Algorithmen und Datenstrukturen

Johannes Waldmann, HTWK Leipzig

Vorlesung, SS 17

Einleitung

Beispiel: Sortieren (Spezifikation 1)

- Eingabe: Zahl $n \in \mathbb{N}$, Zahlenfolge $a \in \mathbb{N}^n$
- Ausgabe: Zahlenfolge $b \in \mathbb{N}^n$ mit:
  - Daten-Erhaltung:
    Multimenge von $a =$ Multimenge von b
  - Monotonie:
    b ist schwach monoton steigend.

Beispiel:

- Eingabe: $n = 4, a = [5, 1, 0, 1]$.
- Multimenge von a? korrekte Ausgabe(n)?

Beispiel: Sortieren (Implementierung 1)

- nur für den Fall $n = 4$
- benutzt Operation $\mathsf{cex}_b(i, j)$ (compare-exchange):
  wenn $b_i > b_j$, dann vertausche b_i mit b_j
- Algorithmus (Ansatz)
  $b := a;\ \ \mathsf{cex}_b(1,2);\ \ \mathsf{cex}_b(3,4);\ \ \mathsf{cex}_b(1,3);\ \ \mathsf{cex}_b(2,4);$
- Beweise: jeder Algorithmus, der nur cex-Operationen verwendet, ist daten-erhaltend.
- erfüllt der o.g. Algorithmus die Monotonie-Eigenschaft? Nein. Warum nicht?
- Füge eine cex-Operation hinzu, so daß resultierender Algorithmus korrekt ist.

Beispiel: Sortieren (Implementierung 2)

- Algorithmus: Sortieren durch Auswählen (selection sort)

```
b := a;
für  nat i  von  1  bis  n
  nat k := i;
  für  nat j  von  i + 1  bis  n
    wenn b[j] < b[k] dann k := j
  // b[k] ist minimal in b[i..n]
  vertausche b[i] mit b[k];
```

- Beispiel-Rechnung für $a = [5, 1, 0, 1]$
- allgemein:
  - Korrektheit (Daten-Erhaltung, Monotonie)?
  - Laufzeit?

Beispiel Sortieren: Einordnung

- das Sortieren ist eine alte und wichtige algorithmische Aufgabe (Lochkartensortiermaschinen, Hollerith, 1908)
- hervorragend geeignet zum Lernen v. Methoden f. Algorithmen-Entwurf, Korrektheitsbeweis, Laufzeit-Analyse
- der gesamte Band 3 von DEK: TAOCP behandelt Sortieren
- . . . wer aber tatsächlich eine Liste sortiert, macht wahrscheinlich etwas *falsch*:

  . . . und hätte besser eine andere Implementierung des abstrakten Datentyps *Menge* benutzen sollen.

Grundbegriffe: Spezifikation, Algorithmus

Eine *Spezifikation* ist eine Relation $S \subseteq E \times A$.

Ein *Algorithmus* ist

- eine Vorschrift zur schrittweisen Rechnung
- auf einer tatsächlichen (konkreten) oder gedachten (virtuellen, abstrakten) Maschine,
- die aus einer Eingabe $e \in E$ eine Ausgabe $R(e) \in A$ erzeugt—oder keine (Notation: $R(e) = \perp$ mit $\perp \notin A$)

Der Algorithmus heißt

- *partiell korrekt* bezüglich S, falls $\forall e \in E : R(e) = \perp \vee (e, R(e)) \in S$
- *total (terminierend)*, falls $\forall e \in E : R(e) \neq \perp$
- *total korrekt* bezüglich S, falls $\forall e \in E : (e, R(e)) \in S$

Der abstrakte Datentyp Menge (Signatur)

- diese Datentypen werden benutzt:
  - Element-Typ E,
  - Typ M der Mengen über E
- diese (getypten) Funkionssymbole
  gehören zur Signatur des abstrakten Datentyps Menge:
  - Konstruktion:
    * $\texttt{empty} : M$
    * $\texttt{insert} : E \times M \to M$
    * $\texttt{union} : M \times M \to M$
  - Test:
    * $\texttt{null} : M \to \mathbb{B}$, dabei ist $\mathbb{B} = \{\text{False}, \text{True}\}$
    * $\texttt{contains} : E \times M \to \mathbb{B}$

Der abstrakte Datentyp Menge (Axiome)

für alle `e,f` $\in E$, `m` $\in M$:

1. `null(empty) = True;`
2. `null(insert(e,m)) = False;`
3. `contains(e,empty) = False;`
4. `contains(e,insert(e,m)) = True;`
5. `e`$\neq$`f` $\Rightarrow$
   `contains(e,insert(f,m)) = contains(e,m);`

- weitere für `delete`,`union` (Übung)
- für jede konkrete Implementierung muß Erfüllung der Axiome nachgewiesen werden.

Implementierungen von Mengen

- naiv: M realisiert durch E^*,
  $\texttt{empty} = []$, $\texttt{insert}(e, [m_1, \ldots, m_k]) = [e, m_1, \ldots, m_k]$
  $\texttt{contains}(e, [m_1, \ldots, m_k])$:
  für i von 1 bis k { wenn $m_i = e$ return True } return False
  lineare Laufzeit

- besser? verschiedene Varianten, siehe Vorlesung.
  - m monoton steigende Folge:
    `contains` logarithmisch,
    aber `insert` und `union` immer noch teuer (linear)
  - m als balancierter Suchbaum:
    `contains` und `insert` logarithmisch,
    `union` linear, in Spezialfällen schneller

Grundbegriff: Datenstruktur

- Def: eine (geeignete) Datenstruktur C
  für einen abstrakten Datentyp S
  . . . ist eine Organisationsform für Daten (ein konkreter Datentyp) mit korrekten (und effizienten) Algorithmen, die die Spezifikation von S implementieren.
- oft gib es mehrere geeignete C für ein S, die passende Wahl hängt dann von weiteren Faktoren ab (z.B. erwartetes Anzahlverhältnis einzelner S-Operationen)
- aus genauer Beschreibung der Struktur (z.B. heap-geordneter vollständig balancierter Binärbaum) können Implementierungen meist direkt abgeleitet werden

Inhalt und Ziel der Vorlesung

- Methoden für Analyse (Korrektheit, Laufzeit) und Entwurf von Algorithmen und Datenstrukturen
- für die Lösung typischer Aufgaben, insbesondere
  - Grundlagen:
    konkrete Implementierungen der Konzepte (abstrakten Datentypen), die in der Modellierung benutzt werden, u.a.:
    (Multi)Menge, Folge, Funktion, Relation, Graph
  - Anwendungen: inbesondere:
    Eigenschaften von Graphen (mit Kanten-Gewichten)
- die Methoden sind unabhängig von diesen Aufgaben nützlich

Aufgabe: Sortieren (Spezifikation 2)

- (impliziert Erfüllung von Spezifikation 1)
  - Eingabe: $n \in \mathbb{N}, a \in \mathbb{N}^n$
  - Ausgabe: Permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ mit: Folge $[a_{\pi(1)}, \ldots, a_{\pi(n)}]$ ist schwach monoton steigend.
- Beispiel: $n = 4, a = [5, 1, 0, 1]$, bestimme π.
- Satz: die Abbildung von a auf $a_\pi = [a_{\pi(1)}, \ldots, a_{\pi(n)}]$

  ist Daten-erhaltend (vorherige Def., Multimengen)
- Beweis: vergl. Anzahl der Vorkommen von z in a und a_π:

  $$\#\{i \mid a_\pi(i) = z\} = \#\{i \mid a_{\pi(i)} = z\} = \#\{\pi^{-1}(j) \mid a_j = z\}|.$$

Implementierungen von Sortierverfahren

- naiv (schon gesehen) Sortieren durch Auswahl

  quadratische Laufzeit

- besser? verschiedene Varianten, siehe Vorlesung, z.B.

```
Set<E> m:=empty;
foreach E e in a { m:=insert(e,m); }
while not(null(m)) {
   (e, m) := extract-min(m); print (e);
}
```

  Laufzeit $n \cdot \log(n)$ (bei guter Implementierung für Mengen)

Aufgabe: kürzeste Wege

- Aufgabenstellung:
  - Eingabe:
    - gerichteter Graph $G = (V, E)$
    - Kanten-Gewichte $w : E \to \mathbb{Q}_{\geq 0}$
    - Knoten $v_0 \in V$
  - Ausgabe: Funktion $l : V \to \mathbb{Q}_{\geq 0} \cup \{+\infty\}$ mit:
    - $l(v) = \min\{\sum_{e \in P} w(e) \mid P$ ist Weg von v_0 zu $v\}$.
- Beispiel (siehe Tafel)
- Algorithmus? siehe Vorlesung.

Aufgabe: Rundreise

- Aufgabestellung:
  - Eingabe:
    - ungerichter Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$
    - Kanten-Gewichte $w : E \to \mathbb{Q}_{\geq 0}$
  - Ausgabe:
    - eine Permutation π von V mit:
    - $\forall i : \pi(i)\pi(i+1) \in E$
    - und $\sum\{w(\pi(i)\pi(i+1))|i \in V\}$ ist minimal (unter allen zulässigen π)
    - dabei Index-Rechnung modulo n, d.h. $\pi(n+1) = \pi(1)$
- Beispiel (siehe Tafel).
- Algorithmen? siehe Vorlesung

Einordnung dieser VL in das Studium

Auswahl von LV zu Programmierung und Algorithmen:

- 1. Sem: Modellierung
- 1./2. Sem: (AO) Programmierung
  (strukturiert, imperativ, objekt-orientiert)
- 2. Sem: *Algorithmen und Datenstrukturen*
- 3. Sem: Softwaretechnik
- 4. Sem: Fortgeschrittene (funktionale) Progr.
- Wahl 5. Sem : Sprachkonzepte parallele Progr.
- 1. Sem Master: Prinzipien von Programmiersprachen
- Master: Theor. Inf. (Komplexität), Verifikation
- Wahl Master: Algorithm Design

Organisatorisches

- jede Woche 2 VL
  Folien und weitere Informationen: `http://www.imn.htwk-leipzig.de/~waldmann/lehre.html`
  (Sprechzeit: mittwoch 11:15, per Mail anmelden)
- jede Woche 1 Übung (pro Seminargruppe),
  dort Vorrechnen Ihrer Lösungen der Übungsaufgaben
  sollen in Kleingruppen (3 bis 4 Leute) bearbeitet werden
- online-Übungsaufgaben im autotool
- Prüfungszulassung: erfolgreich Vorrechnen (60 %)
  und erfolgreich autotool (60 % der Pflichtaufgaben)
- Prüfung: Klausur, *keine* Hilfsmittel

Zeitplan für Übungsserien

Grundsätzlich:

- jeweils am Anfang der Woche x werden die Aufgaben der Übungsserie $x+1$ bekanntgegeben, die in Übungen der Woche $x+1$ vorgerechnet und bewertet werden.

zur Initialisierung: in KW 14 ($=$ aktuelle Woche $=$ Woche 0)

- Ü-Serie 0 (am Ende dieses Skriptes)
  zur Diskussion in Woche 0 (ab morgen),
  ohne Punkte, aber autotool-Pflichtaufgabe zählt
- in Übungen Woche 0: Einteilung in Gruppen
- Ü-Serie 1 wahrscheinlich morgen (Dienstag):
  ähnlich zu Serie 0 (kein neuer Stoff), aber *mit* Punkten.

Bewertung von Übungsserien

- autotool: bewertet individuell (ca. 1 Pflicht-A/Woche)
- Übungen: für jede der aktuellen Aufgaben A (ca. 4 Stück)
  - Übungsleiter ruft einen Teilnehmer T auf, dieser rechnet vor, die Leistung wird bewertet, alle anwesenden Mitglieder der Gruppe G von T erhalten diesen Wert.
  - T darf ersatzweise eine andere aktuelle Aufgabe A' auswählen
  - bei Abwesenheit von T darf G einen anderen Vertreter $T' \in G$ benennen.
  - es wird Gleichverteilung über alle T angestrebt
- Übungsleiter kann von dieser Regelung im Einzelfall abweichen.

Literatur

- K. Weicker und N. Weicker: *Algorithmen und Datenstrukturen*, Springer 2013, `https://link.springer.com/book/10.1007/978-3-8348-2074-7`
- T. Ottman und P. Widmayer: *Algorithmen und Datenstrukturen*, Springer 2012, `https://link.springer.com/book/10.1007/978-3-8274-2804-2`
- T. H. Cormen, C. E. Leiserson, R. L. Rivest: *Algorithms*, MIT Press 1991, `https://mitpress.mit.edu/books/introduction-algorithms`
- D. E. Knuth: *The Art of Computer Programming*, Vol. 3 (Sorting and Searching), Addison-Wesley, 1998, `http://www-cs-faculty.stanford.edu/~uno/taocp.html`

Alternative Quellen

- – Q: Aber in Wikipedia/Stackoverflow steht, daß . . .
  – A: Na und.

- Es mag eine in Einzelfällen nützliche Übung sein, sich mit dem Halbwissen von Nichtfachleuten auseinanderzusetzen.

  Beachte aber `https://xkcd.com/386/`

- In VL und Übung verwenden und diskutieren wir die durch Dozenten/Skript/Modulbeschreibung vorgegebenen Quellen (Lehrbücher, referierte Original-Artikel, Standards zu Sprachen und Bibliotheken)

- . . . gilt entsprechend für Ihre Bachelor- und Master-Arbeit.

Übung KW14

Ablauf der Übung

- Diskussion der unten angegebenen Aufgaben 0.0 bis 0.3
- kurze Erläuterung des kooperativen Übungskonzeptes und Einteilung (durch Übungsleiter) in Gruppen (je 3 bis 4 Personen)
- Bearbeitung der Aufgabe 0.4 (in den neu gebildeten Gruppen) und Diskussion

Übungsserie 0 (Diskussion in KW14)

Aufgabe 0.0 (Leseübung)

- Wie begründen Ottman/Widmayer, daß es in der Vorlesung AD nicht so sehr auf die exakte Formulierung des Algorithmenbegriffs ankommt? In welcher anderen Vorlesung wird das wichtiger und warum? (Quelle mit Seiten- und Zeilennummer angeben.)

Aufgabe 0.1 (Mengen)

- Beweisen Sie mittels der angegebenen Axiome:

  `contains(2,insert(1,insert(3,empty)))=False`

- Geben Sie ein passendes Axiom für die Vereinigung `union` : $M \times M \to M$ an. Verwenden Sie eine passende aussagenlogische Verknüpfung.

- Der Wert von `delete`(e, m) soll die Menge $m \setminus \{e\}$ sein. Geben Sie den Typ von `delete` an.

  Untersuche Sie (Beweis oder Gegenbeispiel):

  - $\forall e \in E, m \in M$: `delete`$(e,$ `insert`$(e, m)) = m$
  - $\forall e \in E, m \in M$: `insert`$(e,$ `delete`$(e, m)) = m$

Aufgabe 0.2 (Sortieren)

Für $a = [3, 1, 4, 1, 5, 9]$:

- geben Sie alle Permuationen π an, die die Spezifikation (Variante 2) des Sortierens erfüllen.

- Führen Sie den Algorithmus *Sortieren durch Auswahl* mit Eingabe a aus.

Aufgabe 0.3 (kürzestes Wege)

Wir betrachten den Graphen G_n auf der Knotenmenge $V_n = \{0, 1, \ldots, n\}^2$ mit den Kanten

- $((x, y), (x + 1, y))$ vom Gewicht 4
- $((x, y), (x, y + 1))$ vom Gewicht 1
- $((x, y), (x + 1, y + 1))$ vom Gewicht 2

(jeweils für alle x, y, für die das angegebene Paar tatsächlich in V_n^2 liegt)

- Zeichnen Sie G_2.
- Geben Sie in G_2 alle Wege von $(0, 0)$ zu $(1, 2)$ an. Geben Sie zu jedem dieser Wege die Kosten an. Bestimmen Sie

$l(1,2)$, wobei l die Funktion aus der Spezifikation ist, und $v_0 = (0,0)$.

- Geben Sie weitere konkrete Funktionswerte von l an.
- Beweisen Sie für $i < j$: Die Funktion l für G_i ist (als Menge von geordneten Paaren) Teilmenge der entsprechenden Funktion für G_j. (Zeichnen Sie G_3)
- Begründen Sie $l(k,0) = 4k$ (in G_k und allen größeren)
- Geben Sie weitere allgemeine Aussagen über l an.

Aufgabe 0.4 (Sortieren mit compare-exchange)

- Begründen Sie, daß folgendes Verfahren jede Eingabe der Breite 4 sortiert. Jedes Paar (i, j) bedeutet $\texttt{cex}(i, j)$.

```
[(1,2), (3,4), (2,3), (1,2), (3,4), (2,3)]
```

- Begründen Sie (jeweils durch ein Gegenbeispiel), daß man keines dieser Paare weglassen kann.

Aufgabe 0.5(A) (autotool)

- Geben Sie ein compare-exchange-Sortierverfahren für 6 Elemente an mit weniger als 15 cex-Operationen.

Übungsserie 1 (Diskussion in KW15)

Aufgabe 1.0 (Leseübung)

- (1 P) Wo und unter welchem Namen erscheint der Algorithmus *Sortieren durch Auswählen* in Weicker/Weicker? (Kapitel, Seitennummer)

- (2 P) Welche zwei Unterschiede gibt es zur Version hier im Skript?

Aufgabe 1.1 (Mengen)

- (1 P) Beweisen Sie mittels der angegebenen Axiome:

  `contains(2,insert(1,insert(2,empty)))=True`

- (1 P) Geben Sie ein passendes Axiom für die Mengendifferenz `difference` : $M \times M \to M$ an.

- (2 P) Wie läßt sich ein Test `equals` : $M \times M \to \mathbb{B}$ auf Mengengleichheit implementieren, wenn nur die bis hier in Skript und Aufgaben angegebenen Operationen des abstrakten Datentyps Menge sowie alle aussagenlogischen Verknüpfungen zur Verfügung stehen?

Aufgabe 1.2 (Sortieren mit compare-exchange)

- (3 P) Begründen Sie, daß folgendes Verfahren jede Eingabe der Breite 4 sortiert. Jedes Paar (i, j) bedeutet $\texttt{cex}(i, j)$.

```
[(2,3), (1,2), (3,4), (2,3), (1,2), (3,4)]
```

- (1 P) Begründen Sie durch ein Gegenbeispiel, daß man die erste Operation $\texttt{cex}(2, 3)$ nicht weglassen kann.

Aufgabe 1.3 (kürzeste Wege)

Wir betrachten den Graphen G_n auf der Knotenmenge $V_n = \{1, \ldots, n\}^2$ und der Kantenmenge $E_n = \{((x_1, x_2), (y_1, y_2)) \mid (x_1 - y_1)^2 + (x_2 - y_2)^2 = 5\}$, die alle Gewicht 1 haben.

- (1 P) Dieser Graph hat eine Beziehung zu einem Brettspiel—welche?
- (1 P) Bestimmen Sie die Abstandsfunktion l_{G_4} von $v_0 = (1, 1)$ aus.
- (1 P) Erklären Sie das Muster der Vorkommen der geraden und ungeraden Funktionswerte von l.
- (1 P) immer noch für $v_0 = (1, 1)$: Geben Sie ein v an, für

das $l_{G_4}(v) > l_{G_5}(v)$ ist.

Komplexität von Algorithmen und Problemem

Plan

- Definition: worst-case-Zeitkomplexität eines Algorithmus
- Definition: Komplexität eines Problems, Komplexitätsklassen
- Bestimmung der Komplexität von Algorithmen mit Schleifen
- asymptotischer Vergleich von Funktionen
- (später) weitere Ressourcen (Platz, parallele Zeit)
- (später) average-case-Komplexität

Literatur: vgl. WW 2.3/2.4, OW S. 4/5

Was bedeutet *Zeit*?

- auf einer *konkreten* Maschine:

  die konkrete (*physikalische*) Zeit

  einer Ausführung des Programms für *eine* Eingabe

- auf einer (konkreten oder) *abstrakten* Maschine:

  die *Anzahl* der Rechenschritte für *eine* Eingabe

  (beachte: Def. Algorithmus: *schrittweise* Ausführung)

- Beispiel: Profiling/Coverage-Analyse (für C-Programme)
  - `gprof` zählt Unterprogramm-Aufrufe
  - `gcov` zählt Ausführungen von Anweisungen (Zeilen)

Coverage mit gcov

https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

- kompilieren mit *Instrumentierung*

  ```
  g++ -fprofile-arcs -ftest-coverage ex.cc -o ex
  ```

- ausführen: `./ex`, erzeugt `ex.gcda`
- Report: `gcov ex.cc`, erzeugt: `less ex.cc.gcov`
  in der ersten Spalte steht Anzahl der Ausführung:

```
        5:    8:  for (int i = 0; i<n; i++) {
        4:    9:    int k = i;
       10:   10:    for (int j = i+1; j<n; j++)
        6:   11:      if (b[j]<b[k]) k = j;
        4:   12:    swap(b[i],b[k]);
```

Vollständiger Quelltext: `https://gitlab.imn.htwk-leipzig.de/waldmann/ad-ss17`

Zeitkomplexität von Algorithmen

- Definition: die Zeitkomplexität des Algorithmus A

  ist die Funktion

  $c_P : \mathbb{N} \to \mathbb{N} : s \mapsto \max\{\text{time}(A, x) \mid \text{size}(x) \leq s\}$

  $c_P(s) =$ größte Laufzeit auf allen Eingaben mit Größe $\leq s$

- Bsp: prüfe, ob $a[1 \ldots n]$ schwach monoton steigt:

```
for nat i von 1 bis n-1
  if a[i] > a[i+1] return False
return True
```

  1 Vergleich für $[8, 1, 7, 2]$, aber $n - 1$ Vgl. für $[1, 2, \ldots, n]$

  die Zeitkomplexität ist $s \mapsto$ (if $s = 0$ then 0 else $s - 1$)

Zeitkomplexität von Problemen

- für jede Funktion $f : \mathbb{N} \to \mathbb{N}$:
  TIME$(f) :=$ die Menge der Probleme (Spezifikationen), die sich durch einen Algorithmus mit Komplexität f lösen (implementieren) lassen.
- Bsp (vorige Folie): Monotonie $\in$ TIME$(n \mapsto n)$,
  Bsp (vorige VL): Sortieren $\in$ TIME$(n \mapsto n(n-1)/2)$
- für jede Menge von Funktionen $F \subseteq \mathbb{N} \to \mathbb{N}$:
  TIME$(F) = \bigcup_{f \in F}$ TIME(f)
- Monotonie $\in$ TIME(linear), Sortieren $\in$ TIME(quadrat.)
- TIME(alle Polynome) (Abkürzung: **P**)
  gilt als die Klasse der *effizient* lösbaren Probleme.
- in der VL *Alg. und DS* sind wir fast immer in **P**

Komplexität von Programmen

- strukturiertes imperatives Programm ist Baum:
  - Blätter: elementare Anweisung
  - innere Knoten (zusammengesetzte Anweisungen): Nacheinanderausführung, Verzweigung, Wiederholung
- die Komplexität bestimmen wir durch:
  - elementar: 1 Schritt
  - zusammengesetzt:
    - Nacheinanderausführung: Summe
    - Verzweigung: 1 plus Maximum über die Zweige
    - Wiederholung (Schleife): parametrische Summe, abhängig von Schleifensteuerung (Laufbereich)
- später: Komplexität für (rekursive) Unterprogramme

Komplexität v. Progr. (Verzweigung)

- Eingabe: $[a_1, \ldots, a_n]$, Ausgabe und Spezifikation egal

```
nat s := 0;
for i from 1 to n
  if a[i] > 0 then s := s + 1
```

- Kosten für die Verzweigung: $1 + 1$ oder $1 + 0$, also wenigstens 1, höchstens 2
- Kosten für die Schleife: wenigstens $\sum_{i=1}^{n} 1 = n$, höchstens $\sum_{i=1}^{n} 2 = 2n$
- *Kosten* für gesamtes Programm: bei jeder Eingabe wenigstens $1 + n$, höchstens $1 + 2n$
- Es gibt für jedes n eine Eingabe mit Kosten $1 + 2n$: Komplexität ist $1 + 2n$

Komplexität v. Progr. (Schleifen)

- Eingabe ist immer $[a_1, \ldots, a_n]$
- `for i from 1 to n { for j from 1 to n  e; }`

  Kosten: $\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = n^2$
- `for i from 1 to n { for j from 1 to i  e; }`

  Kosten: $\sum_{i=1}^{n} \sum_{j=1}^{i} 1 = \ldots$
- ```
  for i from 1 to n
      for j from i+1 to n
          for k from j+1 to n  e
  ```

  geschlossene Darstellung?

