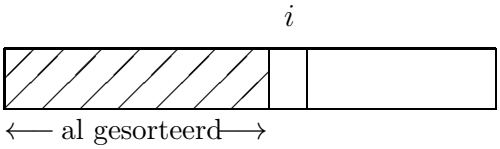
Uitgebreide uitwerking Tentamen Complexiteit, juni 2014

Opgave 1.

a. Bij Insertion sort worden de array-elementen $A[2]$ t/m $A[n]$ een voor een op de goede plaats gezet door herhaald te vergelijken met het element dat er direct links van staat en ze te verwisselen indien ze verkeerd om staan. In dit geval moet het array oplopend gesorteerd worden.
Dus: itereer over i en voeg $A[i]$ op de juiste plek in het gesorteerde stuk $A[1] \cdots A[i-1]$ in door herhaald met de linkerbuur te vergelijken ($A[i-1] <= A[i]$?) en te verwisselen indien nodig (dus als $A[i-1] > A[i]$).



Voor rijtjes van het hier bekeken soort is het aantal vergelijkingen dat Insertion sort *maximaal* doet:
- om $A[2]$ goed te zetten: 1
- om $A[3]$ goed te zetten: 1; immers $A[3]$ is groter dan $A[1]$ en $A[2]$, dus Insertion sort stopt zeker nadat $A[3]$ met $A[2]$ vergeleken is
- om $A[4]$ goed te zetten: 2; immers $A[4]$ is groter dan $A[1]$ en $A[2]$, dus Insertion sort stopt zeker nadat $A[4]$ met $A[2]$ vergeleken is
- om $A[5]$ goed te zetten: 1; immers $A[5]$ is groter dan $A[3]$ en $A[4]$ en alles wat daarvoor staat, dus Insertion sort stopt zeker nadat $A[5]$ met $A[4]$ vergeleken is
- om $A[6]$ goed te zetten: 2; immers $A[6]$ is groter dan $A[3]$ en $A[4]$, dus Insertion sort stopt zeker nadat $A[6]$ met $A[4]$ vergeleken is

In het algemeen:
- om $A[i]$ met i oneven goed te zetten: 1; immers $A[i]$ is groter dan $A[i-1]$ en $A[i-2]$ en alles wat daarvoor staat, dus Insertion sort stopt zeker nadat $A[i]$ met $A[i-1]$ vergeleken is
- om $A[i]$ met i even goed te zetten: 2; immers $A[i]$ is groter dan $A[i-2]$ en $A[i-3]$, dus Insertion sort stopt zeker nadat $A[i]$ met $A[i-2]$ vergeleken is

Dat levert in totaal dus op dat Insertion sort op dit soort rijtjes maximaal $1 + 3 \cdot \frac{n-2}{2} = \frac{3}{2}n - 2$ vergelijkingen doet. Immers: voor het eerste tweetal 1 vergelijking, en voor de overige $\frac{n-2}{2}$ tweetallen maximaal 3. Dit worst case geval komt overigens voor als $A[i] > A[i+1]$ voor alle oneven $i > 1$.

b. Elk sorteeralgoritme voor dit soort rijtjes moet de juiste (= oplopende) volgorde kunnen vinden. Het aantal mogelijke gesorteerde volgordes is in dit geval beperkt, omdat we al weten dat van elk deelrijtje $A[i], A[i+1]$ met i oneven beide elementen groter zijn dan alles wat ervoor staat. Er zijn dus veel minder mogelijkheden/mogelijke antwoorden dan in het algemene geval[1]. Om precies te zijn: 2 per tweetal, immers $A[i] < A[i+1]$ kan voorkomen, of $A[i] > A[i+1]$. De onderlinge relatie met de overige array-elementen ligt

