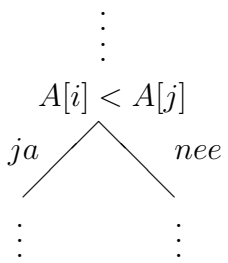
Uitgebreide uitwerking Tentamen Complexiteit, mei 2005

Opgave 1.
a. Elk algoritme (gebaseerd op arrayvergelijkingen) correspondeert met een binaire boom met in de interne knopen 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 $A[i] > A[j]$. Een pad van de wortel naar een blad correspondeert met een executie van het algoritme op een of andere invoer. De hoogte van de boom (wortel op niveau 0) stelt dan precies het worst case aantal vergelijkingen dat het algoritme doet voor. In de bladeren, waar het algoritme stopt, is het eindantwoord bekend. In het geval van **b.** is dat het tweetal indices dat de positie het grootste en het op een na grootste element aangeeft.
b. Er zijn hier $n(n-1)$ mogelijke antwoorden, namelijk alle mogelijke geordende paren van indices (i, j) (betekenis: $A[i]$ is het grootste element en $A[j]$ het op een na grootste). Deze mogelijkheden moeten allemaal kunnen voorkomen aangezien het algoritme voor elke invoerrij werkt. Dus in elke beslissingsboom corresponderend met een algoritme voor dit probleem geldt voor het aantal bladeren: $b \geq n(n-1)$. Uit $h \geq \lceil \lg b \rceil$ volgt dan dat voor zulke beslissingsbomen $h \geq \lceil \lg n(n-1) \rceil$. Aangezien $\lceil \lg n(n-1) \rceil \geq \lg n(n-1) = \lg n + \lg(n-1)$, is $h = \Omega(\lg n)$. Ergo, het aantal vergelijkingen in de worst case voor *elk* algoritme dat de grootste en de op een na grootste vindt is $\Omega(\lg n)$ (immers de afschatting gold voor *elke* beslissingsboom bij dit probleem).
c. Vergelijk de array-elementen twee aan twee: dit zijn $\frac{n}{2} = 2^{k-1}$ vergelijkingen.
Vergelijk de $\frac{n}{2}$ winnaars (=grootsten) weer twee aan twee; de $\frac{n}{4} = 2^{k-2}$ winnaars daarvan weer, etcetera. /p Herhaal dit totdat je één array-element overhoudt: dit is dan de grootste van allemaal. Er zijn nu $n-1$ wedstrijden gespeeld (=vergelijkingen gedaan): immers: $2^{k-1} + 2^{k-2} + \ldots + 2 + 1 = 2^k - 1 = n - 1$, en er waren k rondes nodig (na k stappen heb je nog maar 1 array-element over, met $k = \lg n$). De grootste heeft in alle rondes meegedaan, dus die heeft k wedstrijden gespeeld.
Nu moet de op een na grootste nog gevonden worden.
Bewering: dit moet een van de k spelers zijn die in het toernooi van de grootste verloren heeft, en wel de grootste van die k. Immers: de op een na grootste is kleiner dan precies 1 ander array-element, namelijk het grootste. De op een na grootste verliest dus nooit een vergelijking van een ander element dan de grootste. Hij heeft in het toernooi zeker een keer verloren, want hij is niet de grootste. Dat moet dus van de grootste zelf zijn geweest. Hij zit derhalve tussen de $k = \lg n$ verliezers van de grootste. Het kost nog eens $k - 1 = \lg n - 1$ vergelijkingen om de grootste van deze k verliezers te bepalen.
Het totaal aantal vergelijkingen is dus $n + \lg n - 2$.

