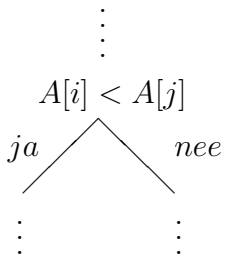
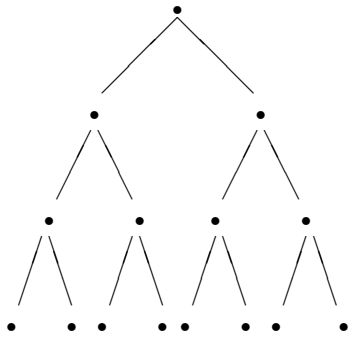
Uitgebreide uitwerking Tentamen Complexiteit, juni 2009

Opgave 1.
a. Een beslissingsboom beschrijft de werking van het betreffende algoritme (gebaseerd op arrayvergelijkingen) op elke mogelijke invoer. In de interne knopen staan de vergelijkingen die het algoritme doet:



De linkersubboom van een knoop beschrijft dan de verdere werking van het algoritme als $A[i] < A[j]$, de rechtersubboom als $A[i] > A[j]$. In de bladeren staat het eindantwoord: het algoritme stopt daar en is klaar. Een pad van de wortel naar een blad correspondeert met een executie (de achtereenvolgens gedane vergelijkingen) van het algoritme op een zekere invoer. De hoogte van de boom (wortel op niveau 0), zijnde de lengte van het langste pad, stelt dan precies het worst case aantal vergelijkingen voor dat het algoritme doet.
b. Gegeven een binaire boom met hoogte h en b bladeren.



Het aantal bladeren in een binaire boom met hoogte h is maximaal als de boom zo vol mogelijk is en alle bladeren derhalve op een geheel gevuld nivo h zitten. Er zijn dan 2^h bladeren. Dus: $b \leq 2^h$. Hieruit volgt meteen dat $h \geq \lg b$ is, en daar h geheel is geldt $h \geq \lceil \lg b \rceil$.
c. We hebben n verschillende elementen $A[1]$ t/m $A[n]$. We zoeken de grootste, de een na grootste en de twee na grootste. We wilen dus weten welk element de grootste is, welk de een na grootste en welk de twee na grootste. Het aantal mogelijke verschillende oplossingen is dan $n(n-1)(n-2)$. Immers: er zijn n array-elementen die de grootste kunnen zijn, dan resteren er nog $n-1$ mogelijkheden voor de een na grootste, en ten slotte nog $n-2$ voor de twee na grootste. Aangezien het algortme voor elk mogelijk invoerrijtje met werken, en dus al die $n(n-1)(n-2)$ verschillende oplossingen moet kunnen vinden, moet het aantal bladeren b in de corresponderende beslissingsboom $\geq n(n-1)(n-2)$ zijn. Dit combinerend met **a.** en **b.** krijgen we dus: worst case aantal vergelijkingen $= h \geq \lceil \lg b \rceil \geq$

$\lceil \lg n(n-1)(n-2) \rceil$. Aangezien verder $\lg n(n-1)(n-2) = \lg n + \lg(n-1) + \lg(n-2) \geq 3\lg(n-2)$, geldt: worst case aantal vergelijkingen $\geq 3\lg(n-2)$.
d.(i) De waarden op de even posities (dat zijn er $\lfloor \frac{n}{2} \rfloor$) zijn allemaal kleiner dan de waarden op de oneven posities (dat zijn er $\lceil \frac{n}{2} \rceil$). Het minimum zit dus tussen de getallen op de even posities en het maximum tussen de getallen op de oneven posities. Een algoritme (in woorden) dat de grootste en de kleinste waarde vindt is (bijvoorbeeld):

Bepaal(*) het minimum van de getallen op de even posities;
Bepaal (*) het maximum van de getallen op de oneven posities;
(*) op de gebruikelijke wijze

Dit algoritme doet $\lfloor \frac{n}{2} \rfloor - 1 + \lceil \frac{n}{2} \rceil - 1 = n - 2$ vergelijkingen.
Het algoritme in pseudocode:

```
min := A[2]; i := 4;
while i <= n do
   if A[i] < min then
      min := A[i];
   fi
   i := i+2;
od
max := A[1]; i := 3;
while i <= n do
   if A[i] > max then
      max := A[i];
   fi
   i := i+2;
od
```

d.(ii) Het algoritme uit **d.(i)** doet minder vegelijkingen dan de theoretische ondergrens, maar dat is *geen tegenspraak*. In **d.(i)** weten we namelijk iets extra's over het array, waardoor het mogelijk sneller kan dan het antal dat als ondergrens in de stelling genoemd wordt. De stelling gaat namelijk over het algemene geval, dus voor willekeurige arrays. Hier hebben we het over een zeer speciale invoer, met een nuttige extra eigenschap. Het algoritme maakt ook dankbaar gebruik van die eigenschap, en levert voor willekeurige arrays niet het goede antwoord.

