# Complexiteit 2023 — college 12

2 mei 2023

---

## Herhaling

# Vandaag

Herhaling hele semester

# Complexiteit 2023 — college 1

7 februari 2023

---

## Introductie

# Praktische informatie

**Docenten**
Jeannette de Graaf
Luc Edixhoven

**Assistentie**
Andy Tatman
Perri van den Berg
Rachel de Jong
Sara Kooistra

`l.j.edixhoven@liacs.leidenuniv.nl`
Maandag en dinsdag: Snellius 131

# Praktische informatie

**Website**

`liacs.leidenuniv.nl/~edixhovenlj/complexiteit`

**Materiaal**

Dictaat met opgaven (zie website) + sheets

**Brightspace**

Hoofdzakelijk voor inleveren van opdrachten en voor cijfers

**Vragen**

Discord en discussieforum op Brightspace, anders e-mail.
Antwoorden binnen 'redelijke termijn'.

# Praktische informatie

**Rooster**

Zie website ;) (en MyTimetable)

**Belangrijkst**

- hoorcolleges op dinsdagen 09u00–10u45, afwisselende zalen
- werkcolleges op dinsdagen 11u00–12u45, Snellius 302–304, 306–308 en 303
- géén college op 28 maart
- tentamen op maandag 5 juni 2023, 09u00–12u00
- hertentamen op maandag 10 juli 2023, 09u00–12u00

# Praktische informatie

**Huiswerk**

- Vier opdrachten, individueel in te leveren
- Samenwerken mag (met mate), overschrijven niet
- Voor elke huiswerkopdracht krijg je een cijfer
- Het gemiddeld huiswerkcijfer telt mee als bonus indien het tentamencijfer $\geq 5{,}0$ is.

# Praktische informatie

**Eindcijfer**

**if** *tentamencijfer* $\geq 5{,}0$ **then**
  eindcijfer = tentamencijfer + gemiddeld huiswerkcijfer$/10$;
**else**
  eindcijfer = tentamencijfer;
**fi**

Vragen tot nu toe?

# Waar gaat het vak over?

**Analyse van algoritmen**

- Algoritmiek (en Datastructuren) (al gehad, enige herhaling)
  - → hoe los je dit probleem op?
- Programmeren en Correctheid (keuzevak jaar 3)
  - → klopt je oplossing?
- Computability (ook dit semester)
  - → kunnen we dit probleem überhaupt oplossen?
- Complexiteit (dit vak)
  - → hoeveel werk kost deze oplossing? (tijd)complexiteit
  - → hoeveel geheugen? ruimtecomplexiteit
  - → en kan het beter? optimaliteit

# En waarom is dat nuttig?

| $n$ | $10$ | $100$ | $1\,000$ | $100\,000$ | $10\,000\,000$ |
|---|---|---|---|---|---|
| $\lg n$ | $3$ | $6$ | $9$ | $16$ | $23$ |
| $\sqrt{n}$ | $3$ | $10$ | $32$ | $316$ | $3\,162$ |
| $n$ | $10$ | $100$ | $1\,000$ | $100\,000$ | $10\,000\,000$ |
| $n \lg n$ | $33$ | $664$ | $9\,966$ | $1\,660\,964$ | $\approx 10^8$ |
| $n^2$ | $100$ | $10\,000$ | $1\,000\,000$ | $10^{10}$ | $10^{14}$ |
| $n^{10}$ | $10^{10}$ | $10^{20}$ | $10^{30}$ | $10^{50}$ | $10^{70}$ |
| $2^n$ | $1\,024$ | $\approx 10^{30}$ | $\approx 10^{301}$ | $\approx 10^{30\,102}$ | veel |
| $n!$ | $3\,628\,800$ | $\approx 10^{157}$ | $\approx 10^{2\,567}$ | $\approx 10^{456\,573}$ | veel |
| $n^n$ | $10^{10}$ | $10^{200}$ | $10^{3\,000}$ | $10^{500\,000}$ | $10^{70\,000\,000}$ |

# En waarom is dat nuttig?

De absurd hoge getallen terzijde: bijvoorbeeld het sorteren van rijtjes is iets dat overal gebeurt en voor gigantische rijtjes, en niet-triviale algoritmen voor sorteren zijn o.a. in $n^2$, $n\sqrt{n}$ en $n \lg n$.

Voor $n = 10\,000\,000$, met $1\,000\,000$ operaties per seconde:

- $n \lg n \rightarrow$ 4 minuten
- $n\sqrt{n} \rightarrow$ 9 uur
- $n^2 \rightarrow$ 3 jaar

# Bonus

$n^3$ → 32 miljoen jaar

Trix ⤴ zou nu net zijn begonnen aan haar derde rijtje.

(`https://topstukken.naturalis.nl/object/trix`)

# Opbouw vak

Het vak bestaat uit grofweg twee delen:

1. Complexiteit en optimaliteit van algoritmen

   (d.w.z. tijdcomplexiteit)

2. Complexiteitstheorie: $\mathcal{P}$ vs $\mathcal{NP}$ en een beetje verder

Vandaag: inleiding basisconcepten en 'het vak in vogelvlucht'

Het laatste college is gereserveerd voor tentamenvoorbereiding. Er zit nog wat speling in het rooster aan het einde van het semester.

# Hoeveelheid werk

Complexiteit = tijdcomplexiteit = hoeveelheid werk

- een maat voor de hoeveelheid werk moet iets vertellen over de efficiëntie van de gebruikte methode
- die maat moet onafhankelijk zijn van de gebruikte computer, programmeertaal, implementatiedetails etc.
- grotere invoeren kosten meer werk, dus complexiteit wordt gewoonlijk uitgedrukt als functie van de grootte van de invoer

# Maximum

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

Een algoritme in *pseudocode* dat het maximum vindt:

```
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;
```

# Maximum: complexiteit

We tellen het aantal operaties:

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

$3n \leq$ totaal aantal operaties $\leq 4n \Longrightarrow \Theta(n)$

# Basisoperatie

Om de complexiteit van een algoritme te bepalen tellen we het aantal keer dat een geschikte basisoperatie wordt uitgevoerd.

- 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) met 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

# Basisoperatie: voorbeelden

| probleem | basisoperatie (meestal) |
|---|---|
| $X$ zoeken in een array | vergelijking van $X$ met een array-element (en/of vergelijking tussen array-elementen onderling) |
| twee polynomen vermenigvuldigen | vermenigvuldiging van twee getallen (en/of optelling) |
| een array sorteren | vergelijking van twee array-elementen |
| graafprobleem | bezoek aan een knoop en/of doorlopen van een tak/pijl |

# Complexiteit

De complexiteit van een algoritme hangt af van de grootte van de invoer, $n$: $f(n)$

- bepaal een geschikte basisoperatie
- tel het aantal keer dat de basisoperatie wordt uitgevoerd
  $\Rightarrow f(n)$

Interessant om te weten is hoe hard $f(n)$ groeit:

- van belang is de orde van grootte van $f(n)$ voor *grote* $n$
  $\Rightarrow O, \Omega, \Theta$

# Grootte van de invoer

De complexiteit hangt af van de grootte van de invoer. Wat wordt hiermee bedoeld?

| probleem | (maat voor de) grootte van de invoer |
|---|---|
| $X$ zoeken in een array | aantal array-elementen |
| twee polynomen vermenigvuldigen | graad van de polynomen (= aantal coëfficiënten) |
| een array sorteren | aantal array-elementen |
| graafproblemen | aantal knopen en/of takken |

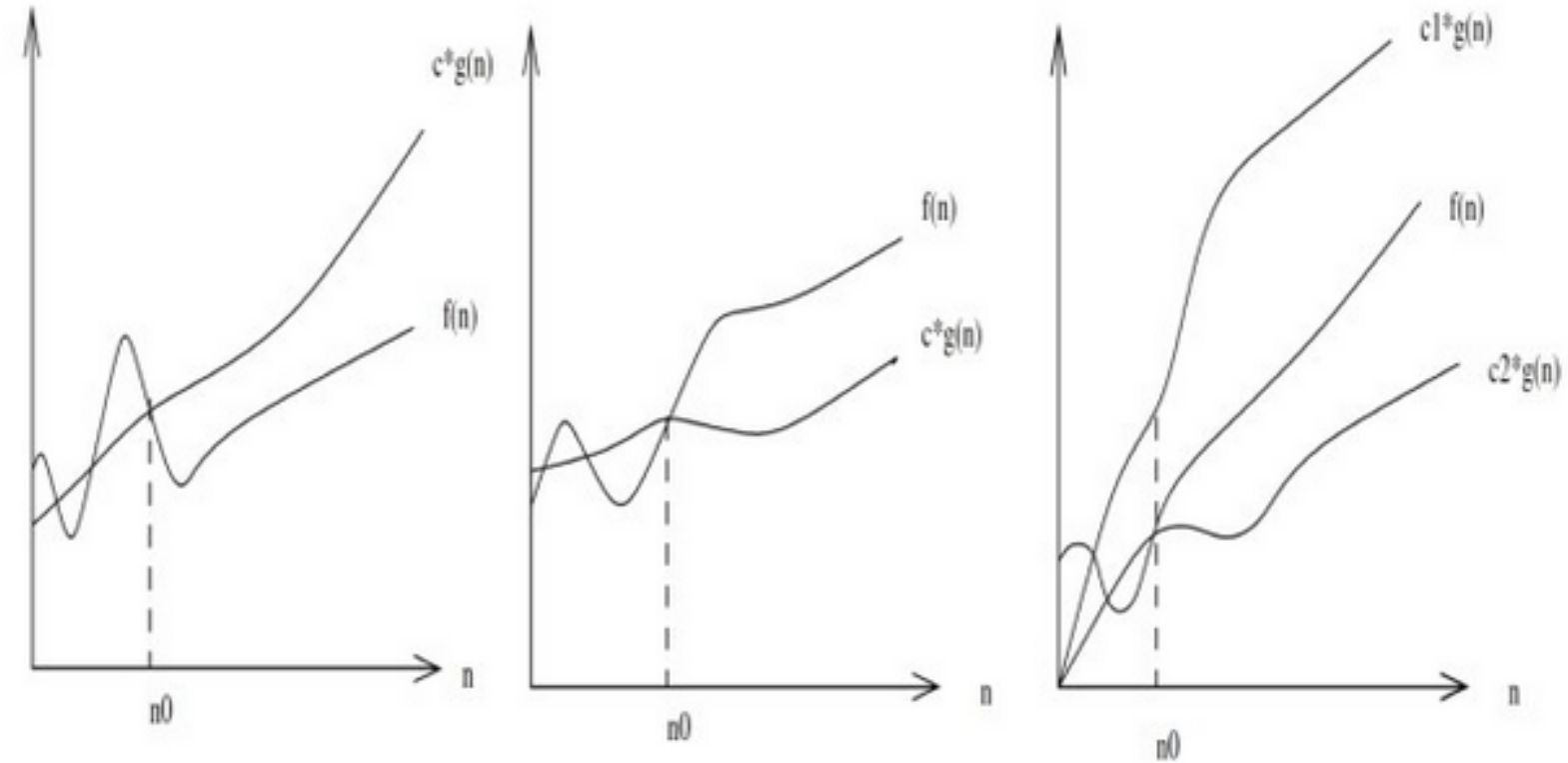
# $O$, $\Omega$ en $\Theta$

Gegeven twee functies $f(n)$ en $g(n)$. $O(g)$, $\Omega(g)$ en $\Theta(g)$ zijn verzamelingen van functies:

1. $f \in O(g)$: er bestaan constanten $c$ en $n_0$ (beide $> 0$) zodat $0 \leq f(n) \leq c \cdot g(n)$ voor alle $n \geq n_0$: asymptotische bovengrens
2. $f \in \Omega(g)$: er bestaan constanten $c$ en $n_0$ (beide $> 0$) zodat $0 \leq c \cdot g(n) \leq f(n)$ voor alle $n \geq n_0$: asymptotische ondergrens
3. $f \in \Theta(g)$: er bestaan constanten $c_1$, $c_2$ en $n_0$ (alle $> 0$) zodat $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ voor alle $n \geq n_0$: asymptotisch gedrag

Officieel dus $f \in O(g)$ en niet $f = O(g)$.

