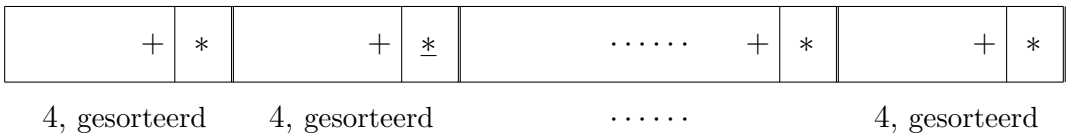
Uitgebreide uitwerking Tentamen Complexiteit, juni 2018

Opgave 1.

a. Een pad van de wortel naar een blad stelt de serie achtereenvolgende arrayvergelijkingen voor die het algoritme doet op zekere invoer. De lengte van zo'n pad correspondeert precies met het aantal arrayvergelijkingen dat op die invoer wordt gedaan. In het bijbehorende blad is het antwoord van het algoritme voor die invoer bekend: het algoritme stopt. Bladeren kunnen derhalve geassocieerd worden met de eindantwoorden/uitkomsten die het algoritme vindt. De hoogte van de beslissingsboom (lengte van het langste pad) stelt dan ook het aantal arrayvergelijkingen in de worst case voor.

b. Vanwege de speciale vorm van de invoerarrays, moet de grootste waarde altijd op een van de posities $4, 8, 12, \cdots, n$ staan: dat zijn $\frac{n}{4}$ mogelijkheden. Kandidaatposities voor de grootste waarde zijn in onderstaand plaatje aangegeven met $*$.



De op twee na grootste waarde kan dan nog op $\frac{n}{2}$ posities staan. Immers, stel dat de grootste op positie $\underline{*}$ staat, dan kan de op twee na grootste waarde een van de $\frac{n}{4} - 1$ elementen op de overige posities $*$ zijn, maar hij kan zich ook op een positie aangegeven met $+$ bevinden, namelijk als de op een na grootste op plek $*$ staat, en de op twee na grootste op plek $+$ in hetzelfde rijtje. Als bijvoorbeeld de een na grootste waarde op positie 4 staat, dan kan het element op positie 3 de op twee na grootste van de hele rij zijn. Verder kan ook het element dat in hetzelfde deelrijtje op de tweede positie daarvan staat als op twee na grootste waarde voorkomen, namelijk als de drie grootste waardes allemaal in hetzelfde deelrijtje staan. In totaal levert zo elk deelrijtje twee kandidaatposities op voor de op twee na grootste, gegeven de positie van de grootste waarde. Dat betekent dus in totaal: $\frac{n}{4} \cdot 2 \cdot \frac{n}{4} = \frac{n^2}{8}$ mogelijke antwoorden van de vorm (p, q) met p de positie van de grootste en q de positie van de op twee na grootste. Als voorbeeld: een mogelijk antwoord kan $(8, 4)$ zijn of $(12, 11)$ of $(16, 14)$, maar nooit $(16, 10)$. Immers $A[16]$ is in dit laatste geval zeker groter dan $A[10]$ (want $A[16]$ was de grootste van het hele array) en $A[12]$ en $A[11]$ zijn groter dan $A[10]$ vanwege de speciale vorm van de invoer. Er zijn dus in totaal zeker al $1 + 2 = 3$ elementen groter dan $A[10]$.

c. Eerst twee belangrijke opmerkingen/stellingen over beslissingsbomen.

- Elk mogelijk antwoord moet als blad kunnen voorkomen omdat het algoritme voor elke mogelijke invoer van het probleem moet werken. Hieruit volgt onmiddellijk dat elke beslissingsboom (corresponderend met een algoritme voor het betreffende probleem) *minstens* zoveel bladeren heeft als er antwoorden mogelijk zijn, dus $b = \#\text{bladeren} \geq$ aantal mogelijke antwoorden.
- Verder hebben we een stelling voor binaire bomen die het verband aangeeft tussen de hoogte en het aantal bladeren b, namelijk $h \geq \lceil \lg b \rceil$.