d. Uit **b.** volgt slechts dat je *minstens* $\Omega(\lg n)$ vergelijkingen nodig hebt. Dit is slechts een ondergrens, die niet scherp hoeft te zijn. Het zou dus best beter kunnen. De toernooimethode (die $\Theta(n)$ is, dus echt slechter dan de ondergrens) zou dus zelfs optimaal kunnen zijn. Je weet uit **b.** en **c.** alleen dat een optimaal algoritme worst case complexiteit in orde van grootte tussen $\lg n$ en n heeft.
(We hebben overigens gezien dat de toernooimethode wel degelijk optimaal is. Met behulp van een adversary argument kun je aantonen dat elk algoritme voor dit probleem ten minste $n + \lceil \lg n \rceil - 2$ vergelijkingen moet doen. Deze ondergrens op de worst case complexiteit is wel scherp, want de toernooimethode is een voorbeeld van een algoritme dat $n + \lceil \lg n \rceil - 2$ vergelijkingen doet.)

Opgave 2.
Het is voldoende te kijken naar een speciale invoer en te bewijzen dat het aantal vergelijkingen daarvoor $\geq cn^2$ is voor zekere constante c (en vanaf zekere n_0). Immers: worst case aantal vergelijkingen $\geq$ aantal vergelijkingen bijzonder geval.
Kies de invoer zo dat de $\frac{n}{7}$ kleinste elementen op de plekken $7, 14, 21, \ldots, n$ staan, in oplopende volgorde. Bij het $\frac{n}{7}$-,$\frac{n}{49}$-, $\frac{n}{343}$-, $\ldots$, 49-, 7-sorteren gebeurt er niets met deze elementen, want er worden steeds alleen elementen vergeleken die een 7-voud van elkaar liggen. Dus na de op een na laatste ronde staan deze waarden nog steeds op dezelfde plek als bij het begin. Er zijn in die eerste rondes wel vergelijkingen gedaan en mogelijk elementen verwisseld, maar niet die op de plekken $7, 14, 21, \ldots, n$.
Pas bij het 1-sorteren gebeurt er iets met deze elementen en worden ze op hun juiste positie gezet. Ze moeten allemaal van plek $7i$ naar plek i. Dit alleen al kost $6 + 13 + 19 + 25 + \ldots + 6 * i + 1 + \ldots + 6 * (\frac{n}{7}) + 1 \geq 6 * (1 + 2 + 3 + \ldots + i + \ldots + \frac{n}{7}) = 3 * \frac{n}{7} * (\frac{n}{7} + 1) \geq 3 * \frac{n^2}{49}$ vergelijkingen. Dus in totaal worden voor deze invoer minstens $3 * \frac{n^2}{49}$ vergelijkingen gedaan.

Opgave 3.
a. Hetzelfde sorteeralgoritme (recursie) wordt toegepast op de eerste $n/3$ elementen, de middelste $n/3$ en de laatste $n/3$. Dat is dus $3 * S(n/3)$ vergelijkingen in de worst case. Vervolgens worden de drie rijtjes samengevoegd. Het ergste geval krijgen we als zolang mogelijk 2 vergelijkingen moeten worden gedaan, dus als we zo lang mogelijk drie rijtjes houden. Dit kan totdat je op een gegeven moment 3 rijtjes van elk 1 element over hebt. Het heeft $2(n-3)$ vergelijkingen gekost om zover te komen, want na elke vergelijking schuift precies 1 element door. Dan nog 1 keer 2 vergelijkingen, en vervolgens nog 1 vergelijking om de grootste van de twee resterende elementen te bepalen. Samen: $2(n-3) + 2 + 1 = 2n - 3$ vergelijkingen. We vinden dus: $S(n) = 3 * S(n/3) + 2n - 3$. De beginwaarde: als $n = 1$ is de rij al gesorteerd, dus je hoeft niets te doen: geen vergelijking nodig. (Eventueel ter controle: als $n = 3$ (de volgende drievoud) zijn er volgens het gegeven algoritme $2 + 1 = 3$ vergelijkingen nodig om te sorteren. Dit vinden we ook als we $n = 3$ in de recurrente betrekking invullen en $S(1) = 0$ gebruiken.) De recurrente betrekking voor $S(n)$ is dus:

$$S(n) = \begin{cases} 0 & n = 1 \\ 3\,S(\frac{n}{3}) + 2n - 3 & n > 1, n = 3^k \end{cases}$$

b. Eerst proberen we de oplossing te vinden door $S(n)$ herhaald in zichzelf in te vullen.
$S(n) = 3S(\frac{n}{3}) + 2n - 3 = 3 * (3S(\frac{n}{3^2}) + 2\frac{n}{3} - 3) + 2n - 3 = 3^2 S(\frac{n}{3^2}) + 2 * 2n - (3 + 3^2) = 3^2(3S(\frac{n}{3^3}) + 2\frac{n}{3^2} - 3) + 2 * 2n - (3 + 3^2) = 3^3 S(\frac{n}{3^3}) + 3 * 2n - (3 + 3^2 + 3^3) = \ldots =$

$3^l S(\frac{n}{3^l}) + l * 2n - (3 + 3^2 + 3^3 + \ldots + 3^l) =$ (neem nu $l = k$, gebruik dat $S(1) = 0$ en gebruik de hint) $3^k S(1) + k * 2n - 3(1 + 3 + 3^2 + \ldots + 3^{k-1}) = 2kn - \frac{3}{2}(3^k - 1) = 2n * {}^3\log n - \frac{3}{2}(n-1)$.
Gebruikt dat $n = 3^k$, dus $k = {}^3\log n$.
Nu nog bewijzen dat $S(n) = 2n * {}^3\log n - \frac{3}{2}(n-1)$ inderdaad de oplossing is van de recurrente betrekking. Dit gaat m.b.v. inductie.
$S(n) = 2n * {}^3\log n - \frac{3}{2}(n-1)$ voldoet inderdaad aan $S(1) = 0$.
Stel nu dat $S(n) = 2n * {}^3\log n - \frac{3}{2}(n-1)$ voor alle 3-machten n met $n < N$ en N een 3-macht > 1. Dan moeten we laten zien dat dan ook $S(N) = 2N * {}^3\log N - \frac{3}{2}(N-1)$.
Er geldt: $S(N) =$ (recurrente betrekking) $3 * S(N/3) + 2N - 3 =$ (inductie-aanname) $3 * (2\frac{N}{3} * {}^3\log \frac{N}{3} - \frac{3}{2}(\frac{N}{3} - 1)) + 2N - 3 = 2N({}^3\log N - 1) - \frac{9}{2}(\frac{N}{3} - 1) + 2N - 3 = 2N * {}^3\log N - 2N - \frac{3}{2}N + \frac{9}{2} + 2N - 3 = 2N * {}^3\log N - \frac{3}{2}N + \frac{3}{2} = 2N * {}^3\log N - \frac{3}{2}(N-1)$.
QED