# $O$, $\Omega$ en $\Theta$




$f \in O(g)$

$f$ groeit hooguit even hard als $g$

$f \in \Omega(g)$

$f$ groeit minstens even hard als $g$

$f \in \Theta(g)$

$f$ groeit even hard als $g$

# $O$, $\Omega$ en $\Theta$

Enkele voorbeelden:

- $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$
- $3n^3 + 6n^2 + 9 \in \Theta(n^3)$
- $42n \in O(n^2)$, maar $42n \notin \Omega(n^2)$
- $2^n \in O(3^n)$, maar $2^n \notin \Omega(3^n)$
- $\log_7 n \in \Theta(\lg n)$ ($\lg = \log_2$)
- $C \in \Theta(1)$ (voor $C > 0$); $0 \in O(1)$, maar $0 \notin \Omega(1)$
- $\sum_{i=1}^{n} i \in \Theta(n^2)$
- $\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\lg n)$

# Ordes van grootte

**Veelgebruikte namen**

| | |
|---|---|
| $\Theta(1)$ | **constant** |
| $\Theta(\lg n)$ | **logaritmisch** |
| $\Theta(n)$ | **lineair** |
| $\Theta(n^2)$ | kwadratisch |
| $\Theta(n^k)$ met $k > 0$ | **polynomiaal** |
| $\Theta(a^n)$ met $a > 1$ | **exponentieel** |
| $\Theta(n!)$, $\Theta(n^n)$, ... | superexponentieel |

# Ordes van grootte

## Gedrag voor grote n

| $n$ | $10$ | $100$ | $1\,000$ | $100\,000$ | $10\,000\,000$ |
|---|---|---|---|---|---|
| $\lg n$ | $3$ | $6$ | $9$ | $16$ | $23$ |
| $\sqrt{n}$ | $3$ | $10$ | $32$ | $316$ | $3\,162$ |
| $n$ | $10$ | $100$ | $1\,000$ | $100\,000$ | $10\,000\,000$ |
| $n \lg n$ | $33$ | $664$ | $9\,966$ | $1\,660\,964$ | $\approx 10^8$ |
| $n^2$ | $100$ | $10\,000$ | $1\,000\,000$ | $10^{10}$ | $10^{14}$ |
| $n^{10}$ | $10^{10}$ | $10^{20}$ | $10^{30}$ | $10^{50}$ | $10^{70}$ |
| $2^n$ | $1\,024$ | $\approx 10^{30}$ | $\approx 10^{301}$ | $\approx 10^{30\,102}$ | veel |
| $n!$ | $3\,628\,800$ | $\approx 10^{157}$ | $\approx 10^{2\,567}$ | $\approx 10^{456\,573}$ | veel |
| $n^n$ | $10^{10}$ | $10^{200}$ | $10^{3\,000}$ | $10^{500\,000}$ | $10^{70\,000\,000}$ |

# Best, worst, average case

De complexiteit van een algoritme hangt af van de soort invoer: worst case, average case, best case

- worst case complexiteit: $g(n)$ is het aantal stappen in het *slechtste* geval; het algoritme doet voor elke mogelijke invoer dus hooguit $g(n)$ stappen
- best case complexiteit: $h(n)$ is het aantal stappen in het *beste* geval; het algoritme doet voor elke mogelijke invoer dus minstens $h(n)$ stappen
- worst case geeft dus een bovengrens, best case een ondergrens
- average case complexiteit: complexiteit 'gemiddeld' over alle mogelijke invoeren

# Best, worst, average 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
```

Dit algoritme sorteert $A[1], \ldots, A[n]$ oplopend.
(met $A[i] > 0$ voor $i = 1, \ldots, n$ en alle $A[i]$ verschillend)

Basisoperatie is de vergelijking $A[i] < A[i + 1]$ op regel 3.

# Best, worst, average 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
```

Best case: $n - 1$ vergelijkingen ($\Theta(n)$) d.e.s.d.a. regel 3 telkens waar is, m.a.w. als $A$ stijgend is

# Best, worst, average 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
```

Worst case: $n^2 - 1$ vergelijkingen ($\Theta(n^2)$) d.e.s.d.a. voor alle $i$ $A[i+1]$ kleiner is dan alle voorgaande, m.a.w. als $A$ dalend is

# Best, worst, average 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
```

Average case: $\Theta(n^2)$, gemiddeld over alle mogelijke invoeren (aangenomen dat ze allemaal even waarschijnlijk zijn)

# Optimaliteit

Bestaat er een efficiënter algoritme voor het probleem?

- heeft te maken met de (inherente) complexiteit van het probleem: soms *kan* het niet beter
- de worst case van een algoritme geeft een bovengrens: het probleem kan worden opgelost in hooguit . . . stappen (namelijk door dat algoritme)
- moeilijker is het bepalen van een (niet-triviale) ondergrens

# Optimaliteit

Een simpele (meestal niet al te scherpe) ondergrens voor de (worst case) complexiteit van een probleem kan soms worden verkregen door het aantal invoer/uitvoer-elementen te tellen:

- het optellen van twee $n \times n$ matrices is $\Omega(n^2)$ (scherp)
- het handelsreizigersprobleem met $n$ steden is $\Omega(n^2)$ (niet scherp)
- het zoeken naar $X$ in een gesorteerd array met $n$ elementen is $\Omega(n)$ (fout!)

# Optimaliteit

- een betere ondergrens op de complexiteit van een probleem vind je i.h.a. door te kijken naar de basisoperatie
- je bewijst zo (een ondergrens op) de complexiteit van een probleem binnen een bepaalde klasse van algoritmen, bijvoorbeeld die gebaseerd op het doen van arrayvergelijkingen
- bewijs stellingen die een ondergrens opleveren voor het aantal (basis)operaties dat nodig is om het probleem op te lossen, d.w.z. (volgende slide)

# Optimaliteit

- laat zien dat elk algoritme voor het probleem minstens . . . elementaire (basis)stappen moet doen in de . . . case (meestal worst case)
- technieken voor het bewijzen van ondergrenzen zijn o.a. beslissingsboomargumenten en adversary-argumenten
- soms ook een ad-hoc argument, zoals voor het vinden van het maximum van $n$ getallen

# Recursie

Complexiteit van recursieve algoritmen:

- in het algemeen is hier het aantal recursieve aanroepen een goede maat voor de hoeveelheid werk
- maar je kunt ook hier het aantal keren tellen dat de basisoperatie wordt uitgevoerd
- een en ander leidt tot recurrente betrekkingen

Als voorbeeld bekijken we een recursief algoritme voor het bepalen van het maximum van $n$ getallen in een array $A$.

# Maximum: recursief

```
1  int grootste(int A[ ], n) {
2      if n = 1 then
3          return A[n];
4      else
5          max := grootste(A, n − 1);
6          if A[n] > max then
7              max := A[n];
8          fi
9      fi
10     return max;
11 }
```

$$C(n) = \begin{cases} 0 & n = 1 \\ C(n-1) + 1 & n > 1 \end{cases} \qquad \text{Oplossing} \Rightarrow C(n) = n - 1$$

# Verdere wiskundige achtergrond

Raadpleeg het dictaat! o.a.:

**Floor en ceiling**

$\lfloor x \rfloor$ = het grootste gehele getal $\leq x$

$\lceil x \rceil$ = het kleinste gehele getal $\geq x$

**Logaritmen**

$\log_b(xy) = \log_b x + \log_b y \qquad \log_b x = \frac{\log_c x}{\log_c b}$

**Sommaties**

$\sum_{i=1}^{n} i = \frac{1}{2}n(n+1) \qquad \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

# Verdere wiskundige achtergrond

## Combinatoriek

Aantal rijtjes ter lengte $k$ van getallen $1$ t/m $n$ (met $k \leq n$):

- volgorde van belang, hoeft niet allemaal verschillend: $n^k$
- volgorde van belang, wel allemaal verschillend: $n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)$
- volgorde van belang, wel allemaal verschillend, $k = n$ — d.w.z. **permutaties**: $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- volgorde niet van belang, wel allemaal verschillend — d.w.z. **combinaties**: $\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$

# Complexiteitstheorie: $\mathcal{P}$ en $\mathcal{NPC}$

Algemener plaatje: welke problemen zijn 'moeilijk'?

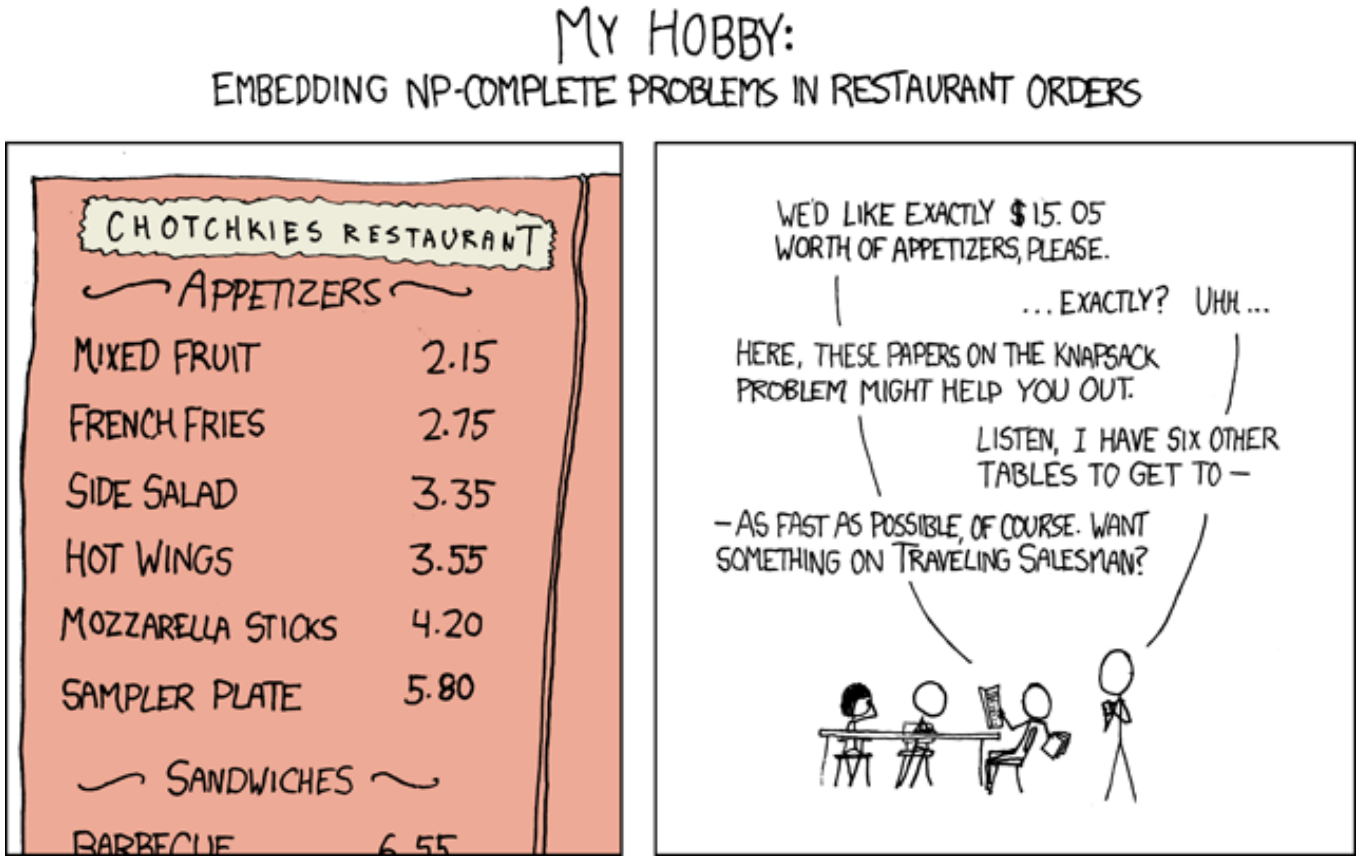
We onderscheiden drie soorten problemen:

- handelbare problemen → in het algemeen (min of meer) efficiënt op te lossen (d.w.z. polynomiale bovengrens): $\mathcal{P}$
- onhandelbare problemen → in het algemeen niet efficiënt op te lossen (bijv. exponentiële ondergrens)
- onbeslisbare problemen → in het algemeen überhaupt niet op te lossen: zie Computability

Complicatie: van veel problemen weten we gewoon niet of ze wel of niet handelbaar zijn: $\mathcal{NPC}$. Hiervoor weten we geen polynomiaal algoritme maar ook geen niet-polynomiale ondergrens.

# $\mathcal{NPC}$

Typische problemen in $\mathcal{NPC}$:




`https://xkcd.com/287/`

Ook bijv. SAT en graafkleuring.

# Reducties en $\mathcal{NPC}$

Met reducties kan je problemen met elkaar in verband brengen: als $A \leq_P B$ dan is $A$ hooguit polynomiaal moeilijker dan $B$.

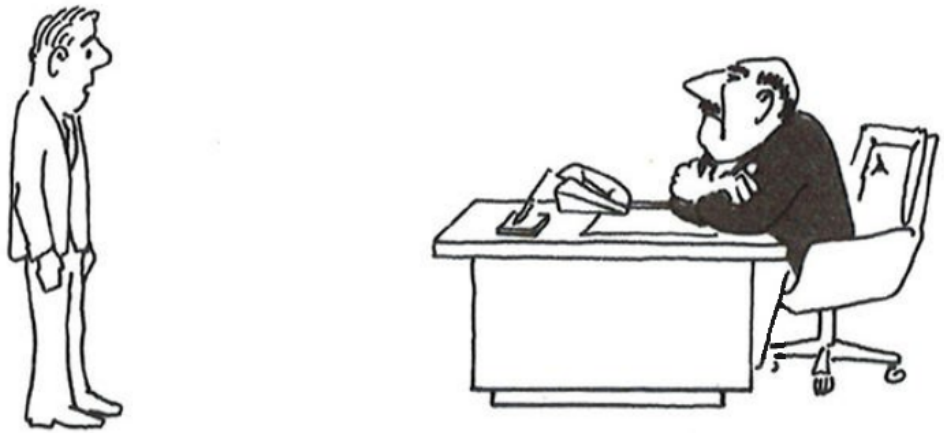
M.a.w.: als $B$ polynomiaal kan, dan $A$ ook! Nu hoef je alleen nog maar te redeneren over $B$.

