$$\underset{\text{übertrag }\Delta}{\approx} \quad \bigwedge_x fxy \neq z \vee \neg \underbrace{\bigvee_u \bigvee_w (mzw \neq x \vee fvy \neq u)}$$

$$\underset{\text{Beh 1}}{\approx} \quad \bigwedge_u \bigwedge_w \underbrace{\neg (mzw \neq x \vee fvy \neq u)}_{\equiv\, mzw = x \,\wedge\, fvy = u}$$

$$\approx \quad \bigwedge_x fxy \neq z \vee \bigwedge_u \bigwedge_w (mzw = x \wedge fvy = u)$$

umbenennen, weil hier noch x steht

$$\underset{\substack{\text{Beh 3}\\ t \text{ neue Var}}}{\approx} \quad \bigwedge_t fty \neq z \vee \bigwedge_u \bigwedge_w (mzw = x \wedge fvy = u)$$

$$\underset{\text{Beh 2}}{\approx} \quad \bigwedge_t \left(fty \neq z \vee \bigwedge_u \bigwedge_w (mzw = x \wedge fvy = u) \right)$$

$$\underset{\text{Beh 2}}{\approx} \quad \bigwedge_t \bigwedge_u \bigwedge_w \left(fty \neq z \vee (mzw = x \wedge fvy = u) \right)$$

4.6 Einige Ausblicke

4.6.1 Unentscheidbarkeit von $\vdash$, $\models$ im PL

Eine naheliegende Frage lautet:

Wie kann man testen, ob ein Ausdruck $\alpha \in PL(S, V)$ gültig ($\Leftrightarrow$ herleitbar) ist?

In der Aussagenlogik genügte es, die Wahrheitstafel aufzustellen; das war etwas mühsam, aber immerhin möglich.

Für die Ausdrücke der Prädikatenlogik existiert kein solches Verfahren! Wir können hier keinen exakten Beweis für diese Tatsache geben, wollen aber doch die Hintergründe beleuchten.

<u>Fakt 1</u> Es gibt kein Verfahren, mit dem man entscheiden könnte, ob ein beliebiges Programm jemals anhält. (vgl. S. 126)

<u>Beweisidee</u>

Es gibt Programme M, die andere Programme als Eingabe bekommen (z.B. Compiler).
Was passiert, wenn solch ein Programm sich selbst als Eingabe bekommt?

<u>Beh:</u> Das ist unentscheidbar!
<u>Bew:</u> Angenommen, es gäbe ein "Superprogramm" S, das diese Frage entscheiden könnte, d.h.

$$\forall M: \quad S(M) = \begin{cases} 1, & \text{falls } M(M) \text{ hält nicht an} \\ 0, & \text{falls } M(M) \text{ hält an} \end{cases}$$

Programm M, gestartet mit Eingabe M

Jetzt erweitern wir Programm S zu folgendem Programm $\tilde{S}$:

S;
addiere Ausgabe von S zu sich selbst,
bis 2 herauskommt.

Offenbar gilt:

$$\forall M: \quad \tilde{S}(M) \text{ hält an} \iff S(M)=1$$
$$\iff M(M) \text{ hält nicht an.}$$

Wenn wir $M := \tilde{S}$ setzen, ergibt sich der Widerspruch

(Diagonalisierung, vgl. S. 122)

$$\tilde{S}(\tilde{S}) \text{ hält an } \Longleftrightarrow S(\tilde{S}) = 1$$
$$\Longleftrightarrow \tilde{S}(\tilde{S}) \text{ hält nicht an.}$$

Also war die Annahme falsch, daß ein Superprogramm S existiert, welches das Problem

"Hält $M(M)$ an?"

entscheidet, d.h., dieses Problem ist unentscheidbar.

$\boxed{\text{Beh}}$ $\boxed{\text{Fakt 1}}$

<u>Bemerkung</u> Um aus dieser Idee einen präzisen Beweis zu gewinnen, muß man präzise sagen, was ein Programm ist bzw. was eine Maschine ist, die ein Programm ausführt.

Wenn man das getan hat, kann man zu einer Maschine mit Programm M und Eingabe E einen Ausdruck $\alpha_{M,E} \in PL(\mathcal{S}, V)$ angeben, der die Rechenschritte von M bei Bearbeitung von E beschreibt und folgende Eigenschaft hat:

$$\alpha_{M,E} \text{ ist erfüllbar} \Longleftrightarrow M(E) \text{ hält an.}$$

Weil das Halteproblem unentscheidbar ist, muß das Erfüllbarkeitsproblem auch unentscheidbar sein.

Wegen α gültig $\iff$ $\neg\alpha$ nicht erfüllbar
ist das Gültigkeitsproblem ebenfalls unentscheidbar.
Wir notieren

Fakt 2 Es gibt kein Verfahren, mit dem man entscheiden könnte, ob ein beliebiger Ausdruck der Prädikatenlogik
- erfüllbar ist
- gültig ist
- im Kalkül hergeleitet werden kann.

Bemerkung Man kann wohl ein Verfahren angeben, das nach und nach alle gültigen Ausdrücke aufzählt:
Dazu müssen nur systematisch alle möglichen Herleitungen im PL-Kalkül erzeugt werden.
Wenn ein Ausdruck α gültig, also herleitbar ist, kommt er irgendwann an die Reihe.
Wenn α nicht gültig ist, kommt α nie an die Reihe —
aber weil man nicht weiß, wie lange man probieren muß, ist dies kein endliches Entscheidungsverfahren!

