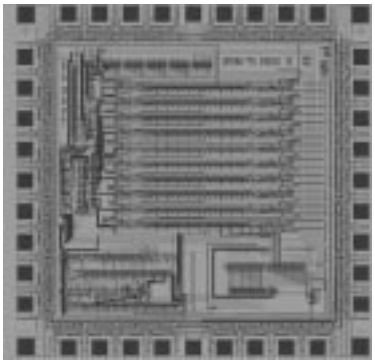
$\mathcal{P} = \mathcal{NP}$?

Frits Beukers

1. Inleiding

De eerste electronische computer, de ENIAC(="Electronic Numeric Integrator and Calculator")) werd in 1946 in gebruik genomen en luidde het tijdperk van electronisch rekenen in. Deze kolos bevatte maar liefst 18.800 radiobuizen en was in staat om zo'n 5000 elementaire bewerkingen per seconde uit te voeren. Voor die tijd een onvoorstelbare snelheid. Maar de tijd heeft niet stilgestaan.

ENIAC, 1946



ENIAC-ON-A-CHIP, 1996

Tegenwoordig bevat een gemiddelde processor in een PC een miljoen of meer logische componenten en is in staat tot enkele miljoenen berekeningen per seconde. Hiermee vergeleken was ENIAC slechts kinderspel. De enorme explosie van rekenkracht die we de afgelopen jaren om ons heen gezien hebben, heeft een onvoorstel effect op onze maatschappij gehad. Hoezeer computers in onze maatschappij verweven zijn moge blijken uit onze zorgen over het millenniumprobleem. Ook in de wereld van het wetenschappelijk rekenen hebben we zo'n omwenteling meegemaakt. Met computers kunnen we tegenwoordig weersvoorspellingen doen, windtunnels simuleren, stromingen van de oceaan berekenen, sterrenstelsels laten botsen, optimale strategiën voor grote organisaties bepalen, het dienstrooster van de NS samenstellen,
medische scans zichtbaar maken, en ga zo maar door. Dit alles vestigt de indruk dat als er zaken zijn die we nu niet kunnen berekenen, dat we dit over een aantal jaren wel zouden moeten kunnen.
Merkwaardig genoeg blijken bij onze pogingen om de grenzen van onze mogelijkheden te verleggen er nieuwe onverwachtte barrières op te duiken. In de natuurkunde stuiten we regelmatig op dat soort muren. Materie blijkt ondeel-

baar op atomair- of quarkniveau, niets kan sneller gaan dan het licht, en we kunnen snelheid en positie van een deeltje niet beiden precies kennen (Heisenberg's onzekerheidsrelatie). Dergelijke begrenzingen kunnen heel onaangenaam zijn, maar ook bijzonder boeiend. Op het gebied van berekenbaarheid blijken er ook zulke nieuwe barrières op te doemen. Een heel bekend voorbeeld is dat van de *chaostheorie* met als belangrijkste voorbeeld de weersvoorspelling. De differentiaalvergelijkingen die het gedrag van de atmosfeer bepalen laten zien dat onnauwkeurigheden in de beschrijving van de beginsituatie zich in de loop der dagen exponentieel vermenigvuldigen, totdat na een week deze uitvergrote onnauwkeurigheden de werkelijke beschrijving totaal overheersen. Om deze reden gaan huidige weersvoorspellingen niet veel verder dan een week. Willen we de weersvoorspelling een dag verder uitstrekken, dan moet de nauwkeurigheid van de beginsituatie met een factor, zeg 2, verbeterd worden. Voor iedere volgende dag moeten we de begintoestand een factor 2 nauwkeuriger kennen. De toestand van de atmosfeer met een dergelijke nauwkeurigheid opmeten wordt al snel onmogelijk, afgezien van het probleem de vereiste berekeningen uit te moeten voeren. We hebben te maken met een chaotisch systeem. Reeds aan het begin van deze eeuw werd een dergelijk gedrag door de beroemde wiskundige H.Poincaré voorspeld. Nu we computers tot onze beschikking hebben, worden we met deze harde werkelijkheid geconfronteerd. Wie belangstelling heeft voor dit onderwerp, raad ik aan het boekje [BCV] (zie referenties achterin) te lezen. In dit stuk willen we echter de berekenbaarheid bespreken van een heel ander soort problemen dan chaotische dynamische systemen. Wij zullen het hier hebben over gehele getallen, lijsten van symbolen en andere discrete zaken. Hier zijn een paar voorbeelden.

Sorteerproblemen Gegeven een lijst woorden, bijvoorbeeld een miljoen stuks. Sorteer ze op alfabet.

Deelsomproblemen Gegeven een rij X van n natuurlijke getallen en een natuurlijk getal S. Ga na of er een deelrij van X bestaat waarvan de som van de elementen gelijk aan S is.

Ontbinden in factoren Gegegeven een natuurlijk getal. Bepaal zijn ontbinding in priemfactoren.

Handelsreiziger probleem Gegegeven een aantal steden waarvan de onderlinge afstanden bekend zijn. Bepaal het kortste gesloten pad dat elk van deze steden precies éénmaal bezoekt.

Schaakprobleem Gegeven een stelling in een schaakpartij. Bepaal of wit in gewonnen positie staat.

Bovenstaande problemen hebben allemaal als gemeenschappelijk kenmerk dat er een oplosmethode is die bestaat uit het uitproberen van alle mogelijkheden. Zo zouden we het handelsreizigerprobleem kunnen oplossen door alle gesloten paden op te schrijven, hun lengte te bepalen, en de kortste daarvan als oplossing te nemen. Een getal van 200 cijfers kunnen we ontbinden door gewoon alle

getallen van 2 tot 10^{100} als deler te proberen. Of deze oplosmethoden efficiënt zijn is weer een heel andere kwestie. Voor een oplosmethode is het namelijk niet alleen belangrijk dat hij een antwoord oplevert, maar ook dat dit gebeurt binnen een tijd waarin het antwoord nog voor ons van nut is. Aan een oplosmethode die eeuwen in beslag neemt hebben we niets. De vraag naar het bestaan van efficiënte methoden zal de leidraad van deze voordracht zijn. We zullen echter eerst beginnen met het vastleggen van wat begrippen.