Alle problemen in $\mathcal{NPC}$ reduceren naar elkaar. Het oplossen van één is voldoende om voor allemaal uitsluitsel te geven, maar dat is tot nu toe nog niemand gelukt.

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP} \qquad \text{(of: } \mathcal{NPC} \stackrel{?}{\subset} \mathcal{P}\text{)}$$

(`https://www.claymath.org/millennium-problems`)

# NPC



"I can't find an efficient algorithm, I guess I'm just too dumb."

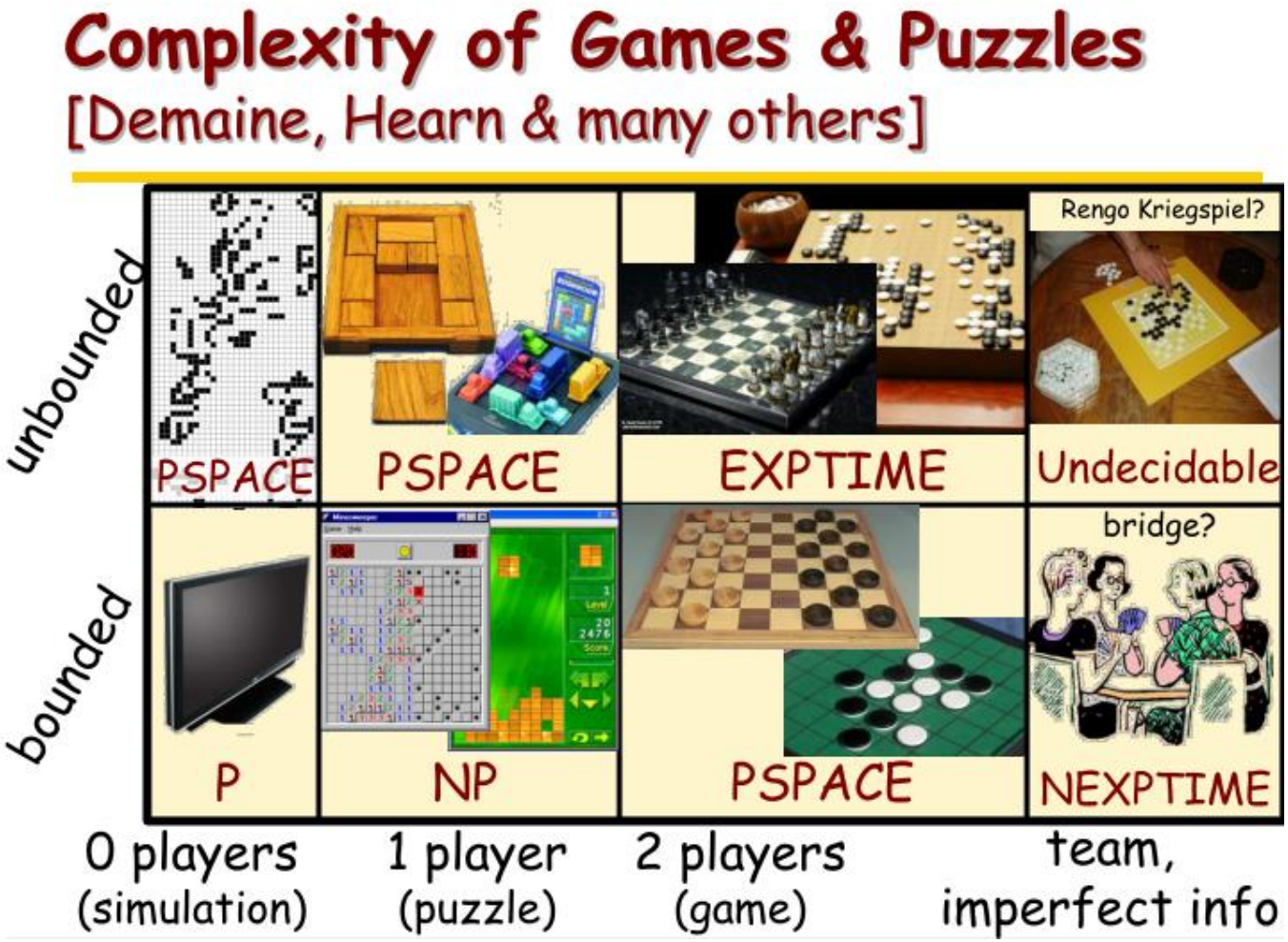
"I can't find an efficient algorithm, because no such algorithm is possible!"

# NPC

"I can't find an efficient algorithm, but neither can all these famous people."

# En verder?

Afhankelijk van hoe het semester loopt nog wat verdieping aan het einde.



# Om te onthouden van vandaag

- basisoperaties
- ↳ complexiteit van algoritmen
  - $O$, $\Omega$ en $\Theta$
  - best, worst en average case
- complexiteit van problemen

# Eerste helft semester

- Volgende week: complexiteit en optimaliteit van zoekalgoritmen
- Tot eind maart: technieken voor ondergrenzen, sorteren en een handjevol andere problemen

# (Werk)college

**Volgende college**: dinsdag 14 februari, 9u00–10u45, zaal C1 (Gorlaeus)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304, 306–308 en 303 (Snellius)
Opgaven uit het dictaat: 1, 2, 3abcde, 4, 6

# Huiswerk

**Huiswerk 1**

Verschijnt vanmiddag (ergens na het werkcollege) op de website.

Inleveren via Brightspace, voor het begin van hoorcollege 4, d.w.z. vóór 28 februari 09u00 (drie weken).

# Complexiteit 2023 — college 2

14 februari 2023

---

## Zoeken

# Vorige keer

- basisoperaties
- ↳ complexiteit van algoritmen
  - $O$, $\Theta$ en $\Omega$
  - best, worst en average case
- complexiteit van problemen

# Vandaag

We bekijken een aantal zoekalgoritmen, waarvan we de complexiteit vergelijken. Met behulp van beslissingsbomen bewijzen we later een ondergrens op de complexiteit.

- zoeken met behulp van sleutelvergelijkingen
- zoekalgoritmen:
  - ongeordend lineair zoeken (opgave 3)
  - geordend lineair zoeken (opgave 4)
  - jump search
  - binair zoeken (opgave 5)

# Zoeken

## Probleem

Gegeven een waarde $X$ en een array $A = A[1], \ldots, A[n]$. Bepaal of $X$ in $A$ zit. Zo ja, geef de index terug; zo nee, geef $-1$ terug.

Basisoperatie: sleutelvergelijkingen van de vorm

```
1 if X = A[i] then                    /* binaire vergelijking */
2 |   gevonden;
3 else
4 |   ...;
```

```
1 if X = A[i] then                   /* drie-weg vergelijking */
2 |   gevonden;
3 else if X < A[i] then
4 |   ...;
5 else
6 |   ...;
```

# Sleutelvergelijking

- een sleutelvergelijking doet voor iedere aanroep maximaal twee ja/nee-vergelijkingen
- kost dus $\Theta(1)$ tijd per aanroep
- de drie-weg vergelijking kost in orde van grootte even veel tijd als de binaire
- we tellen beide soorten sleutelvergelijkingen dan ook als één vergelijking

# Ongeordend lineair zoeken (OLZ)

**Probleem**

Zoek $X$ in een willekeurig array $A = A[1], \ldots, A[n]$.

**Algoritme** (zie opgave 3):

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

# OLZ: complexiteit

Beste case

- je kunt al bij de eerste vergelijking prijs hebben: $1 \in \Theta(1)$

Worst case

- als $X$ niet voorkomt in $A$ of helemaal achteraan: $n \in \Theta(n)$

Average case

- laat $q$ de kans zijn dat $X$ voorkomt in $A$
- iedere positie in $A$ is even waarschijnlijk
- dan worden er in de average case
  $q \times \frac{1}{2}(n+1) + (1-q) \times n \in \Theta(n)$ sleutelvergelijkingen gedaan

# OLZ: optimaliteit

Complexiteit van het probleem (opgave 3e)
Elk algoritme dat een waarde $X$ zoekt in een (willekeurig) array $A$ met $n$ elementen, en dat alleen gebruik maakt van sleutelvergelijkingen ($X =, < A[i]$), doet in de worst case ten minste $n$ vergelijkingen*. Bewijs uit het ongerijmde, te vinden op de website.

Algoritme en optimaliteit
Ongeordend lineair zoeken voldoet aan de eisen van de stelling en doet in de worst case $n$ vergelijkingen. Het is dus optimaal.

*De stelling gaat over algoritmen die werken op alle mogelijke invoerrijtjes (en te zoeken $X$). In het bewijs wordt expliciet gebruikt dat je geen informatie hebt over de invoer, behalve wat je leert uit de sleutelvergelijkingen. Het bewijs gaat niet op voor algoritmen die zijn gemaakt voor een speciaal soort invoer en daarvan gebruik maken.

# Geordend lineair zoeken (GLZ)

**Probleem**

Zoek $X$ in een oplopend gesorteerd array $A = A[1], \ldots, A[n]$.

**Algoritme** (zie opgave 4):

```
1 index := 1;
2 while index ≤ n and A[index] < X do    // basisoperatie
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
```

# GLZ: complexiteit

Best case is nog steeds $1 \in \Theta(1)$

Worst case is nog steeds $n \in \Theta(n)$

- wel kan je vaak eerder stoppen t.o.v. ongeordend zoeken: alleen $n$ vergelijkingen als $X > A[n-1]$

Average case

- laat $q$ de kans zijn dat $X$ voorkomt in $A$
- als $X \in A$: alle $n$ posities even waarschijnlijk
- als $X \notin A$: alle $n+1$ gaten even waarschijnlijk

$$\frac{n}{2} + \frac{n}{n+1} + q \times (\frac{1}{2} - \frac{n}{n+1}) \in \Theta(\frac{n}{2})$$

# Jump search (JS)

$A$ is weer oplopend gesorteerd; kies $k$ met $1 \leq k < n$

```
1 index := k;
  // index is altijd een k-voud
  // vergelijk X met A[k], A[2k], A[3k], ...
2 while index ≤ n and A[index] < X do
3 |   index := index + k;
4 od
5 if index ≤ n then
  |   // A[index − k] < X ≤ A[index]
6 |   lineair zoeken van X in A[index − k + 1], ..., A[index];
7 else
  |   // A[index − k] < X ≤ A[n] of X > A[n]
8 |   lineair zoeken van X in A[index − k + 1], ..., A[n];
9 fi
```

Vergelijk: zoeken in een woordenboek

# JS: complexiteit

Worst case

- $\lfloor \frac{n}{k} \rfloor + k$ sleutelvergelijkingen

Beste keus: $k = \lceil \sqrt{n} \rceil$
Dan doet jump search in het slechtste geval $\Theta(\sqrt{n})$ sleutelvergelijkingen. Dat is beter dan geordend lineair zoeken.

**Vraag**: kan zoeken in een geordend array nog beter?

**Antwoord**: ja, namelijk binair zoeken.

# Binair zoeken (BZ)

```
1  Links := 1; Rechts := n;
2  while Links ≤ Rechts do
3  |   Midden := ⌊(Links+Rechts)/2⌋;
4  |   if X = A[Midden] then
5  |   |   return Midden ;                          // gevonden
6  |   else if X < A[Midden] then
7  |   |   Rechts := Midden − 1 ;    // verder in linkerstuk
8  |   else
9  |   |   Links := Midden + 1 ;    // verder in rechterstuk
10 |   fi
11 od
12 return −1;
```

# BZ: worst case

Zoals besproken tellen we de achtereenvolgende tests $X = A[Midden]$ en $X < A[Midden]$ als één sleutelvergelijking.

Worst case: $\lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$ vergelijkingen

Zij $W(n)$ het aantal drie-weg-vergelijkingen van de vorm $X =, < A[Midden]$ dat het algoritme doet in het slechtste geval. Dan geldt:

- $W(n) = 1 + W(\lfloor n/2 \rfloor)$
- de oplossing van deze recurrente betrekking, met $W(1) = 1$, wordt gegeven door: $W(n) = \lceil \lg(n+1) \rceil = \lfloor \lg n \rfloor + 1$.

  Het bewijs gaat met volledige inductie (opgave 5) en is te vinden op de website.

# BZ: average case

Average case (voor $n = 2^k - 1$): gemiddeld aantal vergelijkingen nodig om $X$ te vinden in $A$ is

- $\frac{1}{n}\sum_{i=0}^{k-1}(i+1)2^i$ als $X$ voorkomt in $A$*
- en dit is gelijk aan $\frac{1}{n}((k-1)2^k+1)$ (opgave 12)
- $k$ als $X$ niet voorkomt in $A$†
- in totaal dus $\frac{q}{n}\sum_{i=0}^{k-1}(i+1)2^i + (1-q)k \in \Theta(\lg(n+1))$, met $q$ de kans dat $X$ voorkomt in $A$.

*onder de aanname dat elke positie even waarschijnlijk is

†het kost altijd $k$ vergelijkingen om dat te constateren

# Zoeken: samengevat

- Ongeordend lineair zoeken: $\Theta(n)$ sleutelvergelijkingen worst case, $\Theta(n)$ average case
- Geordend lineair zoeken: $\Theta(n)$ sleutelvergelijkingen worst case, $\Theta(\frac{n}{2})$ average case
- Jump search: $\Theta(\sqrt{n})$ sleutelvergelijkingen worst case
- Binair zoeken: $\Theta(\lg n)$ sleutelvergelijkingen worst case, $\Theta(\lg n)$ average case

Zou het nog sneller kunnen dan dat?