  Verallgemeinerung (mehr Schleifen dieser Art)?
  ```

Asymptotischer Vergleich von Funktionen

Wiederholung

unterschiedliche Komplexitätsfunktionen
und ihre Auswirkungen bei Änderung der Eingabegröße:

- wenn $f(n) = \log_2 n$, dann $f(2n) = \ldots$
- wenn $f(n) = c \cdot n$, dann $f(2n) = 2 \cdot f(n)$
- wenn $f(n) = n^2$, dann $f(2n) = \ldots$
- wenn $f(n) = 2^n$, dann $f(n+1) = \ldots$

das ist nützlich, aber nicht alle Funktionen sind so einfach

Motivation: Abstraktion von...

- konkreten Kosten elementarer Operationen

  `s := s + 1` kostet 1 oder 2 oder 3? — egal!

  Funktion $n \mapsto 2n$ soll genauso gut sein wie $n \mapsto 3n$.

- Kosten durch unwesentliche Programmteile,

  z.B. Initialisierungen, die bei kleinen Eingaben evtl. die Gesamtkosten dominieren

  $n \mapsto \max(100, n)$ soll genauso gut sein wie $n \mapsto n$

- Sprungstellen, z.B. $n \mapsto 2^{\lfloor \log_2 n \rfloor}$

Definition, Beispiele

- Def: $O(g) := \{f \mid \exists s \geq 0, c \geq 0 : \forall n \geq s : f(n) \leq c \cdot (g(n))\}$

  Funktion f wächst höchstens wie Funktion g

  Bezeichnung aus der Analysis (das Landau-Symbol)

- betrachte: $f_1(n) = n$, $f_2(n) = \max(100, n)$, $f_3(n) = n^2/2$.

- $f_1 \in O(f_2)$, Beweis: wähle $s = 0, c = 1$.

- $f_2 \in O(f_1)$, Beweis: wähle ...

- $f_1 \in O(f_3)$, Beweis: wähle $s = 0, c = 2$.

- $f_3 \notin O(f_1)$, Beweis (indirekt):

  falls s und c lt. Def., dann ... Widerspruch

Eigenschaften

die Relation $\{(f, g) \mid f \in O(g)\}$:

- ist reflexiv
- ist nicht symmetrisch (Gegenbeispiel?)
- ist nicht antisymmetrisch (Gegenbeispiel?)
- ist transitiv
- ist nicht total:

  Für $f_1(n) = n$, $f_5(n) =$ if $2 \mid n$ then n^2 else 0

  gilt $f_1 \notin O(f_2)$ und $f_2 \notin O(f_1)$.

Aber wenn es nur um Polynome geht. . .

. . . wozu dann der Aufwand? Polynome erkennt man leicht!

Nein. Welche der folgenden Funktionen sind (beschränkt durch) Polynome? Von welchem Grad?

- $n \mapsto n^2$
- $n \mapsto n^2 + 10n - 100 \log n$
- $n \mapsto (\log n)^{\log n}$
- $n \mapsto \sum_{k=1}^{n} \lfloor \log k \rfloor$
- $n \mapsto \sum_{k=1}^{n} \lfloor \log(n/k) \rfloor$

Solche Funktionen kommen als Kostenfunktionen von Algorithmen vor und müssen richtig eingeordnet werden.

Abgekürzte Notation für Funktionen

- richtig: $n \mapsto n^2$ (historisch korrekt: $\lambda n.n^2$)

  die gebundene Variable n wird explizit notiert.

- häufig wird Notation der gebundenen Variable vergessen:

  n^2 soll dann eine Funktion bezeichnen.

- Anwendung: $n \in O(n^2)$

- gefährlich wird das hier: ist x^2y quadratisch oder linear?

  unklar, ob $x \mapsto x^2y$ gemeint ist oder $y \mapsto x^2y$ oder...

- lustige work-arounds zur Vermeidung der richtigen Notation: „für ein *festes* y“ u.ä.

Abgekürzte Notation f. Mengen v. Funktionen

- richtig: $f \in O(g)$, denn $O(g)$ ist eine Menge
- stattdessen häufig: $f = O(g)$ . . . auch: $n = O(n^2)$
- gefährlich, weil das Symbol „=“
  dann nicht mehr symmetrisch und transitiv ist:
  $n = O(n^2)$ und $n^2 = O(n^2)$, also folgt (optisch) $n = n^2$
- wie bei allen Abkürzungen:
  für Fachleute nützlich, aber der Anfänger wundert sich.
- Anwendung: $3n + O(n^2) = O(n^2)$
  ist Gleichung zwischen *Mengen* von Funktionen

$$\forall f \in O(n^2) : (n \mapsto 3n + f(n)) \in O(n^2)$$
$$\wedge \quad \forall g \in O(n^2) : \exists f \in O(n^2) : \forall n : 3n + f(n) = h(n)$$

Eine Vergleichsmethode

- Satz: falls $\lim_{n \to +\infty} \frac{f_1(n)}{f_2(n)} = c \in \mathbb{R}$ existiert, dann $f_1 \in O(f_2)$.
- Beweis (Skizze): Definition, Einsetzen, Umformen

  $\lim_n x_n = c \iff \forall \epsilon > 0 : \exists s : \forall n \geq s : c - \epsilon \leq x_n \leq c + \epsilon$,
- Anwendungen

  (für $f_1(n) = n, f_2(n) = \max(100, n), f_3(n) = n^2/2$)
- $\lim_{n \to +\infty} f_1(n)/f_2(n) = 1$, $\lim_{n \to +\infty} f_2(n)/f_1(n) = 1$
- $\lim_{n \to +\infty} f_1(n)/f_3(n)?$, $\lim_{n \to +\infty} f_3(n)/f_1(n)?$,
- Umkehrung diese Satzes *gilt nicht*!

  Bsp: $f_4 = n \mapsto 2^{\lfloor \log_2 n \rfloor}$, vergleiche mit f_1.

Weitere Notationen

(wie immer: alle Funktionen aus $\mathbb{N} \to \mathbb{N}$)

- Def: $O(g) := \{f \mid \exists s \geq 0, c \geq 0 : \forall n \geq s : f(n) \leq c \cdot (g(n))\}$
  $f \in O(g)$ bedeutet: f wächst höchstens so wie g
- Def: $\Omega(g) := \{f \mid g \in O(f)\}$.
  $f \in \Omega(g)$ bedeutet: f wächst wenigstens so wie g
- Def: $\Theta(g) := O(g) \cap \Omega(g)$.
  $f \in \Theta(g)$ bedeutet: f und g wachsen gleich schnell.
- Def: $o(g) := O(g) \setminus \Omega(g), \quad \omega(g) := \Omega(g) \setminus O(g)$.
- Satz: falls $\lim_{n \to +\infty} f(n)/g(n) = 0$, dann $f \in o(g)$
  Beweis (indirekt): falls $g \in O(f)$, dann für ...

Asympt. Vergleich von Standardfunktionen

- Potenzen verschiedener (auch nicht-ganzer) Grade:

  $\forall p, q \in \mathbb{R}_{>0} : p < q \Rightarrow n^p \in o(n^q)$

- Logarithmus und Potenz

  $\forall p, q \in \mathbb{R}_{>0} : \log^p(n) \in o(n^q)$

  Bsp: $\log^{10} n \in o(\sqrt[3]{n})$

- Potenz und Exponential

  $\forall p \in \mathbb{R}_{>0}, q \in \mathbb{R}_{>1} : n^p \in o(q^n)$

- Produkte von Potenz und Logarithmus

  $n^{p_1} \cdot \log^{q_1} n \in o(n^{p_2} \cdot \log^{q_2} n)$ gdw. ...

alle Beweise mittels Limes-Kriterium

Spezielle Funktionen

- Logarithmus
  - in der Informatik grundsätzlich zur Basis 2
  - die Basis ist oft ganz egal: $\log_{10} \in \Theta(\log_2)$.

- iterierter Logarithmus: $\log^*$
  - definiert durch $n \mapsto$ if $n \leq 1$ then 0 else $1 + \log^* \lfloor \log n \rfloor$
  - sehr langsam wachsend

x	0	1	2	4	16	65536	2^{65536}
$\log^* x$	0	0	1	2	3	4	5

- Fakultät, Def: $n! := 1 \cdot 2 \cdot \ldots \cdot n$
  - Satz: $\log(n!) = \Theta(n \log n)$
  - Beachte: es gilt *nicht* $n! \in \Theta(n^n)$

Anwendung

Eine klassische Übungsaufgabe aus [CLR]:

- Ordne das asymptotische Wachstum von

  (d.h., bestimme alle O-Relationen zwischen)

$(\log n)^{\log n}$	$2^{\log^* n}$	$\sqrt{2}^{\log n}$	n^2	$n!$	$(\log n)!$
$\log\log n$	$\log^* n$	$n^{\log\log n}$	$n \cdot 2^n$	$\log n$	1
$(3/2)^n$	n^3	$\log^2 n$	$\log(n!)$	2^{2^n}	$n^{1/\log n}$
$\log^*(\log n)$	$2^{\sqrt{2\log n}}$	n	$n \log n$	2^n	$2^{2^{n+1}}$
$2^{\log n}$	$\exp(n)$	$\log(\log^* n)$	$4^{\log n}$	$(n+1)!$	$\sqrt{\log n}$

- insbesondere: welche Paare sind Θ-äquivalent?

Übungsserie 2 (für KW 16)

Aufgabe 2.1

```
for x = 1 to n
  for y = 1 to n
    if not (x = y)
      for z = 1 to n
        if not (x = z) && not (y = z)
          e
```

- (1 P) Geben Sie die Folge der Belegungen von x, y, z an für $n = 2$,

- (1 P) . . . , für $n = 3$.

- (2 P) Wie oft wird Anweisung $\in$ ausgeführt? Geben Sie die Antwort als Funktion von n an.

Aufgabe 2.2

Die Binärdarstellungen aller Zahlen von 0 bis n werden nebeneinandergeschrieben.
Als Darstellung der 0 benutzen wir dabei die *leere* Ziffernfolge.
Die Gesamt-Anzahl der Vorkommen der 1 nennen wir $B(n)$.
Beispiel: von 0 bis 3: $[[], [1], [1, 0], [1, 1]]$, $B(3) = 4$.

- (1 P) Bestimmen Sie $B(100)$ (z.B. durch ein Programm)
- (1 P) Geben Sie $B(2^w - 1)$ exakt an.
- (2 P) Begründen Sie $B(n) \in \Theta(n \log n)$.

Die Binärdarstellungen von 0 bis $n - 1$ werden durch führende Nullen ergänzt, so daß jede so lang wie die

Darstellung von n wird.
Die Gesamt-Anzahl der dabei hinzugefügten 0 nennen wir $F(n)$.
Beispiel: $[[\underline{0}, \underline{0}], [\underline{0}, 1], [1, 0], [1, 1]]$, $F(3) = 3$.

- (1 P) Bestimmen Sie $F(100)$ (z.B. durch Modifikation des vorigen Programms)

- (2 P) Geben Sie $F(n)$ als Θ einer einfachen Funktion von n an.

Aufgabe 2.2 (Musterlösung einer Teilaufgabe)

- $B(100) = 319$

- experimentell kann Wertetabelle bestimmt werden:

w	0	1	2	3	4	5	6	7	8
$B(2^w - 1)$	0	1	4	12	32	80	192	448	1024

daraus Vermutung: $B(2^w - 1) = w \cdot 2^{w-1}$

Beweis: alle Binärzahlen untereinanderschreiben

$$
\begin{array}{ccc}
 & & 1 \\
 & 1 & 0 \\
 & 1 & 1 \\
1 & 0 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
1 & 1 & 1
\end{array}
$$

das sind 2^w Zeilen (die erste ist leer) mit je w Spalten und in jeder Spalte ist die Hälfte der Einträge gleich 1.

- Zu zeigen ist $B(n) \in \Theta(n \log n)$.

  Bekannt sind

- $\forall w > 0 : B(2^w - 1) = w \cdot 2^{w-1}$ (siehe oben)
- B ist streng monoton
  (wenn $x < y$, dann $B(x) < B(y)$, denn von x bis y sind $y - x$ Zahlen hinzugekommen und jede enthält wenigstens ein 1-Bit)

Beweis:

- für jedes n gibt es w mit $2^w - 1 \leq n < 2^{w+1} - 1$, nämlich $w = \lfloor \log(n+1) \rfloor$.
  Begründung: dann gilt $w \leq \log(n+1) < w+1$ nach Def. von $\lfloor \cdot \rfloor$,
  also $2^w \leq n + 1 < 2^{w+1}$, daraus Behauptung.
- wegen Monotonie von B und bekannten Werten gilt $w \cdot 2^{w-1} = B(2^w - 1) \leq B(n) < B(2^{w+1} - 1) = (w+1) \cdot 2^w$
- in der oberen Schranke schätzen wir w nach oben ab:

$B(n) < (w+1) \cdot 2^{w+1} \leq (\log(n+1)+1) \cdot 2^{\log(n+1)} =$
$(\log(n+1)+1) \cdot (n+1)$

Für $n \geq 1$ setzen wir die Ungleichungskette fort

$\cdots \leq (\log(2n)+1) \cdot 2n = (\log n+1+1) \cdot 2n = (\log n+2) \cdot 2n$

Für $n \geq 4$ weiter: $\cdots \leq (\log n + \log n) \cdot 2n = 4n \log n$

Daraus folgt $B \in O(n \log n)$, denn wir haben gezeigt

$\forall n \geq 4 : B(n) \leq 4n \log n$.

– entsprechend für untere Schranke:

$B(n) \geq w \cdot 2^{w-1} > (\log(n+1)-1) \cdot 2^{\log(n+1)-2} >$
$(\log n - 1) \cdot 2^{\log n - 2} = (\log n - 1) \cdot n/4$

Für $n \geq 4$ ist $\log n \geq 2$ und $\log n - 1 \geq \log n - (\log n/2)$,

also $B(n) \geq \cdots \geq (\log n/2) \cdot n/4 = (1/8) n \log n$

Daraus folgt $n \log n \in O(B)$, denn wir haben gezeigt

$\forall n \geq 4 : n \log n \leq 8B(n)$.

Diskussion:

aus $B(2^w - 1) = w \cdot 2^{w-1}$ mit $n = 2^w - 1$ kann man (sollte man zunächst) die Überschlagsrechnung durchführen:

$n \approx 2^w$, also $w \approx \log n$, also
$B(n) \approx \log n \cdot 2^{\log n - 1} = \log n \cdot n/2 \in \Theta(n \log n)$.

Aber: das ist kein Beweis, man muß nachrechnen daß der Fehler bei „$\approx$“ nicht wesentlich ist, und das geht am sichersten durch exakten Nachweis der beiden O-Beziehungen

Aufgabe 2.3

(4 P)

Welche O, Θ-Relationen gelten zwischen

$f_1(n) = n - \sqrt{n},\ f_2(n) = \sqrt{n}^{\sqrt{n}},\ f_3(n) = n$

Aufgabe 2.4

- (4 P) Für Funktionen $f, g : \mathbb{N} \to \mathbb{N}$:

  Beweisen Sie $\max(f, g) \in \Theta(f + g)$.

  Hierbei bezeichnen

  $\max(f, g)$ die Funktion $n \mapsto \max(f(n), g(n))$

  sowie $f + g$ die Funktion $n \mapsto f(n) + g(n)$.

Einfache Algorithmen auf Zahlen und Feldern

Übersicht

- Euklidischer Algorithmus
- Multiplizieren durch Verdoppeln
- Schleifen-Invarianten
- Schranken
- binäre Suche in geordnetem Feld
- dutch national flag

Der größte gemeinsame Teiler (gcd)

- Die Teilbarkeitsrelation auf $\mathbb{N}$:
  Def: $a \mid c : \Longleftrightarrow \ \exists b \in \mathbb{N} : a \cdot b = c$
- Bsp: $13 \mid 1001, 13 \mid 13, 13 \mid 0, \neg(0 \mid 13), 0 \mid 0$
- Eigenschaften: Halbordnung, nicht total.
- der größte gemeinsame Teiler. Def: $\gcd(x, y) = g$ gdw.
  $g \mid x \wedge g \mid y$ und $\forall h : h \mid x \wedge h \mid y \Rightarrow h \mid g$
  Beachte: hier kommt kein „$<$“ vor!
- Eigenschaften:
  $\gcd(0, y) = y, \ \ \gcd(x, x) = x, \ \ \gcd(x, y) = \gcd(y, x)$
  $\gcd(x, y) = \gcd(x, x - y)$

Algorithmus von Euklid (Variante 1)

```
Eingabe: nat x, nat y;  Ausgabe: gcd(x,y)
// x = X, y = Y (Bezeichung für Eingabe)
while true  // I: gcd(x,y) = gcd(X,Y)
   if x = 0 then return y
   if y = 0 then return x
   if x = y then return x
   if x < y then y := y - x else x := x - y
```

- partielle Korrektheit: wg. Schleifen-Invariante `I`:
  1. anfangs wahr, 2. invariant, 3. schließlich nützlich
- Termination: `x+y` ist *Schranke*: (bound, ranking function)
  - Wertebereich $\subseteq \mathbb{N}$
  - Wert nimmt bei jedem Schleifendurchlauf ab
- Komplexität: linear in $x + y$, exponentiell in $\log x + \log y$

Entwurf und Analyse von Schleifen

- zu jeder Schleife gehören:
  - Spezifikation
    - die *Invariante* I (eine logische Formel)
    - die *Schranke* S (ein Ausdruck mit Wert $\in \mathbb{N}$)
  - Programmtext `while B do K` (B edingung, K örper)

- – I ist vor der Schleife wahr
  - I ist invariant unter K
  - I ist nach der Schleife nützlich
  - K verringert S oder macht B falsch

(David Gries: *The Science of Programming*, Springer, 1981; Kapitel 11)

Multiplizieren durch Verdoppeln

- Multiplikation durch: Halbieren, Verdoppeln, Addieren
- Eingabe: nat a, nat b; Ausgabe: a * b

```
// A = a, B = b
p := ...
while a > 0  // I: a * b + p = A * B
  // Schranke: log (a)
  if odd a then p := ...
  (a, b) = (div(a,2), ...)
return p
```

- Anwendung in der Kryptographie (RSA): Potenzieren (nach einem Modul) durch Halbieren, Quadrieren, Multiplizieren

Algorithmus von Euklid (Variante 2)

- Plan: wiederholte Subtraktion $\Rightarrow$ Division mit Rest
- Def: mod$(x, y) = z \iff \exists f : x = f \cdot y + z \wedge 0 \leq z < y$
- Satz: $y > 0 \Rightarrow \gcd(x, y) = \gcd(\text{mod}(x, y), y)$
- `Eingabe: nat x, nat y;  Ausgabe: gcd(x,y)`

```
// x = X, y = Y (Bezeichung für Eingabe)
while true  // I: gcd(x,y) = gcd(X,Y)
  if y = 0 then return x
  (x,y) := (y, mod(x,y)) // simultan!
```

- partiell korrekt wg. `I` (wie Variante 1)
- Schranke: $x + y$ wie vorher — es gibt besser Schranke

Laufzeit von Euklid (Variante 2)

- Folge der Belegungen $(x_s, y_s), (x_{s-1}, y_{s-1}), \ldots, (x_1, 0)$.
  wobei $\forall k : y_k = x_{k-1} \wedge \mathsf{mod}(x_k, x_{k-1}) = x_{k-2}$ und $x_0 = 0$
- $\forall s > k : x_k > x_{k-1}$ (Rest ist kleiner als Divisor)
- $\forall s > k : x_k - x_{k-1} \geq x_{k-2}$ (Quotient ist ≥ 1)
- daraus folgt $\forall s > k \geq 2 : x_k \geq x_{k-1} + x_{k-2}$
- damit $x_{s-1} \geq F_{s-1}$ für *Fibonacci-Zahlen*
  $F_0 = 0, F_1 = 1, \forall k \geq 0 : F_{k+2} = F_{k+1} + F_k$
- aus $F_k \in \Theta(q^k)$ für $q \approx 1.6$ (Details: nächste Folie)
  folgt $\log_q(x_{s-1}) \geq s$ (Schrittzahl ist $O(\log y)$)

Die Fibonacci-Zahlenfolge

- Def: $F_0 = 0, F_1 = 1, \forall k \geq 0 : F_{k+2} = F_{k+1} + F_k$
- Leonardo Pisano, ca. 1200, `http://www-groups.dcs.st-and.ac.uk/history/Biographies/Fibonacci.html`
-

k	0	1	2	3	4	5	6	10	20	30
F_k	0	1	1	2	3	5	8	55	6765	832040

  wächst exponentiell, Basis ist *goldener Schnitt* $q \approx 1.6$
- Satz: mit $q = (\sqrt{5}+1)/2$ gilt $F_k = (q^k - 1/(-q)^k)/(q + q^{-1})$

  Beweis: Induktion, benutze $q^2 = q + 1$, $q^{-2} = -q^{-1} + 1$
- wegen $\lim_{k\to\infty} 1/(-q)^k = 0$ folgt $F_k \in \Theta(q^k)$

Binäre Suche in geordnetem Feld

Spezifikation:

```
Eingabe: a[1..n] schwach monoton steigend, x
Ausgabe: falls exists i: a[i] = x,
  dann ein i mit a[i] = x, sonst NEIN
```

Implementierung (*divide and conquer*)

```
nat l := ... ; nat r := ...
while ... // Schranke: log (r - l)
  // I: a[1 .. l-1] < x , x < a[r+1 .. n]
  nat m = div (l+r,2)
  if a[m] = x then return m
  else if a[m] < x then ...
  else ...
```

Laufzeit: $\Theta(\log n)$.

Binäre Suche in geordnetem Feld (Var. 2)

Spezifikation (beachte: Grenzen sind l und r)

- Eingabe: $a[l..r]$ schwach monoton steigend, x
- Ausgabe: falls $\exists i : a[i] = x$, dann solch ein i, sonst NEIN

Implementierung (mit Rekursion statt Schleife):

```
search (a [l .. r], x) =
  if  l > r  then return NEIN
  else nat m = div (l+r,2);
       if  a[m] = x  then return m
       else if  a[m] < x  then search(a[m+1 .. r],x)
       else search(a[l .. m-1],x)
```

Korrektheit: $x \in a[l \ldots r] \iff a$ nicht leer und

$(x < a[m] \wedge x \in a[l \ldots m-1])$

$\vee x = a[m] \vee (x > a[m] \wedge x \in a[m+1 \ldots r])$

Binäre Suche (Var. 2) in Java

```
boolean search (List<Integer> a, int x) {
 if (a.isEmpty()) return false; else {
  int m = (a.size() - 1) / 2;
  if (a.get(m) == x) return true;
  else if (a.get(m) < x)
   return search(a.subList( 0,          m), x);
  else
   return search(a.subList(m+1,a.size()), x); }
```

- benutzt `interface List<T>` mit Methode
  `List<T> subList(int from, int to)`
  zur Konstruktion einer Teilfolge (slice) als Ansicht (view) (d.h., ohne Daten zu kopieren)
- in Var. 1 (Schleife) ist diese Ansicht *implizit* repräsentiert (durch l und r), das ist *schlecht* für den Leser

Bemerkung zum Programmierstil

- Das, worüber man (in Spezifikation, in Beweis) spricht, soll im Programmtext explizit repräsentiert sein.

  Bsp. hier: Teilfolgen

- Sonst wird der (optische, mentale) Abstand zwischen Spezifikation, Beweis und Implementierung zu groß . . .

- und deswegen (der Beweis schwierig und) die Implementierung wahrscheinlich falsch.

- Wenn man die explizite Repräsentation nicht vernünftig hinschreiben kann, hat man die falsche Sprache gewählt;

- wenn die Repräsentation nicht effizient ist, dann den falschen Compiler.

Die Flagge färben (Vorbereitung: 2 Farben)

E.W. Dijkstra: *A Discipline of Programming* (Kap. 14), Prentice-Hall, 1976

- Eingabe: Folge $a[1 \ldots n] \in \{\text{Blau}, \text{Rot}\}^*$
- Ausgabe: Folge b mit
  - Multimenge von $a =$ Multimenge von b
  - b schwach monoton bzgl. $B < R$

Ansatz (b wird in a konstruiert)

```
while ( ... ) // Schranke: r - l
  // I: a[1..l-1] = B, a[r+1..n] = R
       if a[l] = B then ...
  else if a[r] = R then ...
  else { a[l] <-> a[r]; ... }
```

Die Flagge färben (wirklich: 3 Farben)

- Eingabe: Folge $a[1 \ldots n] \in \{\text{Blau}, \text{Weiß}, \text{Rot}\}^*$
- Ausgabe: Folge b mit
  - Multimenge von $a =$ Multimenge von b
  - b schwach monoton bzgl. $B < W < R$

Ansatz:

```
l = ... ; m = ... ; r = ...
// I: a[1..l-1]=B, a[l..m-1]=W, a[r+1..n]=R
// Schranke?
while ...
  if      a[m]=B { a[m] <-> a[...]; ... }
  else if a[m]=R { a[m] <-> a[...]; ... }
  else           {                  ... }
```

Hinweis zu Übungsaufgabe 3.3

- Eingabe: Felder $a[1..s], b[1..t]$ schwach monoton steigend.
  Ausgabe: ein Paar (i, j) mit $a[i] = b[j]$, falls das existiert.
- Vorgehen: funktional (rekursiv) $\Rightarrow$ iterativ (Schleife)
- verallgemeinere (Teilfolgen): Eingabe: $a[p \ldots s], b[q \ldots t]$
- Zerlegungen $a = a[p] \cup a[p+1 \ldots s]$, $b = b[q] \cup b[q+1 \ldots t]$
  $a \cap b = (a[p] \cup a[p+1 \ldots s]) \cap b$
  $a[p] < b[q] \Rightarrow a[p] \cap b = \emptyset$, also $a \cap b = a[p+1 \ldots s] \cap b$
  entspr. für $a[p] > b[q]$
- Variablen p, q in Schleife repräsentieren $a[p \ldots s], b[q \ldots t]$
  Invariante: $a[1 \ldots s] \cap b[1 \ldots t] = a[p \ldots s] \cap b[q \ldots t]$

Übungsserie 3 (für KW 17)

Aufgabe 3.1 (Euklid, Variante 1)

- (1 P) Kann man Zeile `if x = y ...` weglassen?
- (1 P) Kann man Zeile `if x = 0 ...` weglassen?
- (1 P) Die beiden Anweisungen `if x = 0 ...`, `if y = 0 ...` können aus der Schleife entfernt und *vor* diese geschrieben werden. Warum? Geben Sie die verschärfte Invariante an.
- (1 P) Interessant ist dabei immer nur die Termination, denn partielle Korrektheit gilt in jedem Fall – warum?

Aufgabe 3.2 (Euklid, Variante 3)

Die GNU Multiprecision-Bibliothek (T. Granlund et al., FSF, 1991–2016)

`https://gmplib.org/manual/Binary-GCD.html`

enthält diesen gcd-Algorithmus, der (wie Variante 1, im Gegensatz zu Variante 2) keine Divisionen benutzt (Division durch 2 ist binär leicht realisierbar).

```
while ...
   (a,b) := (abs(a-b), min(a,b)) // 1
   if even(a) then a := div(a,2) // 2
   if even(b) then b := div(b,2) // 3
```

Die Invariante soll auch hier sein: $I : \gcd(a, b) = \gcd(A, B)$ (der gcd der aktuellen a, b ist der gcd der Eingabe).

- (1 P) Begründen Sie, daß I invariant ist unter Anweisung

1, aber *nicht* invariant ist unter Anweisung 2 (ebenso 3)

- (2 P) Deswegen wird Bedingung U : „wenigstens einer von a, b ist ungerade“ betrachtet.

  Begründen Sie, daß $I \wedge U$ invariant ist unter Anweisungen 2 (ebenso 3).

- (1 P) Begründen Sie, daß $I \wedge U$ invariant ist unter Anweisung 1.

- (1 P) Beschreiben Sie das Verhalten dieses Algorithmus für $a =$ sehr groß, $b =$ sehr klein, und vergleichen Sie mit Euklid in Variante 1.

Aufgabe 3.3 (Suche in mehreren Feldern)

Spezifikation:

Eingabe: Felder $a[1..s], b[1..t]$, jedes ist schwach monoton steigend.

Ausgabe: ein Paar (i, j) mit $a[i] = b[j]$, falls das existiert, sonst NEIN.

- (3 P) Geben Sie eine Implementierung mit einer Laufzeit an, die durch eine lineare Funktion von $s + t$ beschränkt ist.

  Benutzen Sie ein Programm mit *einer* Schleife. Geben Sie deren Invariante und Schranke an.

- (2 P) Geben Sie ein Verfahren an, das die Aufgabe in $o(t)$ löst, falls $s = \sqrt{t}$. Benutzen Sie einen Algorithmus aus

dem Skript.

Aufgabe 3.4 (Suche im Rechteck)

Spezifikation:
Eingabe: ein Feld $a[1..n, 1..n]$, eine Zahl x,
wobei jede Zeile und jede Spalte von a schwach monoton steigen
Ausgabe: ein Paar (i, j) mit $a[i, j] = x$, falls das existiert, sonst NEIN.

- (4 P) Geben Sie eine Implementierung mit einer Laufzeit $O(n)$ an.

  Benutzen Sie ein Programm mit *einer* Schleife. Geben Sie deren Invariante und Schranke an.

Sortieren in Feldern und Listen

Übersicht

- untere Schranke für Sortieren durch Vergleichen [WW] 6.3
- Inversionen [WW] 1.3
- Sortieren durch Einfügen [WW] 4.1.3, [OW] 2.1.2
- Shell-Sort [WW] 4.3, [OW] 2.1.3
- Rechnen mit Folgen (Listen)
- Merge-Sort [WW] 7.1, [OW] 2.4

Entscheidungsbäume

- jedes Sortierverfahren, das die Elemente der Eingabe nur *vergleicht* ($a[i] > a[j]$) und *vertauscht*
  - Beispiele sind: Sortieren durch Auswählen, Compare-Exchange-Netzwerke
  - Nicht-Beispiele: Arithmetik $a[i] + a[j]$, Indizierung $b[a[i]]$
- läßt sich für jede Eingabelänge n als Baum darstellen:
  - innere Knoten sind Vergleiche $a[i] > a[j]$, haben 2 Kinder (Fortsetzung bei $>$, bei $\leq$)
  - Blatt-Knoten sind Ausgaben (Permutationen)
- Jede der $n!$ Permutationen kommt ≥ 1 mal vor
- $\Rightarrow$ Höhe des Baumes ist $\geq \log_2(n!)$

Höhe und Breite von Binärbäumen

- Def: Die *Höhe* $h(t)$ eines Baumes t ist die Länge eines längsten Pfades von der Wurzel zu einem Blatt.

- Def: Die Länge eines Pfades $=$ Anzahl der *Kanten* (Schritte) auf dem Pfad

  (die Anzahl der Knoten ist um eins größer)

- Satz: Für jeden Binärbaum t gilt:

  Die Anzahl der Blätter $B(t)$ ist $\leq 2^{h(t)}$.

- Beweis durch Induktion. Anfang: $h(t) = 0 : t$ ist Blatt.

  Schritt: $h(t) > 0$: t hat Kinder t_1, t_2 mit $h(t_i) \leq h(t) - 1$

  $B(t) = B(t_1) + B(t_2) \leq 2 \cdot 2^{h(t)-1} = 2^{h(t)}$

Untere Schranke für Sortierverfahren

- Für jedes vergleichsbasierte Sortierverfahren A gilt:
  für jedes n gilt: es gibt eine Eingabe der Länge n,
  für die A wenigstens $\lceil \log_2(n!) \rceil$ Vergleiche durchführt.
n	1	2	3	4	5	6	7	8	9	10
$\lceil \log_2(n!) \rceil$	0	1	3	5	7	10	13	16	19	22
- welche speziellen Verfahren (für diese n)
  erreichen diese Schranke?
- Die Komplexität des Sortierens ist $\Omega(n \log n)$.
- welche allgemeinen Sortiervefahren sind in $O(n \log n)$?
  Sortieren durch Auswählen nicht — ist in $\Theta(n^2)$

Nebenrechnung: Wachstum von $\log n!$

- Satz: $\log_2 n! \in \Theta(n \log n)$

- Beweis:
  - Umformung: $\log_2 n! = \log_2 \prod_{k=1}^{n} k = \sum_{k=1}^{n} \log_2 k$
  - Abschätzung nach oben
    (mit $k \leq n$, Monotonie des $\log$, Monotonie der Summe)
    $\sum_{k=1}^{n} \log_2 k \leq \sum_{k=1}^{n} \log_2 n = n \log_2 n$
    deswegen $\log_2 n! \in O(n \log n)$
  - Abschätzung nach unten
    $\sum_{k=1}^{n} \log_2 k \geq \sum_{k=\lceil n/2 \rceil}^{n} \log_2 k \geq (n-1)/2 \cdot \log_2(n/2)$
    Für $n \geq 4$ gilt: $(n-1)/2 \geq (n - n/2)/2 = n/4$
    und $\log_2(n/2) = \log_2 n - 1 \geq (\log_2 n)/2$
    deswegen $n \log n \in O(\log_2 n!)$

Zählen von Inversionen

- Def: für eine Folge $a[1 \ldots n]$: Menge der Inversionen ist

  $\mathsf{Inv}(a) := \{(i,j) \mid 1 \leq i < j \leq n \wedge a[i] > a[j]\}$

  $\#\,\mathsf{Inv}(a)$: Anzahl der Inversionen von a
- Bsp: $a = [4,2,5,3,1]$,
  $\mathsf{Inv}(a) = \{(1,2),(1,4),(1,5),(2,5),(3,4),(3,5)\}$
- für alle a der Länge n: $0 \leq \#\,\mathsf{Inv}(a) \leq n(n-1)/2$
- Satz: wenn $(k,k+1) \in \mathsf{Inv}(a)$

  und b aus a durch $a[k] \leftrightarrow a[k+1]$ entsteht

  dann $\#\,\mathsf{Inv}(b) = \#\,\mathsf{Inv}(a) - 1$.
- Bsp (forgesetzt von oben): $k = 3$, $b = [4,2,3,5,1]$

  $\mathsf{Inv}(b) = \{(1,2),(1,3),(1,5),(2,5),(4,5)\}$

Zählen von Inversionen (Beweis)

Bijektion zwischen $\mathsf{Inv}(a) \setminus \{(k, k+1)\}$ und $\mathsf{Inv}(b)$:
durch vollst. Fallunterscheidung für $(i, j) \in \mathsf{Inv}(a)$:

- (i, j) und $(k, k+1)$ sind disjunkt:

  falls $\{i, j\} \cap \{k, k+1\} = \emptyset$, dann $(i, j) \in \mathsf{Inv}(b)$.

- (i, j) und $(k, k+1)$ stimmen an einer Stelle überein:
  - falls $i < k = j$, dann $(i, j+1) \in \mathsf{Inv}(b)$
  - falls $i < k \wedge k+1 = j$, dann $(i, j-1) \in \mathsf{Inv}(b)$
  - falls $i = k \wedge k+1 < j$, dann $(i+1, j) \in \mathsf{Inv}(b)$
  - falls $i = k+1 < j$, dann $(i-1, j) \in \mathsf{Inv}(b)$

- (i, j) und $(k, k+1)$ stimmen an beiden Stellen überein:
  das ist genau Inversion, die entfernt wird

Inversionen und einfache Sortierverfahren

- Satz: Jedes Sortierverfahren, das nur benachbarte Elemente vertauscht,

  hat Komplexität $\Omega(n^2)$.

- Beweis: jeder solche Tausch entfernt nur eine Inversion,

  es gibt aber Eingaben mit $n(n-1)/2$ Inversionen.

- Anwendung 1: (Satz) jedes Sortiernetz der Breite n, das nur benachbarte Positionen vergleicht (und tauscht),

  benötigt $\geq n(n-1)/2$ CEX-Bausteine.

- Anwendung 2: (Plan) Sortieren durch Auswählen verbessern durch größere Abstände

Inversionen und Laufzeit

- Sortieren durch Auswählen dauert *immer* $|a|^2/2$
- wenn $\mathsf{Inv}(a)$ klein, dann ist a „fast monoton,“
- dann sollte das Sortieren schneller gehen.
- Gibt es Sortierverfahren mit Komplexität in $O(\#\,\mathsf{Inv}(a))$?
- Nein, selbst wenn $\#\,\mathsf{Inv}(a) = 0$, muß man die gesamte Eingabe lesen, um Monotonie festzustellen
- realistisches Ziel: Sortierverfahren in $O(|a| + \#\,\mathsf{Inv}(a))$

Sortieren durch Einfügen

- Plan (Sortieren durch Einfügen):