[1]Bijvoorbeeld met $n = 4$: $A[2] < A[1] < A[3] < A[4]$ kan wel voorkomen, maar $A[3] < A[1] < A[4] < A[2]$ niet

al vast. Dus: het aantal mogelijke sorteringen is hier $2 \cdot 2 \cdot \cdots \cdot 2 = 2^{\frac{n}{2}}$ omdat we $\frac{n}{2}$ paren hebben en voor elk paar 2 mogelijkheden.
Deze antwoorden/sorteringen kunnen allemaal voorkomen onder de gegeven soort invoerrijtjes en moeten dan ook gevonden kunnen worden door elk algoritme voor dit probleem. Derhalve moet elke beslissingsboom (corresponderend met een algoritme voor het probleem) *minstens* zoveel bladeren hebben: $b = \#\text{bladeren} \geq 2^{\frac{n}{2}}$. Dit geldt dus voor elke beslissingsboom corresponderend met een algoritme voor dit probleem. Uit $h \geq \lceil \lg b \rceil$ (eigenschap van binaire bomen) volgt dan dat voor zulke beslissingsbomen $h \geq \lceil \lg(2^{\frac{n}{2}}) \rceil = \lceil \frac{n}{2} \rceil = \frac{n}{2}$ (want n is even. Ergo, het aantal vergelijkingen in de worst case (dat is namelijk de hoogte van de bijbehorende beslissingsboom) voor *elk* algoritme voor het onderhavige probleem is $\frac{n}{2}$ (immers de afschatting gold voor *elke* beslissingsboom bij dit probleem).

c. Deze ondergrens is niet in strijd met de op college gevonden ondergrens van $\lceil n \lg n \rceil$. Immers deze ondergrens geldt voor het algemene geval, dus waarbij de invoer niet aan restricties is gebonden. In deze opgave gaat het over hele speciale rijtjes, en dan zou het mogelijk beter kunnen dan $\lceil n \lg n \rceil$, namelijk door expliciet gebruik te maken van de kennis die je over het rijtje hebt.
Het wordt niet gevraagd, maar de ondergrens is optimaal, want er is een eenvoudig algoritme te geven dat rijtjes van het bekeken type ook inderdaad met $\frac{n}{2}$ arrayvergelijkingen sorteert. Namelijk: gewoon twee aan twee de array-elementen vergelijken en in de juiste volgorde zetten.

d. Gezien bij college: een inversie van $A[1], A[2], \ldots, A[n]$ is een paar $(A[i], A[j])$ waarvoor $i < j$ en $A[i] > A[j]$. M.a.w.: een inversie is een paar $(A[i], A[j])$ dat verkeerd om staat. Merk op dat een oplopend gesorteerd rijtje nul inversies heeft. *Elk* sorteeralgoritme moet dus *alle* aanwezige inversies opheffen. Voor het soort rijtjes uit de opgave geldt dat er maar een beperkt aantal inversies kan voorkomen. Alleen de $\frac{n}{2}$ paren $(A[i], A[i+1])$ met i oneven kunnen verkeerd om staan, de rest staat al goed ten opzichte van elkaar. Het maximale aantal inversies dat in onze rijtjes kan voorkomen is dus $\frac{n}{2}$. Dit komt voor als alle genoemde $\frac{n}{2}$ paren verkeerd om staan.
Voor een sorteeralgoritme (gebaseerd op het doen van arrayvergelijkingen) dat altijd hooguit één inversie opheft per arrayvergelijking geldt dat het aantal vergelijkingen dat wordt gedaan om $A[1], \ldots, A[n]$ te sorteren *ten minste* gelijk is aan het aantal inversies van A **(1 punt)**. Een en ander gecombineerd betekent dat zo'n sorteeralgoritme voor het bekeken soort rijtjes in het ergste geval ten minste $\frac{n}{2}$ arrayvergelijkingen doet.