Wir halten fest:

Der Kalkül der Prädikatenlogik ist
- korrekt
- vollständig
- unentscheidbar
- erlaubt es nicht, Endlichkeit zu charakterisieren

4.6.2 Prädikatenlogik 2. Stufe

<u>Bisher</u>
- Quantoren nur über Individuen-Variablen $x, y, z, \ldots$
- Funktionen und Prädikate kamen nur als Konstante vor.

= Prädikatenlogik erster Stufe.

<u>Natürliche Frage</u> Was passiert wenn man Quantoren auch über Funktionen und Prädikaten verwenden darf?

<u>Beispiel</u>

$f : P \to Q$

$$\alpha_{P,Q} :\equiv \bigvee_f \Big(\bigwedge_x (Px \to Qfx)$$

f injektiv

$$\wedge \bigwedge_{x_1} \bigwedge_{x_2} (fx_1 = fx_2 \to x_1 = x_2)$$

f surjektiv

$$\wedge \bigwedge_y \big(Qy \to \bigvee_x (Px \wedge fx = y)\big) \Big)$$

hier ist f Funktionsvariable, P und Q sind Prädikatenvariablen.

<u>Offenbar gilt:</u>

$\mathfrak{I}(\alpha_{P,Q}) = W$

$\iff$ es gibt bijektive Abbildung

$$f : \mathfrak{I}(P) \longrightarrow \mathfrak{I}(Q)$$

$\iff$ $\mathfrak{I}(P)$ und $\mathfrak{I}(Q)$ sind gleichmächtig.

die Teilmengen von A_S, welche den Variablen P und Q zugeordnet werden

Jetzt definieren wir

$$\beta_{P,Q} :\equiv \bigwedge_x (Px \to Qx)$$

und bekommen

$$\mathcal{I}(\beta_{P,Q}) = W \iff \mathcal{I}(P) \subseteq \mathcal{I}(Q).$$

Setzen wir also

$$\gamma :\equiv \bigwedge_P \bigwedge_Q (\beta_{P,Q} \wedge \alpha_{P,Q} \to \beta_{Q,P}),$$

folgt:

γ in $\mathcal{A}$ erfüllbar

$$\iff \left(\forall p, q \subseteq A_s : \quad p \subseteq q \text{ und } p, q \text{ gleichmächtig} \Rightarrow q \subseteq p\right)$$

$$\iff \quad A_s \text{ ist endlich.}$$

Fakt 3
Mit Ausdrücken 2. Stufe (= PL(2)) läßt sich also die Endlichkeit einer Struktur beschreiben; das geht mit Ausdrücken der 1. Stufe nicht.

"Dafür" gibt es aber für die Prädikatenlogik 2. Stufe keinen korrekten und vollständigen Kalkül !

Ein weiterer Unterschied:
Fakt 4
In PL(2) kann man $\mathbb{N}$ bis auf Isomorphie beschreiben, in PL(1) geht das nicht.

4.6.3 Unvollständigkeit

Theorem 18 und 19 besagen:

Was in jeder Struktur $\mathcal{A}$ aus einer Menge M folgt, ist im Kalkül aus M ableitbar, und umgekehrt.

In der Mathematik interessiert man sich besonders für alle Aussagen, die in bestimmten Strukturen oder in bestimmten Klassen von Strukturen gelten.

Beispiele:

Zahlentheorie: Struktur = $(\mathbb{N}, +, \cdot, 0, 1) =: \mathcal{N}$

Gruppentheorie: Strukturen = alle Gruppen $(G, \circ, 1)$

Auf David Hilbert geht die Frage zurück, ob man z.B. die Zahlentheorie "axiomatisieren" kann, das heißt, ob es eine Menge M (= Axiomensystem) von Ausdrücken mit folgender Eigenschaft gibt:

(i) M enthält die Peano-Axiome und ist entscheidbar

[(ii) Alle $\gamma \in M$ sind in $\mathcal{N}$ gültig] (folgt aus (iii))

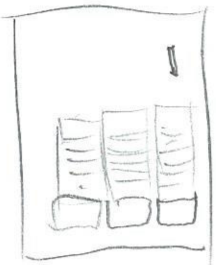
(iii) $\forall \alpha: (M \vdash \alpha \iff \alpha$ gültig in $\mathcal{N})$

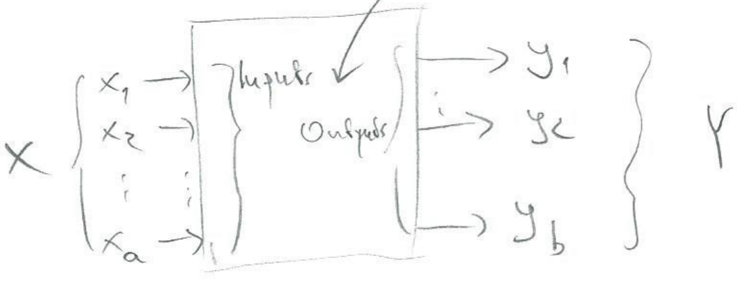
(d.h., $\models_{\mathcal{N}} \alpha$ S. 180)

Kurt Gödel hat gezeigt, daß das nicht geht!