```
for i from 2 to n // I: a[1..i-1] monoton
  füge  a[i]  in  a[1..i-1]  ein
```

- Realisierung (durch Vertauschungen):

```
// I1: a ist Permutation der Eingabe
//     und a[1..i-1] ist monoton steigend
for i from 2 to n { nat j := i-1;
  // I2: a[1..j] monoton und a[j+1..i] monc
  // und (0<j and j<i-1) => a[j] <= a[j+2]
  while (j >= 1 and a[j] > a[j+1])
    a[j] <-> a[j+1]; j := j-1; }
```

- Zeit: $\Theta(n + \#\,\mathsf{Inv}(a))$

Shell-Sort: Vorbereitung

- (Eigenschaft) $a[1..n]$ heißt *k-geordnet*,

  falls $\forall i : a[i] \leq a[i+k]$

  Bsp: $a = [4, 2, 0, 6, 5, 1, 7, 8, 3]$ ist 3-geordnet.

- (Algorithmus) a wird *k-sortiert*:

  für i von 1 bis $k-1$: sortiere $a[i, i+k, i+2k, \ldots]$

  Bsp: 2-Sort von a ist $[0, 1, 3, 2, 4, 6, 5, 8, 7]$

- Satz: für $h < k$: wenn a k-geordnet ist und h-sortiert wird,

  ist es danach immer noch k-geordnet.

Shell-Sort: Vorbereitung—Beweis

- Bezeichnungen: k-geordnetes a , Resultat $b := h$-Sort(a)
- Beweis: zu zeigen ist $\forall i : b[i] \leq b[i+k]$
- Fall 1: wenn $h \mid k$, dann $b[i] \leq b[i+h] \leq \cdots \leq b[i+p \cdot h]$.
- Fall 2: wenn nicht $h \mid k$,

  dann $E = a[\ldots, i-h, i, i+h, \ldots]$

  und $F = a[\ldots, i+k-h, i+k, i+k+h, \ldots]$

  $b[i+k]$ ist $\geq$ als $(i+k)/h$ Elemente von F.

  Davon sind i/h Stück $\geq$ als i/h Stück von E.

  Die sind $\geq$ als die i/h kleinsten von E, d.h. $\geq b[i]$.

Shell-Sort: Idee

- Algorithmus: für eine geeignete Folge $h_l > \ldots > h_2 > h_1$:
  für s von l bis 1: a wird h_s-sortiert durch Einfügen
- Satz (trivial). $h_1 = 1 \Rightarrow$ Algorithmus ist korrekt.
- Komplexität: hängt von h ab. z.B. $h = [1]$: quadratisch
- Ziel bei der Konstruktion von h:
  - h_i groß $\Rightarrow$ Teilfolgen kurz $\Rightarrow$ schnell sortiert
  - dabei genügend viele Inversionen entfernen
  - so daß weitere Schritte (kleine h_i) auch schnell
- D.L. Shell, CACM 2(7) 30–32, 1959.
  ausführliche Analyse in [DEK] Band 3, Abschnitt 5.2.1
- Implementierung: `https://gitlab.imn.htwk-leipzig.de/waldmann/ad-ss17/blob/master/kw17/shell.cc`

Shell-Sort: Hilfssatz 1

- Lemma: Wenn $\gcd(h, k) = 1$ und a ist h-geordnet und k-geordnet,

  dann für jedes $d \geq (h-1) \cdot (k-1)$: a ist d-geordnet

- Beweis: $\exists a, b \in \mathbb{N} : d = ah + bk$ (Satz v. Sylvester, 1884)

- Beispiel: $\gcd(4, 7) = 1$, größte nicht als $a \cdot 4 + b \cdot 7$ darstellbare Zahl ist ....

Shell-Sort: Hilfssatz 2

- Lemma: Für Shell-Sort mit Differenzenfolge $[\ldots, h_{s+2}, h_{s+1}, h_s, \ldots]$ mit $\gcd(h_{s+2}, h_{s+1}) = 1$

  gilt nach h_{s+1}-Sortieren: jede h_s-Teilfolge hat $\leq (n/h_s) \cdot ((h_{s+2} - 1) \cdot (h_{s+1} - 1)/h_s)$ Inversionen

- Beweis: das Resultat ist d-geordnet für $d \geq (h_{s+2} - 1) \cdot (h_{s+1} - 1)$.

  Jede Teilfolge hat Länge $\leq n/h_s$

  und ist d-geordnet für $d \geq (h_{s+2} - 1) \cdot (h_{s+1} - 1)/h_s$.

  Von jeder Position jeder Teilfolge aus gibt es $< (h_{s+2} - 1) \cdot (h_{s+1} - 1)/h_s$ Inversionen.

Shell-Sort: eine geeignete Differenzenfolge

- Satz: Shell-Sort mit $[\ldots, 2^2 - 1, 2^1 - 1]$ dauert $O(n^{3/2})$.
- Vor h_s-Sortieren gilt: Anzahl aller Inversionen in allen h_s-Teilfolgen ist $\leq n \cdot 2^{s+3}$ (nach Lemma 2)
- Anzahl aller Inversionen in allen h_s-Teilfolgen ist $\leq h_s \cdot (n/h_s)^2$

  (Anzahl der Folgen, max. Inversionszahl einer Folge)
- das h_s-Sortieren kostet $\leq \min(h_s(n/h_s)^2, n \cdot 2^{s+3})$

  benutze linke Schranke für $h_s > \sqrt{n}$, dann rechte

  $\leq 2^t + 2^{t+1} + \cdots + 2^{t+t/2} + 2^{t+t/2} + \cdots + 2^t \in O(2^{3t/2}) = O(n^{3/2})$.

  (vereinfachte Rechnung mit $n = 2^t, \sqrt{n} = 2^{t/2}, h_s = 2^s$)

Rechnen mit Folgen (Listen)

- abstrakter Datentyp *Folge* L mit Elementtyp E
  mit Operationen
  - Konstruktoren: empty : L, cons : $E \times L \to L$
  - Accessoren (für nicht leere Folgen):
    head : $L \to E$, tail : $L \to L$
  - Test: null : $L \to \mathbb{B}$

  mit Axiomen: $\forall x \in E, y \in L$:
  - 1. head(cons$(x, y)) = x$, 2. tail(cons$(x, y)) = y$
  - 3. null(empty), 4. $\neg$ null(cons(x, y))
- Implementierung (Mathematik): $L = E^*$
- Impl. (Programmierung): Zeiger, `typedef C * L`
  - leere Liste: `nullptr`
  - nichtleer: `struct C {  E head; L tail; }`

Das Zusammenfügen von monotonen Folgen

- Spezifikation: merge : $L \times L \to L$
  (a, b schwach monoton steigend und $c = \text{merge}(a, b)$) $\Rightarrow$
  - Multimenge$(a) \cup$ MM$(b) =$ MM(c)
  - und c ist schwach monoton steigend
- Bsp. merge$([1, 2, 2, 3, 5], [0, 3]) = [0, 1, 2, 2, 3, 3, 5]$
- Implementierung:
  - merge(empty$, b) = b$; merge$(a,$ empty$) = a$
  - merge(cons(x, a'), cons$(y, b')) =$
    wenn $x \leq y$, dann cons$(x,$ merge$(a',$ cons$(y, b')))$
    sonst cons$(y,$ merge(cons$(x, a'), b'))$
- Korrektheit: benutzt MM(cons$(x, y)) = \{x\} \cup$ MM(y)
- Komplexität (Anzahl der Vergleiche) ist $\leq |a| + |b| - 1$
  Das ist auch die Schranke für die Rekursion.

Merge: Korrektheit (1)

zu zeigen: $\mathsf{MM}(a) \cup \mathsf{MM}(b) = \mathsf{MM}(\mathsf{merge}(a, b))$

Beweis durch Induktion nach $|a| + |b|$

- – falls $\mathsf{null}(a)$, dann $\mathsf{merge}(a, b) = b$ und
  $\mathsf{MM}(a) \cup \mathsf{MM}(b) = \emptyset \cup \mathsf{MM}(b) = \mathsf{MM}(b)$.
  – falls $\mathsf{null}(b)$, dann symmetrisch

- Falls $a = \mathsf{cons}(x, a')$ und $b = \mathsf{cons}(y, b')$ mit $x \leq y$,

  dann $\mathsf{merge}(a, b) = \mathsf{cons}(x, \mathsf{merge}(a', b))$ und

  $\mathsf{MM}(\mathsf{merge}(a, b)) = \mathsf{MM}(\mathsf{cons}(x, \mathsf{merge}(a', b))) =$
  $\{x\} \cup \mathsf{MM}(\mathsf{merge}(a', b)$.

  Wegen $|a'| + |b| < |a| + |b|$ ist Induktionsvoraussetzung anwendbar und es gilt $\cdots = \{x\} \cup (\mathsf{MM}(a') \cup \mathsf{MM}(b))$

Wegen Assoziativität:

$\cdots = (\{x\} \cup \mathsf{MM}(a')) \cup \mathsf{MM}(b) = \mathsf{MM}(a) \cup \mathsf{MM}(b)$

- Falls $\ldots$ und $x > y$, dann symmetrisch.

Merge: Korrektheit (2)

Def: a ist monoton $\iff$ null(a) oder $a =$ cons(x, a') und $\forall z \in$ MM$(a') : x \leq z$ und a' ist monoton.

Zu zeigen: wenn a monoton und b monoton, dann merge(a, b) monoton.

Beweis durch Induktion nach $|a| + |b|$

- Falls null(a), dann merge$(a, b) = b$, das ist nach Voraussetzung monoton. Falls null(b), dann symmetrisch

- Falls $a =$ cons(x, a') und $b =$ cons(y, b') mit $x \leq y$,

  nach Def. Monotonie gilt $\forall x' \in$ MM$(a') : x \leq x'$ und $\forall y' \in$ MM$(b') : y \leq y'$

  Wg. Transitivität gilt $\forall y' \in$ MM$(b') : x \leq y \leq y'$

Also $\forall z \in \mathsf{MM}(a') \cup \mathsf{MM}(b) : x \leq z$.

Wg. Multimengen-Erhaltung ist
$\mathsf{MM}(\mathsf{merge}(a', b)) = \mathsf{MM}(a') \cup \mathsf{MM}(b)$

und damit $\forall z \in \mathsf{MM}(\mathsf{merge}(a', b)) : x \leq z$

Nach Induktion ist $\mathsf{merge}(a', b)$ monoton und (eben gezeigt) x ist $\leq$ jedem Element aus $\mathsf{merge}(a', b)$.

Damit ist $\mathsf{cons}(x, \mathsf{merge}(a', b))$ monoton.

- Falls . . . und $x > y$, dann symmetrisch.

Sortieren durch Zusammenfügen (Merge-S.)

- sort$(a) =$
  - wenn $|a| \leq 1$, dann a
  - sonst: merge(sort(l), sort(r)), wobei $(l, r) =$ split(a)
- split$(a[1 \ldots n]) = (a[1 \ldots h], a[h+1 \ldots n])$ mit $h = \lfloor n/2 \rfloor$
- Bsp: sort$([4,1,3,2]) =$ merge(sort$[4,1]$, sort$[3,2]) =$
  merge(merge(sort$[4]$, sort$[1]$), merge(sort$[3]$, sort$[2])) =$
  merge(merge$([4],[1])$, merge$([3],[2])) =$
  merge$([1,4],[2,3]) = [1,2,3,4]$
- erfüllt die Spezifikation des Sortierens (Beweis?)
- funktioniert für beliebige Eingabelänge (nicht nur 2^e)
- Komplexität (Anzahl der Vergleiche)?
  für Beispiel: 5, allgemein: $O(n \log n)$ (noch zu zeigen)

Merge-Sort: Korrektheit (1)

zu zeigen: $\mathrm{MM}(a) = \mathrm{MM}(\mathrm{sort}(a))$.

Beweis durch Induktion nach $|a|$:

- falls $|a| \leq 1$, dann $\mathrm{sort}(a) = a$ und $\mathrm{MM}(\mathrm{sort}(a)) = \mathrm{MM}(a)$

- falls $|a| \geq 2$, dann

  $\mathrm{MM}(a) = \mathrm{MM}(l) \cup \mathrm{MM}(r)$. Es gilt $|l| < |a|$ und $|r| < |a|$,

  also lt. Induktion $\mathrm{MM}(\mathrm{sort}(l)) = \mathrm{MM}(l)$ und
  $\mathrm{MM}(\mathrm{sort}(r)) = \mathrm{MM}(r)$

  nach Korrektheit von merge gilt
  $\mathrm{MM}(\mathrm{sort}(a)) = \mathrm{MM}(\mathrm{merge}(\mathrm{sort}(l), \mathrm{sort}(r))) =$
  $\mathrm{MM}(\mathrm{sort}(l)) \cup \mathrm{MM}(\mathrm{sort}(r)) = \mathrm{MM}(l) \cup \mathrm{MM}(r) = \mathrm{MM}(a)$

Merge-Sort: Korrektheit (2)

zu zeigen: sort(a) ist monoton.

Beweis durch Induktion nach $|a|$:

- falls $|a| \leq 1$, dann sort$(a) = a$ und a ist monoton.

- falls $|a| \geq 2$, dann

  Es gilt $|l| < |a|$ und $|r| < |a|$, nach Induktion sind sort(l) und sort(r) monoton.

  Also ist Voraussetzung für Aufruf von merge$($sort$(l),$ sort$(r))$ erfüllt.

  Nach Spezifikation von merge ist dann das Resultat monoton.

Komplexität von Merge-Sort

- Bezeichnungen
  - $S(n) =$ Kosten für sort auf Eingabe der Länge n
  - $M(p, q) = p + q - 1$ Kosten für merge von Längen p, q
- Berechnung: $S(n) =$
  - wenn $n \leq 1$, dann 0,
  - sonst $S(\lfloor n/2 \rfloor) + S(\lceil n/2 \rceil) + M(\lfloor n/2 \rfloor, \lceil n/2 \rceil)$
- Bsp: $S(2) = S(1) + S(1) + M(1, 1) = 0 + 0 + 1 = 1$,
  $S(3) = S(1) + S(2) + M(1, 2) = 0 + 1 + 2 = 3$,
  $S(4) = S(2) + S(2) + M(2, 2) = 1 + 1 + 3 = 5, \ldots$
- Werteverlauf:

n	0	1	2	3	4	5	6	7	8	9	10
$S(n)$	0	0	1	3	5						25

  vergleiche mit unterer Schranke für Sortieren!

Asymptotische Komplexität von Merge-Sort

- $S(n) =$ if $n < 2$ then 0 else $S(\lfloor n/2 \rfloor) + S(\lceil n/2 \rceil) + n - 1$
- speziell: $S(2^k) =$ if $k < 1$ then 0 else $2 \cdot S(2^{k-1}) + 2^k - 1$
-

k	0	1	2	3	4	5	6
$S(2^k)$	0	1	5	17			321

- Vermutung: $S(2^k) = \ldots$, Beweis durch Induktion nach k
- für beliebiges n: mit $k = \lfloor \log n \rfloor$ gilt $2^k \leq n < 2^{k+1}$,

  benutze spezielle Werte und Monotonie von S
- Satz: $S \in \Theta(n \log n)$, d.h.: Merge-Sort ist ein asymptotisch optimales Sortierverfahren

Übungsserie 4 (für KW 18)

Aufgaben 4.A (autotool)

- Sortierverfahren für 5 Elemente mit höchstens 8 Vergleichen.
- Eigenschaften von Permutationen (k-Ordnung, Inversionszahl), Beispiel:

```
Gesucht ist eine Permutation von [1 .. 9]
mit den Eigenschaften:
And [ Inversions EQ 7
    , Ordered 6
    , Ordered 4
    ]
```

Aufgabe 4.1 (Permutationen)

Bestimmen Sie die Anzahl $A(n)$ der Permutationen von $[1 \ldots n]$, die sowohl 2-geordnet als auch 3-geordnet sind.

- Zusatz: (1 P) Diese Aufgabe erscheint in einem der im Skript zitierten Bücher. Geben Sie die exakte Fundstelle an.
- (1 P) Geben Sie alle solche Permutationen für $n = 4$ an.
- (1 P) Jede solche Permutation ist auch 5-geordnet, warum?
- (2 P) Beweisen Sie $A(n) = F_{n+1}$ (Fibonnaci-Folge).

Aufgabe 4.2 (Shell-Sort)

Für Eingabe $[10, 9, \ldots, 2, 1]$ (o.ä. — wird dann vom Übungsleiter vorgegeben):

- (2 P) Shell-Sort mit Differenzenfolge $[3, 2, 1]$ (o.ä.) an der Tafel vorrechnen.

- (1 P) Dabei Ordnung und Anzahl der Inversionen mit Schranken aus Hilfssatz 1 und 2 aus Skript vergleichen.

Bestimmen Sie experimentell Zahlen c, d, so daß Shell-Sort mit Differenzenfolge $[c, d, 1]$ möglichs wenig Element-Vergleiche ausführt.

- (2 P) auf Eingabe $[100, \ldots, 2, 1]$

- (Zusatz) auf 100 zufälligen Eingaben der Länge 100

Hinweis: Bei der Bearbeitung der Aufgaben könnte diese Implementierung von Shell-Sort nützlich sein:
`https://gitlab.imn.htwk-leipzig.de/waldmann/ad-ss17/blob/master/kw17/shell.cc`.
Bei der Präsentation im Seminar rechnen Sie an der Tafel *ohne* maschinelle Hilfe.

Zusatz-Aufgabe 4.3 (Der Osterhase)

(Wird diskutiert, sobald eine Gruppe eine Lösung anmeldet und sofern in der Übung Zeit ist. Das und evtl. Vergabe von Zusatzpunkten liegt im Ermessen der Übungsleiter.)

Der listige Osterhase hat ein buntes Osterei versteckt, hinter einem der Grasbüschel $1, 2, \ldots, n$.

Der fleißige Student sucht das Ei.

Er erhebt sich vom Schreibtisch, geht zu Grasbüschel i und sieht nach. Ist das Ei dort, hat er gewonnen.

Ist das Ei nicht dort, geht er zum Schreibtisch zurück, um weitere Übungsaufgaben zu bearbeiten.

Hinter seinem Rücken nimmt der Osterhase das Ei von der Stelle j (diese war ungleich i) und versteckt es in einer der maximal zwei direkt benachbarten Positionen ($j-1$ oder

$j+1$, falls diese in $[1 \ldots n]$ liegen).
Nachdem der Student eine Weile überlegt hat, erhebt er sich vom Schreibtisch, . . . (siehe oben).
Untersuchen Sie für jedes $n \geq 1$:

- Kann der Student das Ei finden?

- Wenn ja, wie lange dauert das?

Beispiel: $n = 3$. Schritt 1: teste Position 2. Falls das Ei nicht dort ist, dann ist es in 1 oder 3. Im nächsten Schritt muß es der Hase nach 2 legen, weil das die einzige Nachbarposition von 1 und von 3 ist. Deswegen ist Schritt 2: teste Position 2 sicher erfolgreich.

Übungsserie 5 (für KW 19)

Aufgabe 5.A (autotool)

Merge von a und b mit $|a| = 2$ und $|b| = 4$ (oder ähnlich) mit wenig Vergleichen.

Aufgabe 5.1 (Merge-Sort)

Für Eingabe $[10, 9, \ldots, 2, 1]$ (o.ä. — wird dann vom Übungsleiter vorgegeben):

- (2 P) Merge-Sort an der Tafel vorrechnen.
- (1 P) Dabei die Vergleichs-Operationen aller Teilschritte zählen und mit Schranken aus Vorlesung vergleichen.

Such-Aufgabe: zur Implementierung von `sort` in der Haskell-Standardbibliothek `Data.List`

- (1 P) Wo ist der zugrundeliegende Algorithmus beschrieben (welches ist die älteste zitierte Quelle)?
- Zusatz (1 P) Welche Änderung gegenüber Merge-Sort enthält dieser Algorithmus, und für welche Eingabefolgen

ist das eine Verbesserung?

Aufgabe 5.2 (Entscheidungsbäume, Merge)

Benutzen Sie Entscheidungsbäume, um eine untere Schranke für die Anzahl der notwendigen Vergleiche bei merge(a, b) als Funktion von $|a|$ und $|b|$ zu erhalten.
Die Blätter des Entscheidungsbaumes sind Permutationen von $a \cdot b$, welche die relative Reihenfolge der Element aus a sowie der Elemente aus b beibehalten.
Beispiel: eine solche Permutation von $[a_1, a_2] \cdot [b_1, b_2, b_3]$ ist $[b_1, a_1, b_2, b_3, a_2]$.

- (1 P) Geben Sie alle Permutationen von $[a_1, a_2, b_1, b_2, b_3]$ mit dieser Eigenschaft an.

- (1 P) Welche untere Schranke folgt daraus für $M(2, 3)$?

  Wird diese von der in VL angegebenen Implementierung

von merge erreicht?

- (1 P) Bestimmen Sie die minimale Anzahl der Blätter sowie die daraus resultierende Schranke allgemein als Funktion von $|a|$ und $|b|$.

  Hinweis: verwenden Sie eine aus der Kombinatorik bekannte Funktion/Notation.

- (1 P) Begründen Sie: unsere Implementierung von merge ist nicht optimal für $|a| = 1$.

- (1 P) Geben Sie für $|a| = 1$ eine optimale Implementierung an. Benutzen Sie einen Algorithmus aus der Vorlesung (leicht modifiziert).

Analyse rekursiver Algorithmen

Übersicht

[WW] Kapitel 7

- Multiplizieren durch Verdoppeln
- Karatsuba-Multiplikation [OW] 1.2.3
- Master-Theorem [WW] 7.2
- Quick-Sort, Laufzeit im Mittel [WW] 7.3
- Median-Bestimmung in Linearzeit [WW] 7.4

Definition, Motivation

- Wortbedeutung *Rekursion*: zurück (re) laufen (cursor)
- inhaltliche Bedeutung: ein *rekursiver Algorithmus*
  löst ein Problem mit Eingabe E durch:
  1. E in (kein, ein oder mehrere) $E_1, \ldots$ zerlegen
     - kein: ergibt Ende der Rekursion
     - ein: lineare Rekursion
     - mehrere: Baum-Rekursion
  2. jedes E_i durch denselben Algorithmus lösen (ergibt A_i)
  3. aus $A_1, \ldots$ die Lösung A von E bestimmen
- Vorteil der Rekursion: man muß nur über 1. (teilen) und 3. (zusammensetzen) nachdenken, denn 2. ist nach Induktion richtig

Termination und Komplexität rekursiver Algorithmen

- Termination: folgt aus Existenz einer *Schranke* S, die bei jedem rekursiven Aufruf abnimmt: wenn $E \longrightarrow E_i$, dann $S(E) > S(E_i) \geq 0$
- lineare Rekursion: Laufzeit $\leq$ Schranke
- Baum-Rekursion: Laufzeit mglw. größer als Schranke

  `a(n) = if n>0 then a(n-1) + a(n-1) else 1`

  Schranke ist n (linear), Laufzeit ist 2^n (exponentiell)
- Laufzeit *implizit* (Bsp. $S(n) = 2 \cdot S(n/2) + n - 1$) *explizite* Form (Bsp. $S \in \Theta(n \log n)$) nach Rechnung

Multiplizieren durch Verdoppeln

(vgl. früher angegebener iterativer Algorithmus)

