Complexiteit

door J.M. de Graaf

LIACS, Universiteit Leiden

www.liacs.leidenuniv.nl/~graafjmde/COMP/

voorjaar 2019

1 Introductie

1.1 Analyseren van algoritmen

De analyse van algoritmen onderscheidt ruwweg:

- correctheid van algoritmen
- (tijd)complexiteit: hoeveelheid werk
- ruimtecomplexiteit: hoeveelheid geheugen
- optimaliteit: kan het nog beter?

De correctheid van algoritmen wordt besproken in het college Programmeren en Correctheid, de overige punten komen bij het vak Complexiteit aan de orde, met de nadruk op tijdcomplexiteit en optimaliteit. We bekijken altijd sequentiële algoritmen.

Vraag: hoe meten we de hoeveelheid werk die een algoritme doet?

1.2 Complexiteit = tijdcomplexiteit = hoeveelheid werk

- we willen een maat voor de hoeveelheid werk die ons iets vertelt over de efficiëntie van de *methode* die het algoritme gebruikt
- deze maat moet *onafhankelijk* zijn van de gebruikte computer, programmeertaal, implementatiedetails en dergelijke
- de complexiteit hangt gewoonlijk af van de *grootte van de invoer*: hoe groter de invoer, hoe hoger de complexiteit

1.3 Voorbeeld

Probleem:
Gegeven een array A ($A[1], A[2], \ldots, A[n]$, ongesorteerd) met $n \geq 1$ gehele getallen. Gevraagd het maximum van deze getallen.

Algoritme:

```
(1)   max := A[1];
(2)   index := 2;
(3)   while index ≤ n do
(4)       if max < A[index] then
(5)           max := A[index];
(6)       fi
(7)       index := index + 1;
(8)   od
(9)   return max;
```

We turven de hoeveelheid werk die dit algoritme doet.
Regel 1 en 2 gebeuren 1 keer, evenals regel 9. De test in regel 3 wordt n keer gedaan, en de test in regel 4 $n-1$ keer. Hetzelfde ($n-1$ keer) geldt voor regel 7. Regel 5 wordt $\leq n-1$ keer uitgevoerd. In totaal worden er ($\leq$) $4n-1$ operaties gedaan: de hoeveelheid werk is dus hooguit $4n-1$. Dit hangt af van de grootte van de invoer n. Aangezien niet alle operaties evenveel tijd kosten wegen ze niet allemaal even zwaar en is de factor $4n-1$ eigenlijk te precies. Beter is: de hoeveelheid werk bedraagt *in orde van grootte* n. Notatie: $\Theta(n)$. Merk op dat het algoritme gebaseerd is op het doen van arrayvergelijkingen. Uit deze vergelijkingen krijg je de informatie die je nodig hebt om het maximum te bepalen. Het aantal arrayvergelijkingen van de vorm $max < A[index]$ is verder een goede maat voor de complexiteit van het algoritme (namelijk $\Theta(n)$). Deze vergelijking is een *basisoperatie* voor dit algoritme.

1.4 Basisoperatie

Om de complexiteit van een algoritme te bepalen kun je volstaan met het tellen van het aantal keer dat een geschikte basisoperatie plaatsvindt. Het aantal uitgevoerde basisoperaties geeft immers een goed idee van de hoeveelheid werk die het algoritme doet. Dus:

- identificeer een operatie die *fundamenteel* is voor het algoritme (dus geen boekhoudoperaties zoals tellerophogingen)
- het totale aantal uitgevoerde operaties moet ruwweg evenredig zijn (in orde van grootte) aan het aantal basisoperaties, ofwel:
- alle andere operaties worden (in orde van grootte) *hooguit even vaak* uitgevoerd als de basisoperatie
- de basisoperatie is dus maatgevend voor de complexiteit van het algoritme

Het aantal keer dat de basisoperatie plaatsvindt is derhalve een goed criterium waarmee je verschillende algoritmen voor hetzelfde probleem kunt vergelijken. Je moet dan wel algoritmen uit dezelfde klasse vergelijken, namelijk die gebaseerd zijn op de betreffende basisoperatie.

1.5 Worst, average en best case complexiteit

De complexiteit van een algoritme

- hangt in het algemeen af van de grootte (n) van de invoer: $f(n)$
    - . bepaal een basisoperatie (en een maat voor de invoergrootte)
    - . tel het aantal basisoperaties $\Longrightarrow f(n)$
    - . van belang is de orde van grootte van $f(n)$: O, Θ, Ω
- hangt af van de soort invoer: *worst case, average case, best case*

. voor welke invoer doet het algoritme het maximale (minimale) aantal basisoperaties en hoeveel zijn dat er? $\Longrightarrow$ worst (best) case complexiteit

. worst case complexiteit is $g(n)$: het algoritme doet voor elke mogelijke invoer *hooguit* $g(n)$ stappen

. best case complexiteit is $h(n)$: het algoritme doet voor elke mogelijke invoer *minstens* $h(n)$ stappen

. worst case geeft dus een *bovengrens*, best case een *ondergrens*

. average case complexiteit is de complexiteit gemiddeld over alle mogelijke invoeren

Als we verschillende algoritmen hebben voor hetzelfde probleem kunnen we de complexiteit(en) ervan vergelijken ..... optimaliteit van een algoritme.

1.6 Optimaliteit

- bestaat er een efficiënter algoritme voor het probleem?

- heeft te maken met de *complexiteit* van het *probleem*: soms kan het niet beter

- een simpele (doch meestal niet al te scherpe) ondergrens voor de complexiteit kun je soms verkrijgen door het aantal invoer- en uitvoeritems te tellen

  Voorbeeld 1. Een ondergrens op de (worst case) complexiteit van het probleem van het (elementsgewijs) optellen van twee n bij n matrices is $\Omega(n^2)$. Immers: altijd zullen in elk geval alle $2n^2$ matrixelementen van de invoer gelezen (je hebt ze i.h.a. allemaal nodig voor het berekenen van de som) en de n^2 matrixelementen van de sommatrix gegenereeerd moeten worden. Deze ondergrens is scherp: het kán ook in $\Theta(n^2)$ stappen.
  Voorbeeld 2. Bekijk nu het Handelsreizigersprobleem: gegeven een volledige (alle takken zijn aanwezig) graaf met n knopen (steden), met bijbehorende afstandsmatrix. Gevraagd: een Hamiltonkring met de kleinste totaallengte. Een ondergrens op de complexiteit is dan $\Omega(n^2)$, want alle $\frac{1}{2}n(n-1)$ afstanden moeten zeker gelezen worden. (Er is eenvoudig een voorbeeld te geven waaruit blijkt dat je, als je niet alle $\frac{1}{2}n(n-1)$ afstanden bekeken hebt, het verkeerde antwoord kunt krijgen, d.w.z. een Hamiltonkring die niet de minimale lengte heeft). Deze ondergrens is niet scherp; er is zelfs geen polynomiaal algoritme voor dit probleem bekend.

  Pas op: niet altijd is het noodzakelijk dat een algoritme voor een probleem alle invoerelementen bekijkt. Pas dus erg op met dit argument. Voor het probleem van het zoeken naar een waarde in een gesorteerd array, bijvoorbeeld, is $\Omega(n)$ geen ondergrens. Je moet dus echt aantonen dat je argument juist is.

- een betere ondergrens vind je over het algemeen door -wat preciezer- naar de basisoperatie te kijken

- je bewijst complexiteit van een probleem zo binnen een bepaalde *klasse van algoritmen*, bijvoorbeeld de klasse van algoritmen gebaseerd op het doen van arrayvergelijkingen

- bewijs stellingen die een *ondergrens* opleveren voor het aantal basisoperaties dat *nodig* is om het probleem op te lossen, d.w.z.

- laat zien dat *elk algoritme* voor het probleem *minstens* ... basisstappen moet doen in de ... case (meestal worst case)

- om een ondergrens te bewijzen kunnen we bijvoorbeeld een beslissingsboomargument of een adversary-argument gebruiken, maar soms is een ad hoc argument voldoende (zie voorbeeld hieronder, waar een bewijs uit het ongerijmde is gebruikt)

- de worst case complexiteit van een algoritme dat het probleem oplost levert een *bovengrens* voor de complexiteit van het probleem: het probleem *kan opgelost worden* in *hooguit* ... stappen

Voorbeeld: elk algoritme (gebaseerd op het doen van arrayvergelijkingen; de basisoperatie is dus het vergelijken van array-elementen) dat het maximum van n array-elementen bepaalt, moet in de worst case ten minste $n-1$ vergelijkingen doen.
We kunnen deze bewering bijvoorbeeld uit het ongerijmde bewijzen.
Bewijs: we mogen aannemen dat het array n *verschillende* waarden bevat[1] en bewijzen dat elk algoritme op dat soort rijtjes altijd ten minste $n-1$ vergelijkingen moet doen in de worst case.
We bewijzen uit het ongerijmde, dus stel dat de bewering niet waar is. Dat betekent dat er een algoritme bestaat dat in de worst case slechts hooguit $n-2$ vergelijkingen nodig heeft. Dat algoritme doet dan op elk invoerrijtje hooguit $n-2$ arrayvergelijkingen.
In een array met n elementen die allemaal verschillen, zijn er $n-1$ *niet* het maximum. Die moeten dus allemaal kleiner zijn dan minstens één ander array-element. Elke arrayvergelijking levert precies één verliezer (de kleinste van de twee) op, dus door het doen van één vergelijking heb je precies één array-element als "niet de grootste" geïdentificeerd. Stel nu dat je algoritme $n-2$ of minder arrayvergelijkingen doet (*); dan weet je aan het eind van hooguit $n-2$ array-elementen zeker dat die niet het maximum zijn. De overige twee zouden beide nog de grootste kunnen zijn; het algoritme kan in het algemeen nooit met zekerheid zeggen welk van de twee de grootste is. Dit is in tegenspraak met de correctheid van je algoritme. Aanname (*) is dus fout. Conclusie: er zijn minstens $n-1$ vergelijkingen nodig.
Merk op dat we hier expliciet gebruiken dat we algoritmen bekijken die werken op alle mogelijke rijtjes, dus ook op rijtjes waarvan we vooraf niets weten over de ordening; als twee elementen niet vergeleken zijn weten we in dat geval dan ook niet welke van de twee de grootste is.
Gevolg: het algoritme uit paragraaf **1.3** is optimaal binnen de bekeken klasse van algoritmen.

1.7 Recursieve algoritmen

- in het algemeen is hier het *aantal recursieve aanroepen* een goede maat voor de hoeveelheid werk

- maar je kunt ook gewoon het aantal basisoperaties tellen

[1] Immers: het aantal vergelijkingen in de worst case is altijd groter of gelijk aan het aantal vergelijkingen voor een speciale invoer. Een ondergrens voor speciale gevallen is dus zeker ook een ondergrens voor de worst case.

- een en ander leidt tot *recurrente betrekkingen*

Voorbeeld: onderstaand algoritme bepaalt recursief het maximum van n getallen $(A[1], \ldots, A[n])$.

```
int grootste(int A[ ], n) {
    if n = 1 then
        return A[n];
    else
        max := grootste(A, n − 1);
        if A[n] > max then
            max := A[n];
        fi
    fi
    return max;
}
```

Definieer $C(n)$ als het aantal recursieve aanroepen, dan voldoet C aan de volgende recurrente betrekking:

$$C(n) = \begin{cases} 1 & n = 1 \\ C(n-1)+1 & n > 1 \end{cases}$$

Oplossen van deze recurrente betrekking geeft: $C(n) = n$. Merk op dat we hier ook gewoon het aantal vergelijkingen kunnen tellen; in elke aanroep behalve de laatste ($n = 1$) wordt een vergelijking gedaan; het aantal aanroepen is dan ook 1 meer dan het aantal arrayvergelijkingen, en derhalve in orde van grootte even groot (als $n > 1$).

1.8 Opgave

Gegeven een array A $(A[0], A[1], \ldots, A[n])$, die op plekken 1 t/m n n verschillende positieve (> 0) getallen bevat ($A[0]$ wordt als sentinel, een soort barrière waardoor je niet uit het array loopt, gebruikt). Het algoritme hieronder sorteert A oplopend.

a. Leg uit waarom het vergelijken van array-elementen (regel 3, tweede test) een goede basisoperatie is als $n > 1$. Hoe zit het trouwens als $n = 1$?
b. Bekijk worst case en best case.

```
(1)   A[0] := 0; i := 1;
(2)   while i < n do
(3)       while i < n and A[i] < A[i + 1] do
(4)           i := i + 1;
(5)       od
(6)       if i < n then
(7)           wissel(A[i], A[i + 1]);
(8)           i := i − 1;
(9)       fi
(10)  od
```

2 Wiskundige achtergrond

2.1 Definities (en gevolgen) en sommaties

- $\lfloor x \rfloor$ = het grootste gehele getal $\leq x$; $\lceil x \rceil$ = het kleinste gehele getal $\geq x$
  Er geldt: $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$ en $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n$ voor elk geheel getal $n \geq 0$
- $\lceil \frac{n}{2} \rceil = \begin{cases} \frac{n}{2} & \text{als } n \text{ even} \\ \frac{n+1}{2} & \text{als } n \text{ oneven} \end{cases}$ $\qquad \lfloor \frac{n}{2} \rfloor = \begin{cases} \frac{n}{2} & \text{als } n \text{ even} \\ \frac{n-1}{2} & \text{als } n \text{ oneven} \end{cases}$
- $\log_b x = y \iff b^y = x$
  Notatie: $\lg x = \log_2 x$ en $\ln x = \log_e x$
  Er geldt: $\log_b x = \frac{\log_c x}{\log_c b}$
- $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$
- Sommaties:
  $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$
  $\sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$
  $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$
  $\sum_{i=a}^{b} q^i = \frac{q^a - q^{b+1}}{1-q}$ voor $q \neq 1$

2.2 Tellen

Soms is het van belang om bijvoorbeeld het aantal permutaties of het aantal deelverzamelingen te kennen. Hieronder een klein (onvolledig) overzichtje.

- Het aantal verschillende geordende rijtjes ter lengte k dat je kunt maken met de getallen (objecten) 1 t/m n, waarbij het rijtje niet uit verschillende getallen hoeft te bestaan is n^k.
  Hier is dus de volgorde van belang is, en elk getal mag meerder malen voorkomen.

- Het aantal geordende rijtjes ter lengte k dat je kunt maken met de getallen 1 t/m n, waarbij het rijtje uit verschillende getallen moet bestaan (en dus moet $k \leq n$) is $n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$
  Hier is dus de volgorde van belang, en alle getallen moeten verschillen.

- Het aantal permutaties (volgordes) van de getallen 1 t/m n (speciaal geval van het voorgaande met $k = n$) is $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$.

- Het aantal manieren om k getallen uit n stuks te kiezen ($k \leq n$) is $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$.
  Nu is de volgorde niet van belang, maar de getallen moeten wel verschillend zijn.
  In feite tellen we hier het aantal verschillende deelverzamelingen van 1 t/m n ter grootte k.

- Het totaal aantal verschillende deelverzamelingen van een verzameling van n objecten is 2^n.

2.3 O, Θ en Ω

Laat $f, g : \mathbb{N} \longrightarrow \mathbb{R}$ (meestal $\mathbb{R}^+$)

- $f \in O(g)$ (of $f(n) \in O(g(n))$) betekent: er bestaan constanten c en n_0 (beide > 0) zodat $0 \leq f(n) \leq cg(n)$ voor alle $n \geq n_0$: asymptotische *bovengrens*
  Betekent: f groeit hooguit even hard als g.

- $f \in \Omega(g)$ (of $f(n) \in \Omega(g(n))$) betekent: er bestaan constanten c' en n_0' (beide > 0) zodat $0 \leq c'g(n) \leq f(n)$ voor alle $n \geq n_0'$: asymptotische *ondergrens*
  Betekent: g groeit hooguit even hard als f, ofwel f groet minstens even hard als g.

- $f \in \Theta(g)$ (of $f(n) \in \Theta(g(n))$) betekent: er bestaan constanten c_1, c_2 en n_0'' (alle > 0) zodat $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ voor alle $n \geq n_0''$: asymptotisch *gedrag*
  Betekent: f en g groeien even hard.

Naamgeving:
$\Theta(1)$: constant; $\Theta(\lg n)$: logaritmisch; $\Theta(n)$: lineair; $\Theta(n^k)$ met $k > 0$: polynomiaal; $\Theta(2^n)$, $\Theta(a^n)$ met $a > 1$: exponentieel

Stelling:
(1) $f \in \Theta(g) \iff f \in O(g)$ en $f \in \Omega(g)$
(2) $f \in O(g) \iff g \in \Omega(f)$

Stelling:
(1) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \alpha$ met $0 < \alpha < \infty \Longrightarrow f \in \Theta(g)$
(2) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow f \in O(g)$, maar $g \notin O(f)$ (ofwel $f \notin \Omega(g)$)
(3) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Longrightarrow f \in \Omega(g)$, maar $f \notin O(g)$

n-faculteit:
(1) $n! = 1 * 2 * 3 * \cdots * n$
(2) $n! \in O(n^n)$ (want $n! \leq n^n$) en $n! \in \Omega(2^n)$
De ondergrens kan wat scherper: $n! \geq (\frac{n}{2})^{\frac{n}{2}}$ voor n even en $n! \geq (\frac{n}{2})^{\frac{n+1}{2}}$ voor n oneven. (Immers: $n! = 1 * 2 * \cdots * \frac{n}{2} * (\frac{n}{2} + 1) * \cdots * (n-1) * n \geq \frac{n}{2} * \frac{n}{2} * \cdots * \frac{n}{2} = (\frac{n}{2})^{\frac{n}{2}}$ voor n even. Analoog n oneven.) Derhalve: $n! \geq (\frac{n}{2})^{\frac{n}{2}}$ als $n \geq 2$.
(3) Uit het voorgaande: $\lg(n!) \geq \frac{n}{2}(\lg n - 1)$ en $\lg(n!) \leq n \lg n$. Ergo $\lg(n!) \in \Theta(n \lg n)$.
(4) $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$: formule van Stirling

2.4 Recurrente betrekkingen

De recurrente betrekkingen die in dit vak voorkomen zullen we als volgt oplossen. Eerst vullen we de recurrente betrekking enige malen in zichzelf in (*iteratie*) om een idee te krijgen van de oplossing. Van de aldus gevonden (mogelijke) oplossing bewijzen we via *substitutie* en met behulp van *volledige inductie* dat het inderdaad de oplossing is. Zie verder het college en werkcollege en oude tentamens.
Hieronder enkele voorbeelden van recurrente betrekkingen met hun oplossing.

1. $T(n) = \begin{cases} 1 & n = 1 \\ 2T(\frac{n}{2}) + n & n = 2^k > 1 \end{cases}$

   Oplossing: $T(n) = n + n \lg n \in \Theta(n \lg n)$

2. $T(n) = \begin{cases} 1 & n = 1 \\ 2T(\lfloor \frac{n}{2} \rfloor) + n & n > 1 \end{cases}$

   Dan geldt: $T(n) \in \Theta(n \lg n)$
   Dit resultaat vind je door m.b.v. volledige inductie zowel $O(n \lg n)$ als $\Omega(n \lg n)$ te bewijzen, ofwel door een handige stelling te gebruiken. Die stelling[2] zegt dat onder bepaalde voorwaarden geldt: als $T(n) \in \Theta(f(n))$ voor tweemachten $n = 2^k$ (of algemener: voor $n = a^k$ met $a \geq 2$), dan is $T(n) \in \Theta(f(n))$ voor alle n.

3. $T(n) = \begin{cases} 1 & n = 1 \\ T(n-1) + 1 & n > 1 \end{cases}$

   Oplossing: $T(n) = n$

4. $T(n) = \begin{cases} 3 & n = 1 \\ T(n-1) + n - 1 & n > 1 \end{cases}$

   Oplossing: $T(n) = 3 + \frac{1}{2}n(n-1)$

3 Beslissingsbomen

Beslissingsbomen worden gebruikt om een ondergrens voor de worst case complexiteit van een probleem te vinden: laat zien dat *alle* mogelijke algoritmen voor het probleem ten minste ... basisoperaties doen door iets te zeggen over de beslissingsbomen die met die algoritmen corresponderen.
Bij een algoritme gebaseerd op het doen van vergelijkingen kan men een beslissingsboom geven die de werking van dat algoritme beschrijft op alle mogelijke invoeren. In de (interne) knopen van de boom staan dan de vergelijkingen, en een pad in de boom geeft een executie van het algoritme aan. Dat wil zeggen een serie achtereenvolgende vergelijkingen die voor de betreffende invoer leiden tot de oplossing van het onderhavige probleem. De hoogte van de boom (= lengte van het langste pad = aantal knopen op het langste pad - 1) geeft dan (ongeveer) het maximale aantal vergelijkingen dat het algoritme doet (worst case!).
We bewijzen eerst een nuttige stelling, daarna bekijken we beslissingsbomen corresponderend met algoritmen gebaseerd op arrayvergelijkingen ($A[i] < A[j]$), zoals algoritmen voor het vinden van het maximum, of sorteeralgoritmen. Als tweede voorbeeld bekijken we beslissingsbomen voor zoekalgoritmen, gebaseerd op sleutelvergelijkingen ($X =, < A[i]$).

3.1 Binaire bomen

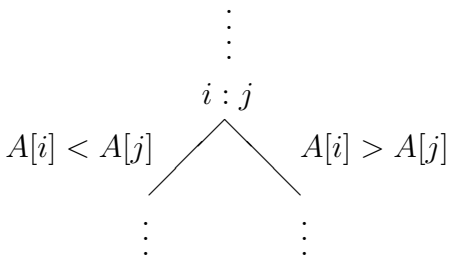
Stelling: Gegeven een *binaire boom* met n knopen (en b bladeren) en hoogte h. Dan geldt: $h \geq \lceil \lg b \rceil$ en $h \geq \lceil \lg(n+1) \rceil - 1 = \lfloor \lg n \rfloor$.

[2] Smoothness rule; zie bijvoorbeeld Introduction to the Design & Analysis of Algorithms, Anany Levitin

Hint bij het bewijs: een binaire boom met hoogte h bevat het maximaal haalbare aantal knopen (c.q. bladeren) als de boom geheel gevuld is. Hierbij is de hoogte van een binaire boom het hoogste niveau dat voorkomt, waarbij de wortel van de boom op niveau 0 zit.

3.2 De grootste vinden

We bekijken hier een array van n elementen, waaruit we de grootste waarde willen weten. We veronderstellen even dat alle array-elementen verschillend zijn. Elk algoritme voor dit probleem dat gebaseerd is op arrayvergelijkingen correspondeert nu met een binaire boom met in de interne knopen de vergelijkingen $A[i] < A[j]$ (genoteerd als $i : j$) die het algoritme doet:



De linkersubboom beschrijft dan de verdere werking van het algoritme als $A[i] < A[j]$, de rechtersubboom als $A[i] > A[j]$. In de bladeren, waar het algoritme stopt, is het eindantwoord (in dit geval de index van het grootste element) bekend. De bladeren zijn hier dus echt anders dan de interne knopen: zij bevatten geen vergelijking, maar het eindantwoord. In zo'n geval noemen we de bladeren ook wel externe knopen.
Een pad van de wortel naar een blad correspondeert met een executie van het algoritme op een of andere invoer. Elke executie correspondeert met precies één pad, maar een pad kan door meerdere (≥ 1) executies gevolgd worden. De hoogte van de boom stelt hier precies het worst case aantal vergelijkingen voor dat het algoritme doet.
Dit soort beslissingsbomen kan worden gebruikt voor de beschrijving van allerlei algoritmen die iets doen met een array A, en waarbij de basisoperatie het doen van vergelijkingen tussen array-elementen is. Dus niet alleen het opsporen van het grootste element in een rij waarden (selectie), maar bijvoorbeeld ook het sorteren van zo'n rij.

Stelling: Elk algoritme dat de grootste (resp. de kleinste) bepaalt uit een array met n elementen, en dat alleen gebaseerd is op het doen van arrayvergelijkingen, doet ten minste $\lceil \lg n \rceil$ vergelijkingen in de worst case.

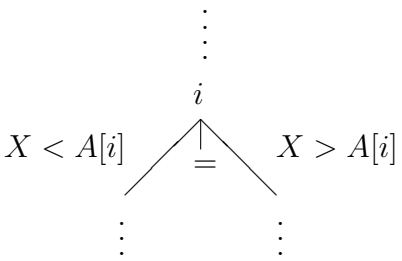
Bewijs:
Merk op dat in de stelling niet geëist wordt dat de array-elementen verschillend zijn. Het is echter voldoende de ondergrens van $\lceil \lg n \rceil$ vergelijkingen te bewijzen voor een array bestaande uit allemaal verschilende waarden. Immers het aantal vergelijkingen in het ergste geval is altijd minstens zo groot als het aantal vergelijkingen in een bijzonder geval. In dit voorbeeld (grootste opzoeken) bevat een blad als eindantwoord de index van het grootste array-element. Er zijn hier n mogelijke antwoorden (want elk van de n verschillende elementen kan het maximum zijn), die allemaal moeten kunnen voorkomen aangezien het algoritme voor elk invoerrijtje werkt. Dus $b \geq n$. Uit de stelling over binaire bomen ($h \geq \lceil \lg b \rceil$) volgt dan dat $h \geq \lceil \lg n \rceil$. Deze afschatting geldt voor elk van dit soort bomen,

en dus voor elk selectie-algoritme gebaseerd op arrayvergelijkingen.

1. Merk op dat dit een niet al te scherpe ondergrens oplevert voor de worst case complexiteit van dit probleem. We wisten al dat er altijd minstens $n-1$ arrayvergelijkingen nodig zijn in het ergste geval. Díe ondergrens is wel scherp; er bestaat immers een algoritme dat $n-1$ vergelijkingen doet.
2. Bovenstaand argument gaat goed als we te maken hebben met algoritmen die op alle mogelijke arrays werken. Het argument levert echter geen ondergrens van $\lceil \lg n \rceil$ op als het gaat om algoritmen die speciaal/alleen op gesorteerde arrays werken. Dan is namelijk het aantal mogelijke eindantwoorden dat kan voorkomen geen n, maar 1.

