Gesamt

Grunddefinitionen (Laufzeit, ...)

Schranken:

$$f \in O(g) \implies \exists c > 0, n_0 : \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

$$f \in \Omega(g) \implies \exists c > 0, n_0 : \forall n \geq n_0 : c \cdot g(n) \leq f(n)$$

$$f \in \Omega(g) \implies \exists c_1, c_2 > 0, n_0 : \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 g(n)$$

$$g \in \Omega(f) \Leftrightarrow f \in O(g)$$

$$g \in \Theta(f) \Leftrightarrow f \in O(g) \wedge f \in \Omega(g)$$

Obere Laufzeitschranke: $O(f(n))$
Untere Laufzeitschranke (für jede Eingabelänge n gibt es eine Eingabe I, sodass der Algo mindestens $\Omega(g(n))$ Schritte braucht)
Fallen obere und untere Laufzeitschranke zusammen haben wir scharfe Laufzeitschranke Theta: $\Theta(f(n))$

Entscheidungsproblem:

Algorithmus A löst das Entscheidungsproblem für formale Sprache L wenn A für jede Eingabe anhält und entweder 0 oder 1 ausgibt wobei $x \in L \Leftrightarrow A(x) = 1$

MAXSUM

Input: $I = \{a_j | j \in J, a_j \in \mathbb{Z}\}$
Ausgabe: $\text{maxsum}(I) := \max\{f(i,j) \in J, i \leq j\}, f(i,j) = \sum_k^i a_k$
Kurz: Intervall Summe

Lineare Lösung DP:

```python
k = 1
maxI = a[1]
maxSuffix = a[1]
while(k < n):
    k = k + 1
    maxSuffix  = max(MaxSuffix + a[k], a[k])
    maxI = max(MaxSuffix, MaxI)
return maxI
```

Korrektheit:
Vor dem k-ten Schritt der Schleife gilt offenbar:
$\text{maxI} = \max\{f(i,j) | 1 \leq i \leq j \leq k\}$ und $\text{maxSuffix} = \max\{f(i,k) | i \leq k\}$
Also ist nach dem Ende der while-Schleife das korrekte Ergebnis berechnet

Baum Grunddef

- Ungerichteter Graph G besteht aus endliche Menge von Knoten V, endlicher Menge von Kanten E und Vorschrift, die jeder Kante e genau zwei Endknoten zuordnet.
- Baum $\leftrightarrow$ azyklischer, zusammenhängender Graph
- Wurzelbaum ist Baum mit Wurzel

Allgemein

- depth(v) $\leftrightarrow$ Länge des Pfades von Wurzel zu Knoten v (Eindeutig sonst Kreis)
- Transversale = Wege dieser Art, die nicht verlängert werden können, von der Wurzel weg orientiert
- Blatt = Knoten am Ende einer Transversalen
- height(v) = λ - depth(v), $\lambda =$ Länge der längsten Transversalen, die durch v führt
- depth(T) eines Baumen = Maximum über die längen seiner Transversalen (Tiefe des Wurzelbaumes nur mit Wurzel = 0, Tiefe des leeren Wurzelbaumes = -1)
- Teilbaum T_u = in u wurzelnder Teilbaum, besteht aus allen Knoten von T, die auf einer Transversalen durch den Knoten u aus der Sicht der Wurzel jenseits von u liegen. Ist wieder Wurzelbaum mit Wurzel u
- Für zwei Knoten $u, v, u \neq v$ gibt es genau einen minimalen Teilbaum der beide Knoten enthält. Dies ist der Teilbaum der im LCA von u, v wurzelt

Binäre Suchbäume

- Wurzelbaum heißt binärer Wurzelbaum, wenn der Ausgangsgrad jedes Knotens durch zwei beschränkt ist

- Für binäre Bäume:
- Voll = Alle Knoten die nicht Blätter sind haben genau zwei Kinder
- Vollständig = Voll + alle Blätter dieselbe Tiefe
- depth(T) = 1 + max{depth$T_{r(T)}^{(left)}$, depth$T_{r(T)}^{(right)}$}
- Strenge Suchbaumeigenschaft: Für jeden Knoten v und alle Knoten $w_1 \in T_v^{left}, w_2 \in T_v^{(right)}$: $key(w_1) < key(v) < key(w_2)$
- (Im linken Teilbaum alles kleiner, im rechten alles größer)
- v überdeckt $u \iff u < v$ aber (u, v) ist leer (zb, 2 < 3, aber $(2, 3)$ ist leer)

Ordnungen

Preorder: Root, Linkestes Kind, weniger Linkes Kind,... (Root, Left, Right)
Postorder: Left, Right, Root
Inorder: Left, Root, Right (Inorder angewandt auf BSB bringt sortierte Schlüssel)

BSB Lemmata

Sei T nichtleerer BSB, $T_{u,v} = T_1', r', T_2'$

$\implies u < v \iff (u = r' \land v \in T_2') \lor (u \in T_1' \land v = r') \lor (u \in T_1' \land v \in T_2')$

u wird von v überdeckt $\iff u = (r' \land v \in T_2') \lor (U \in T_1' \land v = r')$

BSB Operationen

Minimum: if u.l = null return u; return least(u.l)
Maximum: if u.r = null return u; return greatest(u.r)
Laufzeit für beide O(depth T_u)

Vorgänger von u:
if (u.L != null) return greatest(u.L);
while(...)
Gehe zu Parent;
Falls p.R = u oder p Vorfahre ist von u → return p;
sonst return null;

Nachfolger von u:
if(u.R != null) return least(u.R);
while(...)
Gehe zu Parent;
Falls p.L = u oder p Nachfahre ist von u → return

Laufzeit für beide O(depth T)

searchPath(int k, node v) returns node
if(v.key == k || (v.key < k && v.R = null) || (v.key > k && v.L == null)) { return v;}
if(v.key < k) return searchPath(k, v.L);
if(v.key > k) return searchPath(k, v.R);

Laufzeit O(depth$T$)

```
insert(int k, range r) {
    if(root() == null) {
        root = new Node();
        root.parent, root.L, root.R = null;
        root.key = k, root.data = r;
        return;
    }
    v = searchPath(k, root);
    if(v.key == k) v.data = r; return;
    u = new Node();
    u.key = k; u.data = r; u.L, u.R = null; u.parent=v;
    if(v.key > k) v.L = u;
    if(v.key < k) v.R = u;
    return;
}
```

Laufzeit O(depth T)

```
delete(int k) {
    if(root == null) return;
    v = searchPath(k, root);
    if (v.key() != k) return;
    //Falls v Blatt ist
    if (v.L == v.R == null) {
        if(v == root) root == null;
        else {
            w = v.parent;
            if(w.R == v) w.R = null;
            if(w.L == v) w.L == null;
        }
        return;
    }
    //v hat genau ein Kind
    if(v.L == null || v.R == null) {
        w = Kind von V;
        if(v == root)w.parent == null;root = w; return;
        u = v.parent;
        if(u.R == v) u.R = w;
        else u.L = w;
        w.parent = u;
        return
    }
    //v hat genau zwei Kinder
    w = previous(v);
    key x = w.key, range y = w.data;
    delete(w.key); //w hat kein rechtes Kind
    v.key = x, v.data = y;
}
```

Laufzeit O(depth T)

Mittlere Tiefe kanonischer BSB

Mittlere Tiefe: $\frac{1}{n!}\sum_{\pi\in S_n} depth(T_\pi)$
Definitionen:

- $d(\pi) := depth(T_\pi), (\pi \in S_n)$
- $\bar{d}(n) := \frac{1}{n!}\sum_{\pi\in S_n} d(\pi)$
  Wir werden stattdessen nutzen
- $D(\pi) := \sum_{v \text{ ist Blatt von } T_\pi} 2^{depth_{t_\pi}(v)}$
- $\bar{D}(n) := \frac{1}{n!}\sum_{\pi\in S_n} D(\pi)$

Lemma: $\bar{d}(n) \leq \log_2 \bar{D}(n)$
Beweis:

$$\begin{aligned}
\bar{d}(n) &= \sum_{\pi\in S_n} \frac{1}{n!}\log_2(2^{d(\pi)}) \\
(JensenscheU) &\leq \log_2\left(\sum_{\pi\in S_n}\frac{1}{n!}2^{d(\pi)}\right) \\
&\leq \log_2\left(\frac{1}{n!}\sum_{\pi\in S_n}\sum_{v \text{ ist Blatt von } T_\pi} 2^{depth(v)_{depth_\pi}}\right) \\
&= \log_2 \bar{D}(n)
\end{aligned}$$

Lemma: Für $\bar{D}$ gilt

- $\bar{D}(0) = 0$
- $\bar{D}(1) = 1$
- $\bar{D}(n) = \frac{4}{n}\sum_{i=0}^{n-1}\bar{D}(i)$

Lemma: Für $k \geq 2$ gilt: $\bar{D}(k) = \frac{(k+3)\cdot(k+2)\cdot(k+1)}{30}$

Beweis: Da $\bar{D}(2) = 2$ gehen wir weiter mit $k \geq 3$

$\bar{D}(n) = \frac{4}{n}\sum_{i=0}^{n-1}\bar{D}(i) \implies k\bar{D}(k) - (k-1)\bar{D}(k-1) = 4\bar{D}(k-1)$

Es folgt: $\bar{D}(k) = \frac{k+3}{k}\bar{D}(k-1)$
Also:

$$\bar{D}(k) = \frac{(k+3)\cdot(k+2)\cdot(k+1)\cdot k\cdot(k-1)\cdot\ldots\cdot 6}{k\cdot(k-1)\cdot\ldots 3}\cdot\bar{D}(2)$$

$$= \frac{(k+3)\cdot(k+2)\cdot(k+1)}{3\cdot 4\cdot 5}\cdot\bar{D}(2)$$

Schlussendlich folgt da $\bar{d}(n) \leq \log_2\bar{D}(n)$ und $\bar{D}(n) = O(n^3)$

$$\bar{d}(n) = \frac{1}{n!}\sum_{\pi\in S_n} depth T_\pi = O(\log n)$$

Also ist die erwartete Laufzeit von den BSB Operationen O(log n)

(2, 3) Bäume

Def:

- Jeder innere Knoten außer der Wurzel hat zwischen 2 und 3 Kinder
- Wurzel hat maximal 3 Kinder
- Alle Blätter haben dieselbe Tiefe
- → Für einen Baum mit n Knoten garantieren diese Eigenschaften eine Höhe in $\Theta(\log n)$
- Alle Schlüssel werden in den Blättern gespeichert
- Für innere Knoten wird die Höhe des Teilbaums und das kleinste Element aus dem Teilbaum gespeichert
- Ordnungseigenschaft: Elemente linker TB <= Elemente Mittlerer TB <= Elemente rechter TB

levelUp

Input: 2-4 geordnete (2,3) - Bäume mit gleicher Höhe
Output: Für 2 oder 3 Eingabäume → ein (2,3) Baum mit Höhe +1
Fall 1 Input: T_1, T_2
Erzeuge Baum T mit Wurzel t
T_1 und T_2 werden linker und rechter TB von t
min(T) = min(T_1),
height(T)= height(T_1) + 1

Fall 2 Input T_1, T_2, T_3
Erzeuge Baum T mit Wurzel t
T_1, T_2, T_3 werden respektive linker, mittlerer und rechter TB von t
min(T) = min(T_1),
height(T)= height(T_1) + 1

Output: Für 4 Eingabebäume → zwei (2, 3) Bäume mit Höhe +1
Input: T_1, T_2, T_3, T_4
return levelUp(T_1, T_2), levelUp(T_3, T_4)

mergeToSameHeight

Merge T_1, T_2, wobei alle Elemente aus T_1 kleiner als Elemente aus T_2 sind

Fall 1: height(T_1) = height(T_2) → return T_1, T_2

Fall 2: height(T_1) < height(T_2) und Wurzel von T_2 hat 2 Kinder
→ return levelUp(mergeToSameHeight($T_1, T_2.L$), $T_2.R$)

Fall 3: height(T_1) < height(T_2) und Wurzel von T_2 hat 3 Kinder
→ return lvelUp(mergeToSameHeight($T_1, T_2.L$), $T_2.M, T_2.R$)

Falls height(T_1) > height(T_2) wird T_2 analog mit rechtem TB von T_1 gemerged

merge

Fall 1. T_1 oder T_2 leer → return den nichtleeren Baum

Fall 2. mergeToSameHeight(T_1, T_2) gibt zwei Bäume zurück
→ return levelUp(mergeToSameHeight(T_1, T_2))
Fall 3. mergeToSameHeight(T_1, T_2) gibt einen Baum zurück
→ return mergeToSameHeight(T_1, T_2)

Laufzeit: levelUp $\in O(1)$
merge: Für Bäume mit gleicher Höhe $\in O(1)$
Für Bäume mit gleicher Höhe wird mergeToSameHeight & lvlUp genau |height(T_1) - height(T_2)| mal rekursiv aufgerufen
Also ist merge $\in O(|height(T_1) - height(T_2)|)$

split

Eingabe: (2, 3) Baum T, Funktion $f, f = 1 \iff$ übergebenes Element gehört in den rechten Baum, analog $f = 0$ für linken Baum

Output: Zwei (2, 3) Bäume die entsprechend f aufgeteilt wurden

Fall 1: T ist leer
→ return zwei leere Bäume T_1, T_2

Fall 2: t ist Wurzel von T und ist ein Blatt:
falls $f(t) = 1$ → return leerer Baum T_1 und T
falls $f(t) = 0$ → return T und leerer Baum T_1

Fall 3: t ist die Wurzel von T und hat zwei Kinder
falls $f(\min(T.R)) = 1$
rufe rekursiv split($T.L$) auf
dabei erhalten wir T_1, T_2
return $T_1, merge(T_2, T.R)$
---
falls $f(\min(T.R)) = 0$
rufe rekursiv split(T.R) auf
dabei erhalten wir T_1, T_2
return $merge(T.L, T_1), T_2$

Fall 4. t ist die Wurzel von T und hat drei Kinder
$f(\min(T.M)) = 1$
→ split(T.L) gibt uns zwei Bäume T_1, T_2
return $T_1, merge(T_2, levelUp(T.M, T.R))$
$f(\min(T.R)) = 1$
→ split(T.M) gibt uns zwei Bäume T_1, T_2
return $merge(T.L, T_1), merge(T_2, T.R)$
$f(\min(T.M)) = 0 \wedge f(\min(T.R)) = 0$
→ split(T.R) gibt uns zwei Bäume T_1, T_2
return $merge(levelUp(T.L, T.M), T_1), T_2$

Da wir mit jedem Aufruf der Split Operation ein Level tiefer gehen im Baum kommt es bei n Knoten zu $\log(n)$ Aufrufen.
Da die Bäume die während Split gemerged werden höchstens eine Height diff von 1 haben sind die einzelnen merge $\in O(1)$
→ $\in O(\log n)$

Insert

Soll k eingefügt werden:

1. Split mit $f(x) = x \geq k$ → yields T_1, T_2
2. T_1 mit k mergen und das Ergebnis mit T_2 mergen

Delete

Soll k gelöscht werden:

1. Split mit $f(x) = x \geq k$ yields T_1, T_2
2. Split mit $f(x) = x > k$ auf T_2, yields T_{2_1}, T_{2_2}
3. Merge T_1, T_{2_2}

Laufzeit:

- Insert: 1x Split, 2x Merge
- Delete: 2x Split, 1x Merge
- → $O(\log n)$

Hashing

Def:

- Hashfunktion:

$$h_m = h : U \to \{0, 1, \ldots, m-1\}$$

- Schlüssel aus der Menge $S \subset U$ sind durch h_m auf m Buckets $B_0, B_1, \ldots, B_{m-1}$ so verteilt, dass

$$B_i = \{x \in S | h(x) = i\}, (i = 0, 1, \ldots m-1)$$

- Auslastungsfaktor: Bestehe Schlüsselmenge S aus n Elementen und m Buckets dann ist aktueller Auslastungsfaktor $$ \alpha := \frac{n}{m}

$$Grundannahmen:$$

- $n \leq m \ll |U|$
- $\alpha_0 < \alpha \leq 1, \alpha_0 \in (0, 1)$
- Erste Ungleichung → keinen Platz vergeuden, zweite → lineare Auslastung der Hashtabelle gewünscht
- Uniformitätsannahme UF:

1. Es wird eine Menge ausgewählt, für $i \neq j$ ist $X_j \neq X_i$
2. $h(X_0, H(X_1), \ldots, h(X_{n-1}))$ sind eine Folge von unabhängigen Zufallsvariablen aus $\{0, 1, \ldots, m-1\}$. Für alle $i = 0, \ldots, n-1$ ist die zufällige Variable $h(X_i)$ über $\{0, 1, \ldots, m-1\}$ gleichverteilt:

$$P(h(X_i) = j) = \frac{1}{m}, j \in \{0, 1, \ldots, m-1\}$$

   → Ergo es ist gleichermaßen wahrscheinlich, dass jeder Schlüssel in einen der m Buckets platziert wird, unabhängig davon, wo ein anderer Schlüssel platziert war

Um zufällig auszuwählen erst Block auswählen, dann aus diesem Block zufälliges Element poppen

Lemma: Aus UF folgt

$$P(h \text{ ist für } \{X_0, X_1, \ldots, X_{n-1}\} \text{ perfekt }) \leq e^{-\frac{n(n-1)}{2m}} \leq e^{-\frac{\alpha_0(n-1)}{2}}$$

→ Dies zeigt, dass Kollisionen praktisch unvermeidbar sind, denn die Wahrscheinlochkeit, dass h für S perfekt ist geht exponentiell in $n = |S|$ gegen null

Strategien

Umspeichern: Wenn Auslastungsfaktor α eine untere bzw obere Schranke $\in (0, 1)$ erreicht wird in eine neue Hashtabelle halber bzw doppelter Größte umgespeichert. Zeitaufwand $\Theta(n)$

Hashfunktionen

Kriterien

1. schnelle Berechenbarkeit
2. gute Streuung "oft" vorkommender Schlüsselmengen

Bsp.

1. Divisionsrestmethode $h(x) = x \mod m$
2. Multiplikationsmethode $\lfloor((\theta \cdot x) \mod 1) \cdot m\rfloor$, $0 < \theta < 1$
3. Lineare Funktionen über $\mathbb{Z}_p, m = p > |\Sigma|$, prim,
    1. $h(\sigma_0, \sigma_1, \ldots, \sigma_{l-1}) := \left(\sum_{i=0}^{l-1} a_i, \sigma_i\right) \mod p$

Offenes Hashing

B[i] ist eine linked list
→ Bei Kollision wird die Liste um 1 verlängert

- Klasse OpenHashTable
- Hauptdatenfeld hashTable → Array von linkedlists
- hashtable.length = m Anzahl d. Buckets
- Berechnung von $h_m(x) = h(x)$

Methoden

empty(int m) - Erzeuge leeren Hashtabelle mit m Buckets
→ Erzeuge array von linkedlists mit Länge m, $\in O(m)$

insert(key k, range r)

1. Berechne $i = h(k)$

2. Falls $k \in B[i]$ mit Datum r' überschreibe mit r, return
3. Füge neuen Knoten (k, r) zu $B[i]$ hinzu
   $\in O(|B[h(k)]|)$

delete(key k)

1. berechne $i = h(k)$
2. Falls $k \in B[i]$ lösche aus $B[i]$
   $\in O(|B[h(k)]|)$

lookUp(key k)

1. berechne $i = h(k)$
2. Falls $k \in B[i]$ mit Datum r, return r else return null
   $\in O(|B[h(k)]|)$

Laufzeitanalyse

Da offenbar die Algorithmen durch die Länge der jeweiligen Liste beschränkt sind ist zur mittleren Laufzeitanalyse die Bestimmung der mittleren Listenlänge notwendig

Def:

- Für eine Folge von Zufallselement aus U: $X_0, X_1, \ldots, X_n$ ist $B_h(X_0, X_1, \ldots, X_{n-1})$ die Hashtabelle die man erhält, wenn man die ersten $n-1$ Folgenelemente in die leere Hashtabelle mit m buckets längs $h = h_m$ einfügt

3 Analysen, uns interessiert jeweils die erwartete Anzahl von Schlüsselvergleichen für die Operation:

1. Erfolglose Suche: $B_h(X_0, X_1, \ldots, X_{n-1}).lookUp(X_n)$
2. Erfolgreiche Suche nach festem Schlüssel wobei $i \in \{0, 1, \ldots, n-1\}$ beliebig aber fest: $B_h(X_0, X_1, \ldots, X_{n-1}).lookUp(X_i)$
3. Erfolgreiche Suche nach beliebigen Schlüssel wobei $i \in \{0, 1, \ldots, n-1\}$ zufällig und von Folge unabhängig:
   $B_h(X_0, X_1, \ldots, X_{n-1}).lookUp(X_i)$