# Optimaliteit

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

Bewijs met behulp van een beslissingsboomargument.

Gevolg. Binair zoeken is optimaal wat betreft de worst case.

---

*ook als dat speciaal is toegespitst op reeds gesorteerde arrays

†van de vorm $X =, < A[i]$

# Om te onthouden van vandaag

- ongeordend lineair zoeken: $\Theta(n)$ (worst case), optimaal (voor sleutelvergelijkingen en willekeurige arrays)
- geordend lineair zoeken: $\Theta(n)$ (worst case)
- jump search: $\Theta(\sqrt{n})$ (worst case)
- binair zoeken: $\Theta(\lg n)$ (worst case), optimaal (voor sleutelvergelijkingen)

# (Werk)college

**Volgende college**: dinsdag 21 februari, 9u00–10u45, zaal C1 (Gorlaeus)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304, 306–308 en 303 (Snellius)
Opgaven uit het dictaat: 6, 17, 8, 20, 19

De website heeft uitwerkingen van o.a. opgave 6. Gebruik die kennis verstandig.

# Huiswerk

- $\lg n$ in pgfplots: `log2(x)`
- deadline (vóór hoorcollege 4, 28 februari 9u00) is strikt (daarna nog wel feedback, maar geen cijfer)

# Complexiteit 2023 — college 3

21 februari 2023

---

## Ondergrenzen

# Vorige keer

- ongeordend lineair zoeken: $\Theta(n)$ (worst case), optimaal (voor sleutelvergelijkingen en willekeurige arrays)
- geordend lineair zoeken: $\Theta(n)$ (worst case)
- jump search: $\Theta(\sqrt{n})$ (worst case)
- binair zoeken: $\Theta(\lg n)$ (worst case), optimaal (voor sleutelvergelijkingen)

# Vandaag

Technieken om ondergrenzen aan te tonen:

- beslissingsbomen
- adversary-argument

En:

- toernooimethode

# Optimaliteit binair zoeken

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

Bewijs met behulp van een beslissingsboomargument.

Gevolg. Binair zoeken is optimaal wat betreft de worst case.

---

*ook als dat speciaal is toegespitst op reeds gesorteerde arrays

†van de vorm $X =, < A[i]$

# Binaire bomen

De volgende stelling geeft een verband aan tussen de hoogte van een binaire boom en het aantal knopen (resp. bladeren). We kiezen de definitie van niveau zo, dat de wortel op niveau 0 zit. De hoogte van de boom (d.w.z. het hoogste niveau dat voorkomt in de boom) is dan gelijk aan het aantal niveaus $-\,1$.

**Stelling**

Gegeven een binaire boom met $n$ knopen, $b$ bladeren en hoogte $h$. Er geldt:

1. $h \geq \lceil \lg b \rceil$
2. $h \geq \lceil \lg(n+1) \rceil - 1 = \lfloor \lg n \rfloor$

# Algoritme → boom: type 1

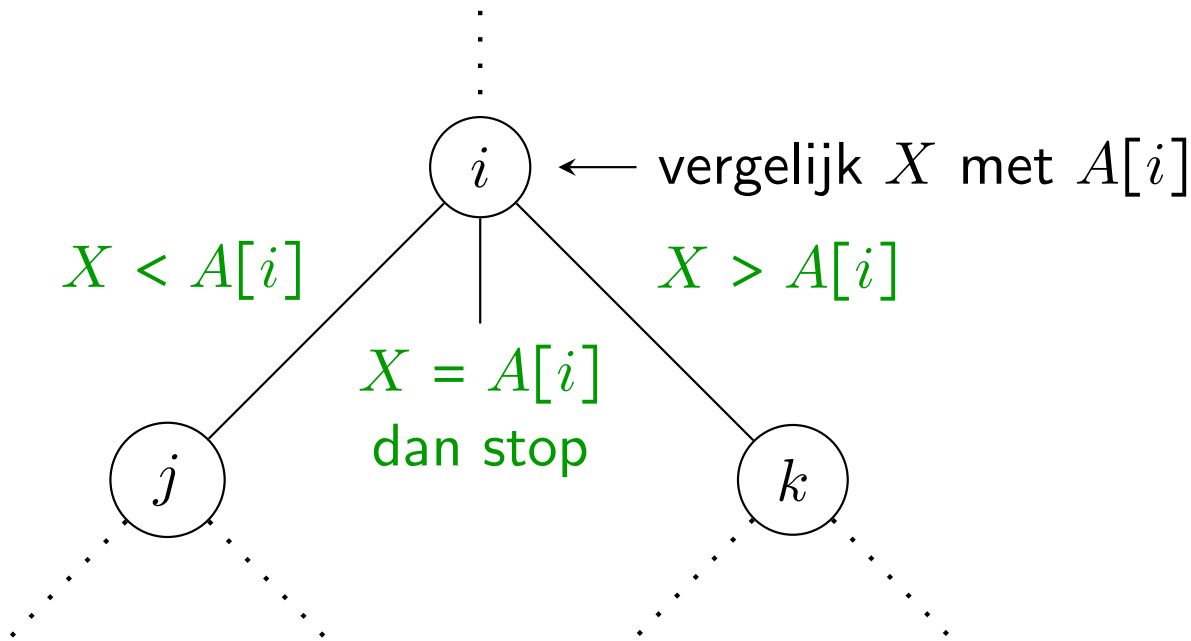
algoritme gebaseerd op het doen van sleutelvergelijkingen
$X =, < A[i]$

$$\Downarrow$$

beslissingsboom:

- binaire boom waarin knopen corresponderen met sleutelvergelijkingen
- een pad vanaf de wortel naar een willekeurige knoop correspondeert met een executie van het algoritme, d.w.z. de achtereenvolgende vergelijkingen van het algoritme op zekere invoer
- het aantal knopen op zo'n pad is het aantal sleutelvergelijkingen dat het algoritme doet op de betreffende invoer

# Beslissingsboom type 1




Beslissingsboom voor algoritmen gebaseerd op **sleutelvergelijkingen**: beschrijft de werking op elke mogelijke invoer

# Beslissingsboom type 1

- wortel van de boom op niveau $0$
- $h$ = hoogte (hoogste niveau dat voorkomt)
- knopen: sleutelvergelijkingen
- $N$ = aantal knopen
- $b$ = aantal bladeren (nu nog niet relevant)
- in een binaire boom: $h \geq \lceil \lg(N+1) \rceil - 1 = \lfloor \lg N \rfloor$

Wat stelt $h$ voor en wat weten we van $N$ bij zoekalgoritmen gebaseerd op sleutelvergelijkingen (corresponderend met deze beslissingsbomen)?

# Beslissingsbomargument

## Bewijs stelling ondergrens zoeken

- Het zoekalgoritme werkt voor elke mogelijke invoer, dus in het bijzonder voor het geval dat $A$ allemaal verschillende waarden bevat.
- Merk nu op dat elke $A[i]$ door het algoritme moet kunnen worden gevonden; $X$ kan immers op elke positie in het array voorkomen. Dat betekent dat er voor elke waarde $A[i]$ (en dus voor elke index $i$) ten minste één corresponderende knoop moet zijn, dus dat we ten minste $n$ verschillende knopen moeten hebben in de beslissingsboom. Derhalve is $N \geq n$.
- Daaruit volgt (zie eerdere slides) dat $h \geq \lfloor \lg n \rfloor$.
- Omdat het aantal vergelijkingen in de worst case gelijk is aan $h + 1$, geldt dat we (in de worst case) gegarandeerd ten minste $\lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$ sleutelvergelijkingen moeten doen.

# Algoritme → boom: type 2

algoritme gebaseerd op het doen van arrayvergelijkingen
$A[i] < A[j]$*

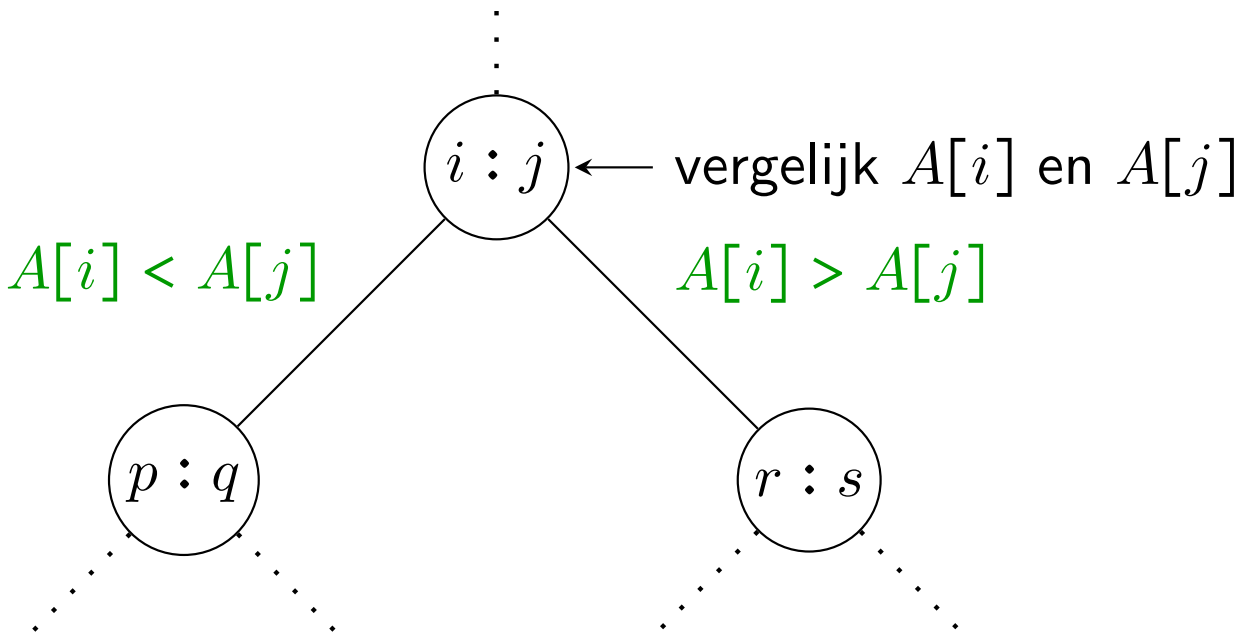
$$\Downarrow$$

beslissingsboom:

- binaire boom waarin de interne knopen corresponderen met arrayvergelijkingen en de bladeren met het eindresultaat
- een pad vanaf de wortel naar een blad correspondeert met een executie van het algoritme, dus de achtereenvolgende vergelijkingen die het algoritme doet voor zekere invoer

*we mogen z.v.v.a. aannemen dat alle $A[i]$ verschillend zijn

# Beslissingsboom type 2




vergelijk vervolgens $A[p]$ en $A[q]$

vergelijk vervolgens $A[r]$ met $A[s]$

Beslissingsboom voor algoritmen gebaseerd op **arrayvergelijkingen**: beschrijft de werking op elke mogelijke invoer

# Beslissingsboom type 2

- wortel van de boom op niveau $0$
- $h$ = hoogte (hoogste niveau dat voorkomt*)
- interne knopen: arrayvergelijkingen
- bladeren: eindresultaten; algoritme stopt hier
- $b$ = aantal bladeren
- in een binaire boom: $h \geq \lceil \lg b \rceil$
- aantal vergelijkingen in de worst case is $h$

Wat stelt $h$ voor en wat weten we van $N$ bij zoekalgoritmen gebaseerd op sleutelvergelijkingen (corresponderend met deze beslissingsbomen)?

*inclusief de speciale bladeren

# Maximum

**Stelling**

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

**Merk op**

We hadden voor het opsporen van het maximum (of het minimum) al een scherpere ondergrens gevonden, namelijk $n - 1$.

# Maximum en minimum

Stelling (zie opgave 27a)
Elk algoritme dat de (indices van de) grootste en de kleinste waarde bepaalt uit een array met $n$ elementen, en dat uitsluitend gebaseerd is op het doen van arrayvergelijkingen, doet ten minste $\lceil 2 \lg(n-1) \rceil$ vergelijkingen in de worst case.

**Merk op**
Met behulp van een adversary-argument kunnen we een scherpere ondergrens vinden, namelijk $\lceil \frac{3n}{2} \rceil - 2$.

# Beslissingsboom type 2

- voor algoritmen gebaseerd op arrayvergelijkingen
- $h$ = aantal vergelijkingen in de worst case
- $b$ = aantal bladeren
- de bladeren bevatten de eindresultaten / eindantwoorden van het algoritme voor alle mogelijke invoeren
- in een binaire boom: $h \geq \lceil \lg b \rceil$
- $b \geq$ aantal eindantwoorden dat kan voorkomen!

# Opgave 28

Gegeven twee oplopend gesorteerde even lange rijen $A$ en $B$ met in totaal $2n$ verschillende getallen. Gevraagd wordt het $n$-de getal (in volgorde van *klein naar groot*) van de in totaal $2n$ elementen van $A$ en $B$.