We moeten dus voor ons probleem het aantal mogelijke antwoorden, in dit geval de index van de grootste waarde en de index van de op twee na grootste waarde (p, q) voor arrays van de beschreven vorm, weten. In **b** zagen we dat het aantal antwoorden gelijk is aan $\frac{n^2}{8}$.

Dit betekent dat $b \geq \frac{n^2}{8}$. Een en ander achter elkaar geschakeld: aantal arrayvergelijkingen in de worst case voor elk algoritme voor het onderhavige probleem $= h \geq \lceil \lg b \rceil \geq \lceil \lg \frac{n^2}{8} \rceil = \lceil \lg n^2 - \lg 8 \rceil = \lceil 2 \lg n - 3 \rceil = \lceil 2 \lg n \rceil - 3$.
d. (i) In tegenspraak met **c**. Immers, $O(\sqrt{\lg n})$ is echt kleiner dan $\lceil 2 \lg n \rceil - 3$ vanaf zekere n. En dat is in tegenspraak met het feit dat het aantal arrayvergelijkingen in het ergste geval altijd, dus voor alle n, minstens $\lceil 2 \lg n \rceil - 3$ moet zijn.
(ii) Het is mogelijk dat er een algoritme voor ons probleem bestaat dat in de worst case $3n - 6$ arrayvergelijkingen doet[1]; $3n - 6$ is immers groter (voor alle $n \geq 8$) dan de ondergrens uit **d**. Dit zegt echter niets over het al dan niet optimaal zijn van dit algoritme. Die ondergrens hoeft namelijk helemaal niet scherp te zijn. Er hoeft dus geen algoritme te bestaan dat $\lceil 2 \lg n \rceil - 3$ vergelijkingen doet. Het zou best kunnen dat algoritme $\mathcal{B}$ toch optimaal is, en dat ons bewijs een te zwakke ondergrens heeft opgeleverd. Het kan ook zijn dat $\mathcal{B}$ inderdaad niet optimaal blijkt, maar daarvoor zouden we bijvoorbeeld een sneller algoritme moeten geven, of op een of andere manier bewijzen dat $3n - 6$ een ondergrens is voor het aantal arrayvergelijkingen. Hoe dan ook, het is mogelijk dat **c** optimaal is, dus de conclusie uit (ii) is onjuist en daarmee is de bewering in totaal niet waar.

Opgave 2.
a. We mogen wel aannemen dat we A oplopend willen sorteren. (Aflopend sorteren gaat precies hetzelfde.) Mergesort hakt het array eerst in twee gelijke (want n is hier een 2-macht) delen van elk $\frac{n}{2}$ elementen. Vervolgens roept het zichtzelf aan op elk van beide helften. Dat levert de term $2M(\frac{n}{2})$ op. Na afloop van die twee aanroepen is elke helft oplopend gesorteerd. We gebruiken vervolgens Merge om de twee gesorteerde delen samen te voegen tot één gesorteerd stuk (ritsen). Dit werkt door herhaald de twee voorste elementen van beide deelrijtjes te vergelijken en de kleinste van die twee in een hulparray B te plaatsten (van klein naar groot). Na elke vergelijking wordt dus precies één element doorgeschoven naar B. Als een van beide deelrijtjes leeg is wordt het andere resterende deelrijtje gewoon achter de rest in B gekopieerd, zonder nog vergelijkingen te doen. (Na afloop wordt B weer teruggekopieerd naar A; dat is nul arrayvergelijkingen.) Dit mergen van twee gesorteerde rijtjes ter lengte $\frac{n}{2}$ kost in het slechtste geval $n - 1$ arrayvergelijkingen. Immers, we hebben worst case als de rijtjes zo lang mogelijk niet leeg raken, dus als op het laatst, wanneer er $n - 2$ elementen zijn doorgeschoven naar B en er dus evenveel arrayvergelijkingen zijn geweest, beide rijtjes nog precies één element bevatten. (Dit moeten dus de grootste en de op-eeen-na-grootste waarde uit A zijn.) Daarna is nog precies 1 vergelijking nodig, samen dus $n - 1$ arrayvergelijkingen in de worst case. Ten slotte de beginconditie: als $n = 1$ is A al gesorteerd, dus dan 0 arrayvergelijkingen.
b. Herhaald invullen[2]:
$M(n) = 2M(\frac{n}{2}) + n - 1 = 2(2M(\frac{n}{2^2}) + \frac{n}{2} - 1) + n - 1 = 2^2 M(\frac{n}{2^2}) + n - 2 + n - 1 = 2^2(2M(\frac{n}{2^3}) + \frac{n}{2^2} - 1) + n - 2 + n - 1 = 2^3 M(\frac{n}{2^3}) + n - 2^2 + n - 2 + n - 1) = 2^3 M(\frac{n}{2^3}) + 3n - (1 + 2 + 2^2) = \cdots =$ (vermoedelijke algemene vorm; klopt met de gevallen $\ell = 1, 2, 3$) $= 2^\ell M(\frac{n}{2^\ell}) + \ell \cdot n - (1 + 2 + 2^2 + \cdots + 2^{l-1}) = 2^\ell M(\frac{n}{2^\ell}) + \ell \cdot n - (2^\ell - 1)$. Kies nu $\ell = k = \lg n$, zodat we de beginwaarde $M(1) = 0$ kunnen gebruiken. Dan vinden we: $M(n) = 2^k M(\frac{n}{2^k}) + k \cdot n - (2^k - 1) = 0 + k \cdot n - (2^k - 1) = n \lg n - n + 1$.
Nu nog met volledige inductie bewijzen dat de gevonden $M(n) = n \lg n - n + 1$, inderdaad de oplossing van de recurrente betrekking is voor *alle* n met n een tweemacht.