Def abgeschwächte UF (impliziert UF offenbar):
Für Folge aus U gilt
$P(X_\chi = X_\mu = 0)$, $P(h(x_\chi) = h(X_\mu)) \leq \frac{1}{m}$

Es gilt für die erwartete Anzahl an Schlüsselvergleichen

1. Erfolglose Suche $\leq \alpha = \frac{n}{m}$
2. Erfolgreiche Suche mit festem Schlüssel $< 1 + \frac{n-1}{m} < 1 + \frac{\alpha}{2}$
3. erfolgreiche Suche nach zufälligem $< 1 + \frac{n-1}{2m} < 1 + \frac{\alpha}{2}$

Satz LZ:
Ist Hashtabelle mit stochastischem Modell UFab ist Laufzeit der 3 Grundop $\in O(1 + \alpha)$

Worst Case:

$$P(L \geq 3 \cdot \lambda(m)) < \frac{1}{m^2}$$

Universelles Hashing

Def:

- Klassen von Hashfunktionen: $\mathcal{H} = \mathcal{H}_m = \{h \mid h : U \to \{0, 1, \ldots, m-1\}\}$ mit gleichverteilten Zufallselementen H: $P(H = h) = \frac{1}{|\mathcal{H}|}$
- c-universelle Klasse von Hashfunktionen
  - Sei $c > 0 \in \mathbb{R}$, Menge von Hashfunktionen $\mathcal{H}_m$ heißt so, wenn für je zwei feste Schlüssel $k_1 \neq k_2$ und $H \in \mathcal{H}$ zufällig gilt

$$P(H(k_1) = H(k_2)) \leq \frac{c}{m}$$

Methoden:
empty(int m) - erzeuge leere Hashtabelle mit m Buckets

1. Wähle $h \in \mathcal{H}_m$ zufällig, initalisiere Datenfelder zur Beschreibung von h entsprechend
2. Erzeuge ein Feld der Länge m von leeren linked lists

Analyse

Wieder geht es um erwartete Anzahl von Schlüsselvergleichen in 3 Fällen wobei H zufällige Hashfunktion aus $\mathcal{H}$ ist, analog zu Offenes Hashing! fügen wir $n-1$ Schlüssel einer n langen Folge ein

1. Erfolglose Suche:

$$E(keyComp(B_h.lookUp(x_n))) := \frac{1}{|H|}\sum_{h\in\mathcal{H}} keyComp(B_h.lookUp(x_n))$$

2. Erfolgreiche Suche fester Schlüssel:

$$E(keyComp(B_h.lookUp(x_i))) := \frac{1}{|H|}\sum_{h\in\mathcal{H}} keyComp(B_h.lookUp(x_i))$$

3.

$$E(keyComp(B_h.lookUp(x_I))) := \frac{1}{|H|n}\sum_{h\in\mathcal{H}}\sum_{i=0}^{n-1} keyComp(B_h.lookUp(x_i))$$

   Lemma E':
   Für $\mathcal{H} = \mathcal{H}_m$ $c-$universelle Klasse von Hashfunktionen gilt
4. Erwartete Anzahl Schlüsselvergleiche erfolglose Suche <= $c \cdot \alpha$
5. ... erfolgreiche Suche fester Schlüssel < 1 + $c \cdot \alpha$
6. ... erfolgrecihe Suche rein zufälliger Schlüssel < $c \cdot \frac{\alpha}{2}$
   → Laufzeit der 3 Grundop $\in O(1+\alpha)$

Bemerkung: Menge $\mathcal{H} = \{0,1,\dots m-1\}^U$ aller Funktionen von U nach $\{0,1,\dots,m-1\}$ ist 1-universell

Geschlossenes Hashing

Kollisionsbewältigung beim Einfügen eines Schlüssels:
Sondierungsfolge von links nach rechts nach einem freien Bucket für diesen Schlüssel durchlaufen - "sondieren"
Offenbar muss immer $\alpha < 1$ sein

Beim suchen ebenfalls sondieren, sobald man leeren Bucket trifft erfolglose Suche (deswegen nicht löschen sondern durch null ersetzen)

Lineares Sondieren

Prähashfunktion mit Sondierungspermutation=
$h(x)+0 \mod m, h(x)+1 \mod m, \dots h(x)+m-1 \mod m$
Mit UF gilt

1. Erwartete Anzahl Schlüsselvergleiche erfolglose Suche geht bei festem α für $m \to \infty$ gegen $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
2. Erw. Anz. erfolgreich rein zufällig fest ... gegen $\frac{1}{2}+\frac{1}{1-\alpha}$

Ideales Hashing: Neu hinzukommende zufällige Schlüssel X sind unabhängig davon, was die Hashwerte der bereits gespeicherten Schlüssel sind, die Sondierungspermutation ist eine rein zufällige Permutation der Menge der Bucketindizes $\{0,1,\dots,m-1\}$

1. Beim idealen Hashing ist erwartete Anzahl erfolglos $< \frac{1}{1-\alpha}$
2. ... erfolglos rein zufällig $< \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$

Quadratisches Sondieren mit Prähashfunktion h

$$h(x,0) = h(x), h(x,1) = (h(x)+1) \mod m, h(x,2) = (h(x)-1) \mod m, h(x,3) = (h(x)+4) \mod m, \dots h(x,k) = \left(h(\right.$$

Sortieren

Def:

- In Situ = Zum Sortieren des Feldes A[0, n) wird nur O(log n) zusätzlicher Speicher gebraucht

Todo Rek tailcall

Suchalgorithmen

Instabil: Quicksort!, Heaps!, Selectionsort!!
Stabil: Mergesort!! (nicht in-situ), Countingsort!, Radixsort!, Bucketsort!
Stabil: Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind werden bewahrt
QS: Ggnbsp: $1, 7_1, 7_2, 4$
SS: Ggnbsp: $9_1, 9_2, 1$

Untere Schranke für vergleichsorientiertes Suchen

Bei deterministischen vergleichsorientierten Suchalgos gibt es im Laufe der Rechnungen immer verzweigungen $s_i < s_j$?
→ Entscheidungsbaum B_n für das vergleichsorientierte Sortieren von Problemstellungen mit Größe n

- Ist voller geordneter binärer Wurzelbaum
- innere Knoten Markierungen der Art $s_i < s_j$
- Ausgehenden Kanten mit 0 (ja), 1 (nein) markiert
- die $n!$ Blätter sind mit allen Permutationen $\pi \in S_n$ markiert
- Für jede Eingabe $s_1, s_2, \ldots, s_n$ gibt es einen eindeutig bestimmten Pfad von der Wurzel zu einem Blatt die immer die ausgehende 0-Kante bei $s_i < s_j$ auswählt wenn dies für die Eingabe wahr ist

Ist A Vergleichsorientierter Suchalgo kann man A eine Folge von Entscheidungsbäumen $(B_n(A))_{n\in\mathbb{N}}$ zuordnen, die für jedes n alle Problemstellungen der Größe n korrekt sortieren. Für jedes n und jede Eingabe $s_1, s_2, \ldots, s_n$ ist $\text{keyComp}_A(s_1, s_2, \ldots, s_n) = depth_{B_n(A)}(v)$

Mit der Stirlingshen Formel $n! = \sqrt{2\pi n} \cdot \frac{n^n}{e^n} \cdot e^{\frac{\Theta_n}{12n}}, \Theta_n \in (0,1)$

Damit folgt $\log_2 n! = n\log_2 n - 1,44n + \Theta(\log_2 n)$

Worstcase:
Da im schlechtesten Fall eines vergleichsorientierten Algos der Baum bis nach unten durchgelaufen werden muss braucht jeder vergleichsorientierte Algo im schlechtesten Fall $\Omega(n\log n)$ Vergleiche

Bew: Wir müssen die Höhe eines Entscheidungsbaums $B_n(A)$ bestimmen, in dem jede Permutaiton als erreichbares Blatt erscheint.

Angenommen $B_n(A)$ hat die Höhe h und l Blätter
→ Es gilt $n! \leq l$ (weil Blätter Permutationen sind)
→ Aber ein binärer Baum der Höhe h hat höchstens 2^h Blätter
→ $n! \leq l \leq 2^h$
→ $h \geq \log(n!) = \Omega(\log(n!))$, also $h = \Omega(n\log n)$

Avg Case:

$c \cdot n\log n \leq \log_2 n! \leq keyComp_A^{\text{average}}(n)$

Quicksort

Generisches QS:

quicksort(left, right)
if(right - left <= 1) return;
π = initaler Pivotindex;
pivot = A[π]
λ = left, ρ = right - 1;
while($\rho - \lambda \neq 0$)
//Bestimmung Vertauschungsindizes im Suchintervall
$\lambda\lambda = \min\{j \in [\lambda, \rho] \mid A[j] \geq pivot\}$
$\rho\rho = \max\{j \in [\lambda, \rho] \mid A[j] \leq pivot\}$
//Vertauschung und Aktualisierung Pivotindex
swap(A[$\lambda\lambda$], A[$\rho\rho$])
if($\pi == \lambda\lambda$) $\pi = \rho\rho$
if($\pi == \rho\rho$) $\pi = \lambda\lambda$
//Aktualisierung Grenzen Suchinterrvall
if($\lambda\lambda < \pi < \rho\rho$)$\lambda = \lambda\lambda + 1, \rho = \rho\rho - 1$
if($\lambda\lambda = \pi < \rho\rho$)$\lambda = \lambda\lambda, \rho = \rho\rho - 1$
if($\lambda\lambda < \pi = \rho\rho$)$\lambda = \lambda\lambda + 1, \rho = \rho\rho$
if($\lambda\lambda = \pi = \rho\rho$)$\lambda = \lambda\lambda, \rho = \rho\rho$
//Rekursion
if($\pi-$ left < right - π) {
left1 = left, right1 = π;
left2 = $\pi + 1$, right2 = right;
}
else {
left1 = $\pi + 1$, right1 = right;
left2 = left, right2 = π;

}
quicksort(left1, right1), quicksort(left2, right2)

Analyse

Annahmen:

- Schlüssel Menge $\{0, 1, \ldots, n-1\}$
- Eingabe Permutaion $\sigma \in S_n$
- $\forall \sigma \in S_n : Q(\sigma(0), \sigma(1), \ldots, \sigma(n-1)) = Q(\sigma)$, Anzahl d. Schlüsselvergleiche von QS auf Eingabe $A[0] = \sigma(0) \ldots A[n-1] = \sigma(n-1)$
- Uns interessiert mittlere Anzahl von Schlüsselvergleichen:

$$\bar{Q}(n) := \frac{1}{n!} \sum_{\sigma \in S_n} Q(\sigma)$$

Einfache deterministische Variante z.B. $\pi = left, \pi = right, \pi = \lfloor \frac{left+right}{2} \rfloor$
Es gilt dann für $n \leq 1 : \bar{Q}(n) = 0$
sonst $\bar{Q}(n) = (n-1) + \frac{2}{n} \sum_{i=0}^{n-1} \bar{Q}(i)$
Und insgesamt:

$$\bar{Q}(n) = 2\ln(2) \cdot n \log_2(n) - \Theta(n)$$

Verfeinert (etwa Median von 3)
$1,188.. \cdot n \log_2(n-1) - \Theta(n)$

In Situ?