Opgave 2.
a. Het array wordt a.h.w. in twee stukken van elk $\frac{n}{2}$ elementen gehakt, en beide stukken worden weer op dezelfde manier (recursieve aanroepen) gereorganiseerd. Dat levert de bijdrage $2W(\frac{n}{2})$. Na afloop daarvan hebben we de linkerhelft en de rechterhelft van het array in de juiste vorm gebracht, hetgeen betekent dat in de linkerhelft (en ook in de rechterhelft) de waarden op de even posities oplopend gesorteerd zijn en de waarden op de oneven posities aflopend. Vervolgens moet gezorgd worden dat de getallen op de even posities links, en de getallen op de even posities rechts oplopend worden gesorteerd. Hiertoe wordt een Merge gedaan (op twee rijtjes met in totaal $n/2$ elementen). Dat kost in de worst case $n/2-1$ vergelijkingen. Hetzelfde geldt voor het aflopend sorteren van de elementen op de oneven posities. Totaal dus nodig: $n/2-1+n/2-1 = n-2$ vergelijkingen

in het ergste geval. Dit verklaart de term $n-2$. Verder: als $n=2$ hebben we twee rijtjes ter lengte 1, die beide uiteraard al gesorteerd zijn. Er hoeft nu dus niets te gebeuren. Derhalve is $W(2)=0$.
b. Eerst proberen we de oplossing te vinden door $W(n)$ herhaald in zichzelf in te vullen. $W(n) = 2W(\frac{n}{2}) + n - 2 = 2\{2W(\frac{n}{4}) + \frac{n}{2} - 2\} + n - 2 = 2^2W(\frac{n}{2^2}) + 2\cdot n - 2^2 - 2 = 2^2\{2W(\frac{n}{2^3}) + \frac{n}{2^2} - 2\} + 2\cdot n - 2^2 - 2 = 2^3W(\frac{n}{2^3}) + 3n - 2^3 - 2^2 - 2 = \ldots =$ (algemene vorm, vermoedelijk) $2^l \cdot W(\frac{n}{2^l}) + l\cdot n - (2^l + 2^{l-1} + \ldots + 2)$ (neem $l = k-1$, dan $\frac{n}{2^l} = 2$) $= 2^{k-1}\cdot W(2) + (k-1)n - (2^{k-1}+2^{k-2}+\ldots+2) = 0 + (k-1)n - (2^k - 2) = n\lg n - n - n + 2 = n\lg n - 2n + 2$.
Nu nog bewijzen dat de gevonden $W(n) = n\lg n - 2n + 2$ inderdaad de oplossing is van de recurrente betrekking. Dit gaat m.b.v. volledige inductie.
(i) $W(n) = n\lg n - 2n + 2$ voldoet inderdaad aan $W(2) = 2\lg 2 - 4 + 2 = 0$.
(ii) Inductie-aanname: stel dat $W(n)$, de oplossing van de recurrente betrekking, gelijk is aan $n\lg n - 2n + 2$ voor alle tweemachten $n \geq 2$ met $n < N$ en met N een tweemacht ≥ 4. Dan moeten we laten zien dat dit dan ook geldt voor de oplossing van de recurrente betrekking voor deze N, ofwel dat $W(N) = N\lg N - 2N + 2$.
Er geldt: $W(N) =$ (recurrente betrekking) $2W(\frac{N}{2}) + N - 2 =$ (inductie-aanname) $2\cdot(\frac{N}{2}\lg\frac{N}{2} - 2\cdot\frac{N}{2} + 2) + N - 2 = N\lg\frac{N}{2} - 2N + 4 + N - 2 = N\lg N - N\lg 2 - N + 2 = N\lg N - 2N + 2$. QED
c.(i) Merge de $n/2$ elementen op de even posities (die zijn oplopend gesorteerd) en de elementen op de oneven posities (die zijn omgekeerd gesorteerd) in omgekeerde volgorde, dus van achter naar voren. Dat levert een oplopend gesorteerd array op, en kost in het slechtste geval $n-1$ vergelijkingen.
c.(ii) Eerst recursief reorganiseren en vervolgens mergen kost in het slechtste geval $n\lg n - 2n + 2 + n - 1 = n\lg n - n + 1$ vergelijkingen. Dat zijn precies evenveel vergelijkingen als Mergesort doet in de worst case.
Andere redenering: de geschetste methode komt eigenlijk op hetzelfde neer als Mergesort; het enige verschil is dat we in plaats van het array doormidden te delen, we in elke stap de splitsing maken door de elementen op de even posities te beschouwen als de ene helft, en de elementen op de oneven posities als de andere helft. Vervolgens ga je die helften dan beide "sorteren" (=reorganiseren; algoritme uit **a.,b**) en daarna Mergen (**c.**). Het "sorteren" gaat analoog aan Mergesort, namelijk recursief twee helften "sorteren" en vervolgens die twee helften via Mergen in de juiste vorm brengen. Het zal dus (naar verwachting) evenveel vergelijkingen kosten als Mergesort op het oorspronkelijke array.