[1] Het is ook niet moeilijk zo'n algoritme te bedenken
[2] Merk op: $M(n) = 2M(\frac{n}{2}) + n - 1$, dus $M(\frac{n}{2}) = 2M(\frac{n}{2^2}) + \frac{n}{2} - 1$, en evenzo $M(\frac{n}{2^2}) = 2M(\frac{n}{2^3}) + \frac{n}{2^2} - 1$

(i) $M(n) = n \lg n - n + 1$ voldoet inderdaad aan $M(1) = 1 \cdot \lg 1 - 1 + 1 = 0$, zien we door invullen.
(ii) Inductie-aanname: stel dat $M(n)$, de oplossing van de recurrente betrekking, gelijk is aan $M(n) = n \lg n - n + 1$ voor alle tweemachten $n < N$ en met N ook een tweemacht. Dan moeten we laten zien dat de oplossing van de recurrente betrekking voor deze N (de volgende tweemacht dus) ook gelijk is aan $M(N)$, dus gelijk is aan $M(N) = N \lg N - N + 1$.
Er geldt:
$M(N) =$ (recurrente betrekking) $2M(\frac{N}{2}) + N - 1 =$ (inductie-aanname: de formule geldt voor $\frac{N}{2}$) $2 \cdot (\frac{N}{2} \lg(\frac{N}{2}) - \frac{N}{2} + 1) + N - 1 = N(\lg N - 1) - N + 2 + N - 1 = N \lg N - N + 1$.
QED.