**b.** Bewijs met behulp van een beslissingsboomargument dat *elk* algoritme dat dit probleem voor het gegeven soort rijtjes 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 $\frac{3n}{2}$ ($n$ is hier even). Net als in **b.** kunnen we een ondergrens bewijzen voor de worst case voor het probleem met dit soort invoerrijtjes. Wat verandert er dan in het bewijs bij **b.** en welke ondergrens levert dat op?

# Selectieprobleem

## Probleem

Gegeven een array $A = A[1], \ldots, A[n]$ met $n$ verschillende getallen. Gegeven verder een geheel getal $k$ met $1 \leq k \leq n$. 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 (in volgorde van klein naar groot).

## Klasse van algoritmen

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

# Selectie: complexiteit

De **complexiteit** van het **probleem**:

Ondergrens

Elk algoritme gebaseerd op arrayvergelijkingen doet voor het selectieprobleem in de worst case altijd ten minste $\lceil \lg n \rceil$ vergelijkingen (beslissingsboomargument). Selectie is dus $\Omega(\lg n)$.

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 (zie later), dus selectie is $O(n \lg n)$.

**Opmerkingen**

Beide grenzen kunnen scherper. We bekijken hierna steeds (het precieze aantal vergelijkingen in) de worst case.

# Speciale gevallen 1

1. $k = n$: het maximum, of
   $k = 1$: het minimum.

   Dit kan met $n - 1$ vergelijkingen. Dat is optimaal (al gezien)!

2. (Variant, met $n \geq 2$) Het maximum en minimum beide opsporen.

   Voor de hand ligt een algoritme met $2n - 3$ vergelijkingen, maar het kan met $\lceil \frac{3n}{2} \rceil - 2$. Dat is optimaal (adversary-argument, zie later).

# Maximum en minimum: een optimaal algoritme

```
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;                                  // voor het gemak: n even
6  while i < n do                          // anders kleine aanpassing
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 |   if gr > grootste then
12 |   |   grootste := gr;
13 |   if kl < kleinste then
14 |   |   kleinste := kl;
15 |   i := i + 2;
```

# Speciale gevallen 2

3. $k = n - 1$: de op één na grootste (dus $n \geq 2$)

   Voor de hand ligt een algoritme met $2n - 3$ vergelijkingen, maar het kan met $n + \lceil \lg n \rceil - 2$ vergelijkingen (toernooimethode). Dit is optimaal. (Hier onbewezen, maar kan met een adversary-argument.)

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

   Er zit een gat tussen de best bekende ondergrens (ongeveer $2n$) en het best bekende algoritme. We kunnen dus niets zeggen over de optimaliteit van dat algoritme. Zie ook opgave 23.

# Toernooimethode

De toernooimethode* is een *optimaal* algoritme voor het vinden van de op één na grootste.

Terminologie:

wedstrijd ↔ arrayvergelijking;

winnaar ↔ grootste van de twee;

speler ↔ array-element; etcetera.

Complexiteit:
De toernooimethode doet $n + \lceil \lg n \rceil - 2$ arrayvergelijkingen. Dit is optimaal (worst case)†.

Geschikte implementatie:
Met behulp van een heap-achtige structuur die de "uitslagen" van het toernooi weergeeft (opgave 29).

---

*Jósef Schreier, 1932

†S. S. Kislitsin, 1964

# Toernooimethode

Algoritme (voor het gemak met $n = 2^\ell$):

- 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 $\ell$ rondes nodig ($\ell = \lg n$).

# Toernooimethode

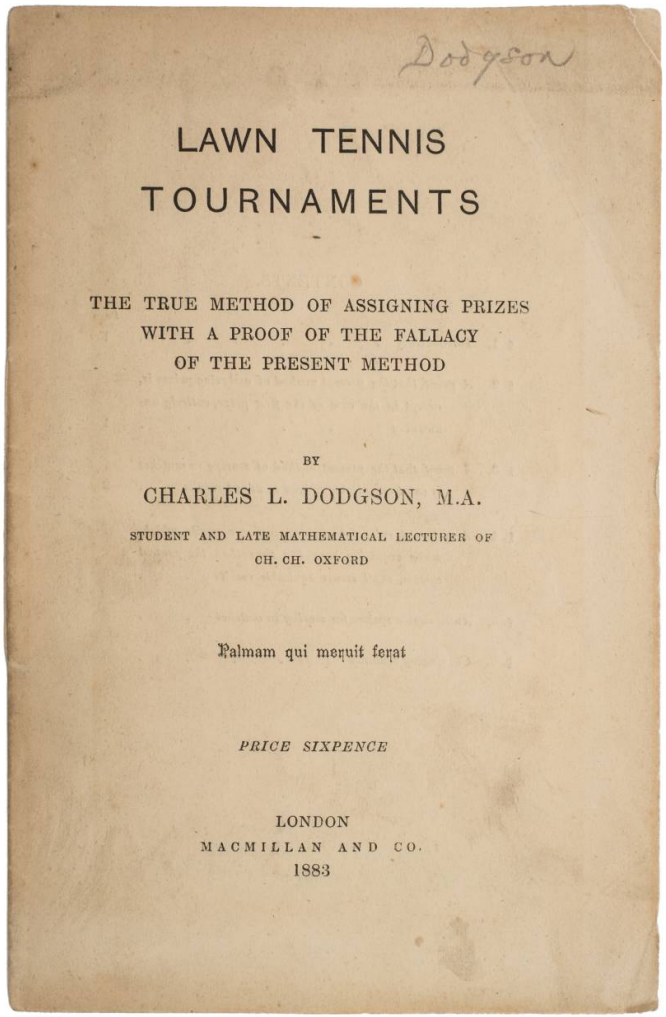
Algoritme (vervolg):

- nu moet de op één na grootste nog worden gevonden.
- dit moet een van de $\ell$ spelers zijn die in het toernooi heeft verloren van de grootste, en wel de grootste van die $\ell$.
- het kost $\ell - 1$ vergelijkingen om deze te bepalen.

Als $n = 2^\ell$ doet de toernooimethode dus $n + \lg n - 2$ arrayvergelijkingen in totaal. Dit is optimaal (worst case).

**Opmerking**: als $n$ geen tweemacht is, is een kleine aanpassing nodig. Het aantal vergelijkingen wordt daarmee $n + \lceil \lg n \rceil - 2$.

# Even tussendoor



Dodgson

LAWN TENNIS
TOURNAMENTS

THE TRUE METHOD OF ASSIGNING PRIZES
WITH A PROOF OF THE FALLACY
OF THE PRESENT METHOD

BY

CHARLES L. DODGSON, M.A.

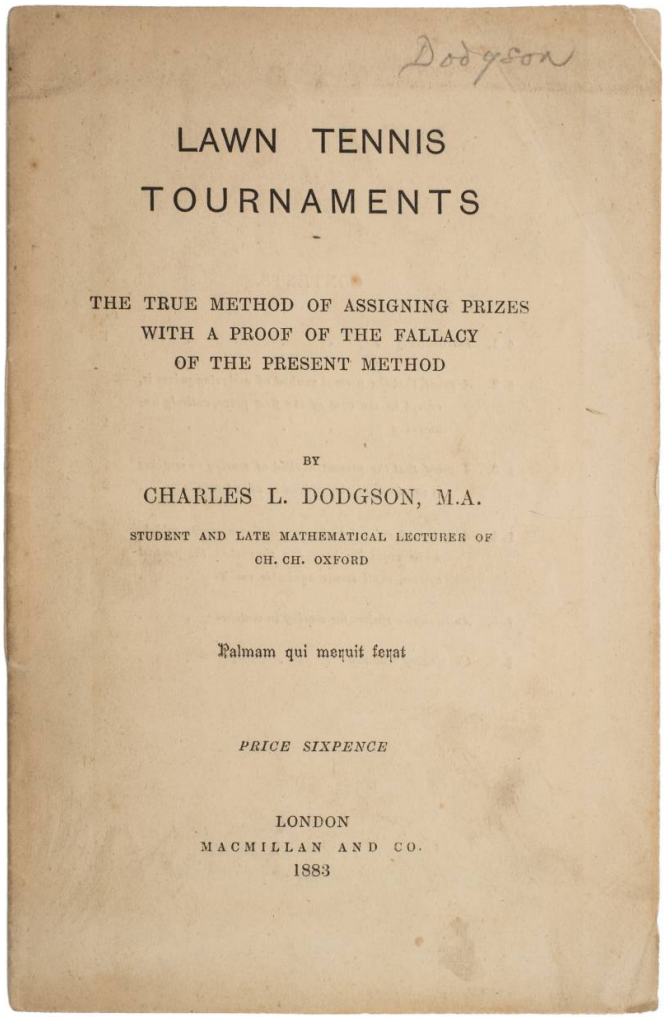
STUDENT AND LATE MATHEMATICAL LECTURER OF
CH. CH. OXFORD

Palmam qui meruit ferat

*PRICE SIXPENCE*

LONDON
MACMILLAN AND CO.
1883

# Even tussendoor


Dodgson

LAWN TENNIS
TOURNAMENTS

THE TRUE METHOD OF ASSIGNING PRIZES
WITH A PROOF OF THE FALLACY
OF THE PRESENT METHOD

BY

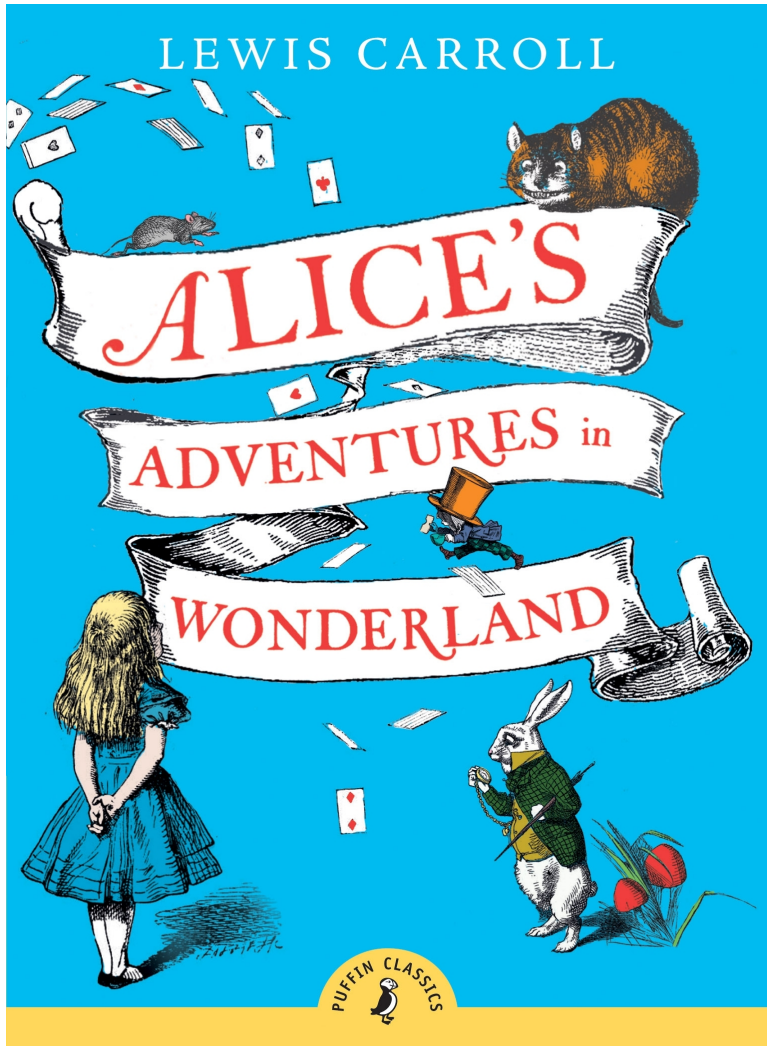
CHARLES L. DODGSON, M.A.

STUDENT AND LATE MATHEMATICAL LECTURER OF
CH. CH. OXFORD

Palmam qui meruit ferat

*PRICE SIXPENCE*

LONDON
MACMILLAN AND CO.
1883



# Ondergrens complexiteit

- doel: bewijzen van een ondergrens op de worst case complexiteit van een probleem
- dus: bewijzen dat elk algoritme voor het probleem ten minste ... stappen nodig heeft in de worst case
- dat kan met behulp van beslissingsbomen, maar het kan ook anders
- het is voldoende om voor elk algoritme een "bad case" invoer te geven (of te weten dat er een bestaat) waarop dat algoritme minstens ... stappen doet
- een manier om dit voor elkaar te krijgen is met behulp van een adversary-argument

# Adversary-argument

Een algoritme speelt een vraag-en-antwoord-spel tegen een adversary (tegenstander).

- algoritme: wil zo veel mogelijk informatie krijgen om met zo weinig mogelijk vragen het probleem op te lossen
- 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: consistentie. De adversary geeft alleen antwoorden die consistent zijn met eerder gegeven informatie.

**Voorbeelden**:

- Wie is het?, Galgje, Mastermind, Zeeslag, ...
- $X$ zoeken in een rij met $n$ verschillende elementen (zie college)
- sorteren

# Adversary-argument

- 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 zo tijdens de uitvoering van een algoritme a.h.w. een bad case -invoer op
- de adversary(-strategie) zorgt ervoor dat hij op elk moment een invoer kan geven die consistent is met de al gegeven antwoorden
- het aantal stappen (vragen) dat *elk 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

# Ondergrens sorteren

Stelling

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

Opmerking: we bewijzen een ondergrens voor de worst case. We mogen daarom aannemen dat alle array-elementen (bijv. getallen) verschillend zijn (of zelfs dat het gaat om 1 t/m n).