```
mult(nat a, nat b) = if a = 0 then 0
   else mod(a,2) * b + 2 * mult(div(a,2),b)
```

- offensichtlich partiell korrekt nach Def. von mod, div
- terminiert, weil `a` eine Schranke ist
  (falls $a > 0$, dann $a > a/2 \geq \lfloor a/2 \rfloor = \mathsf{div}(a, 2)$)
- `mod(a,2) * b` realisiert durch Fallunterscheidung
- Laufzeit (Anzahl der Additionen)
  implizit: $M(a) = \text{if } a = 0 \text{ then } 0 \text{ else } 1 + M(\lfloor a/2 \rfloor)$
  explizit: $M(a) = \text{if } a = 0 \text{ then } 0 \text{ else } 1 + \lfloor \log_2 a \rfloor$
  Beweis: benutzt $a \geq 2 \Rightarrow \lfloor \log_2 a \rfloor = 1 + \lfloor \log_2 \lfloor a/2 \rfloor \rfloor$

Multiplikation von Binärzahlen

mit bisher bekanntem Verfahren:

- `mult(a,b)` kostet $\log a$ Additionen.
- jede dieser Additionen kostet $\geq \log b$ Bit-Operationen
- Multiplikation kostet $\log a \cdot \log b$ Bit-Operation.

geht das schneller? Überraschenderweise: ja!

- Ansatz: $w = \lceil (\log \max(a,b))/2 \rceil$ (Hälfte der Bitbreite)

  $a = 2^w a_1 + a_0, b = 2^w b_1 + b_0$ mit $0 \leq a_0 < 2^w, 0 \leq b_0 < 2^w$
- Bsp. $a = 13, b = 11, w = 2, a = 2^2 \cdot 3 + 1, b = 2^2 \cdot 2 + 3$
- Bestimme $a\cdot$ aus Teilprodukten der a_1, a_0, b_1, b_0

Multiplikation von Binärzahlen

- $a = 2^w a_1 + a_0, b = 2^w b_1 + b_0$
- $a \cdot b = (2^w a_1 + a_0)(2^w b_1 + b_0) = 2^{2w} a_1 b_1 + 2^w (a_1 b_0 + a_0 b_1) + a_0 b_0$
  eine Multiplikation für Bitbreite $2w$
  $\Rightarrow$ vier Mult. für Breite w und 3 Additionen für Breite $2w$
- Kosten: $M(1) = 1, M(2w) = 4M(w) + 6w$
  Daten:

w	1	2	4	8	16	32
$M(w)$	1	16	88	400	1696	28288

  Vermutung: $M(2^k) = 7 \cdot 4^k - 6 \cdot 2^k$, Beweis: Induktion.
- also $M(w) = 7w^2 - 6w$ für $w = 2^k$
- also $M \in \Theta(w^2)$, also keine Verbesserung

Multiplikation von Binärzahlen

- $a \cdot b = (2^w a_1 + a_0)(2^w b_1 + b_0)$

  benutze Nebenrechnung $p = (a_1 + a_0) \cdot (b_1 + b_0)$

  $a \cdot b = 2^{2w} a_1 b_1 + 2^w (p - (a_1 b_1 + a_0 b_0)) + a_0 b_0$
- eine Multiplikation für Bitbreite $2w$

  $\Rightarrow$ *drei* Mult. für Breite w und 6 Additionen für Breite $\leq 2w$
- Kosten $M'(1) = 1, M'(2w) = 3M'(w) + 12w$

w	1	2	4	8	16	32	2^k
$M'(w)$	1	27	129	483	1641	16689	$25 \cdot 3^k - 24 \cdot 2^k$

- $M'(w) = 25(w^{\log_2 3}) - 24w \in \Theta(w^{\log_2 3}) = \Theta(w^{1.58\ldots}) \subseteq o(w^2)$
- A. Karatsuba, J. Ofman: *Multiplication of Many-Digital Numbers by Automatic Computers*, Doklady Akad. Nauk SSSR 145, 293-294, 1962.

Hauptsatz über Rekursionsgleichungen

- beschreibt asymptotisches Wachstum der Lösung f von
  $f(n) =$ if $n < s$ then c else $a \cdot f(n/b) + g(n)$
  für $a \geq 1, b > 1$, und n/b kann $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ sein
- Herleitung und Aussage:
  - bestimme x aus Ansatz $f(n) = n^x$ (ignoriere g)
    $n^x = a \cdot (n/b)^x \Rightarrow x = \ldots$
  - vergleiche g mit n^x, der größere bestimmt das Resultat
    - polynomiell kleiner: $\exists y < x : g \in O(n^y) \Rightarrow f \in \Theta(n^x)$
    - gleich: $g \in \Theta(n^x) \Rightarrow f \in \Theta(n^x \log n)$
    - poly. größer: $\exists y > x : g \in \Theta(n^y) \Rightarrow f \in \Theta(g)$
      (Version mit $g \in \Omega(n^y)$ siehe [CLR])
- Anwenden für beide Varianten der Binärmultiplikation

Begründung des Hauptsatzes (1)

für $n = b^e$, also $e = \log_b n$. Dann $a^e = n^x$ mit $x = \log_b a$.

$$\begin{aligned} f(b^e) &= a \cdot f(b^{e-1}) + g(b^e) \\ &= a(a \cdot f(b^{e-2}) + g(b^{e-1})) + g(b^e) \\ &= a^e \cdot f(b^0) + \sum_{k=0}^{e} a^{e-k} \cdot g(b^k) \end{aligned}$$

$$\begin{aligned} f(n) &= \Theta(n^{\log_b a}) + a^e \cdot \sum_{k=0}^{e} a^{-k} \cdot g(b^k) \\ &= \Theta(n^x) + n^x \cdot \sum_{k=0}^{e} g(b^k)/a^k \end{aligned}$$

drei Fälle: A) n^x am größten, B) Summanden für alle k ähnlich groß. C) Summand für $k = 0$ am größten.

Begründung des Hauptsatzes (2)

- für $n = b^e$, also $e = \log_b n$. Dann $a^e = n^x$ mit $x = \log_b a$.
- $f(n) = \Theta(n^x) + n^x \cdot \sum_{k=0}^{e} g(b^k)/a^k$
- Falls $g \in O(n \mapsto n^y)$ mit $y < x$, dann $g(b^k) \leq c_0 \cdot b^{ky}$,
- also $\cdots \leq \Theta(n^x) + c_0 \cdot n^x \cdot \sum_{k=0}^{e} (b^y/a)^k$
- die Summe ist geometrische Reihe zur Basis $b^y/a < 1$,

  also $\sum_{k=0}^{e} (b^y/a)^k < \sum_{k=0}^{\infty} (b^y/a)^k = \dfrac{1}{1 - b^y/a} = S$
- also $\cdots \leq \Theta(n^x) + c_0 \cdot n^x \cdot S = \Theta(n^x) + \Theta(n^x)$.

Begründung des Hauptsatzes (3)

- für $n = b^e$, also $e = \log_b n$. Dann $a^e = n^x$ mit $x = \log_b a$.
- $f(n) = \Theta(n^x) + n^x \cdot \sum_{k=0}^{e} g(b^k)/a^k$
- Falls $g \in \Theta(n \mapsto n^x)$, dann $c_1 \cdot b^{kx} \leq g(b^k) \leq c_2 \cdot b^{kx}$,
- dann $n^x \cdot \sum_{k=0}^{e} g(b^k)/a^k \in \Theta(n^x \cdot \sum_{k=0}^{e} (b^x/a)^k)$
  $= \Theta(n^x \cdot \sum_{k=0}^{e} 1) = \Theta(n^x \cdot e) = \Theta(n^x \log n)$
- also $f(n) \in \Theta(n^x) + \Theta(n^x \log n) = \Theta(n^x \log n)$

Begründung des Hauptsatzes (4)

- für $n = b^e$, also $e = \log_b n$. Dann $a^e = n^x$ mit $x = \log_b a$.
- $f(n) = \Theta(n^x) + n^x \cdot \sum_{k=0}^{e} g(b^k)/a^k$
- Falls $\exists y > x : g \in \Theta(n^y)$, dann $c_1 \cdot b^{ky} \leq g(b^k) \leq c_2 \cdot b^{ky}$,
- dann $n^x \cdot \sum_{k=0}^{e} g(b^k)/a^k \in \Theta(n^x \cdot \sum_{k=0}^{e} (b^y/a)^k)$

  Summe ist geom. Reihe zur Basis $q = b^y/a > 1$
- $\cdots = \Theta(n^x \cdot \frac{q^{e+1}-1}{q-1}) = \Theta(n^x (b^y/a)^e) = \Theta(n^x \cdot b^{ye}/a^e)$

  $= \Theta(b^{ey}) = \Theta(g(n))$

Anwendungen des Hauptsatzes

- Mult. durch Verdoppeln: $f(n) = f(n/2) + 1$
  Hauptsatz mit $a = 1, b = 2$, also $x = \log_2 1 = 0; n^x = 1$
  Fall 2: $g \in \Theta(n^x)$, also $f \in \Theta(n^x \log n) = \Theta(\log n)$
- Mult. von Binärzahlen: $f(n) = 4f(n/2) + 3n$
  Hauptsatz mit $a = 4, b = 2$, also $x = \log_2 4 = 2; n^x = n^2$
  Fall 1: $g \in O(n^y)$ mit $y = 1 < x$, also $f \in \Theta(n^x)$
- Karatsuba-Mult. $f(n) = 3f(n/2) + 6n$
  Hauptsatz mit $a = 3, b = 2$, also $x = \log_2 3 \approx 1.58$,
  Fall 1: $g \in O(n^y)$ mit $y = 1 < x$, also $f \in \Theta(n^x)$
- $f(n) = 2f(n/2) + n \log n, \quad a = 2, b = 2, x = \log_2 2 = 1,$
  HS *nicht anwendbar*: $g \notin O(n^{1-\epsilon}), g \notin \Theta(n), g \notin \Theta(n^{1+\epsilon})$
  Lösung (mit anderen Methoden): $f \in \Theta(n \log^2 n)$

Quicksort (Algorithmus)

Spezifikation: Ordnet $a[1 \ldots n]$ schwach monoton steigend

$Q(a[1 \ldots n])$: wenn $n \leq 1$, dann fertig, sonst

$p = a[n]$ (das Pivot-Element)

tausche Elemente in $a[1 \ldots n-1]$, bestimme $k \in [1 \ldots n]$,

so daß $\forall 1 \leq i < k : a[i] < p$ und $\forall k \leq i < n : p \leq a[i]$

vgl. *dutch national flag* (mit 2 Farben)

$a[k] \leftrightarrow a[n]$; $Q(a[1 \ldots k-1])$; $Q(a[k+1 \ldots n])$

- Korrektheit:
  - Multimengen-Erhaltung, weil nur getauscht wird
  - Ordnung: vor den beiden Q-Aufrufen gilt:

    $\forall 1 \leq i < k : a[i] \leq a[k]$ und $\forall k < i \leq n : a[k] \leq a[i]$
- Termination: $|a|$ ist Schranke (aber Laufzeit $\notin O(|a|)$)
- C.A.R. Hoare, Computer Journal 5 (1962), 10–15

Quicksort (Laufzeit)

- Kosten für $|a| = n \geq 2$ abhängig von k:

  $Q(n) = Q(k-1) + Q(n-k) + n - 1$

- Komplexität (d.h. *maximale* Laufzeit über alle Eingaben jeder Länge) ist quadratisch,

  betrachte geordnete Eingabe $[1, 2, \ldots, n]$, dann $k = n$

  $Q(n) = Q(n-1) + Q(0) + n - 1 = Q(n-1) + n - 1$

- Laufzeit bei „ungeordneten“ Eingaben ist besser

  $\Rightarrow$ mittlere Laufzeit (über alle Eingaben) ist besser

- besser Wahl des Pivot-Elements ist möglich

Quicksort (mittle Laufzeit — Herleitung)

- Element-Vergleiche nur beim Partitionieren.

  Eingabe-Elemente x, y werden genau dann verglichen, wenn x oder y das Pivot ist und beide nicht bereits durch eine vorigen Pivot getrennt wurden.

- falls Eingabe eine zufällige Permutation von $[1 \ldots n]$,

  betrachte Elemente (nicht Indizes) $x < y$

  und das erste Pivot p mit $x \leq p \leq y$.

- falls $p \in \{x, y\}$, dann wird x mit y verglichen, sonst nicht.

  die Wahrscheinlichkeit dafür ist $2/(y - x + 1)$.

- die erwartete Vergleichszahl ($=$ Laufzeit) ist

  $Q(n) = \sum_{1 \leq x < y \leq n} 2/(y - x + 1)$.

Quicksort (mittle Laufzeit – Rechnung)

- $Q(n) = \sum_{1 \le x < y \le n} 2/(y - x + 1)$
-

n	1	2	3	4	5	6	7
$Q(n)$	0	1	$\frac{8}{3}$	$\frac{29}{6}$	$\frac{37}{5}$	$\frac{103}{10}$	$\frac{472}{35}$
	0	1	2.7	4.8	7.4	10.3	13.5

- $Q(n) = 2\sum_{k=1}^{n-1} \frac{n-k}{k+1} = 2(n+1)\sum_{k=1}^{n-1} \frac{1}{k+1} - 2(n-1)$
- Unter- und Obersumme des Integrals:

  $\int_2^{n+1} (1/x)\mathsf{d}x \le \sum_{k=1}^{n-1} \frac{1}{k+1} \le \int_1^n (1/x)\mathsf{d}x$

  $\log_{\mathsf{e}}(n+1) - \log_{\mathsf{e}} 2 \le \sum_{k=1}^{n-1} \frac{1}{k+1} \le \log_{\mathsf{e}} n$
- Folgerung: $Q \in \Theta(n \log n)$

  Auf zufälligen Eingaben ist Quicksort asymptotisch optimal.

Quicksort (Verbesserungen)

- unser Pivot: $a[n]$, schlecht für geordnete Eingabe
- ist $a[\mathrm{div}(n, 2)]$ besser?

  für geordnete Eingabe ideal

  aber auch für diese Wahl gibt es schlechte Eingaben (Ü)
- zufällige Wahl des Pivot-Index ist besser
- Implementierung in openjdk-1.8.0: *Dual-Pivot Quicksort*, Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch.
  - zwei Pivots $\Rightarrow$ drei Teilfolgen
  - Pivots berechnet aus mehreren Kandidaten

Median-Bestimmung

- Def: Ein Element m einer Folge $a[1 \ldots n]$ heißt Median:
  $\#\{i \mid a[i] < m\} \leq n/2$ und $\#\{i \mid a[i] > m\} \leq n/2$
  (etwas umständlich wg. Möglichkeit von Duplikaten in a)
- Bsp: median$([3, 0, 1, 20, 2]) = 2$ (vgl. mit Mittelwert)
- Motivation: Median ist der ideale Pivot für Quicksort.
  kann man den Median effizient bestimmen?
- ein naheliegendes, Verfahren ist: sort$(a[1 \ldots n])[\lceil n/2 \rceil]$
  das kann man aber in Quicksort nicht benutzen
- überraschenderweise:
  Median kann in Linearzeit bestimmt werden.

Median ohne Sortieren (ein Spezialfall)

- Median von $[a_1, \ldots, a_5]$ mit 6 Vergleichen?

  d.h. ohne zu sortieren, denn $\lceil \log_2(5!) \rceil = 7$

- Implementierung (*replacement selection*)
  - ein KO-Turnier für $[a_1, \ldots, a_4]$ (3 Spiele $=$ Vergleiche)
  - der Sieger a_s ist nicht der Median (sondern zu groß)
  - setze a_5 dort ein, wo a_s begonnen hat,
    a_5 wiederholt die 2 Spiele von a_s
  - der Sieger a_t ist nicht der Median (sondern zu groß)
  - noch ein Spiel zwischen denen, die gegen a_t verloren haben, der Sieger ist der Median

  (A. Hadian und M. Sobel, 1969, zitiert in [DEK] 5.3.3.)

Median und Selektion (Plan)

- Spezif.: select$(k, a) :=$ ein k-kleinstes Element
- gestattet Implementierung median$(a) :=$ select$(\lceil |a|/2 \rceil, a)$
- Implementierung von select$(k, a[1 \ldots n])$ für n groß:
  - $m :=$ ein Wert in der Nähe des Medians von a
  - $L := \{a[i] \mid a[i] < m\}$; $E = \{\cdots = m\}$; $R := \{\cdots > m\}$
  - falls $k \leq |L|$, dann select(k, L)
  - falls $|L| < k \leq |L| + |E|$, dann m
  - falls $|L| + |E| < k$, dann select$(k - (|L| + |E|), R)$
- Korrektheit: $L \ni x < m < y \in R$ und richtig zählen
- Laufzeit: hängt von geeigneter Wahl von m ab:

  Ziele: weder $|L|$ noch $|R|$ groß, m schnell zu berechnen

Selektion in Linearzeit (Realisierung)

- der Wert m wird so bestimmt:
  - teile a in Gruppen $G_1, G_2, \ldots$ je 5 Elemente
  - für jedes i bestimme $m_i :=$ der Median von G_i
  - $m :=$ der Median von $\{m_1, m_2, \ldots\}$ (rekursiv)
- Lemma: für $|a| = n$ gilt $|L| \geq 3n/10, |R| \geq 3n/10$.

  Beweis: zähle Elemente $x \leq m_i \leq m$.

  Folgerung: $|L| \leq 7n/10, |R| \leq 7n/10$
- Komplexität $S(n)$ für select(k, a) mit $|a| = n$, n groß:

  $S(n) \leq 6n + S(n/5) + n + S(7n/10)$

  Ansatz: $S(n) \leq c \cdot n$. Resultat: $S \in O(n)$
- Blum et al., JCSS 7(4) 448-461, 1973: `https://doi.org/10.1016/S0022-0000(73)80033-9`

Übungsserie 6 (für KW 20)

Aufgabe 6.A (autotool)

zur Bestimmung des Medians von 5 Elementen mit 6 Vergleichen.

Aufgabe 6.2 (Rekursionsgleichungen)

Entscheiden Sie, ob der Hauptsatz anwendbar ist.
Falls ja, lösen Sie dann die Gleichung. (Jeweils 2 P)
Falls nein, lösen Sie dann die Gleichung. (Zusatz: 2 P)

- $T_1(n) = 8 \cdot T_1(n/2) + 4 \cdot n^3$
- $T_2(n) = 8 \cdot T_2(n/4) + n^2$
- $T_3(n) = 8 \cdot T_3(n/2) + n \log n$
- $T_4(n) = 9 \cdot T_4(n/3) + n^2 \log n$

Aufgabe 6.2 (ein seltsames Sortierverfahren)

Zum Sortieren einer Folge $a[1 \ldots n]$ wird dieses rekursive Verfahren V vorgeschlagen:

wenn $n \leq 2$, dann sortiere a mit 0 oder 1 Vergleich.
sonst: $p = \lfloor n/3 \rfloor + 1, q = \lceil 2n/3 \rceil$

$$V(a[1 \ldots q]); V(a[p \ldots n]); V(a[1 \ldots q])$$

- (1 P) Wenden Sie V an auf $[8, 2, 4, 5, 1, 6, 3]$

  (oder ähnlich, wird vom Übungsleiter vorgegeben)

  Expandieren Sie dabei nur die obersten rekursiven Aufrufe. Nehmen Sie an, daß die weiteren korrekte Resultate liefern.

- (2 P) Beweisen Sie die Korrektheit von V. Ist es dabei

wesentlich, ob bei der Bestimmung von p und q auf- oder abgerundet wird?

- (1 P) Bestimmen Sie die Laufzeit von V nach dem Hauptsatz über Rekursionsgleichungen. Ist V ein nützliches Sortierverfahren?

Aufgabe 6.3 (Quicksort)

- (2 P) Quicksort für eine vorgegebene Eingabe (z.B. $[1, 3, 5, 7, 9, 8, 6, 4, 2]$) an der Tafel vorrechnen.

- Um das Verhalten von Quicksort von $a[1 \ldots n]$ bei monotoner Eingabe zu verbessern, wählt man als Pivot immer das Element auf dem mittleren Index.

  Man führt dazu (bei jedem rekursiven Aufruf) für $n > 1$ zunächst $a[\lfloor n/2 \rfloor] \leftrightarrow a[n]$ aus und partitioniert dann $a[1 \ldots n-1]$ usw. wie im Algorithmus aus Skript.

  (1 P) Wie schnell werden monotone Eingaben dadurch sortiert? (1 P) Geben Sie eine für dieses Verfahren aufwendigste Eingabe der Länge 8 an.

Zusatz (1 P) . . . beliebiger Länge n.

- Betrachten Sie diese Variante von Quicksort, für eine vorher gewählte Zahl $B \geq 1$:

  bei *jedem* rekursiven Aufruf: wenn die Länge der Eingabefolge $\leq B$ ist, dann kehren wir sofort zurück, ohne die Eingabe zu verändern. Ansonsten (Eingabe länger als B) Partition und Rekursion.

  Nach Rückkehr aller rekursiven Aufrufe sortieren wir das Array durch Einfügen.

  (2 P) Wie teuer ist das abschließende Sortieren (abhängig von Schranke B und Eingabelänge n)?

Dynamische Programmierung

Übersicht

[WW] Kapitel 8

- Definition
- Münzenwechseln
- längste gemeinsame Teilfolge
- Matrixproduktkette
- RNA-Strukturbestimmung

Beispiel, Definition

- $F(0) = 0, F(1) = 1, \forall n \geq 2 : F(n) = F(n-1) + F(n-2)$
  Bestimmung von $F(n)$ lt. Def. benötigt F_n Zeit, also $\Theta(q^n)$
- das geht aber in Linearzeit, mit Array $f[0 \ldots n]$:
  $f[0] := 0; f[1] := 1;$
  für k von 2 bis n: $f[k] := f[k-1] + f[k-2];$
  return $f[n]$
- Definition: *dynamische Programmierung*
  - Implementierung eines rekursiven Algorithmus
  - durch Vorausberechnung und dann Nachnutzung der Resultate rekursiver Aufrufe
  - besonders wirksam, wenn viele Aufrufe übereinstimmen

Beispiel: Geldwechseln (Plan)

- Spezifikation
  - gegeben: Münzwerte $m \in \mathbb{N}_{>0}^k$, Betrag $s \in \mathbb{N}$
  - gesucht: Münz-Anzahlen $a \in \mathbb{N}^k$ mit $s = \sum_{i=1}^{n} m_i a_i$
  - so daß $\sum_{i=1}^{n} a_i$ (Anzahl der Münzen) minimal
- Satz: wenn a eine optimale Lösung für s ist und a' eine optimale Lösung für $s - m_i$, dann $\sum a \leq 1 + \sum a'$.
- deswegen Implementierung: $A(0) = [0, \ldots, 0]$,
  $A(s) :=$ ein Vektor mit minimalem Gewicht aus:
  für $i \in [1, \ldots n]$: falls $s \geq m_i$:
  bestimme $A(s - m_i)$, füge 1 für i hinzu.
- Baum-Rekursion, Verzweigungsgrad k
- dynamische Programmierung: $A(0), A(1), \ldots, A(s)$

Beispiel: Geldwechseln (Realisierung)

- gegeben: Münzwerte $m \in \mathbb{N}_{>0}^k, s \in \mathbb{N}$
  gesucht: minimale Anzahl von Münzen mit Summe s

- rekursiv:

$$A(s) = \textbf{if } s = 0 \textbf{ then } 0$$
$$\textbf{else } 1 + \min\{A(s - m_i) \mid 1 \leq i \leq k, s \geq m_i\}$$

- dynamische Programmierung: benutze Array $a[0 \ldots s]$
  $a[0] := 0$;
  für t von 1 bis s:
  $a[t] := 1 + \min\{a[t - m_i] \mid 1 \leq i \leq k, t \geq m_i\}$
  return $a[s]$

Verstreute Teilfolgen

- Def: $u \in \Sigma^*$ ist (verstreute) Teilfolge (scattered subsequence) von $v \in \Sigma^*$, Notation $u \leq v$,
  falls $\exists i_1 < \ldots < i_{|u|} : \forall k : u_k = v_{i_k}$
- Bsp: $aba \leq bacbacba$ wegen $i = [2, 4, 8]$
- äquivalent: u entsteht aus v durch Löschen von Zeichen (aber ohne Umordnung der nicht gelöschten)
- Eigenschaften der Relation $\leq$ auf Σ^*
  - (trivial) ist Halbordnung, ist nicht total,
  - (nicht trivial — C. Nash-Williams, 1965) jede Antikette (Menge v. paarweise unvergleichbaren Elementen) ist endlich

Gemeinsame Teilfolgen

- Spezifikation (longest common subsequence)
  - gegeben: $v, w \in \Sigma^*$
  - gesucht: u mit $u \leq v \wedge u \leq w$, so daß $|u|$ maximal

  $\mathsf{lcs}(v, w) :=$ die Länge eines solchen u
- Anwendung: kürzeste Editier-Folge
  - von v zu u: Zeichen löschen ($|v| - \mathsf{lcs}(v, w)$ Stück)
  - von u zu w: Zeichen einfügen ($|w| - \mathsf{lcs}(v, w)$ Stück)
- Anwendungen:
  - (algorithmische) Biologie: Vergleich von RNA-Folgen
  - Analyse der Ähnlichkeit von Texten
  - Bestimmung von Patches bei Quelltextverwaltung (automat. Zusammenführung von unabh. Änderungen)

Bestimmung von LCS (Plan)

- lcs(empty, w) $= 0$; lcs(v, empty) $= 0$
- lcs(cons(v_1, v'), cons(w_1, w')) $=$ das Maximum von ...

  (vollst. Fallunterscheidung nach Startposition einer längsten gemeinsamen Teilfolge)
  - 1 in v und 1 in w: falls $v_1 = w_1$, dann $1 +$ lcs(v', w')
  - ≥ 1 in v und > 1 in w: lcs(cons$(v_1, v'), w'$)
  - > 1 in v und ≥ 1 in w: lcs$(v'$, cons$(w_1, w'))$

  jedes Argument von lcs ist Suffix von v bzw. von w.
- dyn. Prog.: berechne $L[i, j] :=$ lcs$(v[i \ldots |v|], w[j \ldots |w|])$

  für i von $|v| + 1$ bis 1, j von $|w| + 1$ bis 1

  benötigt $\Theta(|v| \cdot |w|)$ Zeit (und Platz)

Bestimmung von LCS (Realisierung)

- Eingabe: $v[1 \ldots p], w[1 \ldots q]$, Ausgabe: lcs$(u, w)$
- benutze Array $L[1 \ldots p+1, 1 \ldots q+1]$
  mit Spezifikation $L[i, j] =$ lcs$(v[i \ldots p], w[j \ldots q])$
- Invariante: die bereits zugewiesenen Werte sind korrekt

  $L[*, q+1] := 0; L[p+1, *] := 0;$
  für i von p bis 1: für j von q bis 1:

  $$L[i, j] := \max\{\textbf{ if } v[i] = w[j] \textbf{ then } 1 + L[i+1, j+1] \textbf{ else } L[i, j+1], L[i+1, j]\};$$

  return $L[1, 1];$

- Quelltexte und Profiling: siehe Archiv
- lange gemeinsame Teilfolge durch Analyse von L

LCS (Anwendung bei Quelltextverwaltung)

- Aufgabe (`git rebase`):
  - betrachte eine Datei in Version A,
  - Entwickler 1 ändert $A \to_1 A_1$,
  - Entwickler 2 ändert $A \to_2 A_2$,
  - ... und will Änderungen von Ent. 1 übernehmen

  `https://git-scm.com/book/en/v2/Git-Branching-Rebasing`

- Lösung:
  - berechne kürzeste Editierfolgen (*patch*) $P_i : A \to A_i$, mittels lcs(A, A_i) auf Zeilen (nicht auf Buchstaben)
  - wenn $P_1 \cap P_2 = \emptyset$, dann wende P_2 auf A_1 an
    sonst markiere Konflikt, erfordet manuelle Lösung

RNA-Sekundärstruktur (Definitionen)

- Bezeichnungen:
  - RNA: Molekülkette, steuert Protein-Synthese
  - Primärstruktur: Folge der Moleküle $\in \{A, C, G, U\}^*$
  - Sekundärstruktur: Paarbindungen der Moleküle
    erlaubte Bindungen: $\{\{C, G\}, \{G, U\}, \{U, A\}\}$
  - Tertiärstruktur: 3-dimensionale Anordnung

- Aufgabe (RNA-Sekundärstruktur-Vorhersage):
  - Eingabe: eine Primärstruktur p
  - Ausgabe: eine dazu passende Sekundärstruktur s
    mit minimaler freier (maximaler gebundener) Energie

- s als Klammer-Wort über Alphabet $\{ \texttt{.}, \texttt{(}, \texttt{)} \}$
  ohne Kreuzungen: $a_1 - a_2, b_1 - b_2, a_1 < b_1 < a_2 < b_2$

RNA-Sekundärstruktur (Implementierung)

- Eingabe: $p[1 \ldots n] \in \{A, C, G, U\}^*$
- Ausgabe: max. Bindungszahl aller Sekundärstrukt. für p
- Implementierung (Plan): SP$(p) =$

  wenn $|p| \leq 4$ (minimal hairpin length), dann 0,

  sonst das Maximum von
  - wenn $\{p[1], p[n]\} \in B$, dann $1 +$ SP$(p[2 \ldots n-1])$
  - für k von 1 bis $n-1$: SP$(p[1 \ldots k])$ + SP$(p[k+1 \ldots n])$
- Realisierung durch dynamische Programmierung:

  Matrix $M[1 \ldots n, 1 \ldots n]$ mit Spezif. $M[i, j] =$ SP$(p[i \ldots j])$.

  füllen in geeigneter Reihenfolge; **return** $M[1, n]$;
- Ruth Nussinov et al., SIAM J. Applied Math, 1978

RNA-Sekundärstruktur (Impl. — Detail)

- . . . die richtigen Werte in geeigneter Reihenfolge:
  - schließlich benötigt wird $M[1, n]$
  - $M[i, j]$ hängt ab von $M[i + 1, j - 1]$
    sowie für $i \leq k < j$: $M[i, k]$ und $M[k + 1, j]$
  - Schranke für diese Abhängigkeiten ist: $(j - i)$
- – insgesamt benötigt werden $M[i, j]$ für $1 \leq i \leq j \leq n$
  - Bestimmung nach aufsteigenden Werten der Schranke:
    erst für $j - i = 0$ (Diagonale), dann $j - i = 1$, usw.
- Realisierung:

  for d **from** 0 **to** n : **for** i **from** 1 **to** $n - d$: $j := i + d, \ldots$

  `https://gitlab.imn.htwk-leipzig.de/waldmann/ad-ss17/tree/master/kw20`

Weiteres zu RNA-Strukturen

- Struktur-Vorhersage: mehr Realismus durch:
  - genaueres Energie-Modell z.B. $E(C, G) = 3, \ldots$
  - Pseudoknoten (2 Klammerntypen) `.((.[[..).]].).` (die Dyn.-Prog.-Matrix wird dadurch 4-dimensional)
- Struktur-Design: (ist schwieriger, kein dyn. Prog.)
  - gegeben ist Sekundarstruktur s (Klammernwort)
  - gesucht ist Primärstruktur p, für die s eine (oder sogar: die einzige) optimale Sekundärstruktur ist

  A. Bau et al.: *RNA Design by Program Inversion*, WCB 2013 `http://cp2013.a4cp.org/sites/default/files/uploads/WCB13_proceedings.pdf`
- F. Hufsky, 2016: `http://bioinfowelten.uni-jena.de/2016/12/07/spielen-fuer-die-wissenschaft/`

Greedy (gierige) Algorithmen

Übersicht

- Definition
- Nochmal Münzenwechseln
- Approximations-Algorithmen für Bin-Packing

Beispiel, Definition

- naive Implementierung f. Münzenwechseln:

  $M(s) :=$ bestimme die größte Münze $m_i \leq s$

  und dann $1 + M(s - m_i)$.

- Bsp: $m = [1, 5, 8]$; $19 \rightarrow [8, 8, 1, 1, 1]$, optimal ist $[8, 5, 5, 1]$

  Bsp: $m = [1, 2, 5, 10], s = 19$ (EUR-Münzen/Scheine)

- Vergleich der Algorithmenentwurfsprinzipien:
  - *rekursiv*: Lösung aus Teillösungen zusammensetzen
  - *dyn. Prog.*: diese Teillösungen effizient verwalten
  - *gierig*: eine Teillösung benutzen, andere ignorieren dadurch evtl. inkorrekte (nicht optimale) Ausgabe eventuell reicht diese für den Anwendungsfall aus

Bin Packing (Koffer packen): Spezifikation

- Spezifikation (als Entscheidungsproblem):
  - Eingabe: Kapazität eines Behälters (Eimer, Koffer) $c \in \mathbb{N}$, Anzahl $e \in \mathbb{N}$, Folge x von Zahlen
  - Frage: gibt es eine Zerlegung von x in höchstens e disjunkte zerstreute Teilfolgen, jede mit Summe $\leq c$?
- Spezifikation (als Optimierungsproblem):
  - Eingabe: c, x wie oben
  - Ausgabe: ein minimales e, so daß . . . (wie oben)
- Bsp: $c = 10, e = 4, x = [3, 3, 3, 3, 5, 6, 6, 6]$
- ist Modell für reale Ressourcen-Zuordnungs-Aufgaben

Bin Packing: Algorithmen

- vollständige Lösung durch Aufzählen aller Zuordnungen
  z: Gegenstand $\rightarrow$ Koffer, davon gibt es $e^{|x|}$ viele
- kein effizienter Algorithms (Polynomialzeit) bekannt
- dynamische Programmierung verbessert die Laufzeit nicht, denn es gibt zu viele Teilprobleme
  (jede Teilmenge von x, also $2^{|x|}$ viele)
- wir betrachten gierige Algorithmen:
  schnell (quadratisch), aber nicht optimal.
- interessant ist dann die Frage, wie weit Ausgabe des Algorithmus höchstens vom Optimum entfernt ist
  (wir suchen Schranke für den Approximationsfehler)

Online- und Offline-Algorithmen

- ein Algorithmus (für Bin Packing) heißt *online*, wenn

  jedes Element x_i ohne Kenntnis von $[x_{i+1}, \ldots, x_n]$ verarbeitet (platziert) wird und die Entscheidung nicht zurückgenommen werden kann

- die ursprüngliche Spezifikation verlangt nur einen *offline*-Algorithmus:

  die Eingabe kann vollständig gelesen werden, bevor eine Entscheidung fällt.

- – (Börsen)handel online: Kaufvertrag wirkt sofort
  – Interbankenhandel offline (Ausgleich bei Tagesabschluß)

Online-Algorithms: First Fit (FF)

- First Fit: für i von 1 bis n:
  - lege x_i in den Eimer mit kleinstem Index, in den es paßt.
- Satz: First Fit benutzt ≤ 2 mal soviele Eimer wie nötig.
- ab jetzt: Kapazität 1, Gewichte $x_i \in \mathbb{Q}$, $S := \sum x_i$
- Lemma: optimale Packing benutzt $\geq \lceil S \rceil$ Eimer.
- Lemma: wenn FF Eimer b_1 bis b_k verwendet, dann ist $b_1 > 1/2, \ldots, b_{k-1} > 1/2$.
- Beweis (Satz):
  - $b_k \geq 1/2 \Rightarrow k \cdot 1/2 \leq \sum b_i = S$
  - $b_k < 1/2 \Rightarrow (k-1) \cdot 1/2 < \sum_1^{k-1} b_i = S - b_k < S - 1/2$

Untere Schranke für Online-Algorithmen

- Satz: für jeden Online-Algorithmus A für Binpacking gibt es eine Eingabe, für die A wenigstens $4/3$ der optimalen Eimerzahl verbraucht.

- Beweis: betrachte m mal $1/2 - \epsilon$, dann m mal $1/2 + \epsilon$.
  - nach Verarbeiten der ersten Hälfte sind b Eimer belegt. Es gilt $b < (m/2) \cdot 4/3$ (sonst Gegenbeispiel)
  - nach Verarbeiten der zweiten Hälfte: jeder neue Eimer enthält genau einen Gegenstand die alten Eimer (b Stück) höchstens zwei G.
  - insgesamt $\geq 2m - b$ Eimer benutzt.
  - Optimum ist m, also $2m - b < m \cdot 4/3$ (sonst Gegenb.)
  - addiere beide Ungl $\Rightarrow$ Widerspruch

Schärfere untere Schranke für FF

- Satz: Es gibt Eingaben, für die FF $\geq 5/3$ des Optimums benutzt.

- Beweis (für ϵ klein genug):

  m mal $1/7 + \epsilon$, m mal $1/3 + \epsilon$, m mal $1/2 + \epsilon$.

  – welche Verteilung liefert FF?
  – welches ist die optimale Verteilung?

Übungsserie 7 (für KW 21)

Organisatorisches

- in KW 21 wegen der Feiertage keine Übungen
- stattdessen
  - autotool-Aufgaben (RNA, LCS, Binpack–FFD)
  - ein Hörsaal-Übung (zur regulären Vorlesungszeit)

Aufgabe 7.1 (RNA-Struktur)

- (1 P) Algorithmus aus Skript für vorgegebene Primärstruktur an der Tafel ausführen.

- (1 P) Modifizieren Sie den Algorithmus aus dem Skript so, daß das genauere Energiemodell aus der zitierten Arbeit von Bau et al. (S. 89 unten) benutzt wird.

- (2 P) Der im Skript angegebene Algorithmus bestimmt die maximale Anzahl (bzw. maximale gebundene Energie) der Bindungen, aber nicht die Bindungen selbst.

  Geben Sie einen Algorithmus an, der aus dem fertig befüllten Array M eine optimale Menge von Paarbindungen bestimmt und als Klammerwort, z.B.

`.((((...))))((...))`, ausgibt.

Begründen Sie die Korrektheit, beschreiben Sie die Laufzeit.

Aufgabe 7.2 (Greed)

Wir nennen ein Münzsystem *einfach*, wenn das gierige Verfahren für jeden Betrag eine Darstellung mit minimaler Münzenzahl liefert.

- (1 P) vollständige und gierige Zerlegungen mit gegebenen Parametern an der Tafel ausrechnen.

- (2 P) Welche der Systeme mit drei Münzen $[x, y, 1]$ mit $10 \geq x > y > 1$ sind einfach?

  Geben Sie dabei für die nicht einfachen Systeme jeweils den kleinsten Betrag an, für den der gierige Algoritmus keine optimale Zerlegung liefert.

  Benutzen Sie ggf. Implementierungen aus

https://gitlab.imn.htwk-leipzig.de/waldmann/ad-ss17/tree/master/kw20

Zusatz (1 P) Welche Systeme $[x, y, 1]$ sind einfach?

- (1 P) (Lese-Übung) Der *Pearson-Test* (zitiert in J. Shallit, 2002, `http://www.cs.uwaterloo.ca/~shallit/Papers/change2.ps`) stellt fest, ob ein System einfach ist. Führen Sie diesen Algorithmus für das Münzsystem $[1, 2, 5, 10, 25]$ (oder ein anderes vom Übungsleiter vorgegebenes) aus.

Aufgabe 7.3 (Bin Packing)

First Fit Decreasing (FFD) $=$ First Fit für die absteigend sortierte Eingabefolge.

- Für eine Eingabefolge aus: je $6n$ Gegenständen der Größen $1/2 + \epsilon, 1/4 + 2\epsilon, 1/4 + \epsilon$ und $12n$ Gegenständen der Größe $1/4 - 2\epsilon$, und ϵ klein genug:
  - (1 P) welches ist die optimale Packung?
  - (2 P) welche Packung wird von FFD berechnet? Was genau heißt dabei „ϵ klein genug“?
- (1 P) warum nützt „First Fit Increasing“ (FF für aufsteigend geordnete Eingabe) nichts?

Bäume als Impl. von Mengen, Abbildungen, Folgen

Übersicht

vgl. [WW] 4.4, 6.2

- Suchbäume, Suchen, Einfügen, Löschen
- Höhen-Balance, Rotation
- evtl. Größen-Balance
- ADT Menge, Vereinigung, Durchschnitt
- ADT Folge, disjunkte Vereinigung (Verkettung)

Bäume in der Graphentheorie

- Graph, Knoten, Kanten, Weg, Zusammenhang, Kreis
- Satz: die folgenden Eigenschaften ungerichteter Graphen $G = (V, E)$ sind paarweise äquivalent:
  - G ist zusammenhängend und G ist kreisfrei
  - G ist (kanten)minimal zusammenhängend
    d. h., G ist zshg. und $\forall e \in E : (V, E \setminus \{e\})$ ist nicht zshg.
  - G ist (kanten)maximal kreisfrei
    d. h., jedes Hinzufügen einer Kante erzeugt einen Kreis

  jede kann als Definition von „G ist Baum“ dienen.
- Eigenschaft: Baum mit n Knoten hat $n - 1$ Kanten.

  Beweis durch Induktion nach n

Bäume in der Informatik

- gerichtet:
  - ein Knoten ist Wurzel, hat Eingangsgrad 0
  - Kanten zeigen von Wurzel weg
  - Knoten mit Ausgangsgrad 0: Blätter
  - die anderen: innere Knoten
- geordnet:
  - die Nachbarn jedes Knoten haben eine Reihenfolge
  - können linkes Kind von rechtem Kind unterscheiden
- markiert: (innerer) Knoten enthält Schlüssel
- Bäume sind wichtige Datenstrukturen:
  - Bäume repräsentieren Mengen (von Schlüsseln)
  - Operationen in $O(\text{Grad} \cdot \text{Höhe})$ möglich

Binärbäume

- Def: mit zwei Arten von Knoten:
  - Blattknoten, äußerer Knoten (keine Kinder, kein Schlüssel): Leaf
  - Verzweigungsknoten, innerer Knoten: (zwei Kinder l, r, ein Schlüssel k): Branch(l, k, r)
- Größe (Anzahl der Knoten)

  $\mathsf{size}(\mathsf{Leaf}) = 1, \mathsf{size}(\mathsf{Branch}(l, k, r)) = 1 + \mathsf{size}(l) + \mathsf{size}(r)$
- Höhe (Länge eines längsten Pfades von Wurzel zu Blatt)

  $\mathsf{height}(\mathsf{Leaf}) = 0$,

  $\mathsf{height}(\mathsf{Branch}(l, k, r)) = 1 + \max\{\mathsf{height}(l), \mathsf{height}(r)\}$
- Satz: $\forall t : \mathsf{size}(t) \leq 2^{\mathsf{height}(t)+1} - 1$

Binärbäume (Beispiele Größe, Höhe)