2. Computers en algoritmen

Het eerste onderzoek naar berekenbaarheid werd al in de dertiger jaren gedaan door de wiskundige Alan Turing. In die tijd bestonden er nog geen electronische computers. Men kende wel mechanische rekenmachines en men had ook een primitief begrip van de mogelijkheid zo'n rekenmachine een bepaald rekenschema of programma te laten volgen. Voor zijn onderzoekingen maakte Turing gebruik van een denkbeeldige machine die later bekend is geworden onder de naam (universele) *Turing machine*. We laten voortaan het woordje "universeel" weg. Het getuigt van de visie van Turing dat onze tegenwoordige computers in principe realisaties zijn van een Turing machine, met als enig verschil dat een Turing machine over onbeperkt veel geheugen kan beschikken. Door deze overeenkomst zijn Turing's beschouwingen ook van toepassing op onze tegenwoordige computers. Sterker nog, een Turing machine, een PC of een supercomputer zijn equivalent in die zin dat ze zijn in staat tot het uitvoeren van dezelfde soort berekeningen. Het enige verschil is natuurlijk dat de ene machine langer over een bepaalde berekening zal doen dan de ander. Echter, met betrekking tot de vraag wèlke berekeningen uitgevoerd kunnen worden zijn alle drie machines gelijkwaardig. Zelfs een ijverige klerk die trouw alle instructies opvolgt zou ook aan het Turing model voldoen. Door deze uitwisselbaarheid van machinetypes is het nu niet meer nodig Turing's definitie van zijn machine uit te leggen. We kunnen gewoon denken aan een computer met onbeperkt veel geheugen.

Om computers problemen te laten oplossen hebben we *algoritmen* nodig. Een algoritme is een computerprogramma dat een invoer leest en vervolgens uitvoer produceert die het antwoord op ons probleem geeft. Iedereen heeft wel eens een computerprogramma gezien. Het is altijd een eindige lijst met instructies. Voor ons zijn die instructies niet altijd even helder, voor de computer moeten ze ondubbelzinnige opdrachten voorstellen die één voor één uitgevoerd moeten worden door de processor.

Het soort problemen dat we bekijken heeft trouwens altijd *parameters* in zich. Kijk maar naar de lijst van voorbeeldproblemen uit de vorige paragraaf. In het ontbindingsprobleem gaat het om een probleem om een getal n te ontbinden. Het getal n is de parameter. Bij het sorteeralgoritme is de parameter de te sorteren lijst van woorden. In al deze problemen vormen de parameters de invoer voor het eventuele algoritme dat het probleem op moet lossen. De uitvoer, die op een beeldscherm verschijnt of op papier wordt uitgeprint, kan ook verschillende vormen hebben. Bij het sorteerprobleem is dat gesorteerde lijst, bij het schaakprobleem of deelsomprobleem is dat een simpel "ja" of "nee", al naar

gelang de positieve of negatieve uitkomst. Problemen met alleen "ja" of "nee" als uitvoer zullen voor ons van bijzondere interesse zijn. We noemen ze *beslissingsproblemen*. Het sorteerprobleem is duidelijk geen beslissingsprobleem. Het aardige is echter dat ontbinding in factoren daarentegen wel geformuleerd kan worden als een beslissingsprobleem.

Ontbinding als beslissingsprobleem Gegeven een tweetal positief gehele getallen n, m met $m \leq n$. Ga na of n een echte (dwz $\neq 1, n$) deler $< m$ heeft.

Als we een algoritme hebben om het oorspronkelijke ontbindingsprobleem op te lossen, dan kunnen we bovenstaand probleem uiteraard ook oplossen. Stel nu omgekeerd dat we een algoritme hebben om bovenstaand beslissingsprobleem op te lossen. Dan kunnen we het oorspronkelijke ontbindingsprobleem ook oplossen. We zoeken namelijk de kleinste niet-triviale deler van n door middel van een binaire zoekactie, bij velen beter bekend als het "hoger-lager" spel. Dit spel bestaat eruit dat we, gegeven n, iemand een getal $x < n$ laten raden door alleen met "ja" of "nee" op zijn vragen te antwoorden. De vragen mogen alleen van het type 'is x kleiner dan y?' zijn waarin y een willekeurig door de vraagsteller te benoemen getal is. De handigste manier is ervoor te zorgen dat we het interval waarin x moet liggen bij elke vraag in lengte halveren. Te beginnen bij het interval $[1, n]$. Op deze manier kan de vraagsteller met $\log_2(n) + 1$ vragen klaar zijn.
We kunnen dit spelletje spelen met de niet-triviale delers van een getal n. Geef deze verzameling aan met S. Het algoritme om ons beslissingsprobleem op te lossen speelt de rol van een orakel dat "ja" of "nee" antwoordt op de vraag of S een element $< m$ bevat. Neem als voorbeeld $n = 989$. Als er een echte deler is dan is deze kleiner of gelijk $\sqrt{989} < 32$. De eerste vraag is 'is er een echte deler van n kleiner dan 32?'. Antwoord 'ja'. We zijn op zoek naar de kleinste niet-triviale deler. Dus stellen we de volgende vraag: 'is er een echte deler van n kleiner dan 16?'. Antwoord 'nee'. We weten dus dat $16 \leq d < 32$ waarin d de kleinste echte deler van n is. Hier volgt een kleine tabel van de rest van het vraag-en-antwoord spel.

vraag	antwoord
$d < 24$	ja
$d < 20$	nee
$d < 22$	nee
$d < 23$	nee

Uit de laatste regel concluderen we dat $d = 23$.
Met enige moeite kan op soortgelijke wijze ook het handelsreizigerprobleem geherformuleerd worden als beslissingsprobleem. Voor de details ervan verwijzen we naar [LLRS]
We leggen hier de nadruk op het begrip beslissingsprobleem omdat het een grote rol speelt in de bespreking van het probleem $\mathcal{P} = \mathcal{NP}$? Voordat het zover is besteden we eerst nog wat aandacht aan een merkwaardige klasse van problemen die geen oplossing hebben.

3. Onoplosbare problemen

De problemen die we in de de inleiding noemden hadden allen als eigenschap dat er een algoritme voor hun oplossing bestaat. Ontbinding van een getal n in factoren zou in principe kunnen gebeuren door alle gehele getallen $< \sqrt{n}$ als deler te testen. Het handelsreiziger probleem kunnen we oplossen door alle gesloten paden op te schrijven, hun lengte uit te rekenen, en vervolgens het pad met minimale lengte uit te kiezen. Dit zijn allemaal procedures die in eindige tijd uitgevoerd kunnen worden. Of er ook efficientere oplosmethoden bestaan is een zaak die in de volgende paragrafen aan de orde komt.
In deze paragraaf kunnen we het niet nalaten om een aantal problemen te noemen waarvan het helemaal niet voor de hand ligt dat er een algoritme voor hun oplossing bestaat. Hier zijn twee beroemde voorbeelden. Merk op dat het beide beslissingsproblemen zijn.