We bewijzen de ondergrens via een adversary-argument.

Bovengenoemde ondergrens is scherp: sorteren kán ook in $O(n \lg n)$.

# Adversary en sorteren

- sorteren komt neer op het vinden van de juiste (d.w.z. oplopende) ordening van de array-elementen (bijvoorbeeld $A[3] < A[2] < A[4] < A[1]$)
- er zijn voor een rijtje met $n$ verschillende waarden precies $n!$ mogelijke ordeningen
- de adversary heeft $n!$ kaarten in zijn hand, met op iedere kaart een mogelijke ordening
- na iedere vraag $A[i] \stackrel{?}{<} A[j]$ van het algoritme geeft de adversary antwoord: ja of nee
- de adversary gooit vervolgens alle kaarten weg die inconsistent zijn met dit antwoord

# Adversary en sorteren

- de adversary probeert het spelletje zo lang mogelijk te rekken, opdat het algoritme zo veel mogelijk vragen moet stellen om het antwoord (zeker) te weten
- strategie van de adversary: kies altijd het antwoord dat de meeste kaarten overlaat; kies ja als het niet uitmaakt
- het algoritme is klaar als er nog maar één kaart (ordening) over is
- wat is dan het *minimale* aantal vragen dat de adversary het algoritme kan dwingen te stellen, zodanig dat er maar één ordening overblijft?

# Adversary en sorteren

- omdat de adversary telkens de meeste kaarten probeert over te laten, kan het algoritme bij iedere vraag maximaal de *helft* van de kaarten wegspelen
- het aantal vragen dat het algoritme moet stellen is daarom gegarandeerd minimaal $\lceil \lg n! \rceil \in \Theta(n \lg n)$

Conclusie: voor iedere volgorde van vragen van het algoritme kunnen we zo een bijbehorende *bad case-invoer*[*] construeren die ten minste $\Theta(n \lg n)$ vragen vereist

[*] namelijk een rijtje corresponderend met de ordening die als laatste overbleef tegen de adversary

# Adversary-argument

Een adversary-argument wordt gebruikt om een ondergrens te vinden voor de worst case-complexiteit van een probleem. De adversary

- "speelt" tegen een algoritme volgens een adversary-strategie
- probeert het algoritme zo veel mogelijk vragen te laten stellen
- antwoordt altijd consistent met eerdere antwoorden
- bouwt zo tegen elk algoritme een bad case-invoer op

Het doel is om een ondergrens $f(n)$ af te leiden voor het aantal stappen dat elk algoritme nodig heeft tegen de adversary. Dat betekent dat er voor *elk algoritme* een invoer bestaat waarop minstens $f(n)$ stappen nodig zijn. In dat geval is het aantal stappen in de worst case voor elk algoritme dus $\geq f(n)$.

Merk op dat je niet hoeft te weten hoe zo'n bad case er uitziet, alleen dat hij bestaat.

# Maximum en minimum

**Stelling**

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

**Bewijs**

De status van een array-element geven we aan met:

1. W: $\geq 1$ keer gewonnen, nooit verloren
2. V: $\geq 1$ keer verloren, nooit gewonnen
3. WV: $\geq 1$ keer gewonnen en $\geq 1$ keer verloren
4. N: nog nooit "gespeeld"

De adversary-strategie is gebaseerd op de status van de array-elementen op het moment dat die worden vergeleken door het algoritme.

# Adversary-strategie

| 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 | $x > y$ | — | — | — | — | |
| | begin | $n$ | $0$ | $0$ | $0$ | |
| | eind | $0$ | $1$ | $1$ | $n-2$ | |

# Toelichting

Merk op dat de uitslagen altijd kunnen, want de waarde van een V-element (resp. W-element) mag altijd ongestraft omlaag (resp. omhoog); de eerder gegeven antwoorden blijven dan geldig.

| vergelijking $x$–$y$ | actie van de adversary |
|---|---|
| N–N | kies "frisse" waarden zodat $x > y$ |
| W–N | kies "frisse" waarde voor $y$ ($< x$) |
| W–V | $x$ mag omhoog of $y$ omlaag zodat $x > y$ |
| W–W | laat de grootste winnen |
| WV–V | $y$ mag omlaag |

# Bewijs ondergrens

- om van de beginsituatie naar de eindsituatie te komen *moeten* er $n-2$ vergelijkingen van type 2 worden gedaan, want dat is de enige manier om $n-2$ WV's te krijgen
- tevens *moeten* er minstens $\lceil \frac{n}{2} \rceil$ vergelijkingen van type 1 worden gedaan 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

# Om te onthouden van vandaag

- beslissingsboomargument:
  - hoogte boom $h$, worst case $h + 1$
  - $n$ knopen: $h \geq \lceil \lg(n + 1) \rceil - 1 = \lfloor \lg n \rfloor$
  - $b$ bladeren: $h \geq \lceil \lg b \rceil$
  - "makkelijk", niet altijd even scherp
- adversary-argument
  - zie het probleem als een twee-spelerspel
  - adversary (tegenstander) houdt jou zo lang mogelijk bezig
  - adversary-strategie geeft ondergrens op worst case
- toernooimethode: optimaal algoritme voor het vinden van de op één na grootste

# (Werk)college

**Volgende college**: dinsdag 28 februari, 9u00–10u45, zaal C1 (Gorlaeus)

Opnamen van vorig jaar komen binnenkort op Brightspace.

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304, 306–308 en 303 (Snellius)
Opgaven uit het dictaat: 20, 5d, 27, 22, 32

**Extra werkcollege/vragenuur**: zodadelijk van 13u15 tot 14u00, computerzaal 302–304 (Snellius)

Opmerking: hoeveelheid en volgorde opgaven zijn een advies, het is niet de verwachting dat iedereen dit elke week af krijgt.

# Huiswerk

Inleveren op Brightspace, vóór hoorcollege 4 (28 februari 9u00).
Daarna nog wel feedback maar geen cijfer.

# Complexiteit 2023 — college 4

28 februari 2023

---

Selectie is $O(n)$

Insertion sort

Recurrente betrekkingen

# Vorige keer

- beslissingsboomargument:
  - hoogte boom $h$, worst case $h + 1$
  - $n$ knopen: $h \geq \lceil \lg(n+1) \rceil - 1 = \lfloor \lg n \rfloor$
  - $b$ bladeren: $h \geq \lceil \lg b \rceil$
  - "makkelijk", niet altijd even scherp
- adversary-argument
  - zie het probleem als een twee-spelerspel
  - adversary (tegenstander) houdt jou zo lang mogelijk bezig
  - adversary-strategie geeft ondergrens op worst case
- toernooimethode: optimaal algoritme voor het vinden van de op één na grootste

# Vandaag

- Een lineair algoritme voor het selectieprobleem
- Insertion sort
- Recurrente betrekkingen

# Selectieprobleem is $O(n)$

## Probleem

Gegeven een array $A = A[1], \ldots, A[n]$ met $n$ verschillende getallen, en een geheel getal $k$ met $1 \leq k \leq n$. Gevraagd het $k$-de element in grootte (in volgorde van klein naar groot). Formeler: gevraagd het element $A[i]$ dat groter is dan precies $k - 1$ andere $A[j]$'s.

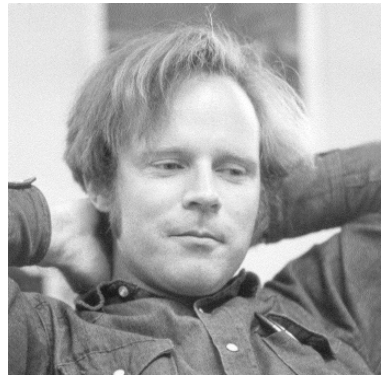
## Complexiteit

Het selectieprobleem is $O(n)$. We bewijzen dit door een algoritme te geven dat de $k$-de in grootte vindt in $O(n)$ vergelijkingen in de worst case*.

*Blum, Floyd, Pratt, Rivest, Tarjan, 1973

# Even tussendoor

Manuel Blum*
CAPTCHA†



Robert Floyd*
Floyd-Warshall†

Vaughan Pratt
Knuth-Morris-Pratt†

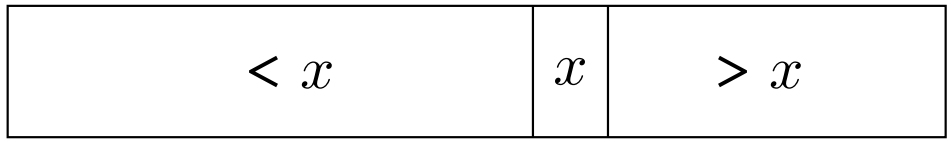
Ron Rivest*
Rivest-Shamir-Adleman†

Robert Tarjan*
Tarjan's SCC†

*Turing Award
†en heel veel meer

# Het idee

- Kies (bijv. willekeurig) een element $x$ uit het array (de spil*)
- Reorganiseer het array als volgt:




- Dit kost ongeveer $n$ arrayvergelijkingen
- De $k$-de in grootte zit in het linker- óf het rechtergedeelte
- Ga dus met één van beide gedeeltes verder en herhaal
- Met geluk: telkens twee gelijke stukken, in totaal ongeveer $n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \ldots + \frac{n}{2^m} \leq 2n$ vergelijkingen
- Average case is $O(n)$, maar worst case is $\Theta(n^2)$
- Oplossing: kies goede spillen

*Engels: pivot

# $O(n)$-algoritme

Algoritme:

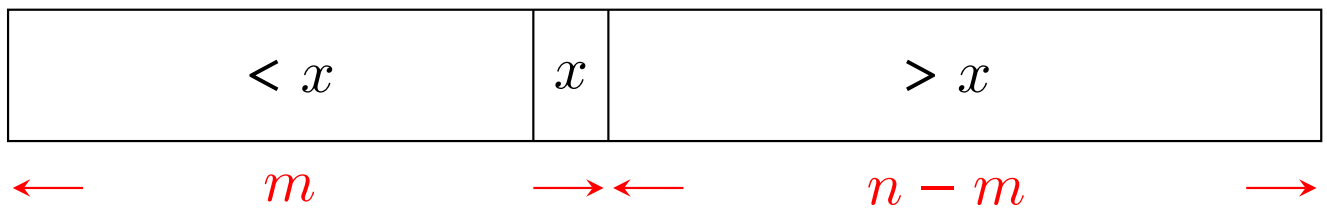
1. verdeel de getallen in $\lfloor \frac{n}{5} \rfloor$ groepjes van $5$ elementen, en één groepje met de resterende $n \bmod 5$*
2. vind de mediaan van elk van de $\lceil \frac{n}{5} \rceil$ groepjes, bijvoorbeeld met Bubblesort of opgave 24
3. vind de mediaan $x$ van de in stap 2 gevonden $\lceil \frac{n}{5} \rceil$ medianen: recursie†

*deze $5$ is niet willekeurig gekozen, maar is optimaal

†de mediaan van $\ell$ elementen is de $\lceil \frac{\ell}{2} \rceil$-de in grootte

# Vervolg algoritme

4. Partitioneer (ongeveer zoals bij Quicksort) alle elementen rond $x$. Stel dat $m$ getallen $\leq x$ zijn en $n - m$ getallen $> x$.




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

# Afschatting

Na stap 3 van het algoritme geldt:

- ten minste $3 \times (\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 \geq \frac{3n}{10} - 6$ elementen zijn groter (resp. kleiner) dan $x$.
- er zijn dus hooguit $\lceil \frac{7n}{10} \rceil + 6$ elementen $\leq x$ (resp. $\geq x$)

Gevolg: in stap 5 wordt het algoritme recursief aangeroepen op hooguit $\lceil \frac{7n}{10} \rceil + 6$ elementen.

# Voorbeeld

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 15 | 11 | 2 | 9 | 5 | 0 | 7 | 3 | 21 | 44 | 40 | 1 | 18 | 20 | 32 | 19 | 35 | 37 | 39 |
| 13 | 16 | 14 | 8 | 10 | 16 | 6 | 33 | 4 | 27 | 49 | 46 | 52 | 25 | 51 | 34 | 43 | 56 | 72 | 79 |
| 17 | 23 | 24 | 28 | 29 | 30 | 31 | 36 | 42 | 47 | 50 | 55 | 58 | 60 | 63 | 65 | 66 | 67 | 81 | 83 |
| 22 | 45 | 38 | 53 | 61 | 41 | 62 | 82 | 54 | 48 | 59 | 57 | 71 | 78 | 64 | 80 | 70 | 76 | 85 | 87 |
| 96 | 95 | 94 | 86 | 89 | 69 | 68 | 97 | 73 | 92 | 74 | 88 | 99 | 84 | 75 | 90 | 77 | 93 | 98 | 91 |

Blauw is de mediaan der medianen ($x$ in stap 3); groen is gegarandeerd lager, evenals het linker oranje gedeelte; rood en het rechter oranje gedeelte zijn gegarandeerd hoger. In het algemeen zijn gegarandeerd ongeveer $\frac{3n}{10}$ elementen uit het array kleiner resp. groter dan $x$. In het ergste geval ga je de recursie dus in op de resterende $\approx 70\%$ van het array.

