stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IN 23/83 MEI

P.M.B. VITÁNYI

AUTOMATENTHEORIE, COMPLEXITEIT EN ALGORITMEN

Automata theory, complexity and algorithms.

kruislaan 413 1098 SJ amsterdam

AUTOMATENTHEORIE, COMPLEXITEIT EN ALGORITMEN*

Paul M.B. Vitányi

SAMENVATTING

In kort bestek worden de voornaamste begrippen en resultaten uit de automatentheorie, de leer der formele talen en de complexiteit van algoritmen, samengevat. Een en ander voorzien van enige historische kanttekeningen, verband met concrete vraagstukken en gelardeerd met voorbeelden.

* Te verschijnen als Sectie 3 "Automatentheorie" en Sectie 4 "Complexiteit van Algoritmen" van Hoofdstuk III "Kern Informatica" in het "Automatiserings Zakboekje", uitgave Koninklijke PBNA, Arnhem.

Inleiding

In opdracht van de Koninklijke PBNA werden van het, door deze organisatie uit te geven, "Automatiserings Zakboekje" de onderdelen "Automaten Theorie" en "Complexiteit van Algoritmen" verzorgt. Een dergelijke uitgave zal deels als inleiding, deels als naslagwerk, worden gebruikt. Van belang is daarom enerzijds een logische opbouw van het materiaal, van basis begrippen naar meer gecompliceerde begrippen, geïllustreerd met voorbeelden en verwijzingen naar concrete toepassingen en breder wetenschappelijk belang, anderzijds veel preciese definities en uitspraken over wetmatigheden. Gezien de beperkte toegestane omvang van het geschrift, ging deze aanpak ten koste van de betrokken bewijzen. Gestreeft werd om tòch een inzicht te geven waarom het een en ander het geval is, en de gebruikte redenatiemethoden (summier) toe te lichten. Noodzakelijkerwijze moest een selectie gemaakt worden uit de vele belangrijke onderwerpen en resultaten die voor opname in aanmerking kwamen. Bijvoorbeeld, parallelle algoritmen en architecturen, die door de opkomst van VLSI technologie (very large scale integrated circuits) bijzonder belangrijk worden, zijn in het geheel niet vermeld. Dit, omdat de practische- en theoretische begripsvorming op dit gebied aan dusdanige slingeringen onderhevig is, dat van uitgekristalliseerde principes niet gesproken kan worden, en opname in een compendium als het onderhavige voorbarig lijkt. Wat betreft de vastliggende theorie werd gekozen voor opname van de onderdelen die, naar mijn inzicht, het meest nuttig voor de niet-specialistische gebruiker zijn. Bijvoorbeeld, terwijl de context vrije grammatica's en -talen uitgebreid behandeld worden, komen de context gevoelige grammatica's en -talen vrijwel niet aan de orde.

Trefwoorden: Automaten, formele talen, complexiteit, algoritmen.

Uit: Automatiserings Zakboekje, Hoofdstuk III (Kern Informatica). Koninklijke PBNA, Arnhem.

3. Automatentheorie.

Traditioneel wordt onder een automaat een min of meer zelfstandige machine verstaan, zoals een sigarettenautomaat, koffie-automaat, of speelautomaat. Nog vroeger werd het woord voornamelijk gebruikt voor ingenieus geconstrueerd, mechanisch aangedreven speelgoed, dat vaak muziek voortbracht. In de Wiskunde en Informatica wordt onder een automaat een abstractie van een informatie- verwerkende machine verstaan. We onderscheiden Eindige Toestands Automaten en Oneindige Toestands Automaten. Automaten hebben *invoer* en *uitvoer*. Deze grootheden bestaan uit *symboolrijen* (ook *woorden*) over een eindig *alfabet* van *symbolen*. Bijvoorbeeld, $\{a,b,c\}$ is een alfabet met symbolen a, b en c, en $acba$ is een symboolrij. De *lengte* $|w|$, van een symboolrij w, is het aantal symbolen in w. Dus $|acba| = 4$. De *lege* symboolrij ϵ is de rij die uit nul symbolen bestaat. Dus $|\epsilon| = 0$. De *concatenatie* van twee symboolrijen is de symboolrij die wordt gevormd door de tweede symboolrij achter de eerste symboolrij te schrijven, zonder tussenruimte. Als x en y symboolrijen zijn dan is xy de concatenatie van die twee. De lege symboolrij ϵ is het identiteitselement voor de concatenatie-operator: $\epsilon w = w\epsilon = w$ voor alle symboolrijen w. Een (formele) *taal* is een verzameling symboolrijen van symbolen uit een of ander alfabet. De verzameling van alle symboolrijen over een gegeven alfabet I wordt geschreven als I^*. Als $I = \{0,1\}$ dan geldt $I^* = \{\epsilon,0,1,00,10,01,11,000,\cdots\}$. I^* is een halfgroep (ook: semigroep) met als operatie concatenatie en als eenheidselement ϵ. We zeggen dat I^* de *vrije* halfgroep op zijn *voortbrengenden*, de elementen van I, is.

3.1. Eindige Toestands Automaten.

De *Eindige Toestands Automaat*, ook wel *Eindige Automaat* of *EA*, is een wiskundig model van een systeem met discrete invoer en uitvoer. Het systeem kan in een eindig aantal interne configuraties of *toestanden* zijn. De toestand van het systeem bevat de informatie over de verwerkte invoer die nodig is om het gedrag van het systeem met betrekking tot de toekomstige invoer te bepalen. In de Informatica vinden we veel voorbeelden van systemen met een eindig aantal toestanden, en de Eindige Automaten vormen een nuttig ontwerpgereedschap voor deze systemen. (Schakelnetwerken in een computer zijn met opzet zo ontworpen, dat ze als eindige toestands systemen kunnen worden beschouwd. Dientengevolge kan het logische ontwerp van een computer gescheiden worden van de electronische implementatie. Ook vaak voorkomende programma's als tekstverwerkers en lexicale analysatoren in compilers worden veelal ontworpen als eindige toestands systemen.) De toestanden van een EA gaan in elkaar over volgens een gegeven verzameling *transities* die gebeuren onder invloed van *invoersymbolen*, gekozen uit een eindig *invoeralfabet* I. Voor ieder invoer symbool is er precies één transitie uit elke toestand (mogelijk terug naar die toestand zelf). Eén toestand vormt de *begintoestand* van de automaat. Sommige toestanden zijn aangewezen als *eind-* of *accepterende* toestanden.

Een *transitiediagram* van een EA is een gerichte graaf. De knopen van de graaf corresponderen met de toestanden van de EA. Als er een transitie is van toestand p naar toestand q onder invoer a, dan is er een met a gemerkte pijl van toestand p naar toestand q in het transitie diagram. De EA *accepteert* een *woord* x, als de rij van opeenvolgende transities, corresponderende met de opeenvolgende symbolen van x, van de begintoestand van de EA leidt naar een accepterende toestand.

Formeel wordt een EA gegeven door een vijftal $A = (Q,I,\delta,q_0,F)$ zodat:

- Q is een eindige niet-lege verzameling toestanden;
- I is een eindig invoeralfabet van invoersymbolen;
- δ is een transitie functie die $Q \times I$ in Q afbeeldt. D.w.z., voor elke toestand q en invoersymbool a is $\delta(q,a)$ een toestand.
- q_0 is een element van Q: de begintoestand.
- F is een deelverzameling van Q: de verzameling van eindtoestanden.

Om het gedrag van een EA op woorden te beschrijven, breiden we de transitie functie δ uit tot een functie δ' die betrekking heeft op toestanden en woorden. De functie δ' beeldt $Q \times I^*$ in Q af, zodat $\delta'(p,w)$ de toestand van de EA is, na het verwerken van het woord w, indien de EA gestart wordt in toestand p. Anders gezegt, $\delta'(p,w)$ is de unieke toestand q zodat er in het transitie diagram een met w gemerkt pad is van p naar q. Als we met ϵ het *lege* woord bestaande uit nul symbolen aangeven, dan wordt δ' inductief gedefinieerd door:

1) $\delta'(p,\epsilon) = p$, en

2) voor alle woorden w en invoer symbolen a geldt: $\delta'(p,wa) = \delta(\delta'(p,w),a)$.

Zonder het verwerken van een invoer symbool kan de EA dus niet van toestand veranderen. Daar $\delta'(p,a) = \delta(\delta'(p,\epsilon),a) = \delta(p,a)$, stemmen δ en δ' overeen op de argumenten waarvoor zij beide gedefinieerd zijn. Voor het gemak gebruiken we verder δ voor δ'.

Een woord x wordt door een eindige automaat $A = (Q,I,\delta,q_0,F)$ *geaccepteerd* als $\delta(q_0,x) = p$ voor een p in F. De *taal* die door A geaccepteerd wordt, aangeduidt door $L(A)$, is de verzameling $\{x \mid \delta(q_0,x) \in F\}$. Een taal is *regulier* als zij de verzameling woorden is die door een eindige automaat geaccepteerd wordt.

Niet-deterministische eindige automaten. De *niet-deterministsche EA* ontstaat door in de (deterministische) EA hierboven (verder aangeduid als DEA) nul, één of méér transities toe te laten vanuit eenzelfde toestand en invoer symbool. Dit nieuwe model zullen we afkorten als *NEA*. Een NEA is gedefinieerd als een vijftal $A = (Q,I,\delta,q_0,F)$ waarbij Q,I,q_0 en F dezelfde betekenis hebben als voor de eerder gedefinieerde DEA, maar δ een afbeelding is van $Q \times I$ in de machtsverzameling $P(Q)$ van Q, d.w.z. de verzameling van alle deelverzamelingen van Q. Het transitiediagram van een NEA wordt gedefinieerd als dat van een DEA, behalve dat nu $\delta(p,a)$ de verzameling van alle toestanden q is, zodanig dat er een met a gemerkte transitie van p naar q is. Net als bij de DEA breiden we de functie δ uit tot een functie δ', van $Q \times I^*$ in $P(Q)$, die de rijen invoer symbolen als volgt verwerkt:

1) $\delta'(p,\epsilon) = \{p\}$,

2) $\delta'(p,wa) = \{q \mid r \in \delta'(p,w) \ \& \ q \in \delta(r,a)\}$.

Merk op dat weer $\delta'(p,a) = \delta(p,a)$ voor een invoersymbool a. We gebruiken weer δ in plaats van δ'. δ wordt uitgebreid tot argumenten uit $P(Q) \times I^*$ door:

3) $\delta(P,w) = \bigcup_{q \in P} \delta(q,w)$,

voor iedere verzameling toestanden P bevat in Q. De *taal* $L(A)$, die door een NEA $A = (Q,I,\delta,q_0,F)$ geaccepteerd wordt, is $\{w \mid \delta(q_0,w)$ *bevat een toestand uit* $F\}$. Omdat iedere DEA een NEA is, volgt dat de klasse van door NEA's geaccepteerde talen de reguliere verzamelingen bevat (d.i. de DEA talen). We kunnen voor iedere NEA een *equivalente* DEA (die dezelfde taal accepteert) construeren. We laten de toestanden van de DEA corresponderen met de deelverzamelingen van toestanden van de NEA. De geconstrueerde DEA bereikt een toestand bestaande uit de verzameling toestanden, die de NEA bereikt zou kunnen hebben, bij het verwerken van dezelfde invoer.

STELLING. *Als L(A) de taal is, die geaccepteerd wordt door een NEA A, dan kunnen we een DEA A' construeren zodanig dat* $L(A') = L(A)$.

Eindige automaten met ϵ-stappen. We kunnen het eindige automaten model uitbreiden met transities voor de lege invoer ϵ. Formeel is een NEA met $\epsilon-$*stappen* een vijftal als hiervoor, met uitzondering van de transitie functie δ, die nu $Q \times (I \cup \{\epsilon\})$ in $P(Q)$ afbeeld. Uitbreiding van δ, acceptatie, geaccepteerde taal, worden *mutatis mutandis* als boven gedefinieerd.