Opgave 2.
a. $P(n)$ = aantal optellingen van array-elementen (reële getallen) nodig om twee $n \times n$ matrices te vermenigvuldigen volgens de aangegeven methode (dus recursief).
Het probleem is teruggebracht tot het 8 keer vermenigvuldigen van twee $\frac{n}{2} \times \frac{n}{2}$ matrices. Dat kost dus samen $8P(\frac{n}{2})$ optellingen. Verder moeten dan nog 4 keer twee $\frac{n}{2} \times \frac{n}{2}$ matrices worden opgeteld. Aangezien een $\frac{n}{2} \times \frac{n}{2}$ matrix $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$ elementen heeft kost het optellen van twee van zulke matrices (dat gebeurt elementsgewijs) precies $\frac{n^2}{4}$ optellingen. Als dat vier keer gebeurt kost dat dus nog eens $4 \cdot \frac{n^2}{4} = n^2$ optellingen. Samen: $8P(\frac{n}{2}) + n^2$ optellingen. Ten slotte: als $n = 1$ reduceert het vermenigvuldigen van de 1×1 matrices

$A = a_1$ en $B = b_1$ tot het berekenen van $a_1 \cdot b_1$, hetgeen 0 optellingen van array-elementen kost.
b. Eerst proberen we de oplossing te vinden door $P(n)$ herhaald in zichzelf in te vullen.

$$P(n) = 8P(\frac{n}{2})+n^2 = 8\{8P(\frac{n}{2^2})+(\frac{n}{2})^2\}+n^2 = 8^2P(\frac{n}{2^2})+8\cdot\frac{n^2}{4}+n^2 = 8^2P(\frac{n}{2^2})+2n^2+n^2 =$$

$$8^2\{8P(\frac{n}{2^3})+(\frac{n}{2^2})^2\}+2n^2+n^2 = 8^3P(\frac{n}{2^3})+8^2\cdot\frac{n^2}{2^4}+2n^2+n^2 = 8^3P(\frac{n}{2^3})+8^2\cdot\frac{n^2}{16}+2n^2+n^2 =$$

$$8^3P(\frac{n}{2^3}) + 4n^2 + 2n^2 + n^2 = 8^3P(\frac{n}{2^3}) + n^2 \cdot (2^2 + 2 + 1)$$

Algemene vorm dus vermoedelijk:

$$P(n) = 8^\ell \cdot P(\frac{n}{2^\ell}) + n^2 \cdot (2^{\ell-1} + \cdots + 2^2 + 2 + 1) = 8^\ell \cdot P(\frac{n}{2^\ell}) + n^2 \cdot (2^\ell - 1)$$

In de laatste stap is de sommatie uit de hint gebruikt. Kies nu $\ell = k$, zodat we de beginconditie $P(1) = 0$ kunnen gebruiken. Immers als $\ell = k$ geldt, omdat $n = 2^k$, dat $\frac{n}{2^\ell} = \frac{2^k}{2^k} = 1$. En dan vinden we:

$$P(n) = 8^k \cdot P(1) + n^2 \cdot (2^k - 1) = n^2(n-1)$$

Nu nog bewijzen dat de gevonden $P(n) = n^2(n-1)$ inderdaad de oplossing is van de recurrente betrekking. Dit gaat m.b.v. volledige inductie.

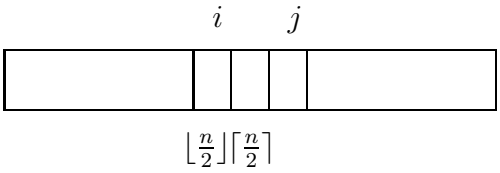
(i) $P(n) = n^2(n-1)$ voldoet inderdaad aan $P(1) = 1^2(1-1) = 0$ zien we door invullen.
(ii) Inductie-aanname: stel dat $P(n)$, de oplossing van de recurrente betrekking, gelijk is aan $n^2(n-1)$ voor alle 2-machten $n \geq 1$ met $n < N$ en met N een 2-macht. Dan moeten we laten zien dat dit ook geldt voor de oplossing van de recurrente betrekking voor deze N (de volgende 2-macht dus) ofwel we moeten aantonen dat $P(N) = N^2(N-1)$.