Opgave 3.
a. We moeten laten zien dat er altijd minstens $n-1$ arrayvergelijkingen plaatvinden, dus zeker niet minder dan $n - 1$.
Geval (i): geen leider. In dat geval zal `gevonden` altijd False blijven, en zullen dus alle $n - 1$ rondes ($i = 1, \ldots, n - 1$) worden gedaan. Voor elke $i < n$ geldt dat direct na regel (4) geldt dat $j \leq n$, en dus dat de eerste test in regel (5) True is en derhalve de arrayvergelijking (2de test regel (5)) ten minste 1 keer gebeurt. Aangezien alle rondes plaatsvinden en per ronde minstens 1 arrayvergelijking, is het aantal arrayvergelijkingen in dit geval ten minste $n - 1$.
Geval (ii): wel een leider. Zeg dat de eerste leider op positie $k < n$ staat. In ronde k vinden $n-k$ arrayvergelijkingen plaats, want $A[k]$ zal vergeleken worden met alle $A[j]$ die erachter staan omdat de 2de test in regel (5) altijd True oplevert voor de leider. Verder geldt voor alle $i < k < n$ dat de loop van regel (5)/(6) de eerste keer wordt binnengegaan met $j \leq n$, en dus de 2de test in regel (5) ten minste 1 keer gebeurt. Dat zijn dus ten minste $k - 1$ arrayvergelijkingen. Totaal: ten minste $k - 1 + n - k = n - 1$ arrayvergelijkingen.
b. In het beste geval worden $n-1$ arrayvergelijkingen gedaan. Blijkbaar kan de $n-1$ dus gehaald worden.
Geval (i): alle $n - 1$ rondes worden gedaan, per ronde altijd minstens 1 vergelijking. In het beste geval hebben we dus voor elke $i = 1, 2, \ldots, n - 1$ slechts 1 vergelijking. Dit komt voor dan en slechts dan als de 2de test in regel (5) meteen de eerste keer faalt[3]. Dus dan en slechts dan als geldt: $A[i] \leq 2A[i + 1]$ voor alle $i = 1, 2, \ldots, n - 1$. Ofwel: $A[1] \leq 2 \cdot A[2] \leq 2^2 \cdot A[3] \leq \cdots \leq 2^{n-1} \cdot A[n]$. Voorbeelden: $1, 1, 1, 1, 1, 1, 1, 1$ en $7, 10, 15, 11, 6, 14, 9, 17$.
Geval (ii): wel een leider. Stel dat de voorste leider op plek k staat. Voor ronde k worden altijd $n - k$ vergelijkingen gedaan, en daarna stopt het algoritme. In de $k - 1$ rondes voor ronde k wordt per ronde altijd ten minste 1 arrayvergelijking gedaan. De enige manier om op totaal $n - 1$ uit te komen is als er in al die rondes precies 1 vergelijking plaatsvindt. Dat is (als bij (i)) d.e.s.d. het geval als $A[i] \leq A[i + 1]$ voor alle $i = 1, \ldots, k - 1$. Ofwel: $A[1] \leq 2 \cdot A[2] \leq 2^2 \cdot A[3] \leq \cdots \leq 2^{k-1} \cdot A[k]$ en $A[k] > 2A[j]$ voor $j = k + 1, \ldots, n$. Voorbeelden: $4, 5, 3, 6, 3, 1, 1, 1$ en $8, 5, 4, 10, 3, 4, 2, 1$.
c. De buitenste while-loop wordt het vaakst (en wel $n - 1$ keer) uitgevoerd als `gevonden` False blijft tot en met ronde $n - 2$. In ronde $n - 1$ (dus als $i = n - 1$) mag gevonden True worden of niet, dat maakt dan niet meer uit. Dit geeft dus de volgende twee situaties:

[3]Behalve eventueel voor $i = n - 1$, dan wordt altijd maar 1 vergelijking gedaan. Echter als die ene vergelijking True is, is $A[n - 1]$ de leider, en dit valt niet onder geval (i).