STELLING. *Als $L(A)$ de taal is geaccepteerd door een NEA A met ϵ-stappen dan kunnen we een NEA A' zonder ϵ-stappen construeren zodanig dat $L(A') = L(A)$.*

Reguliere expressies. Zij I een eindige verzameling symbolen, en zij L, L_1 en L_2 verzamelingen woorden in I^*. De *concatenatie* van L_1 en L_2, geschreven als L_1L_2, is de verzameling $\{xy \mid x \in L_1 \ \& \ y \in L_2\}$. Definieer $L^0 = \{\epsilon\}$ en $L^i = L\,L^{i-1}$ voor alle $i > 0$. De *Kleene afsluiting*, of kortweg *afsluiting*, van L, geschreven als L^*, is de verzameling $L^* = \bigcup_{i=0}^{\infty} L^i$. De betreffende operator wordt *Kleene* * genoemd.

De *reguliere expressies* over een eindig alfabet I,en de verzamelingen die zij voorstellen, worden als volgt inductief gedefinieerd:

1) $\varnothing$ is een reguliere expressie en stelt de lege verzameling voor.
2) ϵ is een reguliere expressie en stelt de verzameling $\{\epsilon\}$ voor.
3) Voor iedere a in I, is a een reguliere expressie die de verzameling $\{a\}$ voorstelt.
4) Als r en s reguliere expressies zijn, die respectievelijk de talen R en S voorstellen, dan zijn $(r+s)$, (rs) en (r^*) reguliere expressies die respectievelijk de verzamelingen $R \cup S$, RS, en R^* voorstellen.

Als we aannemen dat de precedentie van de operaties van hoog naar laag de volgorde * , concatenatie, + heeft, kunnen we haakjes weglaten.

VOORBEELD. $(0+1)^*00(0+1)^*$ stelt de verzameling van alle woorden, die uit nullen en enen bestaan, en tenminste twee opeenvolgende nullen bevatten, voor.

De talen die door eindige automaten geaccepteerd worden zijn precies de verzamelingen die door reguliere expressies voorgesteld worden. Dientengevolge noemen we die talen de reguliere verzamelingen (reguliere talen). Door inductie op de grootte van (i.c. aantal operatoren in) een reguliere expressie kunnen we laten zien dat er een NEA met ϵ-stappen is die dezelfde taal beschrijft. Ook kunnen we aantonen dat er voor iedere DEA een reguliere expressie is die diens taal beschrijft. Tesamen met de voorgaande stellingen volgt dan:

STELLING. *Een taal L wordt door een DEA geaccepteerd dan en slechts dan als L beschreven kan worden door een reguliere expressie.*

Iedere deelverzameling van I^*, met I een eindig alfabet, is een taal. De reguliere talen vormen een (relatief kleine) klasse van talen. De volgende stelling (de *pomp*- of *uvw*-stelling) is nuttig om te bewijzen dat een gegeven taal *niet* regulier is.

STELLING. *Zij L een reguliere verzameling. Er is een constante n, zodanig dat elk woord z uit L, $|z| \geqslant n$, in drie stukken u, v en w verdeeld kan worden, $z = uvw$, $|u| + |v| \leqslant n$ en $|v| \geqslant 1$, en voor elke $i \geqslant 0$ het woord uv^iw tot L behoort. Verder is n niet groter dan het aantal toestanden van de kleinste (met het minste aantal toestanden) EA die L accepteert.*

VOORBEELD. $L = \{x \mid |x| = n^2 \text{ for } n \geqslant 0\}$ is volgens bovenstaande Stelling niet regulier.

STELLING. *De klasse van reguliere verzamelingen is gesloten onder de volgende operaties:*
*(i) Concatenatie en Kleene * .*
(ii) Vereniging, doorsnede en complement: zij vormt dus een Booleaanse algebra.
(iii) Homomorfismen en inverse homomorfismen (op de semigroep).

Myhill-Nerode Stelling en minimalisatie van EA. Met een gegeven taal $L \subseteq I^*$ kunnen we een natuurlijke equivalentie relatie $\sim$ associeren: $x \sim y$, voor $x,y \in I^*$ dan en slechts dan als voor elke $z \in I^*$ geldt dat òf xz en yz beide in L òf xz en yz beide niet in L. Door de equivalentie relatie $\sim$ (reflexief, symmetrisch en transitief) wordt de semigroep I^*, waarin L bevat is, in equivalentie klassen verdeeld. Ieder woord x in I^* kan op zichzelf een equivalentie klasse vormen, maar er kunnen ook minder klassen zijn. Het aantal geinduceerde equivalentie klassen heet de *index* van de equivalentie relatie. Een equivalentie relatie R, zodanig dat $x\,R\,y$ impliceert dat $xz\,R\,yz$ voor alle z, heet *rechts − invariant* (met betrekking tot concatenatie).

STELLING (Myhill-Nerode). *De volgende drie uitspraken zijn gelijkwaardig:*
(i) De verzameling L ,omvat door I^*, wordt geaccepteerd door een EA.
(ii) L is de vereniging van een aantal equivalentie klassen van een rechts-invariante equivalentie relatie R van eindige index.
(iii) Zij de equivalentie relatie $\sim$ gedefinieerd als hierboven (waarbij L dezelfde is als in (i) en (ii)). Dan is $\sim$ van eindige index.

Deze Stelling heeft onder andere het gevolg dat er voor iedere reguliere verzameling een in principe unieke DEA met een minimaal aantal toestanden is, die deze verzameling accepteert.

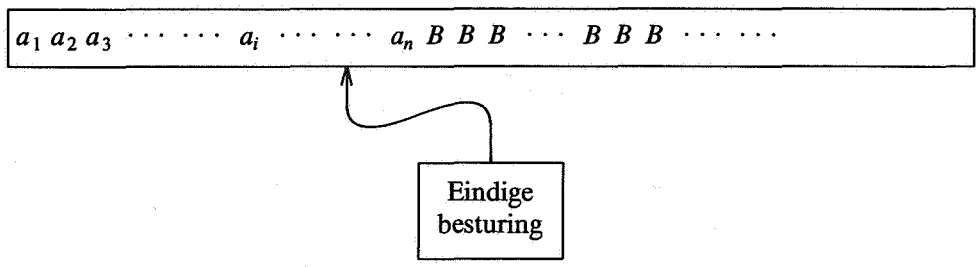
STELLING. *De DEA met het minimum aantal toestanden die een taal L accepteert is op isomorfie (het herbenoemen van de toestanden) na uniek, en de toestanden corresponderen met de equivalentie klassen die door $\sim$ op I^* geinduceerd worden.*

3.2. Oneindige Toestands Automaten.

De Oneindige Toestands Automaat bestaat in wezen uit een EA gekoppelt aan een oneindig groot, dan wel onbeperkt uitbreidbaar, geheugen. Afhankelijk van de manier waarop toegang tot dit oneindig geheugen verkregen wordt, dan wel het geheugen bijgewerkt kan worden, hebben we met verschillende Oneindige Toestands Automaten van doen. De typen die we hieronder behandelen zijn de Turing machine, de pushdown store machine (en de RAM of registerautomaat in sectie 4).

Turing machine en berekenbaarheid. In 1900 formuleerde de wiskundige David Hilbert de wenselijkheid van het vinden van een algoritme die de (on)waarheid van elke wiskundige uitspraak kan bepalen. In 1931 echter bewees Kurt Gödel dat zulk een algoritme niet kan bestaan. Hiertoe was het noodzakelijk dat hij het intuitieve begrip van een effectieve procedure verhelderde en formalizeerde. Aannemende dat iedere effectieve procedure zich in een eindig aantal symbolen, de symbolen gekozen uit een eindig vast alfabet, laat beschrijven, is het makkelijk in te zien dat er, voor het berekenen van veel functies, geen effectieve procedures bestaan. Namelijk, de verzameling functies, die de positieve gehele getallen op {0,1} afbeelden, kan in een één-één correspondentie met de reele getallen gebracht worden. Aangezien echter, onder bovenstaande aanname, de effectieve procedures in een één-één correspondentie met de gehele getallen gebracht kunnen worden, en er geen één-één correspondentie tussen de gehele- en de reele getallen bestaat, moeten er functies zijn zonder bijbehorende effectieve procedures die hun waarden uitrekenen. Heden ten dage is de Turing machine de geaccepteerde formalizering van het intuitieve begrip effectieve procedure. Het is duidelijk dat men niet kan bewijzen dat het Turing machine model equivalent is met ons intuitieve begrip over wat een computer zou kunnen zijn. Er zijn echter dwingende redenen om die equivalentie, bekend als de Church-Turing These, aan te nemen. In het bijzonder is de berekeningskracht van een Turing machine equivalent met die van de digitale computer zoals we die kennen, en ook met die van de meest algemene wiskundige berekeningsbegrippen. Het berekeningsmodel zoals door Alan Turing in 1936 voorgesteld werd, ziet eruit als in de figuur hieronder.

Het basismodel heeft een eindige *besturingseenheid* (eigenlijk een eindige automaat), een *invoerband* die verdeeld is in *cellen*, en een *bandkop* die slechts één cel van de band tegelijkertijd kan lezen. De band heeft een meest linkse cel, maar is naar rechts onbegrensd. Iedere cel op de band kan precies



één symbool, gekozen uit een eindig aantal *bandsymbolen*, bevatten. In het begin van een berekening is de invoer bevat in de n meest linkse cellen, $n \geqslant 0$. De invoer bestaat uit een rij symbolen gekozen uit een deelverzameling van de bandsymbolen die de *invoersymbolen* genoemd worden. De overblijvende, oneindig veel, cellen bevatten elk het *blanco* symbool B, dat een speciaal bandsymbool is dat niet tot het invoeralfabet behoort. In één stap doet de Turing machine, gestuurt door het via de bandkop gelezen symbool en de toestand van de eindige besturing, al het volgende:

1) verandert van toestand;
2) drukt een symbool af in de gelezen cel, en vervangt zodoende het symbool dat daar stond;
3) beweegt de bandkop een cel naar links of een cel naar rechts.

De *taal* $L(M)$ geaccepteerd door een Turing machine M, is de verzameling woorden, over het invoeralfabet, die M in een eindtoestand drijven. De verzameling *eindtoestanden* is een deelverzameling van de verzameling toestanden van de eindige besturing. Voor een Turing machine die een taal L accepteert kunnen we aannemen dat de machine, gestart op invoerwoorden uit L, *stopt*. D.w.z., vanuit de, bij de acceptatie van een invoerwoord bereikte, eindtoestand is geen volgende stap gedefineerd. Voor invoerwoorden, die niet door de machine geaccepteerd worden, is het echter mogelijk dat de hierop gestarte machine nooit stopt.

Recursief opsombare- en recursieve talen. Elke door een Turing machine geaccepteerde taal wordt een *recursief opsombare (r.o.) taal* genoemd. De klasse van r.o. talen is erg uitgebreid, en bevat de contextvrije talen (zie onder) als echte deelverzameling. De klasse van r.o. talen bevat ook talen waarvoor het onmogelijk is mechanisch vast te stellen of een gegeven woord tot die taal behoort: de *lidmaadschap test* is ondoenlijk. Als $L(M)$ zo'n taal is, dan moet elke Turing machine, die $L(M)$ accepteert, voor tenminste één invoer woord dat niet tot $L(M)$ behoort, nooit stoppen. Als een invoer woord w tot $L(M)$ behoort, dan moet M uiteindelijk stoppen als zij gestart wordt op deze invoer. Zolang echter M nog steeds aan het rekenen is op een invoer, kunnen we in het algemeen niet uitmaken of M uiteindelijk de invoer zal accepteren of dat M voor altijd blijft rekenen. De deelverzameling van de r.o. talen die we de recursieve talen noemen heeft deze nadelen niet. Definieer de *recursieve talen* als r.o. talen waarvan het complement ook een r.o. taal is. Voor elke recursieve taal L is er dus een Turing machine M die L accepteert, en ook een Turing machine M' die het complement L' van L accepteert. Als we voor een gegeven invoer w beide machines M en M' op een copie van w starten, en om de beurt een stap laten doen, zal uiteindelijk een van de twee machines moeten stoppen, daar w tot $L \cup L'$ behoort. Daar we deze procedure ook door een enkele Turing machine kunnen laten uitvoeren geldt:

STELLING. *De recursieve talen zijn de talen die geaccepteerd worden door een Turing machine die voor ieder invoerwoord stopt.*

Gevolg van bovenstaande Stelling en voorgaande discussie is dat er r.o. talen zijn die niet recursief zijn. De, in de Informatica belangrijke, contextvrije talen (zie onder) zijn echter alle recursief!