3.3 Zoeken

Gegeven een array A met daarin n verschillende elementen, en een waarde X die gezocht moet worden. Elk zoekalgoritme (gebaseerd op sleutelvergelijkingen) correspondeert dan met een binaire boom met in de interne knopen de twee-weg vergelijkingen $X =, < A[i]$ (genoteerd met de bekeken index i) die het algoritme doet:



Als $X = A[i]$ stopt het algoritme, want dan is X gevonden. Anders gaat het algoritme verder, afhankelijk van de uitslag van de vergelijking $X < A[i]$. Het label van het linkerkind geeft steeds precies de index aan van de entry waarmee X vervolgens wordt vergeleken als $X < A[i]$. Het label van het rechterkind is de entry waarmee X vervolgens wordt vergeleken als blijkt dat $X > A[i]$. Als een knoop geen linkerkind heeft betekent dit dat het algoritme stopt nadat het X met $A[i]$ heeft vergeleken en het heeft ontdekt dat $X < A[i]$. In dat geval is dan bekend dat X niet in het array voorkomt. Een executie van het algoritme waarmee de boom correspondeert eindigt dus in een willekeurige knoop, en alle knopen zijn in dit model van hetzelfde type. Het maximale aantal vergelijkingen dat het algoritme doet is hier gelijk aan de hoogte van de boom plus 1.

Stelling: Elk algoritme dat X opspoort in een array met n elementen, en dat alleen gebaseerd is op het doen van sleutelvergelijkingen, doet ten minste $\lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil$ vergelijkingen in de worst case.

Bewijs:
Het is weer voldoende de ondergrens te bewijzen voor een array bestaande uit allemaal verschillende waarden.
Merk op dat elke $A[i]$ gevonden moet kunnen worden, want het algoritme moet voor elke mogelijke keuze van X (en A) het juiste antwoord opleveren. Dit betekent voor elk van dit soort beslissingsbomen dat deze ten minste n knopen bevat. Immers elke i moet als knoop in elk geval een keer voorkomen. Dus: $h \geq \lfloor \lg n \rfloor$. Voor het aantal vergelijkingen in

de worst case ($= h + 1$) geldt dan dat dit $\geq \lfloor \lg n \rfloor + 1$ is. Ook hier geldt dit weer voor elk van dit soort bomen en dus voor elk zoekalgoritme gebaseerd op sleutelvergelijkingen.

Merk op dat het bovengegeven argument ook een ondergrens van $\lfloor \lg n \rfloor + 1$ oplevert voor algoritmen die alleen op gesorteerde arrays werken, zoals binair zoeken. Immers ook voor gesorteerde arrays is het zo dat de te vinden waarde X op elke mogelijke positie kan staan.

4 Ongeordend en geordend zoeken

Probleem: Gegeven een array A bestaande uit n elementen $A[1], \ldots, A[n]$, en een waarde X. Zoek deze X in het array en geef de index terug van de plek waar hij wordt aangetroffen (-1 indien niet aanwezig).

4.1 Ongeordend lineair zoeken

Algoritme: Lineair zoeken, zie opgave 3.
Een goede basisoperatie is hier het vergelijken van X met array-elementen (ook wel sleutelvergelijkingen genoemd).

Worst case: n (ga zelf na voor welke invoer dat voorkomt)

Average case (onder zekere aannames, zie opgave 3):
$q * \frac{1}{2} * (n + 1) + (1 - q) * n$, met q de kans dat X in A voorkomt.

Optimaliteit: elk algoritme dat X zoekt in een ongeordend array, en dat gebaseerd is op het doen van vergelijkingen tussen X en array-elementen doet in de worst case ten minste n van zulke vergelijkingen (opgave 3.e.).

4.2 Geordend lineair zoeken

We veronderstellen nu dat A oplopend gesorteerd is. We kunnen dan een aangepaste versie van lineair zoeken gebruiken, waarin gestopt wordt als blijkt dat X kleiner is dan een zeker array-element: er hoeft dan niet verder gezocht te worden.

Algoritme: Lineair zoeken, zie opgave 4.

Worst case: n (ga zelf na voor welke mogelijke invoer dat voorkomt; dit zijn er minder dan in het ongeordende geval)

Average case (onder zekere aannames, zie opgave 4):
$\frac{n}{2} + \frac{n}{n+1} + q * (\frac{1}{2} - \frac{n}{n+1}) \in \Theta(\frac{n}{2})$, met q de kans dat X in A voorkomt.

4.3 Jump search

We veronderstellen weer dat A oplopend gesorteerd is. Onderstaand algoritme is in orde van grootte beter (in de worst case) dan lineair zoeken. De k uit het algoritme is een geheel

getal met $1 \leq k \leq n$.

index := k;
// vergelijk X met $A[k], A[2k], A[3k], \ldots$
while index $\leq n$ **and** A[index] $< X$ **do**
    index := index + k;
od
if index $\leq n$
    // A[index $- k$] $< X \leq A$[index]
    lineair zoeken van X in A[index $- k + 1] \ldots A$[index];
else
    // A[index $- k$] $< X \leq A[n]$
    lineair zoeken van X in A[index $- k + 1] \ldots A[n]$;
fi

Worst case: $\lfloor \frac{n}{k} \rfloor + k$ vergelijkingen van de vorm A[index] $< X$. De beste keus voor k is $\lceil \sqrt{n} \rceil$. Dan doet Jump sort in het slechtste geval $\Theta(\sqrt{n})$ sleutelvergelijkingen. Dit is een duidelijke verbetering t.o.v. lineair geordend zoeken.

4.4 Binair zoeken

In een gesorteerd array kunnen we binair zoeken: vergelijk X met het middelste array-element. Als X niet gelijk is aan die array-entry, dan hoeven we slechts in één van beide helften verder te zoeken.

Algoritme: Binair zoeken, zie opgave 5.
We tellen de achtereenvolgende tests $X = A$[Midden] en $X < A$[Midden] als 1 test.

Worst case: $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$ vergelijkingen van de vorm $X =, < A$[Midden]

Average case (voor $n = 2^k - 1$): Het gemiddeld aantal vergelijkingen dat nodig is om een X te vinden die zeker in A zit, onder de aanname dat elke positie even waarschijnlijk is, bedraagt: $\frac{1}{n}\Sigma_{i=0}^{k-1}(i+1)2^i$.
Als X niet aanwezig is kost het altijd k vergelijkingen om dat te constateren.
Met q de kans dat X in A voorkomt is het gemiddeld aantal vergelijkingen dus:
$\frac{q}{n}\Sigma_{i=0}^{k-1}(i+1)2^i + (1-q)k \in \Theta(\lg(n+1))$.

Vraag: wat is de worst case *complexiteit* van binair zoeken in een enkel- of dubbelverbonden lijst? Uit dit voorbeeld blijkt dat de gekozen datastructuur ook invloed kan hebben op de complexiteit van je algoritme. Immers, het aantal vergelijkingen is weliswaar nog steeds $\lfloor \lg n \rfloor + 1$, maar je moet nu telkens delen van de lijst doorlopen om de middelste te bereiken. Het doen van vergelijkingen is hier dan ook geen goede maat voor de hoeveelheid werk.

5 Adversary-argument

Een ondergrens voor de worst case complexiteit van een probleem kan behalve met behulp van beslissingsbomen ook via een adversary-argument worden bewezen. We geven hier

een idee van wat zo'n adversary-argument inhoudt en hoe het werkt, en ter illustratie een eenvoudig voorbeeld. In de volgende paragraaf passen we een iets ingewikkelder adversary-argument toe op het probleem van het simultaan vinden van de grootste en de kleinste.

5.1 Introductie

Een algoritme speelt een vraag-en-antwoord-spel tegen een *adversary* (tegenstander). De bedoeling is om een zeker probleem op te lossen.

- het algoritme wil zo veel mogelijk informatie krijgen om met zo weing mogelijk vragen het probleem op te lossen
- de adversary wil zo weinig mogelijk informatie prijsgeven om ervoor te zorgen dat het algoritme zo veel mogelijk vragen moet stellen om de/een oplossing te vinden
- belangrijkste spelregel is consistentie: de adversary mag altijd alleen antwoorden geven die consistent (= niet in tegenspraak) zijn met eerder gegeven antwoorden
- de adversary beantwoordt de vragen van het algoritme volgens een of andere *adversary-strategie*
- deze strategie wil het algoritme dwingen zo veel mogelijk vragen te stellen
- de adversary bouwt aldus tijdens de uitvoering van een algoritme als het ware een *bad-case invoer* op
- het is niet nodig te weten hoe zo'n bad case er uitziet, alleen dat hij bestaat
- de adversary(strategie) zorgt ervoor dat hij op elk moment een invoer kan geven die consistent is met de op de gestelde vragen gegeven antwoorden

Het aantal stappen (=vragen) dat een *willekeurig* algoritme *ten minste* tegen de adversary(strategie) moet uitvoeren om het juiste antwoord te krijgen geeft een *ondergrens* op de worst case complexiteit van het probleem.

5.2 Voorbeeld: sorteren

Stelling
Elk algoritme gebaseerd op het doen van arrayvergelijkingen dat een array van n elementen (oplopend) sorteert, doet in de worst case ten minste $\lceil \lg n! \rceil = \Theta(n \lg n)$ vergelijkingen.

Bewijs: We bewijzen een ondergrens voor de worst case. We mogen daarom aannemen dat alle array-elementen (bijv. getallen) verschillend zijn.
Sorteren betekent dat je de onderlinge ordening/volgorde wil vinden van de array-elementen. Je bent dan in feite op zoek naar de ordening die de array-elementen in oplopende volgorde oplevert. Er zijn $n!$ verschillende ordeningen mogelijk, namelijk: $A[1] < A[2] < A[3] < \ldots < A[n]$, $A[1] < A[3] < A[2] < \ldots < A[n]$, $A[2] < A[3] < A[1] < \ldots < A[n]$, $A[2] < A[1] < A[3] < \ldots < A[n]$, $A[3] < A[1] < A[2] < \ldots < A[n]$, $A[3] < A[2] < A[1] < \ldots < A[n]$, etcetera. De adversary heeft in het begin alle $n!$ mogelijke volgordes in zijn hand (de

kandidaatoplossingen). Na elk door hem gegeven antwoord op een vraag (dat is hier een arrayvergelijking) van het algoritme gooit hij de rijtjes die niet consistent zijn met dat antwoord weg. Zo heeft hij altijd de rijtjes in zijn hand die kloppen met de antwoorden op de door het algoritme gestelde vragen. Het algoritme stelt vragen van de vorm $A[i] < A[j]$. De strategie van de adversary is als volgt: geef steeds het antwoord dat zo veel mogelijk mogelijkheden over laat. Indien er evenveel rijtjes zijn die aan een ja-antwoord voldoen als rijtjes die aan een nee-antwoord voldoen, zeg dan ja. Merk op dat het algoritme de sortering pas echt zeker kan weten als de adversary nog maar één rijtje in zijn hand heeft. Als deze strategie gebruikt wordt, wordt het algoritme dus gedwongen om ten minste $\lceil \lg n! \rceil$ vergelijkingen te doen. (Immers op zijn best voor het algoritme wordt in elke stap het aantal kandidaatoplossingen in de hand van de adversary gehalveerd.) Om de sortering die uiteindelijk in de hand van de adversary overgebleven is te vinden, zijn dus minstens $\lceil \lg n! \rceil$ vragen (= arrayvergelijkingen) nodig geweest. Dit geldt voor elk sorteeralgoritme dat tegen deze adversary speelt: er is dus altijd een invoerrijtje te vinden (namelijk een die correspondeert met de ordening van het laatst overgebleven rijtje) waarop het algoritme minstens $\lceil \lg n! \rceil$ vergelijkingen doet. Kortom: in het ergste geval heeft *elk* sorteeralgoritme ten minste $\lceil \lg n! \rceil$ vergelijkingen nodig in de worst case.

6 Selectie

6.1 Het selectieprobleem

Probleem: Gegeven n verschillende getallen (bijvoorbeeld), opgeslagen in een array A: $A[1], A[2], \ldots, A[n]$. Laat verder een geheel getal k met $1 \leq k \leq n$ gegeven zijn. Gevraagd de $A[i]$ die groter is dan precies $k-1$ andere $A[j]$'s. M.a.w.: we zoeken de k-de in grootte.

Klasse van algoritmen: We bekijken algoritmen die uitsluitend gebaseerd zijn op het doen van arrayvergelijkingen.

Ondergrens: Elk algoritme voor het selectieprobleem doet in de worst case *ten minste* $\lceil \lg n \rceil$ vergelijkingen. Te bewijzen met een beslissingsboomargument.

Bovengrens: Het probleem kan worden opgelost door het array eerst oplopend te sorteren. De k-de in grootte is dan $A[k]$. Sorteren kan met $\Theta(n \lg n)$ vergelijkingen, dus selectie is *hooguit* in $O(n \lg n)$.

Opmerking. Zowel bovengrens als ondergrens kunnen scherper. We bekijken in het vervolg bovendien steeds het (precieze) aantal vergelijkingen in de worst case.

6.2 Speciale gevallen

In de volgende paragrafen zullen we een aantal bijzondere gevallen van het selectieprobleem tegenkomen. Het gaat hier om:

1. Het geval $k = 1$: het *minimum*, of $k = n$: het *maximum*. Het kan met $n - 1$ vergelijkingen. Dit is ook optimaal (al gezien).

2. (variant) Het *maximum en minimum* beide opsporen. Voor de hand ligt een algoritme met $2n-3$ vergelijkingen, maar het kan met $\lceil \frac{3n}{2} \rceil - 2$. Dit is optimaal (*adversary-argument*, zie verderop).

3. Het geval $k = n-1$: de *op een na grootste*. Voor de hand ligt een algoritme met $2n-3$ vergelijkingen, maar met behulp van de *toernooimethode* kan het met $n + \lceil \lg n \rceil - 2$. Dit is optimaal (kan ook met een adversary-argument bewezen worden).

4. Het geval $k = \lceil \frac{n}{2} \rceil$: de *mediaan* (= de middelste in grootte).

   Zie hiervoor ook de opgaven. Er zit een gat tussen de best bekende ondergrens en het best bekende algoritme. We kunnen dus niets over de optimaliteit van dat algoritme zeggen.

6.3 Maximum & minimum: algoritme

Hieronder een algoritme voor het vinden van zowel maximum als minimum. Dit algoritme doet $\lceil \frac{3n}{2} \rceil - 2$ arrayvergelijkingen en is optimaal. Het algoritme werkt voor even n. Voor n oneven is slechts een heel kleine aanpassing nodig.

```
(1)   if A[1] > A[2] then // n ≥ 2
(2)         grootste := A[1]; kleinste := A[2];
(3)   else
(4)         grootste := A[2]; kleinste := A[1];
(5)   i := 3;
(6)   while i < n do
(7)         if A[i] > A[i + 1] then
(8)               gr := A[i]; kl := A[i + 1];
(9)         else
(10)              gr := A[i + 1]; kl := A[i];
(11)        fi
(12)        if gr > grootste then
(13)              grootste := gr;
(14)        fi
(15)        if kl < kleinste then
(16)              kleinste := kl;
(17)        fi
(18)        i := i + 2;
(19)  od
```

6.4 Op een na grootste: toernooimethode

Gebruikte terminologie : wedstrijd $\longleftrightarrow$ vergelijking; winnaar $\longleftrightarrow$ grootste van de twee; speler $\longleftrightarrow$ array-element; etcetera.

De *toernooimethode* (met voor het gemak $n = 2^k$ een tweemacht):

Laat de spelers twee aan twee tegen elkaar spelen ($\frac{n}{2}$ wedstrijden).

Laat de $\frac{n}{2}$ winnaars weer twee aan twee tegen elkaar spelen; de $\frac{n}{4}$ winnaars daarvan weer, etcetera.

Herhaal dit totdat je één speler overhoudt: dit is de eindwinnaar, dus de grootste van allemaal.

Er zijn nu $n-1$ wedstrijden gespeeld, en er waren k rondes nodig ($k = \lg n$).

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. (Bewijs op college of bedenk dit zelf.)

Het kost nog $k-1$ vergelijkingen om deze te bepalen.

Het totaal aantal vergelijkingen is dus $n + \lg n - 2$.

Als n geen tweemacht is, is een kleine aanpassing van het algoritme nodig. Aantal vergelijkingen: $n + \lceil \lg n \rceil - 2$.

Implementatie: met een heapachtige structuur; zie de opgaven.

6.5 Ondergrens voor opsporen maximum & minimum

Stelling: Elk algoritme gebaseerd op arrayvergelijkingen dat het minimum en het maximum van n (verschillende) waarden vindt, doet in het slechtste geval *ten minste* $\lceil \frac{3n}{2} \rceil - 2$ vergelijkingen.

Bewijs: We onderscheiden W (≥ 1 keer gewonnen, nooit verloren), V (≥ 1 keer verloren, nooit gewonnen), WV (≥ 1 keer gewonnen en ≥ 1 keer verloren) en N (nog nooit "gespeeld"). De strategie van de adversary is als volgt.

wedstrijd x-y	uitslag	N	W	V	WV	type
N-N	x > y	-2	$+1$	$+1$	$--$	1
V-N	x < y	-1	$+1$	$--$	$--$	1
W-N	x > y	-1	$--$	$+1$	$--$	1
W-W	consistent	$--$	-1	$--$	$+1$	2
V-V	consistent	$--$	$--$	-1	$+1$	2
W-V	x > y	$--$	$--$	$--$	$--$	
WV-WV	consistent	$--$	$--$	$--$	$--$	
WV-N	x > y	-1	$--$	$+1$	$--$	1
WV-W	x < y	$--$	$--$	$--$	$--$	
WV-V	x > y	$--$	$--$	$--$	$--$	

En de stand van zaken aan begin en eind:

	N	W	V	WV
aantal begin	n	0	0	0
aantal eind	0	1	1	$n-2$

In de tabel geeft de eerste kolom het soort wedstrijd (= arrayvergelijking) aan. Bijvoorbeeld WV - W geeft aan dat het eerste array-element van de vergelijking zowel gewonnen als verloren heeft (dus vergeleken is en ten minste 1x groter was en ten minste 1x kleiner dan een ander array-element), en het tweede minstens 1 keer gewonnen heeft en nog nooit verloren. Het antwoord van de adversary (die probeert altijd zo weinig mogelijk informatie prijs te geven waardoor het algoritme hopelijk zo veel mogelijk vergelijkingen moet doen) staat in de tweede kolom. De laatste kolommen geven het effect aan van de antwoorden van de adversary. De adversary gebruikt deze strategie tegen elk algoritme, en bouwt zo tegen elk algoritme een bad case op. Hij kiest als het ware de waarde van array-elementen zo dat er voldaan wordt aan de uitslagen zoals de strategie voorschrijft. Hierbij mag de waarde van een element van type W, die alle eerdere vergelijkingen dus gewonnen heeft, altijd ongestraft groter worden gemaakt; de uitslagen van de eerdere vergelijkingen blijven daarmee hetzelfde. Analoog voor elementen van type V: de waarde daarvan mag altijd kleiner worden gekozen. Het algoritme is klaar als er precies $n-2$ elementen van type VW zijn, 1 van type W en 1 van type V. Het aantal onbekeken elementen moet uiteraard nul worden, want anders zou het onbeken element wellicht het maximum of het minimum kunnen zijn.
Om van beginsituatie naar eindsituatie te komen *moeten* in elk geval (dus minstens) $n-2$ vergelijkingen van type 2 gedaan worden (de enige manier om $n-2$ WV's te krijgen) en $\lceil \frac{n}{2} \rceil$ van type 1 (om het aantal N's op 0 te krijgen). Aangezien vergelijkingen van type 1 niets doen met het aantal WV's en die van type 2 niets met het aantal N's zijn er in totaal (je mag ze dus optellen) ten minste $n-2+\lceil \frac{n}{2} \rceil$ vergelijkingen nodig tegen deze strategie.

6.6 Selectie is O(n)

We bekijken weer het algemene probleem. Gezocht: het k-de element in grootte uit n verschillende elementen (getallen bijv.). Onderstaand algoritme is een simplificatie van het originele algoritme van Blum, Floyd, Pratt, Rivest en Tarjan[3]

Algoritme:

1. Verdeel de getallen in $\lfloor \frac{n}{5} \rfloor$ groepjes van 5 elementen, en 1 groepje met de resterende n mod 5
2. Vind de mediaan van elk van de $\lceil \frac{n}{5} \rceil$ groepjes van 5 (of minder), bijvoorbeeld met behulp van Bubblesort
3. Vind de mediaan x van de in stap 2 gevonden $\lceil \frac{n}{5} \rceil$ medianen: recursie
4. Partitioneer (ongeveer zoals bij Quicksort) alle elementen rond x. Stel dat m getallen $\leq x$ zijn en $n-m$ getallen $> x$.

$< x$	x	$> x$

[3]Time Bounds for Selection, 1973

5. Vind de k-de in grootte uit m stuks als $k \leq m$, of de $(k-m)$-de uit $n-m$ als $k > m$: recursie

Merk op: ten minste $3 * \left(\left\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6$ elementen zijn groter (kleiner) dan x. Er zijn dus hooguit $\frac{7n}{10} + 6$ elementen $\leq x$ ($\geq x$). De term O(n) hieronder komt van stap 1, 2 en 4 samen. Laat $T(n)$ = aantal vergelijkingen in de worst case, dan geldt dus:

$$T(n) \leq \begin{cases} const & n \leq ... \\ T(\lceil \frac{n}{5} \rceil) + T(\lceil \frac{7n}{10} \rceil + 6) + O(n) & n > ... \end{cases}$$

Hieruit vinden we, bijvoorbeeld met behulp van inductie, dat $T(n) \leq cn$ voor geschikte keuze van c, en voor alle $n > ....$
Conclusie: selectie is $O(n)$.

Opmerking. Het is essentieel dat $\lceil \frac{n}{5} \rceil + \lceil \frac{7n}{10} \rceil + 6 < n$. Als dat niet het geval is, wordt $O(n)$ niet gehaald. De moeite van het zoeken van een goede pivot x wordt dus beloond. Als we bijvoorbeeld in bovenstaand algoritme in plaats van groepjes van 5 groepjes van 3 zouden gebruiken wordt de $O(n)$ niet gehaald. Als we, aan de andere kant, groepjes van 7 kiezen is het algoritme wel in $O(n)$, maar is het vanwege grotere constanten minder efficiënt. De keuze voor 5 is dus een optimale keuze.

7 Sorteren

Probleem: Gegeven een rij (array) A met n elementen $A[1], \ldots, A[n]$. Sorteer A oplopend (dus $A[i] \leq A[i+1]$ voor alle i; $<$ als alle $A[i]$ verschillend zijn).

7.1 Insertion sort

Deze sorteermethode is -net al vele andere- gebaseerd op het doen van arrayvergelijkingen: herhaald vergelijken en verwisselen (indien nodig). Om de complexiteit te bepalen tellen we het aantal vergelijkingen van de vorm $A[j] > x$. Dit is een goede maat voor de complexiteit. In elke ronde zijn de eerste $i-1$ elementen reeds gesorteerd en gaan we $A[i]$ op de juiste plek in $A[1]$ t/m $A[i-1]$ invoegen. Insertion Sort is een *in situ* sorteeralgoritme: alles gebeurt via interne verwisselingen. Het algoritme:

```
(1)  for i := 2 to n do
         // nu A[i] op de juiste plek in A[1] ... A[i-1] invoegen
(2)      x := A[i];
(3)      j := i - 1;
(4)      while j > 0 and A[j] > x do
(5)          A[j+1] := A[j];
(6)          j := j - 1;
(7)      od
(8)      A[j+1] := x;
(9)  od
```

Het aantal vergelijkingen dat het algoritme doet is (zie college):

1. Worst case: $W(n) = \sum_{i=2}^{n}(i-1) = \frac{1}{2}n(n-1)$
2. Best case: $B(n) = \sum_{i=2}^{n} 1 = n-1$
3. Average case: $A(n) = \frac{1}{4}n(n-1) + n - \sum_{i=1}^{n} \frac{1}{i} \in \Theta(n^2)$

Bij average case nemen we aan dat alle $A[i]$'s verschillend zijn en dat alle $n!$ permutaties van $A[1]$ t/m $A[n]$ even waarschijnlijk zijn. We middelen dan over alle mogelijke permutaties, dat zijn dus in essentie alle mogelijke invoerrijtjes.

Definitie: een *inversie* van de permutatie $A[1], A[2], \ldots, A[n]$ is een paar $(A[i], A[j])$ waarvoor $i < j$ en $A[i] > A[j]$. Met andere woorden: een inversie is een paar $(A[i], A[j])$ dat verkeerd om staat.

Merk op: elk sorteeralgoritme moet *alle* aanwezige inversies opheffen. En verder: als een sorteeralgoritme altijd hooguit één inversie opheft per arrayvergelijking, dan is het aantal vergelijkingen dat wordt gedaan om $A[1], \ldots, A[n]$ te sorteren *ten minste* het aantal inversies van A. Bovendien: een *buursverwisseling* (zoals bij Insertion sort) heft altijd precies één inversie op.

Bewering: Het maximale aantal inversies dat kan voorkomen in een rijtje van n verschillende waarden is $\binom{n}{2} = \frac{1}{2}n(n-1)$

Gevolg: Elk sorteeralgoritme (gebaseerd op arrayvergelijkingen) dat hooguit één inversie opheft per vergelijking doet in de *worst case ten minste* $\frac{1}{2}n(n-1)$ vergelijkingen: $\Omega(n^2)$
Dus: Insertion sort is *optimaal* voor wat betreft de worst case, binnen de klasse van algoritmen gebaseerd op het doen van arrayvergelijkingen.

Merk op: als we een beter sorteeralgoritme gebaseerd op arrayvergelijkingen willen hebben, dan moet dat algoritme in elk geval geen buursverwisselingen doen. Mogelijk leveren algorimen die elementen die verder van elkaar liggen een ordeverbetering op. Dit blijkt inderdaad het geval te zijn.

Bewering: Het *gemiddeld* aantal inversies in een permutatie van n verschillende waarden (bijvoorbeeld de getallen 1 t/m n) is $\frac{1}{4}n(n-1)$. Dit onder de aanname dat alle permutaties even waarschijnlijk zijn.
Hint bij het bewijs: Bekijk alle $n!$ mogelijke permutaties A_i en hun getransponeerden A_i^t. Gebruik verder: als twee waarden verkeerd om staan in A_i, staan ze juist goed in A_i^t en omgekeerd. En: elke inversie komt steeds óf in A_i óf in A_i^t voor (voor elke i).