(1) `gevonden` blijft False, dus er is geen leider.
(2) `gevonden` wordt False voor $i = n - 1$, dus $A[n-1]$ is de leider.
d. Zowel in situatie (1) als in situatie (2) wordt de buitenste while-loop maximaal, dus $n-1$ keer uitgevoerd. Het worst case aantal arrayvergelijkingen gebeurt dus als de binnenste loop voor *elke* i het maximale aantal keer, dat wil zeggen $n - i$ keer, wordt uitgevoerd. Dit blijkt haalbaar voor situatie 1, maar niet helemaal voor situatie 2.
Situatie (1): geen leider. Worst case als voor elke i de binnenste while zo vaak mogelijk wordt gedaan, en het algoritme niet voortijdig stopt. Dit is het geval dan en slechts dan als voor elke i zo lang mogelijk geldt dat $A[i] \leq 2 \cdot A[n]$ en pas in de laatste doorgang (dus $j = n$) wordt ontdekt dat $A[i] \leq 2 \cdot A[n]$. Dit betekent dat A er zo uitziet: $A[1] > 2 \cdot A[2] > 2^2 \cdot A[3] > \cdots > 2^{n-2} \cdot A[n-1]$ en $A[i] \leq A[n]$ voor alle $i = 1, 2, \ldots, n-1$. Het aantal arrayvergelijkingen is dan $n - 1 + n - 2 + \cdots + 2 + 1 = \frac{1}{2}n(n-1)$.
Situatie (2): unieke leider op plek $n-1$, met $A[n-1] > 2 \cdot A[n]$. Merk op dat als voor een $1 \leq i < n-2$ geldt dat $A[i] > 2 \cdot A[n-1]$, dan is ook $A[i] > 2^2 \cdot A[n]$, en dan is $A[i]$ een eerdere leider. Dit kan in onze situatie niet voorkomen, dus er moet gelden dat $A[i] \leq 2 \cdot A[n-1]$ voor elke $i < n - 1$. We weten dus zeker dat de binnenste while voor elke $1 \leq i < n - 1$ in elk geval zal stoppen op $j = n - 1$. In het ergste geval blijft de 2de vergelijking uit regel 5 voor elke $i = 1, 2, \ldots, n-2$ zo lang mogelijk True, totdat uiteindelijk blijkt dat $A[i] \leq 2 \cdot A[n-1]$. Worst case rijtjes zijn dus alle rijtjes waarvoor geldt: $A[i] > 2 \cdot A[j]$ voor alle $j = i+1, \ldots, n-2$ en $A[i] \leq 2 \cdot A[n-1]$ voor $i = 1, 2, \ldots, n-2$ en $A[n-1] > 2 \cdot A[n]$, Dit geeft als karakterisering: $A[1] > 2 \cdot A[2] > 2^2 \cdot A[3] > \cdots > 2^{n-3} \cdot A[n-2]$, $A[i] \leq A[n-1]$ voor $i = 1, 2, \ldots, n-2$ en $A[n-1] > 2 \cdot A[n]$. Het aantal vergelijkingen in de worst case zal hiermee komen op: $n-2+n-3+\cdots+2+1+1 = \frac{1}{2}(n-1)(n-2)+1$. De eerste $n-2$ termen komen van de eerste $n-2$ rondes, de laatste 1 van de vergelijking van $A[n-1]$ met $A[n]$. Voorbeelden: $1000, 490, 240, 110, 50, 20, 11, 5$ en $3^8, 3^7, 3^6, 3^5, 3^4, 3^3, 20, 7$.

Opgave 4.
a. We geven een *niet-deterministisch polynomiaal* algoritme A voor InSet. A heeft als invoer een ongerichte graaf $\mathcal{G} = (V, E)$ en een geheel getal k met $k > 0$. We mogen aannemen (zie de opgave) dat $V = \{1, 2, \ldots, n\}$.
A doet bijvoorbeeld het volgende:

1. Fase 1 (gokfase)
Er wordt een willekeurige string s gegenereerd.
// Deze s stelt hopelijk een onafhankelijke deelverzameling van de knopen van $\mathcal{G}$ voor ter
// grootte k. Dit wordt in fase 2 gecontroleerd.

2. Fase 2 (controlefase)
// Hier wordt gecontroleerd of s inderdaad een onafhankelijke knoopverzameling ter
// grootte k voorstelt van de invoergraaf $\mathcal{G}$. De string s moet in dat geval dus een rijtje
// van k verschillende knopen uit V zijn. Tevens moet de onafhankelijkheidseigenschap
// gelden.
// Zodra een van onderstaande controles niet klopt wordt False geretourneerd of raak je
// in een oneindige lus of iets dergelijks.

- controleer of alle s_i getallen tussen 1 en n zijn (en dus knopen van $\mathcal{G}$ voorstellen): s aflopen en checken of voor elke s_i geldt dat $1 \leq s_i \leq n$. Dat kan in $O(|s|)$ stappen.