# Recurrente betrekking

Laat $T(n)$ het aantal vergelijkingen zijn dat dit (recursieve) algoritme doet in de worst case. Onder de aanname dat $T(n)$ stijgend is, geldt:

$$T(n) = \begin{cases} \text{constant} & n \leq \ldots \\ T(\lceil \frac{n}{5} \rceil) + T(\lceil \frac{7n}{10} \rceil + 6) + O(n) & n > \ldots \text{(bv. } 80) \end{cases}$$

- $T(\lceil \frac{7n}{10} \rceil + 6) < n$ voor $n > 20$
- $O(n)$ komt van stappen 1, 2 en 4 samen

Bewering:

$T(n) \leq cn$ voor geschikte $c$ en $n \geq \ldots$
ofwel: $T(n) \in O(n)$

# Selectie is $\Omega(n)$

Opgave 26

Elk algoritme voor het selectieprobleem dat is gebaseerd op arrayvergelijkingen doet in de worst case ten minste $\lceil \frac{n}{2} \rceil$ van zulke vergelijkingen.

Gevolg

Het selectieprobleem is $\Omega(n)$ (en dus $\Theta(n)$).

# Sorteren

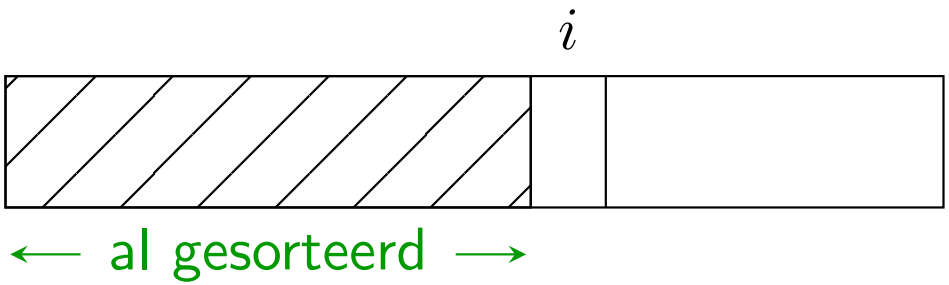
## Probleem

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

We bekijken eerst sorteeralgoritmen die zijn gebaseerd op het doen van arrayvergelijkingen.

# Insertion sort

## Insertion sort




Conceptueel idee: itereer over $i$ en voeg $A[i]$ op de juiste plek in het gesorteerde stuk $A[1], \ldots, A[i-1]$ in door herhaald te vergelijken met de linkerbuur en indien nodig te verwisselen.

# Insertion sort

Het algoritme is gebaseerd op het doen van arrayvergelijkingen ($A[i] < A[j]$).

```
1 for i := 2 to n do
  |   // nu A[i] invoegen op de juiste plek in
  |       A[1], ..., A[i − 1]
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
```

# Observaties

- het aantal arrayvergelijkingen is een goede maat voor de complexiteit
- insertion sort doet eigenlijk steeds compare-exchange-operaties: vergelijk en verwissel (indien nodig). Deze zijn hier vermomd als verschuivingen, waarna pas in de laatste stap $A[i]$ daadwerkelijk pas wordt neergezet.
- de "verwisselingen" zijn steeds buurverwisselingen

# Basisoperatie

Merk op:

- de for-loop (regel 1) doet precies $n-1$ iteraties: $\Theta(n)$
- de vergelijking $A[j] > x$ gebeurt minstens één keer per iteratie v/d for-loop, dus totaal *minstens* $n-1$ keer: $\Omega(n)$
- het vergelijken van array-elementen (want $x = A[i]$) is een zinnige operatie voor sorteren

Dan:

- regels 1, 2, 3 en 8 gebeuren elk precies $n-1 \in \Omega(n)$ keer*
- eerste deel van regel 4 ($j > 0$) gebeurt per iteratie v/d for-loop hooguit één keer vaker dan het tweede deel. Dit is dus hooguit een constante factor 2.
- regels 5 en 6 gebeuren alleen als regel 4 waar is, dus hooguit even vaak als de test op regel 4 (ook orde van grootte)

*je kan ook vinden dat regel 1 maar één keer gebeurt; dan geldt $1 \in \Omega(n)$

# Complexiteit Insertion sort

We tellen het aantal vergelijkingen $A[j] > x$ (waar $x = A[i]$).

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

*onder de aanname dat alle $A[i]$'s verschillend zijn en dat alle $n!$ permutaties (ordeningen) van $A[1], \ldots, A[n]$ even waarschijnlijk zijn. We middelen dan over alle mogelijke permutaties en dat zijn in essentie alle mogelijke invoerrijtjes.

# Best case

Het minimale aantal vergelijkingen van Insertion sort is $n - 1$.

- de eerste vergelijking op regel 4 is altijd aanvankelijk waar (want $i > 1$), dus de tweede vergelijking ($A[j] > x$) wordt elke iteratie minstens één keer uitgevoerd. Totaal $n - 1$ iteraties, dus minstens $n - 1$ keer.
- $n - 1$ kan worden gehaald: dan moet elke keer óf direct $A[i - 1] \leq A[i]$ zijn (dus oplopend gesorteerd), of $j \leq 0$ vóór de tweede vergelijking $A[j] > x$
- twéé goede rijtjes: oplopend gesorteerd, of alleen de eerste twee elementen fout.

# Worst case

Het maximale aantal vergelijkingen van Insertion sort is $\frac{1}{2}n(n-1)$.

- de tweede vergelijking op regel 4 kan hooguit gebeuren zolang $j > 0$: hooguit $i - 1$ keer. Totaal dus hooguit $\sum_{i=2}^{n}(i-1) = \frac{1}{2}n(n-1)$ keer.
- dit kan worden gehaald: dan moet elke keer voor alle $j > 1$ ook gelden $A[j] > A[i]$. Voor $j = 1$ maakt het niet meer uit.
- slechte rijtjes: alle rijtjes waarvoor voor elke $A[i]$ geldt dat $A[i]$ het kleinste of het op één na kleinste element is van $A[1], \ldots, A[i]$

# Inversies

Definitie: een inversie van de permutatie $A[1], \ldots, A[n]$ is een paar $(A[i], A[j])$ waarvoor $i < j$ en $A[i] > A[j]$. M.a.w.: een inversie is een paar $(A[i], A[j])$ dat verkeerd om staat.

Merk op: *elk* sorteeralgoritme moet *alle* aanwezige inversies opheffen.

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 $A$.

Ten slotte: een buurverwisseling (zoals bij Insertion sort) heft altijd precies één inversie op (aangenomen dat de buren verkeed om staan).

# Worst case

**Stelling**

Het maximale aantal inversies dat kan voorkomen in een rijtje van $n$ verschillende waarden is $\binom{n}{2} = \frac{1}{2}n(n-1)$. Dit treedt (o.a.) op bij een aflopend gesorteerd rijtje.

**Gevolg**

*Elk* algoritme dat sorteert met behulp van arrayvergelijkingen en dat per vergelijking hooguit één inversie opheft, doet ten minste $\frac{1}{2}n(n-1)$ vergelijkingen in de worst case.

Conclusie: Insertion sort is optimaal wat betreft de worst case, binnen de beschreven klasse van algoritmen.

# Average case

**Stelling**

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 $n!$ permutaties even waarschijnlijk zijn.

**Gevolg**

*Elk* algoritme dat sorteert met behulp van arrayvergelijkingen en dat per vergelijking hooguit één inversie opheft, doet ten minste $\frac{1}{4}n(n-1)$ vergelijkingen in de average case.

Conclusie: Insertion sort doet gemiddeld $\frac{1}{4}n(n-1) + n - 1 - \sum_{i=2}^{n} \frac{1}{i}$ vergelijkingen, dus Insertion sort is optimaal in orde van grootte wat betreft de average case, binnen de beschreven klasse van algoritmen.

# Inversies

Voor sorteeralgoritmen gebaseerd op arrayvergelijkingen, waarbij per arrayvergelijking hooguit één inversie wordt opgeheven*, geldt:

- # arrayvergelijkingen ≥ # inversies invoerarray
- # arrayvergelijkingen in de worst case ≥ $\frac{1}{2}n(n-1)$

Als je een beter sorteeralgoritme wilt (gebaseerd op arrayvergelijkingen), moet je dus elementen verwisselen die verder van elkaar liggen, zoals gebeurt bij Mergesort, Quicksort en Shellsort (en veel meer).

*zoals bij algoritmen die gebruikmaken van buurverwisselingen, zoals Insertion sort en Bubblesort

# Inversies opheffen

Stel dat $A[i]$ en $A[i+k]$ $(k > 0)$ verkeerd om staan en dat we die verwisselen. Hoeveel inversies worden dan ten minste respectievelijk ten hoogste opgeheven?

Situatie:


met $A[i] > A[i+k]$. Verwissel nu $A[i]$ en $A[i+k]$.

# Recurrente betrekkingen

- rij van Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, 21$
- vanaf het derde element: som van de voorgaande twee
- een recurrente betrekking is een voorschrift om een waarde $T(n)$ te berekenen door middel van zijn voorganger(s), dus bijvoorbeeld $T(n-1)$ of $T(\frac{n}{2})$, of ...

Voor de Fibonacci-getallen geldt:

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

# Recurrente betrekkingen

Een ander voorbeld van een recurrente betrekking:

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

- vorm een vermoeden voor een formule (of sommatie) d.m.v.:
  - herhaalde substitutie en afleiden algemene vorm; of
  - probeer wat termen door te rekenen: zie je een patroon?
- bewijs* je vermoeden met volledige inductie

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

*of niet

# Recurrente betrekkingen

De vorige recurrente betrekking, maar nu voor algemenere $n$, dus niet alleen voor tweemachten:

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

Dan geldt: $T(n) \in O(n \lg n)$ (en overigens ook $T(n) \in \Theta(n \lg n)$)

Dit kan worden bewezen door met behulp van volledige inductie bijvoorbeeld aan te tonen dat $T(n) \leq 2n \lg n$ voor alle $n > 1$.

# Recurrente betrekkingen

Ten slotte nog twee voorbeelden om op te lossen:

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

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

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

   Oplossing: $T(n) = \sqrt{n}\log_4 n = \frac{1}{2}\sqrt{n}\lg n$

# Om te onthouden van vandaag

- selectie kan in $O(n)$
- Insertion sort
  - best case: $n - 1$
  - worst case: $\frac{1}{2}n(n-1)$, optimaal voor buurvergelijkingen
  - average case: $\Theta(n^2)$, optimaal in orde van grootte voor buurvergelijkingen
- inversie: paar $(A[i], A[j])$ zodat $i < j$ maar $A[i] > A[j]$, sorteeralgoritmen moeten alle inversies opheffen
- recurrente betrekkingen
  - vorm een vermoeden (herhaalde substitutie of doorrekenen eerste $x$ termen)
  - bewijs met volledige inductie
  - oefenen

# (Werk)college

**Volgende college**: dinsdag 7 maart, 9u00–10u45, zaal C1 (Gorlaeus)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304, 306–308 en 303 (Snellius)
Opgaven uit het dictaat: 32, 14, 15, 31, 18

# Huiswerk

**Huiswerk 1**

- uitwerking komt binnenkort op de website
- cijfers volgen binnen twee werkweken op Brightspace

**Huiswerk 2**

- volgende week op de website, deadline eind maart

# Complexiteit 2023 — college 5

7 maart 2023

---

Mergesort

Quicksort

Ondergrens sorteren

# Vorige keer

- selectie kan in $O(n)$
- Insertion sort
  - best case: $n - 1$
  - worst case: $\frac{1}{2}n(n-1)$, optimaal voor buurvergelijkingen
  - average case: $\Theta(n^2)$, optimaal in orde van grootte voor buurvergelijkingen
- inversie: paar $(A[i], A[j])$ zodat $i < j$ maar $A[i] > A[j]$, sorteeralgoritmen moeten alle inversies opheffen
- recurrente betrekkingen
  - vorm een vermoeden (herhaalde substitutie of doorrekenen eerste $x$ termen)
  - bewijs met volledige inductie
  - oefenen

# Vandaag

- Recurrente betrekkingen (nog even)
- Mergesort
- Quicksort
- Ondergrens sorteren

# Recurrente betrekkingen

- het kan soms helpen om termen los te laten staan (om een sommatie te zien)
- volledige inductie:
  - *zwak*: neem aan dat het geldt voor $n - 1$*
  - *sterk*: neem aan dat het geldt voor *alle* voorgaande $n$†

Voorbeeld: opgave 14b

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

---

*of voor $\frac{n}{2}$ of ...

†zodat $n = 2^k, k \geq 1$ of ...

# Inversies

Voor sorteeralgoritmen gebaseerd op arrayvergelijkingen, waarbij per arrayvergelijking hooguit één inversie wordt opgeheven*, geldt:

- # arrayvergelijkingen ≥ # inversies invoerarray
- # arrayvergelijkingen in de worst case ≥ $\frac{1}{2}n(n-1)$

Als je een beter sorteeralgoritme wilt (gebaseerd op arrayvergelijkingen), moet je dus elementen verwisselen die verder van elkaar liggen, zoals gebeurt bij Mergesort, Quicksort en Shellsort (en veel meer).