Halting probleem (Turing). Bestaat er een algoritme dat voor elk computerprogramma en voor elke gegeven invoer, kan beslissen of het oneindig lang blijft doorgaan, of dat het daadwerkelijk stopt (termineert). Zoals we weten kan een computerprogramma oneindig lang door blijven gaan als het programma in een zogenaamde loop terechtkomt. Een eenvoudig voorbeeld hiervan is

`Domprogramma`()

begin

(a) Ga naar (a)

(b) STOP

end

Hoewel er een STOP-instructie in dit programma staat, blijft het bij uitvoering in de eerste regel "hangen". Uiteraard is dit een zeer naïef voorbeeld. Veel loops komen op een veel ingewikkelder manier tot stand en voor software testers zou het ideaal zijn om een algoritme te hebben dat het bestaan van deze verborgen loops aan het licht brengt.

Hilbert's tiende probleem Bestaat er een algoritme dat voor elke diophantische vergelijking beslist of het een geheeltallige oplossing heeft? Een diophantische vergelijking is een vergelijking van het type

$$F(x_1, \ldots, x_n) = 0$$

waarin F een polynoom met gehele coëfficienten is in de n variabelen $x_1, \ldots, x_n$ is. Het woord diophantisch heeft betrekking op het feit dat we voor deze vergelijkingen alleen de geheeltallige oplossingen zoeken. De bekendste illustratie van het feit dat diophantische vergelijkingen moeilijk zijn is het laatste probleem van Fermat waarover elders in deze syllabus meer wordt gezegd (zie de bijdrage van R.Tijdeman). Voor de aardigheid zou de

lezer kunnen stoeien met het probleem om aan te tonen dat $y^2 = x^3 - 1$ geen geheeltallige oplossingen x, y heeft.

Een ander saillant detail is dat veel bekende problemen uit de wiskunde kunnen worden geherformuleerd als een diophantische vergelijking. Het bekendste voorbeeld hiervan is het Riemann-vermoeden (zie de bijdrage van R.Tijdeman in dit boekje). De juistheid van dit vermoeden is equivalent met het bestaan van een oplossing van een contrueerbare, maar zeer ingewikkelde, diophantische vergelijking (voor deze constructie zie [****]). Het bestaan van een positief antwoord op Hilbert's tiende probleem zou in het bijzonder impliceren dat er een mechanische oplossing voor het Riemann-vermoeden bestaat.

In de dertiger jaren maakte Turing de volgende schokkende ontdekking.

STELLING 3.1 *Er bestaat geen algoritme om het Halting Probleem op te lossen.*

Deze ontdekking was schokkend omdat dit de eerste stelling in de wiskunde was die het niet-bestaan van een algoritme aantoonde. In de tweede plaats was het een grote verrassing dat men überhaupt een dergelijke non-existentie kon aantonen. Turing's bewijs is echter geniaal, en bovendien makkelijk te presenteren. We geven het hier. Om te beginnen beperkte Turing zich tot algoritmen die positieve gehele getallen als invoer accepteren. Bij veel problemen hebben we al gehele getallen als invoergegeven, maar ook als dat niet zo dan geeft dit geen beperking. Dit berust op de volgende stelling.

STELLING 3.2 *Zij S een eindige verzameling van symbolen. Zij $\mathcal{S}$ de verzameling van eindige rijtjes elementen van S. Dan is $\mathcal{S}$ aftelbaar. Dat wil zeggen, we kunnen de elementen van $\mathcal{S}$ op een rij $S_1, S_2, S_3, \ldots, S_n \ldots$ zetten zó dat elk element van $\mathcal{S}$ in deze rij voorkomt.*

Bewijs We kunnen de verzameling S zien als een alfabet en $\mathcal{S}$ als de verzameling woorden gemaakt met dat alfabet. Het is nu heel makkelijk om alle elementen uit $\mathcal{S}$ op een rij te zetten. Omdat het alfabet eindig is zijn er van elke gegeven lengte L een eindig aantal woorden. We schrijven nu eerste alle woorden van lengte 1 op, vervolgens alle woorden van lengte 2, dan alle woorden van lengte 3, etcetera. Op deze manier weten we zeker dat elk woord op den duur ook inderdaad aan bod komt. Om de rangschikking nog wat ondubbelzinniger te maken zouden we nog een alfabetische volgorde voor de elementen van S kunnen afspreken en binnen elke lengte de woorden alfabetisch rangschikken. **qed**

Elke invoer bestaat uit (hoofd)letters, cijfers, spaties en andere leestekens. Een eindige collectie van tekens dus. Volgens bovenstaande Stelling kunnen we nu alle mogelijke invoeren op een rij zetten. Vervolgens kan elke invoer met zijn rangnummer worden aangegeven. We gebruiken dit rangnummer in ons bewijs als invoer van onze algoritmen.
Wat voor invoer geldt, geldt ook voor algoritmen. Ook deze kunnen we keurig netjes op een rij zetten die we aangeven met $A_1, A_2, A_3, \ldots$. Een algoritme

A_k waar we input l aan geven noteren we met $A_k(l)$ en met deze twee data (algoritme plus invoer) kan de computer aan de slag.
Stel nu dat er een algoritme $\mathcal{A}$ is dat van een gegeven algoritme met gegeven invoer kan beslissen of het termineert of niet. Beschouw nu het volgende algoritme,

`catch`$(j \in \mathbb{N})$

begin

(a) Als $A_j(j)$ termineert volgens $\mathcal{A}$, ga dan in een ONEINDIGE LOOP.