Maschinen

Zustände $\underbrace{z_1,\ldots,z_n}_{Z}$ <u>endlich viele</u>, deshalb:





$\boxed{\text{Endlicher Automat}}$

In Abhängigkeit vom aktuellen Zustand z_j bewirkt Input x_i

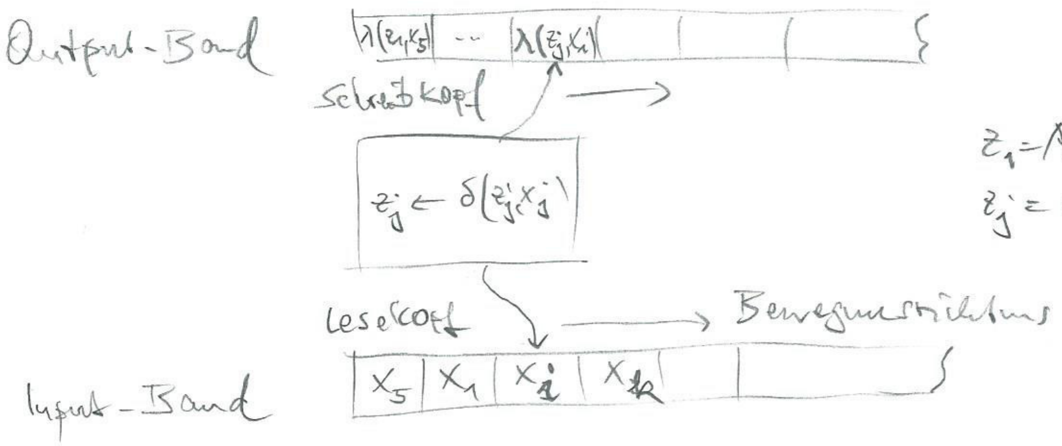
- Zustandsänderung $\quad z_j \leftarrow \delta(z_j, x_i)$
- Output $\quad \lambda(z_j, x_i)$

$$\left.\begin{aligned} \delta &: Z \times X \longrightarrow Z \\ \lambda &: Z \times X \longrightarrow Y \end{aligned}\right\}$$ durch Tabellen darstellbar = Automatentafeln

δ	x_1	x_2	$\ldots$	x_i	$\ldots$
z_1					
z_2					
$\vdots$					
z_j				$\delta(z_j, x_i)$	
$\vdots$					

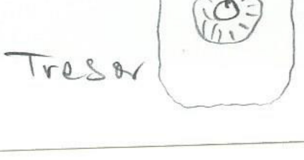
Input / Output: Knöpfe, Schalter, Akustische / Optische Signale

oder (für theoretische Zwecke)



z_1 = Anfangszustand
z_j = aktueller Zustand

Manche endlichen Automaten benötigen keine Outputs. Man interessiert sich dafür, wann sie einen "ausgezeichneten" Zustand erreichen $\longrightarrow$ Akzeptor



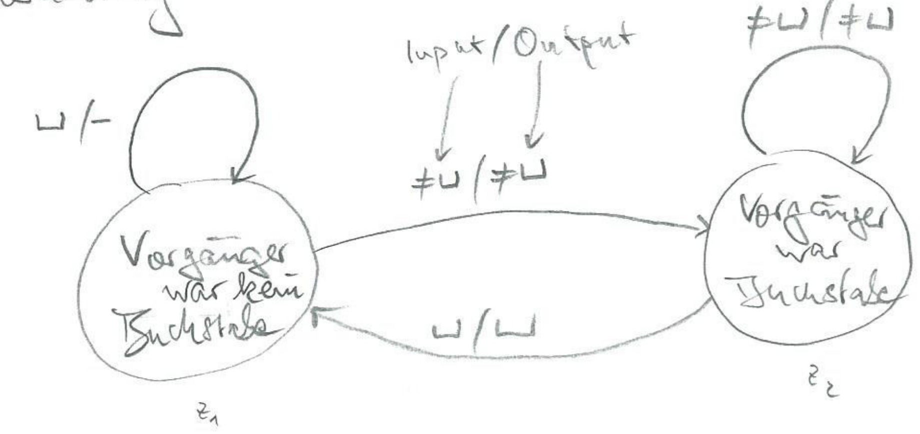
Beispiel überflüssige Blanks eliminieren
- vor Textbeginn
- mehrfache Blanks zwischen Wörtern
  nach Textende

Bsp: ␣␣␣ BLANKS␣␣ ELIMINIEREN␣␣␣

$\Big\downarrow$

BLANKS␣ ELIMINIEREN␣

Beobachtung: Blank bleibt stehen $\Leftrightarrow$ Vorgänger war Buchstabe (generisch: b)



δ	␣	≠␣
z_1	z_1	z_2
z_2	z_1	z_2

λ	␣	≠␣
z_1	–	≠␣
z_2	␣	≠␣

Aus solchen (einfachen) Beschreibungen lassen sich endliche Automaten automatisch generieren.