- controleer dat alle s_i verschillend zijn: s aflopen en voor elke s_i naar rechts lopen en kijken of die waarde nogmaals voorkomt; zo ja stop, zo nee volgende s_i bekijken. Dat kan in $O(|s|^2)$ stappen.
- controleer dat s precies k waardes bevat: s aflopen en tellen. Dat kan in $O(|s|)$ stappen.

Als aan deze voorwaarden is voldaan stelt s een deelverzameling van de knopen van $\mathcal{G}$ voor ter grootte k. Dit is dus een soort syntactische controle.
Hierna volgende de meest essentiële controle. Geef van die dan ook wat explicieter aan hoe je die controle doet. Laten we aannemen dat we voor $\mathcal{G}$ de adjacency-list representatie gebruiken. Als we bij deze controle zijn aanbeland weten we al dat s een knoopverzameling met de juiste grootte voorstelt.

- controleer dat s een *onafhankelijke* knoopverzameling is. Loop alle paren knopen (s_i, s_j) af (dat zijn er $O(|s|^2)$) knopen af, en controleer of s_j in de buurlijst van s_i voorkomt; zo ja stop, want dan zit er een tak tussen s_i en s_j; zo nee volgende paar $((s_i, s_{j+1})$ of (s_{i+1}, s_{i+2}) bekijken. Dat is voor elk paar $O(|E|) \subseteq O(|\mathcal{G}|)$. Totaal is dat dan $O(|s|^2 \cdot |\mathcal{G}|) \subseteq O(|s|^2 \cdot |x|)$ stappen.

Als de vier controles positief zijn stelt s een onafhankelijke knoopverzameling in $\mathcal{G}$ ter groottte k voor en retourneert het verificatie-algoritme (Fase 2) True. Zodra een van de controles niet klopt wordt False geretourneerd of raak je in een oneindige lus of zo.

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

Nu geldt: Fase 2 geeft True $\iff$ s is een onafhankelijke deelverzameling van de knopen van $\mathcal{G}$, bestaande uit k knopen. En dus: het antwoord van A op invoer $\langle \mathcal{G}, k \rangle$ is "ja" (per definitie) $\iff$ er is een executie van A die "ja" oplevert $\iff$ er is een string s waarop A in Fase 2 True geeft $\iff$ (ons concrete algoritme) er is een string s die een onafhankelijke deelverzameling van de knopen van $\mathcal{G}$ voorstelt, bestaande uit k knopen $\iff$ $\mathcal{G}$ heeft een onafhankelijke knoopverzameling bestaande uit k knopen $\iff$ $\langle \mathcal{G} \rangle$ is een ja-instantie van InSet.

Kortom, A is inderdaad een (niet-deterministisch) algoritme voor InSet. Bovendien is het polynomiaal. Immers, voor ja-executies stelt de gegokte s een deelverzameling van de knopen voor. In dat geval is dus $|s| \in O(|V|) \subseteq O(|\mathcal{G}|) \subseteq O(|x|)$. De (ja-)executie met die s kan derhalve zeker wel in: $O(|s|)$ (Fase 1) + $O(|s|)$ + $O(|s|^2)$ + $O(|s|)$ + $O(|s|^2 \cdot |x|)$ (Fase 2) + $O(1)$ (Fase 3) $\subseteq O(|x|) + O(|x|^2) + O(|x|)^3 \subseteq O(|x|^3)$ stappen. Er is dus voor ja-instanties een ja-executie die $O(|x|^3)$ stappen doet, en dat is polynomiaal in $|x|$, de grootte van de invoer. Derhalve is A polynomiaal.