(b) Als $A_j(j)$ niet termineert volgens $\mathcal{A}$, dan STOP.

end

Omdat `catch` een algoritme is, heeft het een rangnummer. Zeg `catch` $= A_r$. Stel dat $A_r(r) =$ `catch`(r) termineert. Uit de code van het algoritme `catch` zien we dat we blijkbaar de STOP-instructie bereikt hebben, met andere woorden, $\mathcal{A}$ heeft besloten dat $A_r(r)$ niet termineert. Dit is in tegenspraak met onze aanname dat $A_r(r)$ termineert. Stel nu dat $A_r(r) =$ `catch`(r) niet termineert. In ons algoritme `catch` zien we dat `catch` in regel (a) blijft hangen en dit komt doordat $\mathcal{A}$ besloten heeft dat $A_r(r)$ termineert. Wederom een tegenspraak! Wat de uitslag van $A_r(r)$ ook is, altijd komen we op een tegenspraak uit. Blijkbaar is het bestaan van $\mathcal{A}$ een onmogelijkheid en daarmee is ook Turing's stelling bewezen.
Het tiende probleem van Hilbert is er één uit D.Hilbert's lijst van 23. Hilbert was rond 1900 één van de bekendste wiskundigen, en in 1900 gaf hij tijdens het wereldcongres voor wiskundigen een voordracht over de wiskundeproblemen die een uitdaging vormden voor de komende eeuw. Een groot aantal problemen van de lijst is in deze eeuw daadwerkelijk opgelost, waaronder ook het tiende probleem. In 1970 toonde Matijasevich, na veel voorbereidend werk van Davis en Robinson, de volgende stelling aan.

STELLING 3.3 (MATIJASEVICH) *Er bestaat geen algoritme dat van een willekeurige diophantische vergelijking beslist of er gehele oplossingen zijn of niet.*

De technieken die hiervoor gebruik zijn vele malen ingewikkelder dan die in het bewijs bij Turing's Halting Probleem. De laatste stap van het bewijs bevat echter weer Turing's idee. Voor wiskundigen zou Matijasevich's resultaat heel geruststellend moeten zijn. Blijkbaar kan het oplossen van diophantische vergelijkingen, en in het bijzonder het Riemann-vermoeden, niet aan computers worden overgelaten. Voor elk nieuw type vergelijking zijn er weer nieuwe ideeën nodig, en dit is wat veel wiskunde zo boeiend maakt.
Tenslotte nog een opmerking om misverstanden te vermijden. Het feit dat er geen algemeen algoritme voor het Halting probleem en Hilbert's tiende probleem bestaat, betekent niet dat alle instanties van deze problemen onoplosbaar zijn. In het geval van `domprogramma` zagen we immers meteen dat het niet termineert. Ook van de diophantische vergelijking $x^2 + y^2 + z^2 = -1$ zal

meteen duidelijk zijn dat het geen oplossingen heeft. Zo zijn er ook talloze andere, wellicht lastiger, gevallen waarvoor we een oplossing kunnen vinden. Het enige wat in ons bovenstaande verhaal beweerd hebben is dat er geen algoritme bestaat dat àlle gevallen aankan. Eem zelfde soort opmerking geldt ook als we het later over moeilijke problemen gaan hebben. Niet alle instanties van een moeilijk probleem hoeven perse moeilijk te zijn.

4. Makkelijke problemen

Gelukkig bestaan er ook problemen die gemakkelijk oplosbaar zijn. Het succes van de computer is voor een groot deel aan dit soort problemen te danken. Neem bijvoorbeeld de sorteer algoritmen die in elk database programma gebruikt worden. Stel we hebben een lijst van L woorden, waarbij L in grote databases kan oplopen tot een miljoen of meer. Gevraagd wordt deze lijst op alfabet te sorteren. Als het woord v in de alfabetische rangschikking aan w voorafgaat noteren we dit als $v < w$. We kunnen ook zeggen dat v vroeger is dan w. Een heel voor de hand liggend algoritme om onze lijst te sorteren zou het volgende zijn. Kies uit de lijst het vroegste woord, d.w.z. het woord dat het eerst in de alfabetische volgorde aan bod komt. Haal dit woord uit onze lijst en zet dit in de uitvoer, bijvoorbeeld naar een printer. Voor het vinden van het vroegste woord zijn $L-1$ tests van het type $a < b$? nodig. Uit de overgebleven lijst kiezen we weer het vroegste woord en herhalen het proces. Voor deze stap zijn $L-2$ tests nodig. Zo doorgaand zal na hooguit $(L-1)L+(L-2)+\cdots+2+1 = L(L-1)/2 < L^2$ tests de lijst op volgorde gezet zijn. Nu is L^2 nog een onacceptabel groot getal als $L = 10^6$, maar gelukkig zijn er ook slimmere algoritmen die het klusje in hooguit $c \cdot L\ln(L)$ stappen klaren, waarin $c > 0$ een constante is. We noteren dit vaak door te zeggen dat er sorteeralgoritmen van de orde $O(L\ln(L))$ zijn. Het symbool O noemen we het *orde symbool*.

Een ander simpel probleem is het optellen of vermenigvuldigen van twee positieve gehele getallen a, b. Normaal gesproken is dit een fluitje van een cent, maar als a, b uit enkele honderden tot duizenden cijfers gaan bestaan, dan wordt het wat anders. Om te zien hoeveel tijd een optelling van twee getallen in beslag kan nemen moeten we

afspreken wat een elementaire bewerking is. Dat wil zeggen een basisbewerking die een vaste maximale tijd in beslag neemt. We nemen hiervoor de optelling van twee cijfers van elk van de getallen. Stel dat a, b uit maximaal L cijfers bestaan. Op de lagere school hebben we geleerd dat we voor de optelling van a, b dan ook L maal twee cijfers moeten optellen. De tijd die zo'n optelling duurt is dus hooguit $c \cdot L$ waarin c een positief getal is, die aangeeft hoe lang een computer over één elementaire bewerking doet. Omdat de waarde van c ons voor dit verhaal niet interesseert gebruiken we het ordesymbool O en zeggen we dat de looptijd van het optelalgoritme van de orde $O(L)$ is. Evenzo hebben we voor de vermenigvuldiging van a, b maximaal L^2 elementaire vermenigvuldigen van cijfers nodig, en daarna nog eens L^2 optellingen. Het vermenigvuldigingsalgoritme van de lagere school is dus van de orde $O(L^2)$. Overigens is het mogelijk door slimme technieken uit de Fourieranalyse (Schönhage-Strassen) deze loop-