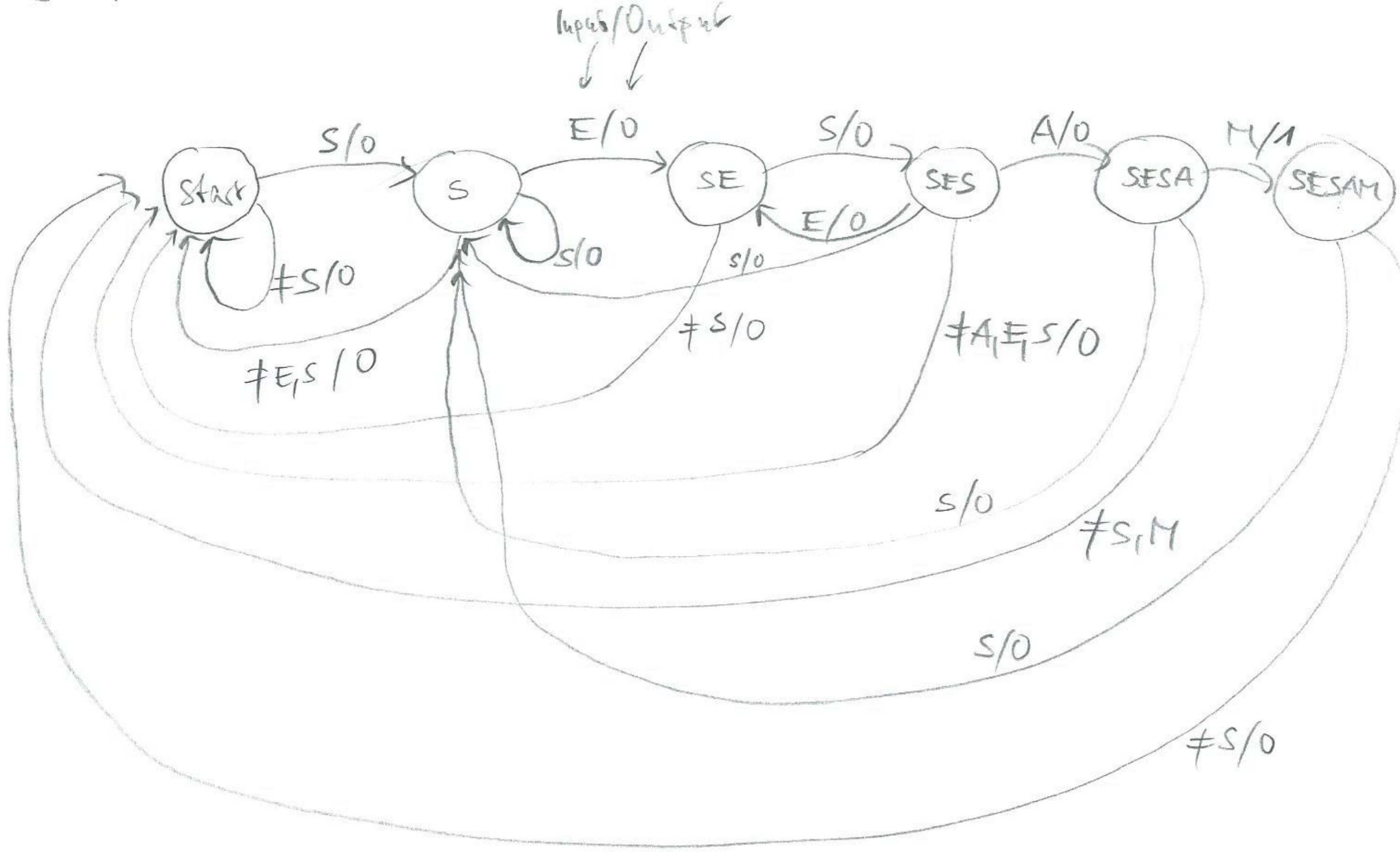
Beispiel :

soll bei Eingabe von SESAM Lampe aufleuchten lassen

als Teilwort eines Strings

Output: 0 oder 1
Input : A, B, ..., Z

statt Automatentafel : Übergangsdiagramm



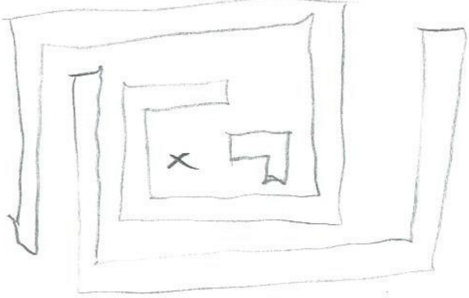
[statt Output 0/1 zu erzeugen: SESAM ist akzeptierender Zustand]

Aus Übergangsdiagramm kann man leicht Automatentafeln berechnen.

Man sieht: In Zuständen lässt sich Information speichern — aber nur <u>endlich viel</u>.

Endliche Automaten können nicht:

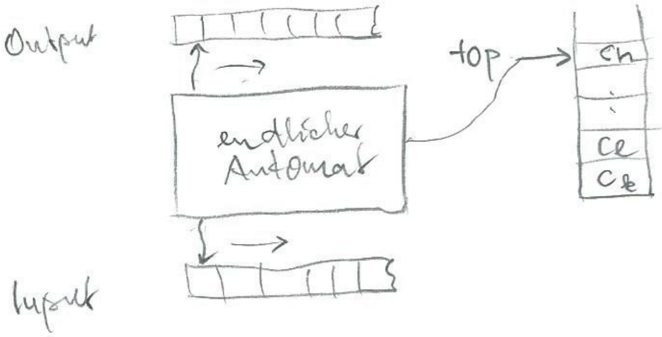
– Klammerausdrücke auf Korrektheit prüfen $\underbrace{(((}_{\text{zählen!}}\ldots)))$
– aus Labyrinthen herausfinden



sie können aber "reguläre Sprachen" erkennen
Anwendung: lexikalische Analyse von Programmen

... dazu braucht man mehr Speicher: (unbeschränkt)

Kellerautomat / Stapelautomat



Stack (Stapel) wie bei Tellern
man kann nur
– oben ein Objekt "drauflegen" push
– oberstes Objekt entfernen und anschauen POP

kann "kontextfreie" Sprachen erkennen;
dazu zählen große Teile der Definitionen gängiger Programmiersprachen