```
  size (Branch(Branch(Leaf,2,Leaf),3,Branch(
= 1 + size(Branch(Leaf,2,Leaf)) + size (Bran
= 1 + 1 + size (Leaf) + size(Leaf) + 1 + siz
= 1 + 1 +   1         +   1        + 1 +
= 7

  height (Branch(Branch(Leaf,2,Leaf),3,Branc
= 1 + max (height (Branch(Leaf,2,Leaf)), hei
= 1 + max (1 + max(height(Leaf),height(Leaf)
= 1 + max (1 + max(0,0)                    ,
= 1 + max (1 + 0, 1 + 0)
= 2
```

Binärbäume (Beweis Größe/Höhe)

Satz: $\forall t$: size$(t) \leq 2^{\text{height}(t)+1} - 1$

Beweis: Induktion nach height(t):

- Anfang: height$(t) = 0 \Rightarrow t =$ Leaf $\Rightarrow$ size$(t) = 1$

- Schritt: wenn $0 < h =$ height(t), dann $t =$ Branch(l, k, r)

  height$(l) \leq$ height$(t) - 1$ und height$(r) \leq$ height$(t) - 1$.

  size$(t) =$ size(Branch$(l, k, r)) = 1 +$ size$(l) +$ size$(r) \leq$
  $1 + 2^{\text{height}(l)+1} - 1 + 2^{\text{height}(r)+1} - 1 \leq$
  $2^{\text{height}(t)-1+1} + 2^{\text{height}(t)-1+1} + (1 - 1 - 1) = 2^{\text{height}(t)+1} - 1$

Binärbäume (Realisierungen)

- in Sprachen mit algebraischen Datentypen (Haskell)

```
data Tree k = Leaf
            | Branch (Tree k) k (Tree k)
```

- in Sprachen mit Zeigern (C, Java, ...)

```
struct branch
  {tree left, K key, tree right}:
leaf = nullptr;
typedef branch * tree;
```

- implizite Realisierungen

  z.B.: $a[1 \ldots n]$, Kinder von $a[i]$ sind $a[2i]$ und $a[2i+1]$

Baum-Durchquerungen

- Abbildung von Binärbaum zu *Folge* von Schlüsseln
  immer l vor r, Lage von k unterschiedlich:
- $\mathsf{pre}(\mathsf{Leaf}) = [], \mathsf{pre}(\mathsf{Branch}(l,k,r)) = [k] \circ \mathsf{pre}(l) \circ \mathsf{pre}(r)$
- $\mathsf{ino}(\mathsf{Leaf}) = [], \mathsf{ino}(\mathsf{Branch}(l,k,r)) = \mathsf{ino}(l) \circ [k] \circ \mathsf{ino}(r)$
- $\mathsf{post}(\mathsf{Leaf}) = [], \mathsf{post}(\mathsf{Branch}(l,k,r)) = \mathsf{post}(l) \circ \mathsf{post}(r) \circ [k]$
- Preorder und Postorder auch für beliebigen Knotengrad.
  Inorder nur für Binärbäume sinnvoll definierbar.
- ohne Berücksichtigung der Reihenfolge: die *Menge* der Schlüssel.
  $\mathsf{keys}(\mathsf{Leaf}) = \emptyset$,
  $\mathsf{keys}(\mathsf{Branch}(l,k,r)) = \mathsf{keys}(l) \cup \{k\} \cup \mathsf{keys}(r)$

Baum-Durchquerungen (Beispiele)

```
t = Branch(Branch(Leaf,2,Leaf),1,Branch(Bran

ino(t)
= ino(Branch(Branch(Leaf,2,Leaf),1,Branch(Br
= ino(Branch(Leaf,2,Leaf)) ++ [1] ++ ino(Bra
= ino(Leaf) ++ [2] ++ ino(Leaf) ++ [1] ++ in
= [] ++ [2] ++ [] ++ [1] ++ ino(Leaf) ++ [4]
= [2,1,4,3]

pre(t) = [1,2,3,4]
post(t) = [2,4,3,1]

keys(t) = {1,2,3,4}
```

Suchbäume — Definition, Suchen

- Def: ein binärer Baum t heißt *Suchbaum* gdw. ino(t) ist monoton steigend.
- äquivalent: *für jeden Teilbaum* Branch(l, k, r) von t gilt: $\forall x \in \mathsf{keys}(l) : x < k$ und $\forall x \in \mathsf{keys}(r) : k < x$
- Such$(K) :=$ Menge der Suchb. mit Schlüsseln vom Typ K
- Spezif.: contains$(x, t) \iff (x \in \mathsf{keys}(t))$, Impl.:
  contains$(x, \mathsf{Leaf}) = \mathsf{false}$
  contains$(x, \mathsf{Branch}(l, k, r)) =$ **if** $x = k$ **then** true **else if** $x < k$ **then** contains(x, l) **else** contains(x, r)
- Korrektheit: durch Induktion nach height(t), ist Schranke für lineare Rekursion, Komplexität: $\Theta(\mathsf{height}(t))$

Suchbäume — Suchen (Beispiel)

```
  contains (4, Branch(Branch(Leaf,2,Leaf),3,
= contains (4, Branch(Leaf,5,Leaf))
= contains (4, Leaf)
= false
```

Suchbäume — Einfügen

- Spezifikation: insert : $K \times \mathsf{Such}(K) \to \mathsf{Such}(K)$,
  $\mathsf{keys}(\mathsf{insert}(x, t)) = \{x\} \cup \mathsf{keys}(t)$
- Implementierung:
  $\mathsf{insert}(x, \mathsf{Leaf}) = \mathsf{Branch}(\mathsf{Leaf}, x, \mathsf{Leaf})$
  $\mathsf{insert}(x, \mathsf{Branch}(l, k, r)) =$
  if $x = k$ **then** $\mathsf{Branch}(l, k, r)$
  else if $x < k$ **then** $\mathsf{Branch}(\mathsf{insert}(x, l), k, r)$
  else $\mathsf{Branch}(l, k, \mathsf{insert}(x, r))$
- Korrektheit: Induktion. Zu zeigen ist (u.a.), daß jeder konstruierte Knoten die Suchbaum-Eigenschaft hat
- Termination/Komplexität: lineare Rekursion mit Schranke $\mathsf{height}(t)$

Suchbäume — Einfügen (Beispiel)