Opgave 3.
a. Alles (in orde van grootte) afschatten op regel 8.
De test in regel 8 wordt altijd minstens $n-1$ keer gedaan. Immers, bij aanvang is `gehad` helemaal op f`False` geinitialiseerd, dus voor $j=1$ is de test in regel 3 `True` (omdat $1 \leq n-1$ aangezien $2 \leq n$) en de test in regel 4 eveneens; de body van de binnenste while wordt dus voor $j=1$ precies $n-1$ keer uitgevoerd want i loopt van 2 tot n. Ergo: #(regel 8) $\geq n-1 \geq 1$.

#(regel 1) = #(regel 2) = $1 \leq n-1 \leq$ #(regel 8)
#(regel 5) = #(regel 6) $\leq$ #(regel 4)
#(regel 4) = #(regel 18) = $n-1 \leq$ #(regel 8)
#(regel 3) = #(regel 4) +1 = $n \leq 2\cdot$ #(regel 8), dus O(#(regel 8))

#(regel 7) = #(regel 8) + 1 per doorgang door de buitenste while, dus in totaal: #(regel 7) $\leq$ #(regel 8) + $n-1 \leq 2 \cdot$ #(regel 8), dus #(regel 7) = O(#(regel 8))
#(regel 9) = #(regel 10) $\leq$ #(regel 8)
#(regel 12) = #(regel 8)
#(regel 14) = #(regel 5)
#(regel 15) $\leq$ #(regel 14)

Aldus is voor alle regels het aantal keren dat ze worden uitgevoerd afgeschat (in orde van grootte) op het aantal maal dat regel 8 wordt gedaan. Dus regel 8 is maatgevend.
b. Er worden minstens $n-1$ vergelijkingen gedaan, en dat aantal kan ook bereikt worden. Dat is dan automatisch het aantal vergelijkingen in de best case. Het aantal is alleen gelijk aan $n-1$ als na de ronde met $j=1$ de test in regel 4 steeds (*) `False` is (voor $j = 2, \ldots, n-1$ (!)) en dat kan alleen als `gehad`[2] t/m `gehad`[$n-1$] op `True` zijn gezet in de eerste ronde. Kortom, dat is alleen het geval als $A[1] = A[2] = \cdots = A[n-1]$. De waarde van $A[n]$ doet er niet toe. (*) Zodra `gehad`[j] namelijk een keer `True` is, wordt de test in regel 8 minstens 1 keer uitgevoerd, dus dan kom je in totaal al boven de $n-1$ vergelijkingen.
c. Het maximale aantal vergelijkingen wordt bereikt als voor elke $j = 1, \ldots n-1$ alle $n-j$ vergelijkingen uit de binnenste while worden gedaan, en dat komt alleen voor als `gehad`[j] `False` is voor elke bekeken j, dus voor $j = 1, \ldots n-1$. Met andere woorden: $A[j]$ is voor elke j ($j = 1, \ldots n-1$) nog niet eerder voorgekomen. Dus: worst case voor rijtjes waarvoor de eerste $n-1$ array-elementen verschillend zijn; alleen de waarde van $A[n]$ doet er niet toe. Het aantal vergelijkingen is dan: $n-1+n-2+\cdots+1 = \frac{1}{2}n(n-1)$.
d. Nu worden niet altijd voor vaste j met `gehad`[j] = `False` alle $n-j$ vergelijkingen gedaan. Dit hangt af van de waarden die al dan niet al eerder zijn geweest.
De best case is evenwel nog steeds $n-1$, want voor $j=1$ worden nog steeds $n-1$ vergelijkingen gedaan. Het komt nu echter voor meer rijtjes voor dan in **b.** gevonden. Namelijk, het best case komt voor als voor $j = 1, 2, \ldots, n-1$ ofwel `gehad`[j] = `True` ofwel `gehad`[i] = `True` voor alle $i = j+1, \ldots, n$. Dus ofwel alle $A[j]$ met $j = 1, \ldots, n-1$ zijn hetzelfde, ofwel er zit precies één andere waarde in het array, en de overige $n-1$ zijn hetzelfde. Dus best case invoerrjtjes hebben ofwel de vorm $a, a, a, \ldots, a$, ofwel $a, a, a, \ldots, b, \ldots, a$, waarbij de andere waarde b op elke plaats mag staan, niet alleen op positie n zoals bij **b.**.
e. We tellen het aantal vergelijkingen:
Algoritme zonder extra test: $n-1+n-2+\cdots+n-\frac{n}{2}$
Algoritme met extra test: $n-1+\frac{n}{2}-2+\cdots+\frac{n}{2}-\frac{n}{2}$
In beide gevallen levert ronde 1 hetzelfde aantal vergeijkingen; verder zijn de laatste $\frac{n}{2}$ getallen gelijk aan $A[1]$, dus die worden door regel 4 in beide gevallen overgeslagen. In beide gevallen hebben we dus $\frac{n}{2}$ termen waarvan de eerste gelijk is en de rest precies een factor $\frac{n}{2}$ scheelt vanwege de extra `gehad`-test. (In de binnenste while worden immers steeds in het tweede geval de laatste $\frac{n}{2}$ elementen niet bekeken.) In het tweede geval worden derhalve $(\frac{n}{2}-1)\cdot\frac{n}{2}$ vergelijkingen minder gedaan.