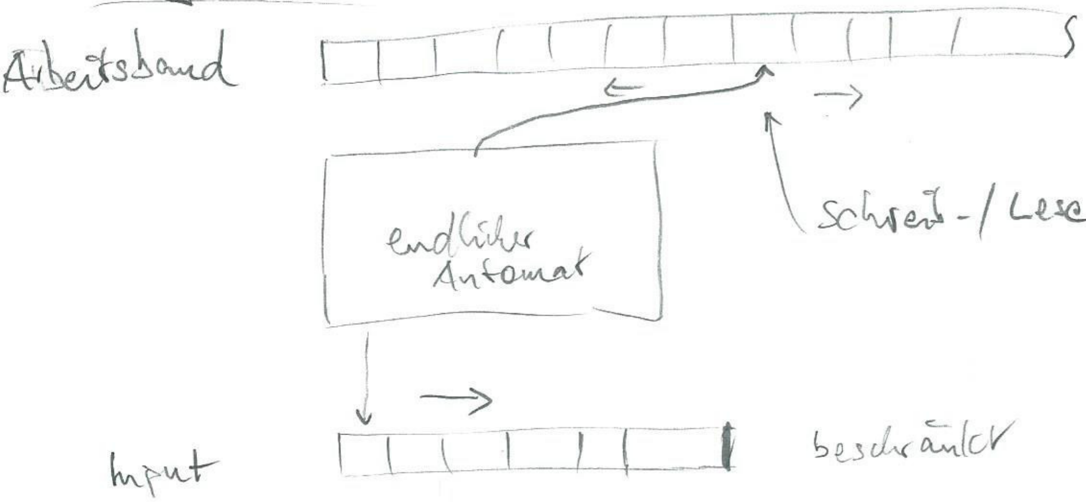
Bsp: Klammerausdrücke: push (, wenn gelesen
pop (, wenn) gelesen.

Labyrinthe: genügt: endlicher Automat + unbegrenzter Speicher für eine Zahl $\in \mathbb{Z}$ mit Zugriff lesen, +1, −1
→ Alg. Geom I

Stapelautomaten können nicht erkennen, ob ein Wort über $\{a,b,c\}$ in $\{a^n b^n c^n \mid n \geq 1\}$ liegt.

dazu braucht man besseren Zugriff auf den unbeschränkten Speicher:

Turing-Maschine

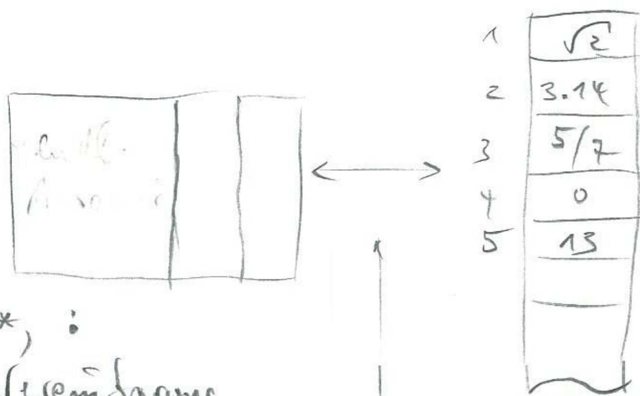


erinnert an Magnetbandspeicher

Church'sche These Was man überhaupt berechnen kann, läßt sich mit einer Turing-Maschine berechnen.

(bewiesen für zahlreiche alternative Maschinenmodelle, z.B. RM, und andere Formalisierungen von Berechenbarkeit)

„Laufzeit" von Algorithmen:
Anzahl der Schritte einer idealisierten Maschine.
Oft verwendet man statt der TM die Real-RAM:



unendlich viele Speicherzellen, mit 1, 2, 3, ... indiziert.
Jedes speichert eine reelle Zahl

+, −, *, :
je nach Vereinbarung
$\lfloor x \rfloor$, $\sin x$, $\sqrt{x}$

LOAD, STORE(i)

LOAD $(\underbrace{c(5)}_{13})$

Wir werden nun ein Beispiel für eine Anwendung der Sweep-Technik kennenlernen, das ein klein wenig mehr Überlegung erfordert, aber dafür um so verblüffender ist.

2.2.3 Die maximale Teilsumme

Aktienkurs

Angenommen, jemand führt über die Kursschwankungen einer Aktie Buch. Das Auf oder Ab am i-ten Tag wird als reelle Zahl an der i-ten Stelle eines Arrays *Variation* gespeichert, für $1 \leq i \leq n$.

Eine interessante Frage lautet: Wieviel hätte man verdienen können, wenn man diese Aktie am richtigen Tag gekauft und am richtigen Tag wieder verkauft hätte? Der maximale Gewinn wird durch die maximale Summe

$$Variation[i] + Variation[i+1] + \ldots + Variation[j]$$

beschrieben, die sich ergibt, wenn i von 1 bis n und j von i bis n laufen, falls wenigstens eine solche Summe nicht-negativ ist; wenn dagegen eine Aktie stets fällt, kauft man sie am besten gar nicht und macht den Gewinn Null.

Ein Beispiel ist in Abbildung 2.1 zu sehen. Vom dritten bis zum siebten Tag steigt der Kurs der Aktie insgesamt um den Wert 10. Durch Ausprobieren der anderen Möglichkeiten kann man sich davon überzeugen, daß es keine lückenlose Teilfolge von Zahlen gibt, die eine größere Summe hätte. So lohnt es sich zum Beispiel nicht, die Aktie auch am achten Tag noch zu behalten, weil der Verlust von 9 später nicht mehr ausgeglichen wird.