Partieel- en totaal recursieve functies. Behalve als taal accepteerder kan de Turing machine ook gezien worden als berekenaar van functies van de gehele getallen naar de gehele getallen. Als we de gehele getallen *unair* representeren dan wordt het gehele getal $i \geqslant 0$ gerepresenteerd door het woord 0^i. Voor een functie met k argumenten, $i_1,i_2, \ldots, i_k$, worden deze gehele getallen vervangen door voorgaande representaties, gescheiden door "1"en, dus als $0^{i_1}10^{i_2}1, \ldots, 10^{i_k}$. Indien de Turing machine, op zo'n invoer gestart, uiteindelijk stopt (al dan niet in een accepterende toestand) met op de band 0^m voor een of andere m, dan zeggen we dat $f(i_1,i_2, \ldots, i_k) = m$, waarbij f de functie met k argumenten is die door de Turing machine uitgerekend wordt. Is $f(i_1,i_2, \ldots, i_k)$ gedefinieerd voor alle $i_1,i_2, \ldots, i_k$, dan noemen we f een *totaal recursieve functie*. Elke functie $f(i_1,i_2, \ldots, i_k)$ die door een Turing machine uitgerekend wordt noemen we een *partieel recursieve functie*. In bepaalde zin vormen de partieel recursieve functies een analogon van de r.o. talen, daar zij berekend worden door Turing machines die op een gegeven invoer misschien nooit stoppen. De totaal recursieve functies corresponderen met de recursieve verzamelingen daar zij berekend worden door Turing machines die voor elke invoer stoppen. Alle gewone aritmetische functies, zoals vermenigvuldiging, machtsverheffing, faculteit, afgeronde logaritme, zijn totaal recursieve functies.

Niet-deterministische Turing machines. Een *niet−deterministische Turing machine* wordt net zo gedefinieerd als de bovenstaande (deterministische) Turing machine met de volgende wijziging. Voor een gegeven toestand en symbool dat op de band gelezen wordt, heeft de machine niet één maar een eindig aantal keuzen voor de uit te voeren stap. Iedere dergelijke keuze bestaat uit een nieuwe toestand, een bandsymbool om af te drukken op de gelezen cel, en een richting voor de beweging van de bandkop. De niet-deterministische Turing machine accepteert een gegeven invoer als er een eindige rij keuzen is, die tot een accepterende toestand leidt. Net zoals bij de eindige automaat stelt de toevoeging van niet-determinisme de Turing machine niet in staat om nieuwe talen te accepteren. In feite voegt niet-determinisme, al dan niet in combinatie met andere denkbare uitbreidingen, geen extra kracht toe. Of dit ook zo is als we bijvoorbeeld de tijd (aantal stappen) die een machine in zijn berekening mag gebruiken gaan begrenzen, vormt een van de meest diepe, onopgeloste problemen uit de complexiteitstheorie, zie sectie 4. Zonder restricties op de te gebruiken hulpbronnen aan tijd en/of geheugenruimte geldt echter:

STELLING. *Als L door een niet-deterministische Turing machine M wordt geaccepteerd, dan wordt L ook door een deterministische Turing machine M' geaccepteerd.*

Contextvrije grammatica's en pushdown store automaten. De contextvrije talen zijn, evenals de eerdere reguliere verzamelingen, van groot practisch belang. Bijvoorbeeld, in het definieren van programmeertalen, in het formalizeren van het begrip parseren, in het simplificeren van vertalingen van programmeertalen, en in andere symboolrij verwerkingstoepassingen. Contextvrije grammatica's zijn bijvoorbeeld nuttig voor het beschrijven van aritmetische expressies, met willekeurige nesting van gebalanceerde haakjes, en blokstructuur in programmeertalen. Geen van deze aspecten kan gerepresenteerd worden door reguliere expressies. Een *contextvrije grammatica* (*cvg* of alleen *grammatica*) bestaat uit een viertal $G = (V,T,P,S)$ waarbij V een eindige verzameling van *variabelen* (ook *niet−terminalen* of *syntactische categorieën*) is, en T een verzameling van *terminalen*. We nemen aan dat V en T disjunct zijn. P is een eindige verzameling productieregels van de vorm $A \rightarrow x$, waarbij A een variabele is en x een symboolrij over $(V \cup T)^*$. Tenslotte is S een speciale variabele die het startsymbool genoemd wordt. De *taal* $L(G)$ die door de grammatica G voortgebracht wordt definieren we als volgt. Als $A \rightarrow x$ een productie in P is, en als y en z symboolrijen in $(V \cup T)^*$ zijn dan geldt $yAz \Rightarrow yxz$ (*yAz leidt yxz direct af*). Stel dat $x_1,x_2, \ldots, x_m$ symboolrijen in $(V \cup T)^*$ zijn, $m > 0$, en

$$x_1 \Rightarrow x_2 \Rightarrow x \cdots \Rightarrow x_{m-1} \Rightarrow x_m,$$

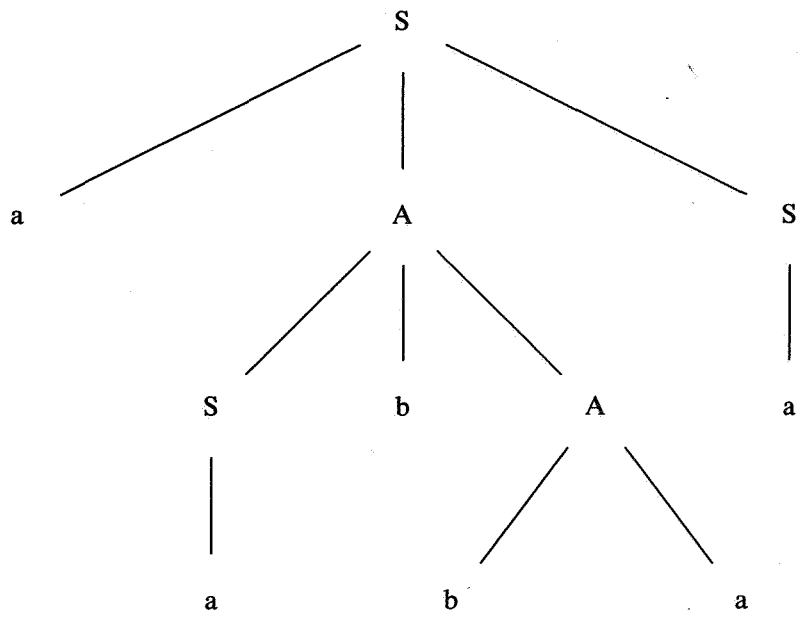
dan geldt $x_1 \overset{*}{\Rightarrow} x_m$, of x_1 *leidt* x_m *af* in de grammatica G. Dus, $\overset{*}{\Rightarrow}$ is de reflexieve en transitieve afsluiting van $\Rightarrow$. De *taal* die door G *voortgebracht* wordt is $L(G) = \{w \mid w \in T^* \ \& \ S \overset{*}{\Rightarrow} w\}$. Twee

grammatica's G en G' zijn *equivalent* als $L(G) = L(G')$. Het is nuttig om afleidingen als *bomen* te representeren. Zulke plaatjes, die *afleidingsbomen* dan wel *parseringsbomen* genoemd worden, leggen een structuur op de woorden van een taal, die nuttig is in toepassingen als compilatie van programmeertalen. De knopen van een afleidingsboom zijn gemerkt met terminalen of variabelen van de grammatica, of mogelijk met ϵ. Als een inwendige knoop n met A gemerkt is, en de zonen van n zijn gemerkt met (van links naar rechts) $X_1, X_2, \ldots, X_k$, dan moet $A \to X_1 X_2 \cdots X_k$ een productie in P zijn. De *wortel* van de boom, d.w.z. de knoop zonder vader, is gemerkt met het *sentence* symbool S; de *bladeren* van de boom, d.w.z. de knopen zonder zonen, zijn gemerkt met terminale symbolen.

VOORBEELD. $G = (\{S,A\},\{a,b\},P,S)$, en P bestaat uit $S \to aAS \mid a$ en $A \to SbA \mid SS \mid ba$, waarbij de verticale streep een afkorting is voor verschillende manieren om wat links van de pijl staat te herschrijven. Bijvoorbeeld, als $A \to x_1$, $A \to x_2, \ldots, A \to x_k$, dan schrijven we kortweg $A \to x_1 \mid x_2 \mid \cdots \mid x_k$. Een afleiding in G is:

$$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa\ .$$

De bijpassende afleidingsboom is hieronder afgebeeld.



Aannemende dat de zonen, van een vader in de boom, de zelfde ordening van links naar rechts hebben als de symbolen in de corresponderende symboolrijen, noemen we het terminale woord, gevormt door de terminalen van de bladeren van links naar rechts te lezen, de *oogst* van de boom. Aangezien het mogelijk is elke variabele in de symboolrij te herschrijven, kunnen er vele verschillende afleidingen, van hetzelfde terminale woord, met dezelfde afleidingsboom zijn. Er kunnen echter ook verschillende afleidingen, voor hetzelfde terminale woord, met verschillende afleidingsbomen zijn. In dat geval noemen we de desbetreffende grammatica *ambigue*. Indien iedere grammatica, die een gegeven contextvrije taal voortbrengt, ambigue is heet die taal *inherent ambigue*. De taal

$$L = \{a^n b^n c^m d^m \mid n>0, m>0\} \cup \{a^n b^m c^m d^n \mid n>0, m>0\}\ ,$$

is inherent ambigue. We definieren een programmeertaal meestal door een , min of meer, contextvrije grammatica. Een programma is een woord dat door die grammatica voortgebracht wordt. Daar in programmeertalen de betekenis van een programma (d.w.z. met welke opeenvolging van computer handelingen het programma overeenstemt, hetgeen vastgesteld wordt door het programma te "vertalen") volgt uit de afleidingsboom, zijn ambigue talen onbruikbaar, daar het "vertalen" van hetzelfde programma tot meer dan één resultaat zou kunnen leiden.

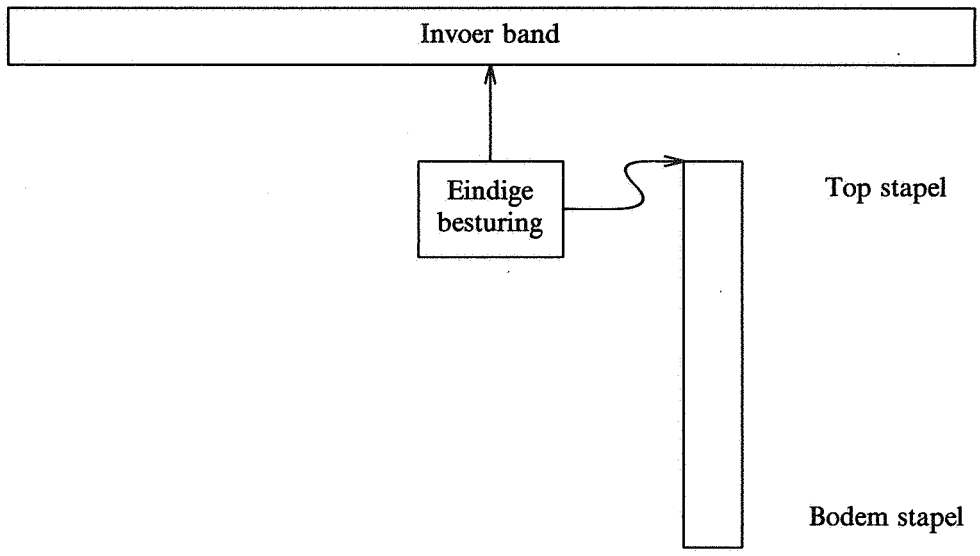
Er zijn verschillende manieren om de vorm van de productieregels in een contextvrije grammatica te beperken zonder de uitdrukkingskracht te verminderen. Elke niet-lege contextvrije taal L kan gegenereerd worden door een contextvrije grammatica G met de volgende eigenschappen.