Gevolg: Elk algoritme dat sorteert met behulp van arrayvergelijkingen en dat per vergelijking ten hoogste één inversie opheft, moet *ten minste* $\frac{1}{4}n(n-1)$ vergelijkingen doen in de *average case*
Dus: Insertion sort moet ten minste $\frac{1}{4}n(n-1)$ vergelijkingen doen in het gemiddelde geval en is derhalve in orde van grootte optimaal binnen de bekeken klasse van algoritmen, namelijk $\Theta(n^2)$.

7.2 Mergesort

Dit is een recursief sorteeralgoritme dat gebruik maakt van extra geheugenruimte in de vorm van een hulparray. De functie Merge voegt twee (hier ongeveer even grote) gesorteerde (deel)arrays samen tot één gesorteerd stuk. De aanroep voor het sorteren van $A[1]$ t/m $A[n]$ wordt MergeSort$(A, 1, n)$.

MergeSort(A, p, r)::
// sorteert $A[p], \ldots, A[r]$

```
if p < r then
    q := ⌊(p+r)/2⌋;
    MergeSort(A, p, q);
    MergeSort(A, q + 1, r);
    Merge(A, p, q, r);
fi
```

Merge(A, p, q, r)::
// voegt reeds gesorteerde deelrijtjes $A[p], \ldots, A[q]$ en $A[q+1], \ldots, A[r]$ samen tot een
// gesorteerd stuk $A[p], \ldots, A[r]$

```
i := p; j := q + 1; k := p;
while i ≤ q and j ≤ r do
    if A[i] < A[j] then
        hulp[k] := A[i]; i := i + 1; k := k + 1;
    else
        hulp[k] := A[j]; j := j + 1; k := k + 1;
    fi
od
if i > q then // eerste helft is op
    kopieer A[j], ..., A[r] naar hulp;
else // tweede helft is op
    kopieer A[i], ..., A[q] naar hulp;
fi
kopieer hulp[p], ..., hulp[r] terug naar A;
// hulp is een hulparray ter grootte n (net als A)
```

De complexiteit:

Voor het bepalen van de complexiteit van MergeSort tellen we het aantal vergelijkingen tussen array-elementen. Ga na dat dit een goede basisoperatie is.

We bekijken eerst de complexiteit van Merge.
Stel dat we met behulp van Merge twee gesorteerde rijtjes van resp. k en m elementen (met $k + m = n$) samenvoegen tot één gesorteerde rij. Dan geldt:

1. Het aantal vergelijkingen in de *worst case* is $n - 1$
2. Het aantal vergelijkingen in de *best case* is minimum(k, m)

Stelling over Merge: Elk algoritme, gebaseerd op arrayvergelijkingen, dat twee gesorteerde arrays (rijen) van lengte m samenvoegt tot één gesorteerd array, doet in het *slechtste geval ten minste* $2m-1$ van zulke vergelijkingen ($= n-1$ voor $m = \frac{n}{2}$, n even). Voor twee arrays ter lengte $m-1$ resp. m is dat *ten minste* $2m-2$ ($= n-1$ voor $m = \lceil \frac{n}{2} \rceil$, n oneven).
Bewijs: zie college.

Gevolg: Binnen de klasse van samenvoegalgoritmen gebaseerd op arrayvergelijkingen is het beschreven Merge-algoritme optimaal.

Voor de complexiteit van MergeSort hebben we nu het volgende.

1. Zij $T(n)$ = aantal vergelijkingen in de *worst case*, met $n = 2^k$. Dan geldt:

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

   De oplossing van deze recurrente betrekking is: $T(n) = n \lg n - n + 1 \in \Theta(n \lg n)$.

   Opmerking: als n geen tweemacht is, wordt de recurrente betrekking:

   $$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1 & n > 1 \end{cases}$$

   Dan geldt eveneens: $T(n) \in \Theta(n \lg n)$. (Zelfs: $T(n) = n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$.)

   MergeSort is dus (in orde van grootte) optimaal voor wat betreft de worst case. Zie namelijk de eerste stelling in de volgende paragraaf voor een ondergrens op het sorteren van een rij van n elementen. Er is echter extra geheugenruimte (hier $\Theta(n)$) nodig.

2. Zij $B(n)$ = aantal vergelijkingen in de *best case*, met $n = 2^k$. Dan geldt:

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

   Oplossing: $B(n) = \frac{n}{2} \lg n \in \Theta(n \lg n)$.

3. Uit bovenstaande volgt dat ook het aantal vergelijkingen in de *average case* gelijk is aan $\Theta(n \lg n)$.

7.3 Theoretische ondergrens voor sorteren

We bekijken sorteeralgoritmen gebaseerd op het doen van vergelijkingen van de vorm $A[i] < A[j]$. We mogen verder aannemen dat A allemaal *verschillende* waarden bevat en dat het algoritme stopt zodra de sortering gevonden is. Zo'n type algoritme correspondeert met een *beslissingsboom* die de achtereenvolgende vergelijkingen die het algoritme voor elke mogelijke invoer uitvoert representeert (als paden vanaf de wortel naar een blad). Elk pad van de wortel tot een blad correspondeert met een executie van het algoritme. *Interne knopen* corresponderen met de *vergelijkingen*, de *bladeren*/externe knopen met de

mogelijke *sorteringen*. Merk op dat het aantal bladeren dus ten minste $n!$ bedraagt, met n het aantal te sorteren waarden. (Overigens geldt zelfs dat het aantal bladeren gelijk is aan $n!$, aangezien er ook precies $n!$ verschillende te onderscheiden soorten invoer mogelijk zijn.)

Stelling: Het aantal vergelijkingen in de *worst case* is voor elk algoritme dat sorteert middels arrayvergelijkingen *ten minste* $\lceil \lg n! \rceil$ (dus $\Omega(n \lg n)$).

Bewijs: worst case aantal vergelijkingen (algemeen) $\geq$ worst case aantal vergelijkingen (verschillende waarden) = hoogte beslissingsboom $\geq \lceil \lg n! \rceil$.

Stelling: Het aantal vergelijkingen in de *average case* is voor elk algoritme dat sorteert middels arrayvergelijkingen $\Omega(n \lg n)$. Dit onder de aanname dat alle $n!$ volgordes even waarschijnlijk zijn.
Het bewijs maakt gebruik van onderstaand lemma/gevolg over binaire bomen.

Gegeven een binaire boom $\mathcal{B}$ met b bladeren.
Definitie: De *externe padlengte* E van $\mathcal{B}$ is de som van de lengtes van alle paden van de wortel naar een blad:

$$E = \Sigma_{\text{bladeren}} \text{lengte pad wortel naar blad}$$

Lemma: Zij E de externe padlengte van $\mathcal{B}$. Dan geldt: $E \geq b(\lg b - 1)$.
Gevolg: De gemiddelde lengte van een pad van de wortel naar een blad $= \frac{E}{b} \geq \lg b - 1$.

Voor het precieze bewijs van bovenstaande stelling over het gemiddeld aantal vergelijkingen: zie college.

Opmerking: voor de externe padlengte geldt zelfs het volgende.
Lemma: Zij E de externe padlengte van $\mathcal{B}$. Dan geldt: $E \geq b \lg b$.
Extra opgave: bewijs dit lemma. Hint: het bewijs gaat met behulp van volledige inductie.

7.4 Quicksort

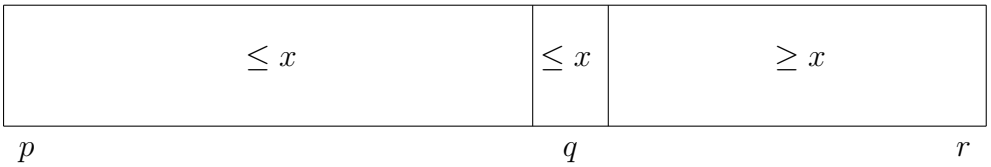
De sorteermethode Quicksort is net als Mergesort recursief, gebaseerd op het principe van verdeel en heers. Quicksort werkt in tegenstelling tot Mergesort met interne verwisselingen, en heeft geen extra geheugenruimte nodig dan die gerelateerd aan de recursie. De methode werd in 1962 bedacht door C.A.R. Hoare, en is in de praktijk een van de snelste. De aanroep voor het sorteren van $A[1]$ t/m $A[n]$ wordt Quicksort(A,1,n).

Het algoritme:

QuickSort(A, p, r)::
// sorteert $A[p], \ldots, A[r]$ oplopend

if $p < r$ **then**
    $q :=$ Partitie(A, p, r);
    QuickSort(A, p, q);
    QuickSort$(A, q + 1, r)$;
fi

De functie Partitie reorganiseert het array via interne verwisselingen tot de vorm zoals in onderstaand plaatje. Hierin is x een der array-elementen, bijvoorbeeld $A[p]$. Dit noemen we de spil (of pivot). De complexiteit van het algoritme hangt af van de keuze van deze spil.



Partitie (A, p, r)::
// reorganiseert het (deel)array $A[p], \ldots, A[r]$ zoals in bovenstaand plaatje

(*) // hier komt nog wat
$x = A[p]$; $i := p - 1$; $j := r + 1$;
while $i < j$ **do**
    $j := j - 1$; // loop met j naar links
    while $A[j] > x$ **do**
        $j := j - 1$;
    od // tot je een waarde $A[j] \leq x$ vindt
    i := i + 1; // loop met i naar rechts
    while $A[i] < x$ **do**
        $i := i + 1$;
    od // tot je een waarde $A[i] \geq x$ vindt
    if $i < j$ **then**
        wissel($A[i], A[j]$);
    fi // verwissel; $A[i]$ en $A[j]$ staan dan weer in het goede stuk
od
return j;

De *basisoperatie* is hier weer het vergelijken van array-elementen $A[j] > x$ en $A[i] < x$. Enige opmerkingen bij Partitie:

1. Partitie stopt met $i = j$ of $i = j + 1$
2. Na afloop is altijd $j \geq p$ en $j \leq r - 1$. Quicksort wordt dus op echt kleinere rijtjes recursief aangeroepen
3. Elk array-element wordt precies één keer met x vergeleken, behalve $A[q]$ en eventueel $A[q + 1]$
4. Partitie doet altijd $\Theta(m)$ vergelijkingen, met m het aantal elementen van het (deel)array $A[p], \ldots, A[r]$
5. Er worden elementen verwisseld die ver uit elkaar kunnen liggen, dit in tegenstelling tot bijvoorbeeld Insertion sort
6. Vraag: wat gebeurt er met gelijke array-elementen?

Complexiteit:

1. Een *bad case* voor Quicksort is het reeds gesorteerde rijtje bestaande uit n verschillende waardes, bijvoorbeeld $1, 2, \ldots, n$. Hierop doet Quicksort $\Sigma_{k=3}^{n+1} k = \frac{1}{2}n(n+3)-2 \in \Theta(n^2)$ vergelijkingen. Dit is in orde van grootte dus even slecht als Insertion sort.

2. Laat $W(n)$ het aantal vergelijkingen voorstellen dat Quicksort doet in de *worst case*. Dan voldoet W aan:

   $$W(n) = \begin{cases} \max_{1 \leq q \leq n-1}(W(q) + W(n-q)) + \Theta(n) & n > 1 \\ c & n = 1 \end{cases}$$

   voor zekere constante c. Dan geldt (zie college) dat $W(n) \leq dn^2$ voor zekere $d > 0$. Gecombineerd met de bad case hebben we dus het volgende resultaat.

3. Quicksort doet $\Theta(n^2)$ vergelijkingen in de worst case.

4. Een *good case* (voor n een tweemacht) wordt verkregen als het aantal vergelijkingen, B(n), voldoet aan:

   $$B(n) = \begin{cases} 2B(\frac{n}{2}) + \Theta(n) & n > 1 \\ 0 & n = 1 \end{cases}$$

   Dan is $B(n) \in \Theta(n \lg n)$. Dit is zelfs het beste geval voor Quicksort.

De keuze van de spil heeft grote invloed op de complexiteit. Standaard het eerste array-element ($A[p]$) als spil kiezen is een slechte keuze. Ter verbetering voegen we op plek (*) in Partitie toe: Kies een *slim* array-element en wissel dat met $A[p]$. Slim kan zijn: kies een *random* array-element. De kans dat x de i-de in grootte is is dan voor elke i even groot, namelijk $\frac{1}{n}$ (onder de aanname dat alle $A[i]$ verschillend zijn). De worst case blijft uiteraard $\Theta(n^2)$, maar hieronder zien we nu de reden waarom Quicksort heet zoals het heet.

Stelling: Quicksort doet $O(n \lg n)$ vergelijkingen in de *average case*.

Opmerking. Als we in Quicksort in elke stap de mediaan berekenen en die als spil gebruiken, dan is ook de worst case complexiteit $O(n \lg n)$. Echter de betrokken constanten (hoeveelheid overhead) zijn groot (zie 5.7), dus in de praktijk is dat niet handig.

7.5 Shellsort

Shellsort, genoemd naar zijn bedenker, Donald Shell, was een van de eerst gevonden sorteeralgoritmen met worst case complexiteit minder dan kwadratisch. Het werkt door in achtereenvolgende rondes array-elementen die wat verder van elkaar liggen te vergelijken. In het begin worden veel korte deelrijtjes gesorteerd, waarvan de elementen ver van elkaar liggen. De afstand tussen vergeleken elementen neemt af in elke ronde: er worden dan per ronde steeds minder rijtjes, maar wel langere gesorteerd. De rij wordt als het ware voorgesorteerd, totdat in de laatste ronde de rij als geheel wordt gesorteerd, waarbij buurelementen vergeleken worden. Als sorteermethode in elke ronde gebruiken we Insertion sort. Shellsort sorteert dus met behulp van vergelijk-verwissel (indien nodig) operaties (*compare-exchanges*).

Definitie: Een rij $A[1], A[2], \ldots, A[n]$ heet *k-gesorteerd* als geldt: $A[i] \leq A[i+k]$ voor elke $i = 1, 2, \ldots, n-k$.
Merk op: 1-gesorteerd = gesorteerd.

Voorbeeld: Het rijtje $5, 8, 24, 13, 7, 18, 31, 19, 44, 63, 82, 29$ is 4-gesorteerd en 6-gesorteerd.

Shellsort gebruikt een rijtje *stapgroottes* (increments) $h_t, h_{t-1}, \ldots, h_2, h_1 = 1$. De rij A met n elementen wordt gesorteerd door achtereenvolgens subrijen te sorteren van elementen die telkens op afstand h_i van elkaar liggen. Met andere woorden: A wordt h_i-gesorteerd voor $i = t, \ldots, 1$. Elk van de h_i subrijtjes heeft maximaal $\lceil \frac{n}{h_i} \rceil$ elementen. Omdat $h_1 = 1$ weet je zeker dat Shellsort correct sorteert.

Voorbeeld: Hier $n = 12$. Als incrementrijtje kiezen we 6, 4, 3, 2, 1.
We nemen als beginrijtje: 7, 19, 24, 13, 31, 8, 82, 18, 44, 63, 5, 29. Dan zijn de tussenresultaten van Shellsort als volgt:

Na 6-sorteren: 7, 18, 24, 13, 5, 8, 82, 19, 44, 63, 31, 29
Na 4-sorteren: 5, 8, 24, 13, 7, 18, 31, 19, 44, 63, 82, 29
Na 3-sorteren: 5, 7, 18, 13, 8, 24, 31, 19, 29, 63, 82, 44
Na 2-sorteren: 5, 7, 8, 13, 18, 19, 29, 24, 31, 44, 82, 63
Na 1-sorteren: 5, 7, 8, 13, 18, 19, 24, 29, 31, 44, 63, 82

Het algoritme:

```
(1)   h = n div 2; // ⌊n/2⌋ dus
(2)   while h > 0 do
(3)       for i := h + 1 to n do
(4)           temp := A[i]; j := i;
(5)           while j − h > 0 do
              // invoegen op de juiste plek
(6)               if temp < A[j − h] then
(7)                   A[j] := A[j − h]; // schuif
(8)                   j := j − h;
(9)               else
(10)                  "exit binnenste while";
(11)              fi
(12)          od
(13)          A[j] := temp; // zet neer
(14)      od
          // de rij is nu h-gesorteerd
(15)      h := h div 2; // oorspronkelijke keuze van Shell
(16)  od
```

Regel (4) t/m (13) is Insertion sort op deelarrays.

Een zeer belangrijke eigenschap van Shellsort is de volgende (zonder bewijs).
Stelling: Als een ℓ-gesorteerd array h-gesorteerd wordt met behulp van compare-exchanges (vergelijk en verwissel indien nodig), dan blijft het array ℓ-gesorteerd.

Het aantal arrayvergelijkingen is ook voor Shellsort een goede maat voor de complexiteit. De complexiteit van Shellsort hangt in hoge mate af van de gekozen serie stapgroottes. De analyse is in het algemeen extreem moeilijk en nog zeer incompleet. Voor de increments zoals door Shell zelf voorgesteld (gebruikt in het algoritme hierboven) kunnen we wel vrij eenvoudig de worst case complexiteit bepalen. Helaas is dit nu net geen goed voorbeeld van de efficiëntie van Shellsort. Er zijn veel betere keuzes voor de rij increments mogelijk.

1. Neem als stapgroottes $h_t = \lfloor \frac{n}{2} \rfloor$, en $h_i = \lfloor \frac{h_{i+1}}{2} \rfloor$ voor $i = t-1, \ldots, 1$. Voor het gemak nemen we $n = 2^k$, dus $t = \lfloor \lg n \rfloor = k$.

   Stelling A: Het aantal arrayvergelijkingen dat Shellsort met deze incrementserie doet is in de *worst case* $\Omega(n^2)$.
   Bewijs: We geven een *bad case*. Neem als invoer een rij met n verschillende getallen, met de $\frac{n}{2}$ grootste in oplopende volgorde op de even posities en de $\frac{n}{2}$ kleinste in oplopende volgorde op de oneven posities. In ronde 1 tot en met $t-1$ verandert de rij niet. In de laatste ronde moeten in elk geval de $\frac{n}{2}$ kleinste getallen naar voren worden gebracht. Dit alleen al kost $2+3+\ldots+\frac{n}{2} = (\Sigma_{i=1}^{\frac{n}{2}} i) - 1 = \frac{n^2}{8} + \frac{n}{4} - 1 \in \Theta(n^2)$ vergelijkingen. Zie verder het college.

   Stelling B: Het aantal vergelijkingen dat Shellsort met deze increments doet is in de *worst case* $O(n^2)$.
   Bewijs: Het h_i-sorteren bestaat uit h_i Insertion sorts op subrijtjes van elk $\frac{n}{h_i}$ elementen. Dat kost dus $h_i * O((\frac{n}{h_i})^2) \in O(\frac{n^2}{h_i})$ vergelijkingen. In totaal (over t rondes) is dat $O(\Sigma_{i=1}^{t} \frac{n^2}{h_i})$ vergelijkingen. Nu geldt in dit speciale geval voor het rijtje stapgroottes: $\Sigma_{i=1}^{t} \frac{1}{h_i} = 2 - \frac{2}{n} < 2$ (zie college). Ergo: het totaal aantal vergelijkingen is in $O(n^2)$.

   Gevolg van A en B: In de *worst case* doet Shellsort met Shell's increments $\Theta(n^2)$ vergelijkingen.

2. Neem als increments (Hibbard): $2^k - 1, 2^{k-1} - 1, \ldots, 7, 3, 1$ (met $k = \lfloor \lg n \rfloor$). Merk op dat opeenvolgende increments hier relatief priem zijn.

   Stelling (zonder bewijs): In de worst case doet Shellsort met Hibbard's increments $O(n^{\frac{3}{2}})$ vergelijkingen.

3. Het kan nog beter, maar de beste serie stapgroottes is nog niet bepaald.

7.6 Optimaal sorteren

Voor sorteeralgoritmen gebaseerd op het doen van arrayvergelijkingen hebben we bewezen dat het aantal vergelijkingen in de *worst case* ten minste $\lceil \lg n! \rceil$ bedraagt.
Vraag: hoe dicht kan men in de buurt van deze ondergrens komen?

Bekijk de volgende tabel:

n	2	3	4	5	6	7	8	9	10	11	12	...	21
$\lceil \lg n! \rceil$	1	3	5	7	10	13	16	19	22	26	29	...	66
$\mathrm{M}(n)$	1	3	5	8	11	14	17	21	25	29	33	...	74
$\mathrm{B}(n)$	1	3	5	8	11	14	17	21	25	29	33	...	74

Voor $n = 1$ zijn zowel $\lceil \lg n! \rceil$ als $\mathrm{M}(n)$ als $\mathrm{B}(n)$ gelijk aan 0. Hierbij is $\mathrm{M}(n)$ het aantal vergelijkingen dat Mergesort doet in de worst case op een rij van n elementen. Verder is $\mathrm{B}(n)$ het aantal vergelijkingen dat *Binary Insertion sort* in de worst case doet voor een rij met n elementen. Binary Insertion sort werkt als Insertion sort, maar gebruikt voor het zoeken van de plek waar $A[i]$ moet komen *binair zoeken* in plaats van lineair zoeken. (Een vraag: is het aantal vergelijkingen bij Binary Insertion sort nog wel een goede maat voor de complexiteit?) Om de plek te vinden waar $A[i]$ moet worden ingevoegd in $A[1], \ldots, A[i-1]$ zijn dan in het slechtste geval $\lceil \lg i \rceil$ vergelijkingen nodig. Dus:

$$B(n) = \Sigma_{i=2}^{n} \lceil \lg i \rceil \geq \lceil \Sigma_{i=2}^{n} \lg i \rceil = \lceil \lg n! \rceil$$

Vraag: Kan het nog beter? Bijvoorbeeld: kunnen 5 elementen met 7 vergelijkingen gesorteerd worden? **Antwoord**: Ja (Demuth, 1956). Zie college.

De methode van Demuth kan gegeneraliseerd worden tot het algoritme Merge Insertion sort (Ford & Johnson). Dit algoritme werkt als volgt:

1. Vergelijk de n elementen twee aan twee.
2. Sorteer de $\lfloor \frac{n}{2} \rfloor$ winnaars (de grootsten dus) recursief
3. Voeg nu de $\lfloor \frac{n}{2} \rfloor$ verliezers (en de losse waarde als n oneven is) op de juiste plek in *via een of andere handige volgorde* (zie college).

Merge Insertion sort sorteert bijvoorbeeld 10 elementen in 22 vergelijkingen (optimaal!), 12 elementen in 30 vergelijkingen (optimaal!) en 21 elementen in 66 vergelijkingen (optimaal!). **Vraag**: Is Merge Insertion sort optimaal voor wat betreft het aantal arrayvergelijkingen? **Antwoord**: Nee, bijvoorbeeld $n = 47$. Dit is de eerste waarde waarvan men zeker weet dat het algoritme daarvoor niet optimaal is. Er is voor het sorteren van 47 elementen een betere manier bekend.

7.7 O(n)-sorteren

We bekijken hier twee sorteermethoden die buiten de categorie van sorteeralgoritmen vallen die we hiervoor bekeken. Ze blijken beide een worst case complexiteit $O(n)$) te hebben. Dit is niet in tegenspraak met de eerder afgeleide ondergrens van $\lceil \lg n! \rceil$, aangezien deze twee algoritmen niet gebaseerd zijn op het doen van arrayvergelijkingen.

7.7.1 Counting sort

Invoer: een array A met n getallen $A[1], \ldots, A[n]$.
Aanname: elke $A[i]$ is een geheel getal tussen 1 en k (voor zekere k).
Uitvoer: een array B, voorstellende het array A oplopend gesorteerd.

Het basisidee (als alle $A[j]$'s verschillen): bepaal voor elke $X = A[j]$ het aantal array-elementen kleiner dan X. Deze informatie kan dan gebruikt worden om X meteen op de juiste positie in het uitvoerarray te zetten. Pas dit idee aan voor de situatie waarin sommige waarden meer dan eens in A voorkomen.

Het algoritme:

for $i := 1$ **to** k **do**
    $C[i] := 0$; // initialisatie
od
for $j := 1$ **to** n **do**
    $C[A[j]] := C[A[j]] + 1$; // telt aantal keer dat $A[j]$ in A voorkomt
od
for $i := 2$ **to** k **do**
    $C[i] := C[i] + C[i-1]$;
od
// $C[i]$ bevat nu het aantal getallen $\leq i$
for $j := n$ **downto** 1 **do** // !!
    $B[C[A[j]]] := A[j]$;
    $C[A[j]] := C[A[j]] - 1$;
od

Complexiteit: De complexiteit van Counting sort wordt bepaald door het aantal toekenningen aan array-elementen, en dat zijn er $O(n + k) \in$ (indien $k \in O(n)$) $O(n)$. De benodigde extra geheugenruimte is ook $O(n + k)$.

Counting sort is een *stabiele* sorteermethode, dat wil zeggen dat gelijke waarden uit het invoerarray A in dezelfde volgorde in het uitvoerarray B komen.

7.7.2 Radix sort

De sorteermethode Radix sort (Radixsort(A,d)) sorteert n getallen van elk d cijfers, waarbij elk cijfer een waarde heeft tussen 0 en $k - 1$ (bijvoorbeeld $k = 2$, of $k = 10$). De getallen worden gesorteerd door ze achtereenvolgens op het i-de cijfer te sorteren, te beginnen bij het minst significante cijfer.

Het algoritme:

for $i := 1$ **to** d **do**
// minst significante cijfer eerst, dus van rechts naar links
    sorteer A op het i-de cijfer
    // met een stabiele (!) methode
od

Het gebruik van een stabiele sorteermethode voor het sorteren op het i-de cijfer is essentieel. Een *voorbeeld* van de werking van Radix Sort met $n = 8$ en $d = 3$:

329		720		720		329
457		455		329		355
657		355		436		436
839	$\longrightarrow$	836	$\longrightarrow$	839	$\longrightarrow$	455
436		457		455		457
720		657		355		657
455		329		457		720
355		839		657		839

Complexiteit: Als we Counting sort gebruiken, dan kost elke ronde $O(k + n)$ stappen. totaal dus $O(dk + dn) \in O(n)$ als d een constante is en $k \in O(n)$. Een nadeel van deze methode is dat er net als bij Counting sort $O(n + k)$ extra geheugenruimte nodig is.

8 Polynoomevaluatie en matrixvermenigvuldiging

Zij $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ een polynoom van graad $n \geq 1$, met alle a_i reële getallen ($a_i \in \mathbb{R}$).
Gegeven: $a_0, a_1, \ldots, a_{n-1}, a_n$ en x. Gevraagd: de uitkomst $p(x)$.