Summe = 10

2	-4	3	-1	4	-1	5	-9	2	-6	5	-3	5
↑ 1		↑ 3				↑ 7						↑ n

Abb. 2.1 Die maximale Teilsumme aufeinanderfolgender Zahlen hat hier den Wert 10.

maximum subvector

Wie läßt sich die maximale Teilsumme effizient berechnen? Dieses hübsche Problem ist unter dem Namen *maximum subvector* bekannt. Es findet sich in der lesenswerten Sammlung *Programming Pearls* von J. Bentley [15].

kubisch

quadratisch

Die naive Berechnung der maximalen Teilsumme führt zu einem Algorithmus mit drei geschachtelten **for**-Schleifen für i, j und einen Summationsindex k; seine Laufzeit ist in $\Theta(n^3)$. Er läßt sich zu einem $\Theta(n^2)$-Verfahren verbessern, indem man nach jeder Erhöhung von j nicht die ganze Summe von i bis $j+1$ neu berechnet, sondern lediglich zur alten Summe den neuen Summanden *Variation*[j+1] hinzuaddiert. Auch ein Ansatz nach dem Verfahren *divide and conquer* ist möglich. Er liefert eine Laufzeit in $\Theta(n \log n)$.

linear

Man kann aber die gesuchte maximale Teilsumme sogar in optimaler Zeit $\Theta(n)$ berechnen, wenn man das Sweep-Verfahren richtig anwendet! Die Objekte sind die Einträge

$$Variation[1], Variation[2], \ldots, Variation[n],$$

die wir in dieser Reihenfolge besuchen. In der Variablen $MaxSoFar_{j-1}$ merken wir uns die bisher ermittelte maximale Teilsumme der ersten $j-1$ Zahlen.

Wodurch könnte $MaxSoFar_{j-1}$ übertroffen werden, wenn im nächsten Schritt nun auch *Variation*[j] besucht wird? Doch nur von einer Teilsumme der ersten j Einträge, *die den j-ten Eintrag einschließt!* Diese Situation ist in Abbildung 2.2 dargestellt. Die bisherige maximale Teilsumme der Positionen 1 bis $j-1$ hat den Wert 10; gebildet wird sie von den Einträgen an den Positionen 4 bis 7. Dieses alte Maximum wird nun „entthront" von der Summe der Einträge an den Positionen $j-5$ bis j, die den Wert 11 hat. 9

? könnte -20 sein

-1	2	-3	5	-2	-1	8	-2	0	1	-9	3	-1	3	-4	8	2
↑ 1			↑ 4	MaxSoFar 10		↑ 7					↑ $j-5$	MaxEndingHere				↑ j

bisher besucht

Abb. 2.2 Die bisher ermittelte maximale Teilsumme *MaxSoFar* kann nur von einer Teilsumme übertroffen werden, die sich bis zum neu hinzukommenden Eintrag *Variation*[j] erstreckt.

mitzuführende Information

Um die Variable *MaxSoFar* in dieser Situation korrekt aktualisieren zu können, müssen wir während des *sweep* mehr Information mitführen als nur *MaxSoFar* selbst. Wir benötigen auch die maximale Teilsumme der schon betrachteten Einträge, die in dem zuletzt neu hinzugekommenen Eintrag endet. Mit $MaxEndingHere_j$ bezeichnen wir das Maximum aller Summen

$$Variation[h] + Variation[h+1] + \ldots + Variation[j],$$

wobei h zwischen 1 und j läuft, falls wenigstens eine dieser Summen nicht-negativ ist; andernfalls soll $MaxEndingHere_j$ den Wert Null haben. Im Beispiel von Abbildung 2.2 ist $MaxEndingHere_j$ die Summe der letzten sechs Einträge. Dagegen wäre $MaxEndingHere_3 = 0$.

209

Wir müssen uns nun noch überlegen, wie *MaxEndingHere* aktualisiert werden soll, wenn der *sweep* von Position $j-1$ zur Position j vorrückt. Dabei hilft uns folgende einfache Beobachtung: Für festes j wird das Maximum aller Summen

$$Variation[h] + Variation[h+1] + \ldots + Variation[j-1] + Variation[j],$$

für dasselbe $h \leq j-1$ angenommen, das auch schon die Summe

$$Variation[h] + Variation[h+1] + \ldots + Variation[j-1]$$

maximiert hatte, es sei denn, es gibt keine positive Teilsumme, die in Position $j-1$ endet, und $MaxEndingHere_{j-1}$ hatte den Wert Null. In beiden Fällen gilt

$$MaxEndingHere_j = \max(0, MaxEndingHere_{j-1} + Variation[j]).$$

Damit haben wir folgenden Sweep-Algorithmus zur Berechnung der maximalen Teilsumme:

```
MaxSoFar := 0;
MaxEndingHere := 0;
for j := 1 to n do
    MaxEndingHere :=
                max(0, MaxEndingHere + Variation[j]);
    MaxSoFar := max(MaxSoFar, MaxEndingHere);
write("Die maximale Teilsumme beträgt ", MaxSoFar)
```

Theorem 2.2 *In einer Folge von n Zahlen die maximale Teilsumme konsekutiver Zahlen zu bestimmen, hat die Zeitkomplexität $\Theta(n)$.*

schnell und elegant

Dieser Algorithmus ist nicht nur der schnellste unter allen Mitbewerbern, sondern auch der kürzeste! Darüber hinaus benötigt er keinen wahlfreien Zugriff auf das ganze Array *Variation*, denn die Kursschwankung des j-ten Tages wird ja nur einmal angeschaut und danach nie wieder benötigt.

Man könnte deshalb dieses Verfahren als nicht-terminierenden Algorithmus implementieren, der am j-ten Tag die Schwankung *Variation*$[j]$ als Eingabe erhält und als Antwort den bis jetzt maximal erzielbaren Gewinn ausgibt; schade, daß dieser erst im nachhinein bekannt wird. Bei diesem *on-line-Betrieb* wären der Speicherplatzbedarf und die tägliche Antwortzeit konstant.

on-line-Betrieb

Übungsaufgabe 2.2 Man formuliere den oben vorgestellten Sweep-Algorithmus zur Bestimmung der maximalen Teilsumme als *on-line*-Algorithmus.

Der Sweep-Algorithmus zur Bestimmung der maximalen Teilsumme wird uns später in einem überraschenden Zusammenhang gute Dienste leisten: bei der Berechnung des *Kerns* eines Polygons, der Menge aller Punkte im Polygon, von denen aus das ganze Polygon sichtbar ist.