1) Iedere variabele en iedere terminal komen voor in afleidingen van woorden in L.

2) Er zijn geen producties van de vorm $A \rightarrow B$ met A en B variabelen.

Verder, als ϵ niet tot L behoort, dan behoeven er geen producties van de vorm $A \rightarrow \epsilon$ te zijn, en kunnen we afdwingen dat iedere productieregel in G van de vorm $A \rightarrow BC$ of $A \rightarrow a$ is: de *Chomsky Normaal Vorm*. Alternatief, kunnen we iedere productieregel van G van de vorm $A \rightarrow ax$ met x een rij variabelen (misschien leeg) maken: de *Greibach Normaal Vorm*. Zulke normaalvormen zijn van belang bij parseringskwesties.

Pushdown automaten. Zoals de reguliere talen hun equivalente eindige automaten hebben, zo hebben de contextvrije talen de equivalente pushdown (stapel) automaten. (De pushdown automaat die met de contextvrije talen correspondeert is de niet-deterministische versie; de deterministische versie correspondeert met een echte deelverzameling van de contextvrije talen. Gelukkig correspondeert de deterministische versie met het taalgenererend vermogen van de syntax van de meeste programmeertalen.) Een *pushdown automaat* (*PDA*) heeft een *invoerband*, een *eindige besturing* en een *stapel*. De stapel is een "last in first out" oneindig geheugen, en bestaat uit een rij van symbolen uit een eindig *stapel alfabet*. Om een symbool uit de stapel te lezen, moeten alle later opgeslagen symbolen verwijdert worden: het symbool moet boven op de stapel komen. Het bovenste symbool heet de *top* van de stapel. Zie onderstaand figuur.



De machine is niet-deterministisch, en heeft in iedere situatie een eindig aantal keuzen voor de volgende stap. De stappen zijn van twee typen. Het eerste type *verbruikt* een invoer symbool. Afhankelijk van het invoersymbool, het symbool op de top van de stapel en de toestand van de

eindige besturing, zijn een aantal keuzen mogelijk. Iedere keus bestaat uit een nieuwe toestand voor de eindige besturing en een (mogelijk lege) symboolrij om het bovenste stapelsymbool mee te vervangen. Na het uitvoeren van zulk een keuze verschuift de kop op de invoerband een symbool naar rechts. In het tweede type stap (ϵ-stap genaamd) wordt het invoersymbool niet verbruikt, en de kop op de invoerband beweegt niet. Dit type stap stelt de PDA in staat de stapel te manipuleren zonder invoersymbolen te lezen.

Er zijn twee natuurlijke manieren om de *taal* die door een PDA geaccepteerd wordt te definieren. De eerste manier is, om de taal te definieren als de verzameling van alle invoer woorden waarvoor er een rij stappen is die de PDA zijn stapel doet legen. Deze taal wordt de *taal* die met een *lege stapel* geaccepteerd wordt genoemd. De tweede manier is om, net zoals bij de eindige automaat, een aantal toestanden als eindtoestanden aan te wijzen, en de geaccepteerde *taal* te definieren als de verzameling van alle invoerwoorden die voor een mogelijke keuze van stappen de PDA in een *eindtoestand* drijven. Een PDA is *deterministisch* (*DPDA*), als de verzameling keuzen voor de volgende stap in iedere situatie precies uit één mogelijkheid bestaat. Dat wil onder meer zeggen dat als er voor een toestand q en een stapel symbool Z (op top) een ϵ-stap gedaan kan worden, dan kan er in die situatie geen invoersymbool verbruikt worden.

STELLING. *De verzameling van talen, die door een PDA met lege stapel geaccepteerd worden, is gelijk aan de verzameling van talen, die door een PDA door eindtoestand geaccepteerd worden. Deze verzameling is de verzameling van contextvrije talen.*

STELLING. *De verzameling van talen die door DPDA's geaccepteerd worden, de deterministisch contextvrije talen, is een echte deelverzameling van de contextvrije talen. (Bijvoorbeeld de verzameling van palindromen, woorden die van achter naar voren geschreven hetzelfde blijven, is contextvrij maar niet deterministisch context vrij.)*

STELLING (pomp- of uvwxy stelling voor cvt's). *Zij L een contextvrije taal. Er is een constante n, die alleen van L afhangt, zodanig dat, als $z \in L$ en $|z| \geqslant n$, we z kunnen schrijven als $uvwxy$ zo dat:*

1) $|v| + |x| > 0$;

2) $|vwx| \leqslant n$;

3) *voor alle $i \geqslant 0$ geldt dat $uv^i wx^i y \in L$.*

Analoog met de soortgelijke Stelling voor de reguliere talen wordt de pompstelling gebruikt om aan te tonen dat een gegeven taal *niet* contextvrij is. Bijvoorbeeld $L = \{a^i b^i c^i \mid i > 0\}$ is dus niet contextvrij.

STELLING. *De contextvrije talen zijn gesloten onder vereniging, concatenatie en Kleene *, homomorfisme, en invers homomorfisme; maar niet onder doorsnede of complement.*

Door te appeleren aan het feit dat iedere contextvrije taal door een pushdown store machine geaccepteerd kan worden, blijkt dat de *Dyck taal* een fundamentele rol speelt. De Dyck taal met k generatoren is de taal bestaande uit de rijen goed geneste haakjes van k typen. De volgende grammatica genereert de Dyck taal met k generatoren.

$$G = (\{S\}, \{1,2,\ldots,k,1',2',\ldots,k'\}, \{S \to SS, S \to iSi' \mid ii' (1 \leqslant i \leqslant k)\}, S)$$

STELLING (Chomsky). *Elke contextvrije taal L kan voorgesteld worden als $L = h(D \cap R)$, waarbij h een homomorfisme, D een Dyck taal, en R een reguliere verzameling is.*

De Dyck talen representeren eigenschappen die contextvrije talen aantrekkelijk maken als formeel raamwerk voor programmeertalen: de goed geneste blokstructuur. Ofschoon iedere contextvrije taal door een PDA geaccepteerd kan worden, biedt dit ons niet direct een deterministische procedure om

zulk een taal te *herkennen*, d.w.z. dat van ieder candidaat woord bepaald wordt of het wel of niet tot de taal behoort. Er zijn diverse deterministische algoritmen om contextvrije talen te herkennen. Voor de deterministische contextvrije talen vormt de bijbehorende DPDA een efficient herkenningsmechaniek. Deze talen worden veel gebruikt om programmeertalen te beschrijven. Compiler genererende systemen hebben vaak een syntactische specificatie nodig, die alleen de representatie van deterministisch contextvrije talen toelaten. Voorbeelden van zulke beperkte typen contextvrije grammatica's zijn: LR(k)-, LL(k)-,SLR(k)-,LALR(k)-grammatica's ($k \geqslant 0$). Deze grammatica's hebben efficiente parseringsalgoritmen. De talen voortgebracht door de LR(k) grammatica's ($k > 0$) zijn precies de deterministisch contextvrije talen.

3.3. Beslisbare en onbeslisbare problemen.

Een probleem is *beslisbaar* als er een algoritme (equivalent: Turing machine) bestaat, die als invoer een instantie van het probleem heeft en bepaalt of het antwoord in die instantie "ja" of "nee" is. Een probleem is *onbeslisbaar* als het niet beslisbaar is. We kunnen een probleem zien als een taal over de gebruikte symbolen om het probleem in uit te drukken: de woorden in die taal bestaan uit de instanties van het probleem. De deelverzameling van deze taal waarvoor het antwoord voor iedere daarin bevatte instantie van het probleem "ja" is, kunnen we ook als taal opvatten. Aannemende dat het beslisbaar is of een gegeven symboolrij een instantie van het probleem beschrijft, is een probleem dus beslisbaar als de taal van "ja" instanties recursief is.

STELLING. *De recursieve talen zijn gesloten onder vereniging, doorsnede en complement en vormen dus een Booleaanse algebra. (De r.o. talen zijn gesloten onder vereniging en doorsnede maar niet onder complement.)*

Het is onbeslisbaar of een willekeurige gegeven Turing machine, gestart op een willekeurige gegeven invoer, ooit zal stoppen, d.w.z. ooit een eindtoestand bereikt: dit is het *stop probleem* voor Turing machines. Een *Universele* Turing machine is een Turing machine die, door een gedeelte van zijn invoer te gebruiken als een beschrijving van een andere Turing machine, elke andere Turing machine gestart op elke invoer, kan simuleren. Zulke machines zijn relatief eenvoudig te construeren.

STELLING. *Het is onbeslisbaar of een gegeven Universele Turing machine op een willekeurige invoer zal stoppen.*

Een instantie van *Post's Correspondentie Probleem* (*PCP*) bestaat uit twee lijsten, $A = w_1, w_2, \ldots, w_k$ en $B = x_1, x_2, \ldots, x_k$, van symboolrijen over een gegeven alfabet I. Deze instantie van PCP heeft een oplossing als er een rij gehele getallen $i_1, i_2, \ldots, i_m$ bestaat, $m > 0$, zodat $w_{i_1} w_{i_2} \cdots w_{i_m} = x_{i_1} x_{i_2} \cdots x_{i_m}$. De rij $i_1, i_2, \ldots, i_m$ heet dan een oplossing van deze instantie van PCP.

STELLING. *Het PCP is onbeslisbaar.*

Om te bewijzen dat een probleem A onbeslisbaar is, reduceren we veelal een reeds bekend onbeslisbaar probleem tot probleem A. (Bijvoorbeeld, door aan te tonen, dat *als* zo'n probleem A beslisbaar zou zijn dan ook het stopprobleem voor Universele Turing machines (of het PCP) beslisbaar was.)

De volgende problemen betreffende contextvrije grammatica's en -talen zijn onbeslisbaar.
- Of een willekeurige contextvrije grammatica ambigue is.
- Of een contextvrije grammatica een inherent ambigue taal voortbrengt.
- Of $L(G) = T^*$ voor een willekeurige contextvrije grammatica $G = (V, T, P, S)$.
- Of de doorsnede van $L(G)$ en $L(G')$ leeg is voor willekeurige cvg's G, G'.
- Of de talen van twee willekeurige contextvrije grammatica's gelijk zijn.
- Of het complement van een contextvrije taal een contextvrije taal is.
- Of de doorsnede van twee contextvrije talen contextvrij is.
- Of $L(G)$ regulier is voor willekeurige cvg G.

- Of een gegeven contextvrije grammatica een deterministisch contextvrije taal voortbrengt.
- Als L, L' willekeurige deterministisch contextvrije talen zijn:

  of de doorsnede van L en L' leeg is;
  of L bevat is in L';
  of de doorsnede van L en L' deterministisch contextvrij is;
  of de doorsnede van L en L' contextvrij is;
  of de vereniging van L en L' deterministisch contextvrij is.

Beslisbare problemen betreffende contextvrije talen:
- de lidmaatschapskwestie: behoort een woord tot $L(G)$ voor G contextvrij?
- is $L(G)$ leeg, eindig, oneindig voor G contextvrij?
- is $L(G) = T^*$ voor G deterministisch contextvrij?
- is $L(G)$ regulier voor G deterministisch contextvrij?
- is $L(G) = R$, voor R een gegeven reguliere taal en G deterministisch contextvrij?

Het is momenteel nog niet bekend of de gelijkheid (equivalentie) van twee deterministisch contextvrije talen (grammatica's) beslisbaar is. Merk op dat de gelijkheid van twee reguliere talen beslisbaar is en dat de gelijkheid van twee contextvrije talen onbeslisbaar is.