→ right2 - left2 >= right1 - left1
→ Endrekursion/Tail Call → In situ

Heapsort

Heapbedingung: Elternknoten > Kinderknoten
Heap nahezu vollständig:

- Alle Tiefelevels bis auf das letzte vollständig
- Letztes Level von links nach rechts bis zu einer bestimmten Stelle voll

Heap als Array:

- Knoten durch Index in Niveauordnung dargestellt
- Schlüssel die sie halten sind Feldeinträge
- Heapsize → bis zu welchem Index der Heap reicht
- Feldgröße n maximale Größe des Heaps

Es gilt:

- Knoten i hat kein Kind $\iff 2i + 1 \geq n$
- Knoten i hat genau ein Kind $\iff 2i + 1 = n - 1$
- Knoten i hat genau zwei Kinder $\iff 2i + 2 \leq n - 1$
- Falls linkes Kind von i existiert ist es $2i + 1$
- Falls rechtes Kind von i existiert ist es $2i + 2$
- Falls i nicht Wurzel so ist parent von i durch $\lfloor \frac{(i-1)}{2} \rfloor$
- depth(A) = $\lfloor \log_2(n) \rfloor$

Beweisskizze
Ist das $d-1$ Tiefenniveau vollständig vorhanden sind das die Knoten $2^{d-1} - 1, 2^{d-1}, \ldots, 2^{d-1} + (k-1), \ldots, 2^d - 2$
Dann ist
Kinder von $2^{d-1} - 1 = 2^d - 1, 2^d$ ...(immer parent *2 + 1)

heapify(k, l)
Vorbedingung: Für alle Nachfahren k' von k gilt die Heapbedingung

if(2k+1 >= l) return; //k hat kein Kind
//Berechne max Kind
if(2k+1 == l-1) maxchild = 2k+1
if(A[2k+1] > A[2k+2]) maxchild = 2k+1
else maxchild = 2k+2
if(A[k] >= A[maxchild]) return; //Bereits Heap

//Rekursion
swap(A[k], A[maxchild]); heapify(maxchild, l)

Korrektheit & Analyse

Korrektheitsbeweisskizze

- Induktion über Anzahl Knoten von A[0, l)
- Wenn die HB in der Wurzel nicht erfüllt ist überträgt ein Aufruf der Funktion das Problem auf genau eines seiner Kinder und löst das Problem rekursiv für den entsprechenden TB

Analyse Lemma:
keyComp(heapify(k, l)) $\leq 2 \cdot \lfloor \log_2(\frac{l}{k+1}) \rfloor$

Bewskizze:

- T: TB mit Wurzel A[k]
- Knoten A[k] bewegit sich im Baum T nach unten. Wird immer mit größerem Kind ausgetauscht
- Maximal $r_0 = height(T)$ = Schritte, da $r_{0} = $ die log Formel und in jedem Aufruf max 2 Vergleiche folgt das Lemma

buildheap

Durchlaufe Knoten in reverse Level Ordnung beginndend mit dem vorletzten Tiefenniveau von unten nach oben und korrigieren die Heap-Eigenschaft (heapify), erste Knoten für den etw zu tun ist, ist parent(n-1) = $\lfloor \frac{(n-2)}{2} \rfloor = \lfloor \frac{n}{2} \rfloor - 1$

→ buildheap()
for($j = \lfloor \frac{n}{2} \rfloor - 1, \ldots, 0$): heapify(j, n)

Laufzeit $\in O(2n)$

heapsort

mit Feld A[0, n)

```
heapsort() {
    if n <= 1 return;
    buildheap();
    heapsize = n;
    while(heapsize >= 2) {
        swap(A[0], A[heapsize-1]);
        heapsize--;
        heapify(0, heapsize);
    }
}
```

Schlüsselvergleiche $< 2n(\lfloor \log_2(n) \rfloor + 1)$ Also loglinear

Mergesort

Merge Vorbed: Teilfelder A[v, v + l/2) und A[v + l/2, v+l) sind aufsteigend sortiert

```
merge(int v, l) {
    //bis eines der Teilfelder leer ist
    while(i != v + l/2 || j = v + l) {
        i = v, j = v + l/2, k = v;
        mini = min(B[i], B[j]);
        A[k] = mini;
        k++;
        if(mini == B[i]) i++;
        else j++;
    }
    if(i == v + l/2) {
        while(j != v + l) {
            A[k] = B[j];
            j++; k++;
        }
    }
    else {
        while(i != v + l/2) {
            A[k] = B[i];
            i++; k++;
        }
    }
}
```

Analyse:

1. Die Anzahl der Schlüsselvergleiche für das Mergen eines Arrays A der Länge l durch den Algo ist durch $l-1$ nach oben und durch $\frac{l}{2}$ abgerundet nach unten beschränkt

Bew: Nach jedem Schlüsselvergleich wird ein Schlüssel vom Hilfsfeld auf das Hauptfeld umkopiert. Dies geschieht, bis eine Hälfte des Hilfsfeldes leer ist. Im besten Fall reichen $\frac{l}{2}$ Vergleiche, das ist gd der Fall wenn jeder Schlüssel aus der linken Hälfte <= jedem Schlüssel aus der rechten Hälfte ist

Im schlechtesten Fall müssen die Schlüssel immer abwechselnd eingefügt werden und man kommt zu $l-1$ Vergleichen

2. Jeder vergleichsorientierte Algo A der einen Array der Länge l in der vorstehend spezifizierten Weise mischt, benötigt im schlechtesten Fall $l-1$ Vergleiche zwischen Schlüsselelementen

Bew: Nehmen wir an A sei ein Algo der mit weniger als $l-1$ Vergleichen worst case mischt. Nehmen wir an der Inhalt vom Array A ist $A[0] < A\left[\frac{l}{2}\right] < A[1] < A\left[\frac{l}{2}-1\right] < \cdots < A\left[\frac{l}{2}-1\right] < A[l-1]$

Dann kann eines der Paare $A[i], A\left[i+\frac{l}{2}\right]$ nicht miteinander verglichen worden sein. Wir können diese swappen und diese Permutation würde genau das gleiche Ergebnis bringen. Widerspruch

$\Rightarrow$ Merge $\in O(l), l$ Schlüsselvergleiche

Mergesort

```
mergesort(int v, int l) {
    if(l <= 1) return
    if(l == 2) sort(A[v, v+l) durchSchl.Vergl.)
    return;

    //Rekursion
    mergesort(v, l/2);
    mergesort(v + l/2,  ceil(l/2));
    merge(v, l);
}
```

Mergesort Analyse:
Wir nehmen an, dass Eingabelänge 2er Potenz ist: $n = 2^k$

Vergleiche $V(n)$berechnet durch:
$V(1) = 0$
$V(n) \leq n - 1 + 2 \cdot V\left(\frac{n}{2}\right)$

Für alle $l \leq \log_2 n = k$ gilt:

$$V(n) \leq (n-1) + (n-2) + \cdots + (n-2^{l-1}) + 2^l V\left(\frac{n}{2^l}\right)$$

Wegen $V\left(\frac{n}{2^k}\right) = V(1) = 0$ ist

$$V(n) \leq k \cdot n - \sum_{l=0}^{k-1} 2^l \leq k \cdot n - n + 1$$

Selectionsort

Finde kleinstes Element im unsortierten Teil, swappe mit dem ersten Element, wiederhole für kleineren Teil.

```
hochsterIndex = n-1;
einfuegeIndex = 0;
while(einfuegeIndex < hochsterIndex) {
    minPosition = einfuegeIndex;
    for(int idx von einfuegeIndex + 1 bis hochsterIndex) {
        if(A[idx] < A[minPosition]) minPosition=idx
    }
    swap(A[minPosition], A[einfuegeIndex]);
    einfuegeIndex++;
}
```

Um ein Array mit n Elementen mittels selectionSort zu sortieren muss $n-1$ Mal das Min bestimmt und ebenso oft getauscht werden

- Bei der ersten Bestimmung des Minimums sind n-1 Vergleiche notwendig, bei der zweiten n-2, ...
- → $(n-1) + (n-2) + \cdots + 3 + 2 + 1 = \frac{(n-1)\cdot n}{2} \in \Theta(n^2)$

Sehr wichtig (warum auch immer): Um ein Array mit n Elementen zu sortieren macht selectionSort höchstens n Schlüsseltausche

Countingsort

Sortiert n Eleente in $O(n+k)$ wenn alle Schlüssel $\in \mathbb{N} \in [0, k-1]$ sind

countsort(A: Array mit n Elementen mit Schlüsseln aus $[0, k-1]$)
GS1: Für i von 0 bis $k-1$: $C[i] = 0$
GS1: Für i von 0 bis $n-1$: $C[A[i].key] + +$
GS1: Für i von 0 bis $k-1$: $C[i] = C[i] + C[i-1]$
GS1: Für j von $n-1$ bis 0 : $B[C[A[j].key] - 1] = A[j].key$,
$C[A[j].key] = C[A[j].key] - 1$

Beweisskizze korrekt:

Nach GS2: $C[i]$ enthält die ANzahl der Elemente, deren Schlüssel genau i ist
Nach GS3: $C[i]$ enthält die ANzahl der Elemente, deren Schlüssel $\leq i$ ist
GS4: Die Elemente mit Schlüsseln i werden von rechts nach links an de richtige Stelle im sortierten Array B gespeichert

Wenn noch t Elemente mit dem Schlüssel $\leq i$ zu platzieren sind und wir ein Element mit dem Schlüssel i platzieren müssen, speichern wir es in $B[t-1]$ und subtrahieren dann eins von der Anzahl der Elemente mit dem Schlüssel $\leq i$ die wir noch platzieren müssen

Radixsort

Erst nach letzter Ziffer sortieren (mit stabilem Algo eig immer Countingsort!, dann nach vorletzter... wenn keine Ziffer vorhanden in Zahl kleinste (0) einsetzen)

Korrektheitsbewskizze:
Wir können die folgende Eigenschaft durch Induktion zeigen_ Nach der r-ten Ausführung der Schleife sind die r-Tupel, die aus den letzten r Komponenten der Schlüssel bestehen, korrekt angeordnet!