```
  insert (4,Branch(Branch(Leaf,3,Leaf),5,Bra
= Branch(insert(4,Branch(Leaf,3,Leaf)),5,Bra
= Branch(Branch(Leaf,3,insert(4,Leaf)),5,Bra
= Branch(Branch(Leaf,3,Branch(Leaf,4,Leaf)),
```

Suchbäume — Einfügen (Beweis)

1. Lemma $t \in \mathsf{Such}(K) \iff t = \mathsf{Leaf} \vee t = \mathsf{Branch}(l, k, r) \wedge l \in \mathsf{Such}(K) \wedge r \in \mathsf{Such}(K) \wedge \forall x \in \mathsf{keys}(l) : x < k \wedge \forall x \in \mathsf{keys}(r) : k < x$.

Beweis:

- „$\Rightarrow$“ :ino$(\mathsf{Branch}(l, k, r))$ monoton $\Rightarrow$ ino(l) monoton und ino(r) monoton, also nach Induktion $l, r \in \mathsf{Such}(K)$.

2. Korrektheit des Einfügens: zu zeigen sind

- $\mathsf{keys}(\mathsf{insert}(x, t)) = \{x\} \cup \mathsf{keys}(t)$ — das ist einfach

- $x \in K, t \in \mathsf{Such}(K) \Rightarrow \mathsf{insert}(x, t) \in \mathsf{Such}(K)$

  Bsp. betrachte letzten Zweig des Algorithmus: zeige, daß (unter den gegebenen Voraussetzungen)

Branch$(l, k,$ insert$(x, r)) \in$ Such(K).

Es gelten:

$\forall y \in$ keys$(l) : y < k$ weil t Suchbaum ist

$\forall y \in$ keys(insert$(y, r)) : k < y$, weil keys(insert$(x, r)) = \{x\} \cup$ keys(r) nach Induktion und $k < x$ (haben wir getestet) und $\forall y \in$ keys$(r) : k < y$, weil t Suchbaum ist.

Persistente und ephemere Datenstrukturen

- wir betrachten *persistente* (unveränderliche) Suchbäume:
  - werden einmal konstruiert und nie verändert,
  - Operationen (wie insert) erzeugen neuen Baum
- das Gegenteil sind *ephemere* (veränderliche) Strukturen:
  - Operationen können Daten „in-place“ verändern
- vollständige Persistenz hat diese Vorteile:
  - Semantik nicht von implizitem Zustand abhängig
  - beliebiges *sharing* von Teilstrukturen möglich
  - einfache Parallelisierung von Operationen
- ist „Operationen erzeugen neuen Baum“ ineffizient?

  nein, es werden nur $O(\text{height}(t))$ neue Knoten allokiert.

Bestimmen und Löschen des Minimums

- Spezifikation:

  extractMin : Such$(K) \to K \times$ Such(K)

  wenn keys$(t) = M \neq \emptyset$,

  dann extractMin$(t) = (m, t')$

  mit $m = \min M$ und keys$(t') = M \setminus \{m\}$.

- Implementierung: extractMin(Branch$(l, k, r)) =$
  - falls $l =$ Leaf, dann ...
  - sonst **let** $(m, l') =$ extractMin(l) **in** ...

- lineare Rekursion, Laufzeit: $O($height$(t))$

Suchbäume — Löschen

- Spezifikation: delete : $K \times \mathsf{Such}(K) \to \mathsf{Such}(K)$

  $\mathsf{keys}(\mathsf{delete}(x, t)) = \mathsf{keys}(t) \setminus \{x\}$

- Implementierung:
  - $\mathsf{delete}(x, \mathsf{Leaf}) = \ldots$
  - $\mathsf{delete}(x, \mathsf{Branch}(l, k, r)) =$
    - falls $x < k$, dann $\mathsf{Branch}(\mathsf{delete}(x, l), k, r)$
    - falls $x > k$, dann ...
    - falls $x = k$, dann
      - falls $r = \mathsf{Leaf}$, dann ...
      - sonst **let** $(m, r') = \mathsf{extractMin}(r)$ **in** ...

- lineare Rekursion, Laufzeit: $O(\mathsf{height}(t))$

Übungsserie 8 für KW 22

Aufgabe 8.A (autotool)

- Rekonstruktionsaufgaben pre/in/post-order
- Einfügen/Löschen in Suchbäumen

Grundsätzliches

Auch wenn es nicht explizit dasteht: *alle Aussagen sind zu begründen*.

Es mag ja sein, daß in der Schule trainiert wird, „bestimmen Sie“ von „geben Sie an“ zu unterscheiden.

Sie sind jetzt aber an einer Hochschule, und dort gibt es nicht für jeden Arbeitsschritt eine separate Einladung und Anleitung,

denn das Studienziel ist, daß Sie wissenschaftliche Methoden des Fachgebietes *selbständig* anwenden.

Das wissenschaftliche Vorgehen zeichnet sich gerade dadurch aus, daß jeder Schritt begründet und jedes Resultat überprüfbar ist.

Aufgabe 8.1 (Bäume)

Für die durch
$B(n) :=$ **if** $n >$
0 **then** Branch$(B(n-1), n, B(n-1))$ **else** Leaf
definierten Binärbäume:

- (1 P) Wieviele innere Knoten hat $B(n)$?
- (1 P) Wieviele Blätter?
- (2 P) Summe der Schlüssel?

Aufgabe 8.2 (Suchbäume)

- (1 P) Gegen Sie für $M = \{3, 5, 7, 8, 11, 13, 25\}$ Suchbäume t mit keys$(t) = M$ der Höhe $2, 3, 4, 5$ an.

- (1 P) Für einen Suchbaum t wird contains$(20, t)$ ausgewertet. Dabei wird der gesuchte Wert der Reihe nach mit $[5, 40, 30, 10, 11, 33, 14]$ verglichen — angeblich. Begründen Sie, daß das nicht stimmen kann.

- (2 P) Beweis oder Gegenbeispiel:
  - $\forall x \in K, t \in \text{Such}(K) : x \in \text{keys}(t) \Rightarrow$ insert$(x, \text{delete}(x, t)) = t$
  - $\forall x \in K, t \in \text{Such}(K) : x \notin \text{keys}(t) \Rightarrow$ delete$(x, \text{insert}(x, t)) = t$

Aufgabe 8.3 (Suchbäume)

Implementieren Sie diese Spezifikation

- up : $K \times \mathsf{Such}(K) \to \mathsf{Such}(K)$

  $\mathsf{keys}(\mathsf{up}(x,t)) = \{k \mid k \in \mathsf{keys}(t) \wedge x \leq k\}$

in Zeit $O(\mathsf{height}(t))$.

(3 P) Ergänzen Sie den Ansatz

- $\mathsf{up}(x, \mathsf{Leaf}) = \ldots$

- $\mathsf{up}(x, \mathsf{Branch}(l,k,r)) =$
  - falls $x \leq k$, dann...
  - falls $x > k$, dann...

(1 P) Bestimmen Sie $\mathsf{up}(5,t)$ für $t =$ der Baum, der durch

Einfügen von $[1, 8, 2, 7, 3, 6, 4, 5]$ in dieser Reihenfolge in Leaf entsteht.

Balancierte Suchbäume

Motivation

- Laufzeit vieler Algorithmen f. Suchbäume: $O(\mathsf{height}(t))$.
- für jeden binären Suchbaum t gilt
  $\mathsf{height}(t) \leq \#\,\mathsf{keys}(t) < 2^{\mathsf{height}(t)}$ (Beweis: Induktion)
  also $\log_2 \#\,\mathsf{keys}(t) < \mathsf{height}(t) \leq \#\,\mathsf{keys}(t)$.
- ein *effizienter* Suchbaum ist ein solcher mit Höhe
  $\log_2 \#\,\mathsf{keys}(t)$ . . . oder $\Theta(\log \#\,\mathsf{keys}(t))$.
- das kann erreicht werden durch eine *strukturelle Invariante* („jeder Teilbaum ist balanciert“)
- das Herstellen der Invariante (bei jedem Konstruktor-Aufruf) darf aber nicht zu aufwendig sein

Zu strenge Balance

- Ein Binärbaum t heißt *vollständig*, wenn jedes Blatt die Tiefe height(t) hat.
  - $\Rightarrow \#\,\mathsf{keys}(t) = 2^{\mathsf{height}(t)} - 1$, also $\mathsf{height}(t) = \log_2 \#\,\mathsf{keys}(t)$
  - gibt es nur für diese Schlüssel-Anzahlen!
- . . . *fast vollständig*, falls jede Blatt-Tiefe $\in \{h, h-1\}$ mit $h = \mathsf{height}(t)$ und Blätter mit Tiefe h links von denen mit Tiefe $h-1$ stehen
  - $\Rightarrow 2^{h-1} \leq \#\,\mathsf{keys}(t) \leq 2^h - 1$, $\Rightarrow h = \lceil \log_2 \#\,\mathsf{keys}(t) \rceil$
  - für jede Schlüssel-Anzahl genau ein solcher Baum,
  - finde t_1, t_2 mit $\mathsf{keys}(t_1) = \{2 \ldots 7\}$, $\mathsf{keys}(t_2) = \{1 \ldots 7\}$,
  - was folgt für die Komplexität des Einfügens?

Höhen-Balance (Definition)

- Def: ein Baum t heißt *höhen-balanciert*,

  falls für jeden Teilbaum Branch(l, k, r) von t gilt:

  height(l) − height$(r) \in \{-1, 0, +1\}$.

- die Menge der höhen-balancierten Suchbäume: AVL(K),

  mit Höhe h: AVL$_h(K)$

  Adelson-Velski, Landis; 1962

- das ist eine nützliche strukturelle Invariante, denn:
  - $t \in$ AVL$(K) \Rightarrow$ height$(t) \in \Theta(\log($size$(t)))$
  - eine durch Einfügen eines Schlüssels zerstörte Invariante läßt sich leicht reparieren

Höhen-Balance erzwingt logarithmische Höhe

- Satz: $t \in \mathsf{AVL}(K) \Rightarrow \#\,\mathsf{keys}(t) \geq F_{\mathsf{height}(t)+2} - 1$

  mit Fibonacci-Zahlen $F_0 = 0, F_1 = 1, F_{n+2} = F_n + F_{n+1}$

- Beweis:
  - $\mathsf{height}(t) = 0$. Dann $t = \mathsf{Leaf}$, $\#\,\mathsf{keys}(t) = 0 = F_2 - 1$
  - $\mathsf{height}(t) = 1$. Dann $t = \ldots$
  - $\mathsf{height}(t) = h \geq 2$. Dann $t = \mathsf{Branch}(l, k, r)$
    mit ($\mathsf{height}(l) = h - 1$ und $\mathsf{height}(r) \geq h - 2$)
    oder ($\mathsf{height}(l) \geq h - 2$ und $\mathsf{height}(r) = h - 1$).
    dann $\#\,\mathsf{keys}(t) \geq (F_h - 1) + 1 + (F_{h+1} - 1) = F_{h+2} - 1$

- Anwendg.: $\log(F_h) \in \Theta(h) \Rightarrow \log \#\,\mathsf{keys}(t) \geq c \cdot \mathsf{height}(t)$,
  $\Rightarrow \mathsf{height}(t) \in O(\log \#\,\mathsf{keys}(t))$

Operationen für balancierte Suchbäume

- aus dem Code für insert(x, t), delete(x, t)
  für unbalancierte Suchbäume
  wird Code für balancierte Suchbäume:
- jeder Aufruf des Konstruktors Branch(l, k, r)
  wird ersetzt durch *smart constructor*
  balance(Branch$^*(l, k, r)$),
- dabei gilt $\forall x \in \mathsf{keys}(l) : x < k$, $\forall y \in \mathsf{keys}(r) : k < y$,
  $l, r \in \mathsf{AVL}(K)$ und $|\mathsf{height}(l) - \mathsf{height}(r)| \leq 2$.
  Resultat $\in \mathsf{AVL}(K)$
- Zusatzkosten $O(\mathsf{height}(t))$, falls Rebalancieren in $\Theta(1)$
  für insert: tatsächlich nur $\Theta(1)$

Balancieren durch Rotieren (1)

- Knoten mit Höhendifferenz 2 :
  - Fall 1: $t = \text{Branch}^*(l, k, r)$,
    $l = \text{Branch}(ll, lk, lr) \in \text{AVL}_{h+2}(K)$, $r \in \text{AVL}_h(K)$
    - Fall 1.1: $ll \in \text{AVL}_{h+1}(K), lr \in \text{AVL}_{\{h,h+1\}}(K)$
    - Fall 1.2: $ll \in \text{AVL}_h(K), lr \in \text{AVL}_{h+1}(K)$
  - Fall 2 (Fälle 2.1, 2.2) symmetrisch

- löse Fall 1.1 durch *Rotation nach rechts*
  - ersetze t durch $t' = \text{Branch}(ll, lk, \text{Branch}(lr, k, r))$.
    $\text{keys}(t) = \text{keys}(t'), t \in \text{Such}(K) \Rightarrow t' \in \text{Such}(K)$
    $\text{Branch}(lr, k, r) \in \text{AVL}_{h+1}(K), t' \in \text{AVL}_{h+2}(K)$.
  - beachte: funktioniert auch für $lr \in \text{AVL}_{h-1}(K)$,
    das wird für Behandlung von 1.2 benötigt

Balancieren durch Rotieren (2)

- Fall 1.2: $t = \text{Branch}^*(\text{Branch}(ll, lk, lr), k, r)$,
  $ll \in \text{AVL}_h(K), lr \in \text{AVL}_{h+1}(K), r \in \text{AVL}_h(K)$
- eine Rechts-Rotation allein hilft nicht (nachrechnen!)
- $lr = \text{Branch}(lrl, lrk, lrr)$ mit $lrl, lrr \in \text{AVL}_{\{h,h-1\}}(K)$
- Links-Rot. in $l \Rightarrow l' = \text{Branch}(\text{Branch}(ll, lk, lrl), lrk, lrr)$
  $\text{keys}(l) = \text{keys}(l')$,$l \in \text{Such}(K) \Rightarrow l' \in \text{Such}(K)$,
  $\text{Branch}(ll, lk, lrl) \in \text{AVL}_{h+1}(K)$
- dann erfüllt $\text{Branch}^*(l', k, r)$ die Voraussetzung von 1.1
  (ggf. unter Benutzung von „beachte:“)
  Rechts-Rotation ergibt $t' \in \text{AVL}_{h+2}(K)$.

Korrektheit und Anzahl der Rotationen: Insert

- für alle Aufrufe balance(Branch$^*(l, k, r)$) ist zu zeigen
  $|\,\text{height}(l) - \text{height}(r)| \leq 2$
- folgt aus $\text{height}(t) \leq \text{height}(\text{insert}(x, t)) \leq \text{height}(t) + 1$
- Begründung:
  - durch nicht balanc. Einfügen steigt Höhe um max. 1
  - betrachte untersten Knoten (Teilbaum) t mit Höhendifferenz 2
    dann $\text{height}(\text{balance}(t)) = \text{height}(t)$ nur im Fall ...
    und dieser tritt beim Einfügen nicht ein.
  - $\Rightarrow$ nach Rotieren wieder Original-Höhe
  - $\Rightarrow$ Knoten darübe benötigen keine Rotationen
- insgesamt beim Einfügen höchstens einmal rotieren

Korrektheit und Anzahl der Rotationen: Delete

- für alle (smart) Konstruktor-Aufrufe Branch$^*(l, k, r)$ zeige:
  $|\,\text{height}(l) - \text{height}(r)| \leq 2$
- folgt aus $\text{height}(t) - 1 \leq \text{height}(\text{delete}(x, t)) \leq \text{height}(t)$
- wenn beim Rotieren die Höhe verringert wird, dann muß darüber auch evtl. auch noch rotiert werden, usw.
- Das Löschen eines Knotens kann $\Theta(\text{height}(t))$ Rotationen erfordern.
- Bsp: betrachte Fibonacci-Baum
  $T_0 = T_1 = \text{Leaf}, T_{k+2} = \text{Branch}(T_{k+1}, T_k)$
  und lösche den Knoten ganz rechts

Ergänzungen

- AVL-Operationen ausprobieren: benutze autotool-Aufgabe
- Quelltexte `https://gitlab.imn.htwk-leipzig.de/waldmann/ad-ss17/tree/master/kw22/avl`
  `https://gitlab.imn.htwk-leipzig.de/autotool/all0/blob/master/collection/src/Baum/AVL/Ops.hs`
- exakte Spezifikationen mit vollständig maschinell überprüften Beweisen (Nipkow et al., 2004):
  `https://www.isa-afp.org/browser_info/current/AFP/AVL-Trees/AVL.html`

Anwendungen Suchbäume

- realisieren abstrakten Datentyp (ADT) *Menge*
  - `java.util: TreeSet<K> implements Set<K>`
  - contains : $K \times \mathsf{Set}(K) \to \mathbb{B}$, empty, insert, delete
  - Komplexität $O(\mathsf{height}(t))$
  - für balancierte Bäume: $O(\log \# \mathsf{keys}(t))$

- in inneren Knoten Schlüssel *und Wert* speichern:

  realisiert ADT *Abbildung*

  - lookup: $K \times \mathsf{Map}(K, V) \to V \cup \{\bot\}$,
    empty, insert (update), delete (Ü: welche Typen?)
  - Komplexität wie oben

Weitere Operationen für Suchbäume

- Massen-Operationen: intersection, difference,
  union: Set$(K) \times$ Set$(K) \to$ Set(K),
  compose (join): Map$(A, B) \times$ Map$(B, C) \to$ Map(A, C)
- Komplexität für Union(t_1, t_2): durch Iterieren über Schlüsselmenge des kleineren Baumes:
  $\min(|t_1|, |t_2|) \cdot \log \max(|t_1|, |t_2|)$
- durch Implementierung von balance(Branch$^*(l, k, t)$)
  für beliebige $|\,\text{height}(l) - \text{height}(r)|$
  in $O(\text{height}(t_1) + \text{height}(t_2))$
- schnelleres Union für disjunkte Mengen
  (das ist auch nützlich in Teilbäumen)

Übungsserie 9 für KW 23

Aufgabe 9.A

- (autotool) Einfügen und Löschen in AVL-Bäumen

Aufgabe 9.1 (Rotieren)

Aufgabe 13.2–4 aus [Cormen, Leiserson, Rivest, Stein] (deutsche Ausgabe als E-Buch in der Bibliothek)

Die Aussage ist zu ergänzen: „. . . in jeden anderen binären Suchbaum *mit der gleichen Schlüsselmenge* . . .“, sonst ist sie trivial falsch.

(2 P) Lösen Sie die Aufgabe für den Spezialfall, daß t_1 ein vollständiger binärer Suchbaum mit 7 Schlüsseln und t_2 eine rechtsläufige Kette ist.

(2 P) Lösen Sie die Aufgabe allgemein.

Aufgabe 9.2 (AVL-Struktur)

- (2 P) Was ist die minimale Tiefe (d.h. Abstand von der Wurzel) der Blätter aller AVL-Bäume mit Höhe h?

- (2 P) Leiten Sie daraus einen weiteren Beweis für $\text{height}(t) \in O(\log \text{size}(t))$ ab.

- (3 P) Konstruieren Sie einen Baum $t = \text{Branch}(l, k, r) \in \text{AVL}(K)$ mit $\text{size}(l)/\,\text{size}(r) > 1000$.

  (Nicht konkret aufmalen, sondern beweisen, daß es einen gibt. Welche Höhe hat er?)

Aufgabe 9.3 (AVL-Operationen)

- (3 P) Bestimmen Sie $t_1, \ldots, t_7$ für $t_0 = \mathsf{Leaf}, t_k = \mathsf{insert}(k, t_{k-1})$.

  Welche Balance-Operationen finden dabei statt?

- (3 P) Bestimmen Sie $t_1, \ldots, t_7$ für $t_0 =$ vollständiger binärer Suchbaum mit Schlüsseln $\{1, 2, \ldots, 7\}$, $t_k = \mathsf{delete}(k, t_{k-1})$.

  Welche Balance-Operationen finden dabei statt?

Heap-geordnete Bäume

Übersicht

- der abstrakte Datentyp PWS (PQ)
- heap-geordnete Bäume [WW] Kap. 10, [OW] Abschn. 6.1
- vollständig balancierte Binärbaume (Anwendung: Heap-Sort)
- evtl. Ausblick: Quake Heaps

Motivation und Plan

- abstrakter Datentyp: *Vorrang-Warteschlange* (priority queue) (Priorität $\in \mathbb{N}$, kleiner $\Rightarrow$ wichtiger), Operationen:
  - insert : $(K \times \mathbb{N}) \times \mathrm{PQ}(K) \to \mathrm{PQ}(K)$
  - extractMin : $\mathrm{PQ}(K) \hookrightarrow (K \times \mathrm{PQ}(K))$ (partielle Ftk.) bestimmt und entfernt einen wichtigsten Eintrag
  - decrease : $(K \times \mathbb{N}) \times \mathrm{PQ}(K) \to \mathrm{PQ}(K)$ erhöht Priorität

  das ist der Plan, tatsächlich sind die Typen noch anders
- wird u.a. für Bestimmung kürzester Wege benötigt
- die Menge soll *nicht vollständig geordnet* werden, weil das evtl. zu teuer ist (also kein Suchbaum)

Binäre Heaps (Plan)

- wir benutzen Binärbäume t mit diesen Eigenschaften:
  - *heap-geordnet*: für jeden Teilbaum Branch(l, k, r) von t:
    $\forall x \in \mathsf{keys}(l) : k \leq x, \quad \forall y \in \mathsf{keys}(r) : k \leq y$
  - fast vollständig balanciert:
    alle Blätter in Tiefe h oder (rechts davon) $h - 1$.

- diese Forderung ist diesmal *nicht zu streng*, denn es gibt genügend viele Repräsentationen einer festen Menge von Schlüsseln

  (Bsp: $\{1, 2, \ldots, 7\}$)

- Operationen in $O(\mathsf{height}(t)) = O(\log \# \mathsf{keys}(t))$ . . . oder noch weniger

Binäre Heaps (Implementierung)

- die fast vollständige Balance wird durch eine *implizite* Baumdarstellung erreicht:
  - Branch-Knoten stehen in einem Array $A[1 \ldots n]$
  - K_1 ist die Wurzel
  - $K_i = \text{Branch}(K_{2i}, A[i], K_{2i+1})$
  - für $i > n$: $K_i = \text{Leaf}$
- diese Form bleibt immer erhalten

  (evtl. ändern wir n um 1)
- die Operationen reparieren ggf. die Heap-Ordnung

  durch Vertauschung von Elementen.

Binäre Heaps: Einfügen, Verringern

- insert$(k, A[1 \ldots n])$:
  - füge neues Element an: $A[n+1] := k$,
  - repairere Heap-Ordnung (auf Pfad von $n+1$ zu Wurzel)
    - $i := n+1$, ($A[i]$ ist evtl. *zu klein*, wandert nach oben)
    - solange $i > 1$ und $A[\text{parent}(i)] > A[i]$:
      $A[\text{parent}(i)] \leftrightarrow A[i]; i := \text{parent}(i)$.
- Korrektheit (Invariante): $\forall j : j \neq i \Rightarrow A[\text{parent}(j)] \leq A[j]$.
- Laufzeit: $O(\text{height}(t)) = O(\log n)$.
- gleicher Algorithmus zum Reparieren nach decrease(i, v, A): $A_{\text{neu}}[i] := v \leq A_{\text{vorher}}[i]$

Binäre Heaps: Minimum bestimmen und entfernen

- Heap-Ordnung $\Rightarrow$ das Minimum steht immer in $A[1]$
- Minimum entfernen: $A[1] := A[n]$, $n := n - 1$,
  reparieren ($A[i]$ ist evtl. *zu groß*, wandert nach unten)
  - $i := 1$,
  - solange $A[i] > A[\text{left}(i)] \vee A[i] > A[\text{right}(i)]$:
    $A[i] \leftrightarrow \min(A[\text{left}(i)], A[\text{right}(i)])$
    $i :=$ vorige Position des Minimums
- Korrektheit (Invariante):
  $\forall j : (\text{parent}(j) \neq i) \Rightarrow A[\text{parent}(j)] \leq A[j]$
- Laufzeit: $O(\text{height}(t)) = O(\log n)$.

Binäre Heaps: Heap erzeugen

- gegeben: Folge (Multimenge) von Schlüsseln M
  gesucht: binärer Heap t mit keys$(t) = M$
- triviale Lösung: fortgesetztes Einfügen
  Laufzeit (mit $h = \log n$): $\sum_{k=1}^{h} 2^k \cdot k = \Theta(n \log n)$
  (welches ist der schlechteste Fall?)
- alternative Lösung:
  - Eingabe in $A[1 \ldots n]$ schreiben,
  - für i von $n/2$ bis 1: repariere „$A[i]$ ist zu groß“.

  Invariante: $\forall j : j > i \Rightarrow$ Teilbaum ab j ist heap-geordnet
  Laufzeit: $\sum_{k=1}^{h} 2^k \cdot (h - k) = \Theta(n)$
- NB: Unterschied Heap/Suchbaum: beweise, daß es nicht möglich ist, einen Suchbaum in $O(n)$ zu konstruieren.

Bemerkung zur Implementierung (I)

- wir hatten in erster Näherung diese Typen behauptet:
  - insert : $(K \times \mathbb{N}) \times \mathsf{PQ}(K) \to \mathsf{PQ}(K)$
  - decrease : $(K \times \mathbb{N}) \times \mathsf{PQ}(K) \to \mathsf{PQ}(K)$

- das geht gar nicht, denn um einen Schlüssel zu verringern, muß man wissen, wo er steht — man kann ihn aber nicht suchen (weil der Heap kein Suchbaum ist)

- Lösung: das Einfügen liefert einen *Verweis* in den Heap, der beim Verringern benutzt wird:
  - insert : $(K \times \mathbb{N}) \times \mathsf{PQ}(K) \to (\mathsf{Ref}(K) \times \mathsf{PQ}(K))$
  - decrease : $(\mathsf{Ref}(K) \times \mathbb{N}) \times \mathsf{PQ}(K) \to \mathsf{PQ}(K)$

Bemerkung zur Implementierung (I)

- diese Verweise muß man richtig verwalten,

  denn durch Operationen wie $A[\text{parent}(i)] \leftrightarrow A[i]$ werden Plätze getauscht.

- Lösung: ein zusätzliches Array R, Zähler t
  - bei jedem insert: $t := t + 1; R[t] = n + 1$
    (dort beginnt die Bahn des eingefügten Elements,
    t ist der Verweis-Rückgabewert des insert)
  - bei jedem $A[i] \leftrightarrow A[j]$: auch $R[i] \leftrightarrow R[j]$
  - decrease(t, v) bezieht sich dann auf $A[R[t]]$.

  Details, Beispiele: siehe [WW]

Anwendung binärer Heaps: Heapsort

- das ist ein Sortierverfahren:
  - Eingabe in $A[1 \ldots n]$ schreiben
  - Heap-Ordnung herstellen (Komplexität $\Theta(n)$)
  - fortgesetztes extractMin (Komplexität $\Theta(n \log n)$) liefert Elemente in aufsteigender Reihenfolge

- wenn man dafür Max-Heap (statt Min-Heap) benutzt, kann man Heap *und Ausgabe* im gleichen Array abspeichern

- ergibt asymptotisch optimales Sortierverfahren mit wenig (d.h. konstantem) zusätzlichen Platzbedarf

  (vgl. merge-sort)

Ausblick: schnellere Heaps

- hier gezeigte binäre Heaps realisieren
  insert, decrease, extractMin jeweils in $O(\log n)$
- das kann ein balancierter Suchbaum auch
- es gibt Heaps (z.B. Fibonacci heaps, Quake heaps)
  für die decrease schneller geht (in $O(1)$)
  und die anderen Op. immer noch in $O(\log n)$.
  (decrease ist häufigste Operation im später betrachteten Algorithmus von Dijkstra zum Bestimmen kürzester Wege)
- dafür sind fortgeschrittene Analyse-Techniken nötig
- T. M. Chan: *Quake Heaps*, 2013, `http://link.springer.com/chapter/10.1007/978-3-642-40273-9_3`

Übungsserie 10 für KW 24

Aufgabe 10.A (autotool)

- insert, decrease, extractMin für binäre Heaps

Aufgabe 10.1 (Binäre Heaps)

- (3 P) Repräsentiert das Array einen Heap? Wenn nein, repräsentiert es einen Zwischenzustand bei der Ausführung von insert? von extractMin?
  - $[2, 11, 3, 7, 6, 4, 8, 12, 10]$
  - $[2, 5, 3, 8, 6, 7, 4]$
  - $[1, 6, 3, 7, 5, 2, 8, 9]$

- (2 P) Eine Folge von Operationen, die dann vom Übungsleiter vorgegeben wird, an der Tafel vorrechnen (ohne Hilfsmittel).

  Beispiele: Für den durch das Array $[1, 2, \ldots, 7]$ implizit repräsentierten Heap:

– solange extractMin, bis der Heap leer ist.
– für k von 1 bis 7: die Priorität von k auf $-k$ verringern.

Aufgabe 10.2 (Kombination Suchbaum und Heap)

Wir betrachten binäre Bäume, deren Schlüssel Zahlenpaare sind, so daß für die ersten Komponenten die Heap-Eigenschaft und für die zweiten Komponenten die Suchbaum-Eigenschaft gilt. (Es wird keine Balance-Eigenschaft gefordert.)

- (1 P) Geben Sie einen solchen Baum t mit folgender Schlüsselmenge an:

$$\{(5,1),(4,3),(7,4),(3,0),(1,5),(6,7)\}$$

- (2 P) Beweisen Sie: für jede Schlüsselmenge $M \subseteq \mathbb{N} \times \mathbb{N}$, deren ersten Komponenten paarweise verschieden sind

und deren zweiten Komponenten paarweise verschieden sind, gibt es genau solchen Baum t mit keys$(t) = M$.

Hinweis: Induktion nach der Größe von M.

- (2 P) Geben Sie ein Verfahren zum Einfügen an.

  Hinweis: Rotationen. Fügen Sie $(2, 2)$ in t ein.

Aufgabe 10.3 (Young-Tableaux)

(4 P) [CLRS] Aufgabe 6-3 a,c,d,e.

Direkter Zugriff (Hashing und Anwend.)

Übersicht

[WW] 9, [CLRS] 11

- Counting-Sort,
- Hash-Tabellen
- Radix-Sort, Tries

Motivation

- Aufgabe: sortiere eine Folge von 1.000.000 Zahlen.

  Lösung: Merge-, Heap-, Quick-, Bubble-Sort usw.

- Aufgabe: sortiere eine Folge von 1.000.000 natürlichen Zahlen aus dem Intervall $[0 \ldots 1000]$

  Lösung: . . .

- Hinweis:

  Aufgabe: sortiere eine Folge von 1.000.000 Bits

Sortieren durch Zählen

- Aufgabe: sortiere eine Folge a von n natürlichen Zahlen aus dem Intervall $[0 \ldots B]$
- Lösung: benutze Array $c[0 \ldots B]$, initialisiert mit $[0, \ldots, 0]$
  - für i von 1 bis n: $\{\ k := a[i]; c[k] := c[k] + 1\ \}$
    Invariante: $\forall k : c[k] = \#\{j \mid 1 \leq j < i \wedge a[j] = k\}$
  - für k von 0 bis B: drucke $c[k]$ mal k
- Laufzeit ist $n + B$, also für festes B: Sortieren in $O(n)$?
- Ja. Das ist kein Widerspruch zu Schranke $\Omega(n \log n)$, denn diese gilt nur für *vergleichsbasierte* Verfahren
- Algorithmus „Sortieren durch Zählen“ (counting sort) benutzt *direkten Zugriff* (Zahlenwerte der Eingabe)

Hashing zur Implementierung des ADT Menge

- Spezifikation:
  - empty, contains, insert : $K \times \text{Set}(K) \to \text{Set}(K)$, delete
  - Semantik: *dynamisch* (ephemer, mutable)
    Operationen zerstören vorige Werte
- Plan zur Implementierung:
  - benutze Array $a[0..m-1]$ von Wahrheitswerten
  - Funktion (hash) $h : K \to [0..m-1]$
  - contains$(x, S) \iff a[h(x)]$
- dazu müssen diese Fragen geklärt werden:
  - welcher Wert für m (Kapazität)?
  - welche Funktion h?
  - was passiert bei Kollision $x \neq y \wedge h(x) = h(y)$?

Hashfunktionen (1)

- Ziel: Funktion (hash) $h : K \to [0..m-1]$
- Realisierung in zwei Schritten:
  - von K zu $\mathbb{N}$
  - von $\mathbb{N}$ zu $[0..m-1]$
- Schritt 1 ist grundsätzlich deswegen möglich, weil alle Daten binär im (Haupt)speicher stehen und jede Bitfolge eine natürliche Zahl beschreibt.
- wir nehmen deswegen an, daß $K = \mathbb{N}$

Hashfunktionen (2)

- Ziel $h : \mathbb{N} \to [0..m-1]$
  - h soll Wertebereich möglichst gleichmäßig ausnutzen
  - Wert $h(x)$ sollte von allen Bits von x abhängen
- Divisionsmethode: $h(x) = \mathsf{mod}(x, m)$
  - m sollte nicht von der Form 2^e sein
  - Ü: m sollte nicht $2^e - 1$ sein
- Multiplikationsmethode: $h(x) = \lfloor m \cdot \mathsf{mod}(q \cdot x, 1) \rfloor$