3.4. De Chomsky hierarchie.

Van de drie belangrijke taalklassen hierboven: regulier, contextvrij en recursief opsombaar, hebben we alleen de contextvrije talen grammaticaal gekarakteriseerd. De z.g. Chomsky hierarchie bestaat uit vier klassen van talen, die van de kleinste tot de grootste klasse echt in elkaar bevat zijn in de volgorde: regulier, contextvrij, contextgevoelig, recursief opsombaar.(De deterministisch contextvrije talen bevatten echt de reguliere talen en zijn echt bevat in de contextvrije talen, en de recursieve talen bevatten echt de contextgevoelige talen en zijn zelf echt bevat in de recursief opsombare talen.) Noam Chomsky definieerde de vier bovenstaande hoofdklassen door middel van (z.g. generatieve) grammatica's, als volgt.

Een (*generatieve*) *grammatica* is een viertal $G = (V,T,P,S)$ waarbij V,T, en S hetzelfde gedefinieerd zijn als in de restrictie tot contextvrije grammatica's, en P een verzameling van producties van de vorm $x \rightarrow y$ is, waarbij x en y willekeurige symboolrijen over de variabelen en de terminalen mogen zijn, $x \neq y$. We schrijven $uxv \Rightarrow uyv$ als $x \rightarrow y$ een productie is. Als boven, staat $\overset{*}{\Rightarrow}$ voor de reflexieve en transitieve afsluiting van $\Rightarrow$. De door G voortgebrachte taal is weer $L(G) = \{w \mid w \in T^* \ \& \ S \overset{*}{\Rightarrow} w\}$.

STELLING. *De klasse van talen die door bovenstaande onbeperkte grammatica's wordt gegenereerd is precies de klasse van r.o. talen.*

Een grammatica is *contextgevoelig* als voor elke productie $x \rightarrow y$ in P geldt dat $|y| \geq |x|$. Er zijn contextgevoelige talen die niet contextvrij zijn, zoals: $L = \{w \mid |w| \text{ is een priemgetal}\}$.

Als alle producties van een contextvrije grammatica van de vorm $A \rightarrow wB$ of $A \rightarrow w$ zijn, waarbij A en B variabelen zijn, en w een (mogelijk lege) rij van terminalen is, heet de grammatica *rechts lineair*. Met $A \rightarrow Bw$ en $A \rightarrow w$ *links lineair*. Een rechts- of links lineaire grammatica heet een *reguliere grammatica*. De reguliere grammatica's brengen precies de reguliere verzamelingen (talen) voort. De r.o. talen worden gegenereerd door onbeperkte grammatica's en geaccepteerd door Turing machines; de contextvrije talen worden gegenereerd door contextvrije grammatica's en geaccepteerd door pushdown automaten; en de reguliere talen worden gegenereerd door reguliere grammatica's en geaccepteerd door eindige automaten. Voor de contextgevoelige talen geldt dat ze gegenereerd worden door contextgevoelige grammatica's en geaccepteerd worden door *lineair begrensde automaten* (*LBA*'s). Een LBA is een Turing machine die, in plaats van een potentieel oneindig lange band om op te rekenen, het moet stellen met de portie band waarop de invoer aanvankelijk geplaatst is. Iedere contextgevoelige taal kan door een niet-deterministische LBA

geaccepteerd worden. De vraag of dit ook geldt voor deterministische LBA's is een van de oudste onopgeloste problemen uit de automaten theorie: het *LBA Probleem*. Deze kwestie kan ook beschouwd worden als een probleem in de berekeningscomplexiteit, zie in de volgende sectie.

4. Complexiteit van Algoritmen.

In de vorige sectie "Automatentheorie" kwamen we de begrippen *berekenbare functie* en *beslisbaar probleem* tegen. In de practijk is het echter niet voldoende om te weten dat er een oplossingsmethode bestaat, maar moet de oplossing ook daadwerkelijk bepaald worden. Het is dus nodig, dan wel nuttig, om vooraf te weten hoeveel tijd, geheugenruimte en andere hulpbronnen er met de uitvoering van een gegeven algoritme, die het probleem in kwestie oplost, op de gebruikte rekenmachine gemoeid zijn. De complexiteitstheorie poogt boven- en ondergrenzen te geven aan de inherente berekeningscomplexiteit van problemen. Deze inherente complexiteit hangt af van de aard van de veronderstelde rekenapperatuur. *Berekeningsmodellen* zijn abstracties en formaliseringen van zulke (bestaande of denkbare) apperaturen. Om de prestaties van een algoritme te analyseren, en tot nauwkeurige uitspraken te komen over de benodigde rekentijd of geheugengebruik, is zulk een berekeningsmodel noodzakelijk. Omgekeerd, zullen verschillende berekeningsmodellen vaak verschillende algoritmen oproepen, en voor hetzelfde algoritme een verschillende complexiteit geven. Varianten van de Turing machine in sectie 3 zijn veel gebruikte berekeningsmodellen. Ofschoon oppervlakkig gezien geheel onrealistisch, behalve misschien met betrekking tot berekeningen die veel gebruik maken van massageheugens als banden of schijven, is de Turing machine simpel genoeg om (moeilijk te krijgen) ondergrenzen aan de complexiteit van problemen te geven. In het uiterste geval door te laten zien dat ze onbeslisbaar zijn (zie sectie 3). Het meest gebruikte berekeningsmodel is echter de *random access machine* (*RAM*) of *register automaat*, zie onder. Algoritmen kunnen volgens diverse criteria geevalueerd worden. De prestaties worden uitgedrukt in termen van de grootte van de op te lossen probleeminstantie.

Een *probleem* is een algemene vraag die beantwoord moet worden, welke vraag gewoonlijk verschillende *parameters* of vrije variabelen heeft, waarvan de waarden ongespecificeerd zijn. Een probleem wordt gegeven door: (1) een algemene beschrijving van al zijn parameters, en (2) een verklaring betreffende welke eigenschappen het antwoord, of de *oplossing*, moet hebben. Een *instantie* van het probleem wordt verkregen door specifieke waarden voor alle parameters te geven. De *grootte* van een probleem instantie is een maat voor de hoeveelheid invoer data. Voor deze grootte wordt meestal de lengte van de specificatie van de probleeminstantie in binaire codering genomen. De probleemgrootte kan echter ook in andere parameters uitgedrukt worden. Bij een graafprobleem, bijvoorbeeld, kan de probleemgrootte uitgedrukt worden in het aantal knopen p en het aantal kanten e van de te onderzoeken graaf G. Als we aannemen dat we de knopen coderen als binaire getallen tussen 1 en p zou dit voor de binaire probleemgrootte leiden tot orde van grootte $p^2 \log p$.

De door een algoritme benodigde uitvoeringstijd (geheugenruimte), uitgedrukt als een functie van de grootte van de probleeminstantie, door voor iedere mogelijke invoergrootte de *grootste* hoeveelheid tijd (geheugenruimte) te nemen die de algoritme nodig kan hebben om een probleeminstantie van die grootte op te lossen, heet de *tijdcomplexiteit* (*ruimtecomplexiteit*) van het algoritme. We onderscheiden *slechtste geval* complexiteit en *gemiddelde* complexiteit. Voorgaande definitie gaat vanzelfsprekend over de slechtste geval complexiteit. Bij de *gemiddelde geval* complexiteit wordt de complexiteit bepaald als het (gewogen) gemiddelde van de hoeveelheid tijd of geheugenruimte die het algoritme nodig heeft voor de (waarschijnlijkheidsdistributie van de) verschillende probleeminstanties van grootte n. Het gedrag van de complexiteit in de limiet heet de *asymptotische* tijd (ruimte) complexiteit.

Naarmate de rekenapperatuur sneller wordt, en wij daardoor grotere problemen aankunnen, bepaalt de asymptotische complexiteit van de algoritme steeds meer de toename in formaat van problemen die de snellere machines aankunnen.

N.B. Volgens de in zwang zijnde notationele conventie definieren we:
- $f(n) \in O(g(n))$ als er een positieve constante c is zodat $|f(n)| < c.g(n)$ voor alle n.

VOORBEELD. Voor een gegeven probleem hebben we de beschikking over vijf algoritmen, $A_1, A_2, \ldots, A_5$, met de onderstaande tijdcomplexiteit (uitgedrukt in milliseconden).

Algoritme	Tijdcomplexiteit	Maximale probleemgrootte per:				
		sec.	min.	uur	etmaal	jaar
A_1	n	1000	60.000	3.600.000	86.400.000	31.536.000.000
A_2	$n \log_2 n$	140	4.893	200.000	3.940.000	1.051.200.000
A_3	n^2	31	244	1.897	9.295	177.583
A_4	n^3	10	39	153	442	3.159
A_5	2^n	9	15	21	25	34

Stel nu dat onze machines 100 maal zo snel worden:

Algoritme	Tijdcomplexiteit	Max. probleem grootte vóór versnelling	Max. probleem grootte ná versnelling
A_1	n	m_1	$100m_1$
A_2	$n \log_2 n$	m_2	$\approx 100m_2$ voor grote m_2
A_3	n^2	m_3	$10\ m_3$
A_4	n^3	m_4	$4{,}6\ m_4$
A_5	2^n	m_5	$m_5+6{,}6$

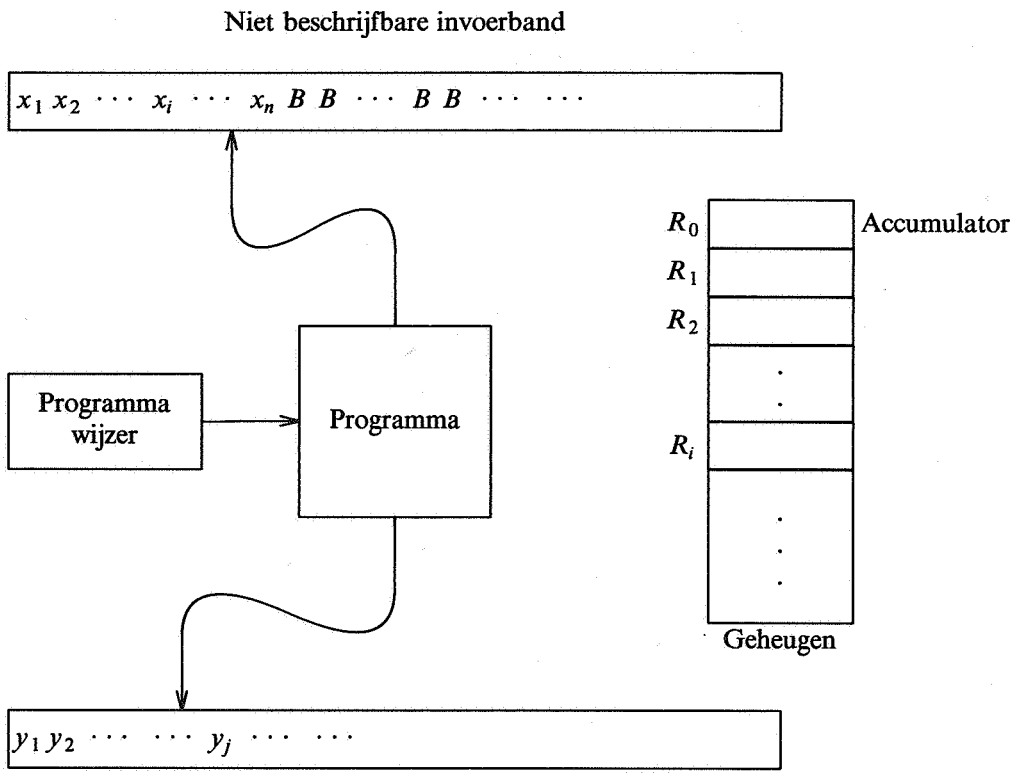
Een algoritme met een snel groeiende complexiteit kan natuurlijk beter zijn dan een algoritme met langzamer groeiende complexiteit, voor de kleine probleeminstanties, afhankelijk van de constanten.

VOORBEELD. Stel dat de tijdcomplexiteit van de algoritmen $A_1, A_2, \ldots, A_5$ gegeven wordt door respectievelijk $1000n$, $100n \log n$, $10n^2$, n^3, 2^n . Dan is A_5 het beste voor problemen van grootte $1 \leqslant n < 10$, A_3 het beste voor problemen van grootte $9 < n < 59$, A_2 het beste voor $58 < n < 1025$, en A_1 het beste voor problemen groter dan 1024 .