Für $A[i].key = (z_1^i \ldots, z_d^i)$ und $1 \leq r \leq d$ sei $B_r[i] = z_{d-r+1}^i, \ldots, z_d^i$. Nach der r-ten Ausfürhung der SChleife wird das Array B_r korrekt sortiert.
Nach der d-ten Ausführung der Schleife wird das Array A korrekt sortiert.

Satz: Radixsort sortiert n b-bit Zahlen korrekt in $\Theta\left(\frac{b(n+2^r)}{r}\right)$ wobei r eine positive ganze Zahl mit $r \leq b$ ist.

Bew: Jede b-bit-Zahl kann als d-r-ziffrige Dezimalzahl betrachtet werden, die jeweils auf r Bits gespeichert sind, für $d = \lceil \frac{b}{r} \rceil$.

Hier: r-Bit-Ziffer → Dezimalzahl in [0, 2^r)
Da jede r-Bit-Ziffer (streng) kleiner als $k = 2^r$ ist können wir radixSort anwenden und den Satz beweisen.

Wie r wählen?

Falls $b < \log n$: $r = b$, andernfalls, $r = \log n \to \Theta\left(\frac{bn}{\log n}\right)$
Falls $b \in O(\log n) \to \Theta(n)$

Bucketsort

Wir wollen reelle Zahlen $\in [0, 1)$ sortieren
Grundidee:

- Das Intervall $[0, 1)$ wird in n gleichlange Teilintervalle $\left[\frac{j}{n}, \frac{j+1}{n}\right)$
  $(j = 0, 1, \ldots, n-1)$ partitioniert, denen Buckets $q[j]$ zugeordnet werden
- Für jedes $i = 0, 1, \ldots, n-1$ verfahren wir wie folgt:
  Wir legen den Schlüssel $A[i]$ genau dann in das Bucket $q[j]$, wenn

$$\frac{j}{n} \leq A[i] < \frac{j+1}{n} \iff j = \lfloor n \cdot A[i] \rfloor \$\$gilt Anschließend werden die Buckets zb mit Bucketsort/Heapsort sortiert und abschl$$

Rückspeicherung:
für $j = 0, 1, \ldots, n-1$:
speichere $q[j]$ auf $A\left[\sum_{i=0}^{j-1} \lambda_i, \sum_{i=0}^{j} \lambda_i\right]$, wobei $\lambda_i = |q[i]||$

Erwartete Laufzeit O(n)

hybridSort: mit heapsort die Bucketssortieren

B-Bäume

Grund def

Auf Hintergrundspeicher kännen Daten nur blockweise gelöscht oder überschrieben werden → um ein Byte zu ändern ganzen Block lesen/schreiben

Blöcke = Seiten
Wir studieren Anzahl von storage-read / storage-write
Platzbedarf wird in belegten Seiten gemessen

B-Bäume

- Datenstruktur im Hintergrundspeicher
- n = Anzahl gespeicherter Schlüssel im BBaum
- B-Baum T ist gerichteter geordneter Wurzelbaum
- Jeder Knoten x von T hat als Attribute
  - Eine geordnete Folge von Schlüsseln $k_1 < k_2 < \cdots < k_b$
  - Eine Folge succ(1), succ(2),...,succ(b+1) von Nachfolgerknoten, die entweder alle vorhander oder - gdw x ein Blatt ist - alle nicht vorhanden sind
  - b ist abhängig von x
- Alle Blätter von T haben dieselbe Tiefe
- $\exists t \geq 2 \in \mathbb{N}$ Verzweigungsfaktor oder Branching Factor, sodass für die ANzahl der Schlüssel b, die jeder Knoten x von T trägt gilt
  - $b \leq 2t$
  - Ist x von der Wurzel verschieden, so ist $b \geq t$
  - Ist T vom leeren Baum verschieden und ist x die Wurzel ist $b \geq 1$
  - Verallgemeinerte Suchbaumeigenschaft: Ist x beliebiger Knoten von T, der b Schlüssel $k_1 < k_2 < \cdots < k_b$ trägt und kein Blatt ist, und sind $\kappa_1, \kappa_2, \ldots \kappa_{b+1}$ beliebige Schlüssel der Nachfolgerknoten succ(1), succ(2), ..., bzw. succ(b+1), so ist

$$\kappa_1 < k_1 < \kappa_2 < k_2 < \cdots < \kappa_b < k_b < \kappa_{b+1}$$

Graphen Grunddef

Grunddef

- Adjazent = Benachbart
- Zwei Kanten inzident wenn sie genau einen Knoten gemeinsam haben
- $\sum_{v \in V} \text{degree}(V) = 2 \cdot |E|$
- → Anzahl der Knoten mit ungeradem grad ist gerade
- Teilgraph: Knoten rausnehmen
- Induzierter Teilgraph: alle Kanten behalten der Knoten
- azyklischer zusammenhängender Graph = (freier) Baum
- Ohne Zusammenhang Wald
- Gerichteter Graph heißt symmetrisch, falls G zu jeder Kante auch die entsprechende invertierte Kante enthält
- Eingangs/Ausgangsgrad eines Knotens (offensichtlich)

Lemmata

- Ist G kreisfrei, so gilt $|E| \leq |V| - 1$
- Ist G zusammenhängend so gilt $|E| \geq |V| - 1$

Lemmata alles Äquivalent

- G ist Baum
- Für je zwei Knoten $u, v \in V$ gibt es einen eindeutig bestimmten Weg in G, der u und v miteinander verbindet
- G ist ein minimaler zusammenhängender Graph: Das Entfernen einer beliebigen Kante führt zur Zerstörung des Graphes G
- G ist zusammenhängend und es gilt $|E| = |V| - 1$
- G ist kreisfrei und es gilt $|E| = |V| - 1$
- G ist ein maximaler kreisfreier Graph: Hinzufügen einer Kante führt dazu, dass ein Kreis entsteht

Ein aufspannender Baum existiert stets für einen zusammenhängenden Graphen G Bew (exemplarisch):
Entweder ist der Graph selbst ein Buam, oder aber er enthält mehr als $|V(G)| - 1$ Kanten. Dann enthält er aber einen Kreis, den wir durch Entfernen einer Kante aufbrechen können, ohne den Zusammenhang zu zerstören. Wir erhalten einen aufspannenden Teilgraphen T. Nun iterieren wir diesen Prozess. Da sich die Kantenzahl unseres Graphen in jeder Iteration um genau 1 verringert, gilt schließlich die GLeichung $|E(T)| = |V(T)| - 1$. Jetzt sind wir sicher, dass es sich bei T um einen Baum handelt, da wir den Zusammenhang steht erhalten haben. Da sich die Knotenmenge nicht verändert hat ist T auch ein aufspannender Baum von G.

Datenstrukturen

- Adjazenzmatrix: Eintrag von $A(G)$ in der i-ten Zeile und der $j-$ten Spalte ist genau dann 1, wenn (v_i, v_j) eine Kante in G ist, andernfalls ist der Eintrag 0 - gut für Graphen mit vielen Kanten
- Adjazenzliste: Für jeden Knoten v_i des Graphen wir eine Liste seiner Nachfolger gehalten - gut bei dünnen Graphen
- Kantenliste (edgelist) - Für jeden Knoten Edge objekte

Zusammenhangskomponenten

- Graph schwach zusammenhängend, falls der unterliegende Graph von G den man mittels Ersetzung aller gerichteten durch ungerichtete Kanten erhält zusammenhängend ist
- stark zusammenhängend, wenn je zwei seiner Knoten gegenseitig erreichbar sind

Breiten/Tiefensuche

Grunddef

- Edge entweder unclassifiedEdge, treeEdge oder otherEdge
- Vertex hat Datenfeld color
  - whie = Knoten war noch nicht in Queue/Stack
  - grey = Knoten gerade in Q/S
  - black = Knoten wurde bereits aus Q/S entfernt

Breitensuche:
Vorbed:

- Übergebener Anfangsknoten v_0 gehört zur Knotenmenge
- Graph ist zusammenhängend
- Alle Knoten von G haben die Farbe white, alle Kanten unclassifiedEdge
  Nachbed:
- Alle Knoten von G Farbe black
- Alle Kanten von G entweder als treeEdge oder otherEdge
- Teilgraph aus treeEdges (und allen Knoten) ist ein aufspannender Baum von G

```
bfs(Vertex $v_{0}$)

Erzeuge eine leere Queue x für Knoten

Färbe die Wurzel v0 mit grey ein
Füge v0 zur Queue x hinzu
while(!x.empty)
        v = x.top()
        x.pop()
        für alle zu v inzidenten, bisher unklassifizierten Kanten e:
                u = endknoten(e)
                if(color(u) == white)
                        e.classification = treeEdge
                        u.color = grey
                        x.add(u)
                else e.classification = treeEdge
        v.color = black
```

Tiefensuche:
Vorbed/Nachbed gleich wie bei BFS

```
x = stack(vertex)
v0.color = grey
x.add(v0)
while(!x.empty)
        v = x.top
        x.pop
        for e in v.edges:
                u = endknoten(e)
                if(u.color == white)
                        e.classification = treeEdge
                        u.color = grey
                        x.add(u)
                else e.classification = otherEdge
        v.color = black
```

Korrektheit

Klausel 1: Der Teilgraph T des Eingabegraphen, dessen Knotenmenge aus allen grauen und schwarzen Knoten besteht, und dessen Kanten die treeEdge-Kanten sind, ist ein Baum

Klausel 2: Die Queue/Stack enthält genau die grauen Knoten. Die weißen Knoten waren noch nie in Q/S, schwarzen bereits wieder verlassen

Invarianz von K2 offensichtlich: Jeder weiße Knoten der grau wird muss sofort in Q/S. Wenn ein Knoten entfernt wird, wird er black gefärbt.