Opgave 4.
a. We geven een *niet-deterministisch polynomiaal* algoritme A voor MAX2SAT.
Idee voor het certificaat: een waardering die ten minste k clausules waarmaakt.
A heeft als invoer een logische formule ϕ in 3-CNF en een geheel getal $k > 0$: invoer

$x = \langle \phi, k \rangle$.
Opmerking: het aantal verschillende logische variabelen in ϕ, n, kan in $O(|\phi|^2)$ stappen worden bepaald, namelijk door ϕ af te lopen en voor elke logische variabele te controleren of die elders in ϕ ook voorkomt. Dit is polynomiaal, dus we kunnen wel aannemen dat het aantal logische variabelen n bekend is.
A doet het volgende:

1. Fase 1 (gokfase). Er wordt een string s gegenereerd, die we hierna interpreteren als een rij boolese waarden (T of F).
(Deze s stelt hopelijk een waardering voor van ϕ die ten minste k clausules waarmaakt; dit wordt in Fase 2 gecontroleerd.)

2. Fase 2 (controlefase). Hier wordt gecontroleerd of s een waardering van ϕ voorstelt die ten minste k clausules waarmaakt. Interpretatie van s: de i-de boolese waarde uit s stelt de waarde van de i-de logische variabele, x_i, voor.

- controleer dat er precies n logische waarden in s staan, dus dat s inderdaad een waardering van de logische variabelen van ϕ voorstelt. Dat kan in $O(|s|)$ stappen, namelijk ϕ aflopen en tellen.
- controleer dat s ten minste k clausules waarmaakt: v.l.n.r. door ϕ lopen; per clausule kijken of deze door s wordt waargemaakt, dus: literalwaardes in s ophalen ($O(|s|)$); als een literal in de bekeken clausule waar is een teller ophogen wn naar de volgende clausule gaan. Als aan het eind die teller $\geq k$ dan eindigt deze controle positief. Dit is $O(|\phi| \cdot |s|)$ $= O(|x| \cdot |s|)$.
Als beide controles positief zijn stelt s een waardering voor van ϕ die minstens k clausules waarmaakt, 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 $\Longleftrightarrow$ s is een waardering voor ϕ. die ten minste k clausules waarmaakt.