4.1. Het RAM berekeningsmodel.

De RAM (random access machine of registerautomaat) modelleert, min of meer waarheidsgetrouw, een sequentieele rekenmachine met één accumulator. Het programma wordt apart van de data opgeslagen en kan zichzelf niet wijzigen. Een RAM bestaat uit een invoerband waarvan alleen gelezen kan worden, een uitvoerband waarop alleen geschreven kan worden, en een geheugen, zie figuur.

De invoerband bestaat uit een rij cellen die elk een geheel getal kunnen bevatten. Iedere maal dat er een getal van de invoer gelezen wordt beweegt de leeskop een cel naar rechts. De uitvoerband bestaat, net als de invoerband, uit een rij cellen die bij aanvang van de berekening alle onbeschreven zijn. Het uitvoeren van een schrijfinstructie bestaat uit het afdrukken van een geheel getal in de cel die zich onder de schrijfkop bevindt, en het verschuiven van de schrijfkop naar de rechtsaangrenzende cel. Het geheugen bestaat uit een (onbegrensde) rij *registers*, $R_0, R_1, \ldots, R_i, \cdots$, die ieder een geheel getal van willekeurige grootte kunnen bevatten. Met R_i duiden we zowel het $i+1$-ste register als het in dat register bevatte getal aan. De onbeperktheid van het aantal registers, en van de getallen die in één register passen, is een redelijke aanname in die gevallen waarin de grootte van het probleem klein genoeg is om in het snelgeheugen van een computer te passen, en de in de berekening gebruikte getallen klein genoeg zijn om in een computerwoord te passen. Het programma van de RAM wordt niet in dit geheugen opgeslagen. We kunnen daarom aannemen dat het programma zichzelf niet wijzigt. Het programma bestaat uit een rij van, mogelijk van een *label* voorziene, *instructies*. Deze instructies lijken op die welke we gewoonlijk in de assembleercode van echte computers aantreffen. Voor de *deterministische* versie van de RAM eisen we dat dezelfde label niet bij twee verschillende



instructies voorkomt. We hebben beschikking over aritmetische-, indirecte adressering-, invoer-, uitvoer- en vertakkingsinstructies. Alle berekeningen vinden plaats in het eerste register R_0, de *accumulator*. We hebben de beschikking over het repertoire hieronder.

	Aard	code	kosten in tijdeenheden
1.	Directe toewijzing	$R_i \leftarrow b$	$k(b)$
2.	Indirecte toewijzing	$R(R_i) \leftarrow b$	$k(R_i)+k(b)$
3.	Indirecte toewijzing	$R_i \leftarrow R(R_j)$	$k(R_j)+k(R(R_j))$
4.	Optelling	$R_i \leftarrow a+b$	$k(a)+k(b)$
5.	Aftrekking	$R_i \leftarrow a-b$	$k(a)+k(b)$
6.	Voorwaardelijke vertakking	IF a COMP b THEN GOTO L_1 ELSE GOTO L_2	$k(a)+k(b)$
7.	Onvoorwaardelijke vertakking	GOTO L	1
8.	Leesopdracht	READ(R_i)	$k(R_i)$
9.	Schrijfopdracht	WRITE(R_i)	$k(R_i)$
10.	Accepteer	ACCEPT	1

In deze tafel mogen a en b operanden zijn van de vorm i (een geheel getal) of R_i (de inhoud van het register R_i); de relatie COMP mag $<,\leqslant,=,\geqslant,>$ of $\neq$ zijn (met de gewone betekenis). $R(R_j)$ betekent indirecte adressering: $R(R_j)$ staat voor R_i als de inhoud van register R_j gelijk is aan i. Als $R_j<0$ terwijl $R(R_j)$ geevalueerd wordt, wordt de berekening beeindigd met een foutmelding. Bij de uitvoering van een instructie van type 1-5, 8 of 9 wordt de programmawijzer (die het nummer van de volgende uit te voeren instructie bijhoudt) met 1 opgehoogd. Dientengevolge worden de programma instructies in volgorde uitgevoerd tot er een instructie van type 6 of 7 verschijnt. Bij een instructie van het type GOTO L , L een *label*, wordt de programma wijzer op het nummer van de door L gemerkte instructie gezet. Hierna worden de instructies weer in volgorde uitgevoerd. Wordt een voorwaardelijke vertakking met a COMP b ontmoet, en is a COMP b waar, dan is die instructie equivalent met GOTO L_1 . Is a COMP b onwaar, dan is de instructie gelijkwaardig met GOTO L_2 . Het programma termineert als ACCEPT ontmoet wordt, dan wel een vertakking die equivalent is met GOTO L terwijl L in het programma niet voorkomt, of een instructie die een negatief direct adres inhoudt. READ R_i betekent dat de inhoud van de cel onder de leeskop, op de invoerband, in register R_i geplaatst wordt; WRITE R_i betekent dat de inhoud van register R_i in de cel onder de schrijfkop, op de uitvoerband, geschreven wordt. De berekening begint met de eerste instructie van het programma, de leeskop op de eerste cel van de invoerband, alle registers met inhoud 0 , en met de schrijfkop op de meest linkse cel van de uitvoerband. In het algemeen definieert een RAM programma een afbeelding van invoer naar uitvoer. Aangezien de berekening niet voor alle mogelijke invoer hoeft te termineren, is de afbeelding een partiele (d.w.z. niet noodzakelijk voor alle argumenten gedefinieerd). Net zoals de Turing machines van de voorgaande sectie, kunnen de RAM's als taalherkenner en als functieberekenaar beschouwd worden.

VOORBEELD. Bereken $f(n)=n^n$ voor $n \geqslant 1$ en $f(n)=0$ voor $n \leqslant 0$, n een geheel getal. Bij aanvang van de berekening is n in de eerste cel van de invoerband geplaatst, en $f(n)$ wordt in de eerste cel van de uitvoerband geschreven.

```
      READ R1
      IF R1<1 THEN GOTO L5 ELSE GOTO L1
L1:   R2 ← R1
      R3 ← 1
L2:   R1 ← R1−1
      R4 ← R2
L3:   R4 ← R4−1
      R5 ← R5+R3
      IF R4<1 THEN GOTO L4 ELSE GOTO L3
L4:   R3 ← R5
      R5 ← 0
      IF R1<1 THEN GOTO L5 ELSE GOTO L2
L5:   WRITE R3
      ACCEPT
```

De *looptijd* van een RAM programma wordt bepaald met gebruikmaking van een kostenfunctie k die de kosten in tijdeenheden per uitgevoerde instructie aangeeft. Deze functie k van de gehele getallen in de natuurlijke getallen is, per instructie, aangegeven in de voorgaande instructietabel. Verschillende keuzen van k kunnen leiden tot geheel verschillende looptijden van een en hetzelfde programma. De *tijdcomplexiteit* van een RAM programma is de functie $T(n)$ die het maximum is, genomen over alle invoer van grootte n, van de som van de *tijden* zoals die per uitgevoerde instructie gevergd worden. De meest voorkomende waardetoekenningen voor k zijn:

- het *uniforme* kostencriterium: $k(x)=1$ voor alle x;
- het *logaritmische* kostencriterium: $k(x)$ is gelijk aan de lengte van de binaire codering van x.

De *geheugen* − of *ruimtecomplexiteit* van een RAM programma is een functie $S(n)$ die het maximum is, genomen over alle invoer van grootte n, van de som van $k(x_i)$ over alle registers R_i, $i \geqslant 0$, waarbij x_i het grootste getal is dat op enig moment van de berekening in register R_i is opgeslagen.

VOORBEELD. Voor de tijdcomplexiteit in het voorgaande Voorbeeld geldt het volgende. De tijdkosten worden bepaald door een buitenlus en een binnenlus. Elke maal dat we de buitenlus met label L_2 doorlopen, wordt de met L_3 gelabelde binnenlus n maal doorlopen. De buitenlus zelf wordt ook n maal doorlopen, en zodoende komen we op een tijdcomplexiteit $O(n^2)$ onder het uniforme kostencriterium. De bepaling van de logaritmische tijdcomplexiteit is lastiger. De kosten om de i-de buitenlus te doorlopen zijn gelijk aan (voor i tussen 1 en n):

$$k(n-i)+k(n)+(\textit{kosten binnenlus met } n^i \textit{ in } R_3)+ \\ +k(n^{i+1})+k(n-i)+\textit{constante}\cdot k(n).$$

De kosten om de j-de binnenlus in de i-de buitenlus te doorlopen worden gegeven door (voor i tussen 1 en n):

$$k(n-j)+k(jn^i)+k(n-j).$$

Zodoende vinden we dat $T(n) \in O(n^3 \log n)$. Onder het uniforme kostencriterium is de gebruikte ruimtecomplexiteit $O(1)$ omdat we maar 6 registers gebruiken, terwijl onder het logaritmische kostencriterium de ruimtecomplexiteit $O(n \log n)$ is, daar het grootste getal in een der gebruikte 6

registers gedurende de berekening niet groter is dan n^n. Voor het beschouwde RAM programma geldt dus:

	Uniforme kosten	Logaritmische kosten
Tijdcomplexiteit	$O(n^2)$	$O(n^3 \log n)$
Ruimtecomplexiteit	$O(1)$	$O(n \log n)$

Hoe realistisch zijn de kostenmaten? De uniforme kostenmaat is realistisch, voor het onderhavige programma, indien aangenomen kan worden dat een enkel computerwoord getallen van de orde n^n kan bevatten. Is n^n groter dan wat in een enkel computerwoord past, dan wordt de logaritmische kostenmaat realistischer. Het uniforme kostencriterium is gebaseerd op de aanname dat het uitvoeren van iedere RAM instructie één tijdeenheid kost en dat ieder register in één geheugen eenheid past. Het logaritmisch kostencriterium houdt rekening met de beperkte lengte van een echt computerwoord. Daar registers willekeurig grote getallen kunnen bevatten, neemt deze kostenmaat in aanmerking dat er ongeveer $\log n$ bits nodig zijn om een getal n, bevat in een register, te representeren; en verder dat de kosten verbonden met het uitvoeren van de gebruikte instructies proportioneel zijn met de lengte van de operanden.

STELLING. *(i) Een berekening die op een RAM onder het logaritmische kostencriterium $T(n)$ tijd vergt, kan door een RAM onder het uniforme kostencriterium in $O(T(n))$ tijd uitgevoerd worden.*
(ii) Een berekening die op een RAM onder het uniforme kostencriterium $T(n)$ tijd vergt, kan door een RAM onder het logaritmische kostencriterium in $O(T(n)^2)$ tijd uitgevoerd worden.

Met betrekking tot de ruimtecomplexiteit is een zinvol analogon voor bovenstaande stelling niet te geven, omdat elke berekening uitgevoerd kan worden door een RAM die slechts twee registers gebruikt (twee register RAM's zijn equivalent met Turing machines, Minsky, 1961). Daarom hebben alle problemen ruimtecomplexiteit $O(1)$ onder het uniforme kostencriterium! Een bewijs voor dit feit wordt gegeven door middel van een simulatie algoritme die buitengewoon grote getallen genereert. Op een echte computer zouden deze grote getallen niet in één computerwoord passen, maar zouden vele computerwoorden nodig zijn om de inhoud van de betrokken twee registers te bevatten. Daarom zullen we de ruimtecomplexiteit altijd meten onder het logaritmisch kostencriterium.