Conclusie: A is een niet deterministisch polynomiaal ($O(|x|^3)$) algoritme voor InSet.

b. Aan te tonen: ϕ heeft een waarmakende waardering $\iff$ $\mathcal{G}_\phi$ heeft een onafhankelijke knoopverzameling met m knopen (met m het aantal clausules van ϕ).
"$\Longrightarrow$": Stel ϕ is een ja-instantie van 3Kleur. Dan bestaat er een waardering w die ϕ waarmaakt, en dus uit elke clausule minstens één literal True maakt. Kies nu uit elke

clausule één waargemaakte literal en laat I de verzameling knopen in $\mathcal{G}_\phi$ zijn die met deze literals correspondeert. Dan bevat I ten duidelijkste m verschillende knopen en elke knoop zit in een andere driehoek. Bewering: I is een onafhankelijke knoopverzameling. Immers, neem twee knopen v_j^r en v_k^s uit I. Deze komen uit verschillende driehoeken. Omdat de corresponderende literals l_j^r en l_k^s allebei True zijn onder w, kunnen ze niet elkaars tegengestelde/negatie zijn. Uit de constructie van $\mathcal{G}_\phi$ uit ϕ volgt dan dat er geen tak zit tussen v_j^r en v_k^s. Dit geldt voor alle paren knopen uit I, dus I is inderdaad een onafhankelijke knoopverzameling (ter grootte m).
"$\Longleftarrow$": Stel $\langle \mathcal{G}_\phi, m \rangle$ is een ja-instantie van InSet. Dan bevat $\mathcal{G}_\phi$ dus een onafhankelijke knoopverzameling J met m knopen. Uit de constructie van $\mathcal{G}_\phi$ volgt dat er nooit een tweetal knopen van J uit dezelfde driehoek kan komen, omdat binnen een driehoek alle drie knopen onderling verbonden zijn. Dus elke knoop uit J komt uit een andere driehoek. Het zijn er m, dus elke knoop moet uit een andere driehoek komen. Uit deze J construeren we nu een waardering w die ϕ waarmaakt. Definieer w als volgt: alle met de knopen uit J corresponderende literals zetten we op True, de logische variabelen die hierdoor nog geen waarde hebben gekregen geven we de waarde False (bijvoorbeeld). Merk op dat dit een goedgedefinieerde waardering is. Immers, voor elk tweetal knopen uit J geldt dat er geen tak tussen zit, dus (uit de constructie) zijn de corresponderende literals niet elkaars tegengestelde. Tevens maakt deze waardering uit elke clausule zeker één literal waar, dus maakt w de hele formule ϕ waar.
c. Q is NP-hard als $P \leq_\mathrm{P} Q$ voor alle P uit $\mathcal{NP}$.
Q is NP-volledig als (i) $Q \in \mathcal{NP}$ en (ii) Q is NP-hard.
d. (i) Gegeven is dat 3SAT $\in \mathcal{NPC}$ en 3SAT $\leq_\mathrm{P}$ InSet. Merk nu op dat In3Set $\in \mathcal{P}$. Dit komt omdat het brute force algoritme: genereer alle $\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$ drietallen knopen en controleer van elk van die drietallen of tussen elk tweetal knopen (dat zijn er maar 3) wel of niet een tak zit; polynomiaal is. Omdat $\mathcal{P} \subseteq \mathcal{NP}$ geldt dat In3Set $\leq_\mathrm{P}$ 3SAT (want 3SAT is NP-hard). Uit 3SAT $\leq_\mathrm{P}$ InSet en de transitiviteit van $\leq_\mathrm{P}$ volgt ten slotte dat In3Set $\leq_\mathrm{P}$ InSet.[4]
(ii) 3SAT $\in \mathcal{NPC}$, dus in het bijzonder is 3SAT NP-hard. Omdat 3SAT $\leq_\mathrm{P}$ InSet is ook Inset NP-hard (dit volgt uit de transitiviteit van $\leq_\mathrm{P}$. Samen met InSet $\in \mathcal{NP}$ volgt daaruit dat InSet $\in \mathcal{NPC}$. De omgekeerde reductie is dus niet nodig. Die zegt overigens ook niet heel veel, want daaruit haal je intuïtief alleen dat InSet hooguit even moeilijk is als een probleem (3SAT) dat heel erg moelijk is. Dat zegt dus helemaal niet dat InSet ook heel moeilijk zou moeten zijn.
(iii) Stel dat InSet $\leq_\mathrm{P}$ In3Set. Aangezien In3Set $\in \mathcal{P}$ kunnen we concluderen dat InSet ook in polynomiale tijd kan worden opgelost en dus in $\mathcal{P}$ zit. We weten bovendien uit (ii) dat InSet $\in \mathcal{NPC}$, waarmee we dus een NP-volledig probleem polynomiaal gereduceerd hebben naar een probleem uit $\mathcal{P}$. Volgens een stelling van college heeft dat tot gevolg dat dan $\mathcal{P} = \mathcal{NP}$. (Het bewijs van die stelling gebruikt de transitiviteit van $\leq_\mathrm{P}$ en de NP-hardheid van InSet om aan te tonen dat $\mathcal{NP} \subseteq \mathcal{P}$. Samen met $\mathcal{P} \subseteq \mathcal{NP}$ levert dit op dat $\mathcal{NP} = \mathcal{P}$.)