Kern

2.3 Sweep in der Ebene

Ihre wahre Stärke zeigt die Sweep-Technik in der Ebene. Die Objekte, die in unseren Problemen vorkommen, werden zunächst Punkte sein, später Liniensegmente und Kurven. Wir besuchen sie in der Reihenfolge, in der sie von einer senkrechten Geraden angetroffen werden, die von links nach rechts über die Ebene wandert.

Die Vorstellung dieser wandernden Geraden (*sweep line*), die die Ebene von links nach rechts „ausfegt“ und dabei keine Stelle ausläßt, hat dem Sweep-Verfahren seinen Namen gegeben.

sweep line

2.3.1 Das dichteste Punktepaar in der Ebene

Wir beginnen mit dem Problem *closest pair*, das uns aus dem Eindimensionalen schon bekannt ist. Gegeben sind n Punkte $p_1, \ldots, p_n$ in der Ebene, gesucht ist ein Paar mit minimalem euklidischem Abstand. Wir sind zunächst etwas bescheidener und bestimmen nur den minimalen Abstand

closest pair

$$\min_{1 \leq i < j \leq n} |p_i p_j|$$

selbst. Der Algorithmus läßt sich später leicht so erweitern, daß er auch ein dichtestes Paar ausgibt, bei dem dieser Abstand auftritt.

Erinnern wir uns an unser Vorgehen im Eindimensionalen: Die Punkte waren dort Zahlen auf der X-Achse. Wir hatten sie von links nach rechts besucht und dabei für jede Zahl den Abstand zu ihrer unmittelbaren Vorgängerin betrachtet. War er kleiner als *MinSoFar*, der bisher ermittelte minimale Abstand, mußte *MinSoFar* aktualisiert werden.

In der Ebene können wir nicht ganz so einfach verfahren. Wenn wir mit der *sweep line* auf einen neuen Punkt r treffen, kann der Abstand zu dessen direktem Vorgänger q, d. h. zu dem Punkt, der als letzter vor r von der *sweep line* erreicht wurde, viel größer sein als der Abstand von r zu noch weiter links liegenden Punkten; siehe Abbildung 2.3.

Unterschied zum Eindimensionalen

Eines aber ist klar: Wenn r mit einem Punkt p links von r ein Paar bilden will, dessen Abstand kleiner ist als *MinSoFar*, so

20

Gesuchte: $\max\limits_{1 \le i \le k \le n} \left(\sum_{j=i}^{k} A[j] \, , \, 0 \right)$

naiv:

```
MaxSoFar := 0 ;
for i := 1 to n do
    for k := i to n do
        sum := 0 ;
        for j := i to k do
            sum := sum + A[j] ;
        MaxSoFar := max(MaxSoFar, sum)
write(MaxSoFar)
```

$\Theta(n^3)$

besser:

```
MaxSoFar := 0 ;
for i := 1 to n do
    sum := 0 ;
    for j := i to n do
        sum := sum + A[j] ;
        MaxSoFar := max (MaxSoFar, sum)
write (MaxSoFar)
```

$\Theta(n^2)$

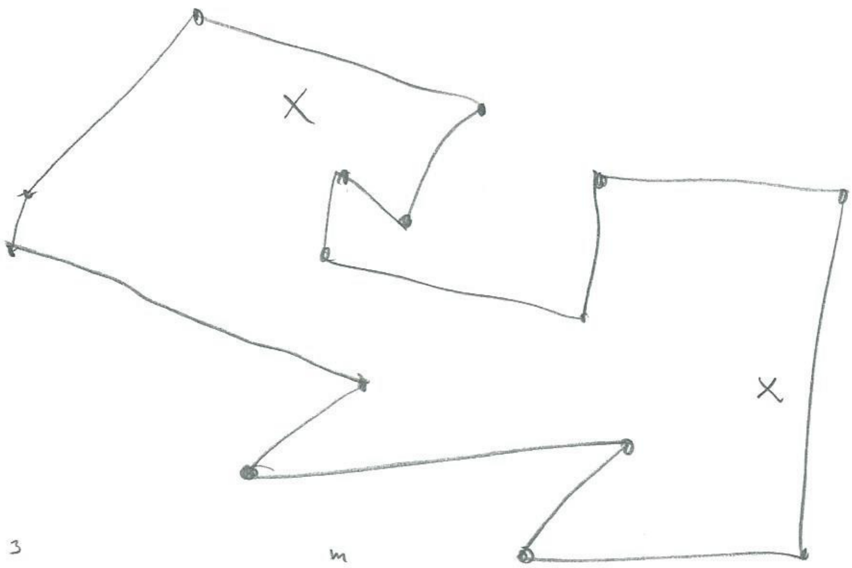
optimal:

$\Theta(n)$

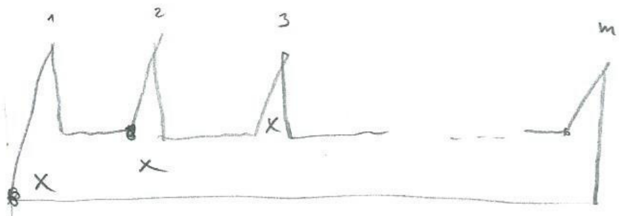
```
MaxSoFar := 0;
MaxEndingHere := 0;
for j := 1 to n do
    MaxEndingHere := max(MaxEndingHere + A[j], 0);
    MaxSoFar := max(MaxSoFar, MaxEndingHere)
write(MaxSoFar)
```