4.2. P versus NP

Wanneer wij ons beperken tot algoritmen met een door een polynoom, in de grootte van de probleeminstantie, begrensde tijdcomplexiteit, doet het er in het algemeen niet toe welk (deterministisch) berekeningsmodel gekozen wordt. Om deze observatie te formaliseren, definieren we P als de klasse van berekeningen die door een RAM in polynomiale tijd, polynomiaal in de grootte van de invoer, uitgevoerd kunnen worden. Beide kostencriteria leveren dan, als gevolg van voorgaande Stelling, dezelfde klasse op. De klasse P blijkt invariant te zijn onder veranderingen van machinemodel, tot in feite ieder redelijk deterministisch model van een computer, zoals bijvoorbeeld de Turing machine. Dit fenomeen ontvangt veel aandacht, daar de klasse P beschouwd wordt als een benadering van de klasse van practisch uitvoerbare problemen. Het tot dusver behandelde RAM model wordt *deterministisch* genoemd, omdat er in ieder stadium van de berekening een éénduidig bepaalde volgende stap is. Er zijn echter goede redenen, om deze aanname te verzwakken tot het toelaten van eindig veel mogelijke volgende stappen (vergelijk de niet-deterministische varianten van de Turing machine, LBA, PDA en EA in Sectie 3). De *niet−deterministische RAM* is net zo gedefinieerd als de deterministische RAM, behalve dat we niet meer eisen dat instructies unieke labels hebben: twee of meer opdrachten mogen hetzelfde label hebben. Een berekening op zo'n niet-

deterministische RAM verloopt hetzelfde als een berekening op een deterministische RAM tot een sprongopdracht ontmoet wordt, en twee of meer instructies de label, waarnaar gesprongen wordt, gemeen hebben. De invoer wordt geaccepteerd, of leidt tot een functiewaarde, als er tenminste één toegelaten volgorde van uit te voeren instructies is die eindigt met de ACCEPT instructie. Merk op, dat als een niet-deterministische machine een toegestane volgorde van instructies afwerkt die niet in een ACCEPT instructie eindigt, dit niet betekent dat de invoer in een of andere zin afgewezen wordt. In feite geeft dit geen informatie, daar een andere volgorde wel tot de ACCEPT instructie kan leiden. Onder deze afspraak, van wat acceptatie/berekening van functiewaarde door een niet-deterministische RAM inhoudt, kunnen we de tijdcomplexiteit als volgt definieren:

Zij M een niet-deterministische RAM, A een verzameling natuurlijke getallen en $T(n)$ een functie op de natuurlijke getallen. M accepteert A onder het kostencriterium k in tijd $T(n)$ als:

(i) M precies die invoer accepteert die tot A behoort;

(ii) er voor iedere x in A een berekening (volgorde van keuzen) van M met invoer x is, die tot de ACCEPT instructie leidt, en die ten hoogste $T(n)$ tijdeenheden gebruikt onder het k kostencriterium, waarbij n de lengte van x is.

Voor het geheugengebruik $S(n)$ onder het kostencriterium k gelden *mutatis mutandis* dezelfde definities.

VOORBEELD. Stel dat we de samengesteldheid van een getal x willen bepalen. Een niet-deterministische RAM kan twee candidaat- factoren x_1 en x_2 "raden" , en het product x_1x_2 uitrekenen. Indien $x_1x_2 = x$ voor een keuze van x_1 en x_2, terwijl $x_1, x_2 \neq 1$, dan is x samengesteld. Zo kan een nietdeterministische RAM de verzameling van samengestelde natuurlijke getallen in $O(n^2)$ tijd onder het logaritmische kostencriterium accepteren. (Hierbij is n de lengte van de binaire representatie van de getallen.) Alle bekende algoritmen voor dit probleem kosten op een deterministische RAM meer dan polynomiale tijd. (Polynomiaal in de lengte van de binaire representatie van de getallen).

Het voorbeeld suggereert een alternatieve definitie voor een niet-deterministische berekening. Om een probleeminstantie op te lossen "raden" we een oplossing en verifieren die vervolgens deterministisch. Hoe verhoudt zich nu het geheugen- en tijdgebruik tussen deterministische en niet-deterministische RAM's ? Iedere niet-deterministische RAM kan door een deterministische RAM gesimuleerd worden. Alles wat de deterministische RAM moet doen is systematisch alle toegestane berekeningsvolgorden van i instructies aflopen, voor $i = 1,2, \cdots$. Zo'n simulatie kost zeer veel tijd en geheugenruimte: deterministische ruimte $c^{S(n)}$ voor niet-deterministische ruimte $S(n)$ en deterministische tijd $c^{T(n)}$ voor niet-deterministische tijd $T(n)$, waarbij c een passende constante groter dan 1 is. Wat betreft de geheugenruimte kan het beter.

STELLING (Savitch). *Als A door een niet-deterministische RAM in geheugenruimte $S(n)$ wordt geaccepteerd dan kunnen we een deterministische RAM vinden die A in geheugenruimte $O(S(n)^2)$ accepteert, mits $S(n) \geqslant \log n$.*

Aangezien geheugengebruik voor logaritmische kosten RAM's en Turing machines gelijk is, zegt de stelling dat LBA's door deterministische Turing machines in $O(n^2)$ geheugenruimte gesimuleerd kunnen worden (zie het LBA Probleem in Sectie 3). Voor het tijdgebruik is echter geen betere methode bekend dan bovenstaande simulatie. Men vermoedt dat dit het best mogelijke is. Dit vermoeden wordt vaak geuit in termen van programma's die in uitvoeringstijd, polynomiaal in de lengte van de invoer, begrenst zijn. Naar analogie van de klasse P definieren we NP als de klasse van talen die in $niet-deterministisch$ polynomiale tijd (polynomiaal in de grootte van de probleeminstantie) geaccepteerd worden. Zoals met P leiden alle redelijke niet-deterministische modellen tot dezelfde klasse NP . Per definitie geldt dat P bevat is in NP . Men vermoedt dat P echt bevat is in NP , d.w.z. $P \neq NP$. Vele belangrijke praktijk problemen blijken tot de klasse NP te behoren, en het zou nuttig zijn om deterministische polynomiale tijd algoritmen voor zulke problemen

te hebben. Er is aangetoond dat veel van deze problemen in het verschil van NP en P liggen, in geval $P \neq NP$, en er dan dus geen polynomiale tijd begrensde deterministische algoritmen voor deze problemen bestaan! In het bijzonder geldt dit voor de $NP-complete$ problemen.

Een probleem A is *polynomiaal reduceerbaar* tot probleem B als er een, in deterministische polynomiale tijd berekenbare, functie f bestaat met de volgende eigenschappen. De functie f beeldt een instantie x van probleem A op een instantie y van probleem B af, $f(x)=y$, zodanig dat een oplossing voor y door een, in deterministische polynomiale tijd berekenbare, functie g omgezet wordt in een oplossing voor x.

Een probleem B is $NP-hard$ als elk probleem A in NP polynomiaal reduceerbaar is tot probleem B.

Een probleem B is $NP-compleet$ als B NP-hard is en als bovendien B in NP ligt.

Voor NP-complete problemen A geldt dus:
(i) Het probleem A ligt in NP;
(ii) als het probleem A in P ligt dan is $P = NP$.

Voor de goede orde: als voor een probleem A bovenstaande (i) en (ii) gelden, dan hoeft A nog niet NP-compleet te zijn. Het begrip NP-compleetheid is een formalisering van de "moeilijkste" problemen in NP . Het blijkt dat verbazend veel problemen NP-compleet zijn, en dus, voor zover bekend, exponentieel veel tijd kosten op een deterministische computer.

4.3. NP-complete problemen.

In wezen zijn er twee manieren waarop een probleem NP-compleet bewezen wordt: direct of door reductie van een bekend NP-compleet probleem tot het nieuwe probleem. Het eerste probleem waarvan de NP-compleetheid bewezen werd (door S. Cook in 1971) betreft het volgende beslissingsprobleem uit de Booleaanse logica.
Zij $U = \{u_1,u_2, \ldots, u_m\}$ een verzameling van Booleaanse variabelen. Een *waarheidstoekenning* aan U is een functie $t: U \rightarrow \{T,F\}$. Als $t(u) = T$ zeggen we dat u "waar" is onder t; als $t(u) = F$ zeggen we dat u "onwaar" is onder t. Voor iedere variabele u in U zijn u en $\bar{u}$ *literals* over U. De literal u is waar onder t dan en slechts dan als de variabele u waar is onder t; de literal $\bar{u}$ is waar onder t dan en slechts dan als de variabele u onwaar is onder t.
Een *clause* over U is een verzameling literals over U, zoals $\{u_1,\bar{u}_3,u_7\}$. Zij representeert de *disjunctie* van deze literals en is *vervuld* (satisfied) door een waarheidstoekenning dan en slechts dan als tenminste één van haar elementen waar is onder de toekenning. De bovenstaande clause is waar tenzij $t(u_1) = F$, $t(u_3) = T$, en $t(u_7) = F$. Een collectie C van clausen over U is *vervulbaar* (satisfiable) dan en slechts dan als er een waarheidstoekenning aan U bestaat die alle clausen in C tegelijkertijd vervuld. Zo'n waarheidstoekenning wordt *vervullend* (satisfying) genoemd. Het probleem wordt nu als volgt gespecificeerd.

SATISFIABILITY (SAT). Instantie: Een verzameling variabelen U en een collectie van clausen C over U.
Vraag: Bestaat er een vervullende waarheidstoekenning voor C?

VOORBEELD. $U = \{u_1,u_2\}$ en $C = \{\{u_1,\bar{u}_2\},\{\bar{u}_1,u_2\}\}$ geven een instantie van SAT waarvoor het antwoord "ja" is. Een vervullende waarheidstoekenning wordt gegeven door $t(u_1) = t(u_2) = T$. Vervanging van C door $C' = \{\{u_1,u_2\},\{u_1,\bar{u}_2\},\{\bar{u}_1\}\}$ geeft echter een instantie waarvoor het antwoord "nee" is.

STELLING (Cook). *SAT is NP-compleet.*

Het is makkelijk in te zien dat SAT in NP ligt: een niet-deterministisch algoritme hoeft slechts een waardetoekenning voor de gegeven variabelen te raden, en te verifieren of de toekenning alle clausen in de gegeven collectie C vervult. Dat alle problemen in NP in deterministisch polynomiale tijd tot

SAT te reduceren zijn, wordt aangetoond door alle niet-deterministische polynomiale tijd Turing machine berekeningen te coderen in instanties van SAT .
In het algemeen wordt van een nieuw probleem de NP-compleetheid bewezen door een bekend NP-compleet probleem (polynomiaal) tot het nieuwe probleem te reduceren. De volgende zes NP-complete problemen zijn de meest gebruikte.

3-SATISFIABILITY (3-SAT). Instantie: Collectie $C = \{c_1,c_2, \ldots, c_m\}$ van clausen over een eindige verzameling U van variabelen zo dat elke clause drie literals bevat.
Vraag: Is er een waarheidstoekenning aan U die alle clausen in C vervult?

3-DIMENSIONAL MATCHING (3-DM). Instantie: Een verzameling M bevat in $W \times X \times Y$, waarbij W, X, en Y disjunct zijn en hetzelfde aantal q elementen bevatten.
Vraag: Bevat M een *matching*, d.w.z., een deelverzameling M' met q elementen waarvan er geen twee op enig coordinaat overeenkomen?

VERTEX COVER (VC). Instantie: Een graaf $G = (V,E)$ en een positief geheel getal $K \leqslant \mid V \mid$.
Vraag: Bestaat er een deelverzameling V' van V zodanig dat $\mid V' \mid \leqslant K$ en, voor iedere kant $\{u,v\}$ in E, behoort tenminste òf u òf v tot V'? Zo'n V' heet een *vertex cover*.

CLIQUE. Instantie: Een graaf $G = (V,E)$ en een positief geheel getal $J \leqslant \mid V \mid$.
Vraag: Bevat G een *kliek* van grootte J of meer, d.w.z., een deelverzameling V' van V zodat $\mid V' \mid \geqslant J$ en ieder paar knopen in V' is verbonden door een kant in E?

HAMILTONIAN CIRCUIT (HC). Instantie: Een graaf $G = (V,E)$.
Vraag: Bevat G een Hamiltoniaans circuit, d.w.z., een geordende lijst vertices $<v_1,v_2, \ldots, v_n>$ van G, met $n = \mid V \mid$, zodanig dat $\{v_n,v_1\}$ en alle $\{v_i,v_{i+1}\}$, $1 \leqslant i < n$, tot E behoren?