tijd te verbeteren tot $O(L\ln(L))$, maar dit algoritme valt ver buiten het bestek van dit verhaal. Het zal verder hopelijk duidelijk zijn dat als $N = \max(a, b)$, het verband met L gegeven wordt door $\log_{10} N < L \leq \log_{10} N + 1$. De looptijd van het naïeve vermenigvuldigingsalgoritme is dus $O(\log_{10}(N)^2) = O(\ln(N)^2)$. Algoritmen van het bovenstaande type hebben de eigenschap dat hun looptijd *polynomiaal* in de lengte van de invoer L is. Dat wil zeggen, hun looptijd is van de orde $O(L^a)$ voor zekere $a > 0$. Bij het naïeve optelalgoritme is $a = 1$, bij het naïeve sorterenen vermenigvuldigen is $a = 2$.
Het is lang niet bij alle problemen duidelijk of er een oplossing bestaat met een polynomiale looptijd. Neem bijvoorbeeld het probleem om te testen of een getal N priem is. Een methode zou zijn om voor $d = 2, 3, \ldots, [\sqrt{N}]$ te testen of d een deler van N is. Als dat niet zo is dan is N priem. Een dergelijke methode neemt maximaal $\sqrt{N}$ stappen in beslag en is dus van orde $O(\sqrt{N})$. Dit is echter geen polynomiale methode. De lengte van de invoer is immers het aantal cijfers L van N en dat is ongeveer $\log_{10}(N)$. Dus de looptijd van onze naïeve priemtest is dus $O(10^{L/2})$. De looptijd hangt nu exponentieel af van de invoerlengte, en dat is heel vervelend. Exponentiële functies hebben de neiging razend snel te groeien. Dit kunnen we mooi in de volgende tabel zien, waarin een aantal polynomiale functies en exponentiële functies met elkaar vergelijken. In deze, tabel ontleend aan [GG], nemen we $c = 10^{-6}$ seconde als tijd nodig voor één basisbewerking.

L	10	20	30	40	50
L^2	0.0001 sec	0.0004 sec	0.0009 sec	0.0016 sec	0.0025 sec
L^3	0.001 sec	0.008 sec	0.027 sec	0.064 sec	0.125 sec
L^5	0.1 sec	3.2 sec	24.3 sec	1.7 min	5.2 min
2^L	0.001 sec	1 sec	17.9 min	12.7 dag	35.7 jaar
3^L	0.059 sec	58 min	6.5 jaar	3855 eeuw	2×10^8 eeuw

Uit deze tabel zien we dat bij exponentiële looptijden een kleine toename in de invoerlengte een gigantische, bijna explosieve toename van de maximale looptijd inhoudt. In het bijzonder zien we dat het naïeve priemtest algoritme voor getallen van 50 cijfers tot in de eeuwigheid kan duren!
Hopelijk is het nu begrijpelijk waarom we graag algoritmen met polynomiale looptijd willen hebben. We maken hier meteen de aantekening bij dat een polynomiaal algoritme nog niet alles zaligmakend hoeft te zijn. Een beroemd voorbeeld hiervan is het *lineaire programmeringsprobleem*, dat in polynomiale tijd oplosbaar is (Khachian, 1978). De theorie achter lineaire programmering werd vlak na de tweede wereldoorlog door G.B.Dantzig ontwikkeld en had tot doel de ingewikkelde logistiek van het Amerikaanse leger te optimaliseren. Later volgden er ook toepassingen in de economie en de pioniers op dit gebied, Koopmans en Kantorowitz, hebben voor hun werk de nobelprijs in de economie ontvangen. Kort gezegd komt het lineaire programmeringsprobleem erop neer dat we $x_1, x_2, \ldots, x_n$ moeten bepalen zó dat een lineaire functie van de vorm

$$c(x_1, \ldots, x_n) = c_1 x_1 + c_2 x_2 + \cdots + c_n x_n$$

een maximale waarde heeft onder een aantal beperkende condities van de vorm

$$\begin{aligned} x_1, x_2, \ldots, x_n &\geq 0 \\ a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &\leq b_2 \\ &\cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_n \end{aligned}$$

De functie c zou bijvoorbeeld een winstfunctie kunnen zijn en $x_1, x_2, \ldots, x_n$ investeringen in diverse activiteiten. De ongelijkheden staan model voor de beperkingen waaraan we in het dagelijks leven allemaal bloot aan staan. Op de middelbare school worden lineaire programmeringsproblemen voor $n = 2$ nog wel eens geformuleerd en met grafieken opgelost. In de praktijk werkt men vaak met tientallen of honderden variabelen en wordt het zogenaamde *simplexalgoritme* gebruikt. Dit algoritme is tamelijk eenvoudig te begrijpen en op een computer te programmeren. Vandaar dat het simplexalgoritme één van de vaakst gebruikte algoritmen in de praktijk is. Het is echter heel lastig om een goede afschatting van de looptijd van het algoritme te geven. In de praktijk, op doorsnee problemen, lijkt het zich polynomiaal te gedragen t.a.v. het aantal variabelen n. Echter, men is er in geslaagd om gekunstelde voorbeelden te maken waarin de looptijd van het simplexalgoritme exponentiëel met n toeneemt. Het aardige is dat deze gevallen in de praktijk zo zelden voorkomen, dat we er in de regel geen last van hebben. In 1978 slaagde de rus Khachian erin een nieuw algoritme (de zogenaamde *ellipsoïde methode*) te geven dat het lineaire programmeringsprobleem aantoonbaar in polynomiale tijd oplost. Dit was een opzienbarende vondst die aantoonde dat lineaire programmering een polynomiaal, dus makkelijk op te lossen, probleem is. Ironisch genoeg gebruikt men echter nog steeds het simplex-algoritme in plaats van het polynomiale ellipsoïde algoritme. Reden hiervoor is de grote eenvoud van het simplex-algoritme. De voordelen van dit makkelijk te onderhouden en meestal snelle algoritme wegen op tegen de paar lastig op te lossen gevallen. Die neemt men graag voor lief.
Een andere kanttekening die we bij polynomialiteit van een algoritme kunnen plaatsen is dat een algoritme met looptijd $O(L^{10})$, dus met een hoge exponent, ook niet echt prettig is. Praktisch gezien zal een dergelijk algoritme ook veel te veel tijd gaan kosten voor grotere invoerlengtes L.
Daar staat tegenover dat de klasse van polynomiale algoritmen een zeer belangrijke rol speelt in theoretische beschouwingen over efficiëntie van algoritmen. Eén mooie eigenschap is dat polynomialiteit van een algoritme tamelijk ongevoelig is voor het soort computer of computermodel dat gebruikt wordt.
Voor theoretische beschouwingen zullen we trouwens alleen naar beslissingsproblemen kijken en we komen tot het volgende belangrijke begrip,