Er geldt:
$P(N) =$ (recurrente betrekking) $8P(\frac{N}{2})+N^2 =$ (inductie-aanname) $8\cdot(\frac{N}{2})^2(\frac{N}{2}-1)+N^2 = 8\cdot\frac{N^2}{4}(\frac{N}{2}-1)+N^2 = 2N^2(\frac{N}{2}-1)+N^2 = N^3-2N^2+N^2 = N^3-N^2 = N^2(N-1)$ QED

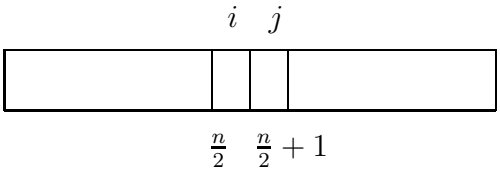
Opmerking. Je kunt het inductiebewijs geven door te bewijzen dat: als het voor N geldt, dan ook voor $2N$. En dan $P(2N) =$ (recurrente betrekking) $8P(N) + (2N)^2 = 8N^2(N-1) + 4N^2 = (2N)^3 - 4N^2 = (2N)^3 - (2N)^2$. QED

Opgave 3.
a. Als $A = x, x, x, \ldots, x, x$ dan stoppen de twee binnenste while-loops altijd meteen op de test $A[j] > x$ respectievelijk $A[i] < x$. Dit levert telkens een verwisseling op (mits $i < j$). Dus in elke ronde (doorgang buitenste lus) schuift j een plek naar links en i een plek naar rechts totdat ze elkaar tegenkomen met $i = j$ (dat gebeurt als n oneven is) op het middelste array-element, of (n even) totdat ze elkaar passeren en j stopt op $\frac{n}{2}$ en i op $\frac{n}{2}+1$. Dus q wordt gelijk aan het midden $\lceil\frac{n}{2}\rceil$ als n oneven, en $\frac{n}{2}$ voor n even. Samen: $q = \lceil\frac{n}{2}\rceil$.

Hieronder de situatie bij de laatste keer dat i en j gestopt zijn met $i < j$ en $A[i]$ en $A[j]$ verwisseld worden.

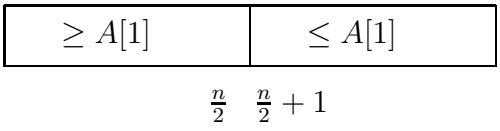


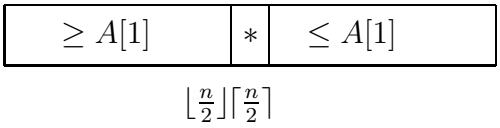
n oneven



n even

Als $A = 1, 2, 3, \ldots, n-1, n$ zal j in de eerste ronde meteen helemaal naar links doorlopen omdat alle $A[j]$ met $j > 1$ groter zijn dan $x = A[1]$. Vervolgens stopt i op positie 1. Omdat nu $i = j$ vindt geen verwisseling plaats, stopt het algoritme meteen en is $q = 1$.
b. Kijk om het aantal verwisselingen in het ergste geval en de bijbehorende worst case rijtjes te vinden goed naar het algoritme. Dan zie je dat ... het aantal verwisselingen maximaal is als i en j zo langzaam mogelijk naar elkaar toelopen; immers elke keer dat i en j stoppen vindt een verwisseling plaats; i en j moeten dus zo vaak mogelijk stoppen. Dit vindt plaats als i en j elke keer meteen nadat ze een positie zijn opgeschoven stoppen, en dit komt voor als telkens $A[j] \leq x = A[1]$ en $A[i] \geq x = A[1]$ totdat i en j elkaar tegenkomen ongeveer in het midden. In dat geval is het aantal verwisselingen $\lfloor \frac{n}{2} \rfloor$ zoals in **a.** al berekend. Merk op dat voor even n moet gelden (wil A een worst case rijtje zijn) dat $A[i] \geq A[1]$ voor $i = 2, \ldots, \frac{n}{2}$ en $A[j] \leq A[1]$ voor $j = \frac{n}{2} + 1, \ldots, n$ en dan $q = \frac{n}{2}$. Voor n oneven geldt iets soortgelijks, met dien verstande dat het middelste array-element alles mag zijn. Als $A[\lceil \frac{n}{2} \rceil] > A[1]$ loopt j nog 1 positie extra door naar links en wordt $q = \lfloor \frac{n}{2} \rfloor$, als $A[\lceil \frac{n}{2} \rceil] \leq A[1]$ wordt $q = \lceil \frac{n}{2} \rceil$.
Zie hieronder een illustratie van de algemene karakterisering voor n even resp. n oneven.