Opgave 4.
a. Kenmerkend voor dit algoritme is het doen van optellingen (de sommen $A[i] + \cdots + A[j]$ worden bepaald en het antwoord vergeleken met t); de rest is veelal boekhouding. Intuïtief: het meeste werk zal gebeuren in de binnenste while-lus, dus als maat voor de complexiteit kun je het aantal keer dat regel 5 wordt uitgevoerd nemen. (Overigens is het aantal keer dat de tweede test in regel 4 wordt gedaan ook een goede maat voor de complexiteit.) We gaan dat wat preciezer maken.
#(regel 1) = 1
#(test in regel 2) $= n + 1$, want de buitenste while is in feite een for-loop
#(regel 3) = n = #(regel 9) = #(regel 7)
#(regel 8) $\leq$ #(regel 7)
#(regel 6) = #(regel 5)
Merk op dat voor alle $i \leq n$ de beide tests in regel 4 bij de eerste doorgang door de binnenste while True zijn (namelijk $j = i \leq n$ en hoeveel $< t$). Dus voor elke i wordt regel 5 ten minste 1 keer uitgevoerd en derhalve: #(regel 5) $\geq n$. Hieruit volgt, samen met wat hierboven staat, dat #(regel 1, 3, 6, 7, 8, 9) $\leq$ #(regel 5). En verder dat #(regel 2) = O(#(regel 5)). Nu nog het verband tussen regel 4 en regel 5.
Aangezien per doorgang door de buitenste while, dus voor elke i, #(regel 5) = #(volledige test regel 4 = True) = #(volledige test regel 4) -1 geldt voor het totaal aantal vergelijkingen #(volledige test regel 4) = #(regel 5) $+n \leq 2*$#(regel 5), dus #(regel 4) = O(#(regel 5)). Elke regel is dus in orde van groote af te schatten op regel 5.
Merk op dat we de twee tests uit regel 4 hier voor het gemak als één test bekijken. Je kunt eventueel onderscheid maken tussen de eerste en de tweede test uit regel 4, en voor elk van beide tests bewijzen dat ze (in orde van grootte) hooguit even vaak als regel 5 gedaan worden. Neem dan bijvoorbeeld aan dat er sprake is van short circuiting. In dat geval is #(tweede test regel 4) $\leq$ #(eerste test regel 4) en #(eerste test regel 4) = #(regel 5) $+1$ en verder als boven.
b. Beste geval (zo weinig mogelijk vergelijkingen) als voor elke i de binnenste while-lus meteen na de eerste doorgang stopt. Dan moet dus voor elke i hoeveel $\geq t$ worden direct bij de eerste stap ($i = j$), behalve eventueel als $i = n$ (want dan stopt de lus sowieso meteen na de eerste doorgang vanwege de eerste test in regel 4; de waarde van de tweede test doet er dan niet meer toe). Dat is het geval als $A[i] \geq t$ voor elke i behalve eventueel voor $i = n$. Het aantal optellingen is dan n, namelijk 1 voor elke i.
c. Worst case als de binnenste lus voor elke i zo ver mogelijk doorgaat, dat wil zeggen voor

j vanaf i tot en met n. Voor alle $j \leq n$ moet hoeveel dus $< t$ zijn. Als dan $j = n$ wordt (en hoeveel $= A[i] + \cdots + A[n-1] < t$), wordt $A[i] + \cdots + A[n]$ berekend. Vervolgens wordt j opgehoogd en stopt de while-lus. De waarde van $A[i] + \cdots + A[n]$ doet er dus niet meer toe (voor elke i). Voor de worst case moet dus gelden dat voor elke i: $A[i] + \cdots + A[k] < t$ voor $k = i, \ldots, n-1$. Dit is equivalent met de eis: $A[1] + \cdots + A[n-1] < t$. Het worst case aantal optellingen wordt dus gedaan d.e.s.d.a. A voldoet aan: $A[1] + \cdots + A[n-1] < t$. Dit kan ook echt voorkomen. (Merk op dat alle $A[i] \geq 1$, dus opdat dit kan voorkomen moeten we aannemen dat $t \geq n$.) Het aantal optellingen is dan $n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{1}{2}n(n+1)$.

d. In dit algoritme wordt in elke doorgang door de while-lus ofwel een + gedaan, en dan wordt j opgehoogd ofwel een $-$, en dan wordt i opgehoogd (tenzij hoeveel $= t$ en dan stopt het hele algoritme). Dus in elke ronde wordt ofwel i ofwel j opgehoogd (tenzij je stopt). Het worst case geval krijg je als je zolang mogelijk doorgaat dus als zo lang mogelijk hoeveel $< t$ blijft, dus als de lus pas stopt als i of j $n+1$ wordt en de andere index gelijk aan n. Dit komt bijvoorbeeld voor als je eerst helemaal naar rechts loopt met j (totdat $j = n$) en dan met i geheel naar rechts. Dit komt voor als $A[1] + \cdots + A[n-1] < t$ maar $A[1] + \cdots + A[n] > t$ en als ook $A[i] + \cdots + A[n] > t$ voor elke $i < n$. Dit is bijvoorbeeld het geval als $A[1] + \cdots + A[n-1] < t$ en $A[n] \geq t$ (bijvoorbeeld het rijtje $1, 1, 1, \ldots, 1, t$). Het aantal optellingen is dan n (regel 8) $+n$ (regel 9) $= 2n$.
Er zijn (veel) meer voorbeeldrijtjes waarvoor het algoritme $2n$ vergelijkingen doet. Zoals een rijtje met afwisselend een getal kleiner dan t en een getal groter dan t. Bijvoorbeeld: $1, t+1, 1, t+1, \ldots, 1, t+1$. Hiervoor worden j en i afwisselend opgehoogd.