DEFINITIE 4.1 *De verzameling van alle beslissingsproblemen waarvoor een algoritme met polynomiale looptijd bestaat geven we aan met $\mathcal{P}$.*

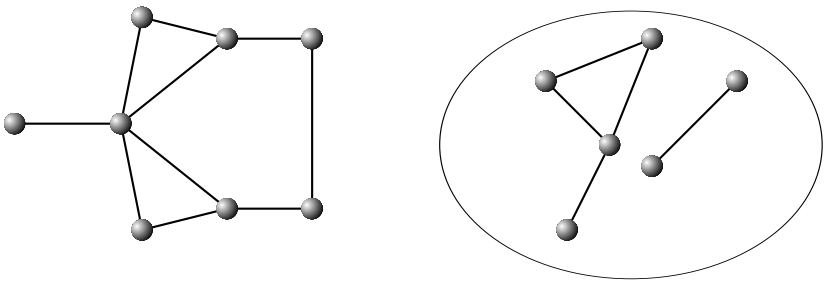
5. Moeilijke problemen

In de vorige paragraaf zagen we dat we het liefst een polynomiaal algoritme voor de oplossing van een probleem willen hebben. Helaas kan dat niet altijd. We hebben immers al gezien dat er onoplosbare problemen zijn. Maar ook binnen de klasse van oplosbare problemen kunnen we voorbeelden geven die geen polynomiale oplossing toelaten. Bekende voorbeelden hiervan zijn het probleem van de *Presburger rekenkunde* (zie [GG,SC]), *Busy beaver problem* en het *Wegblokkade spel* (zie [SC]).
Daarnaast is er ook een groot aantal problemen waarvoor geen polynomiale oplossing bekend is, maar waarvan evenmin aangetoond kan worden dat er geen polynomiaal algoritme voor bestaat. Het bekendste voorbeeld hiervan is het probleem van de ontbinding in factoren. Ondanks de boeiende ontwikkelingen van de afgelopen vijftien jaar op dit gebied is het ontbinden van een doorsnee getal van zo'n 150 cijfers nog steeds een huzarenstukje waar honderden of duizenden computers bij betrokken zijn. Het beste algoritme, de *General Number Theory Sieve*, heeft een looptijd van de orde $O(\exp(\sqrt[3]{\ln(N)\ln\ln(N)}))$.
Om te benadrukken waarom dit probleem zo belangrijk is, het zogenaamde RSA-protocol in de cryptografie is gebaseerd op onze onkunde om grote getallen in factoren te ontbinden. Het is echter wel zo dat voor dit protocol grote toepassingen gezien worden in datacommunicatie, waaronder ook electronische financiële transacties.
Veel beslissingsproblemen waarvoor noch een polynomiaal algoritme, noch het niet-bestaan ervan kan worden aangetoond hebben een interessant kenmerk gemeen. Dit verwoorden we in de volgende definitie.

Definitie 5.1 *De verzameling $\mathcal{NP}$ is precies die verzameling van beslissingsprobleem waarbij de juistheid van een 'ja'-antwoord in polynomiale tijd geverifieerd kan worden.*

Allereerst moeten we opmerken dat $\mathcal{P} \subset \mathcal{NP}$. Het antwoord op ons beslissingsprobleem is immers in polynomiale tijd gegeven. Het interessante aan $\mathcal{NP}$ is dat er ook veel problemen toe behoren waarvan we niet weten of ze in $\mathcal{P}$ bevat zijn.
Neem als voorbeeld het deelsomprobleem. Stel we hebben 100 getallen van drie cijfers. Gevraagd wordt na te gaan of er een deelverzameling bestaat waarvan de som van de elementen gelijk is aan 50000. Ondanks alle wiskundekennis van tegenwoordig is er voor de oplossing van dit probleem weinig beters bekend dan dat we alle mogelijke deelverzamelingen uitproberen. Dit zijn dus $2^{100} \sim 10^{30}$ pogingen die we moeten uitvoeren. Als iemand er na vele dagen of jaren achter komt dat er inderdaad zo'n deelverzameling bestaat, dan is er een heel eenvoudige manier om anderen daarvan te overtuigen. Geef die ander gewoon de getallen uit de bewuste verzameling en laat hem verifiëren dat hun som inderdaad 50000 is. Dit bewijs van de juistheid van het 'ja'-antwoord kan in zeer korte tijd gevoerd worden, maar het daadwerkelijk vinden van dit bewijs is daarentegen een heel ander verhaal. In de definitie van $\mathcal{NP}$ gaat het ons echter alleen om het bestaan van een polynomiaal bewijs, er wordt niets gezegd over de moeite die men nodig had om eraan te komen.

Een ander voorbeeld uit $\mathcal{NP}$ is dat van de ontbinding van getallen als beslissingsprobleem. Het vinden van het antwoord op de vraag of er een deler van n bestaat die kleiner dan m is, is een notoir moeilijk probleem, vooral als m, n uit honderd of meer cijfers bestaan. Als het antwoord 'ja' is, dan bestaat er een heel eenvoudig bewijs voor de juistheid hiervan. We nemen gewoon de bewuste deler d, controleren of $d < m$ en controleren of d inderdaad n deelt.
Een derde voorbeeld uit de klasse $\mathcal{NP}$ is het Hamilton probleem. Dit gaat over grafen. Een graaf is een collectie van punten en lijnsegmenten zodat de uiteinden van elk lijnsegment bestaan uit een punt. Hier zijn wat voorbeelden.