  - für eine irrationale Zahl q, z.B. $(\sqrt{5}+1)/2$
  - dabei bedeutet $\mathsf{mod}(\cdot, 1) : \mathbb{R} \to \mathbb{R} : z \mapsto z - \lfloor z \rfloor$
  - Ü: kann mit Ganzzahl-Arithmetik implementiert werden, benutze $q \approx Q/2^e$ für passendes $Q \in \mathbb{N}$ und $m = 2^e$.

Hashing mit Verkettung (Chaining)

- das *Kollisions-Problem*: falls $x \neq y \wedge h(x) = h(y)$:
  $S_0 = \text{empty}; S_1 = \text{insert}(x, S_0); \text{contains}(y, S_1) = ?$
- Lösung (Plan)
  - Element-Typ des Arrays ist nicht $\mathbb{B}$, sondern $\text{Set}(K)$
  - meist repräsentiert durch einfach verkettet Liste
- Lösung (Realisierung)
  - $\text{contains}(x, S) = \text{contains}(x, a[h(x)])$.
  - $\text{insert}(x, S) : a[h(x)] := \text{insert}(x, a[h(x)])$
  - $\text{delete}(x, S)$ entsprechend
- Laufzeit: $1 +$ ((maximale) Größe der Mengen $h(x)$)
  bei gleichmäßigem Hashing: $1 + \text{size}(S)/m = 1 + \alpha$

Hashing mit offener Adressierung

- Motivation: Vermeidung des Aufwandes (Zeit und Platz) für die Verwaltung der Mengen (z.B. Zeiger für Listen)
- Tabellen-Einträge sind $K \cup \{\perp\}$, bei Konflikt zw. x und y wird y an einer anderen Stelle *in* der Tabelle gespeichert.
- Beispiel (lineares Sondieren)
  - insert(x, S): für p von $h(x)$ bis mod$(h(x) + m - 1, m)$:
    falls $a[p] = \perp$, dann $a[p] := x$ und fertig,
    falls $a[p] = x$, dann fertig
  - contains(x, S): für p von $h(x)$ bis mod$(h(x) + m - 1, m)$:
    falls $a[p] = \perp$, dann false
    falls $a[p] = x$, dann true
  - delete(x, S)? Vorsicht!

Offene Adressierung und Delete

- naives Löschverfahren $a[h(x)] = \perp$ ist nicht korrekt
- gelöschte Einträge müssen gesondert markiert werden
- die so markierten Stellen können beim Einfügen wieder benutzt werden

Lineares Sondieren und Haufen (Clustering)

- Laufzeit von contains und insert ist
  $O($(maximale) Länge eines belegten Abschnitts$)$
- selbst bei gleichmäßiger Hashfunktion
  wächst diese Größe stärker als size$(S)/m$,
  denn die Wahrscheinlichkeit dafür, daß ein neuer Eintrag in ein Intervall I fällt, ist $|I|/m$.
  $\Rightarrow$ es ist zu erwarten, daß große Haufen (cluster) noch größer werden.
- zum Vergleich: bei Chaining: die Wsk, daß ein neuer Eintrag in eine bestimme Menge fällt, ist $1/m$.
  $\Rightarrow$ es ist zu erwarten, daß alle Mengen ähnlich groß sind.

Offene Adr. mit doppeltem Hashing

- Plan: bei Kollision von x mit $y \Rightarrow$ *verschiedene* Sondierungs-Folgen für x und $y \Rightarrow$ keine Haufen-Bildung

- Realisierung:
  - (linear: für p in $[h(x), h(x)+1, h(x)+2, \ldots]$ ...)
  - doppeltes Hashing:
    benutze zwei Hashfunktionen h_1, h_2 und
    für p in $[h_1(x), h_1(x)+h_2(x), h_1(x)+2h_2(x), \ldots]$ ...

- damit durch mod$(h_1(x) + i \cdot h_2(x), m)$ alle Werte in $[0..m-1]$ erreicht werden, muß $\gcd(h_2(x), m) = 1$ sein.

  zum Beispiel: m prim und $0 < h_2(x) < m$

  oder $m = 2^e$ und $h_2(x)$ ungerade

Erwartete Laufzeit f. doppeltes Hashing

- unter der Annahme, daß alle Werte von $\mathrm{mod}(h_1(x) + i \cdot h_2(x), m)$ gleich wahrscheinlich sind,

  bestimmen wir die erwartete Laufzeit für eine erfolglose Suche (oder das Einfügen eines neuen Elementes)

- Kollision für $i = 0$ mit Wsk $1/\alpha$

  unter dieser Voraussetzung:
  Kollision für $i = 1$ mit Wsk. $1/\alpha$, gesamt Wsk.: $1/\alpha^2$

  erwartete Schrittzahl insg. $\leq \sum_{k \geq 0} (1/\alpha)^k = 1/(1-\alpha)$

- zur Erinnerung: Chaining: $1 + \alpha$.

  Vergleiche Zahlenwerte für $\alpha = 1/2, \alpha = 2/3$

Eine Variante des doppelten Hashing

- insert(x, S), beim Sondieren mit $p = h_1(x) + i \cdot h_2(x)$:

  falls $a[p] = y$ mit $y \neq \perp$ und $y \neq x$:

  – bisher: bestimme nächste Sondierungs-Adresse für x
  – alternativ (Brent, 1973)
    * bestimme Ersatz-Adresse $q = p + h_2(y)$ für y,
    * falls $a[q]$ frei ist: $a[p] := x; a[q] := y$
    * sonst weiter sondieren für x.

- vgl. [WW] Algorithmus 9.10 oder [OW] Abschnitt 4.3.4

- Erweiterung dieser Idee: Kuckucks-Hashing (evtl. Ü)

Amortisierte Komplexität

Übersicht

[CLRS] Kapitel 17.4

- Motivation
- Definition
- Beispiel: Vergrößerung von Hashtabellen
- Beispiel: Quake Heaps

Motivation

- wie wählt man die Größe einer Hashtabelle?
  - Größe muß vor erster Operation fixiert werden, kann danach nicht geändert werden
  - zu klein $\Rightarrow$ hoher Belegungs-Faktor α, hohe Laufzeiten (viele Kollisionen), Überlauf (keine freien Plätze)
  - zu groß $\Rightarrow$ Speicher verschenkt (viele freie Plätze)
- anzustreben ist $\alpha = 1/2$ (oder ähnlich, je nach Verfahren)
- $\#\,\mathsf{keys}(S)$ vorher unbekannt, α wird überschritten:
  re-hashing (alle Elemente in eine Tabelle doppelter Größe übertragen, mit neuer Hashfunktion)
- Komplexität: 1. das ist teuer, 2. das geschieht selten

Def. und Bsp.: Amortisierte Komplexität

- Ziel: Beschreibung der Gesamtkosten einer Folge von Operationen im schlechtesten Fall.
- Def: Operationen $o \in O$ mit tatsächlichen Kosten $c(o)$ haben amortisierte Kosten $c_a(o)$,
  falls $\forall s \in O^* : \sum_{o \in s} c(o) \leq \sum_{o \in s} c_a(o)$
- Bsp: die *amortisierten* Kosten für insert mit re-hashing sind $O(1)$ für jede Folge, die mit leerer Tabelle beginnt.
- *tatsächliche* Kosten für insert (re-hash bei $\alpha > 1/2$)
  $1, 2, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 17, 1, \ldots$
  die Partialsummen p_i dieser Folge:
  $1, 3, 6, 7, 12, 13, 14, 15, 24, 25, 26, \ldots, 31, 48, 49, \ldots$
  es gilt $p_i \leq 3 \cdot i$ (d.h.: kleiner als die P.S. von $3, 3, 3, \ldots$)

Die Potential-Methode

- Potentialfunktion P : Datenstrukur $\to \mathbb{N}$

  Bsp: $P(T) = 2 \cdot$ Anzahl der insert seit letztem re-hash

- amortisierte Kosten von Operation o (bzgl. P):

  $c_a(o) = c(o) + P(\text{nach } o) - P(\text{vor } o)$

- denn $\sum_{o \in s} c_a(o) = \sum_{o \in s} c(o) + P(\text{nach } s) - P(\text{vor } s)$

  wenn $P(\text{Anfang}) = 0$, dann $\sum_{o \in s} c(o) \leq \sum_{o \in s} c_a(o)$

- Bsp: Hashing. $P(\text{leer}) = 0$ und
  - $c_a(\text{insert})$ ohne re-hash: $1 + 2(e+1) - 2e = 3$
  - $c_a(\text{insert})$ mit re-hash: $(2e+1) + 2 \cdot 1 - 2e = 3$

Amortisierte Analyse von Quake Heaps

- Ziel: Prioritätswarteschlange mit amortisierten Kosten
  insert : $O(1)$, decrease : $O(1)$, extractMin : $O(\log n)$
- Realisierung:
  - Liste von vollständig höhen-balancierten Bäumen mit binären und unären Knoten
  - tatsächliche Kosten:
    insert : $O(1)$, decrease : $O(1)$, extractMin : $O(n)$
  - Beweis der amortisierten Kosten durch geeignetes Potential (im wesentlichen: Anzahl der unären Knoten)
- T. M. Chan: *Quake Heaps*, 2013, `http://link.springer.com/chapter/10.1007/978-3-642-40273-9_3`

  W. Mulzer: *Erdbebenhaufen*, `https://page.mi.fu-berlin.de/mulzer/notes/ha/quake.pdf`

Strukturelle Invarianten von Quake Heaps

- Q-Heap ist Liste von Bäumen mit Eigenschaften:
  - innere Knoten binär oder unär, Schlüssel in Blättern
  - alle Blätter gleich tief
  - jeder inn. Knoten i hat Verweis auf kleinstes Blatt $m(i)$
  - jedes Blatt b hat Verw. $h(b)$ auf höchstes i mit $m(i) = b$
- insert(x, S) in $O(1)$:

  ein neuer Baum (Blatt) mit Schlüssel x der Höhe 0
- decrease(x, v, S) in $O(1)$ (mit x in Blatt b):
  - Eintrag in b wird auf v verringert
  - falls $h(b)$ keine Wurzel ist: $h(b)$ wird aus seinem Vorgänger entfernt (dieser ist binär, wird unär) dadurch entsteht neuer Baum mit $h(b)$ als Wurzel

extractMin für Quake Heaps

- extractMin(S):
  - bestimme Wurzel w, für die $m(w)$ minimal ist
    Kosten: $O($Anzahl Bäume$)$
  - lösche Knoten auf Pfad von w zu $m(w)$,
    dadurch entstehen neue Bäume,
    Kosten: $O($Höhe$)$.
  - Operat. zum Reduzieren der Baum-Anzahl u. -Höhe
- Reduktion der Anzahl: solange es zwei Bäume gleicher Höhe h gibt: fasse diese zu einem Baum (mit binärer Wurzel) der Höhe $h+1$ zusammen

  Kosten: $O($Anzahl Bäume vorher$)$,

  Resultat: $O($Höhe $+ \log n)$ Bäume

Logarithmische Höhe und Erdbeben

- mit $n_i =$ Anzahl der Knoten aller Bäume in Höhe i
  ($n_0 =$ Anzahl der Blätter $=$ Anzahl der Schlüssel)
  soll immer gelten: $\forall i : n_{i+1} \leq 3/4 \cdot n_i$.
- $\Rightarrow$ die Höhe jedes Baumes ist dann $\leq \log_{4/3} n \in O(\log n)$
- „Erdbeben“ (quake): falls $\exists i : n_{i+1} > 3/4 \cdot n_i$,
  dann $i_0 :=$ das kleinste solche i,
  lösche alle Knoten der Höhe $> i_0$.
  Kosten: Anzahl dieser Knoten $(= D)$
- insgesamt: $c(\text{insert}) = O(\log n +$ Anzahl Bäume $+ D)$
  was ist die passende Potentialfunktion, für die
  $c_a(\text{insert}) = O(\log n)$ und $c_a(\text{insert}), c_a(\text{decrease}) \in O(1)$?

Potential für Quake Heaps

- $P(H) = K + 2 \cdot B + 4 \cdot U$ mit
  - $B =$ Anzahl der Bäume,
  - $K =$ Anzahl aller Knoten, $U = \ldots$ der unären Knoten

- $c_a(o)$ so festlegen, daß $c_a(o) \geq c(o) + \Delta(P)$ für
  - insert: $c = 1$, $\Delta(K) = 1, \Delta(B) = 1, \Delta(U) = 0$,$c_a = 4$
  - decrease: $\ldots$
  - extractMin: $c = O(\log n + B_0 + D)$, $c_a = O(\log n)$
    - Pfad löschen: $\Delta(K + B) < 0, \Delta(U) \leq 0$
    - Bäume zusammenfassen:
      $\Delta(B) = \log n - B_0, \Delta(K) = -\Delta(B), \Delta(U) = 0$
    - Erdbeben: $\Delta(K) \leq 0$, $\Delta(B) \leq n_i$, $\Delta(U) \leq -n_i/2$

Übungsserie 11 für KW 25

Aufgabe 11.A (autotool: Hashing)

- Hashing mit Verkettung
- Hashing mit linearem Sondieren
- doppeltes Hashing

Aufgabe 11.1 (Hashing)

Für

- (1 P) Hashing mit linearem Sondieren,
- (1 P) doppeltes Hashing,
- (2 P) doppeltes Hashing in der Variante von Brent (siehe Skript bzw. dort angegebene Literatur)

den Algorithmus an einem Beispiel an der Tafel vorführen.

Aufgabe 11.2 (Kuckucks-Hashing)

engl. *cuckoo hashing*

- (1 P) Zitieren Sie die wissenschaftliche Primär-Quelle (die referierte Publikation der Erfinder) für diesen Algorithmus (Autor, Titel, Jahr, Ort (Konferenz))

- (1 P) Geben Sie die Invariante des Verfahrens an.

  Welche Laufzeit der Operationen (contains, insert) ergibt sich daraus?

  Wo ist der Laufzeit-Unterschied zu den in der VL betrachteten Verfahren?

- (2 P) Führen Sie den Algorithmus an einem Beispiel an der Tafel vor.

Aufgabe 11.3 (Amortisierte Analyse)

Wiederholung (1. Semester):

- Schlange und Keller sind Listen mit beschränktem Zugriff:

- Ein Keller (stack) hat die Operationen push (Einfügen) und pop (Extrahieren, beides am linken Ende).

- Die Warteschlange (queue) (ohne Prioritäten) hat die Operationen enqueue (Einfügen am rechten Ende) und dequeue (Extrahieren am linken Ende)

Die Schlange S soll durch zwei Keller L, R implementiert werden.

- Invariante: der Inhalt von s ist append$(L, \text{reverse}(R))$.

- leere Schlange: $L = R =$ leerer Keller.
- enqueue$(x, S) =$ push(x, R)
- dequeue$(S) =$
  1. falls L leer, dann: solange R nicht leer: push(pop$(R), L)$
  2. pop(L)

Aufgaben:

- (1 P) Führen Sie
  $S_1 =$ enqueue$(3, S_0), S_2 =$ enqueue$(4, S_1), S_3 =$
  dequeue$(S_2), S_4 =$ enqueue$(5, S_3), S_5 =$ dequeue(S_4) vor.
- (1 P) welches sind die tatsächlichen Kosten von enqueue und dequeue (als Funktion von $|L|$ und $|R|$)? Die Kosten von push und pop sind 1.

- (2 P) Geben Sie eine Potentialfunktion P an, bezüglich derer die amortisierten Kosten von enqueue und dequeue $O(1)$ sind.

  Geben sie $P(S_0), P(S_1), \ldots$ für die o.g. Operationsfolge an.

Aufgabe 11.4 (Zusatz) (Quake Heaps)

- (2 P) beschreiben Sie das Verhalten von QH für: Einfügen mehrerer Schlüssel, dann wiederholtes extractMin, bis der Heap leer ist.

  Welche Komplexität hat dieses Sortierverfahren?

  Geben Sie den Verlauf des Potentials an.

- (2 P) Geben Sie eine Operationsfolge an, die mit leerem Heap beginnt und ein Erdbeben enthält.

  Geben Sie den Verlauf des Potentials an.

- (2 P) Das Potential im Skript (übernommen aus zit. Skript von Mulzer) ist anders als das im Artikel von Chan. Ist der Unterschied wesentlich?

Graphen: Eigenschaften, einfache Algorithmen

Überblick

Andreas Brandstädt: *Graphen und Algorithmen*, Teubner 1994, https://link.springer.com/book/10.1007/978-3-322-94689-8

Hansjoachim Walther, Günther Nägler: *Graphen — Algorithmen — Programme*, Fachbuchverlag Leipzig 1987

- Bezeichnungen, Eigenschaften
- Datenstrukturen zur Repräsentation von Graphen
- Tiefensuche, Breitensuche
- topologisches Sortieren, Zusammenhangskomponenten

Motivation, Definition, Beispiele

- $G = (V, E)$, (ungerichtet) $E \subseteq \binom{V}{2}$, (gerichtet) $E \subseteq V^2$
- Graph $=$ zweistellige Relation (ungerichtet: symmetrisch)
- Relationen (und damit auch Graphen) sind fundamental für Prädikatenlogik, Modellierung, Informatik.
- Graphen, mit denen derzeit viel Geld verdient wird:
  - das WWW (Knoten: URLs; Kanten: Verweise)
  - das sogenannte soziale Netzwerk
    (Knoten: Personen, genauer: Werbe-Ziele;
    Kanten: sogenannte Freundschaften)
- Graphentheorie: Eulerkreise (1736), Landkartenfärbung (Kempe 1879) ...
- Graph-Algorithmen: Anwendung von Datenstrukturen

Spezielle Graphen, Graph-Operationen

- Spezielle Graphen:
  - unabhängiger Graph I_n (n Knoten, keine Kanten)
  - vollständiger Graph (Clique) K_n (n Knoten, alle Kanten)
  - Pfad P_n (n Knoten, $n-1$ Kanten)
  - für $n \geq 3$: Kreis C_n (n Knoten, n Kanten)
- Operationen:
  - Komplement $c(V,E) = \overline{(V,E)} = (V, \binom{V}{2} \setminus E)$
  - disjunkte Summe $(V_1,E_1) \cup (V_2,E_2) = (V_1 \cup V_2, E_1 \cup E_2)$
  - Verkettung (join) $G_1 + G_2 = c(c(G_1) \cup c(G_2))$
- weitere spezielle Graphen:
  - vollständig bipartit $K_{a,b} = I_a + I_b$,
  - Stern $K_{1,b}$, Rad $K_1 + C_n$

Eigenschaften von Graphen

- Knotenzahl: $|V(G)|$, n; Kantenzahl: $|E(G)|$, m
- G ungerichtet: Nachbarn $G(u) := \{v \mid \{u, v\} \in E(G)\}$
  mit verkürzter Notation für Kanten: $\{v \mid uv \in E(G)\}$
- G gerichtet: Nachfolger $G(u) := \{v \mid (u, v) \in E(G)\}$
  Vorgänger: $G^-(u)$, mit Spiegelbild $G^- := (V, E^-)$.
- Grad eines Knotens $\deg_G(u) = \#G(u)$
  Minimalgrad: $\delta(G) = \min\{\deg_G(u) \mid u \in V\}$
  Maximalgrad: $\Delta(G) = \max\{\deg_G(u) \mid u \in V\}$
- G ist r-regulär, falls $\forall u \in V : \deg_G(u) = r$
- Bsp: Petersen-Graph $(\binom{5}{2}, \{uv \mid u \cap v = \emptyset\})$ ist 3-regulär.
  (Das kann man ohne Zeichnung nachprüfen.)

Verbindungen, Abstände

- $u \sim_G v$: u und v sind *verbunden* in G,

  falls es in G einen Pfad $u \to \ldots \to v$ gibt
  - G ungerichtet: $\sim_G$ ist eine Äquivalenzrelation auf V.
  - Die Äquivalenzklassen $V/\sim_G$ heißen *Zusammenhangskomponenten*
  - G heißt zusammenhängend, wenn $|V/\sim_G| = 1$

- $\mathrm{dist}_G(u, v) :=$ die Länge (d.h. Anzahl der Kanten)

  eines kürzesten Pfades in G von u nach v.

  ($+\infty$, falls es keinen solchen gibt)
  - diese Def. gilt für ungerichtete und für gerichtete G
  - Durchmesser $\mathrm{diam}(G) := \max_{u,v \in V} \mathrm{dist}_G(u, v)$
  - Radius $\mathrm{rad}(G) := \min_{u \in V} \max_{v \in V} \mathrm{dist}_G(u, v)$

Die Größe spezieller Teilgraphen

- Unabhängigkeitszahl $\alpha(G)$:

  die maximale Größe der unabhängigen Teilmengen in G

- Cliquenzahl $\omega(G)$:

  die maximale Größe der vollständigen Teilmengen in G

- Taillenweite (girth) $g(G)$:

  die minimale Größe der Kreise in G.

- Übungsaufgaben: bestimme alle im Skript gennanten Parameter für alle im Skript genannten Graphen.

  $g(\text{Petersen}), \chi(\text{Petersen}), \omega(K_{3,3}), \mathrm{rad}(C_5), \alpha(C_7), \ldots$

Repräsentationen von Graphen

- Annahme: direkter Zugriff für Knoten: $V = [1, 2, \ldots, n]$
- Aufgabe: (effiziente) Repräsentation von E
  - schneller Zugriff: Adjazenz-Matrix
    $a : V^2 \to \mathbb{B}$ mit $uv \in E \iff a[u, v] =$ true
  - wenig Platz: Adjanzenz-Listen:
    für jeden $u \in V$: die Menge $G(u)$ repräsentiert als (einfach verkettete) Liste
- wir verwenden grundsätzlich die AL-Darstellung,

  wenigstens zur Repräsentation der Eingabe von Graphalgorithmen

  (intern könnten diese andere Datenstrukturen benutzen)

Das Durchlaufen von Graphen (Schema)

- – Eingabe: Graph $G = (V, E)$
  – Ausgabe: Folge von Knoten $s \in V^*$, Relation p auf V
- done $:= \emptyset$; todo $:= \{v_0\}$; $p[v_0] = \perp$; while todo $\neq \emptyset$
  wähle $v \in$ todo (z.B. den ältesten oder den neuesten)
  todo $:=$ todo $\setminus \{v\}$; done $:=$ done $\cup \{v\}$
  für alle $u \in G(v) \setminus$ done : todo $:=$ todo $\cup \{u\}$; $p[u] := v$
- $v \in$ done $\iff$ $c[v] =$ schwarz, (todo: grau, sonst: weiß)
- Invarianten/Eigenschaften:
  – p ist die Vorgänger-Relation eines Baumes
  – jedes $x \in$ todo $\cup$ done ist von v_0 aus erreichbar
  durch Weg $P(x) : v_0 \to \ldots \to p[p[x]] \to p[x] \to x$
  – jeder Nachbar von $x \in$ done ist schwarz oder grau
  – schließlich: done $=$ die von v_0 erreichbaren Knoten,

Breitensuche (breadth first search, BFS)

- implementiere todo als Warteschlange (queue):
  - Extrahieren von v am linken Ende,
  - Einfügen aller $u \in G(v)$ am rechten Ende.
- Bezeichnung $d(x) := |P(x)|$ oder $+\infty$, falls x weiß

  Satz: $\forall x : d(x) = \mathrm{dist}_G(v_0, x)$
- Beweis:

  Inv.: $\mathrm{dist}_G(v_0, x) \leq d(x)$, weil $P(x)$ ein Weg $v_0 \to^* x$ ist.

  Inv.: todo $= [x_1, \ldots, x_k] \Rightarrow d(x_1) \leq \ldots \leq d(x_k) \leq d(x_1) + 1$.

  Zeige Satz durch Induktion über $\mathrm{dist}_G(v_0, x)$
- Laufzeit von BFS(G) ist $O(n + m) = O(|V| + |E|)$.

  Beweis: für jeden Knoten und für jede Kante $O(1)$.

Tiefensuche (depth first search, DFS)

- implementiere todo als Keller (stack):
  - Extrahieren und Einfügen am linken Ende
- alternative Formulierung mit Rekursion: $\mathrm{DFS}_1(x) :=$

  $c[x] :=$ grau;
  for $y \in G(x)$: **if** $y \in$ weiß **then** $\{p[y] := x;\ \mathrm{DFS}_1(y);\}$
  $c[x] :=$ schwarz

- Erweiterung gegenüber Schema: besuche *alle* Knoten

  $\mathrm{DFS}(G)$: while (weiß $\neq \emptyset$) { wähle $v_0 \in$ weiß; $\mathrm{DFS}_1(v_0)$ }
- $\mathrm{DFS}(G) = (V, \{(p[x], x) \mid x \in V, p[x] \neq \perp\})$
  ist der DFS-Wald von G
  (eine disjunkte Vereinigung von Bäumen)

Bezeichnungen für DFS-Bäume und -Ordnung

- ordne $F = \mathsf{DFS}(G)$ durch zeitliche Reihenfolge der Bäume sowie in jedem Knoten zeitliche R. der Kinder

  $F = [t_1, \ldots, t_k]$ mit $t_i \in (V)$

  $\mathsf{Tree}(V) = \{\mathsf{Branch}(v, [t_1, \ldots, t_k]) \mid v \in V, t_i \in \mathsf{Tree}(V)\}$

- die DFS-Reihenfolge von G

  ist die Preorder-Reihenfolge von F, Notation $<_{\mathsf{pre}(F)}$

  (Reihenfolge der Entdeckung $=$ der Graufärbung)

- in Anwendungen wird auch die Postorder-Reihenfolge von F benutzt, Notation $<_{\mathsf{post}(F)}$

  (Reihenfolge der Erledigung $=$ der Schwarzfärbung)

DFS-Klassifikation der Kanten

- Def.: Mit $F = \mathrm{DFS}(G)$: $(x, y) \in E(G)$ heißt
  - Baum-Kante, falls $x \to_F y$
  - Rückwärts-Kante, falls $y \to_F^* x$
  - Vorwärts-Kante, falls $x \to_F^{\geq 2} y$
  - Quer-Kante, sonst.

- Satz: Mit $F = \mathrm{DFS}(G)$ gilt: $(x, y) \in E(G)$ ist
  - Baum- oder Vorwärtskante, falls $x <_{\mathrm{pre}(F)} y \wedge y <_{\mathrm{post}(F)} x$
  - Rückwärtskante, falls $y <_{\mathrm{pre}(F)} x \wedge x <_{\mathrm{post}(F)} y$
  - Querkante, falls $y <_{\mathrm{pre}(F)} x \wedge y <_{\mathrm{post}(F)} x$

  (Der Fall $x <_{\mathrm{pre}(F)} y \wedge x <_{\mathrm{post}(F)} y$ kommt nicht vor.)

DFS für ungerichtete Graphen

- Bezeichnungen:

  $\{u, v\}$ ist ungerichtete Baum-, Rückwärts-, Querkante
  $\Longleftrightarrow$ (u, v) oder (v, u) ist Baum-, . . . -kante.

- Satz: für ungerichtete Graphen $G = (V, E)$:

  jedes $e \in E$ ist Baum-Kante oder Rückwärts-Kante

  bezüglich $F = \mathsf{DFS}(G)$.

  (d.h., es gibt keine Querkanten)

- Beweis: falls xy Querkante, (x, y) gerichtete Querkante,

  dann nach Def. $y <_{\mathsf{post}(F)} x$,

  aber Kante (y, x) verlangt $x <_{\mathsf{post}(F)} y$, Widerspruch.

Gerichtete kreisfreie Graphen (DAG)

- Bezeichnung: DAG (directed acyclic graph):

  gerichteter Graph ohne gerichteten Kreis

- Satz: Für jeden gerichteten Graphen G:

  G ist DAG $\iff$ DFS(G) erzeugt keine Rückwärtskanten

- Beweis:

  „$\Rightarrow$“: (indirekt)

  falls (x, y) Rückwärtskante, dann Kreis $y \rightarrow_F^* x \rightarrow y$

  „$\Leftarrow$“: (indirekt) falls Kreis C in G,
  $y \in V(C)$ mit y minimal bzgl. $<_{\mathsf{pre}(F)}$ und $(x, y) \in E(C)$.
  Dann $y \rightarrow_T^* x$ und (x, y) wird Rückwärtskante

Topologisches Ordnen

- Def: eine totale Ordnung $(V, \leq)$ heißt *topologische Ordnung* für gerichteten Graph $G = (V, E)$, falls $\forall (x, y) \in E : x < y$.

- Anwendung: V: Aufgaben, E: Abhängigkeiten, $(<)$: Ausführungsreihenfolge

- Satz: G besitzt topologische Ordnung $\iff$ G ist DAG.

- Beweis:

  „$\Rightarrow$“: (indirekt) betrachte Kreis C in G und $(V(C), \leq)$

  „$\Leftarrow$“: (direkt) mit $F = \mathrm{DFS}(G)$ ist $>_{\mathrm{post}(F)}$ eine topologische Ordnung.

Minimal-Gerüste

Überblick

- Def. Minimalgerüst,
- Algorithmus von Kruskal
- ADT Mengensystem
- Implementierung Mengensystem durch disjoint-set-forest

Definition Minimalgerüst

- Def. Ein Baum T heißt *Gerüst* von G, falls $V(T) = V(G)$.
- Satz: G ist zusammenhängend $\iff$ G besitzt Gerüst.
- Def. ein (kanten-)gewichteter Graph

  ist ein Paar (G, w) mit $w : E(G) \to \mathbb{R}$
- Def. für $H \subseteq G$: $w(H) = \sum\{w(e) \mid e \in E(H)\}$
- Def. T ist ein Minimalgerüst (MST) für (G, w),

  falls $w(T) = \min\{w(T') \mid T' \text{ ist Gerüst für } G\}$
- Motivation: Verbindungen von Bestandteilen technischer Systeme (Rohrleitungen, Straßen, Leiterbahnen,...)

Algorithmus (Schema)

- schrittweise Konstruktion eines MST durch Hinzufügen von *sicheren* Kanten (die die Invariante nicht verletzen)

- // Invariante: $\exists T : (V, A) \subseteq T$ und T ist MST für G
  $A := \emptyset$; while A ist kein Gerüst
  wähle sichere Kante $e \in E(G)$; $A := A \cup \{e\}$

- Satz: A erfüllt Invariante,
  C eine Zsgh-Komponenten von (V, A),
  e eine leichteste Kante zw. C und $V \setminus C$.
  Dann ist e sicher.

  Beweis: in T gibt es eine Kante f zw. C und $V \setminus C$.

  Dann ist $T \setminus \{f\} \cup \{e\}$ auch ein MST für G.

Algorithmus (Eigenschaften, Realisierungen)

- in jedem Fall gilt
  - Anfang: (V, A) hat $|V|$ Komponenten der Größe 1
  - Schluß: (V, A) ist eine Komponente der Größe $|V|$.

- Algorithmus von Kruskal:

  C: eine beliebige Komponente

- Algorithmus von Prim:

  zusätzliche Invariante: Kanten von A bilden einen Baum.

  C: die Knotenmenge dieses Baums

Algorithmus von Kruskal (1956)

- Plan: $e :=$ eine leichteste Kante, die eine beliebige Komponente C von A mit $V \setminus C$ verbindet

- Realisierung:
  - $E' :=$ sortiere E nach aufsteigendem Gewicht
  - für xy aus E': falls $x \not\sim_A y$, dann $A := A \cup \{xy\}$.

- Kosten:
  - $\Theta(m \log m)$ für Sortieren von E
  - m mal feststellen, ob $x \sim_A y$
    für einen solchen Test haben wir $O(\log n)$ Zeit,
    ohne die Laufzeit asymptotisch zu erhöhen.
    (beachte: $n - 1 \leq m$ wg. Zusammenhang von G)

Union-Find-Strukturen

- [OW] Kapitel 6.2, [CLR] Kapitel 21
- abstrakter Datentyp zur dynamischen Repräsentation von Äquivalenzklassen einer Relation R
  - Union(x, y): $R := R \cup \{(x, y)\}$
  - equiv(x, y): gilt $x \sim_R y$ mit $(\sim_R) = (R \cup R^-)^*$?
- Realisierung mittels
  - find(x): *kanonischer Repräsentant* von $[x]_{\sim_R}$, so daß $\forall x, y$: equiv$(x, y) \iff$ find$(x) =$ find(y)
- Beispiel für kanonische Repräsentation:
  - Menge $\mathbb{N} \times (\mathbb{N}_{>0})$, Äq.-Relation $(x_1, y_1) \sim (x_2, y_2) \iff x_1 \cdot y_2 = x_2 \cdot y_1$, Repr. von $[(x, y)]_\sim$ ist $(x/g, y/g)$ mit $g = \gcd(x, y)$

Mengensysteme (DSF, disjoint set forest)

- zur Implementierung von Union / find
  - find(x): die Wurzel des Baumes, der x enthält.
  - Union(x, y): find(x) wird Kind von find(y).

- Realisierung des Baumes durch Rückwärts-Zeiger (jeder Knoten auf Vorgänger) in Array $p : V \to V \cup \{\perp\}$

  find(x) : **if** $p[x] = \perp$ **then** x **else** find$(p[x])$

  Satz: $\forall x$: find(x) ist eine Wurzel

- Union(x, y) :

  $s :=$ find$(x); t :=$ find$(y);$ **if** $s \neq t$ **then** $p[s] := t$

  korrekt, aber es können unbalancierte Bäume entstehen.