Bew K1 Vollständige Induktion über Anzahl der Schleifendurchläufe λ:

IA $\lambda = 0$ nur Wurzel ist in Q/S und ist grau

IS $\lambda \to \lambda + 1$
Sei T_λ der in K1 angegebene Teilgraph nach dem λ-ten Schleifendurchlauf. Nach IV handelt es sich um einen Baum.
Im $\lambda + 1$ten Durchlauf kommen alle zum Spitzenknoten von Q/S, der ja als grauer Knoten bereits zu T_λ gehört, adjazenten weißen Knoten hinzu. Die jeweiligen Verbindungskanten werden Tree-Kanten. Damit ist klar, dass der Graph $T_{\lambda+1}$ der Graph aus K1 nach dem $\lambda + 1$ten Durchlauf der Schleife, zusammenhängend ist.
Nach IV ist T_λ Baum und es gilt $E(T_\lambda) = V(T_\lambda) - 1$. Im $\lambda + 1$ten Durchlauf der Schleife kommen genauso viele Knoten wie Kanten hinzu. Um klassifiezierungen finden nicht statt folglich gilt $E(T_{\lambda+1}) = V(T_{\lambda+1}) - 1$ und damit ist nach Lemma4 $T_{\lambda+1}$ Baum

Laufzeit:

Nach K2 kommt jeder Knoten genau einmal in Q/S
Folglich gibt es $|V|$ Schleifendurchläufe, von denen jeder offensichtlich $\Theta(1 + degree(v))$ Kosten verursacht wobei v derjenige Knoten ist, der während des Durchlaufs gerade besucht wird, und degree(v) den Grad des Knotens v bezeichnet
mit summe der degrees = 2 * E folgt dann die Laufzeit $\Theta(|E| + |V|)$

Um Zusammenhangskomponenten zu finden
Setze Datenfeld n_components des aktuellen Graphen auf 0
Durchlaufe mit Platzhalter alle Knoten v von G und:
falls v noch weiß ist

n_components++
führe bfs(v) bzw sdfs(v) aus

Tiefensuche rek:

```
dfsr(vertex v)
        forall e in v.edges
                u = endknoten(e)
                if u.color = white
                        e.classification = treeEdge
                        u.color = grey
                        dfsr(u)
                else e.classification = otherEdge
        v.color = black
```

Breiten/Tiefensuche gerichtete Graphen

- Vertex hat edges und zwei Felder X.d Zeitpunkt Entdeckung in der Tiefensuche, X.f Verarbeitung fertig
- globale Variable time

```
depthFirstSearch(Graph G = (V, E))
        time = 0
        for v : V
                 if v.color = white dfs(v)

dfs(vertex v)
        time++
        v.color = grey
        for e : v.edges:
                u = endknoten(e)
                if u.color = white
                        e.classification = treeEdge
                        dfs(u)
                else e.classification = otherEdge
        time++
        v.color = black
        v.f = time
```

Starke Zusammenhangskomponenten

```
stronglyConnectedComponents(Graph G = (V, E))
GS1:depthFirstSearch(G) //berechne x.f für alle Knoten
GS2:Berechne G^-1 (drehe edges um)
GS3:Führe depthFirstSearch(G^-1) aus aber betrachte in der Hauptschleife die Knoten in absteigender
Reihenfolge von x.f
 GS4: Die Knoten jedes DFS-Baums im DFS-Wald die in GS3 berechnet wurden werden seperat ausgegeben. DIese
Knoten bilden eine seperate starke Zusammenhangskomponente
```

Toposort

Def: Eine topologische Sortierung für G ist eine Knotenpermutation, sodass gilt: Ist $(v_i, v_j) \in E$, so ist $i < j$
Die induzierte Funktion ord : V $\rightarrow \{1, 2, \ldots, n\}, ord(v_i) = i$ welche jedem Knoten seine Orndungsnummer zuweist, wird auch topologische Ordnungsfunktion für G genannt

Es gibt genau dann eine topologische Sortierung für G wenn G azyklisch ist
G ist genau dann azyklisch, wenn G keine starke Zusammenhangskomponente der Größe ≥ 2 hat

```
topoSort(azyklischer Graph G = (V, E))
GS1: depthFirstSearch(G) //berechne x.f
GS2: Gebe Knoten in absteigender Reihenfolge von x.f aus
```

MST

Def

- Ein Schnitt in G = (V, E) ist eine nichttriviale Partition (X, V \ X) der Knotenmenge V von G
- Eine Kante e kreuzt den Schnitt wenn der eine Knoten von e zu X, der andere zu V\X gehört

Allgemein MST Algos färben Kanten grün (gehört zu MST) bzw rot (gehört nicht zu MST)
MST Färbbarkeit:

1. Eine bisher ungefärbte Kante $e \in E$ heißt gd grünfärbbar, wenn es einen Schnitt (X, V \ X) mit den folgenden Eigenschaften gibt
    1. Die Kante e kreuzt den Schnitt
    2. Keine grüne Kante kreuzt den Schnitt
    3. Unter allen ungefärbten Kanten die den Schnitt Kreuzen ist die Kante e von minimalem Gewicht
2. Eine bisher ungefärbte Kante e heißt genau dann rotfärbar, wenn es einen Kreis C in G gibt mit
    1. C enthält e
    2. C enthält keine rote Kante
    3. Unter allen ungefärbten Knaten die in C liegen ist e von maximalem gewicht
3. Bisher ungefärbte Kante heißt Färbbar wenn (1) oder (2)

Prim

Prim Grünfärbbarkeit im $t+1$ durchlauf d schleife gdw die Zusammenhangskomponente U_t mit einer anderen Zusammenhangskomponente W_t verbindet und unter allen Kanten mit dieser Eigenschaft min gewicht hat

Impl:

```
mstPrim(Vertex u)
        q = priorityqueue()
        u.color = grey
        for w : adjazeteKnoten(u)
                w.lightgreen = (u, w)
                w.priority = cost(u, w)
                w.color = blue
                q.add(w)
        while(Anzahl grüne Kanten != |V| -1)
                prim-greencoloringrule(u)

prim-greencoloringrule(vertex u):
        v = q.top(), q.remove(), v.color = grey
        e = v.lightgreen(), e.color = green

        for w : adjazenteKnoten(v):
                if(w.color = white):
                        w.lightgreen = (w, v)
                        w.priority = cost(w, v)
                if(w.color = blue)
                        e = (w, v)
                        if(e.cost < w.priority)
                                w.lightgreen = e
                                q.siftup(w, e.cost)
```

Laufzeit $O(|V| \cdot \log|V| + |E| \cdot \log|V|)$
Bew: Verwaltung Priorityqueue useless ist dominant was Laufzeit angeht
Jeder Knoten wird genau einmal in die Queue eingefügt und einmal herausgenommen: $O(|V| \cdot \log|V|)$
Es gibt höchstens soviele siftup Ops wie Kanten → $O(|E| \log|V|)$

Kruskal

GS1 Sortiere Kanten mit aufsteigendem weight
GS2 wende kruskals Färberegel an
Kruskalsfärberägel: Kante wird genau dann grüngefärbt, wenn sie zwei Zusammenhangskomponenten von W_t verbindet sonst rotgefärbt
Korrektheit bew:
Sei für $t = 0, 1, \ldots, m-1$

$$W_t := (V, \{e \mid e \text{ ist nach } t \text{ durchlaeufen gruengefaerbt}\})$$

der Grüne wald zu diesem Zeitpunkt

- Seien $T_1, T_2, \ldots, T_k$ die Zusammenhangskomponenten von W_t und sei $e_{t+1} = (a, b)$ die Kante aus GS2
  Fall 1: Die Knoten a und b gehören beide zu T_i für ein $i = 1, 2, \ldots, k$. Dann schließt die Kante e_{t+1} in $T_i \cup \{e_{t+1}\}$ einen Kreis, längsdessen sie

gemäß Def MST Färbbarkeit rotgefärbt werden kann

Fall 2: Die Knoten a und b gehören zu unterschiedlichen Bäumen T_i bzw T_j des Waldes W_t. Dann kreuzt die Kante e_{t+1} den Schnitt $(V(T_i), \bigcup_{k \neq i} V(T_k))$, längs dessen sie gemäß Def MST F. grüngefärbt werden kann.

Laufzeit ist in $O(|E| \cdot \log_2 |V|)$ Bew
GS1 ist in $O(|E| \log_2 |E|) = O(|E| \log_2 |V|)$
In GS2 werden zuerst $|V|$ makeSet Op und dann $|V| - 1$ union und $2|E|$ find Op durchgeführt
Wegen der Implementation von unionfind sicher dies, dass dies in der Zeit $O(|E| \cdot \log_2^* |V|)$ geschehen kann

Union find

- Ein Wald über dem Menge $V = \{1, 2, \ldots, n\}$ der durch eine Folge zulässiger makeset und union op auf den anfänglichen Aufruf von empty(n) erzeugt wurde heißt unkomprimiert
- Für einen Knoten v eines Wurzelbaumes T ist height$_T v$ die Länge des längsten Weges vom einem Blatt von T zum Knoten v
- Sei W unkomprimierter Wald und x ein Knoten aus W dann ist $size(x) \geq 2^{height(x)}$
-

empty(int n) - erzeuge ein Feld parent vom Typ int mit Index bereich 1, 2, ..., n

makeset(int x) - parent[x] = -1

union(int x, int y) - vorbed x != y repräsentanten
falls |parent(x)| < |parent(y)|:
parent(y) = parent(y) + parent(x)
parent(x) = y
andernfalls:
parent(x) = parent(x) + parent(y)
parent(y) = x

simpleFind(int x)
if parent(x) < 0 return x
sonst return simpleFind(parent(x))
Wegen Oben Laufzeit $O(\log_2 n)$

find(int x)
if parent(x) < 0 return x
sonst
root = find(parent(x))
parent(x) = root
return root

DP (Knapsack Varianten)

DP für maxKnapsack
Eingabe Problemstellung $I := (w_1, w_2, \ldots, w_n, c_1, c_2, \ldots, c_n, W)$
GS1 Erzeuge $Opt[0, \ldots, n, 0, \ldots, W]$
$Opt[k, 0] = 0, k \in [0, n]$
$Opt[0, V] = 0, V \in [1, W]$
$\beta = (0, 0, \ldots, 0)$
GS2 for $k \in 1, 2, \ldots, n$
for $V \in 1, 2, \ldots W$
Falls $V \geq w_k$ dann $m = Opt[k-1, V - w_k] + c_k$
Sonst $m = 0$
$Opt[k, V] = \max\{Opt[k-1, V]\}$
GS3: $V = W$
for $k \in n, n-1, \ldots, 1$:
Falls $Opt[k, V] > Opt[k-1, V] : \beta_k = 1, V = V - w_k$

return β

Laufzeit: $O(n \cdot W)$ im Einheitskostenmaß wobei n die Anzahl der Gegenstände und W das zulässige Gesamtgewicht des Rucksacks ist.
$O(nW) = O(n2^{bin|W|})$
→ Der Algo wird polynomiell, wenn Gesamtgewicht W durch ein Polynom in der binären Eingabelänge der Instanz beschränkt ist.