Opgave 5.

a. We geven een *niet-deterministisch polynomiaal* algoritme A voor TSP+$\triangle$. A heeft als invoer een volledige ongerichte graaf $\mathcal{G}$ met $\mathcal{G} = (V, E)$, en gewichten $c(u, v)$ op de takken (u,v) (die voldoen aan de driehoeksongelijkheid) en een geheel getal $k \geq 0$. De invoer x is dus van de vorm: $x = < \mathcal{G}, c, k >$ (c de gewichtsfunctie die de gewichten op alle takken geeft). Veronderstel dat de knopen genummerd zijn met 1 t/m n, waarbij $n = |V|$.
A doet het volgende:
1. Fase 1. Genereer een random string s, hierna te interpreteren als een rij knopen uit V, dus als een rij gehele getallen.
2. Fase 2. Hier wordt gecontroleerd of s een Hamiltonkring voorstelt met totaalgewicht hooguit k.
* controleer dat s bestaat uit precies n integers (knoopnummers): s aflopen, tellen en vergelijken met n. Dat kan in O($|s|$) stappen. (Deze mag je ook weglaten, want volgt uit de volgende twee condities.)
* controleer dat alle integers tussen 1 en n zitten (wederom s aflopen): O($|s|$)
* controleer dat alle knopen uit s verschillend zijn: O($|s|^2$)
* controleer dat het totaalgewicht van de Hamiltonkring s (als de eerste drie controles positief waren) $\leq k$ is. Hiertoe de achtereenvolgende tweetallen knopen v_i, v_{i+1} uit s aflopen (O($|s|$) en voor elk van deze takken (v_i, v_{i+1}) het gewicht ophalen en optellen. Als we de adjacency-matrix-representatie voor gewogen grafen gebruiken is dat O(1), bij de adjacency-list is dat O($|V|$). Totaal kan het dus zeker in O($|s| \cdot |V|$) = O($|s| \cdot |x|$).
(De eerste drie controles zijn een soort syntactische controle: er wordt gekeken of s inderdaad een verzameling van precies n verschillende knopen uit V voorstelt. Dan is het hier dus automatisch een Hamiltonkring) Als een van deze vier tests negatief uitvalt wordt

(de rest niet meer gecontroleerd en wordt) door Fase 2 False teruggegeven (of oneindige loop), en anders True.
3. Uitvoerfase. Als Fase 2 True gaf, dan wordt ja uitgevoerd, anders geen uitvoer.