We zullen drie methodes voor het evalueren van polynomen bekijken en vergelijken: de voor de hand liggende manier, de methode van Horner, en een methode die gebruikt maakt van preprocessing. Die laatste brengt het polynoom p eerst (ten koste van de nodige $+$-en en $*$-en) in een handige vorm, waardoor het evalueren zelf minder kost.

8.1 Voor de hand liggende methode

```
pol := a_0 + a_1 * x;
macht := x;
for i := 2 to n do
    macht := macht * x;
    // berekent x^2, x^3, ...
    pol := pol + a_i * macht;
od
```

Basisoperaties bij dit algoritme (en in het algemeen bij rekenkundige algoritmen) zijn de optelling ($+/-$) en de vermenigvuldiging ($*$). Het aantal $(+, -)$'s is in dit geval n, het aantal $*$ is $2n - 1$.

8.2 Methode van Horner

Schrijf $p(x)$ als:

$$p(x) = ([\ldots([a_n * x + a_{n-1}] * x + a_{n-2}) * x + \ldots a_2] * x + a_1) * x + a_0$$

De *methode van Horner* werkt dan als volgt:

```
pol := a_n;
for i := n − 1 downto 0 do
    pol := pol ∗ x + a_i;
od
```

Het aantal $(+, -)$'s is nog steeds n, maar het aantal $*$'s is met dit algoritme gedaald naar n. We kunnen ons afvragen of de methode van Horner optimaal is of niet.

Schema's: Rekenkundige algoritmen (algoritmen gebaseerd op de rekenkundige operaties $+, -, *$ en $/$ kunnen we beschrijven met behulp van *schema's*.
Een *schema* is een eindige serie stappen van de vorm $s_i := q \circ r$; Hierin is $\circ = *, /, +$ of $-$. Verder zijn q en r constanten (bijvoorbeeld $1, -1, \pi^2, \ldots$) of invoerwaarden (hier dus a_k's of x) of tussenresultaten van eerdere stappen. De laatste stap uit het schema berekent het eindresultaat (hier is dat $p(x)$).

Als voorbeeld het schema voor de methode van Horner ($n = 2$):

$s_1 := a_2 * x;$
$s_2 := s_1 + a_1;$
$s_3 := s_2 * x;$
$s_4 := s_3 + a_0;$

Stelling Een schema (dat alleen $+, -$ en $*$ gebruikt) om $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ te berekenen moet ten minste n $(+, -)$ stappen doen en n $*$ stappen.
Bewijs: zie college
Gevolg: De methode van Horner is optimaal.

8.3 Met preprocessing

Het idee is als volgt: bewerk het polynoom tot een polynoom in een speciale vorm waarop een nieuw evaluatie-algoritme sneller werkt.

Zonder beperking der algemeenheid kunnen we veronderstellen dat $p(x)$ een *monisch* polynoom is, dat wil zeggen dat $a_n = 1$. Laat dus $p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1x + a_0$. Verder zullen we aannemen dat $n = 2^k - 1$.

In de *speciale vorm* ziet het polynoom er dan zo uit:

$$p(x) = (x^j + b) * q(x) + r(x),$$

waarin q en r ook weer monisch zijn, beide met graad $2^{k-1} - 1$, en $j = 2^{k-1}$.
Als $q(x) = x^{j-1} + q_{j-2}x^{j-2} + \ldots + q_0$ en $r(x) = x^{j-1} + r_{j-2}x^{j-2} + \ldots + r_0$, dan geldt:

$$b + 1 = a_{j-1}, q_l = a_{l+j}, b * q_l + r_l = a_l$$

voor $l = 0, 1, \ldots, j - 2$. Het is niet moeilijk een algoritme te schrijven dat deze speciale vorm bepaalt. Zie hiervoor de opgaven.

Een voorbeeld (met $n = 7$):

$p(x) = x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x + 1 = (x^4 + 2)[(x^2 + 4)(x + 6) + (x - 20)] + [(x^2 - 10)(x - 10) + (x - 107)]$

Merk op dat Horner voor dit voorbeeldpolynoom 7 $(+, -)$'s gebruikt en 7 $*$'s. Als het polynoom in de speciale vorm staat heb je voor het evalueren van p in een waarde x slechts 5 $*$'s nodig en 10 $(+, -)$'s.

Als in het algemene geval het polynoom in de speciale vorm gebracht is kan het als volgt geëvalueerd worden:

1. Evalueer $q(x)$ en $r(x)$ *recursief*
2. Bereken de x^j's: nodig hiervoor zijn $x, x^2, x^4, \ldots, x^{2^{k-1}}$. Deze kunnen van tevoren berekend worden. Dat kost $k - 1$ $*$'s
3. Vermenigvuldig $(x^j + b)$ met $q(x)$ en tel er $r(x)$ bij op. Dat kost 1 $*$ en 2 $(+, -)$'s

We gaan nu het aantal $(+, -)$'s en het aantal $*$'s voor deze methode uitrekenen. Zij daartoe $M(k) =$ het aantal $*$'s dat gedaan wordt om een monisch polynoom van graad $2^k - 1$ te evalueren, zonder de berekening van de x^j mee te tellen. Zij verder $A(k) =$ het aantal $(+, -)$'s dat gedaan wordt om een monisch polynoom van graad $2^k - 1$ te evalueren.
Dan voldoen $M(k)$ en $A(k)$ aan de volgende recurrente betrekkingen:

$$M(k) = \begin{cases} 0 & k = 1 \\ 2M(k-1) + 1 & k > 1 \end{cases}$$

en

$$A(k) = \begin{cases} 1 & k = 1 \\ 2A(k-1) + 2 & k > 1 \end{cases}$$

Oplossing: $M(k) = 2^{k-1} - 1 = \frac{n-1}{2}$ en $A(k) = 3 * 2^{k-1} - 2 = \frac{3n-1}{2}$
Zie college en opgaven.

8.4 Matrixvermenigvuldiging

Ook een bekend rekenkundig probleem is het vermenigvuldigen van twee (vierkante) matrices. Stel dat A en B twee $n \times n$ matrices zijn met elementen a_{ij} en b_{ij} $(1 \leq i, j \leq n)$. De elementen c_{ij} van het (matrix-)product $C = A \cdot B$ zijn dan als volgt gedefinieerd: $c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$.

Een standaard $\Theta(n^3)$ algoritme hiervoor is recht-toe recht-aan uit de definitie:

```
for i := 1 to n do
    for j := 1 to n do
        c_ij := 0;
        for k := 1 to n do
```

$c_{ij} := c_{ij} + a_{ik} * b_{kj}$;
od
od
od

Een voorbeeld met $n = 2$:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

In totaal kost de recht-toe recht-aan methode (hieronder links) voor dit voorbeeld 8 vermenigvuldigingen van array-elementen. Het kan echter door 'herschrijven' met 7 (hieronder rechts):

$$\begin{aligned} 19 &= 1*5+2*7 \\ 22 &= 1*6+2*8 \\ 43 &= 3*5+4*7 \\ 50 &= 3*6+4*8 \end{aligned}$$

$$\begin{aligned} M_1 &= (1+4)*(5+8) = 65 \\ M_2 &= (3+4)*5 = 35 \\ M_3 &= 1*(6-8) = -2 \\ M_4 &= 4*(7-5) = 8 \\ M_5 &= (1+2)*8 = 24 \\ M_6 &= (3-1)*(5+6) = 22 \\ M_7 &= (2-4)*(7+8) = -30 \\ 19 &= M_1 + M_4 - M_5 + M_7 \\ 22 &= M_3 + M_5 \\ 43 &= M_2 + M_4 \\ 50 &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

Algemeen, uitgedrukt in array-elementen a_{ij} en b_{ij}: $M_2 = (a_{21} + a_{22}) * b_{11}$; $M_4 = a_{22} * (b_{21} - b_{11})$; dan $M_2 + M_4 = c_{21}$, etc.

We nemen nu aan dat $n = 2^k$. We kunnen een matrixproduct dan ook *recursief* berekenen door een n bij n matrix op te splitsen in vier $\frac{n}{2}$ bij $\frac{n}{2}$ matrices als volgt:

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Analoog aan het 2×2 geval kunnen we de berekening van de vier C_{ij}'s herschrijven zodanig dat we nog maar 7 vermenigvuldigingen van $\frac{n}{2} \times \frac{n}{2}$ matrices hoeven te doen.

$S_1, \ldots, S_{10}$ zijn $\frac{n}{2} \times \frac{n}{2}$ matrices die alle de som of het verschil van twee deelmatrices van A en B zijn. $P_1, \ldots, P_7$ zijn $\frac{n}{2} \times \frac{n}{2}$ matrices die alle het product van twee matrices S en/of deelmatrices van A en B zijn:

$$
\begin{array}{ll}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{array}
$$

$$
\begin{array}{lll}
P_1 &= A_{11} \cdot S_1 &= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\
P_2 &= S_2 \cdot B_{22} &= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 &= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\
P_5 &= S_5 \cdot S_6 &= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\
P_6 &= S_7 \cdot S_8 &= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\
P_7 &= S_9 \cdot S_{10} &= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}
\end{array}
$$

Merk op dat de berekening in de rechterkolom alleen voor onze informatie is, de enige *feitelijke* matrixvermenigvuldigingen staan in de middenkolom bij de berekening van de P_i. Dit zijn er 7.

De matrices C_{11}, C_{12}, C_{21} en C_{22} kunnen nu als volgt bepaald worden:

$$
\begin{array}{lll}
C_{11} &= P_5 + P_4 - P_2 + P_6 &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= P_1 + P_2 &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= P_3 + P_4 &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= P_5 + P_1 - P_3 - P_7 &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{array}
$$

We krijgen dus zeven keer een (recursieve) matrixvermenigvuldiging van $\frac{n}{2} \times \frac{n}{2}$ matrices, en we doen achttien optellingen van $\frac{n}{2} \times \frac{n}{2}$ matrices per recursiestap.

Het aantal vermenigvuldigingen van array-elementen $M(k)$, respectievelijk het aantal $+/-$ van array-elementen $A(k)$ voldoen aan de volgende recurrente betrekkingen ($n = 2^k$):

$$
M(k) = \begin{cases} 1 & k = 0 \\ 7M(k-1) & k > 0 \end{cases}
$$

$$
A(k) = \begin{cases} 0 & k = 0 \\ 7A(k-1) + 18 \cdot (2^{k-1})^2 & k > 0 \end{cases}
$$

Oplossing: $M(k) = 7^k = 2^{\lg 7^k} = 2^{k \cdot \lg 7} = n^{\lg 7} \approx n^{2.81}$
$A(k) = 6 \cdot 7^k - 6 \cdot 4^k \in \Theta(n^{\lg 7})$.

Het recursieve algoritme dat we op deze manier gekregen hebben is bedacht door Strassen in 1969. Er zijn in de tussentijd verbeteringen op deze methode bedacht (tot ongeveer $\Theta(n^{2,37})$), maar deze zijn voornamelijk theoretisch interessant.

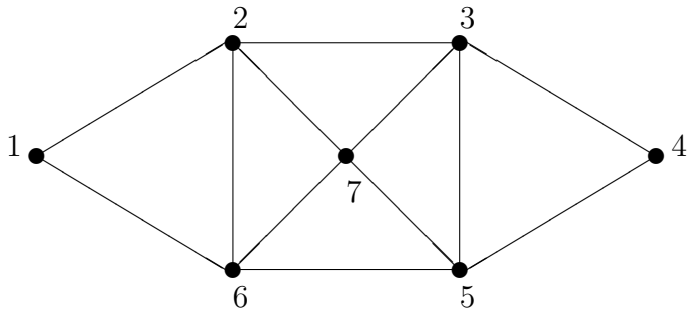
9 Eulerkringen en Hamiltonkringen

We bekijken hier twee problemen uit de grafentheorie die erg op elkaar lijken, maar waarvan de moeilijkheidsgraad toch essentieel verschilt: het bestaan van een *Eulerkring* respectievelijk een *Hamiltonkring* in een graaf.

9.1 Eulerkringen

Gegeven een samenhangende, ongerichte graaf $\mathcal{G} = (V, E)$.
Een kring in $\mathcal{G}$ die alle takken van $\mathcal{G}$ bevat heet een *Eulerkring*. Merk op: een kring bestaat per definitie uit allemaal verschillende takken.

Voorbeeld:



Voor deze graaf is 1 2 3 4 5 3 7 5 6 7 2 6 1 een Eulerkring.

Stelling: Een samenhangende ongerichte graaf heeft een Eulerkring $\Longleftrightarrow$ de graad van iedere knoop is even.
Bewijs: Zie college

Het bewijs van "$\Longleftarrow$" uit de stelling is constructief, en levert meteen een methode hoe men een Eulerkring kan construeren.

Algoritme voor de constructie van een Eulerkring (zie college voor een voorbeeld):

1. Controleer de samenhangendheid en controleer of elke knoop even graad heeft;
2. Als deze controle positief is dan
   3 Kies een knoop v;
   4 Construeer een kring $\mathcal{C}$ (van v naar v);
   // gewoon lopen en steeds andere takken bewandelen tot je in v terug bent
   5 Zolang er nog onbewandelde takken zijn doe het volgende
      6 Zoek de eerste knoop w van $\mathcal{C}$ die nog onbewandelde takken heeft;
      7 Construeer een kring (van w naar w) bestaande uit ongebruikte takken;
      8 Voeg deze kring in in $\mathcal{C}$ op de plek van w;

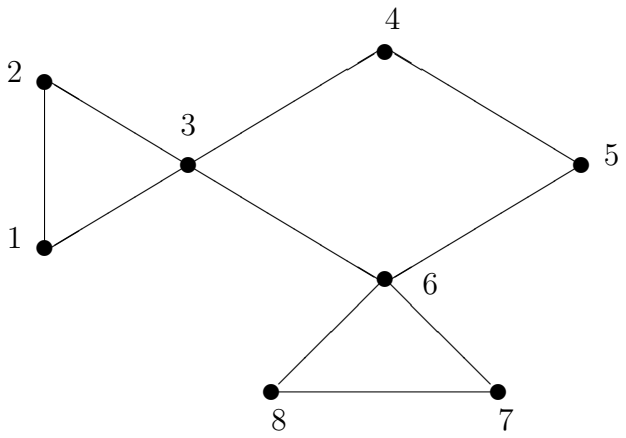
Complexiteit van dit algoritme: $O(|V| + |E|)$, dus lineair in de grootte van de probleeminvoer $\mathcal{G}$.
Immers: Stap 1 kan met behulp van Depth First Search ($O(|V| + |E|)$) en dan tegelijkertijd per knoop het aantal buren tellen en controleren of de graad even is ($O(|E|)$) in ($O(|V| + |E|)$) stappen in totaal. Gebruik een kopie van $\mathcal{G}$ ($O(|V| + |E|)$ om die te maken) en houdt de kring $\mathcal{C}$ bij als enkelverbonden lijst, met een pointer naar de eerste knoop die nog onbewandelde takken heeft. Dan is in te zien dat stap 4 t/m 8 $O(|E|)$ tijd kost. Zie

verder de opgaven.
Merk op dat we om de complexiteit te beschrijven verschillende soorten operaties tellen en op één hoop gooien.

9.2 Hamiltonkringen

Een probleem dat erg lijkt op het probleem van het vinden van een Eulerkring is het vinden van een *Hamiltonkring*.
Gegeven een ongerichte (of gerichte) graaf $\mathcal{G} = (V, E)$. Een *Hamiltonkring* in $\mathcal{G}$ is een kring die elke knoop precies één keer bevat.

Voorbeeld:



Bovenstaande graaf heeft geen Hamiltonkring (wel een Hamiltonpad overigens). De graaf die ontstaat door aan de voorbeeldgraaf de tak (1,8) toe te voegen heeft wel een Hamiltonkring, namelijk: 1, 2, 3, 4, 5, 6, 7, 8

Het **Hamiltonprobleem** (HC) voor ongerichte grafen (analoog voor gerichte grafen):
Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Hamiltonkring?

Naamgeving: Als een graaf $\mathcal{G}$ een Hamiltonkring heeft spreken we van een ja-instantie van het probleem. Als zo'n kring niet bestaat is $\mathcal{G}$ een nee-instantie.

Opmerkingen:

1. Een exponentieel algoritme is snel gevonden.
2. Er is geen polynomiaal algoritme voor dit probleem bekend.
3. Er is ook niet bewezen dat een exponentieel algoritme nodig is.
4. Als $\mathcal{G}$ een Hamiltonkring heeft is er een makkelijke manier om iemand daarvan te overtuigen. Immers als je een kandidaat-Hamiltonkring geeft is in een polynomiaal (dus makkelijk) aantal stappen te controleren dat dit inderdaad een Hamiltonkring is. Er bestaat dus voor ja-instanties een *certificaat* (in dit geval de Hamiltonkring)

waarmee je eenvoudig (polynomiaal) kunt bewijzen dat de graaf inderdaad een ja-instantie is.

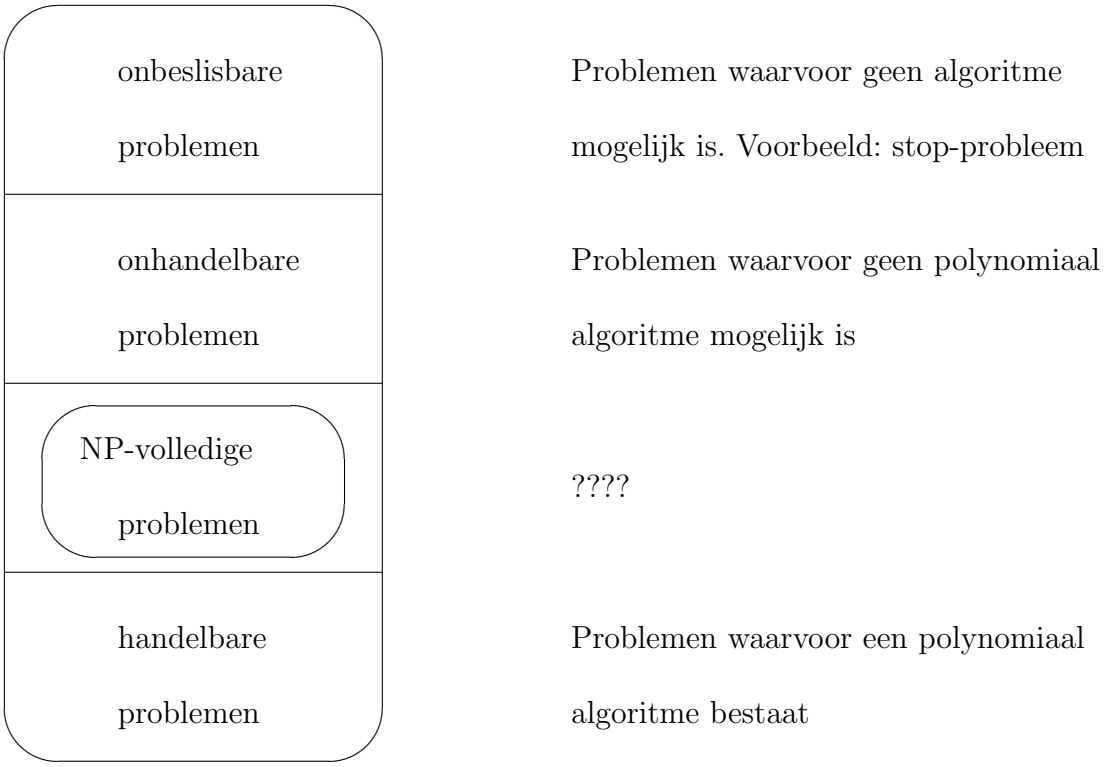
5. Bestaat er voor nee-instanties ook zo'n certificaat?

6. Het Hamiltonprobleem is een bekend voorbeeld van een NP-volledig probleem.

10 NP-volledigheid

10.1 Introductie

In de vorige hoofdstukken zijn we problemen tegengekomen waarvoor een polynomiaal ($O(n^k)$), met n een maat voor de grootte van de invoer van het probleem) algoritme bestaat. Er zijn echter ook problemen waarvan bewezen is dat er geen polynomiaal algoritme voor bestaat. Voor deze problemen bestaat wellicht wel een exponentieel algoritme ($O(2^n)$) (of zelfs super-exponentieel, bijvoorbeeld $O(n^n)$), of het probleem is zelfs onoplosbaar. Een polynomiaal algoritme kan over het algemeen voor behoorlijk grote invoer nog in acceptabele tijd worden uitgevoerd, een exponentieel algoritme kan dat zeker niet. Daarom noemen we problemen waarvoor een polynomiaal algoritme bestaat handelbaar en problemen die een exponentieel algoritme nodig hebben (of erger) onhandelbaar. De klasse van handelbare problemen noteren we met $\mathcal{P}$. We kunnen de ruimte van alle problemen aldus proberen op te delen in onbeslisbare (onoplosbare) problemen, onhandelbare problemen en handelbare problemen.



Er blijft echter nog een grote groep problemen over waarvan het onbekend is of ze handelbaar of onhandelbaar zijn. Hierbinnen bevindt zich de klasse van *NP-volledige problemen*, waartoe ook het in het vorige hoofdstuk genoemde Hamiltonkringprobleem behoort. In het volgende zullen we vooral deze klasse van problemen bekijken.

De klasse van *NP-volledige problemen* heeft enkele interessante eigenschappen, zoals:

1. Voor geen enkel NP-volledig probleem is tot dusver een polynomiaal algoritme gevonden. Men vermoedt dat ze onhandelbaar zijn, maar dat heeft tot dusver ook nog niemand kunnen bewijzen

2. Als er een polynomiaal algoritme bestaat voor willekeurig welk NP-volledig probleem, dan is *elk* NP-volledig probleem in polynomiale tijd oplosbaar. Omgekeerd: als er van één enkel NP-volledig probleem bewezen wordt dat het onhandelbaar is, dan zijn *alle* NP-volledige problemen onhandelbaar.

De theorie van NP-volledigheid beperkt zich tot *beslissingsproblemen*, dat wil zeggen problemen waarvoor het antwoord ja of nee is. Veel interessante problemen zijn echter *optimalisatieproblemen*. Die moeten dus omgezet worden naar een beslissingsprobleem. We zullen de corresponderende beslissingsproblemen steeds zo formuleren dat geldt: als het beslissingsprobleem onhandelbaar is, dan is het corresponderende optimalisatieprobleem dat ook.

10.2 Voorbeelden van NP-volledige problemen

Alvorens wat preciezer te maken wat NP-volledigheid (en aanverwante begrippen) nu eigenlijk is eerst enige voorbeelden van bekende NP-volledige problemen. Merk op dat voor al deze problemen eenvoudig een exponentieel algoritme gevonden kan worden.

10.2.1 CNF-satisfiability (SAT)

Dit is een probleem uit de propositielogica. Een *literal* is een Boolese variabele of de negatie daarvan (dus x of $\neg x$). Een *clause* (*clausule*) is een disjunctie ($\vee$) van literals, bijvoorbeeld $x_1 \vee \neg x_3 \vee x_4 \vee \neg x_6$. Een logische formule staat in CNF (*conjunctieve normaalvorm*) als hij bestaat uit een conjunctie ($\wedge$) van clausules. Een voorbeeld van een logische formule ϕ in CNF: $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge \neg x_1$.
Een *waardering* of *waarheidstoekenning* van een verzameling logische variabelen is een toekenning van de waarde True of False aan elk van de logische variabelen uit die verzameling.
Voorbeeld: de waardering w met $w(x_1) =$ True, $w(x_2) =$ True en $w(x_3) =$ False maakt de logische formule ϕ (zie hierboven) False.

Beslissingsprobleem: Gegeven een logische formule ϕ in CNF. Bestaat er een waardering van de in ϕ voorkomende logische variabelen zodat ϕ de waarde True krijgt (dus een waardering die ϕ waar maakt)?

10.2.2 Subset Sum (SUM)

Beslissingsprobleem: Gegeven een getal $t \in \mathbb{N}$ en een eindige verzameling $S \subset \mathbb{N}$. Bestaat er een deelverzameling $S' \subseteq S$ met $\Sigma_{s \in S'} s = t$?

Voorbeeld: Neem $S = \{1, 4, 16, 64, 256, 1040, 1093, 1284, 1344\}$ en $t = 3754$, dan is dit een ja-instantie voor het probleem. Immers $S' = \{1, 4, 16, 256, 1040, 1093, 1344\}$ voldoet.

10.2.3 Hamiltonkring (HC)

Een Hamiltonkring in een ongerichte (of gerichte) graaf is een kring die *elke* knoop precies *één* keer bevat.

Beslissingsprobleem: Gegeven een graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Hamiltonkring?

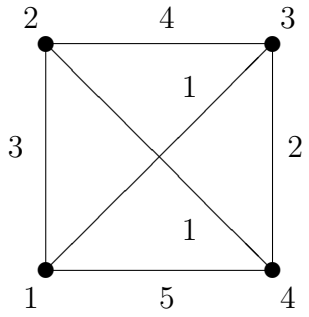
Een hieraan gerelateerd optimalisatieprobleem is het zogenaamde Handelsreizigersprobleem (Travelling Salesperson Problem).

10.2.4 Travelling Salesperson Problem (TSP)

Zij $\mathcal{G}$ een ongerichte graaf. $\mathcal{G}$ heet *volledig* indien tussen *elk* tweetal knopen van $\mathcal{G}$ een tak zit.

Optimalisatieprobleem: Gegeven een volledige, ongerichte graaf $\mathcal{G} = (V, E)$ met gewichten op de takken. Gevraagd wordt een Hamiltonkring in $\mathcal{G}$ met minimaal totaalgewicht.
Beslissingsprobleem: Gegeven een volledige, ongerichte graaf $\mathcal{G} = (V, E)$ met gewichten op de takken, en een geheel getal $k \geq 0$. Bestaat er in $\mathcal{G}$ een Hamiltonkring met totaalgewicht $\leq k$?

Voorbeeld: Een Hamiltonkring in onderstaande graaf is bijvoorbeeld $1, 2, 3, 4$. Deze heeft totaalgewicht 14. De Hamiltonkring met minimaal gewicht is $2, 4, 3, 1$. Deze heeft totaalgewicht 7.