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

Nu geldt: het antwoord van A op invoer $\langle \phi, k \rangle$ is "ja" $\Longleftrightarrow$ er is een executie van A die "ja" oplevert $\Longleftrightarrow$ er is een string s waarop A in Fase 2 True geeft $\Longleftrightarrow$ er is een string s die een waardering van ϕ voorstelt die ten minste k clausules waarmaakt $\Longleftrightarrow$ er bestaat een waardering voor ϕ die ten minste k clausules waarmaakt $\Longleftrightarrow$ ϕ is een ja-instantie van MAX2SAT
Kortom, A is inderdaad een algoritme voor MAX2SAT. Bovendien is het polynomiaal. Immers voor ja-instanties is er een ja-executie waarin s een goede waardering voorstelt. Dus dan is $|s| = O(|\phi|) = O(|x|)$. De executie met die s is dus: $O(|s|)$ (Fase 1) + $O(|s|$ + $O(|x| \cdot |s|)$ (Fase 2) + $O(1)$ (Fase 3) = $O(|x|) + O(|x| \cdot |x|) = O(|x|^2)$.
Conclusie: A is een niet deterministisch polynomiaal ($O(|x|^2)$) algoritme voor MAX2SAT.
b.(i) T is een polynomiale reductie, d.w.z. T beeldt invoer x van het ene probleem P af op invoer $T(x)$ van het andere probleem Q, en er geldt:
(a) $T(x)$ is in een polynomiaal aantal stappen uit x te berekenen. Polynomiaal wil hier zegggen: $O(|x|^l)$ voor zekere $l \geq 0$.
(b) x is een ja-instantie voor probleem $P \Longleftrightarrow T(x)$ is een ja-instantie voor probleem Q.

Notatie: $P \leq_P Q$.
b.(ii) P is NP-hard als $Q \leq_P P$ voor alle Q uit $\mathcal{NP}$ (dus alle problemen uit $\mathcal{NP}$ reduceren (polynomiaal) tot P).
P is NP-volledig als (1) $P \in \mathcal{NP}$ en (2) P is NP-hard.
c. (i) is niet waar, (ii) wel.
(i) Uit MAX2SAT $\leq_P$ 3SAT volgt eigenlijk alleen maar (intuïtief) dat MAX2SAT hooguit even moeilijk is als het heel moeilijke (namelijk NP-volledige) probleem 3SAT. Dat zegt verder nog niets over de moeilijkheidsgraad van MAX2SAT. MAX2SAT zou zelfs nog in $\mathcal{P}$ ($\in \mathcal{NP}$) kunnen zitten. Dus (i) is onjuist.
(ii) Uit 3SAT $\leq_P$ MAX2SAT en 3SAT NP-volledig (en dus NP-hard) volgt dat ook MAX2SAT NP-hard is $^{(*)}$. Immers $Q \leq_P$ 3SAT $\leq$ MAX2SAT voor alle $Q \in \mathcal{NP}$, en uit de transitiviteit van $\leq_P$ volgt dan dat $Q \leq_P$ TFSAT voor alle $Q \in \mathcal{NP}$. Samen met het feit (bewezen in **a**) dat MAX2SAT $\in \mathcal{NP}$ weten we dus dat MAX2SAT NP-volledig is.
$^{(*)}$ Je kunt ook de volgende stelling uit het dictaat toepassen (komt op hetzelfde neer):
Stel P is een probleem waarvoor geldt dat $Q \leq_P P$ voor een of andere $Q \in \mathcal{NPC}$. Dan is P NP-hard. In deze opgave is $Q =$ 3SAT en $P =$ MAX2SAT.
(iii) Gegeven is dat 2SAT in $\mathcal{P}$ zit, en dus in $\mathcal{NP}$ (want $\mathcal{P} \subseteq \mathcal{NP}$). Omdat 3SAT NP-volledig is, dus in het bijzonder NP-hard, reduceert *elk* NP-probleem naar 3SAT, en derhalve ook 2SAT. Dus er bestaat inderdaad een polynomiale reductie van 2SAT naar 3SAT.
(iv) Stel dat er een polynomiale reductie van 3SAT naar 2SAT bestaat: 3SAT $\leq_P$ 2SAT. Omdat 2SAT $\in \mathcal{P}$ (gegeven) geldt dan ook dat 3SAT $\in \mathcal{P}$, en daarmee zitten alle NP-problemen in $\mathcal{P}$ en daarmee is $\mathcal{P} = \mathcal{NP}$ $^{(*)}$. Samengevat: als er zo'n reductie bestaat is $\mathcal{P} = \mathcal{NP}$, en dat is zeer onwaarschijnlijk. Conclusie: hoogstwaaschijnlijk bestaat zo'n reductie niet.
$^{(*)}$ Dit staat ook in de volgende stelling uit het dictaat:
Als een of ander NP-volledig probleem in $\mathcal{P}$ zit, dan is $\mathcal{P} = \mathcal{NP}$.