Merk op dat geldt: als x een ja-instantie is van TSP+$\triangle$, dan bestaat er een Hamiltonkring met totaalgeicht $\leq k$, dus dan is er een string s waarop Fase 2 True oplevert. M.a.w. als x een ja-instantie is, dan is er een executie van A die ja oplevert. Bovendien geldt voor die string s dat $|s| \leq |x|$. De executie met die s is dus $O(|x|) + O(|x|) + O(|x|^2) = O(|x|^2)$ (het werk uit Fase 1 en Fase 2 opgeteld).
Anderzijds, als x een nee-instantie is bestaat er juist geen Hamiltonkring met gewicht $\leq k$, dus zal Fase 2 ook voor geen enkele s True opleveren, dus dan is er geen executie van A die ja oplevert.
Conclusie: A is een niet deterministisch polynomiaal ($O(|x|^2)$) algoritme voor TSP+$\triangle$.
b. Stel dat we de adjacency-matrix-representatie voor (gewogen) grafen gebruiken. Een algoritme dat de gegeven transformatie T uitvoert doet bijvoorbeeld het volgende:
* maak een kopie van $\mathcal{G}$: dit wordt $\mathcal{G}'$. Dit is $O(|\mathcal{G}|)$.
* loop alle tweetallen knopen (u, v) $(u \neq v)$ af en voor elke tak die in $\mathcal{G}$ voorkomt maak je een overeenkomstige tak met gewicht 1 in $\mathcal{G}'$. Komt een tak niet voor in $\mathcal{G}$ dan wordt dat een tak met gewicht 2 in $\mathcal{G}'$. Dat komt hier neer op het aflopen van alle matrix-elementen uit de kopie: een 0 verander je in een 2 en een 1 blijft een 1. Dit geeft de graaf met de bijbehorende gewichten op de takken. Dit is ook $O(|\mathcal{G}|)$.
* $k = |V|$ nog bepalen. Dat betekent het aantal knopen tellen, dus ook $O(|\mathcal{G}|)$.
Bovenstaand algoritme construeert $T(\mathcal{G})$ in een polynomiaal aantal stappen, namelijk $O(|\mathcal{G}|)$. (Kijk zelf wat je voor complexiteit krijgt als je grafen met behulp van de adjacency-list representeert, of ...)
c. "$\Longrightarrow$": $\mathcal{G}$ heeft een Hamiltonkring $v_1, v_2, \ldots, v_n$ (dus met $n = |V|$ takken). Dan is dit (uiteraard) ook een Hamiltonkring in $\mathcal{G}'$. Bovendien bevat deze Hamiltonkring alleen takken die ook in $\mathcal{G}$ zitten, dus die hebben gewicht 1 in $\mathcal{G}'$. Zijn totaalgewicht $|V| * 1 = |V|$. Dus $\mathcal{G}'$ heeft een Hamiltonkring met totaalgewicht $\leq |V|$.
"$\Longleftarrow$": Zij $v_1, v_2, \ldots, v_n$ een Hamiltonkring in $\mathcal{G}'$ met totaalgewicht $\leq |V|$. Aangezien de kring $n = |V|$ takken bevat en $\mathcal{G}'$ alleen takken van gewicht 1 en 2 heeft moeten de gewichten van alle takken uit de kring wel 1 zijn. Met andere woorden: alle (v_i, v_{i+1}) en (v_n, v_0) zitten ook in $\mathcal{G}$. Dus $v_1, v_2, \ldots, v_n$ is ook een Hamiltonkring in $\mathcal{G}$.
d. 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.
e. Als HC $\in \mathcal{NPC}$ dan geldt in het bijzonder dat $Q \leq_P$ HC voor alle $Q \in \mathcal{NP}$. Als verder HC $\leq_P$ TSP+$\triangle$ dan geldt (vanwege de transitiviteit van $\leq_P$) dat dan ook $Q \leq_P$ TSP+$\triangle$ voor alle $Q \in \mathcal{NP}$. Met andere woorden: TSP+$\triangle$ is NP-hard (*). Aangezien ook TSP+$\triangle$ $\in \mathcal{NP}$ volgt uit **a.**, **b.** en **c.** dat TSP+$\triangle$ NP-volledig is. Dus bewering (ii) is waar.
Uit TSP+$\triangle \in \mathcal{NPC}$ en HC $\leq_P$ TSP+$\triangle$ volgt eigenlijk alleen maar (intuïtief) dat HC eenvoudiger is dan een heel moeilijk (namelijk NP-volledig) probleem. Dat zegt verder nog niets over de moeilijkheidsgraad van HC. HC zou zelfs nog in $\mathcal{P}$ kunnen zitten. Dus bewering (i) is onwaar.

(*) Dit komt eigenlijk neer op de volgende stelling uit het dictaat:
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 $Q =$ HC en $P =$ TSP+$\triangle$.