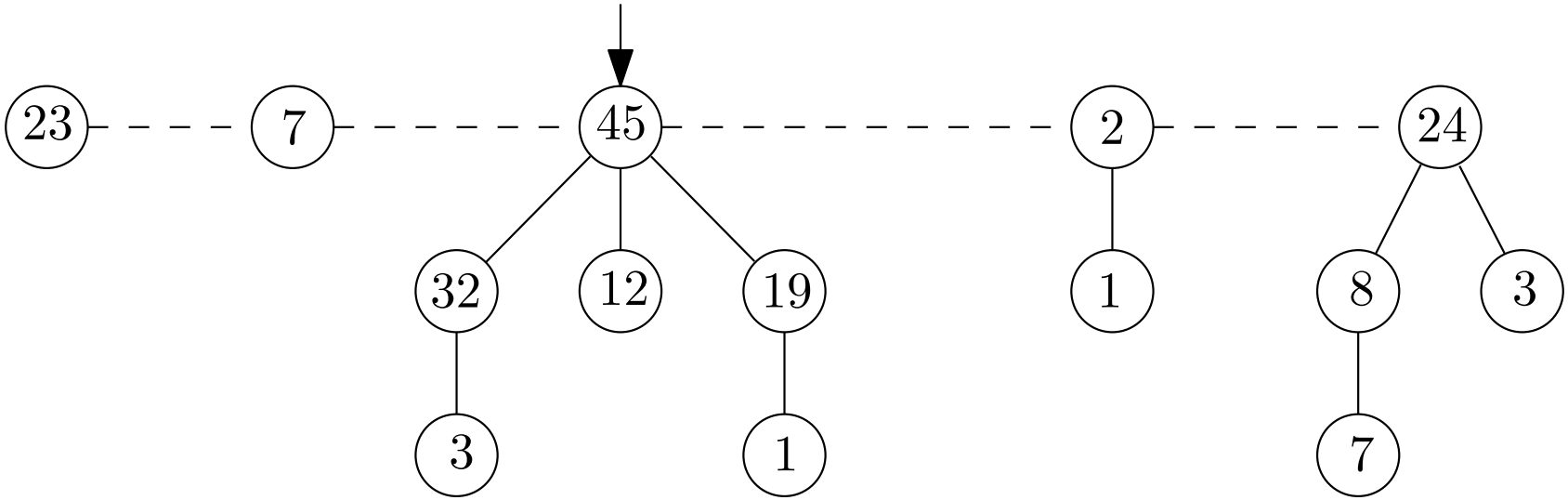
Ein Fibonacci-Heap ist eine Sammlung von ungeordneten Binomialbäumen, die alle die Heap-Eigenschaft erfüllen und deren Wurzeln durch eine doppelt-verkettete zirkuläre Liste verbunden sind.

Zusätzlich wird ein Zeiger auf die Wurzel mit dem größten Schlüssel unterhalten.

Für jeden Knoten x gibt es label

- $degree[x]$ (Anzahl der Kinder im Baum)
- $mark[x]$ (gibt an, ob x ein Kind verloren hat, seit x das Kind eines anderen Knotens wurde (zu Beginn false))

Fibonacci-Heap - Beispiel



Einfügen eines Knotens x in einen Fibonacci-Heap

- Der Knoten x wird als Baum der Größe 1 angesehen und direkt in die Wurzelliste eingefügt
- $\mathit{degree}[x] := 0$
- der Zeiger auf das maximale Element wird bei Bedarf auf x gesetzt

Suchen des maximalen Elements

- Direkt durch den Zeiger auf das maximale Element

Löschen des maximalen Elements m

- lösche m aus der Wurzelliste und füge dafür alle direkten Kinder von m in die Wurzelliste ein
- **Konsolidierung:** tue folgendes solange es in der Wurzelliste Knoten mit der gleichen Anzahl an Kindern gibt:
  - Finde zwei Wurzeln x und y mit gleicher Anzahl Kindern. Es sei $key[x] \geq key[y]$
  - Lösche y as der Wurzelliste und mache es zu einem Kind von x

Wie funktioniert der Konsolidierungsschritt genauer?

- Ein Hilfsarray $A[0, \ldots M]$ wird angelegt (man kann ein hinreichend großes M berechnen) und alle Felder mit *nil* vorinitialisiert.
- Es werden iterativ alle Knoten x der Wurzelliste durchgegangen.
  - Falls $A[\mathit{degree}[x]] = \mathit{nil}$ wird ein Zeiger auf n in $A[\mathit{degree}[x]]$ eingetragen.
  - Falls $A[\mathit{degree}[x]]$ einen Zeiger auf einen Knoten y enthält, ist ein Paar gefunden. $A[\mathit{degree}[x]]$ wird auf *nil* gesetzt. O.B.d.A. habe x den größeren Wert als y. Der Knoten y wird nun Kind von x und ein Zeiger auf x in $A[\mathit{degree}[x]]$ eingetragen.