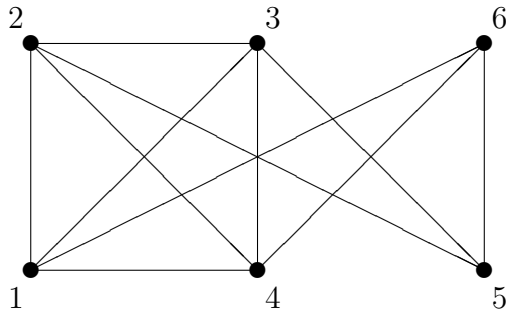
Merk op dat, net als bij de hieropvolgende problemen, geldt: als het optimalisatieprobleem handelbaar (in polynomiale tijd oplosbaar) is, dan zeker ook het bijbehorende beslissingsprobleem. En dus ook het omgekeerde: als het beslissingsprobleem onhandelbaar is, dan is ook het oorspronkelijke optimalisatieprobleem onhandelbaar.

10.2.5 Clique (Kliek)

Een *kliek* in een ongerichte graaf $\mathcal{G} = (V, E)$ is een deelverzameling $V' \subseteq V$ zodanig dat voor elk tweetal knopen $u, v \in V'$ $(u \neq v)$ geldt dat $(u, v) \in E$. (Met andere woorden: tussen elk tweetal knopen uit V' zit een tak.) De grootte van de kliek is het aantal knopen van die kliek.

Optimalisatieprobleem: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Gevraagd wordt een kliek met zo veel mogelijk knopen (een maximale kliek).
Beslissingsprobleem: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$ en een geheel getal k $(0 \leq k \leq |V|)$. Bestaat er in $\mathcal{G}$ een kliek ter grootte ten minste k?
Opmerking: het is equivalent om te vragen naar een kliek ter grootte gelijk aan k. Immers, elke deelverzameling van een kliek is weer een kliek. Dus als er een kliek ter grootte $\geq k$ bestaat, dan ook een ter grootte $= k$. En natuurlijk geldt ook: als er een kliek W ter grootte $= k$ bestaat, dan ook een ter grootte $\geq k$ (namelijk W). Ergo: er is een kliek ter grootte $\geq k \iff$ er is een kliek ter grootte $= k$.

Voorbeeld: In onderstaande graaf is $\{1, 4, 6\}$ een kliek ter grootte 3. Een maximale kliek in deze graaf is er een ter grootte 4: $\{1, 2, 3, 4\}$.

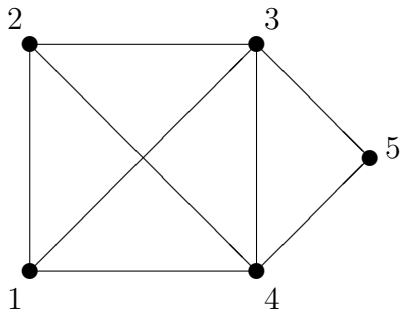


10.2.6 Graafkleuring (Kleur)

Een *kleuring* van (de knopen van) een ongerichte graaf $\mathcal{G} = (V, E)$ is een afbeelding $c : V \to S$, waarin S een eindige verzameling (van kleuren) is, en wel zó dat als $(v, w) \in E$ dan $c(v) \neq c(w)$. Met andere woorden: aangrenzende knopen hebben niet dezelfde kleur.

Optimalisatieprobleem: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Gevraagd een kleuring van $\mathcal{G}$ met zo weinig mogelijk kleuren.
Beslissingsprobleem: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$ en een geheel getal $k > 0$. Bestaat er een kleuring van $\mathcal{G}$ die hooguit k kleuren gebruikt (ofwel, is $\mathcal{G}$ k-kleurbaar) ?
Equivalente formulering: is er een kleuring met precies k kleuren? Bewijs van de equivalentie zoals hiervoor bij Clique.

Voorbeeld: de volgende graaf kan met 4 kleuren gekleurd worden, maar niet met minder dan 4 (bedenk zelf waarom niet). Kleur bijvoorbeeld 1 en 5 rood, 2 blauw, 3 geel en 4 paars. Dit is dus een minimale kleuring.



10.2.7 Gemeenschappelijke eigenschappen

1. Voor alle zes voorbeeldproblemen is eenvoudig een exponentieel algoritme op te schrijven

2. In alle voorbeeldproblemen kan in polynomiale tijd gecontroleerd worden of een kandidaatoplossing een echte oplossing is

3. Voor al deze problemen geldt: het lijkt extreem moeilijk (exponentieel) om voor gegeven invoer x te bepalen of het antwoord "ja" of "nee" moet zijn. Echter, *als* x een *ja-instantie* is, dan is er een eenvoudige (polynomiale) manier om iemand daarvan te overtuigen.

4. Voor ja-instanties bestaat er een zogenaamd *certificaat* (in de voorbeelden steeds een voorgestelde oplossing, maar dit zou iets anders kunnen zijn) dat gebruikt kan worden om te laten zien dat het antwoord inderdaad "ja" is.

5. Bovendien is dit certificaat kort (polynomiaal)

6. Eigenschap 2 t/m 5 betekent dat de genoemde problemen in $\mathcal{NP}$ zitten. Verderop in dit dictaat wordt alles wat formeler gemaakt

7. Ja-instanties zijn eenvoudig te verifiëren met de juiste hint (certificaat). Hoe zit het met nee-instanties?

10.3 De klasse $\mathcal{P}$

10.3.1 Definities

Definitie: Een algoritme heet *polynomiaal begrensd* als zijn worst case complexiteit van boven begrensd is door een functie die polynomiaal is in de lengte van de invoer. Dus: worst case is in $O(p(n))$ voor een zeker polynoom p, en met n (een maat voor) de lengte van de invoer.

Definitie: Een probleem heet *polynomiaal begrensd* als er een polynomiaal begrensd algoritme voor bestaat.

Definitie: De klasse van beslissingsproblemen die *polynomiaal begrensd* zijn noteren we als $\mathcal{P}$.

Vraag: Waarom handelbaar = polynomiaal begrensd?
Antwoord:

- als een probleem niet in $\mathcal{P}$ zit is het zeker onhandelbaar
- de klasse $\mathcal{P}$ heeft mooie afsluitingseigenschappen: zo is bijvoorbeeld een algoritme dat bestaat uit de opeenvolging van polynomiaal begrensde algoritmes, zelf ook weer polynomiaal begrensd
- de klasse $\mathcal{P}$ is onafhankelijk van het berekeningsmodel en van gebruikte coderingen

10.3.2 De lengte van de invoer

De complexiteit van een algoritme wordt gegeven als functie van de grootte/lengte van de invoer. Tot dusver hebben we als maat voor de grootte van de invoer bijvoorbeeld n, het aantal getallen, gebruikt. Is dit voldoende of moeten we de lengte van de invoer preciezer bekijken? In dat geval is de codering van de invoer van belang. Maakt het dan nog uit wat voor codering gebruikt wordt, bijvoorbeeld binaire of decimale codering? Aan de hand van een voorbeeld zullen we daar iets over zeggen.

Probleem: Gegeven een geheel getal $n > 0$. Heeft n echte delers, met andere woorden, is $n = a * b$ voor zekere $a, b > 1$?

Algoritme:
// gewoon alle mogelijke delers proberen

```
gevonden := False;
m := 2;
while not gevonden and m < n do
    if n mod m = 0 then
        gevonden := True;
    else
        m := m + 1;
    fi
od
// m is nu de kleinste deler van n
```

De worst case complexiteit van dit algoritme is in $O(n)$ (indien we de berekening van n mod m voor 1 stap tellen, anders $O(n^2)$). De lengte van de invoer is het aantal karakters van de codering, in dit geval van het getal n. Als we de unaire codering gebruiken is het algoritme dus lineair in de lengte van de invoer. Nemen we de binaire codering, of in het algemeen de l-aire codering met $l > 1$, dan is het algoritme exponentieel in de lengte van de invoer (voor elke $l > 1$ dus). We gaan er vanaf nu vanuit dat we altijd een representatie kiezen met basis > 1. Dit is niet alleen gebruikelijker dan de unaire representatie, maar er geldt dan ook i.h.a. dat de precieze codering (keuze van de basis) geen invloed heeft op het polynomiaal of niet-polynomiaal zijn van de complexiteit.

Verder kunnen we net als vroeger het aantal elementen/knopen/takken/... in de invoer als maat voor de grootte van de invoer nemen, in plaats van de werkelijke lengte van

de gecodeerde invoer. Dit komt omdat we alleen geïnteresseerd zijn in polynomiaal of niet-polynomiaal.

10.4 De klasse $\mathcal{NP}$

10.4.1 Voorbereiding

Een *certificaat* kun je gebruiken om aan te tonen dat een ja-instantie van een probleem inderdaad een ja-instantie is. Veel beslissingsproblemen zijn geformuleerd als "er-is-een"-vragen. In dat geval kun je bij een certificaat denken aan een *kandidaat-oplossing* (*voorgestelde oplossing*) van het probleem. Dan moet daarvan geverifieerd worden dat hij aan de criteria van het probleem voldoet, dus een *echte* oplossing van het probleem is.

Vanuit de achterliggende theorie gezien worden zowel probleeminvoeren als certificaten beschreven door strings. Zo'n string bestaat bijvoorbeeld uit de karakters van het toetsenbord, of uit nullen en enen, of .... Er wordt verder afgesproken hoe je wat ingewikkelder abstracte datatypen, zoals bijvoorbeeld een graaf, codeert als string. Dit kan bijvoorbeeld door een graaf $\mathcal{G}$ met n knopen (genummerd 1 t/m n) te noteren als: $x =< \mathcal{G} >= n$; knoop, buren; knoop, buren; ...; knoop, buren. (Dat is gewoon de adjacency-list representatie, lineair genoteerd.) Dit heet de *codering* van het probleem. Om een en ander precies te maken wordt als berekeningsmodel wel de Turingmachine gebruikt (zie hoofdstuk 11). Een Turingmachine is een vereenvoudigd model van een computer, waarvan de werking heel precies is gedefinieerd. Een Turingmachine kan maar een beperkt aantal operaties doen en werkt op invoer in de vorm van een string van bijvoorbeeld nullen en enen, maar is toch even krachtig als elke ander computer.[4] Wij zullen in dit college wat informeler te werk gaan en niet al te precies met de codering omgaan, en ook geen Turingmachineprogramma's schrijven. Dit is in veruit de meeste gevallen voldoende.
Als theoretisch raamwerk zullen we niet-deterministische algoritmen gebruiken. (Als we precies zouden zijn: niet-deterministische Turingmachines.) De klasse $\mathcal{NP}$ wordt overigens ook wel beschreven in termen van formele talen. Immers, als we bijvoorbeeld de standaard binaire codering gebruiken kunnen we een beslissingsprobleem zien als een functie van $\{0,1\}^*$ naar $\{0,1\}$ ({nee, ja}). Beide benaderingen zijn equivalent.

10.4.2 Niet-deterministische algoritmen

De definitie van $\mathcal{NP}$ gebruikt niet-deterministische algoritmen. Zo'n algoritme bestaat uit twee fases en een uitvoerstap.

1. Niet-deterministische *gokfase*
   Er wordt een willekeurige string s in het geheugen geschreven. Elke keer dat het algoritme wordt uitgevoerd kan dit een andere string zijn.
   // Deze string is het certificaat, het is een soort gok van de oplossing van het
   // probleem; s kan echter ook een onzinstring zijn.

2. Deterministische *verificatiefase*
   Zowel de invoer van het probleem als de string s mogen hier gebruikt worden. Er

[4] Volgens de Church-Turing these kan elk algoritme dat door een elektronische computer kan worden uitgevoerd ook door een Turing machine worden uitgevoerd en omgekeerd.

wordt True of False geretourneerd of het programma gaat in een oneindige loop en stopt nooit.
// Hier wordt gecontroleerd of s een oplossing van het probleem is bij de gegeven
// invoer, met andere woorden, er wordt gecontroleerd of s een ja-antwoord
// rechtvaardigt.

3. Uitvoerstap
   Als fase 2 True retourneert geeft het algoritme antwoord "ja". Anders is er geen uitvoer.

Het aantal stappen dat een niet-deterministisch algoritme doet is het aantal stappen nodig om s te schrijven (dus het aantal karakters waaruit s bestaat) + het aantal stappen dat gedaan wordt in de verificatiefase (+1 voor de uitvoerstap).

Definitie: Het antwoord van een niet-deterministisch algoritme A voor invoer x is "ja" $\Longleftrightarrow$ *er is een executie* van A die "ja" als uitvoer geeft.
Het antwoord van A is "nee" als er voor geen enkele executie, dus voor geen enkele string s, een uitvoer is.

Er geldt dus: het antwoord van een niet-deterministisch algoritme A voor invoer x is "ja" $\Longleftrightarrow$ er bestaat een string s (certificaat) waarmee je kunt aantonen dat x een ja-instantie is.

Definitie: Een niet-deterministisch algoritme heet *polynomiaal begrensd* als er een polynoom p bestaat zodat voor elke invoer ter grootte n waarvoor het antwoord "ja" is, er een executie van het algoritme is die "ja" oplevert in hooguit $p(n)$ stappen.
De string s mag dus niet te lang zijn (polynomiaal), en het verificatie-algoritme uit fase 2 moet polynomiaal begrensd zijn in x (en s).

Definitie: $\mathcal{NP}$ is de klasse van beslissingsproblemen waarvoor er een polynomiaal begrensd niet-deterministisch algoritme bestaat.
$\mathcal{NP}$ betekent: $\mathcal{N}$on-deterministic $\mathcal{P}$olynomial time.

Stelling: De zes voorbeeldproblemen zitten alle in $\mathcal{NP}$.
Bewijs: zie volgende paragraaf voor een voorbeeld, en verder het college en de opgaven

Stelling: $\mathcal{P} \subseteq \mathcal{NP}$.
Bewijs: zie college.

Een van de belangrijkste vragen uit de theoretische informatica is: is $\mathcal{P} = \mathcal{NP}$ of is $\mathcal{P}$ een echte deelverzameling van $\mathcal{NP}$?

10.4.3 HC $\in \mathcal{NP}$

Gegeven een (gerichte of ongerichte) graaf $\mathcal{G}$. We noteren de (op een of andere manier) gecodeerde graaf $< \mathcal{G} >$ als x.
Vraag: heeft deze graaf een Hamiltonkring?

Om te bewijzen dat HC $\in \mathcal{NP}$ is het voldoende om een polynomiaal (begrensd) niet-deterministisch algoritme A te geven voor HC.

1. Fase 1 (gokfase)
   Er wordt een string s gegenereerd, hierna te interpreteren als een rij gehele getallen.

2. Fase 2 (verificatiefase)
   Er wordt gecontroleerd of s een Hamiltonkring voorstelt:

   (1) controleer dat er precies n integers staan: $O(|s|)$
   (We nemen aan dat we weten dat V precies n knopen bevat; zo niet, dan moeten we de knopen eerst nog even tellen, maar dat is polynomiaal: $O(|V|)$, dus niet van invloed op de polynomialiteit van fase 2)

   (2) controleer dat elke integer tussen 1 en n zit: $O(|s|)$
   (We nemen hier aan dat we weten dat de knopen genummerd zijn met 1 t/m n. Als dat niet zo is moeten we voor elke s_i de knoopverzameling V aflopen om te kijken of s_i daarin voorkomt: dit blijft polynomiaal, namelijk $O(|s| \cdot |V|)$).

   (3) controleer dat alle knopen uit s verschillend zijn: $O(|s|^2)$

   (4) controleer dat tussen opeenvolgende knopen uit s een tak zit in de graaf (en tussen de laatste en de eerste): $O(|s| \cdot |x|)$

   Merk op: Stap (1) t/m (3) controleert dat s een rij van n verschillende knopen van $\mathcal{G}$ voorstelt (soort syntactische controle), stap (4) controleert dat het een Hamiltonkring is. Als de vier tests positief zijn wordt True geretourneerd, anders False (of er wordt in een oneindige loop gegaan).

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

Er geldt: als x een ja-instantie is bestaat er een string s (het certificaat) die een Hamiltonkring voorstelt. Dus dan is er een executie van A die "ja" als uitvoer geeft. Voor nee-instanties bestaat er geen Hamiltonkring en is er dus geen "goede" string. Derhalve levert geen enkele executie van A daarop een uitvoer.
Ergo: A geeft antwoord "ja" voor invoer $x \iff x$ is een ja-instantie voor HC
(en A geeft antwoord "nee" $\iff x$ is een nee-instantie)

En verder: A is polynomiaal begrensd.
Namelijk als x een ja-instantie is, en s een goede string, dan is $|s| \in O(|x|)$, en de verificatiefase is dan $O(|x|^2)$, polynomiaal in x.
Dus voor elke invoer x waarvoor A "ja" antwoordt is er een executie die dat in $O(|x|^2)$ stappen doet.

10.5 De klasse $\mathcal{NPC}$

10.5.1 Inleiding

De term *NP-compleet* of *NP-volledig* wordt gebruikt voor beslissingsproblemen die de moeilijkste in $\mathcal{NP}$ zijn in de volgende betekenis: als er een polynomiaal begrensd algoritme

bestaat voor *één* NP-volledig probleem, dan bestaat er meteen voor *elk* probleem in $\mathcal{NP}$ een polynomiaal begrensd algoritme.
De definitie van NP-volledigheid maakt gebruik van het begrip reducties.

10.5.2 Reducties

Zij T een functie van de invoerverzameling I van een beslissingsprobleem P naar de invoerverzameling I' van een beslissingsprobleem Q. T beeldt dus elke $x \in I$ af op een $T(x) \in I'$.

Definitie T heet een *polynomiale reductie* (of *polynomiale transformatie*) van P naar Q als de volgende drie punten gelden:

1. T kan berekend worden in polynomiaal begrensde tijd (als functie van $|x|$)
2. Voor elke x uit I geldt: als x een ja-instantie is voor P dan is $T(x)$ een ja-instantie voor Q
3. Voor elke x uit I geldt: als x een nee-instantie is voor P dan is $T(x)$ een nee-instantie voor Q

We kunnen punt 3 ook als volgt formuleren (beide formuleringen zijn equivalent):

3′. Voor elke x uit I geldt: als $T(x)$ een ja-instantie is voor Q dan is x een ja-instantie voor P

Definitie: Een probleem P is *polynomiaal reduceerbaar* (of *polynomiaal transformeerbaar*) naar Q als er een polynomiale reductie bestaat van P naar Q.
Notatie: $P \leq_{\mathrm{P}} Q$.

Stelling: Als $P \leq_{\mathrm{P}} Q$ en Q zit in $\mathcal{P}$, dan zit P ook in $\mathcal{P}$.
Bewijs: zie college.

10.5.3 NP-hard en NP-volledig

Definitie: Een probleem Q is *NP-hard* (ofwel *NP-moeilijk*) als *elk* probleem P in $\mathcal{NP}$ polynomiaal reduceerbaar is tot Q, dat wil dus zeggen dat $P \leq_{\mathrm{P}} Q$ *voor alle* $P \in \mathcal{NP}$.

Definitie: Een probleem Q is *NP-volledig* als

1. $Q \in \mathcal{NP}$
2. Q is NP-hard

Notatie: De klasse van NP-volledige problemen geven we aan met $\mathcal{NPC}$ (NP-complete).

Stelling: Als een of ander NP-volledig probleem in $\mathcal{P}$ zit, dan is $\mathcal{P} = \mathcal{NP}$.
Bewijs: zie college.

Deze stelling zegt dus iets heel sterks: als één enkel NP-volledig probleem P polynomiaal begrensd is, dan zijn *alle* problemen uit $\mathcal{NP}$ polynomiaal begrensd.
Omgekeerd geldt: als een willekeurig probleem in $\mathcal{NP}$ niet polynomiaal begrensd is, dan zijn *alle* NP-volledige problemen niet polynomiaal begrensd.

We geven hieronder een methode om aan te tonen dat een gegeven probleem P NP-volledig is. Hiertoe bewijzen we de volgende stelling met bijbehorend lemma.

Lemma: $\leq_{\mathrm{P}}$ is *transitief*, dat wil zeggen: als $P_1 \leq_{\mathrm{P}} P_2$ en $P_2 \leq_{\mathrm{P}} P_3$ dan is $P_1 \leq_{\mathrm{P}} P_3$ (zie de opgaven).

Stelling: Stel Q is een probleem waarvoor geldt dat $P \leq_{\mathrm{P}} Q$ voor een of andere $P \in \mathcal{NPC}$.
Dan is Q NP-hard.
Als bovendien $Q \in \mathcal{NP}$, dan geldt dat $Q \in \mathcal{NPC}$.
Bewijs: zie college.

Dus door een bekend NP-volledig probleem te reduceren tot Q reduceren we impliciet alle problemen uit $\mathcal{NP}$ tot Q. De stelling geeft ons derhalve een methode om aan te tonen dat een probleem Q NP-volledig is:

1. Bewijs dat $Q \in \mathcal{NP}$
2. Kies een bekend NP-volledig probleem P
3. Toon aan dat $P \leq_{\mathrm{P}} Q$

Stap 3 valt weer uiteen in:

3a. Geef een functie T van I (de invoerverzameling van P) naar I' (de invoerverzameling van Q) die elke $x \in I$ afbeeldt op een element van I'

3b. Laat zien dat T berekend kan worden in polynomiaal begrensde tijd

3c. Toon aan dat T voldoet aan: $x \in I$ is een ja-instantie voor $P \iff T(x) \in I'$ is een ja-instantie voor Q

In 1971 bewees *Stephen Cook* op een directe manier (dus door een reductie te geven van alle problemen uit $\mathcal{NP}$ naar SAT) dat SAT NP-volledig is. Sindsdien is met behulp van de *reductiemethode* van zeer veel bekende problemen aangetoond dat ze NP-volledig zijn.

De zes eerder genoemde voorbeeldproblemen zijn alle NP-volledig. Dit valt te bewijzen door een keten van reducties te bewijzen, te beginnen met SAT. Bijvoorbeeld:
SAT $\longrightarrow$ 3SAT $\longrightarrow$ Kliek en 3SAT $\longrightarrow$ HC $\longrightarrow$ TSP, waarin $P \longrightarrow P'$ betekent dat $P \leq_{\mathrm{P}} P'$.

10.6 Voorbeelden van reducties

10.6.1 Een eenvoudige reductie

Als eenvoudig voorbeeld van een reductie zullen we het Hamiltonpadprobleem reduceren tot het Hamiltonkringprobleem.

HP: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Gevraagd: heeft $\mathcal{G}$ een Hamiltonpad?
HC: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Gevraagd: heeft $\mathcal{G}$ een Hamiltonkring?

Transformeer een ongerichte graaf $\mathcal{G}$ naar een ongerichte graaf $\mathcal{G}'$ door één knoop toe te voegen en deze met een tak te verbinden met alle andere knopen. Noem deze transformatie T. Deze transformatie is een polynomiale reductie, dus er geldt (via deze T):

Bewering: HP $\leq_{\mathrm{P}}$ HC.

Bewijs:
(1) De constructie is polynomiaal: eerst $\mathcal{G}$ kopiëren ($O(|\mathcal{G}|)$), dan de knoop toevoegen (dat is $O(|V|)$ als we bijvoorbeeld voor grafen de adjacency list representatie kiezen met een lijst in plaats van een array) en $|V|$ takken aanleggen ($\in O(2|V|) = O(|V|)$ als we vooraan in de buurlijsten toevoegen). Dit is samen $O(|\mathcal{G}|)$.
(2) $\mathcal{G}$ heeft een Hamiltonpad $\Longleftrightarrow$ $\mathcal{G}'$ heeft een Hamiltonkring.
"$\Longrightarrow$": Zij $v_1, v_2, \ldots, v_n$ een Hamiltonpad in $\mathcal{G}$, dus alle v_i zijn verschillend, alle knopen van V komen voor en er is een tak tussen v_i en v_{i+1} voor elke $i = 1, 2, \ldots, n-1$ ($n = |V|$). Dan is $v_1, v_2, \ldots, v_n, v$ een Hamiltonkring in $\mathcal{G}'$. Immers alle $n+1$ knopen verschillend, takken tussen v_i en v_{i+1} (want Hamiltonpad in $\mathcal{G}$) en v is verbonden met alle v_i (constructie), dus in het bijzonder met v_n en v_1.
"$\Longleftarrow$": Stel dat we een Hamiltonkring in $\mathcal{G}'$ hebben: $v, {v'}_1, {v'}_2, \ldots, {v'}_n$. Deze bevat dus alle knopen van $\mathcal{G}'$, waaronder v. Bewering: dan is ${v'}_1, {v'}_2, \ldots, {v'}_n$ een Hamiltonpad in $\mathcal{G}$. Immers: er zit een tak tussen ${v'}_i$ en ${v'}_{i+1}$ voor $i = 1, \ldots, n$ en alle n knopen zijn verschillend en zitten in V (volgt uit de Hamiltonkring van $\mathcal{G}'$ en de constructie van $\mathcal{G}'$ uit $\mathcal{G}$).

10.6.2 Nog een eenvoudige reductie

We bekijken het Hamiltonkringprobleem voor gerichte grafen en voor ongerichte grafen.

HC1: Gegeven een *gerichte* graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Hamiltonkring?
HC2: Gegeven een *ongerichte* graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Hamiltonkring?

Bewering: HC1 $\leq_{\mathrm{P}}$ HC2

Hint bij het bewijs (voor de rest van het bewijs zie college):
Transformeer een gerichte graaf $\mathcal{G} = (V, E)$ als volgt naar een ongerichte graaf $\mathcal{G}' = (V', E')$. Laat $V' = \{v_1, v_2, v_3 : v \in V\}$, dus elke knoop $v \in V$ wordt afgebeeld op een drietal knopen v_1, v_2, v_3. Laat verder $E' = \{(v_1, v_2), (v_2, v_3) : v \in V\} \cup \{(v_3, w_1) : (v, w) \in E\}$, met andere woorden: binnen elk drietal knopen corresponderend met v loopt een tak tussen v_1 en v_2 en tussen v_2 en v_3, en voor elke tak van v naar w in $\mathcal{G}$ komt een tak in $\mathcal{G}'$ tussen v_3 en w_1. Noem deze transformatie T. Dan geldt:

1. T kan in polynomiaal begrensde tijd berekend worden
2. $\mathcal{G}$ is een ja-instantie voor HC1 $\Longleftrightarrow$ $\mathcal{G}'$ is een ja-instantie voor HC2, ofwel
   $\mathcal{G}$ heeft een gerichte Hamiltonkring $\Longleftrightarrow$ $\mathcal{G}'$ heeft een ongerichte Hamiltonkring

Opmerking: er geldt ook HC2 $\leq_{\mathrm{P}}$ HC1. Bedenk zelf een eenvoudige reductie.

10.6.3 SAT en 3SAT

We bekijken de volgende twee problemen.