Fractional Knapsack:

Zulässige Lösungen sind Vektoren wobei $0 \leq \beta_i \leq 1$
GS1 Nummeriere die n Gegenstände so um, dass danach $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n}$
GS2 $R = W$ (Restgewicht)
$i = 0$ (Index des letzen visitierten Gegenstandes
$\beta = (0, 0, \ldots, 0)$ (Befüllung des Rucksacks)

Solange $R > 0$ und $i < n$ führe aus:
$i++$
Falls $w_i \leq R$ so
$\beta_i = 1$
$R = R - w_i$
Andernfalls
$\beta_i = \frac{R}{w_i}$
$R = 0$
return β

MaxSimpleKnapsack

Weights = cost (wenn etwas 2 wiegt ist cost 2)
BasicGreedyKnapsack$_k$ → max k Gegenstände
Mit Güte $1 + \frac{1}{k}, \in O(n^{k+1}|I|)$

GS1:Sortiere die n Gegenstände s.d.
$w_1 \geq w_2 \geq \cdots \geq w_n$
GS2:
$opt = 0$ (Wert der besten bisherigen Befüllung des Rucksacks)
$\beta = (0, 0, \ldots, 0)$
Erzeuge ein

- Feld $\tau \left[1 \ldots \sum_{v=1}^{k} \binom{n}{v}\right]$ von booleschen Vektoren der Länge n
- Feld $\gamma \left[1 \ldots \sum_{v=1}^{k} \binom{n}{v}\right]$ von natürlichen Zahlen
  und initialisiere diese Felder wie folgt:
  τ enthält alle Teilmengen von $\{1, \ldots, n\}$ mit höchstens k Elementen als charakteristische Vektoren
- Feld $\tau \left[1 \ldots \sum_{v=1}^{k} \binom{n}{v}\right]$ von booleschen Vektoren der Länge n
  Für alle $j = 1, 2, \ldots, \sum$ ist $\gamma[j] = \sum_{i=1}^{n} \tau[j][i] w_i$
  GS3:
  Für $j = 1, 2, \ldots, \sum$ führe aus:
  $R = W - \gamma[j]$
  $i = 0$
  solange $R > 0$ und $i < n$ führe aus:

1. $i++$
2. Falls $w_i \leq R$ und $\tau[j][i] = 0$, so
   1. $\tau[j][i] = 1$
   2. $R = R - w_i$
   3. $\gamma[j] = \gamma[j] + w_i$
   Falls $\gamma[j] > opt$:
3. $opt = \gamma[j]$
4. $\beta = \tau[j]$
   return β

Backtracking

Die allgemeine Situation
Sei $n \in \mathbb{N}$ und seien $M_1 = \{a_1 < a_2 < \cdots < a_\alpha\}, M_2 = \{b_1 < b_2 < \cdots < b_\beta\}, \ldots, M_n = \{z_1 < z_2 < \ldots z_\omega\}$
total geordnete endliche Menge derart, dass für jede zulässige Eingabe $I \in Input\Pi$ der "Größe" n
$SolI \subset M(n) = M_1 \times M_2 \times \cdots \times M_n$ ist
Die Menge Sol I muss durch einen Polynomialzeitalgorithmus in $M(n)$ identifizierbar sein

Der Backtrack-Baum zur Eingaben der Größe n des Problem Π ist ein gerichteter geordneter Wurzelbaum:

- Menge der Knoten V = root $\bigcup_{r=1}^{n} M_1 \times \cdots \times M_r$
- Wurzel ist mit demen Knoten aus M_1 durch eine Kante verbunden
- Für jedes $r = 1, 2, \ldots n-1$ ist jeder Knoten $(m_1, \ldots, m_r) \in M_1 \times \cdots \times M_r$ für jedes $m_{r+1} \in M_{r+1}$ mit $(m_1, \ldots, m_r, m_{r+1}) \in M_1 \times \cdots \times M_r \times M_{r+1}$ durch eine Kante verbunden
- Für $r = 1, 2, \ldots n-1$ ergibt sich die Anordnung der Kinder $(m_1, \ldots, m_r, m_{r+1}), m_{r+1} \in M_{r+1}$ des Knotens $(m_1, \ldots, m_r)$ kanonisch aus der Ordnung der Elemente der Menge M_{r+1}
- Die Blätter des Backtrack-Baumse sind alle Elemente aus $M_1 \times \cdots \times M_n$

Jeder Weg im Backtrack Baum von der Wurzel zu einem Blatt entspricht für jede Eingabe I der Größe n umkehrbar eindeutig dem Aufgbau einer potentiellen Lösung für I von links nach rechts. Ob diese Lösung auch zulässig ist, hängt von der akutellen Eingabe I ab.

Auf der Suche nach einer opt. Lsg durchlaufen wir die Knoten des Backtrackbaumes zur Problemgröße n in Vorordnung.
Trick zur Beschränkung der Laufzeit: Für möglichst viele Knoten u die kein Blatt sind nach der Inspektion von u den in u wurzelnden TB T_u zu entfernen. Dazu gibt es zwei Gründe

1. Kein Blatt von T_u ist eine zulässige Lösung

2. Der Baum T_u hat zwar Blätter, die zulässigen Lösungen entsprechen, aber die zugehörigen Werte liegen unterhalb einer uns zu diesem Zeitpunkt bereits bekannten unteren Schranke für opt. lsg
   → Wir brauchen Variable optSol für bisher beste Lsg und Variable globalLower für untere Schranke für Opt(I) (opt. Lsg von I)

Zum abschneiden pruning($m_1, \ldots, m_r$) returns bool
Vorbed: $u = (m_1, \ldots, m_r)$ ist ein Knoten, aber kein Blatt des Backtrack Baumes zur aktuellen Eingabe I der Größe n
Nachbed: Wird true zurückgegeben, so gilt für alle $m_{r+1} \in M_{r+1}, \ldots, m_n \in M_n$

- Es ist $v = (m_1, \ldots, m_r, m_{r+1}, \ldots, m_n) \notin Sol(I)$, oder
- falls v zulässig ist, so ist $Val(l, v) < globalLower$
  Impl:

```
backtracking(u)
Falls u Blatt mit u in Sol(I) ist, und Val(I, u) >= globalLower gilt:
        optSol = u
        globalLower = Val(I, u)
        return
If(pruning(u) == false):
        for all m_r+1 in M_r+1:
                backtracking(m1, ..., mr, mr+1)
        return
```

pruning für Knapsack:

```
pruning2(m1, ..., mr)
if |=P(m1, m2, ..., mr) return true //Eingabe unzulässig
if(m1, m2, ..., mr) < globalLower return true
globalLower = max{globalLower, lower(m1, m2, ..., mr)}
return false
```

NP Vollständigkeit

Gibt es für zwei durch ihre formalen Sprachen L_1 und L_2 dargestellten Entscheidungsprobleme eine in deterministischer Polynomialzeit berechenbare Transformation f der Eingaben I_1 des Problems auf Eingaben $f(I_1)$ des zweiten Problems, sodass

$$I_1 \in L_1 \iff f(I_1) \in L_2$$

so kann man bezogen auf PolynomialzeitAlgos sagen, das Problem L_1 sei nicht schwerer als das Problem L_2
Schwerste Probleme nennt man NP-Vollständig

Formal: Sei $f : \{0,1\}^* \to \{0,1\}^*$ eine vermäge eines TT berechenbare Funktion. Sprache L_1 heißt längs der FUnktion f auf die Sprache L_2 reduzierbar, wenn für alle Wörter $w \in \{0,1\}^*$ die Äquivalenz

$$w \in L_1 \iff f(w) \in L_2$$

gilt
$f \in FP$ → Polynomialzeitreduzierbar
$f \in FL$ → logspace-reduzierbar

- Logspace reduzierbarkeit ⇒ Polynomialzeitreduzierbarkeit
- Beide Reduzierbarkeitsrelationen sind transitiv (Da FL und FP abgeschlossen gegenüber Verkettung)
- Formale Sprache L_1 ist auf eine formale Sprache reduzierbar in polynomialer Zeit, falls es eine Funktion $f \in FP$ gibt längs derer L_1 auf L_2 reduzierbar ist ($L_1 \leq L_2$)

Sei $\leq$ eine transitive Reduzierbarkeitsrelation für formale Sprachen und C eine Komplexitätsklasse formaler Sprachen, die bzgl $\leq$ abgeschlossen ist

1. Eine formale Sprache L heißt C-hart bzgl $\leq$, wenn für jedes L' aus C gilt $L' \leq L$
2. Eine formale Sprache L heißt $C-$vollständig bzgl $\leq$ wenn L bzgl $\leq C-$hart ist und überdies L zu C gehört

Alle NP-Vollständigkeitsbeweise:

1. NP-Algorithmus der das Problem löst entwerfen
2. NP-Härte durch Reduktion zeigen

SAT

Grunddef

- Boolesche Variablen und Negationen heißen Literale
- Literal y widerspricht Literal z wenn es eine Variable x_i gibt sd $\{y, z\} = \{x_i, \neg x_i\}$
- Klausel ist Disjunktion von Literalen
- Monom ist Konjunktion von Literalen
- Konjuntive Form ist Konjunktion von Klauseln, wobei keine Klausel doppelt auftritt
- disjunktive Form ist Disjunktion von Monomen, wobei kein Monom doppelt auftritt
- formale Sprache SAT ist die Menge aller erfüllbaren konjunktiven Formen
- Cook Levin: Sat ist NP-Vollständig