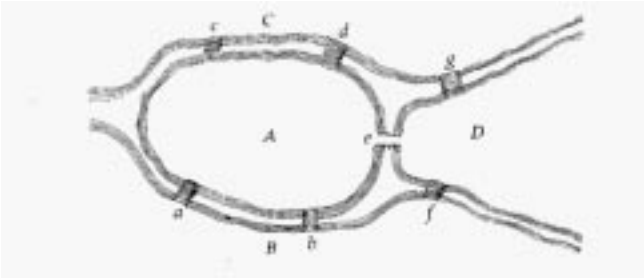
Samenhangend *Niet samenhangend*

Een graaf heet samenhangend als we van elk punt via de segmenten naar elk ander punt kunnen wandelen. Een bekend probleem is de vraag of er een *Eulerpad* in zo'n graaf bestaat. Dat is gesloten pad zó dat elk segment precies éénmaal doorlopen wordt. Het bekendste voorbeeld hiervan is het klassieke Königsberger bruggen probleem (zie pagina 13), waar bij een wandelaar tijdens een rondwandeling elke brug van de zeven bruggen in het plaatje precies éénmaal wil oversteken.
Euler gaf hiervoor de volgende oplossing,

STELLING 5.2 (EULER) *Een samenhangende graaf bevat een Eulerpad precies dan als in elk punt van de graaf een even aantal segmenten bijeenkomen.*

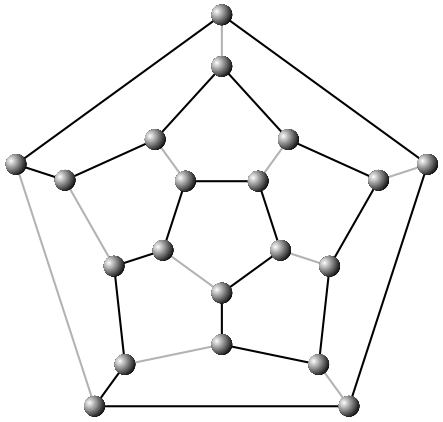
Het is een leuke uitdaging hier een bewijs voor trachten te vinden. In het bijzonder zien we met deze steling dat het bruggenprobleem geen oplossing heeft.
Een variant hierop is het *Hamilton probleem* waarin gevraagd wordt of een graaf een zogenaamd *Hamilton pad* bevat. Dat is een gesloten pad dat elk punt precies éénmaal bezoekt. De naam Hamilton komt van de bekende wiskundige Rowan Hamilton, die ook verantwoordelijk is voor de Hamiltonse mechanica, de geometrische optica en de quaternionen. Hier zien we een voorbeeld van een graaf met een Hamiltonpad.

a. Königsberg in de 18^e *eeuw.*



b. De zeven bruggen van Königsberg

Bron: NORMAN BIGGS et al., *Graph Theory 1736–1936,* Clarendon Press, Oxford, 1976



Hoeveel Hamiltonpaden zijn er in deze graaf?

Wonderlijk genoeg is er géén simpel criterium om te beslissen of een graaf een Hamiltonpad bevat. In een graaf met zo'n honderd punten kan het vinden van een Hamiltonpad een enorme zoektocht zijn. Maar ook hier geldt weer, als iemand mij een graaf geeft dat een Hamiltonpad bevat, dan kan hij mij in korte tijd van dit feit overtuigen door gewoon dat Hamiltonpad aan te geven. Het aardige is dat bij $\mathcal{NP}$-problemen de 'nee'-antwoorden niet in polynomiale tijd bewijsbaar hoeven zijn. Dit zien we het makkelijkst bij het Hamilton probleem. Stel dat iemand mij een graaf voorlegt en beweert dat het geen Hamiltonpad bevat. Hoe zou iemand mij in korte (d.w.z. polynomiale) tijd daarvan kunnen overtuigen? Er is nu niets om ons te laten zien, er immers geen Hamiltonpad. Het is momenteel niet bekend of de afwezigheid van een Hamiltonpad in polynomiale tijd verifieerbaar is. Hoogstwaarschijnlijk niet.
Een aardig detail is dat het Hamiltonpad probleem gezien kan worden als speciaal geval van het handelsreizigerprobleem. De punten van de graaf kunnen gezien worden als steden. De afstand tussem twee van deze steden zetten we op nul als ze verbonden worden door een lijnstruk van de graaf en 1 indien ze niet verbonden worden. Het zal duidelijk zijn dat de graaf een Hamiltonpad bevat precies dan als er de kortste rondweg langs de steden lengte nul heeft.
Hoewel de klasse $\mathcal{NP}$ slechts een deelverzameling van alle "moeilijke problemen" vormt, zijn wel veel voor de praktijk interessante problemen erin vertegenwoordigd. De naam $\mathcal{NP}$ betekent trouwens niet "niet polynomiaal" zoals veel mensen denken. Het staat voor *niet-deterministisch polynomiaal*. Deze benaming heeft betrekking op het niet-deterministisch computermodel. Grof gezegd is dit een computer die kan beschikken over een onbeperkte hoeveelheid naast elkaar werkende processoren. Dit is niet een erg realistisch computermodel, maar voor theoretische beschouwingen heeft het wel z'n waarde. Het verschil met de conventionele computer is dat we nu een instructie tot onze beschikking hebben waarmee we twee nieuwe processoren parallel aan het werk kunnen zetten. We noemen deze instructie BRANCH. Om een idee te geven hoe je zo'n computer zou laten werken geven we hier het voorbeeld van een oplossing van het deelsomprobleem zoals gegeven in [L].
We hebben en rij getallen $X = (x_1, \ldots, x_n)$ en een getal S en we nemen aan dat alle processoren over deze getallen kunnen beschikken. We willen weten of er een deelrij van X bestaat zó dat de som van de elementen S is. We gebruiken hiervoor het programma `testsom`.

`testsom`(index i, tussentijdse deelsom T)

begin

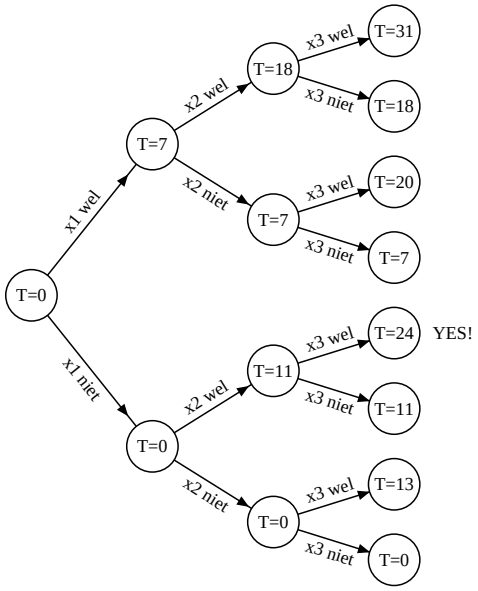
als $i > n$ en T is gelijk aan S roep "YES!"

als $i > n$ STOP

BRANCH(`testsom`$(i+1, T)$, `testsom`$(i+1, T+x_i)$)

end

Vervolgens starten we ons programma op met `testsom`$(1, 0)$. Merk op dat bij elke BRANCH-instructie een splitsing wordt gemaakt aan de hand van de beslissing of we x_i wel of niet in onze deelsom meenemen. Het totale aantal tijdstappen voor dit algoritme bedraagt n en uiteraard is dit lineair in de invoer van onze n getallen. De winst zit hem in de parallelle structuur van onze denkbeeldige machine.
In het volgende plaatje zien we hoe achtereenvolgende processoren aan het werk worden gezet in het geval dat $x_1 = 7, x_2 = 11, x_3 = 13, S = 24$.