$*$: don't care

Voorbeeldrijtjes: $x, x, x, \ldots, x, x$ en $\frac{n}{2} + 1, \ldots, n-1, n, 1, 2, \ldots, \frac{n}{2}$ (even n).
c. We kijken weer naar het algoritme om een goede karakterisering te vinden van worst case rijtjes. Merk op dat de while-loop gewoon een for-loop is (j loopt van 1 tot en met $n-1$). Er worden zo veel mogelijk verwisselingen gedaan als voor elke j de test of

$A[j] \leq x = A[n]$ True oplevert. In totaal doe je in het ergte geval dus $n - 1 + 1$ (een na laatste regel) verwisselingen, en dat komt voor dan en slechts dan (zie boven) als $A[j] \leq A[n]$ voor $j = 1, 2, \ldots, n - 1$. De uiteindelijke q is de laatste waarde van $i + 1$, en aangezien i gelijk oploopt met j in de worst case wordt dit gelijk aan n. Merk op dat hier telkens een array-element met zichzelf verwisseld is, dus eigenlijk zijn het zinloze verwisselingen.
Voorbeeldrijtjes: $x, x, x, \ldots, x, x$ en $1, 2, \ldots, n - 1, n$.
d. Binnen Quicksort is de splitsing door `Partitie` van groot belang. Gunstig is als q ongeveer halverwege het array zit; je splitst dan in twee ongeveer gelijke helften. Ongunstig is het als q ver naar links of ver naar rechts zit; je splitst dan in een heel klein en een heel groot stuk. (Zie ook college.) Als we naar de uiteindelijke q kijken zien we dat in de worst case (wat betreft het aantal verwisselingen) PartitieH heel gunstig splits, en bovendien de helft minder verwisselingen doet dan PartitieL. Deze laatste splitst in het ergste geval juist heel slecht.
Merk nog op dat voor willekeurige grote rijtjes het bij Quicksort op een gegeven moment vaak zal voorkomen dat je rijtjes gelijke getallen zal moeten partitioneren/sorteren. In dat geval doet PartitieH het dus duidelijk beter dan PartitieL. Voor het oplopende rijtje leveren beide algoritmen een slechte splitsing, maar PartitieH doet dan geen enkele verwisseling, tegenover PartitieL n verwisselingen.

Opgave 4.
a. We geven een *niet-deterministisch polynomiaal* algoritme A voor half3SAT. A heeft als invoer een logische formule ϕ in 3-CNF met een even aantal clausules.
Idee voor het certificaat: een waardering van de in ϕ voorkomende logische variabelen die precies de helft van de clausules True maakt en de andere helft False.
A doet bijvoorbeeld het volgende:

1. Fase 1 (gokfase). Er wordt een willekeurige string s gegenereerd. (Deze s stelt hopelijk een goede waardering voor; dit wordt in Fase 2 gecontroleerd.)

2. Fase 2 (controlefase). Hier wordt gecontroleerd of s inderdaad een waardering van de in ϕ voorkomende logische variabelen voorstelt die de helft van de clausules van ϕ waarmaakt (en de andere helft False).
- tel het aantal verschillende logische variabelen in ϕ (noem dit n). Dat kan in $O(|\phi|^2)$ stappen.
- controleer dat s bestaat uit booleans, d.w.z. elke s_i is T of F. Dat kan eenvoudig in $O(|s|)$ stappen.
- Controleer dat s precies n boolese waarden bevat: $(O(|s|))$.