Effiziente Implementierung von Union in DSF

- Basis: Union(x, y) :

  $s :=$ find$(x); t :=$ find$(y);$ **if** $s \neq t$ **then** $p[s] := t$

- Verbesserung:

  ... **if** height$(s) <$ height(t) **then** $p[s] := t$ **else** $p[t] := s$

- dann gilt $\forall x$: height$(x) \leq \log($size$(x))$

- height(x) in Array $H : V \to \mathbb{N}$ abspeichern.

  Aktualisieren: $p[t] := s; H[s] := \max\{H[s], 1 + H[t]\}$

Effiziente Implementierung von find in DSF

- Basis: find(x) : **if** $p[x] = \perp$ **then** x **else** find$(p[x])$
- Verbesserung: Pfad-Kompression:
  - Wurzel s für x bestimmen wie bisher
  - für jeden Knoten y auf Pfad von x zu s: $p[y] := s$
- Berechnung von $H[]$ wird nicht geändert
  - Bestimmung der tatsächlichen Höhe ist zu teuer
  - Pfadverkürzungen wirken sich nicht auf H aus
  - es gilt $\forall x$: height$(x) \leq H[x]$
- insgesamt wird Laufzeit deutlich verbessert

  (auf $o(\log^*(n)) \subset o(\log n)$, Details siehe Literatur.)

Zusammenf./Ausblick: MST-Algorithmen

- Algorithmus von Kruskal (1956) Laufzeit
  $O(m \log m) \subseteq O(m \log n)$

- Der Algorithmus von Prim (1957) (und Jarnik, 1930) zur Bestimmung eines MST
  - ist ähnlich zu später behandeltem Algorithmus von Dijkstra zur Bestimmung kürzester Pfadlängen,
  - wird deswegen danach und nur kurz (oder als Übungsaufgabe) behandelt.
  - hat Laufzeit $O(n \log n + m)$

Übungsserie 12 für KW 26

Aufgabe 12.A (autotool)

- Breitensuche
- Tiefensuche
- Minimalgerüst

Aufgabe 12.1 (vorrechnen)

Algorithmen an der Tafel ausführen, ggf. auch als Teil der Diskussion einer der folgenden Aufgaben.

- (1 P) Breitensuche
- (1 P) Tiefensuche
- (1 P) Algorithmus von Kruskal zur Bestimmung eines Minimalgerüstes
- (1 P) Union/Find-Operationen in Disjoint Set Forest

Aufgabe 12.2 (Graph-Eigenschaften)

- (1 P) Bestimmen Sie Durchmesser und Radius des Petersen-Graphen.

- (1 P) Geben Sie Algorithmen zur Bestimmung von Durchmesser und Radius von Graphen an ($=$ Aufgabe [WW] 4.15).

- (2 P) Wenn $\mathrm{rad}(G) = r$ bekannt ist, z.B. $r = 3$, welches sind der minimale und der maximale Wert von $\mathrm{diam}(G)$? Geben Sie Graphen an, die diese Werte erreichen.

Aufgabe 12.3 (Tiefensuche)

- (2 P) Sei DFS$(G) = [\ldots, t_1, \ldots, t_2, \ldots]$ der Tiefensuchwald eines gerichteten Graphen G (d.h., der Baum t_1 wird vor dem Baum t_2 durchlaufen).

  Geben Sie ein Beispiel an, bei dem G eine Kante von t_2 nach t_1 enthält.

  Begründen Sie, daß G niemals eine Kante von t_1 nach t_2 enthält.

- (2 P) Seien $t \in$ DFS(G), Branch$(x, [\ldots, s_1, \ldots, s_2, \ldots])$ Teilbaum von t und Knoten $y \in s_1, z \in s_2$.

  Begründen Sie: falls $y \rightarrow_G^* z$, dann $y \rightarrow_G^* x$.

  Hinweis: benutzen Sie $<_{\text{post}(F)}$

Aufgabe 12.4 (Minimalgerüste)

- (1 P) Lösen Sie Ihre Instanz der autotool-Aufgabe *Minimalgerüst* durch den Algorithmus von Kruskal.

- (3 P) [CLR] Problem 23-4 (Alternative Algorithmen . . .)

Kürzeste Wege

Überblick

WW 10

- von einem Knoten: Algorithmus von Dijkstra WW 5.3
- zwischen allen Knoten: Algorithmus von Floyd und Warshall

Definitionen

- Spezifikation: kürzeste Wege von *einem* Knoten zu *allen* anderen (single source shortest paths)
  - Eingabe:
    - kantengewichteter Graph (G, w), Gewichte ≥ 0,
    - Knoten $s \in V(G)$
  - Ausgabe: Funktion (Array)
    $d : V \to \mathbb{R} \cup \{\infty\}$ mit $d[t] = \mathrm{dist}_{(G,w)}(s, t)$

- Bezeichnungen:
  - $\mathrm{dist}_{(G,w)}(s, t) := \min\{w(P) \mid P$ ist Pfad von s nach $t\}$
  - $w(P) = \sum\{w(e) \mid e \in P\}$ (wurde schon definiert)
  - das ist sinnvolle Verallgemeinerung von $\mathrm{dist}_G(s, t)$

Algorithmus von Dijkstra

- Plan: Anpassung der Breitensuche
  - Spezifikation: Länge $\rightarrow$ Gewicht, $|P| \rightarrow w(P)$
  - Implementierung: Warteschlange (queue) $\rightarrow$ Prioritätswarteschlange (heap)

- $c[\cdot] :=$ weiß; $d[\cdot] := \infty$; $H := \{(s, 0)\}$; $c[s] :=$ grau; $d[s] := 0$
  while grau $\neq \emptyset$
    $x :=$ extractMin(H); $c[x] :=$ schwarz
    for $y \in G(x) \setminus$ schwarz
      $d[y] := \min\{d[y], d[x] + w(x, y)\}$
      if $c[y] =$ weiß **then** {insert$(y, d[y], H)$; $c[y] :=$ grau}
      else if $c[y] =$ grau **then** decrease$(y, d[y], H)$

- Inv: $\forall v : \mathrm{dist}(s, v) \leq d[v]$, $\forall v \in$ schwarz : $\mathrm{dist}(s, v) = d[v]$

Algorithmus von Dijkstra: Korrektheit

- $\forall v : \mathrm{dist}(s, v) \leq d[v]$:

  wenn $d[y] := d[x] + w(x, y)$, dann gibt es nach Induktion einen Weg mit diesen Kosten von s zu y (über x).

- $\forall v \in$ schwarz : $\mathrm{dist}(s, v) = d[v]$

  Beweis indirekt: betrachte erstes v, für das direkt vor Schwarz-Färbung gilt $\mathrm{dist}(s, v) < d[v]$.

  dann gibt es Pfad $P : s \rightarrow^* v$ mit Kosten $< d[v]$.

  $b :=$ der erste nicht schwarze Knoten auf P,

  $a :=$ Vorgänger von b ($a \in$ schwarz, $b \in$ grau)

  nach Induktion $d[a] = \mathrm{dist}(s, a)$

  nach Konstruktion $w(P) \geq d[b] \geq d[v]$, Widerspruch.

- Bemerkung: hier wurde $\forall e : w(e) \geq 0$ benutzt.

Algorithmus von Dijkstra: Laufzeit

- Kosten für die Bestandteile des Algorithmus:
  - Verwaltung der Knotenfarben (schwarz, grau, weiß) durch direkten Zugriff (Array) in $O(1)$
  - effiziente Prioritätswarteschlange (z.B. Quake Heap)
    - n mal extractMin, jeweils $O(\log n)$
    - m mal insert oder decrease, jeweils $O(1)$
- Gesamtkosten: $O(n \log n + m)$

Algorithmus von Dijkstra: über den Autor

- Edsger W. Dijkstra, 1930–2002
- ACM Turing Award 1972, `http://amturing.acm.org/award_winners/dijkstra_1053701.cfm`
- Manuskripte:
  `http://www.cs.utexas.edu/users/EWD/`

  Lesetipp: EWD 1305 *Answers to questions from students of Software Engineering*

Kürzeste Wege für alle Knotenpaare

- [CLR] Kapitel 25, [OW] 9.5.3, [WW] 8.2
- Spezifikation: (all pairs shortest paths)
  - Eingabe:
    kantengewichteter Graph (G, w), Gewichte ≥ 0,
  - Ausgabe: Funktion (Array)
    $d : V^2 \to \mathbb{R} \cup \{\infty\}$ mit $d[s, t] = \mathrm{dist}_{(G,w)}(s, t)$
- mögliche Implementierung:
  for $s \in V$: Algorithmus von Dijkstra mit Start s
  Kostet $O(n^2 \log n + nm)$,
  erfordert schnelle (komplizierte) PQ-Implementierung
- betrachten hier einfache Algorithmen in $O(n^3)$
- auch als Wiederholung zu dynamischer Programmierung

Der tropische Halbring

- auf der Menge $\mathbb{T} = \mathbb{R} \cup \{+\infty\}$ definieren die Operationen
  - $\oplus$: Minimum (neutrales Element $+\infty$)
  - $\otimes$: Addition (neutrales Element 0)

  einen Halbring (Monoid $(\mathbb{T}, \oplus, +\infty)$, Monoid $(\mathbb{R}, \otimes, 0)$, Distributivgesetz)

- $\mathbb{T} =$ tropischer Halbring, zu Ehren des brasilianischen Mathematikers Imre Simon (1943–2009)

- dieser Halbring paßt zu Minimierung von Wegen, denn
  - $\otimes$: Reihenschaltung, Addition entlang eines Weges
  - $\oplus$: Parallelschaltung, Auswahl eines kürzesten
  - ∞: *kein* Weg; 0: *ein* Weg der Länge 0

Matrizen über Halbringen

- Bezeichnung: $\mathrm{Mat}_n(S)$: Menge der $n \times n$-Matrizen über S
- Satz: S ist Halbring $\Rightarrow$ $\mathrm{Mat}_n(S)$ ist Halbring, mit
  - $\oplus_{\mathrm{Mat}(S)}$ ist die übliche Matrixaddition
    $(A \oplus_{\mathrm{Mat}(S)} B)_{i,j} = A_{i,j} \oplus_S B_{i,j}$, neutral: Null-Matrix
  - $\otimes_{\mathrm{Mat}(S)}$ ist übliche Matrixmultiplikation
    $(A \otimes_{\mathrm{Mat}(S)} B)_{i,j} = (A_{i,1} \otimes_S B_{1,j}) \oplus_S \cdots \oplus_S (A_{i,n} \otimes_S B_{n,j})$
    neutral: Einheitsmatrix
- Bsp. in $\mathrm{Mat}_2(\mathbb{T})$: $0 = \begin{pmatrix} \infty & \infty \\ \infty & \infty \end{pmatrix}, 1 = \begin{pmatrix} 0 & \infty \\ \infty & 0 \end{pmatrix},$

$$\begin{pmatrix} 2 & \infty \\ 3 & 0 \end{pmatrix} \oplus \begin{pmatrix} 1 & 3 \\ 2 & \infty \end{pmatrix} = \ldots, \begin{pmatrix} 2 & \infty \\ 3 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 3 \\ 2 & \infty \end{pmatrix} = \ldots$$

Matrizen von Weg-Kosten

- gewichteter Graph (G, w) mit $V(G) = \{1, \ldots, n\}$
  ist eine tropische Matrix W
  mit $W_{x,y} =$ **if** $(x, y) \in E$ **then** $w(x, y)$ **else** ∞
- für die Potenzen von W gilt:
  $(W^k)_{x,y} =$ min. Kosten aller Wege der Länge k von x zu y
- die all-pairs-shortest-paths-Matrix zu W ist demnach
  $W^* = 1 \oplus W \oplus W^2 \oplus \ldots$ (unendl. Summe in $\mathrm{Mat}_n(\mathbb{T})$)
- alle Kosten $\geq 0 \Rightarrow$ kürzeste Wege haben $< n$ Kanten
  $W^* = 1 \oplus W \oplus W^2 \oplus \cdots \oplus W^{n-1}$ (endliche Summe)
  $= (1 \oplus W)^{n-1}$ (Addition ist komm. und idempotent)

Berechnung von Matrix-Potenzen

- Berechnung von M^n für $M \in \mathsf{Mat}_n(S)$ durch

  $R := 1; \textbf{for } k \in [1, \ldots, n] \{R := R \otimes M\}$

  kostet $n \cdot n^3$ Operationen.

- Berechnung durch

  $R := M; \textbf{for } k \in [1, \ldots, \lceil \log_2 n \rceil] \{R := R \otimes R\}$

  kostet $\log n \cdot n^3$ Operationen.

- mittels dynamischer Programmierung ist $O(n^3)$ erreichbar

Algorithmus von Floyd und Warshall

- Eingabe: Kostenmatrix $W \in \mathrm{Mat}_n(\mathbb{T})$, Elemente ≥ 0
  Ausgabe: Matrix der kürzesten Wege $W^* = (1 \oplus W)^{n-1}$
- Implementierung: berechne Folge $M_0, \ldots, M_{n-1} = W^*$
  mit $M_h[x, y] :=$ minimale Kosten aller Wege
  $x = p_0 \to p_1 \to \cdots \to p_k = y$ mit $p_1, \ldots, p_{k-1} \in \{1, \ldots, h\}$
  - h ist Parameter der dynamischen Programmierung
  - $M_0 := 1 \oplus W$ (Wege haben Länge 0 oder 1)
  - $M_{h+1}[x, y] := M_h[x, y] \oplus M_h[x, h+1] \otimes M_h[h+1, y]$
    (Fallunterscheidung: Weg von x nach y benutzt $h+1$,
    dann höchstens einmal, oder gar nicht)
- Kosten: $O(n^3)$
- R. Floyd, CACM 5(6) 1962; S. Warshall, JACM 9(1) 1962

Übungsserie 13 für KW 27

Allgemeines

- Mo. 7:30 N001 Vorlesung (= Zusammenfassung)
- Mo. 12:00 G119 Hörsaal-Übung

  (Serie 13 sowie ggf. frühere Zusatzaufgaben)
- Do. oder Fr. (online-Stundenplan beachten!)

  Hörsaal-Übung (Prüfungsvorbereitung) (Hr. Wenzel)

  Fragen, die dann diskutiert werden sollen, sind Herrn Wenzel *vorher* per Mail mitzuteilen.

Aufgaben 13.A (autotool)

zu Algorithmen von

- Dijkstra,
- Prim,
- Floyd und Warshall

Das sind dann nicht unbedingt Pflicht-Aufgaben. Nutzen Sie trotzdem diese Gelegenheit, sich diese Algorithmen vom autotool vorführen zu lassen.

Aufgabe 13.1 (Algorithmus von Prim)

Der Algorithmus von Prim zur Bestimmung eines Minimalgerüstes entsteht aus dem Algorithmus von Dijkstra zur Bestimmung kürzester Pfade, indem die Anweisung $d[y] := \min\{d[y], d[x] + w(x, y)\}$ durch die Anweisung $d[y] := \min\{d[y], w(x, y)\}$ ersetzt wird.
Jedesmal, wenn dort $w(x, y) < d[y]$, notieren wir x als Vorgänger von y (in einem Array p durch die Zuweisung $p[y] := x$). Diese Vorgänger-Zuordnung kann wechseln, solange y grau ist.
Die Vorgängerkanten bilden am Ende des Algorithmus einen Baum, der ein Minimalgerüst für (G, w) ist.

- (1 P) Führen Sie den Algorithmus von Prim für Ihre Instanz der autotool-Aufgabe *Minimalgerüst* durch.

- (2 P) Begründen Sie: jedesmal, wenn ein x als Minimum extrahiert wird, dann ist die Kante $\{x, p[x]\}$ eine sichere Kante für den Schnitt zwischen $C =$ schwarz und $V \setminus C$. (Das ist [WW] Aufgabe 5.6)

Aufgabe 13.2 (Algorithmus von Dijkstra)

- (2 P) Wie unterscheidet sich asymptotisch die Komplexität einer Implementierung des Algorithmus von Dijkstra mit fast vollständig balancierten binären Heaps von einer Implementierung mit Quake Heaps
  - für Graphen mit $|E| \in \Theta(|V|^2)$ (sogenannte dichte Graphen)?
  - für Graphen mit $|E| \in \Theta(|V|)$ (sogenannte dünne Graphen)?

- (2 P) [CLR] Übung 24.3–8. Was passiert für $W = 1$? Für $W = 2$? Zusatz : 24.3–9.

Aufgabe 13.3 (all pairs shortest paths)

- (2 P) Für jede der drei im Skript angegebenen Variante der Berechnung der Matrix W^*: kann die Rechnung abgekürzt (d.h., die Komplexität verringert) werden, falls die kürzesten Pfade nur von einem Knoten aus zu bestimmen sind?

- (2 P) Welche Information über G enthält die Matrix A^*, wenn A die Adjazenzmatrix des Graphen G ist und der zugrundeliegende Halbring $\mathbb{B}$ (Wahrheitswerte) mit den Operationen $\oplus = \vee$ und $\otimes = \wedge$?

Weitere Beispiele zu Graphen und -Algorithmen

Färbungen (Definition, Beispiele)

- eine (konfliktfreie) c-Färbung von G

  ist Abb. $f : V \to \{1, \ldots, c\}$ mit $\forall uv \in E : f(u) \neq f(v)$

- chromatische Zahl $\chi(G) = \min\{c \mid G \text{ hat eine } c\text{-Färbung}\}$

- Bsp: χ von: $P_5, C_5, I_5, K_5, K_{3,3}$, Petersen.

Färbungen (Geschichte)

- Der Vierfarbensatz: G ist planar (besitzt kreuzungsfreie ebene Zeichnung) $\Rightarrow \chi(G) \leq 4$
- Diese Vermutung (Guthrie 1852, Cayley 1878) und ihre Beweisversuche (Kempe 1879, Tait 1880) sind ein Ursprung der modernen Graphentheorie.
- Beweis (mit Computerhilfe) durch Appel und Haken 1977
- vereinfacht durch Robertson, Saunders, Seymour, Thomas 1995
- maschinell verifizierter Beweis durch Gonthier 2005, siehe `http://www.ams.org/notices/200811/`

Chromatische Zahl und Maximalgrad

- Satz: $\chi(G) \leq \Delta(G) + 1$
- Beweis: Konstruktion einer Färbung durch den folgenden (offensichtlichen) Greedy-Algorithmus:

  Für eine beliebige Reihenfolge $[v_1, \ldots, v_n]$ der Knoten:

  $c(v_i) :=$ mex M mit $M = \{c(v_j) \mid j < i \wedge v_j v_i \in E\}$,
  mex $M := \min(\mathbb{N}_{>0} \setminus M)$
- Bsp: ein G und eine Knotenreihenfolge, so daß die vom Algorithmus berechnete Färbung nicht minimal ist.
- Beweis: folgt aus $c(v_i) \leq |M| + 1$ und $|M| \leq |G(v_i)|$
- Aufgabe: für welche Graphen gilt $\chi(G) = \Delta(G) + 1$?

Der Satz von Brooks

- Brooks 1941: G zusammenhängend und G nicht vollständig und G kein ungerader Kreis $\Rightarrow \chi(G) \leq \Delta(G)$.

- Beweis: (nach Cor Hurkens, `https://www.win.tue.nl/~wscor/OW/CO1b/`)
  - x mit Grad $\Delta(G)$, hat Nachbarn y, z mit $xy \notin E$.
  - wenn $G' = G \setminus \{y, z\}$ zshg., dann $T = \mathsf{BFS}(G', x)$. Knotenreihenfolge: y, z, dann $<_{\mathsf{post}(F)}$ (endet mit x) es ist immer eine Farbe aus $\{1, \ldots, \Delta(G)\}$ frei.
  - wenn G' nicht zshg., dann einfachere Fälle, siehe Quelle.

- Anwendung: $\chi(\text{Petersen}) \leq 3$

Hohe chromatische Zahl ohne Dreiecke

- Def: $\omega(G) := \max\{n : K_n \subseteq G\}$ (größte Clique in G)
- Satz: $\chi(K_n) = n$, Satz: $\omega(G) \leq \chi(G)$.
- Satz: $\forall c : \exists G_c : \omega(G) = 2 \wedge \chi(G) = c$

  Bsp: $G_2 = P_2, G_3 = C_5, G_4 = \ldots$
- Beweis (Mycielski, 1955)
  - Konstruktion von $M(G)$ für $G = (V, E)$ als
    - Knoten: $V \cup V' \cup \{0\}$ mit $V' =$ disjunkte Kopie von V
    - Kanten: $E \cup \{xy' \mid xy \in E\} \cup \{x'0 \mid x \in V\}$
  - Eigenschaften:
    - G ohne Dreieck $\Rightarrow M(G)$ ohne Dreieck
    - $\chi(M(G)) = \chi(G) + 1$
- Bsp: $C_5 = M(P_2)$. Def: Grötzsch-Graph $:= M(C_5)$

Effizientes Färben

- die Spezifikation
  - Eingabe G,
  - Ausgabe: konfliktfreie Färbung $c : V(G) \to 2$, falls existiert, NEIN sonst

  hat effizienten Algorithmus:

  BFS, Schichten abwechselnd färben, auf Konflikte prüfen.

- für „Eingabe G, Ausgabe: $\ldots c : V(G) \to 3$,$\ldots$“

  ist kein Polynomialzeit-Algorithmus bekannt.

Komplexitätstheorie

- . . . untersucht Schwierigkeiten von *Problemen*

  gewünschtes Resultat: „P ist schwer“,

  d.h., besitzt *keinen* Polynomialzeit-Algorithmus

- das ist aber sehr schwer nachzuweisen,

  vgl. auch die „Million-Dollar-Frage“ P $\stackrel{?}{=}$ NP

  `http://www.claymath.org/millennium-problems/p-vs-np-problem`

  – P: Menge der in Polynomialzeit lösbaren Probleme
  – NP: Menge der durch Suchbäume polynomieller Tiefe lösbaren Probleme

  2-Färbung $\in$ P, 3-Färbung $\in$ NP, offen: 3-Färbung $\in$ P

Reduktionen

- wenn man für $A \in$ NP nicht zeigen kann, daß $A \notin$ P,
- dann zeigt man ersatzweise „A gehört zu den schwersten Problemen in NP“,

  Def: falls jedes $B \in$ NP sich *auf* A *reduzieren* läßt:
- $B \leq_P A$, „A wenigstens so schwer wie B“
- . . . durch Angabe von Polynomialzeitverfahren für

  f : Eingabe für $B \rightarrow$ Eingabe für A mit
  $\forall x : x \in B \iff f(x) \in A$
- Bsp: $\forall B \in$ NP : $B \leq_P$ Halteproblem für NP $\leq_P$ SAT

  SAT $=$ erfüllbare aussagenlogische Formeln $\leq_P$ 3COL $=$ 3-färbbare Graphen ($\Rightarrow$ SAT und 3COL gleich schwer)

Zusammenfassung

Übersicht

was haben wir hier eigentlich gelernt?

- Ideen — z.B.: Balance
- Sätze — zeigen die Nützlichkeit der Ideen

  z.B.: t ist AVL-balanciert $\Rightarrow$ height$(t) \leq \log_{1.6}$ size(t)
- Methoden — sind Quellen für Ideen

  z.B.: Induktion über die Höhe von Bäumen

und *warum*?

- Methoden sind *universell* ($\Rightarrow$ *wiederverwendbar*)

Algorithmen

- (Entwurf $\Rightarrow$) Spezifikation $\Rightarrow$ Algorithmus ($\Rightarrow$ Programm)

- wesentliche Eigenschaften von Algorithmen:
  - Korrektheit (bzgl. Spezifikation)
  - Komplexität (bzgl. Kostenmodell)

- Methoden für Korrektheitsbeweise

  $\Rightarrow$ Methoden zum Algorithmenentwurf

  - Induktion (von $n-1$ auf n)
    $\Rightarrow$ lineare Rekursion (Iteration)
  - Induktion (von mehreren $n_i < n$ auf n)
    $\Rightarrow$ baum-artige Rekursion (divide and conquer)

Datenstrukturen

- *konkrete* Repräsentation (z.B.: Baum, Hash-Tabelle)

  eines *abstrakten* (mathemat.) Begriffs (z.B.: Menge)

- die *Axiome* des abstrakten Datentyps sind die Spezifikation für die Operationen des konkreten Datentyps

- die *Idee* hinter jeder Datenstruktur besteht aus:
  - *Invariante* (Baum ist Suchbaum, ist AVL-balanciert, $x \in T \iff T_1(h_1(x)) = x \vee T_2(h_2(x)) = x$,...)
  - effizienten *Algorithmen* zum Erhalt der Invariante (Rotation, Insert mit Verdrängen alter Einträge,...)

Komplexität, Effizienz

- Komplexität eines *Algorithmus* A ist Funktion c_A : Eingabegröße $s \mapsto$ maximale Laufzeit von A auf Eingaben der Größe $\leq s$
- asymptotischer Vergleich von Komplexitäten $c_A \in O(c_B)$ (unabh. von Hardware, Programm, Sprache, Compiler)
- Komplexität c_S einer *Aufgabe* (einer Spezifikation S): beste Komplexität aller Algor., die S implementieren
  - *obere* Schranke für c_S: Angabe *eines* Algorithmus,
  - *untere* Schranke: direkter *Unmöglichkeitsbeweis* (Bsp: Entscheidungsbäume f. Sortierverfahren) oder *Reduktion* von anderer Spezifikation bekannter Komplexität (PQ mit insert *und* extractMin in $O(1)$?)
- These: S effizient lösbar $\iff$ $c_S \in$ Polynome

Sortieren (Technik)

- untere Schranke (Entscheidungsbäume): $\Omega(n \log n)$
- Verfahren und ihre Eigenschaften:
  - naiv: durch Einfügen, durch Auswählen: $O(n^2)$
  - verbessert: Shell-Sort: $O(n^{3/2})$
  - optimal: Merge-Sort
  - im Mittel optimal: Quick-Sort
  - optimal: Quicksort mit Median in $O(n)$
- Datenstrukt., mit denen man auch optimal sortieren kann:
  - balancierter Suchbaum (alle einfügen, dann Inorder)
  - Prioritätswarteschlange (Heap erzeugen, z.B. durch Einfügen, dann extractMin bis leer)

Sortieren (Einschätzung)

dieses Semester:

- diese Algorithmen, ihre Eigenschaften, deren Beweise:

  wurde gelehrt und geübt, wird geprüft.

- Implementierung von Algorithmen: VLn Programmierung

ab 3. Semester:

- wer Sortier-Algorithmen tatsächlich selbst implementiert, macht es falsch — es gibt Bibliotheken

- wer mittels Bibliotheksfunktion sortiert, macht es sehr wahrscheinlich auch falsch — denn er benutzt eine Liste, meint aber eine Menge, für die gibt es deutlich bessere Implementierungen

Suchbäume und Heaps

- Gemeinsamkeit: balancierte Binärbäume, mit Schlüsseln
- Unterschied:
  - Suchbaum: implementiert Menge
    hat Operation contains
  - heap-geordneter Baum: impl. Prioritätswarteschlange
    hat Operation extractMin (und *kein* contains)
- Diskussion:
  - man kann jeden Suchbaum als PQ verwenden,
  - aber dann sind insert und decrease zu teuer,
  - $\Rightarrow$ schlechtere Laufzeit für Dijkstra auf dichten Graphen

Wie weiter?

- 3./4. Sem.: Softwaretechnik/Softwareprojekt
  Entwurf von Softwaresystemen, d.h. Spezifikation ihrer Komponenten, so daß Anwendungsaufgabe gelöst wird
  Implementierung der Komponenten:
  Auswahl passender Datenstrukturen und Algorithmen
- in *jeder* Informatik-Vorlesung werden fachspezifische Algorithmen und Datenstrukturen untersucht
  (z.B. Bildverarbeitung: Quad-Trees, Künstliche Intelligenz: heuristische Suche, Theoretische Inf.: CYK-Parser)
- und dabei die hier gezeigten Entwurfs- und Analyse-Methoden angewendet
  (z.B. divide and conquer, dynamische Prog., O-Notation)

Wie weiter? (Werbung für meine Vorlesungen)

Bachelor

- 4. Sem: fortgeschrittene Programmierung (u.a. generische Polymorphie, Funktionen höherer Ordnung)
- 5. Sem: Sprachkonzepte der parallelen Programmierung

Master

- 1. Sem: Prinzipien von Programmiersprachen

  (Syntax, Semantik (statische, dynamische), Pragmatik)

- – Compilerbau (a.k.a. praktische Semantik)
  – Constraint-Programmierung
  – Symbolisches Rechnen

Eine Aufgabe für die Semesterpause

- Def: $c : V \to \{0, 1, \ldots, |E|\}$ heißt *graceful*, wenn
  - c injektiv
  - und $\{|c(x) - c(y)| : xy \in E\} = \{1, \ldots, |E|\}$.
- Def: $G = (V, E)$ heißt graceful,
  wenn für G eine graceful Färbung existiert.
- Bsp: wer ist graceful: $K_4, K_5, C_6, C_7, P_8, P_9, K_{1,4}$,
  vollst. Binärbaum der Höhe 2, 3 (autotool), $\ldots$, n
- Aufgabe: *jeder Baum ist graceful*. (Geben Sie einen Algorithmus an, der jeden Baum graceful färbt.)
- zum Üben: 1. Pfade, 2. Raupen ($=$ Löschen der Blätter ergibt Pfad), 3. Hummer ($=$ Löschen . . . Raupe)

Vorsicht! nicht zuviel Zeit investieren!

Plan der Vorlesungen/Übungen

Übersicht nach Kalenderwochen

- KW 14: 2 VL:
  - 1. Einführung
  - 2. Komplexität von Algorithmen und Problemen
- KW 15: Übungen Serie 1 (zu VL 1,2) — 2 VL:
  - 3. Asymptotischer Vergleich von Funktionen
  - 4. Einfache Algorithmen (1) Schleifen
- KW 16: Übungen Serie 2 (zu VL 3) — 1 VL:
  - 5. Einfache Algorithmen (2) Rekursion

- KW 17: Übungen Serie 3 (zu VL 4,5) — 2 VL:
  - 6. Sortieren (1) (Schranke, Inversionen, Shell-Sort)
  - 7. Sortieren (2) (Merge-Sort)
- KW 18: Übungen Serie 4 (zu VL 6) — 1 VL (Fr.):
  - 8. Analyse rekursiver Algorithmen
- KW 19: Übungen Serie 5 (zu VL 7) — 2 VL:
  - 9. Quick-Sort, Median
  - 10. Dynamische Programmierung
- KW 20: Übungen Serie 6 (zu VL 8,9) — 2 VL:
  - 11. greedy Algorithmen
  - 12. Bäume (1) Suchbäume

- KW 21: keine Übungen — 1 VL (Mo.):
  - Hörsaal-Übung Serie 7 (zu VL 10, 11) sowie Zusatz-Aufgaben
- KW 22: Übungen Serie 8 (zu VL 12) — 2 VL:
  - 13. Bäume (2) Balance, AVL
  - 14. Prioritätswarteschlangen, heap-geordnete Bäume
- KW 23: Übungen Serie 9 (zu VL 13) — 1 VL (Do.):
  - 15. Direkter Zugriff (1) Counting-Sort, Hashing
- KW 24: Übungen Serie 10 (zu VL 14) — 2 VL:
  - 16. amortisierte Analyse
  - 17. Graphen: Eigenschaften, einfache Algorithmen

- KW 25: Übungen Serie 11 (zu VL 15,16) — 2 VL (beide Mo.):
  - 18. Minimalgerüste, Mengensysteme
  - 19. kürzeste Wege (single source)
- KW 26: Übungen Serie 12 (zu VL 17,18) — 2 VL:
  - 20. kürzeste Wege (all pairs)
  - 21. weitere Algorithmen auf Graphen
- KW 27: Übungen Serie 13 (zu VL 19,20) — 2 VL (beide Mo.):
  - 23. Zusammenfassung, Ausblick
  - Hörsaal-Übung (Zusatz-Aufgaben, Fragen)

Wörter, die es nicht gibt

Häufige Übersetzungsfehler in der Informatik

- *merge* — mischen? richtig: *zusammenfügen*, einordnen, verzahnen, verschränken

  `http://www.roadtrafficsigns.com/merge-signs`
  thru traffic merge left: Durchgangsverkehr links mischen?

- (minimal) spanning tree — (kleinster) Spannbaum?

  wenn schon, dann „aufspannender Baum“, aber die empfohlene deutsche Bezeichnung (z.B. [Brandstädt], [Walther/Nägler] ist *(Minimal)gerüst*.

- *side effect* — Seiteneffekt? richtig: *Nebenwirkung*.

- *control flow* — Kontrollfluß?

  richtig: *Programmablauf(steuerung)*

  (*to control* — kontrollieren? richtig: *steuern*)

- *problem* — Problem? besser: *Aufgabe*

  sowohl bei Hausaufgaben als auch in der Komplexitätstheorie: die Aufgabe (nicht: das Problem) der 3-Färbung.

  [CLR] (engl.) verwendet *exercise* (einfache Übungsaufgabe) und *problem* (umfangreiche Hausaufgabe)