Opgave 5. Tape-symbolen: $\Gamma = \{0, 1, 2, b\}$
Toestanden: $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_Y, q_N\}$
En de transitietabel:

[4]Het feit dat In3Set $\in \mathcal{P}$ is ook ter sprake gekomen in opgave 68 bij het werkcollege

$q\backslash s$	0	1	2	b
q_0	$(q_0, 0, 1)$	$(q_0, 1, 1)$	$(q_1, 2, 1)$	$(q_N, b, -1)$
q_1	$(q_2, 2, -1)$	$(q_3, 2, -1)$	$(q_N, 2, 0)$ (*)	$(q_4, b. - 1)$
q_2	$(q_N, 0, 0)$ (*)	$(q_N, 1, 0)$ (*)	$(q_5, 0, 1)$	$(q_N, b, 0)$ (*)
q_3	$(q_N, 0, 0)$ (*)	$(q_N, 1, 0)$ (*)	$(q_5, 1, 1)$	$(q_N, b, 0)$ (*)
q_4	$(q_N, 0, 0)$ (*)	$(q_N, 1, 0)$ (*)	$(q_Y, b, -1)$	$(q_N, b, 0)$ (*)
q_5	$(q_N, 0, 0)$ (*)	$(q_N, 1, 0)$ (*)	$(q_1, 2, 1)$	$(q_N, b, 0$ (*)

(*) komt niet voor

Uitleg toestanden.
q_0: we beginnen in toestand q_0 en lopen naar rechts totdat we een 2 tegenkomen. Als x geen 2 bevat (we lezen een b) gaan we een stapje terug en eindigen we in de nee-toestand op het laatste karakter van x.
q_1: we hebben zojuist een 2 gelezen en het volgende karakter (daar staat de lees/schrijfkop nu op) moet dus straks naar links opgeschoven worden = over de 2 heengekopieerd worden. We laten voor het gemak een 2 achter, behalve als we aan het eind van x zijn. We gebruiken drie vervolgtoestanden: q_2 als we een 0 lezen; q_3 als we een 1 lezen en q_4 als we een b lezen. In alle gevallen een stapje terug zodat we op de te overschrijven 2 komen te staan.
q_2: we staan op de te overschrijven 2 en hebben zojuist een 0 gelezen; die moet over de 2 gekopieerd worden en we gaan weer een positie naar rechts. Daar staat een 2 (hebben we daar in toestand q_1 neergezet). Verder in toestand q_5.
q_3: we staan op de te overschrijven 2 en hebben zojuist een 1 gelezen; die moet over de 2 gekopieerd worden en we gaan weer een positie naar rechts. Daar staat een 2 (hebben we daar in toestand q_1 neergezet). Verder in toestand q_5.
q_4: we staan op de 2 en we hadden zojuist het eind van de string bereikt; de 2 wordt in b veranderd en we gaan een positie naar links, waar we eindigen in de ja-toestand.
q_5: we staan zeker op een 2 (zie werking toestand q_2 en q_3). Van daaruit moeten we het karakter erna ophalen en over deze 2 heenkopieren, dus ... naar rechts en naar toestand q_1.