Art Gallery Problem

Grundriß = einfaches Polygon mit n Ecken, P



wieviele Wächter benötigt? (stationär, 360° Sicht)



benötigt m Wächter, hat 3m Kanten

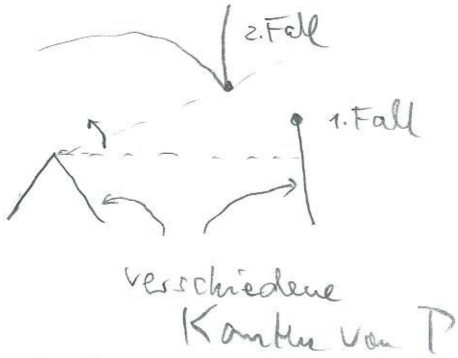
Chvatal $\lfloor \frac{n}{3} \rfloor$ Wächter sind immer ausreichend und manchmal erforderlich

Beweis

Diagonale von P = Strecke, die 2 Ecken verbindet und sonst ganz im Inneren liegt

Lemma P kein Dreieck $\Rightarrow$ es gibt Diagonale

Bew: P konvex: klar. Sei P nicht konvex $\Rightarrow$ es gibt spitze Ecke



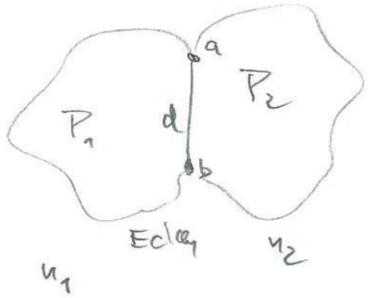
$\square$

Wenn man also kreuzungsfreie Diagonalen einzeichnet, solange es geht, bleiben nur Dreiecke übrig — P ist trianguliert.

Es gibt viele Triangulationen von P, aber

Lemma Jede Triangulation von P hat $n-2$ Dreiecke.

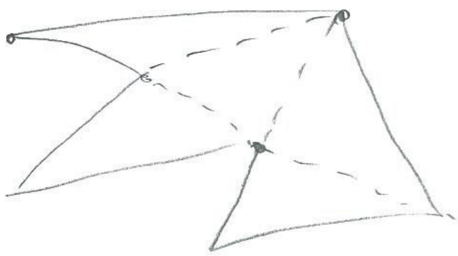
Bew Ind(n). $n=3$ ✓ $n>3$: Sei d Diagonale einer Triang.

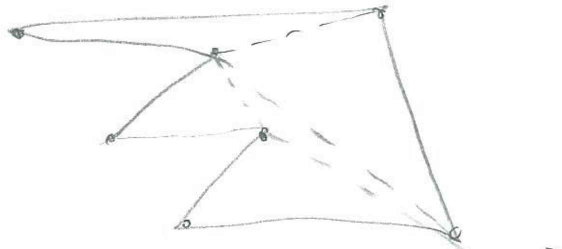


$n_1 + n_2 = n+2$ (a, b doppelt gezählt)

Ind.Vor: P_1 hat $n_1 - 2$ Dreiecke
P_2 " $n_2 - 2$ "

$\Rightarrow P$ hat $n_1 - 2 + n_2 - 2 = n-2$ Dreiecke. □



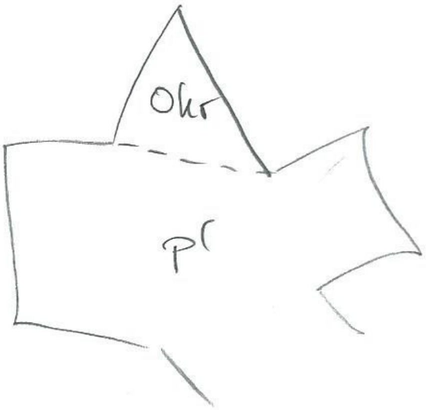


Sei T eine Triang. von P.
Wollen jetzt die Ecken von P so färben, dass gilt:
die Endpunkte einer jeden Kante erhalten unterschiedliche Farben. Frage: Wieviele Farben braucht man?

Lemma 3 Farben genügen.
Bew Ind(n). $n=3$:

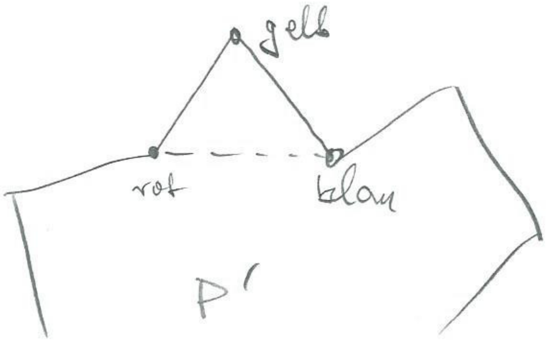
$n>3$: f: Kante von P $\longrightarrow$ anliegendes Dreieck
kann nicht injektiv sein, weil: n Kanten, $n-2$ Dreiecke.

$\Rightarrow$ es gibt ein Dreieck mit 2 Außenkanten in P, ein "Ohr".



Abschneiden! Verbleibendes Polygon P' hat $n-1$ Ecken

$\Rightarrow$ P' mit 3 Farben färbbar.



Jetzt Ohr wieder annähen:

$\Rightarrow$ Färbung von P mit 3 Farben.

$\square$

Sei "rot" die Farbe, die am wenigsten häufig vorkommt

$\Rightarrow$ $\leq \left\lfloor \frac{n}{3} \right\rfloor$ viele Ecken sind rot

Dort stellen wir die Wächter auf.

Dann ist ganz P überwacht, denn:

- in jedem Dreieck von T müssen alle drei Farben vorkommen; also gibt es eine rote Ecke.
- Von ihr aus ist das ganze Dreieck sichtbar.

$\square$