PARTITION. Instantie: Een eindige verzameling A en een gewicht $s(a)$ voor iedere a in A. $s(a)$ is een natuurlijk getal.
Vraag: Is er een deelverzameling A' van A zodanig dat de som van de $s(a)$'s , gesommeert over alle a's in A', gelijk is aan de som van de $s(a)$'s , gesommeert over alle a's in $A - A'$?

Bekende NP-complete of NP-harde problemen komen voor in bijv.:
- Grafentheorie: bijv. het bepalen van het chromatisch getal van een graaf;
- Netwerk ontwerp: bijv. het handelsreizigerprobleem;
- Volgorde en wachttijd problemen;
- Netwerk stroom problemen in de operationele analyse;
- Informatie-opslag en databases;
- Wiskundig programmeren: bijv. geheeltallig programmeren (ook 0-1);
- Algebra en getallentheorie;
- Spelen en puzzels;
- Logica;
- Automatentheorie: bijv. of twee reguliere expressies dezelfde verzameling voorstellen;
- Programma- en code-optimalisatie;

Het blijkt dat, behalve voor de klasse NP, ook voor andere complexiteitsklassen complete problemen te geven zijn. Tengevolge van bovengenoemde Savitch's Stelling, is geheugenruimtegebruik een complexiteitsmaat die stabiel is onder overgang van een deterministisch berekeningsmodel naar een niet-deterministisch berekeningsmodel. De klasse van problemen, die opgelost kunnen worden met gebruik van geheugen polynomiaal in de grootte van de probleeminstantie, is volgens deze Stelling dezelfde voor deterministische- en niet-deterministische algoritmen. We noemen deze klasse van problemen *PSPACE*. Bijna per definitie is de klasse NP bevat in PSPACE. De definitie van *PSPACE − compleet* is analoog met die van NP-compleet, met een passende wijziging in het reductiebegrip. Per definitie is elk PSPACE-compleet probleem NP-hard , maar een NP-compleet probleem hoeft niet PSPACE-compleet te zijn. (Tenzij bewezen zou worden dat $PSPACE = NP$, hetgeen onwaarschijnlijk is.)

VOORBEELD. Het probleem of twee reguliere expressies dezelfde verzameling voorstellen is PSPACE-compleet.

4.4. Efficiente algoritmen.

Het is bekend dat de efficientie van een algoritme in aanzienlijke mate afhangt van de voor de gegevens gebruikte datastructuren. De *linked list* biedt wat dit betreft vele mogelijkheden, evenals *stapels*, *buffers* en *gebalanceerde bomen*. We bekijken een paar algemene technieken voor het ontwerp van efficiente algoritmen.

Recursie. Een procedure die zichzelf, direct of indirect, oproept wordt *recursief* genoemd. Het gebruik van recursie stelt ons vaak in staat om korte en krachtige beschrijvingen van een algoritme te geven. Om de complexiteit van een recursieve algoritme te bepalen gebruiken we *recurrente betrekkingen*. Bijvoorbeeld, stel dat de looptijd $T(n)$ van een procedure voor het probleem van grootte n voldoet aan $T(n) = 2T(n/2)+O(n)$ en $T(1) = c$, c constant. Hieruit volgt dat $T(n) \in O(n \log n)$.

Binair zoeken. Om een specifiek element in een geschikte zoekruimte *binair* op te zoeken, verdelen we de zoekruimte in twee delen van gelijke grootte, en bepalen in welk van de twee het gezochte element ligt. Vervolgens herhalen we deze procedure in de gevonden deel-zoekruimte, enz. Voor een zoekruimte die n elementen bevat, kost dit proces $\log n$ iteraties om het gezochte element te vinden. Als bijv. n items opgeslagen zijn, met behulp van een gebalanceerde binaire zoekboom, kost elke iteratie een constant aantal stappen, en het zoeken van een element dus $O(\log n)$ stappen.

Verdeel-en-heers. Hieronder verstaat men de verdeling of opsplitsing van een probleem in twee of meer deelproblemen van kleiner formaat, waarvan de oplossingen eenvoudig samengesteld kunnen worden tot een oplossing van het oorspronkelijke probleem. Dit leidt veelal tot een recursieve algoritme. Veel sorteer-algoritmen hebben deze vorm, bijvoorbeeld het bekende *quicksort* dat $O(n \log n)$ gemiddelde tijdcomplexiteit en $O(n^2)$ slechtste geval tijdcomplexiteit heeft.

Balanceren. In het binaire zoeken èn in de verdeel-en-heers strategie is het fundamenteel dat het probleem in twee deelproblemen van gelijke grootte verdeeld wordt. Om goede algoritmen te ontwerpen is het noodzakelijk om een balans te bewaren.

Dynamisch programmeren. Recursieve technieken zijn nuttig als een probleem met gebruik van een redelijke inspanning in deelproblemen verdeeld kan worden, en de som van de grootten van de deelproblemen klein gehouden kan worden. Als echter (bijvoorbeeld) een probleem van grootte n resulteert in n deelproblemen van grootte $n-1$, dan zal een recursieve algoritme een exponentiele looptijd hebben. In zulke gevallen zal de dynamische programmeertechniek vaak tot een efficientere algoritme leiden. Dynamisch programmeren berekent de oplossing van alle deelproblemen als volgt. De berekening schrijdt voort van kleine deelproblemen tot grote deelproblemen, terwijl de antwoorden in een tabel opgeslagen worden. Het voordeel van de methode is, dat als een deelprobleem eenmaal opgelost is, het antwoord opgeslagen is en niet meer opnieuw uitgerekend hoeft te worden. Dit in tegenstelling met de recursie techniek, waarbij hetzelfde deelprobleem, telkens als het in de berekening opnieuw ontmoet wordt, ook opnieuw uitgerekend wordt.

Probabilistische algoritmen. Voor sommige *moeilijke* problemen (ook NP-complete) zijn er algoritmen te vinden die voor "de meeste", "bijna alle", dan wel "alle in de practijk voorkomende", probleeminstanties snel zijn. Men kan dan, als alternatief voor de "slechtste geval" prestatie garantie, een prestatie analyse vanuit de "gemiddelde geval" benadering uitvoeren. Een dergelijke analyse berust vaak op empirisch onderzoek. De vereiste probabilistische analyses zijn vaak moeilijk, omdat we meestal niet weten wat de waarschijnlijkheidsverdeling (betreffende het in de practijk voorkomen) van de probleeminstanties is. Een voorbeeld is de *Simplex* methode om lineaire programmeerproblemen op te lossen. Zij werkt in de practijk vrijwel altijd (lineair) snel maar kan theoretisch gesproken niet (voor alle gevallen) sneller dan exponentieel *langzaam* gegarandeerd worden.

Een geheel andere methode, zoals door M. Rabin eind zeventiger jaren op diverse problemen toegepast werd, is een algoritme te leveren die altijd snel is, maar waarvan het antwoord niet geheel betrouwbaar is. Als echter elke maal, dat de algoritme op dezelfde probleeminstantie uitgevoerd wordt, zij met een kans van meer dan 1/2 een juist antwoord geeft, en de gebeurtenissen statistisch gesproken onafhankelijk zijn (door bij elke executie van het algoritme een random factor in te bouwen, waarvan de waarschijnlijkheidsverdeling bekend is), dan is de kans dat hetzelfde antwoord, na 100 maal de algoritme uit te voeren, nòg fout is kleiner dan 2^{-100} . Een toepassing van dit idee wordt gegeven door zeer snelle probabilistische primaliteitstesten.

Ook de door J. von Neumann in de prehistorie van het computer tijdperk ontwikkelde *Monte Carlo* methoden zijn een fraai voorbeeld van probabilistische algoritmen.

VOORBEELD. Een mooi voorbeeld van een efficiente algoritme wordt gevormt door Strassen's matrixvermenigvuldigingsalgoritme. Zij illustreert diverse van de bovengenoemde technieken. Om twee $n \times n$ matrices over de reële getallen met elkaar te vermenigvuldigen heeft de standaard algoritme $O(n^3)$ vermenigvuldigingen nodig. In 1969 bewees Strassen dat asymptotisch $O(n^{2,81})$ vermenigvuldigingen genoeg zijn. Dit volgt uit een slimme methode om twee 2×2 matrices, met elementen over een willekeurige ring, in 7 vermenigvuldigingen te vermenigvuldigen, en deze methode recursief toe te passen. Om het matrixproduct

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix},$$

uit te rekenen bepalen we eerst:

$$\begin{aligned} p_1 &= (a_{12}-a_{22})(b_{21}+b_{22}), \\ p_2 &= (a_{11}+a_{22})(b_{11}+b_{22}), \\ p_3 &= (a_{11}-a_{21})(b_{11}+b_{12}), \\ p_4 &= (a_{11}+a_{12})b_{22}, \\ p_5 &= a_{11}(b_{12}-b_{22}), \\ p_6 &= a_{22}(b_{21}-b_{11}), \\ p_7 &= (a_{21}+a_{22})b_{11}. \end{aligned}$$

Bereken dan de c's volgens:

$$\begin{aligned} c_{11} &= p_1+p_2-p_4+p_6, \\ c_{12} &= p_4+p_5, \\ c_{21} &= p_6+p_7, \\ c_{22} &= p_2-p_3+p_5-p_7. \end{aligned}$$

We hebben nu de product matrix in 7 vermenigvuldigingen en 18 optellingen/aftrekkingen uitgerekend. De matrixelementen kunnen over een willekeurige ring gekozen worden, daar het bovenstaande schema dit toelaat. In het bijzonder kunnen we de elementen kiezen uit een ring van matrices. Beschouw nu het vermenigvuldigen voor twee $n \times n$ matrices met $n = 2^k$. Zij $T(n)$ het aantal aritmetische operaties dat nodig is. Door elke matrix in vier $n/2 \times n/2$ matrices te verdelen, en bovenstaand schema toe te passen blijkt:

$$T(n) = 7T(n/2)+18(n/2)^2 \text{ , voor } n \geqslant 2.$$

(De term $18(n/2)^2$ volgt uit de 18 optellingen/aftrekkingen, van $n/2 \times n/2$ matrices, nodig om de c-matrices te bepalen.) Hieruit volgt dat:

$$T(n) \in O(7^{\log n}) = O(n^{\log 7}) \approx O(n^{2,81}) .$$

(Als n niet een macht van twee is, bedden we de matrices in grotere matrices, waarvan de dimensies gelijk zijn aan de volgende tweemacht, in.)

GESELECTEERDE LITERATUUR

Gebruikt bij het vervaardigen van deze text:

Aho, A. V., J.E. Hopcroft & J.D. Ullman, *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.

Garey, M.R., & D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* H. Freeman, San Francisco, 1978.

Hopcroft, J.E.,& J.D.Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass., 1979.

Standaardwerk over algoritmen, datastructuren en aanverwante onderwerpen in meerdere volumes waarvan, op tijd van dit schrijven, de volgende verschenen zijn:

Knuth, D.E., *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968 (2e herziene editie 1973).

Knuth, D.E., *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 1969 (2e herziene editie 1981).

Knuth, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, Mass.,1973.

Toegepaste Automaten theorie:

Kohavi, Z., *Switching and Finite Automata Theory.* Computer Science Series, McGraw-Hill, Inc., 1970 (2e herziene editie 1978).

Een klasse apart vormt het buitengewoon populaire boek van D.R. Hofstadter, *Gödel, Escher, Bach, An Eternal Golden Braid.* Harvester Studies in Cognitive Science, **12**, Hassocks, Harvester, 1979. Ook uitgegeven bij Penguin Books en in nederlandse vertaling. Via tal van "analogiën" met het werk van de graficus M.C. Escher en de componist J.S. Bach tracht Hofstadter de, voor de Informatica zo belangrijke, fundamentele onbeslisbaarheidsresultaten van de wiskundige logicus K. Gödel voor de geinformeerde leek te verhelderen. Het boek heeft in ieder geval de verdienste dat het de ideëen, die ten grondslag liggen aan de informatica, een wijdere verbreiding gegeven heeft. Dit onverlet de bezwaren die men kan hebben betreffende verkeerde analogiën, pijnlijke humor, en misleidende uitspraken.