Varianten:

- Konjunktive Form $F = \{K_i \mid 1, 2, \ldots, r\}$ ist genau dann erfüllbar, wenn es eine Funktion

$$\alpha : F = \{K_i \mid 1, 2, \ldots, r\} \to \{x_1, x_2, \ldots, x_n, \bar{x_1}, \bar{x_2}, \ldots, \bar{x_n}\}$$

  derart gibt (widerspruchsfreie Auswahlfkt), dass

- für jedes $i = 1, 2, \ldots, r$ das Literal $\alpha(K_i)$ ein Literal der Klausel K_i ist

- für alle $i \neq j$ die Literale $\alpha(K_i)$ und $\alpha(K_j)$ nicht widersprechen

  Die konjunktive Form F heißt genau dann (exakte) $k - CF$ wenn für jedes $i = 1, 2, \ldots, r$ gilt: Die Klausel K_i besteht aus (genau) k paarweise verschiedenen Literalen, die einander nicht widersprechen

3SAT <= SAT
Eingabe: eine konjunktive Form $F = \{K_i \mid i = 1, 2, \ldots, r\}$
Ausgabe: eine exakte 3-CF, erfüllbar gdw F erfüllbar

GS1:

- Streiche aus jeder Klausel mehrfach vorkommende Literale bis auf einen Repräsentanten
- Streiche jede Klausel, in der eine Variable und deren Negation vorkommt ($x \vee \bar{x}$ immer erfüllt)
  GS2(Klauseln Länge 1 entfernen)
  Grundidee: Solche Klauseln können nur erfüllt werden, wenn man das Literal eins setzt, diesen Wert setzen wir nun ein
  Für jede Klausel $K' \in F$ die davon betroffen ist gibt es zwei Fälle

1. Entweder ist K' dadurch erfüllt, dann kann sie gestrichen werden. Ist F danach leer war Eingabeform erfüllbar und wir geben eine erfüllbare exakte $3 - CF$ aus
2. Oder es ist nicht der Fall, dann kommt $\bar{y}$ in K' vor und es muss ein anderes Literal true sein, um K' zu erfüllen. $\bar{y}$ wird aus K' entfernt, ist K' nun leer ist eine Erfüllung nicht mehr möglich > wir geben eine unerfüllbare exakte 3-CF
   GS3 (Klauseln Länge 2 ersetzen)
   Für jede Klausel $K \in F$:
   Ist $K = y_1 \vee y_2 (y_1, y_2 \text{ Literale})$, so ersetze K durch die Klauseln $(y_1 \vee y_2 \vee z_k)$ und $(y_1 \vee y_2 \vee \bar{z}_k)$

Klauseln Länge ≥ 4
Für jedes $(y_1 \vee y_2 \vee \cdots \vee y_k) \in F$:
Ersetze $y_1 \vee y_2 \vee \cdots \vee y_k$ durch:
$(y_1 \vee y_2 \vee z_{K,1})$
$(\bar{z}_{K,1} \vee y_3 \vee z_{K,2})$
...
$(\bar{z}_{K,i} \vee y_{i+2} \vee z_{K,i+1})$
...
$(\bar{z}_{K,k-3} \vee y_{k-1} \vee y_k)$

2SAT NL-vollständig

- Zunächst wissen wir, dass die Disjunktion $y_1 \vee y_2$ von zwei Literalen logisch sowohl zu der Implikation $\bar{y_1} \to y_2$ als auch zu der Implikation $\bar{y_2} \to y_1$ äquivalent ist
- Ist F eine 2-CF über $\{x_1, x_2, \ldots, x_m\}$ so erstzen wir zunächst jede Klausel von F die nur aus einem Literal y besteht durch $y \vee y$
- Nun ordnen wir dem so modifizierten F den folgenden gerichteten Graphen $G(F)$ zu.
- Knotenmenge von $G(F)$ ist $\{x_1, x_2, \ldots, x_m, \neg x_1, \neg x_2, \ldots, \neg x_m\}$ zwei Knoten y_1 und y_2 sind genau dann durch eine Kante $y_1 \to y_2$ verbunden, wenn $\bar{y_1} \vee y_2$ zur Klauselmenge von F gehört: Ist $y_1 = 1$, so muss auch $y_2 = 1$ sein, damit F unter $y_1 = 1$ noch erfüllt werden kann
- Daraus folgt, dass die Existenz eines gerichteten Weges von y_1 nach y_2 in G(F) bedeutet, dass unter $y_1 = 1$ Formel F nur dann erfüllt werden kann, wenn auch $y_2 = 1$ ist. Ist insbesondere $y_2 = \neg y_1$ so folgt in dieser Situation, dass die Belegung $y_1 = 1$ nicht zu einer erfüllenden Belegung aller Variablen aus $\{x_1, x_2, \ldots, x_m\}$ erweitert werden kann
- Wir erhalten: Gibt es eine Variable x_i, so dass es in G(F) sowohl einen Weg von x_i nach $\neg x_i$ als auch einen gerichteten Weg von $\neg x_i$ nach x_i gibt, so ist F nicht erfüllbar
- Um zu zeigen, dass 2SAT $\in NL$ entwerfe nichtdet. log. speicherbeschränkten algo
- Vollständigkeit durch Ähnlichkeit mit GAP

MaxkSat NP vollständig

Input k-CF F und $c \in \mathbb{N}$
Akzeptiere gdw es einen Booleschen Vektor gibt, der mindestens c Klauseln erfüllt
MAX3SAT ist NP-vollständig: Die Funktion $F \mapsto (F, |F|)$ ist offenbar FP-Reduktion von 3SAT auf MAX3SAT

Dreidimensionales Matching

H, B, G endliche disjunkte Mengen gleicher Mächtigkeit q und $P \subset H \times B \times G$ dreistellige Relation (P heißt Präferenztripelmenge)
Teilrelation $M = \{(h_i, b_i, g_i \mid 1, 2, \dots, q) \subset P\}$ heißt dreidimensionales Matching mit $H = \{h_1, h_2, \dots, h_q\}, B = \dots, G = \dots$
→ M ist dreidimensionales Matching gdw jedes $h \in H, b \in B, g \in G$ durch ein Tripel aus M überdeckt wird

Reduktion 3SATexakt auf 3DM (ewig langer Beweis)

Partitionsproblem

Eingabe: Paar (A, s) - endliche Indexmenge A, $s : A \to \mathbb{N}$
Ausgabe: Akzeptiere gdw $\exists A' \subset A$, sodass $\sum_{a \in A'} s(a) = \sum_{a \notin A'} s(a)$
Existiert A' nennt man dies partitionierende Teilmenge von A
NP-Vollständigkeit Bew. durch Reduktion 3Dm → Partition (ewig langer Beweis)

Knapsack

Wir reduzieren Partition auf Knapsack. Sei (A, s) eine Instanz von Partition, o.B.d.A $A = \{1, 2, \dots, n\}$
GS1: Ist $\sum_{i=1}^{n} s(i)$ ungerade gib unlösbare Instanz von Knapsack aus
GS2: Berechne Instanz $I(A, s) := \left(w_1 = c_1 = s(1), \dots, w_n = c_n = s(n), W = C = \frac{1}{2} \cdot \sum_{i=1}^{n} s(i)\right)$

Clique

$G = (V, E)$ ungerichtet, $V' \subset V$ heißt

- Clique wenn durch V' induzierter Teilgraph vollständiger Teilgraph ist
- unabhängige Menge (IS), wenn der durch V' in G induzierte Teilgraph keine Kanten enthält
- Vertex Cover (VC), wenn jede Kante e aus E zu einem Knoten V' indiziert
  Komplementärer Graph $\bar{G} = (V, \bar{E})$, $(u, v) \in \bar{E} \iff (u, v) \notin E$

Lemmata

- V' ist genau dann Clique in G, wenn V' im komplementären Graphen $\bar{G}$ unabhängige Menge ist
- V' ist genau dann eine Clique in G, wenn $V \setminus V'$ in $\bar{G}$ Vertex Cover ist

Entscheidungsprobleme, Eingabe immer Graph G, $c \in \mathbb{N}, c \leq |V|$

1. Clique: Akzeptiere $\iff$ G eine Clique V' mit $|V'| \geq c$
2. IS: Akzeptiere $\iff$ G eine IS V' mit $|V'| \geq c$
3. Vertex Cover: Akzeptiere $\iff$ G VC Clique V' mit $|V'| \leq c$

3SATexakt → Clique
$F = \{K_1, K_2, \dots, K_r\}$ wobei für $j = 1, 2, \dots, r$ die Klausel $K_j = a_j \vee b_j \vee c_j$ ist
Wir konstruieren in PZ Graphen $G_F := (V_F, E_F)$
Mit $V_F := \{(K_j, a_j), (K_j, b_j), (K_j, c_j) \mid j = 1, 2, \dots, r\}$
$\{(K, d), (K', d')\} \in E_F \iff K \neq K' \vee d' \neq d$

Es gilt

- G_F nur Cliquen der Mächtigkeit kleiner oder gleich r haben kann
- jede widerspruchsfreie Auswahlfunktion α zu einer r-Clique $\{(K, \alpha(K)) \mid K \in F\}$
- jede r-Clique als Graph einer widerspruchsfreien Auswahlfunktion angesehen werden kann
  qed
  → IS & VC Np vollständig

Hamiltonscher Kreis enthält jeden Knoten aus Graphen genau einmal - Finden NP-Vollständig

TSP

Eingabe $\gamma : \{1, 2, \dots, n\}^2 \to \mathbb{N}$ mit $\gamma(i, i) = 0$ und $c \in \mathbb{N}$
Ausgabe: Akzeptiere $\iff$ $\exists$ Permutaion π von $\{1, 2, \dots, n\}$ sodass
$\gamma(\pi) := \gamma(\pi(n), \pi(1)) + \sum_{i=1}^{n-1} \gamma(\pi(i), \pi(i+1)) \leq c$

Man sieht gleich, dass $G = (\{1, 2, \dots, n\}, E) \mapsto (\gamma_G, 0)$
wobei $\gamma_G : \{1, 2, \dots, n\}^2 \to \mathbb{N}$ wobei für $i \neq j$
$\gamma_G(i, j) = 0$ falls $(i, j) \in E$, 1 falls nicht enthalten Reduktion von Hamitlon auf TSP ist