Als aan deze twee controles voldaan is, kunnen we s dus interpreteren als een waardering van de in ϕ voorkomende logische variabelen. Zo stelt s_i dan de waarheidswaarde van variabele x_i voor.
De eerste twee controles zijn een soort syntactische controle. In de volgende test wordt gekeken of s een *goede* waardering voorstelt.

- controleer dat de waardering uit s precies de helft van de clausules van ϕ True maakt

en de andere helft False.
Loop v.l.n.r. door ϕ en haal de waarheidswaarde van de 3 logische variabelen in de aan de beurt zijnde clausule op uit s (dat is $O(|s|)$) en controleer of een der variabelen daaruit True is. Als dat het geval is dan hogen we een tellertje op die het aantal clausules dat True wordt gemaakt. Als dat niet het geval is hogen we een ander tellertje op, dat aangeeft hoeveel clausules False worden gemaakt. Als we ϕ helemaal door hebben gelopen testen we of beide tellertjes aan elkaar gelijk zijn. Zo ja, dan levert deze controle True op. Dit is in totaal $O(|\phi||s|)$.

Als alle drie controles positief zijn stelt s een goede waardering voor ϕ voor en wordt True geretourneerd. Zodra een van de controles niet klopt wordt False geretourneerd of raak je in een oneindige lus. Er geldt dus: Fase 2 geeft True $\iff$ s is een goede waardering.

3. Fase 3 (uitvoerfase). Als Fase 2 True oplevert, dan wordt "ja" uitgevoerd, anders geen uitvoer.

Nu geldt: het antwoord van A op invoer ϕ is "ja" $\iff$ er is een executie van A die "ja" oplevert $\iff$ er is een string s waarop A in Fase 2 True geeft $\iff$ er is een string s die een goede waardering van ϕ voorstelt $\iff$ ϕ heeft een waardering die precies de helft van de clausules True maakt en de andere helft False $\iff$ ϕ is een ja-instantie van half3SAT. Kortom, A is inderdaad een (niet-deterministisch) algoritme voor half3SAT. Bovendien is het polynomiaal. Immers voor ja-instanties (invoer ϕ waarop A "ja" geeft) is er een ja-executie waarin s een goede waardering, dus i.h.b. een waardering, voorstelt. In zo'n geval is dus $|s| = O(|\phi|)$. De executie met die s kan dus zeker wel in: $O(|s|)$ (Fase 1) + $O(|\phi|^2)$ + $O(|s|)$ + $O(|s|)$ + $O(|\phi||s|)$ (Fase 2) + $O(1)$ (Fase 3) = $O(|\phi|)$ + $O(|\phi|^2)$ = $O(|\phi|^2)$ stappen, en dat is polynomiaal als functie van de invoer ϕ van A.
Conclusie: A is een niet-deterministisch polynomiaal ($O(|\phi|^2)$) algoritme voor half3SAT.

b. $T(\phi) = \phi \wedge (u \vee \neg u \vee v) \wedge \cdots \wedge (u \vee \neg u \vee v) \wedge (u \vee v \vee w) \wedge \cdots \wedge (u \vee v \vee w)$, waarvan $2m$ stuks $(u \vee v \vee w)$ en m stuks $(u \vee \neg u \vee v)$, met m het aantal clausules van ϕ.
Merk op dat de clausules $(u \vee \neg u \vee v)$ altijd waar zijn, wat de waarde van u en v ook is.
Nu het gevraagde bewijs.
$\Longrightarrow$ ϕ is een ja-instantie van 3SAT, dus er bestaat een waardering ω van de logische variabelen x_i uit ϕ die ϕ waarmaken. We breiden ω uit tot een waardering W van de logische variablen uit $T(\phi)$. We nemen $W(x_i) = \omega(x_i)$ en $W(u) = W(v) = W(w) =$ False. Deze waardering maakt ϕ waar en de $2m$ clausules $(u \vee v \vee w)$ False, dus de helft van de clausules ($2m$ stuks) van $T(\phi)$ wordt waargemaakt, en de andere helft (ook $2m$ stuks) is False. QED
$\Longleftarrow$ $T(\phi)$ is een ja-instantie van half3SAT, dus er bestaat een waardering W van de logische variabelen x_i, u, v, w uit $T(\phi)$ die precies $2m$ clausules van $T(\phi)$ waarmaakt en de $2m$ andere False. Merk op dat die waardering nooit $(u \vee v \vee w)$ kan waarmaken, want dan zouden in totaal zeker al $3m$ clausules waar zijn. Dus, die waardering W moet alle $2m$ clausules $(u \vee v \vee w)$ False maken. De m clausules $(u \vee \neg u \vee v)$ zijn altijd True. Er moet verder dus gelden dat W ook alle m clausules van ϕ waarmaakt. Dus W (beperkt tot de x_i) is een waarmakende waardering voor ϕ, m.a.w. er bestaat een waarmakende waardering voor ϕ. QED