Ook voor het ontbindingsprobleem als beslissingsprobleem kan men zich voorstellen dat we een bij elke stap exponentieel toenemend aantal processoren aan het werk zetten die elk een getal als deler proberen.
Het zal de lezer niet ontgaan zijn dat we de klasse $\mathcal{NP}$ hebben ingevoerd als beslissingsproblemen met polynomiale verificatie voor de 'ja'-instanties, maar dat de benaming van deze klasse betrekking heeft op niet-deterministische algoritmen. Dat deze twee zaken desondanks op hetzelfde neerkomen wordt ons verteld door de volgende boeiende stelling.

STELLING 5.3 (COOK,1971) *Zij A een beslissingsprobleem. Dan zijn de volgende beweringen equivalent,*

1. *De 'ja'-instanties van A zij polynomiaal verifieerbaar.*
2. *$A \in \mathcal{NP}$*
3. *A is in polynomiale tijd herleidbaar tot het Hamilton-pad probleem.*

Deze stelling bevat een derde ingredient, namelijk dat elk probleem behorend tot $\mathcal{NP}$ polynomiaal herleidbaar is tot het Hamilton-pad probleem. Een voorbeeld van een polynomiale reductie van één probleem tot een ander zagen

we al bij de formulering van het ontbindingsprobleem als beslissingsprobleem. Daar hadden we enige overhead nodig in de vorm van een binaire zoekactie. Deze overhead draagt echter polynomiaal bij aan de looptijd van het algoritme. Zolang dit het geval blijft spreken van een polynomiale reductie. Het opmerkelijke van de Stelling van Cook is dat elk probleem uit de klasse $\mathcal{NP}$ polynomiaal herleidbaar is tot het Hamilton-pad probleem. Dit betekent in het bijzonder dat als we ooit een polynomiaal algoritme voor het Hamilton-pad probleem zouden vinden, in één klap alle problemen uit $\mathcal{NP}$ polynomiaal oplosbaar zijn, inclusief ontbinding in factoren.
In het oorspronkelijk werk van Cook wordt niet het Hamiltonpad-probleem genoemd maar het *Satisfiability probleem* (zie [L]) voor logische uitdrukkingen. Vlak daarna ontdekte Karp dat het satisfiability probleem polynomiaal kan worden teruggevoerd diverse andere problemen in de klasse $\mathcal{NP}$. Voorbeelden hiervan zijn het Hamiltonpad-probleem en het deelsom probleem. Problemen van deze soort noemen we $\mathcal{NP}$-complete problemen. In het algemeen noemen we een beslissingsprobleem A *$\mathcal{NP}$-compleet* als $A \in \mathcal{NP}$ en als elk $\mathcal{NP}$-probleem polynomiaal tot probleem A kan worden herleid. In de paar jaar volgend op Cook's ontdekking breidde de lijst van $\mathcal{NP}$-complete problemen zich uit tot enkele honderden. Elk van deze problemen heeft dus de eigenschap dat als we een polynomiaal oplossingsalgoritme zouden vinden, elk $\mathcal{NP}$-probleem automatisch een polynomiale oplossing heeft. De omkering van deze bewering geldt uiteraard ook. Met andere woorden,

$$A\ \mathcal{NP}-\text{compleet en } A \in \mathcal{P} \iff \mathcal{NP} = \mathcal{P}$$

Om zich te oriënteren zou de lezer zelf eens moeten proberen een oplossing te vinden voor het deelsomprobleem, een computerprogramma hiervoor schrijven en hiermee problemen met $n = 40$ (zeg) aanpakken. Velen hebben dit al gedaan en zijn hierdoor gesterkt in de mening dat het deelsomprobleem waarschijnlijk geen polynomiaal probleem is. Dit impliceert automatisch het volgende vermoeden,

Vermoeden 5.4

$$\mathcal{NP} \neq \mathcal{P}$$

Het is absoluut niet duidelijk hoe men een dergelijke uitspraak zou moeten bewijzen, ondanks de vele pogingen daartoe. Het hardnekkige voortbestaan van dit vermoeden en het grote belang ervan voor computationele toepassingen hebben ervoor gezorgd dat dit vermoeden een plaats heeft gekregen onder de bekendste problemen in de wiskunde.

6. Literatuur

Bij de voorbereiding van deze tekst heb ik dankbaar gebruik gemaakt van de artikelen en boeken die hieronder vermeld staan. Vooral de algemene en meer populaire artikelen zijn een grote steun geweest bij de voorbereiding. Mijn dank gaat uit naar Jan Karel Lenstra die mij deze artikelen te leen gaf.

Boeken

[BCF] H.W.Broer, J.van de Craats, F.Verhulst, *Het einde der voorspelbaarheid? Chaos: ideeën en toepassingen*, Epsilon Uitgaven, Utrecht 1995

[LLRS] E.L.Lawler, J.K.Lenstra, A.H.G.Rinnooy Kan, D.B.Shmoys, editors *The travelling salesman problem* Wiley and Sons, 1985

[P] C.H.Papadimitriou, *Computational Complexity*, Addison-Wesley 1994

[RS] G.Rozenberg, A.Salomaa, *Cornerstones od undecidability*, Prentice Hall, 1994

Artikelen

[GG] R.L.Graham, M.R.Garey, The Limits to Computation

[L] E.A. Lamagaa, Infeasible Computations, Abacus Vol 4 (1987), p18-33

[LP] H.R.Lewis, C.H.Papadimitriou, The Efficiency of Algorithms, Scientific American 238(1), p96-109

[SC] L.J.Stockmeyer, A.K.Chandra, Intrinsically Difficult Problems, Scientific American, May 1979, p124-133