SAT: Gegeven een logische formule ϕ in CNF. Bestaat er een waardering van de in ϕ voorkomende logische variabelen die ϕ True maakt?
3SAT: Gegeven een logische formule ϕ in 3-CNF. Bestaat er een waardering die ϕ True maakt?

Definitie Een logische formule ϕ staat in 3-CNF als ϕ een conjunctie ($\wedge$) van clausules ($\vee$) is, waarbij *elke clausule* bestaat uit *drie verschillende literals*.

Er geldt: SAT $\leq_{\mathrm{P}}$ 3SAT

Hint bij het bewijs (de rest op college):
Transformeer een ϕ (waarbij we er vanuit gaan dat de $l_1, l_2, \ldots, l_k$ al verschillend zijn) in CNF naar een ϕ' in 3-CNF volgens:

$l_1 \Longrightarrow (l_1 \vee \widetilde{l_2} \vee \widetilde{l_3}) \wedge (l_1 \vee \widetilde{l_2} \vee \neg\widetilde{l_3}) \wedge (l_1 \vee \neg\widetilde{l_2} \vee \widetilde{l_3}) \wedge (l_1 \vee \neg\widetilde{l_2} \vee \neg\widetilde{l_3})$
$l_1 \vee l_2 \Longrightarrow (l_1 \vee l_2 \vee \widetilde{l_3}) \wedge (l_1 \vee l_2 \vee \neg\widetilde{l_3})$
$l_1 \vee l_2 \vee l_3 \Longrightarrow l_1 \vee l_2 \vee l_3$
$l_1 \vee l_2 \vee l_3 \vee l_4 \Longrightarrow (l_1 \vee l_2 \vee \widetilde{l_5}) \wedge (l_3 \vee l_4 \vee \neg\widetilde{l_5})$
$l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5 \Longrightarrow (l_1 \vee l_2 \vee \widetilde{l_6}) \wedge (l_3 \vee \neg\widetilde{l_6} \vee \widetilde{l_7}) \wedge (l_4 \vee l_5 \vee \neg\widetilde{l_7})$

En in het algemeen voor $k \geq 4$:

$l_1 \vee l_2 \vee \ldots \vee l_{k-1} \vee l_k \Longrightarrow (l_1 \vee l_2 \vee \widetilde{l_{k+1}}) \wedge (l_3 \vee \neg\widetilde{l_{k+1}} \vee \widetilde{l_{k+2}}) \wedge (l_4 \vee \neg\widetilde{l_{k+2}} \vee \widetilde{l_{k+3}}) \wedge \ldots \wedge (l_{k-2} \vee \neg\widetilde{l_{2k-4}} \vee \widetilde{l_{2k-3}}) \wedge (l_{k-1} \vee l_k \vee \neg\widetilde{l_{2k-3}})$

Hierin zijn $\widetilde{l_{k+1}}, \widetilde{l_{k+2}}, \ldots, \widetilde{l_{2k-2}}$ steeds nieuwe, frisse, logische variabelen.
Een clausule met k (verschillende) literals wordt zo getransformeerd in een conjunctie van $k-2$ clausules met elk 3 verschillende literals.
Noem deze transformatie even T. T is gedefinieerd op een clausule, maar kan worden uitgebreid tot formules. Het beeld van een conjunctie van clausules onder T definiëren we als een conjunctie van de beelden van de samenstellende clausules:

$$\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m \longrightarrow T(C_1) \wedge T(C_2) \wedge \ldots \wedge T(C_m) = T(\phi)$$

Voor deze T geldt:

- T kan met een polynomiaal begrensd algoritme berekend worden.
- ϕ is een ja-instantie van SAT $\Longleftrightarrow$ $T(\phi)$ is een ja-instantie van 3SAT.
- Ofwel: er is een waardering die ϕ waarmaakt $\Longleftrightarrow$ er is een waardering die $T(\phi)$ waarmaakt.
- Conclusie uit de vorige punten: SAT $\leq_{\mathrm{P}}$ 3SAT.

Voor verdere uitwerking zie het college.
Aangezien bovendien geldt dat SAT $\in \mathcal{NPC}$ (Cook) en 3SAT $\in \mathcal{NP}$ weten we nu dat 3SAT NP-volledig is.
Overigens geldt ook: 3SAT $\leq_{\mathrm{P}}$ SAT. Bedenk zelf het (eenvoudige) bewijs.

10.6.4 3SAT en Kliek

We bekijken de volgende twee problemen.

3SAT: Gegeven een logische formule ϕ in 3-CNF. Bestaat er een waardering die ϕ True maakt?
Kliek: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$ en een geheel getal k $(0 \leq k \leq |V|)$. Bestaat er in $\mathcal{G}$ een kliek ter grootte ten minste k? (Equivalent: $= k$.)

Er geldt: 3SAT $\leq_{\mathrm{P}}$ Kliek

Hint bij het bewijs:
Zij ϕ een logische expressie (formule) in 3-CNF, met zeg m clausules: $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$.
Laat hierbij $C_r = l_1^r \vee l_2^r \vee l_3^r$ $(r = 1, \ldots, m)$ en l_1^r, l_2^r en l_3^r steeds verschillend bij vaste r.
We construeren nu een ongerichte graaf $\mathcal{G}_\phi = (V, E)$ als volgt: Voor elke clausule C_r uit ϕ doen we 3 knopen v_1^r, v_2^r en v_3^r in V (deze corresponderen dus met l_1^r, l_2^r en l_3^r). $\mathcal{G}_\phi$ heeft derhalve $3m$ knopen. Er komt een tak tussen twee knopen v_i^r en v_j^s als:

. v_i^r en v_j^s in verschillende drietallen zitten (dus als $r \neq s$), en

. de bijbehorende l_i^r en l_j^s zó zijn dat $l_i^r \neq \neg l_j^s$, met andere woorden: l_i^r en l_j^s zijn niet elkaars negatie

Definieer nu de transformatie T als: $T(\phi) =< \mathcal{G}_\phi, m >$. Dan geldt:

- T kan in polynomiaal begrensde tijd berekend worden.

- ϕ is een ja-instantie van 3SAT $\Longleftrightarrow$ $T(\phi)$ is een ja-instantie van Kliek.

- Ofwel: er is een waardering die ϕ waarmaakt $\Longleftrightarrow$ $\mathcal{G}_\phi$ heeft een kliek ter grootte (minstens) m. (We zullen bewijzen dat $\mathcal{G}_\phi$ een kliek ter grootte $= m$ heeft.)

- Conclusie uit de vorige punten: 3SAT $\leq_{\mathrm{P}}$ Kliek.

Bewijs:
(1) De transformatie T kan in polynomiaal begrensde tijd uitgevoerd worden.
Immers: gegeven een logische formule ϕ. Een keer ϕ doorlopen levert het aantal clausules m: $O(|\phi|)$.
Vervolgens nog $\mathcal{G}_\phi$ uit ϕ bepalen. Construeer bijvoorbeeld eerst de knopen en dan de takken. Loop door ϕ heen en maak per clausule drie nieuwe knopen (die corresponderen met de drie literals): $O(|\phi|)$.
Dan de takken. Loop door ϕ heen en kijk voor elke literal "naar" welke andere knopen er een tak moet komen (daartoe nogmaals ϕ doorlopen en naar alle knopen corresponderend met literals die niet in dezelfde clausule zitten en die ook niet de negatie zijn van die

literal, een tak aanleggen: $O(|\phi|^2)$.
In totaal kan de constructie dus polynomiaal in $|\phi|$
(2) Er is een waardering die ϕ waarmaakt $\Longleftrightarrow$ $\mathcal{G}_\phi$ heeft een kliek ter grootte m (= het aantal clausules van ϕ).
"$\Longrightarrow$": Stel w is een waarheidstoekenning (van de in ϕ voorkomende logische variabelen) die ϕ waarmaakt. Dan bevat elke clausule C_r ten minste één literal die True is onder w. Zo'n l_i^r correspondeert met een knoop v_i^r in $\mathcal{G}_\phi$. Kies nu uit elke clausule C_r zo'n waargemaakte literal. Dit correspondeert dan in $\mathcal{G}_\phi$ met een verzameling $V' \subseteq V$ met precies m knopen.
Bewering: V' is een kliek.
Neem daartoe een willekeurig tweetal knopen uit V'. Dit zijn altijd knopen uit verschillende drietallen omdat V' bestaat uit knopen die corresponderen met literals uit verschillende clausules. Dus, zeg, v_i^r en v_j^s (met $r \neq s$). Deze knopen corresponderen met literals l_i^r en l_j^s die beide waargemaakt worden door w. Dus l_i^r en l_j^s kunnen nooit elkaars negatie zijn. Er zit dus een tak tussen v_i^r en v_j^s.
Conclusie: tussen elk tweetal knopen uit V' zit een tak.
"$\Longleftarrow$": Stel nu dat $\mathcal{G}_\phi$ een kliek V'' ter grootte m bevat. Omdat in een kliek elk tweetal knopen door een tak verbonden is kan V'' geen knopen uit hetzelfde drietal bevatten. Dus V'' bevat precies één knoop per drietal (er zijn immers precies m drietallen), en dat correspondeert met één literal per clausule in ϕ. Kies nu een waardering w van de in ϕ voorkomende logische variabelen, die precies True oplevert op de literals die corresponderen met de knopen uit V''. (De waarheidswaarden van de overige logische variabelen mogen naar believen worden ingevuld.) Merk op dat dit inderdaad goed gedefinieerd is (w is consistent). Immers tussen twee knopen die corresponderen met literals die elkaars negatie zijn zit in $\mathcal{G}_\phi$ nooit een tak, dus zo'n tweetal kan nooit in V'' zitten, dus w zal nooit op één variabele zowel True als False worden in bovenstaande definitie van w.
Deze waardering w maakt elke clausule van ϕ waar, en dus ook ϕ zelf.

We hebben nu:

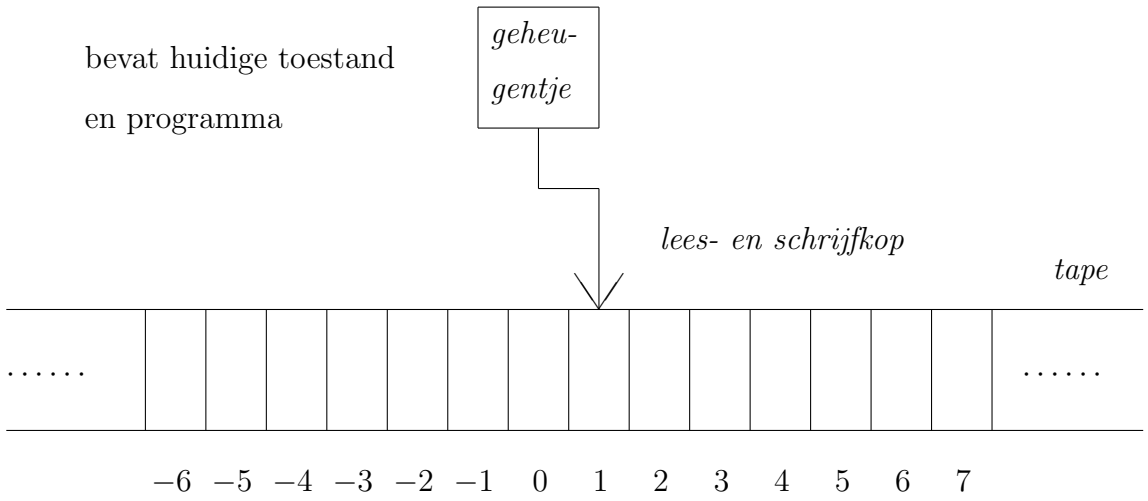
> 3SAT $\leq_{\mathrm{P}}$ Kliek.
> 3SAT $\in \mathcal{NPC}$ (dit volgde uit SAT $\leq_{\mathrm{P}}$ 3SAT en SAT $\in \mathcal{NPC}$ volgens Cook).

Gevolg: Kliek is NP-hard.
Uit de opgaven weten we verder: Kliek $\in \mathcal{NP}$. Samen betekent dit: Kliek $\in \mathcal{NPC}$.

11 Turingmachines

Een Turingmachine is een vereenvoudigd, theoretisch model van een computer, in 1936 bedacht door Alan Turing, een van de grondleggers van de zgn. *berekenbaarheidstheorie*. Een Turingmachine is net zo krachtig als bestaande computers: elk probleem dat met een echte computer kan worden opgelost kan ook met een Turingmachine opgelost worden (en omgekeerd). Wij bekijken hier deterministische Turingmachines voor beslissingsproblemen.

Een deterministische one-tape Turing machine (DTM) heeft een onbegrensde opslagcapaciteit in de vorm van een oneindige tape, met genummerde geheugenplaatsen.In elke geheugenplaats past precies één symbool uit een tevoren gedefiniëerd alfabet. Verder heeft de Turing machine een lees- en schrijfkop die vooruit en achteruit langs de tape kan worden bewogen, en waarmee symbolen kunnen worden gelezen, geschreven of uitgewist. De Turing machine bevindt zich steeds in één van een eindig aantal toestanden Q.



Een DTM-programma ziet er dan als volgt uit. Het bevat enkele definities en het eigenlijke programma (beschreven middels een transitiefunctie).

- Γ: een eindige verzameling *tape-symbolen* (waaronder inputsymbolen Σ en blanco). Voorbeeld: $\Gamma = \{0, 1, b\}$.
- Q: een eindige verzameling *toestanden*, waaronder een begintoestand q_0 en twee eindtoestanden q_Y en q_N. Voorbeeld: $Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$.
- $\delta : Q - \{q_Y, q_N\} \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 0, 1\}$ een *transitiefunctie* die bepaalt wat er gebeurt als in een bepaalde toestand een bepaald karakter wordt gelezen.

In de begintoestand staat de invoerstring x op plek 1 t/m $|x|$ en de rest van de tape is blanco. Het programma start in toestand q_0 met de lees- en schrijfkop op positie 1 en stopt als de toestand q_y (yes) of q_N (no) bereikt wordt. Op elk moment van de berekening

bevindt de DTM zich in een bepaalde toestand en wordt een symbool van de tape gelezen. De volgende stap wordt bepaald door de transitiefunctie.
De transitiefunctie (en dus in feite het DTM-programma) kan behalve via een tabel (zoals in het voorbeeld hieronder), ook worden weergegeven met behulp van een toestandsdiagram (zie het derdejaarsvak Fundamentele informatica 3).

Voorbeeld: het volgende DTM-programma gaat na of een gegeven invoerstring op 00 eindigt (en geeft als uitvoerstring overigens de oorspronkelijke string met de laatste twee posities blanco). Zie verder het college.

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

$$Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$$

En de transitiefunctie:

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

$\delta(q, s)$

12 Opgaven

Opgave 1.
Bekijk het volgende probleem: gegeven een reëel getal x en een geheel positief getal n, gevraagd x^n.
a. Geef de complexiteit van het meest voor de hand liggende algoritme hiervoor. Wat is de basisoperatie in dit geval en waarom?
b. Idem, nu voor het algoritme dat x herhaald kwadrateert en het antwoord uit de tussenresultaten samenstelt: $x^{13} = x^8 * x^4 * x$, waarbij overigens x^2 niet gebruikt wordt.
c. Is het antwoord van **b.** optimaal? Hint: bekijk x^{15}.

Opgave 2.
Deze opgave gaat over het vermenigvuldigen van matrices.
a. Bereken het matrixproduct

$$\begin{pmatrix} 1 & 3 \\ 5 & 7 \end{pmatrix} \begin{pmatrix} 8 & 4 \\ 6 & 2 \end{pmatrix}$$

op de gebruikelijke manier (zie ook hieronder). Hoeveel $*$'s en hoeveel $+/-$'s heb je hiervoor nodig gehad?

Nu het algemene geval: gegeven twee $n \times n$-matrices A en B (met reële getallen). Bepaal het matrixproduct van deze twee matrices. Bekijk het "gebruikelijke" algoritme: dit algoritme berekent de componenten c_{ij} van de productmatrix AB via de formule $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ voor $1 \leq i, j \leq n$.

b. Hoeveel vermenigvuldigingen $(*)$ en hoeveel optellingen $(+/-)$ doet het "gebruikelijke" algoritme, uitgedrukt in n?
c. Dezelfde vraag, maar nu uitgedrukt in het aantal array-elementen, ook een mogelijke maat voor de grootte van de invoer.
d. Een vermenigvuldiging is i.h.a. tijdrovender dan een optelling. Dus er wordt voorgesteld om als basisoperatie het vermenigvuldigen van twee getallen te nemen. Is dit inderdaad altijd een goede maat voor de complexiteit? Hint: denk aan het geval dat de matrices allemaal gehele getallen bevatten.
e. Laat zien: de *complexiteit* van elk algoritme voor het vermenigvuldigen van twee n bij n matrices is altijd minstens $\Omega(n^2)$.
f. Er zit een ordeverschil tussen de complexiteit van het "gebruikelijke" algoritme en de ondergrens uit **e.** Betekent dit dat het algoritme niet optimaal is of dat de ondergrens niet scherp genoeg is?

Opgave 3.
Gegeven een ongeordend array A ($A[1]$, ..., $A[n]$) dat n gehele getallen bevat. Gegeven is tevens een geheel getal X. Onderstaand algoritme spoort X op in het array via *lineair zoeken*. Als X in het array zit wordt de index opgeleverd van de/een plek waar X staat, anders -1.

```
(1)  index := 1;
(2)  while index ≤ n and A[index] ≠ X do
(3)      index := index + 1;
(4)  od
(5)  if index ≤ n then
(6)      return index;
(7)  else
(8)      return −1;
(9)  fi
```

a. Waarom is het vergelijken van X met array-elementen ($A[index] \neq X$) een goede basisoperatie?
b. Wat is het aantal vergelijkingen in het beste geval en voor welke invoer komt dat voor?
c. Wat is het aantal vergelijkingen in het slechtste geval en voor welke invoer komt dat voor?
d. Wat is het aantal vergelijkingen in het gemiddelde geval (average case)? Neem hierbij aan dat alle elementen verschillend zijn (dus als X in A voorkomt zijn alle posities in het array even waarschijnlijk).
e. Laat zien dat het algoritme optimaal is (in de worst case) als we alleen kijken naar algoritmen die uitsluitend gebaseerd zijn op het doen van vergelijkingen tussen X en array-elementen (sleutelvergelijkingen).
* **f.** Het moge duidelijk zijn dat je ook informatie kan krijgen over de positie van X door behalve vergelijkingen van de vorm $A[index] \neq X$ ook vergelijkingen tussen array-elementen onderling te doen. Laat zien dat elk algoritme gebaseerd op deze twee soorten vergelijkingen altijd ten minste n vergelijkingen in totaal nodig heeft in de worst case. Ons algoritme is dus ook optimaal binnen deze grotere klasse van algoritmen.

Opgave 4.
Gegeven een oplopend gesorteerd array A ($A[1]$, ..., $A[n]$) dat n gehele getallen bevat. Gegeven is verder een geheel getal X. We gebruiken een aangepaste versie van het zoekalgoritme uit de vorige opgave. Net als daar is het vergelijken van X met array-elementen ($A[index] < X$) een goede basisoperatie.

```
(1)  index := 1;
(2)  while index ≤ n and A[index] < X do
(3)      index := index + 1;
(4)  od
(5)  if index ≤ n and X = A[index] then
(6)      return index;
(7)  else
(8)      return −1;
(9)  fi
```

a. Wat is het aantal vergelijkingen in het beste geval en voor welke invoer komt dat voor?
b. Wat is het aantal vergelijkingen in het slechtste geval en voor welke invoer komt dat voor? Vergelijk dit met ongeordend zoeken (opgave 3).
c. Wat is het aantal vergelijkingen in het gemiddelde geval? Laat ook zien dat dit $\Theta(\frac{n}{2})$ is. Neem hierbij aan dat (1) als X in A voorkomt alle posities in het array even waarschijnlijk

zijn en (2) als X niet in A zit alle "gaten" tussen de opeenvolgende array-elementen even waarschijnlijk zijn. Laat q de kans zijn dat X in A zit.

Opgave 5. *Binair zoeken.*
Bekijk het volgende algoritme dat X opspoort in een oplopend gesorteerd array A met $n \geq 1$ elementen:

```
Links := 1; Rechts := n;
while Links ≤ Rechts do
    Midden := ⌊(Links+Rechts)/2⌋;
    if X = A[Midden] then
        return Midden;
    else
        if X < A[Midden] then
            Rechts := Midden − 1
        else
            Links := Midden + 1
        fi
    fi
od
return −1;
```

a. Analyseer het beste en het slechtste geval. Zij $W(n)$ = het aantal 3-weg-vergelijkingen van de vorm $X =, < A[Midden]$ dat het algoritme doet in het slechtste geval.
Leid hiervoor de recurrente betrekking $W(n) = 1 + W(\lfloor n/2 \rfloor)$ af.
b. Laat zien dat de oplossing van deze recurrente betrekking, met randvoorwaarde $W(1) = 1$, gegeven wordt door: $W(n) = \lceil \lg(n+1) \rceil$. Gebruik inductie.
Opmerking: voor $n = 2^k - 1$ is direct in te zien dat $W(n) = k = \lg(n+1)$.
c. Analyseer het gemiddelde geval. Neem aan dat n een 2-macht min 1 is, en neem aan dat alle n posities in het array *en* alle $n+1$ "gaten daartussen" even waarschijnlijk zijn. Gebruik hierbij onder andere dat $\Sigma_{t=0}^{k-1}(t+1)2^t = (k-1)2^k + 1$.
d. Geef een beslissingsboomargument om aan te tonen dat het algoritme in het slechtste geval optimaal is.
e. Stel dat we binair zoeken en lineair zoeken toepassen op een *enkelverbonden lijst*. Is binair zoeken dan nog steeds efficiënter dan lineair zoeken?

Opgave 6. *Bubblesort.*
Bekijk het volgende algoritme om een array A met n gehele getallen op plekken 1 t/m n oplopend te sorteren:

```
(1)  Boven := n; Gewisseld := True;
(2)  while Gewisseld do
(3)      Boven := Boven − 1; Gewisseld := False;
(4)      for j := 1 to Boven do
(5)          if A[j] > A[j + 1] then
(6)              Wissel(A[j], A[j + 1]); Gewisseld := True fi od
(7)  od
```

Hierbij is *Wissel* uiteraard een functie die de inhoud van zijn argumenten verwisselt.

a. Wat is hier de worst case, en hoeveel arrayvergelijkingen $A[j] > A[j+1]$ doet Bubblesort in dit geval? Neem aan dat het array een permutatie van $\{1, 2, \ldots, n\}$ bevat.
b. Idem voor de best case.
c. Is het aantal arrayvergelijkingen een goede basisoperatie? Waarom (niet)?
d. We passen het algoritme aan door onder- en bovengrens van de for-loop variabel(er) te maken, en correct te updaten. Het idee is om te lopen vanaf de plek waar in de vorige ronde voor het eerst gewisseld werd (*Onder*) tot de plek waar in de vorige ronde voor het laatst gewisseld werd (*Boven*). Veranderen best case en worst case hiermee (essentieel)?

Opgave 7. *Insertion sort.*

```
(1)   for i := 2 to n do
          // nu A[i] op de juiste plek in A[1] ... A[i − 1] invoegen
(2)       x := A[i];
(3)       j := i − 1;
(4)       while j > 0 and A[j] > x do
(5)           A[j + 1] := A[j];
(6)           j := j − 1;
(7)       od
(8)       A[j + 1] := x;
(9)   od
```

a. Laat zien dat het aantal arrayvergelijkingen (tweede test regel (4)) een goede maat is voor de complexiteit.
b. Hoeveel arrayvergelijkingen doet Insertion sort in de worst case?
c. Geef *alle* worst case gevallen voor Insertion sort (dus niet alleen de omgekeerd gesorteerde rij). Hoeveel zijn dat er? Zonder beperking der algemeenheid kun je volstaan met alleen invoerrijtjes met n verschillende waarden, zeg 1 t/m n te bekijken.

Opgave 8.
We bekijken het "handige" algoritme *Slowsort*. Het sorteert een array met gehele getallen van klein naar groot. De getallen bevinden zich op de plaatsen 1 t/m n.

```
(1)   Min := 1
(2)   while Min ≤ n do
(3)       Kmin := Min + 1;
(4)       while Kmin ≤ n and A[Min] ≤ A[Kmin] do
(5)           Kmin := Kmin + 1;
(6)       od
(7)       if Kmin ≤ n then
(8)           Wissel(A[Min], A[Kmin]);
(9)       else
(10)          Min := Min + 1;
(11)      fi
(12)  od
```

Het doen van arrayvergelijkingen (de tweede test in regel 4) is hier een goede basisoperatie. (Eenvoudig is in te zien dat de *hele* test uit regel 4 een goede maat voor de complexiteit is. Je kunt echter ook aantonen dat het eerste deel van die test in orde van grootte even vaak gebeurt als het tweede deel.)
a. Hoeveel arrayvergelijkingen doet het algoritme in het beste geval (best case) en voor welke invoer is dat?
b. Als **a**, maar nu in het slechtste geval (worst case).

Opgave 9.
Gegeven twee algoritmen, A en B, die hetzelfde probleem oplossen. Veronderstel dat algoritme A $f(n) = n^2 + 4\,n$ stappen in het slechtste geval doet en algoritme B $g(n) = 29\,n + 3$, voor invoer ter grootte n. Welk algoritme is nu beter in de worst case?
Dezelfde vraag, maar nu doet algoritme A $5\,n^2 + n \lg n + 3$ stappen in de worst case en algoritme B $2\,n^2 + n - 4$.

Opgave 10.
Orden de volgende functies met behulp van de O/Θ-notatie:

$\sqrt{n}$	n^2	$n!$	$n^{1+\varepsilon}$ met $\varepsilon > 0$	$(3/2)^n$	n^3	$\lg^2 n$
$\lg(n!)$	2^{2^n}	$n + 9$	$\ln \ln n$	$n\,2^n$	$n^2 + \lg n$	$\ln n$
1	$2^{\lg n}$	$\lg n$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
n	2^n	$n \lg n$	$2^{2^{n+1}}$	$n^3 + n^7 + 8$	$1/n$	$\log_3 n$

Opgave 11.
Toon aan dat $\sum_{i=1}^{n} \frac{1}{i} \in O(\lg n)$.

Opgave 12.
Laat zien: $\sum_{i=0}^{k-1} (i+1)\,2^i = (k-1)\,2^k + 1$.