c. P is NP-hard als $Q \leq_P P$ voor alle Q uit $\mathcal{NP}$.
P is NP-volledig als (i) $P \in \mathcal{NP}$ en (ii) P is NP-hard.

d. (i) Omdat half3SAT $\in \mathcal{NP}$ en 3SAT NP-hard is (want NP-volledig) weten we zeker dat er een polynomiale reductie van half3SAT naar 3SAT bestaat, dus half3SAT $\leq_P$ 3SAT. Dit volgt meteen uit de definitie van NP-hard.
(ii) Als 3SAT $\leq_P$ half3SAT weten we, samen met het feit dat 3SAT NP-hard is, dat voor elk NP-probleem Q geldt: Q $\leq_P$ 3SAT $\leq_P$ half3SAT. Vanwege de transitiviteit van $\leq_P$ volgt daaruit: Q $\leq_P$ half3SAT voor alle NP-problemen Q. Dus half3SAT is NP-hard. Samen met **a.** volgt hier dan uit dat half3SAT $\in \mathcal{NPC}$.
(Intuitief geredeneerd: SAT $\leq_P$ half3SAT betekent dat half3SAT minstens zo moeilijk is als het heel moeilijke (namelijk $\mathcal{NPC}$) probleem 3SAT. Dus half3SAT is ook heel moeilijk of nog erger. Samen met half3SAT $\in \mathcal{NP}$ volgt daar dan uit dat half3SAT in $\mathcal{NPC}$ zit.)

Opgave 5. Een mogelijk DTM-programma:

$$\Gamma = \{0, 1, b\}, \Sigma = \{0, 1\}$$

$$Q = \{q_0, q_1, q_2, q_Y, q_N\}$$

q	0	1	b
q_0	$(q_1, 0, +1)$	$(q_N, 1, 0)$	$(q_N, b, 0)$
q_1	$(q_2, 0, +1)$	$(q_1, 1, +1)$	$(q_N, b, 0)$
q_2	$(q_N, 0, 0)$	$(q_N, 1, 0)$	$(q_Y, b, 0)$

$\delta(q, s)$

Idee van de werking: We moeten 01^n0 herkennen, dus je moet eerst precies één 0 lezen (toestand q_0). Vervolgens verwachten we nul of meer 1-en (toestand q_1) en daarna moet weer precies één 0 (toestand q_2) komen. Eenvoudigweg naar rechts lopen en onderweg van toestand veranderen. Zodra je iets verkeerds leest stoppen in de nee-toestand, bijvoorbeeld als x niet met 0 begint. We hoeven niet terug te lopen om op het eerste karakter te eindigen. De string wordt niet veranderd; telkens zet je hetzelfde karakter terug dat je gelezen hebt.

Betekenis der toestanden:

- q_0: het eerste karakter moet een 0 zijn, anders stoppen in q_N.
- q_1: wordt gebruikt om nul of meer 1-en te lezen. Zolang je 1-en leest doorlopen. Als je een 0 leest moet dat de achterste nul zijn. Een blanco lezen kan niet: stoppen in q_N.
- q_2: we hebben net een 0 gelezen na een aantal enen. de string x moet nu klaar zijn, dus als we iets anders dan blanco lezen is dit fout: stop in toestand q_N.