Opgave 13.
Bewijs dat voor gehele $n \geq 1$ geldt:

$$\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$$

Hint: Voor elke n geldt dat $2^k \leq n \leq 2^{k+1} - 1$ voor een of andere gehele $k \geq 0$.

Opgave 14.
Los de recurrente betrekkingen uit **a.** t/m **g.** exact op.

a. $T(n) = \begin{cases} 3 & n = 1 \\ T(n-1) + n - 1 & n > 1 \end{cases}$

b. $T(n) = \begin{cases} 1 & n = 1 \\ 2\,T(n-1) + 1 & n > 1 \end{cases}$

c. $T(n) = \begin{cases} 1 & n = 1 \\ 2\,T(\frac{n}{2}) + n & n > 1, n = 2^k, k > 0 \end{cases}$

*__d.__ $T(n) = \begin{cases} 2 & n = 2 \\ \sqrt{n}\,T(\sqrt{n}\,) + c\,n & n > 2 \end{cases}$

voor n is 2 tot de macht een macht van 2: 2, 4, 16, 256, ...; c is een positieve constante.

__e.__ $T(n) = \begin{cases} 1 & n = 0 \\ (n+1)\,T(n-1) & n > 0 \end{cases}$

Wat wordt de oplossing als de recurrente betrekking geldt voor $n > 1$ en we als beginwaarde $T(1) = 1$ hebben?

__f.__ $T(n) = \begin{cases} 0 & n = 1 \\ 3\,T(\frac{n}{3}\,) + 2 & n > 1, n = 3^k, k > 0 \end{cases}$

__g.__ $T(n) = \begin{cases} 0 & n = 1 \\ 2\,T(\frac{n}{4}\,) + \sqrt{n} & n > 1, n = 4^k, k > 0 \end{cases}$

__h.__ Bewijs met volledige inductie dat voor $T(n)$, de oplossing van onderstaande recurrente betrekking,

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(\lfloor \frac{n}{2} \rfloor\,) + n & n > 1 \end{cases}$$

geldt dat $T(n) \leq 2n \lg n$ voor alle $n \geq 2$. Zie ook college.

Opgave 15.
We bekijken het volgende *recursieve* algoritme om de grootste en de op een na grootste waarde te vinden uit een rij van n getallen met $n = 2^k$ (k geheel, ≥ 1).
Als $n \geq 4$ bepalen we eerst recursief de grootste en de op een na grootste van de eerste $n/2$ elementen, vervolgens die uit de resterende $n/2$. Daarna moeten we hieruit nog de grootste en de op een na grootste van de gehele rij afleiden. Noem het aantal (array)vergelijkingen dat dit algoritme doet $G(n)$.

__a.__ Leg duidelijk uit waarom $G(n)$ voldoet aan de volgende recurrente betrekking:

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

__b.__ Los de recurrente betrekking uit __a.__ op door deze herhaald in zichzelf in te vullen en bewijs met behulp van volledige inductie dat de aldus gevonden oplossing inderdaad voldoet. Schrijf G als functie van n. *Hint*: gebruik dat $\sum_{i=1}^{l-1} 2^i = 2^l - 2$.

Opgave 16.
Bekijk het volgende *recursieve* algoritme voor het aflopend sorteren van een rij van n verschillende getallen met $n \geq 0$ en n even.
Als $n \geq 2$ bepalen we eerst de grootste en de kleinste waarde door twee keer het hele array langs te lopen, en verwisselen we die met resp. het voorste en het achterste array-element. Nu staan $A[1]$ en $A[n]$ dus goed. Daarna sorteren we *recursief* de rest van het array. Noem het aantal vergelijkingen dat dit algoritme doet $S(n)$.

a. Leg duidelijk uit waarom $S(n)$ voldoet aan de volgende recurrente betrekking:

$$S(n) = \begin{cases} 0 & n = 0 \\ S(n-2) + 2n - 3 & n \geq 2,\ n \text{ even} \end{cases}$$

b. Los de recurrente betrekking uit **a.** op door deze herhaald in zichzelf in te vullen en bewijs met behulp van volledige inductie dat de aldus gevonden oplossing inderdaad voldoet. Laat bij het herhaald invullen de termen met n, $n-2$, $n-4$, etc. gewoon staan en veeg de 3-en bij elkaar. Dan is de algemene vorm beter te herkennen. Gebruik ten slotte dat $S(0) = 0$.
Hint bij het uitrekenen: gebruik –met even n– een van de volgende sommaties:

$$\sum_{i=1}^{\frac{n}{2}} i \;=\; \frac{n}{4} \cdot (\frac{n}{2} + 1); \quad \sum_{i=2,\ i \text{ even}}^{n} i \;=\; \frac{n}{2} \cdot (\frac{n}{2} + 1)$$

Opgave 17.
Bekijk het onderstaande voor de hand liggende algoritme om het maximum van n getallen (opgeslagen in een array A op de plekken 1 t/m n) op te sporen:

```
(1)   max := A[1]; i := 2;
(2)   while i <= n do
(3)       if max < A[i] then
(4)           max := A[i];
(5)       fi
(6)       i := i + 1;
(7)   od
(8)   return max;
```

Het aantal arrayvergelijkingen is altijd hetzelfde (namelijk $n-1$), maar het aantal toekenningen van de vorm $max := A[..]$; (regel 1 en regel 4) hangt af van het invoerrijtje. We gaan hier het aantal van dit soort toekenningen bekijken .

a. Hoeveel toekenningen doet het algoritme in de best case en voor wat voor invoerrijtjes komt dat voor?
b. Dezelfde vraag, maar nu voor de worst case.

Opgave 18.
Gegeven een array A ($A[1], A[2], \ldots, A[n]$ met $n \geq 3$). Een element uit het array heet een *major* als het *meer* dan ($>$) $n/2$ keer in het array voorkomt (voor even n is dat dus $\geq n/2 + 1$ keer, voor oneven n is dat $\geq \lceil n/2 \rceil$ keer). Merk op: er is altijd hooguit één *major*.
We bekijken hieronder een algoritme dat die *major* oplevert als deze bestaat (*found* is dan True). Als er geen *major* is blijft de boolean *found* False. Het hulparray *gehad* geeft per index aan of het getal A[index] reeds is bekeken.
Het algoritme gaat als volgt te werk. Slechts de eerste $\lceil n/2 \rceil$ elementen worden doorlopen; dit is voldoende, want als de *major* dan nog niet gevonden is, bestaat die ook niet. Voor elke $A[i]$ wordt geteld hoe vaak die waarde in A voorkomt. Voor elk voorkomen van $A[i]$

wordt daarbij op de overeenkomstige plek in *gehad* de waarde op True gezet, met als gevolg dat deze verder overgeslagen wordt in regel (8) t/m (20). Het algoritme stopt zodra de *major* gevonden is.

```
(1)   i := 1;
(2)   while i ≤ n do
(3)       gehad[i] := False; i := i + 1;
(4)   od
(5)   found := False;
(6)   i := 1;
(7)   while i ≤ ⌈n/2⌉ and not found do
(8)       if not gehad[i] then
(9)           gehad[i] := True; tel := 1;
(10)          j := i + 1;
(11)          while j ≤ n do
(12)              if A[j] = A[i] then
(13)                  tel := tel + 1; gehad[j] := True;
(14)              fi
(15)              j := j + 1;
(16)          od
(17)          if tel > n/2 then
(18)              found := True; major := A[i];
(19)          fi
(20)      fi
(21)      i := i + 1;
(22)  od
```

a. Leg uit waarom het aantal *vergelijkingen* tussen array-elementen uit regel (12) een goede basisoperatie is. Laat hierbij onder meer expliciet zien dat deze vergelijking ten minste $n-1$ keer gebeurt en dat het aantal keer dat de test in regel (8) wordt uitgevoerd kleiner of gelijk is aan het aantal arrayvergelijkingen in regel (12).
b. Laat zien hoe het algoritme werkt op het voorbeeldrijtje: 3, 4, 3, 4, 5, 5, 3, 5, 5, 5, 5. Hoeveel vergelijkingen doet het algoritme op deze invoer?
c. Hoeveel vergelijkingen tussen array-elementen doet het algoritme in het slechtste geval en voor wat voor invoerrijtjes komt dat voor?
d. Idem, maar nu in het beste geval.
e. Voeg in bovenstaand algoritme tussen regel (11) en (12) de test **if not** *gehad*[*j*] **then** in. Beantwoord nu dezelfde vraag als **c.** voor het aangepaste algoritme. Is de test uit regel (12) nog steeds een goede maat voor de complexiteit van het algoritme?

Opgave 19.
In deze opgave bekijken we een algoritme dat de grootste en de kleinste waarde uit een array A (met entries $A[1]$ t/m $A[n]$ en n even, $n \geq 2$) bepaalt.
Hiertoe worden eerst de array-elementen twee aan twee vergeleken en worden de grootste waarden op de even posities gezet en de kleinste op de oneven posities. Vervolgens wordt de kleinste waarde uit de $n/2$ oneven posities gevonden, en de grootste uit de $n/2$ even posities. De functie *Wissel* verwisselt de waarden van zijn argumenten.
Het algoritme gaat als volgt:

```
(1)   i := 1;
(2)   while i < n do
(3)       if A[i] > A[i + 1] then
(4)           Wissel(A[i], A[i + 1]);
(5)       fi
(6)       i := i + 2;
(7)   od
(8)   i := 1; min := A[1];
(9)   while i ≤ n do
(10)      if A[i] < min then
(11)          min := A[i];
(12)      fi
(13)      i := i + 2;
(14)  od
(15)  i := 2; max := A[2];
(16)  while i ≤ n do
(17)      if A[i] > max then
(18)          max := A[i];
(19)      fi
(20)      i := i + 2;
(21)  od
```

a. Laat zien dat het totaal aantal arrayvergelijkingen (regel 3, regel 10 en regel 17 samen) een goede basisoperatie is. Hoeveel van zulke vergelijkingen doet het algoritme?
b. We bekijken nu het aantal toekenningen uit de regels 4, 8, 11, 15 en 18 samen. Hoeveel van zulke toekenningen doet het algoritme in het beste geval en voor welke invoerrijtjes? Dezelfde vraag voor de worst case. Neem aan dat de functie *Wissel* 3 toekenningen doet.
c. Pas het algoritme aan zodat het ook voor oneven n werkt. Hoeveel arrayvergelijkingen doet het algoritme voor algemene n? Beantwoord ook vraag **b.** voor het aangepaste algoritme.

Opgave 20.
Gegeven een array A ($A[1], A[2], \ldots, A[n]$, met n een geheel getal > 1). Onderstaand algoritme "sorteert" A zodanig dat na afloop alle even elementen van A *vooraan* staan en de oneven elementen van A *achteraan*.

```
(1)   i := 1; j := n;
(2)   while i < j do
(3)       if (A[i] mod 2 = 0) then
(4)           i := i + 1;
(5)       else
(6)           if (A[j] mod 2 = 0) then
(7)               wissel(A[i], A[j]);
(8)               i := i + 1;
(9)           fi
(10)          j := j − 1;
(11)      fi
(12)  od
```

De functie *wissel* verwisselt uiteraard de waarden van zijn argumenten. Verder betekent $A[i]$ **mod** 2: de rest bij deling van $A[i]$ door 2.

We bekijken drie mogelijke operaties die in aanmerking komen om de complexiteit van het algoritme mee te beschrijven:
(I) het vergelijken van i en j in regel (2)
(II) het vaststellen of een array-element even of oneven is in regel (3) en regel (6)
(III) het verwisselen van array-elementen in regel (7)

a. Geef van elk van deze drie operaties aan of ze de complexiteit van het algoritme goed beschrijven of niet. Zo ja, schat dan voor elke regel (die met **fi** of **od** uiteraard niet) het aantal keren dat deze wordt uitgevoerd af op het aantal keren dat de betreffende operatie ((I) of (II) of (III)) wordt gedaan. Zo nee, motiveer waarom niet.
b. We bekijken operatie (I). Hoe vaak wordt deze operatie gedaan in de *best case* en voor wat voor invoerrijtjes komt dat voor (alle gevallen)? Idem voor de *worst case*. Let op de situatie n even of oneven. Motiveer je antwoord!
c. Dezelfde vraag als **b**, maar nu voor operatie III.
d. Hoe vaak wordt operatie II (regel (3) en regel (6) samen) gedaan in de worst case en voor wat voor invoerrijtjes komt dat voor? Geef alle gevallen.

Opgave 21.
Gegeven is een array A ($A[1], A[2], \ldots, A[n]$) dat $n \geq 2$ verschillende gehele getallen bevat. Onderstaand algoritme sorteert het array oplopend via een variant op Selection sort. Herhaald wordt eerst gecontroleerd of het array al gesorteerd is, en zo niet, wordt de kleinste van $A[i]$ t/m $A[n]$ gezocht en op positie i gezet.

```
(1)   i := 1;  gesorteerd := False;
(2)   while not gesorteerd do
(3)       gesorteerd := True;  j := i;
(4)       while gesorteerd and j ≤ n − 1 do
(5)           if A[j] > A[j + 1] then
(6)               gesorteerd := False;
          fi
(7)           j := j + 1;
          od
(8)       if not gesorteerd then
(9)           kleinste := i;  j := i + 1;
(10)          while j ≤ n
(11)              if A[j] < A[kleinste] then
(12)                  kleinste := j;
                  fi
(13)              j := j + 1;
              od
(14)          temp := A[i];  A[i] := A[kleinste];  A[kleinste] := temp;
          fi
(15)      i := i + 1;
      od
```

Men kan eenvoudig laten zien dat het *vergelijken* van array-elementen in regel (5) en (11) *samen* maatgevend is voor de complexiteit van het algoritme.

a. Leg uit waarom het vergelijken van array-elementen in regel (11) alleen niet maatgevend is voor de complexiteit van het algoritme. Geef een voorbeeldinvoer die dit illustreert.
Opmerking. Uit **e.** zien we dat ook het vergelijken van array-elementen in regel (5) alleen niet maatgevend is voor de complexiteit van het algoritme.
b. Laat zien dat het aantal arrayvergelijkingen dat in regel (5) gedaan wordt altijd ten minste $n - 1$ is.

In de volgende onderdelen moeten telkens de arrayvergelijkingen uit regel (5) en (11) samen worden geteld.

c. Leid uit het algoritme af hoeveel arrayvergelijkingen er worden gedaan in het *beste geval*, voor algemene n. Voor wat voor invoerrijtjes komt dat voor? Geef *alle* gevallen en leg op basis van het algoritme uit hoe je aan je antwoord komt.
d. Leid uit het algoritme af hoeveel arrayvergelijkingen er worden gedaan in het *slechtste geval*, voor algemene n. Voor wat voor invoerrijtjes komt dat voor? Geef *alle* gevallen en leg op basis van het algoritme uit hoe je aan je antwoord komt.
e. Analyseer en leg uit wat er gebeurt als het invoerarray A aflopend gesorteerd is. Geef aan hoeveel vergelijkingen het algoritme in dat geval doet en licht je anwoord toe. Je mag wel aannemen dat n even is.

Opgave 22.
a. Geef een algoritme dat bepaalt of de n getallen in een array A ($A[1]$ t/m $A[n]$) alle verschillend zijn. A hoeft niet gesorteerd te zijn. Gebruik alleen vergelijkingen van de vorm $A[i] = A[j]$. Hoeveel van dit soort vergelijkingen doet je algoritme in de best case en hoeveel in de worst case?
b. Laat zien dat elk algoritme dat alleen gebaseerd is op vergelijkingen van de vorm $A[i] = A[j]$, ten minste $\frac{1}{2}n(n-1)$ van dit soort vergelijkingen moet doen in de worst case.
c. We bekijken nu algoritmen die ook vergelijkingen van de vorm $A[i] < A[j]$ doen. Geef in woorden een algoritme dat echt minder (dus in orde van grootte) arrayvergelijkingen doet ($<$ en $=$ samen) dan het algoritme uit **a.**

Opgave 23.
a. Geef een algoritme om de mediaan (de middelste in grootte) van drie verschillende gehele getallen a, b en c te vinden.
b. Breng de mogelijke invoeren onder in verschillende "klassen".
c. Hoeveel vergelijkingen doet het algoritme in het slechtste geval? En in het beste geval? En in het gemiddelde geval (geef ook aan wat "gemiddeld" betekent)?
d. Hoeveel vergelijkingen zijn in het slechtste geval echt nodig? Hierbij gaat het om een willekeurig algoritme.

Opgave 24.
Geef een methode (algoritme) om de mediaan van 5 sleutels met maximaal 6 vergelijkingen te vinden.

Opgave 25.
a. Geef een methode (algoritme) waarmee 4 sleutels met maximaal 5 vergelijkingen gesorteerd kunnen worden. Merk op dat de ondergrens voor sorteren hier gehaald wordt.
b. Dezelfde vraag, maar nu moeten 5 sleutels met maximaal 7 vergelijkingen gesorteerd kunnen worden.

Opgave 26.
Toon met een eenvoudig (ad-hoc) argument aan: elk algoritme voor het selectieprobleem dat gebaseerd is op arrayvergelijkingen, doet in de worst case ten minste $\lceil \frac{n}{2} \rceil$ vergelijkingen. Hint: toon aan dat alle array-elementen bekeken (=vergeleken met een ander array-element) moeten zijn.

Opgave 27.
a. Welke ondergrens op de worst case complexiteit levert een beslissingsboomargument op bij het vinden van het minimum *en* het maximum van een rij met $n > 1$ getallen?
b. Welke ondergrens op de worst case complexiteit levert een beslissingsboomargument op bij het vinden van de vijf grootste waarden uit een rij met n getallen? Merk op dat je niet hoeft te weten welke van die vijf de grootste is en welke de een-na-grootste, etcetera.
c. Welke ondergrens op de worst case complexiteit (hier: aantal arrayvergelijkingen) levert een beslissingsboomargument op voor het samenvoegen van twee *gesorteerde* rijen getallen A en B tot één gesorteerde rij? We bekijken hierbij alleen algoritmen die gebaseerd zijn op het doen van arrayvergelijkingen. Laten verder A en B respectievelijk m en n getallen bevatten. Je antwoord wordt dan een functie van m en n.

Opgave 28.
Gegeven twee oplopend gesorteerde even lange rijen A en B met in totaal $2n$ getallen. Gevraagd wordt het n-de getal (in volgorde van klein naar groot) van de in totaal $2n$ elementen van A en B.
a. Geef een algoritme dat dit probleem in maximaal n arrayvergelijkingen oplost door een merge toe te passen.
b. Bewijs met behulp van een beslissingsboomargument dat *elk* algoritme dat dit probleem oplost m.b.v. arrayvergelijkingen ten minste $\lceil \lg n \rceil + 1$ vergelijkingen moet doen in de worst case.
c. We zoeken nu de n-de waarde in grootte zoals boven, maar de ene gesorteerde rij bevat $\frac{n}{2}$ elementen en de andere $\frac{3n}{2}$. Net als in **b.** kunnen we een ondergrens voor de worst case bewijzen voor het probleem met dit soort invoerrijtjes. Wat verandert er dan in het bewijs van **b.** en welke (betere) ondergrens levert dat op?

Opgave 29.
In deze opgave bekijken we de implementatie van de *toernooimethode* met behulp van een soort heap (hoopstructuur). Deze heapachtige structuur wordt hier voorgesteld door een array H ter grootte $2n - 1$. De n verschillende invoergetallen bevinden zich in het array A op de posities 1 t/m n.
a. Laat zien dat onderstaand algoritme (VindMax(A,n) geheten) de grootste van de n invoergetallen in $H[1]$ zet.

```
(1)   VindMax(a, n)::
(2)       Zet A[1], ..., A[n] in H[n], ..., H[2n − 1];
(3)       laatste := 2n − 2;
(4)       while laatste ≥ 2 do
(6)           H[laatste/2] := max(H[laatste], H[laatste + 1]);
(7)           laatste := laatste − 2;
(8)       od
```

b. Leg uit hoe de array-elementen die van de winnaar (de grootste) verloren hebben bepaald kunnen worden.
c. Schrijf een stukje programma om, nadat VindMax voltooid is, de op-een-na-grootste te vinden.
d. Hoeveel vergelijkingen doet het programma in totaal? En hoeveel extra geheugenruimte kost deze implementatie?

Opgave 30.
Voer het $O(n)$-algoritme voor het bepalen van de k-de in grootte uit voor $k = 5$ en het invoerrijtje 100, 80, 20, 70, 60, 33, 14, 15, 29, 17, 18, 44, 86, 50, 12.

Opgave 31.
Onderzoek het volgende probleem: bepaal of een string met n bits twee opeenvolgende nullen bevat, met als basisoperatie het bekijken van één bitpositie. Doe de gevallen $n = 2, 3, 4, 5$. Geef *of* een adversary-argument om het bekijken van elke bit af te dwingen, *of* geef een "slimmer" algoritme.

Opgave 32. Bewijs met behulp van een adversary argument dat elk algoritme gebaseerd op arrayvergelijkingen dat het maximum bepaalt van n verschillende array-elementen, in de worst case ten minste $n - 1$ vergelijkingen moet doen.
Hint: je kunt een adversary strategie gebruiken die vrijwel hetzelfde is als degene die gebruikt is voor het probleem van het simultaan vinden van het maximum en het minimum.

Opgave 33.
Bekijk de klasse van graafalgoritmen die alleen vragen stellen van de vorm: "zit er een tak tussen knoop v en knoop w ?". Bewijs met een adversary-argument dat elk algoritme uit die klasse dat checkt of een ongerichte graaf met n knopen samenhangend is, in de worst case altijd ten minste $\lfloor n/2 \rfloor * \lceil n/2 \rceil$ ($= \frac{n^2}{4}$ als n even, en $= \frac{n^2-1}{4}$ als n oneven) van die vragen moet stellen.
Hint: voordat de adversary vragen van het algoritme beantwoordt splitst zij de knoopverzameling in twee disjuncte verzamelingen van respectievelijk $\lfloor n/2 \rfloor$ en $\lceil n/2 \rceil$ knopen.

Opgave 34.
Het tussenvoegen van sleutels kan bij Insertion sort met binair zoeken: zo ontstaat *Binary Insertion sort*. Bekijk het aantal vergelijkingen in de worst case voor $n = 6$ en 7. Hoe zit het met het aantal verschuivingen van array-elementen? Bediscussieer de zo verkregen verbetering/verslechtering.

Opgave 35.
Teken de beslissingsboom voor *Bubblesort* met $n = 3$. Idem voor *Mergesort* met $n = 3$.

Opgave 36.
Het aantal vergelijkingen dat het sorteeralgoritme *Mergesort* doet in de best case wordt gegeven door de volgende recurrente betrekking:

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

Bereken $B(n)$ voor n een tweemacht.

Opgave 37.
Gegeven een array A dat nullen en enen bevat (in totaal n stuks).
a. Geef een algoritme dat A sorteert en daarbij (hooguit) $\Theta(n)$ arrayvergelijkingen doet.
b. Het bestaan van zo'n algoritme is niet in tegenspraak met de stelling uit hoofdstuk 7.3. Waarom niet?
c. Geef een algoritme dat A sorteert en daarbij 0 arrayvergelijkingen doet. Wat is de complexiteit van dit algoritme?
d. Geef een algoritme dat helemaal geen vergelijkingen doet, dus ook geen vergelijkingen van de vorm $A[i] = 1$ of $A[i] = 0$.

Opgave 38.
Heapsort. Bekijk het volgende algoritme, dat een array A met n elementen oplopend sorteert:

```
SiftUp(A, i, j) :
    x := A[i];
    while 2 * i ≤ j do
        g := 2 * i; // zoek grootste kind van i
        if g < j then
            if A[g + 1] > A[g] then g := g + 1; fi
        fi // g geeft nu het grootste kind aan
        if x < A[g] then
            A[i] := A[g]; i := g;
        else
            j := 0; // exit loop
        fi
    od
    A[i] := x;

HeapSort(A, n) :
    for i := n div 2 downto 1 do SiftUp(A, i, n) od
    for i := n downto 2 do Wissel(A[1], A[i]); SiftUp(A, 1, i − 1); od
```

a. Hoeveel vergelijkingen tussen array-elementen doet *SiftUp* in het slechtste geval?
b. Hoeveel werk kost de constructie van de heap —de eerste for-loop uit *HeapSort*— ten hoogste?
c. En hoeveel werk kost de tweede for-loop?
d. Geef een bovengrens voor de worst case complexiteit van Heapsort.

Opgave 39.

Een sorteermethode heet *stabiel* als getallen (of karakters of ...) met dezelfde waarde, in de uitvoer in dezelfde volgorde staan als in de invoer.
a. Is de sorteermethode Insertion sort stabiel? En Mergesort?
b. Dezelfde vraag voor Bubblesort (opgave 6), Slowsort (opgave 8) en Heapsort (opgave 38).

Opgave 40.
a. Hoe werkt *Quicksort* op het rijtje 5 5 5 5 5 5 5 5 ?
Hoeveel vergelijkingen en hoeveel verwisselingen worden gedaan?
b. Laat $n = 2^k$. Hoeveel vergelijkingen doet Quicksort voor een rijtje van n dezelfde getallen? En hoeveel verwisselingen?
c. Is Quicksort stabiel?
d. Laat n even zijn. Hoeveel vergelijkingen doet Quicksort op het rijtje $n, n-1, \ldots, 2, 1$? En hoeveel verwisselingen? Hoeveel verwisselingen worden gedaan op het reeds gesorteerde rijtje $1, 2, \ldots, n-1, n$?

Opgave 41.
a. Zoals bekend doet *Quicksort* $\frac{1}{2}n(n+3) - 2 = \Theta(n^2)$ vergelijkingen op reeds gesorteerde rijtjes zoals $1, 2, 3, \ldots, n$. Bekijk wat er gebeurt met het gesorteerde rijtje als voor $n > 2$ nu niet steeds het eerste array-element $A[p]$ wordt gekozen als pivot, maar de mediaan van $A[p]$, $A[r]$ en $A[\lceil \frac{p+r}{2} \rceil]$. (Vervolgens wordt deze mediaan verwisseld met $A[p]$ en wordt verdergegaan met Partitie na de regel met (*).)
b. Neem aan dat we een rijtje met n verschillende getallen hebben. Hoe wordt het rijtje opgesplitst door Partitie als de pivot x het i-de element in grootte is (van klein naar groot: het 1-e element in grootte is de kleinste, het n-de in grootte de grootste)? M.a.w. wat is de q die Partitie dan oplevert?
c. Laat $n = 10$. We bekijken weer de "gewone" versie van Quicksort, dus waarbij als pivot steeds het eerste array-element wordt gekozen. Hoeveel vergelijkingen doet Quicksort als Partitie om en om de slechtste en de beste splitsing oplevert? Neem aan dat Partitie in elke stap $m + 1$ vergelijkingen doet voor een (deel)rijtje van m elementen. Vergelijk dit aantal met het aantal vergelijkingen dat gedaan wordt voor het gesorteerde rijtje. Idem met het aantal vergelijkingen dat Quicksort doet voor het rijtje waarbij in elke stap de (deel)rij in twee ongeveer gelijke stukken wordt verdeeld (best case). Ligt het aantal vergelijkingen dichter bij de best case of bij de worst case?
d. Maak aannemelijk dat een invoerrijtje waarbij afwisselend slecht en goed wordt gesplitst (zie **c.**) in orde van grootte evenveel vergelijkingen doet als het rijtje waarbij in elke ronde in twee gelijke delen wordt gesplitst ($\Theta(n \lg n)$ dus). Gebruik hiertoe dat in het eerste geval in *twee* rondes drie deelrijtjes zijn "ontstaan" van resp. 1, $\frac{m-1}{2}$ en $\frac{m-1}{2}$ (geval m is oneven) elementen, terwijl we in het tweede geval na *een* ronde een rijtje ter lengte $\frac{m-1}{2}$ en een rijtje ter lengte $\frac{m+1}{2}$ hebben. Hierin is m de lengte van het (deel)rijtje waarop we Quicksort (dus Partitie) loslaten. Een en ander suggereert dat Quicksort het meestal goed ($= \Theta(n \lg n)$) doet: het gemiddelde zal "dus" rond $\Theta(n \lg n)$ liggen.

Opgave 42.
Sorteer met behulp van *Shellsort* het rijtje 7, 19, 24, 13, 31, 8, 82, 18, 44, 63, 5, 29. Gebruik achtereenvolgens de volgende rijtjes stapgroottes (increments) en vergelijk het aantal vergelijkingen dat in elk van die drie gevallen wordt gedaan:
(i) 8, 4, 2, 1 (Shell); (ii) 7, 3, 1 (Hibbard); (iii) 4, 1 (h_2 ongeveer $1,72 * n^{\frac{1}{3}}$).

Opgave 43.
Bewijs dat het aantal vergelijkingen dat *Shellsort* met sprongafstanden (=stapgroottes = increments) $n/5$, $n/25$, $n/125$, ..., 25, 5, 1, in de worst case doet $\Omega(n^2)$ is. Neem aan dat n een macht van 5 is. Hint: vergelijk het geval van het college, met n een macht van 2.

Opgave 44. Hoeveel vergelijkingen doet Shellsort op het reeds gesorteerde rijtje, zoals $1, 2, 3, \ldots, n$? Laat $n = 2^k$ en gebruik Shell's sprongafstanden (increments).

Opgave 45.
a. Illustreer de werking van *Counting sort* aan de hand van het invoerrijtje: 7, 1, 3, 1, 2, 4, 5, 7, 2, 4, 3.
b. Veronderstel dat de laatste for-loop nu van links naar rechts loopt: **for** $j := 1$ **to** n **do**. Sorteert het algoritme dan nog? Zo ja, is de methode nu nog stabiel?

Opgave 46.
a. *Radix sort* kan ook gebruikt worden voor het alfabetisch sorteren van woorden van gelijke lengte. Sorteer nu het volgende rijtje: wet, zon, oor, kin, was, het, oog, pet, zin, pas.
b. Toon aan met behulp van volledige inductie dat Radix sort werkt. Bewijs daartoe: na de i-de sorteerslag staan de getallen (woorden) corresponderend met de laatste i cijfers (letters) onderling in de juiste volgorde.

Opgave 47. Polynoomevaluatie met preprocessing.
a. Breng het polynoom $p(x) = x^7 + 4\,x^6 - 8\,x^4 + 6\,x^3 + 9\,x^2 + 2\,x - 3$ in de speciale vorm $(x^4 + b)\,q(x) + r(x)$, waarbij q en r polynomen van graad 3 zijn en ook weer in de gewenste speciale vorm staan. Vergelijk het voorbeeld uit dit dictaat/college.
b. $A(k)$, het aantal $+/-$'s dat gedaan wordt om een polynoom van graad $n = 2^k - 1$ -dat reeds in de speciale vorm staat- te evalueren, voldoet aan de recurrente betrekking:

$$A(k) = \begin{cases} 1 & k = 1 \\ 2\,A(k-1) + 2 & k > 1 \end{cases}$$

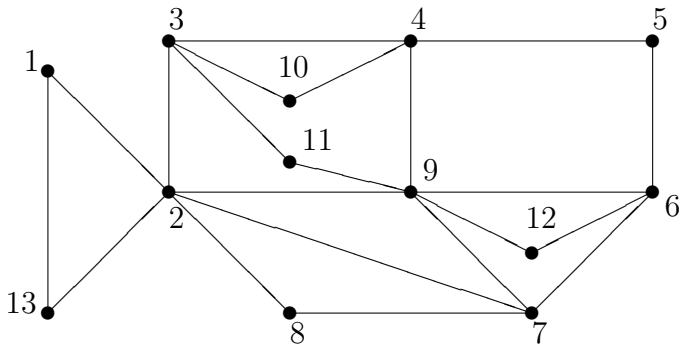
Laat zien dat $A(k) = 3 \cdot 2^{k-1} - 2$.
c. Ook het totaal aantal vermenigvuldigingen $M'(k)$ (inclusief de berekening van x, x^2, $x^4, \ldots, x^{2^{k-1}}$) kan worden bepaald. Hiervoor geldt dat $M'(k)$ (uitgedrukt in n) $= \frac{n-1}{2} + \lg(n+1) - 1$. Waarom is dit aantal vermenigvuldigingen niet in strijd met de theoretische ondergrens van n vermenigvuldigingen voor het evalueren van de waarde van een n-de graads polynoom in een gegeven x-waarde?

Opgave 48.
Gegeven een n-de graads polynoom p met kopcoëfficiënt 1 en n een 2-macht min 1 met $n \geq 1$. Zo'n polynoom wordt gerepresenteerd door zijn coëfficiënten, die zich in een integer array P met $n+1$ elementen bevinden. De coëfficiënt van x^t zit in $P[t]$, voor $t = 0, 1, \ldots, n$. Neem verder aan dat het statement $Q :=$ *new int*$[j]$; dynamisch een array Q met j integers maakt, en dat een **print**-opdracht al zijn door komma's gescheiden argumenten achtereenvolgens afdrukt — als ze tussen " en " staan uiteraard letterlijk.
We bekijken het volgende recursieve preprocessing algoritme, dat $p(x)$ in de speciale vorm brengt zoals bedoeld in het dictaat/college.

```
(1)   Doe(P, n) ::
(2)       if (n == 1) then
(3)           print "[x + " , P[0] , "]";
(4)       else
(5)           j := (n + 1)/2;
(6)           b := P[j − 1] − 1;
(7)           Q := new int[j];
(8)           R := new int[j];
(9)           t := 0;
(10)          while (t < j) do
(11)              Q[t] := P[j + t];
(12)              R[t] := P[t] − b ∗ Q[t];
(13)              t := t + 1;
(14)          od
(15)          print "[(x ↑ " , j , " + " , b , ") ∗ ";
(16)          Doe(Q, j − 1);
(17)          print " + ";
(18)          Doe(R, j − 1);
(19)          print "]";
(20)      fi
```

Voorbeeld: voor $p(x) = x^3 + 6x^2 + 5x + 4$ is $n = 3$ en $P[3] = 1$, $P[2] = 6$, $P[1] = 5$ en $P[0] = 4$. Hiervoor zou afgedrukt worden: $[(x \uparrow 2 + 4) * [x + 6] + [x + -20]]$, wat de ontbinding $x^3 + 6x^2 + 5x + 4 = (x^2 + 4)[x + 6] + [x - 20]$ representeert.
In het algemeen wordt een ontbinding $p(x) = (x^j + b)q(x) + r(x)$ gemaakt, waarbij q met Q en r met R correspondeert.
a. (Ter controle:) Voer het algoritme uit voor het polynoom $p(x) = x^7 + 4\,x^6 - 8\,x^4 + 6\,x^3 + 9\,x^2 + 2\,x - 3$, met $n = 7$.
b. Laat zien dat het aantal keren dat regel (12) (en dus ook regel (11) en regel (13)) wordt uitgevoerd maatgevend is voor de complexiteit van het algoritme. Leg ook kort uit hoe recursie hierin een rol speelt.
c. Vervolgens kijken we ook nog hoeveel werk het algoritme zelf, kortom het "preprocessen", doet voor het fabriceren van de ontbinding.
Hoeveel optellingen/aftrekkingen A(k) en vermenigvuldigingen V(k) uit regel (6) (niet de berekening van de array-index) en (12) tezamen worden er gedaan door het algoritme voor een vast n-de graads polynoom (met $n = 2^k - 1$), recursieve aanroepen (regels (16) en (18)) meegeteld? Geef recurrente betrekkingen met randvoorwaarden en los deze op.
Er zal blijken: $A(k) = k\,2^{k-1} - 1$ en $V(k) = (k-1)\,2^{k-1}$.

Opgave 49.
We bekijken in de deze en volgende opgaven het probleem van het vinden van een Eulerkring in een graaf. Eulerkringen hebben onder andere toepassingen in DNA-computing.
Construeer een Eulerkring in onderstaande ongerichte graaf met behulp van het op college gegeven algoritme.



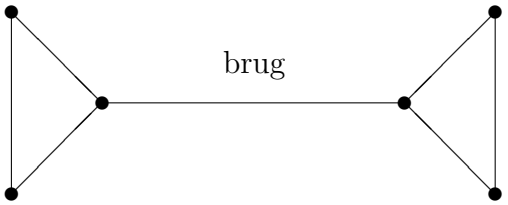
Opgave 50.
We implementeren het algoritme om een Eulerkring te vinden als volgt. De graaf $\mathcal{G} = (V, E)$ wordt gerepresenteerd middels de adjacency list. De op te bouwen Eulerkring wordt als enkelverbonden lijst gerepresenteerd, met een extra pointer die wijst naar de eerste knoop in de kring die nog ongebruikte takken "heeft". Zodra een tak gekozen wordt verwijderen we die uit de graaf $\mathcal{G}$ (eventueel gebruiken we een kopie). Laat zien dat het algoritme zo inderdaad (worst case) complexiteit $O(|V| + |E|)$ heeft.

Opgave 51.
Gegeven een ongerichte graaf. Een *Eulerpad* is een pad in de graaf dat elke tak precies één keer bevat. Begin- en eindpunt hoeven dus niet hetzelfde te zijn; als dit wel het geval is spreken we van een Eulerkring.
Een gevolg van de stelling dat een samenhangende ongerichte graaf een Eulerkring heeft d.e.s.d. als de graad van elke knoop even is, is: een ongerichte graaf $\mathcal{G}$ bevat een Eulerpad indien er precies twee knopen zijn met oneven graad. Bewijs dit. Leg ook uit hoe zo'n Eulerpad geconstrueerd kan worden.

Opgave 52.
Een ander (maar soortgelijk) algoritme voor het construeren van een Eulerkring in een ongerichte graaf is het algoritme van Fleury:

1. Start in een willekeurige knoop;
2. Kies bij het opbouwen van de kring steeds nog niet gebruikte takken;
3. Hierbij wordt een tak die doorlopen wordt meteen verwijderd.
4. Een *brug* wordt pas gekozen als er geen alternatief is.
5. Stop als er geen ongebruikte takken meer zijn;

Een *brug* is een tak met de eigenschap dat als hij wordt weggelaten uit een samenhangende (deel)graaf er twee samenhangscomponenten ontstaan.



a. Gebruik dit algoritme om een Eulerkring te vinden in de graaf uit opgave 49.
b. Wat is de (worst case) complexiteit van dit algoritme? Bedenk zelf een implementatie.

Opgave 53.
Bewijs dat zowel voor het handelsreizigersprobleem als voor het graafkleuringsprobleem geldt: als het optimalisatieprobleem in een polynomiaal aantal stappen kan worden opgelost, dan geldt dat ook voor het bijbehorende beslissingsprobleem.

Opgave 54.
Beschouw het volgende roosterprobleem. Een organisatie heeft 200 leden en 17 commissies. Jaarlijks moet in de laatste week van mei elke commissie een hele middag bij elkaar komen om te vergaderen. Een lijst van alle commissies en hun leden is beschikbaar. Sommige mensen zitten in meer dan een commissie. Is het mogelijk om de commissievergaderingen in vijf middagen te roosteren zodat elk lid de vergaderingen van alle commissies waar hij/zij deel van uitmaakt kan bijwonen?
Laat zien dat dit roosterprobleem niets anders is dan het 5-kleurenprobleem (dat is Kleur met $k = 5$).

Opgave 55.
a. Laten A en B twee polynomiaal begrensd algoritmen zijn. Stel C is het algoritme dat eerst een aanroep naar subroutine A doet en vervolgens, met dezelfde invoer, naar subroutine B. Dan is dit algoritme ook polynomiaal begrensd. Toon dit aan.

b. Gegeven twee polynomiaal begrensde algoritmen A en B. Laat de uitvoer van A nu gebruikt worden als invoer voor B. Dan is het samengestelde algoritme B $\circ$ A ook polynomiaal begrensd. Toon dit aan.

Gevolg. (a) Een algoritme dat een constant aantal aanroepen doet naar polynomiaal begrensde subroutines, is zelf ook polynomiaal begrensd. (b) De samenstelling van een constant aantal polynomiaal begrensde algoritmen is zelf ook polynomiaal begrensd.

Opgave 56. Schrijf een DTM-programma dat van een gegeven invoerstring bestaande uit nullen en enen nagaat of deze even lengte heeft. De invoerstring mag tijdens of na de werking van het programma niet veranderd worden.
Geef de transitiefunctie in de vorm van een tabel, en beschrijf de werking van je DTM-programma ook in woorden. Geef tevens aan hoe je verzameling tape-symbolen er uitziet.

Opgave 57. Schrijf een DTM-programma dat van een gegeven invoerstring bestaande uit nullen en enen nagaat of het een palindroom is. Een string is een palindroom als deze van voor naar achter gelezen hetzelfde is als van achter naar voor gelezen. Een voorbeeld van een palindroom is de string 01110101110. De invoerstring mag na afloop beschadigd/vernietigd zijn.
Geef de transitiefunctie in de vorm van een tabel, en beschrijf de werking van je DTM-programma ook duidelijk in woorden. Geef tevens aan hoe je verzameling tape-symbolen er uitziet.

Opgave 58.
Voor een beslissingsprobleem P definiëren we het complement $\overline{P}$ als het probleem dat dezelfde invoerverzameling heeft als P, maar dat precies het tegengestelde vraagt als P. Voor een invoer x geldt dus dat x een ja-instantie is van $\overline{P}$ dan en slechts dan als x een nee-instantie is van P.
Voorbeeld. HC: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Hamiltonkring? Dan is $\overline{HC}$: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ geen Hamiltonkring?

a. Laat zien: als P $\in \mathcal{P}$ dan ook $\overline{P} \in \mathcal{P}$ en omgekeerd.
b. Zij co-$\mathcal{NP} = \{P : \overline{P} \in \mathcal{NP}\}$. Bewijs dat $\mathcal{P} \subseteq$ co-$\mathcal{NP}$.
c. Bewijs dat $\mathcal{P} =$ co-$\mathcal{P}$, waarbij co-$\mathcal{P} = \{P : \overline{P} \in \mathcal{P}\}$.
d. Bewijs nu het volgende: als $\mathcal{NP} \neq$ co-$\mathcal{NP}$ dan $\mathcal{P} \neq \mathcal{NP}$.

Opgave 59.
Bewijs van de volgende beslissingsproblemen dat ze in $\mathcal{NP}$ zitten:
a. SAT; **b.** SubsetSum; **c.** Kliek; **d.** TSP

Opgave 60. KnapZak probleem (KZ)
Veronderstel dat we een knapzak hebben met capaciteit C (een geheel getal > 0) en n objecten, genummerd 1 t/m n, met gewicht $g_1, g_2, \ldots g_n$ en met waarde $w_1, w_2, \ldots, w_n$. (Alle g_i en w_i zijn geheel en > 0.)

Beslissingsprobleem: Gegeven een geheel getal k, is er een deelverzameling van de objecten die (1) in de knapzak past en (2) een totale waarde $\geq k$ heeft? M.a.w. bestaat er een deelverzameling V van de objecten (dus van $\{1, 2, \ldots, n\}$) met totaalgewicht $\Sigma_{i \in V} g_i \leq C$ en met totale waarde $\Sigma_{i \in V} w_i \geq k$?

Laat zien dat KZ in $\mathcal{NP}$ zit door een niet-deterministisch algoritme voor KZ te geven. De invoer x voor het probleem (en dus voor het algoritme) zijn alle gewichten en waarden van de objecten, en k en C, dus $x = \langle (g_1, w_1), (g_2, w_2), \ldots, (g_n, w_n), k, C \rangle$. We mogen wel aannemen dat het aantal objecten n bekend is (ze tellen kan immers polynomiaal in $|x|$).

Opgave 61.
Laat zien dat het volgende probleem in $\mathcal{P}$ zit.
Gegeven een logische formule ϕ in DNF (*disjunctieve normaalvorm*). Bestaat er een waardering van de in ϕ voorkomende logische variabelen die ϕ waar maakt?
Een formule staat in disjunctieve normaalvorm als hij bestaat uit een disjunctie ($\vee$) van conjuncties ($\wedge$) van literals.

Opgave 62.
Bewijs dat de relatie $\leq_{\mathrm{P}}$ transitief is, dat wil zeggen: laat zien dat als $P_1 \leq_{\mathrm{P}} P_2$ en $P_2 \leq_{\mathrm{P}} P_3$ dan ook $P_1 \leq_{\mathrm{P}} P_3$.

Opgave 63.
Beschouw de volgende beslissingsproblemen:

3Kleur: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Kunnen de knopen van $\mathcal{G}$ zo gekleurd

worden met hooguit *drie* kleuren dat geen tweetal aangrenzende knopen dezelfde kleur heeft?

4Kleur: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Kunnen de knopen van $\mathcal{G}$ zo gekleurd worden met hooguit *vier* kleuren dat geen tweetal aangrenzende knopen dezelfde kleur heeft?

a. Laat zien dat 3Kleur polynomiaal reduceerbaar is tot 4Kleur. *Hint* bij de reductie: voeg een extra knoop toe met takken naar de andere knopen.
b. Toon aan: 3Kleur $\in \mathcal{NP}$.
c. Veronderstel dat 3Kleur NP-volledig is. Volgt nu uit **a.** dat ook 4Kleur $\in \mathcal{NPC}$? Zo ja waarom, zo nee waarom niet?

Opgave 64.
Er bestaat een polynomiale reductie van SAT naar 3Kleur, waaruit volgt dat 3Kleur NP-hard is. Samen met opgave **63.b** betekent dat dat 3Kleur $\in \mathcal{NPC}$ en dus ook 4Kleur $\in \mathcal{NPC}$ (want 4Kleur zit natuurlijk ook in $\mathcal{NP}$). In het algemeen is kKleur met $k = 3, 4, \ldots$ NP-volledig.
Echter 1Kleur en 2Kleur zitten beide in $\mathcal{P}$.
(1) Wanneer is een ongerichte graaf te kleuren met 1 kleur (geef een "dan en slechts dan als ..."-antwoord), en wanneer met 2 kleuren?
(2) Geef een polynomiaal algoritme dat nagaat of een gegeven ongerichte graaf met 1 kleur gekleurd kan worden, en dat als dat kan zo'n kleuring oplevert. Idem voor 2 kleuren.

Opgave 65.
Laat zien dat het beslissingsprobleem TSP NP-volledig is als gegeven is dat HC2 $\in \mathcal{NPC}$.
HC2 is het Hamiltonkringprobleem voor ongerichte grafen.
Reduceer daartoe HC2 tot TSP: beeld een graaf $\mathcal{G} = (V, E)$ af op de volledige graaf $\mathcal{G}' = (V, E')$, met $E' = \{(i, j) : i, j \in V\}$ en definieer de gewichten op de takken van $\mathcal{G}'$ als:

$$c_{ij} = \begin{cases} 0 & \text{als} \quad (i,j) \in E \\ 1 & \text{als} \quad (i,j) \notin E \end{cases}$$

Opgave 66.
We bekijken de volgende twee beslissingsproblemen:

Kliek: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$ en een positief geheel getal $k \leq |V|$. Heeft $\mathcal{G}$ een *kliek* bestaande uit $\geq k$ knopen?
Een kliek is een deelverzameling V' van V zodat voor elk tweetal knopen $u, v \in V'$ met $u \neq v$ geldt dat $(u, v) \in E$ (m.a.w. tussen elk tweetal knopen uit V' zit een tak).

VertexCover (VC): Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$ en een positief geheel getal $k \leq |V|$. Heeft $\mathcal{G}$ een *vertex cover* bestaande uit $\leq k$ knopen?
Een vertex cover is een deelverzameling V' van V zodat voor elke tak $(u, v) \in E$ ten minste één van de uiteinden u of v bevat is in V'.

a. Toon aan dat VC $\in \mathcal{NP}$.

We gaan Kliek reduceren tot VC. De reductie is gebaseerd op het complement $\overline{\mathcal{G}}$ van een graaf $\mathcal{G}$. $\overline{\mathcal{G}}$ is gedefinieerd als $(V, \overline{E})$, met $\overline{E} = \{(u, v) : u \neq v \text{ en } (u, v) \notin E\}$. De transformatie T beeldt $\mathcal{G}$ af op $\overline{\mathcal{G}}$ en k op $|V| - k$, dus $T(\langle \mathcal{G}, k \rangle) = \langle \overline{\mathcal{G}}, |V| - k \rangle$.

b. Laat zien dat de transformatie T polynomiaal begrensd is.
c. Bewijs: $\mathcal{G}$ heeft een kliek ter grootte $\geq k \iff \overline{\mathcal{G}}$ heeft een vertex cover ter grootte $\leq |V| - k$.

Opgave 67.
Bekijk het volgende beslissingsprobleem:

3Kleur: Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Is er een manier om de knopen van $\mathcal{G}$ te kleuren zodat aangrenzende knopen verschillend gekleurd zijn en zodat hoogstens 3 kleuren gebruikt zijn?

We willen 3Kleur reduceren tot SAT. Definieer daartoe een transformatie die een ongerichte graaf als volgt op een logische expressie $\phi_{\mathcal{G}}$ (in CNF) afbeeldt:

- voor elke knoop $v \in V$ zijn er drie logische variabelen v_1, v_2 en v_3, corresponderend met de drie mogelijke kleuren voor v.
- voor elke knoop $v \in V$ hebben we de volgende vier clausules:
  $\phi_v^1 = v_1 \vee v_2 \vee v_3; \phi_v^2 = \neg v_1 \vee \neg v_2; \phi_v^3 = \neg v_2 \vee \neg v_3; \phi_v^4 = \neg v_1 \vee \neg v_3$.
- voor elke tak $e = (v, w) \in E$ hebben we de volgende drie clausules:
  $\phi_e^1 = \neg v_1 \vee \neg w_1; \phi_e^2 = \neg v_2 \vee \neg w_2; \phi_e^3 = \neg v_3 \vee \neg w_3$.

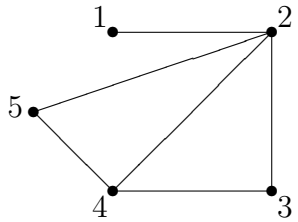
Het beeld van $\mathcal{G}, \phi_{\mathcal{G}}$, is dan de conjunctie van $\phi_v^1, \phi_v^2, \phi_v^3, \phi_v^4, \phi_e^1, \phi_e^2, \phi_e^3$ voor alle $v \in V$ en $e \in E$. Dat is dus een conjunctie van $4 * |V| + 3 * |E|$ clausules.

a. Laat zien dat 3Kleur $\leq_P$ SAT.
b. Het is bekend dat SAT NP-volledig is. Volgt nu uit **a** en **63.b** dat ook 3Kleur NP-volledig is? Motiveer je antwoord.
c. Bestaat er een polynomiale reductie van SAT naar 3Kleur? Waarom (niet)?

Opgave 68.
We bekijken het volgende beslissingsprobleem:

Independent Set (InSet): Gegeven een ongerichte graaf $G = (V, E)$ en een geheel getal K met $K > 0$.
Vraag: is er een onafhankelijke verzameling I ($I \subseteq V$) met $|I| = K$?
Een deelverzameling I van de knopen V heet *onafhankelijk* als geldt: voor alle $i, j \in I$ is er *geen* tak tussen i en j. Verder wordt met $|I|$ het aantal knopen van I bedoeld.

Voorbeeld : zij G als hieronder en $K = 3$, dan is $I = \{1, 3, 5\}$ een onafhankelijke knoopverzameling ter grootte K.



a. Toon aan dat InSet $\in \mathcal{NP}$.
b. Het is bekend dat InSet $\in \mathcal{NPC}$. Gegeven is nu een beslissingsprobleem P. Geef van de volgende beweringen aan of ze waar zijn of niet (waarom/waarom niet)? Motiveer je antwoord en formuleer duidelijk gebruikte stellingen.
(i) Als gegeven is dat $P \in \mathcal{NP}$ en $P \leq_{\mathrm{P}}$ InSet, dan is P NP-volledig.
(ii) Als gegeven is dat $P \in \mathcal{NP}$ en InSet $\leq_{\mathrm{P}} P$, dan is P NP-volledig.
(iii) Uit de NP-volledigheid van InSet volgt onmiddelijk dat $P \leq_{\mathrm{P}}$ InSet.
c. We fixeren nu de invoerwaarde K voor het Independent Set probleem. Voor $K = 2$ krijgen we dan het beslissingsprobleem 2InSet.

2InSet: Gegeven een ongerichte graaf G. Bevat G een onafhankelijke knoopverzameling bestaande uit 2 knopen?

Geheel analoog definiëren we 3InSet, 4InSet, 5InSet, ... Beantwoord nu de volgende twee vragen.

(i) Is 2InSet NP-volledig? Leg uit waarom/waarom niet.
(ii) Zijn 3InSet, 4InSet, 5InSet, ... NP-volledig of niet? Motiveer je antwoord.

EINDE