*zoals bij algoritmen die gebruikmaken van buurverwisselingen, zoals Insertion sort en Bubblesort

# Inversies opheffen

Stel dat $A[i]$ en $A[i+k]$ $(k > 0)$ verkeerd om staan en dat we die verwisselen. Hoeveel inversies worden dan ten minste respectievelijk ten hoogste opgeheven?

Situatie:


met $A[i] > A[i+k]$. Verwissel nu $A[i]$ en $A[i+k]$.

# Verdeel en heers

Mergesort en Quicksort zijn sorteermethoden die allebei zijn gebaseerd op de verdeel-en-heers-strategie:

```
1 Sorteer(rij)::
2 |   if de rij heeft meer dan één element then
3 |   |   Verdeel de rij in twee stukken: linkerrij en rechterrij;
4 |   |   Sorteer(linkerrij);
5 |   |   Sorteer(rechterrij);
6 |   |   Combineer linkerrij en rechterrij;
7 |   fi
```

Mergesort stopt het meeste werk in de combineerstap, Quicksort in de verdeelstap. Beide sorteermethoden zijn gebaseerd op arrayvergelijkingen, maar doen niet uitsluitend buurverwisselingen, zoals Insertion sort en Bubblesort.

# Mergesort*

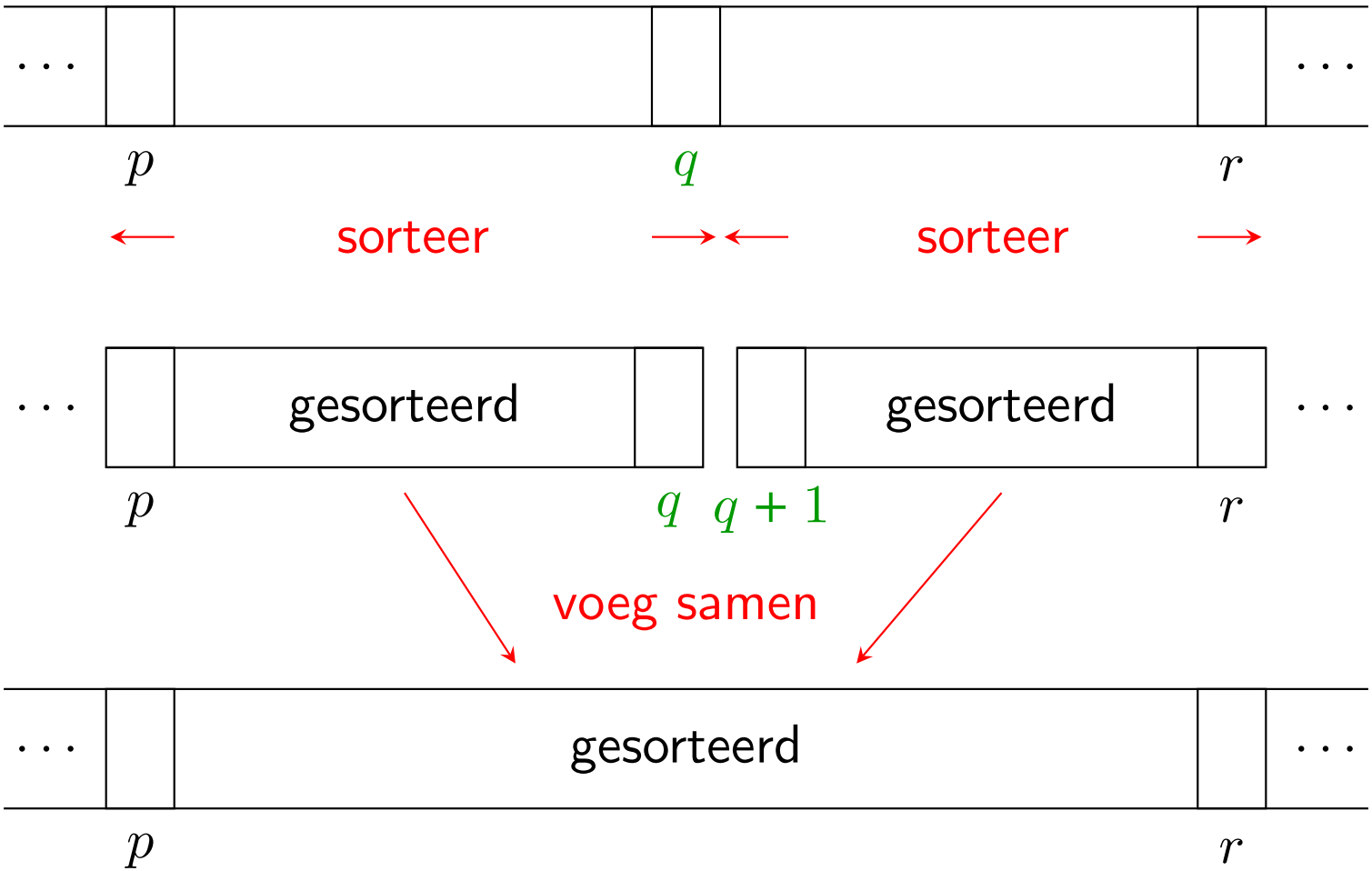
Het (recursieve) Mergesort-algoritme:

```
1 Mergesort(A, p, r)::
  | // sorteert A[p], ..., A[r]
2 | if p < r then
3 | | q := ⌊(p+r)/2⌋;
4 | | Mergesort(A, p, q);                    // verdeel
5 | | Mergesort(A, q + 1, r);                    // en
6 | | Merge(A, p, q, r);          // heers (voeg samen)
7 | fi
```

Aanroep: Mergesort$(A, 1, n)$

*John von Neumann, 1945; Wikipedia: known for [waslijst] + 93 more

# Mergesort


$p$
$q$
$r$
sorteer
sorteer
gesorteerd
gesorteerd
$p$
$q$ $q+1$
$r$
voeg samen
gesorteerd
$p$
$r$

# Samenvoegen

```
1  Merge(A, p, q, r)::
2      i := p; j := q + 1; k := p;
3      while i ≤ q and j ≤ r do
4          if A[i] < A[j] then
5              hulp[k] := A[i]; i := i + 1; k := k + 1;
6          else
7              hulp[k] := A[j]; j := j + 1; k := k + 1;
8          fi
9      od
10     if i > q then                          // eerste helft is op
11         kopieer A[j], ..., A[r] naar hulp;
12     else                                   // tweede helft is op
13         kopieer A[i], ..., A[q] naar hulp;
14     fi
15     kopieer hulp[p], ..., hulp[r] terug naar A;
```

# Merge

- Merge($A, p, q, r$) voegt de reeds gesorteerde deelrijtjes $A[p], \ldots, A[q]$ en $A[q+1], \ldots, A[r]$ samen tot een gesorteerd stuk $A[p], \ldots, A[r]$
- $hulp$ is een hulparray ter grootte $n$ (net als $A$)
- geheel analoog kan een functie Merge($A, B, C, k, m$) worden geschreven die twee gesorteerde rijen $A$ ($k$ elementen) en $B$ ($m$ elementen) samenvoegt tot de gesorteerde rij $C$ ($n = k + m$ elementen)

# Merge

- voor het bepalen van de complexiteit tellen we het aantal vergelijkingen van de vorm: $A[i] < A[j]$
- er worden altijd $2n$ verplaatsingen van array-elementen gedaan
- is het aantal arrayvergelijkingen hier wel een goede maat voor de complexiteit?

# Complexiteit Merge

Stel dat we met behulp van Merge twee gesorteerde rijtjes van respectievelijk $k$ 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 $\min(k, m)$

**Let op**: *binnen Mergesort* is het aantal vergelijkingen een goede maat voor de complexiteit. Het aantal vergelijkingen is in dat geval altijd $\Theta(n)$, net als het aantal verplaatsingen van array-elementen. In het algemene geval is dit niet zo (bijvoorbeeld $k = 1$ en $m = n - 1$).

# Worst case Mergesort

Zij $T(n)$ het aantal vergelijkingen van Mergesort in de worst case op $n$ elementen, met $n = 2^k$.

Dan geldt:

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

Oplossing: $T(n) = n \lg n - n + 1 \in \Theta(n \lg n)$

# Worst case Mergesort

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: $T(n) \in \Theta(n \lg n)$.
Je kunt zelfs bewijzen: $T(n) = n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1$.

Mergesort is in orde van grootte optimaal voor de worst case (immers: de ondergrens voor sorteren met arrayvergelijkingen is $\Omega(n \lg n)$). Er is echter extra geheugenruimte nodig ter grootte $\Theta(n)$.

# Best case Mergeworst

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

# Optimaliteit Merge

## Stelling

1. *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.
   Voor $m = \frac{n}{2}$ ($n$ even) is dit dus ten minste $n - 1$.
2. voor het samenvoegen van twee rijtjes van lengte $m - 1$ respectievelijk $m$ is dat ten minste $2m - 2$.
   Voor $m = \lceil \frac{n}{2} \rceil$ ($n$ oneven) is dit ten minste $n - 1$.

**Gevolg**

Binnen de klasse van samenvoegalgoritmen gebaseerd op arrayvergelijkingen is het beschreven Merge-algoritme optimaal, althans voor twee ongeveer even lange rijtjes.

# Bewijs

We geven een klasse van invoerrijtjes waarop *elk* samenvoegalgoritme (gebaseerd op arrayvergelijkingen) *ten minste* $2m - 1$ vergelijkingen moet doen. Dat bewijst dan de stelling.

Kies stijgende rijtjes $A = a_1, a_2, \ldots, a_m$ en $B = b_1, b_2, \ldots, b_m$ zó dat alle $a_i$ en $b_j$ verschillend zijn en $a_i < b_j \Leftrightarrow i < j$:

$$b_1 < a_1 < b_2 < \ldots < a_{i-1} < b_i < a_i < b_{i+1} < \ldots < b_m < a_m$$

Dan *moet* elk samenvoegalgoritme $a_i$ vergelijken met $b_i$ $(i = 1, 2, \ldots, m)$ én $a_i$ met $b_{i+1}$ $(i = 1, 2, \ldots, m - 1)$.

Het bewijs van deze bewering gaat uit het ongerijmde.

# Sorteren (ondergrens)

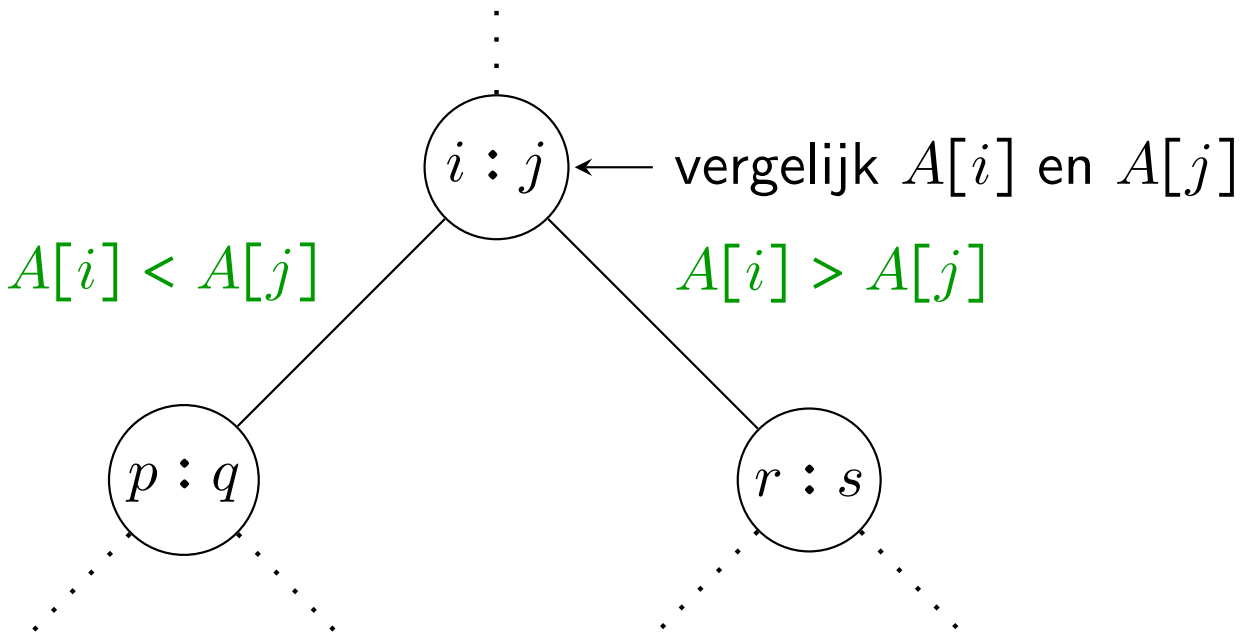
We bekijken sorteeralgoritmen gebaseerd op het doen van vergelijkingen van de vorm $A[i] < A[j]$.

Aannames (z.v.v.a.):

- $A$ bevat $n$ verschillende waarden. (We bepalen immers een ondergrens voor de worst case.)
- het sorteeralgoritme stopt zodra de sortering (onderlinge ordening) is gevonden.

Zo'n algoritme komt overeen (voor elke $n$) met een beslissingsboom die de series vergelijkingen representeert die het algoritme uitvoert voor elke mogelijke invoer (ter grootte $n$). Elk pad van de wortel tot een blad komt overeen met een executie van het algoritme.

# Beslissingsboom

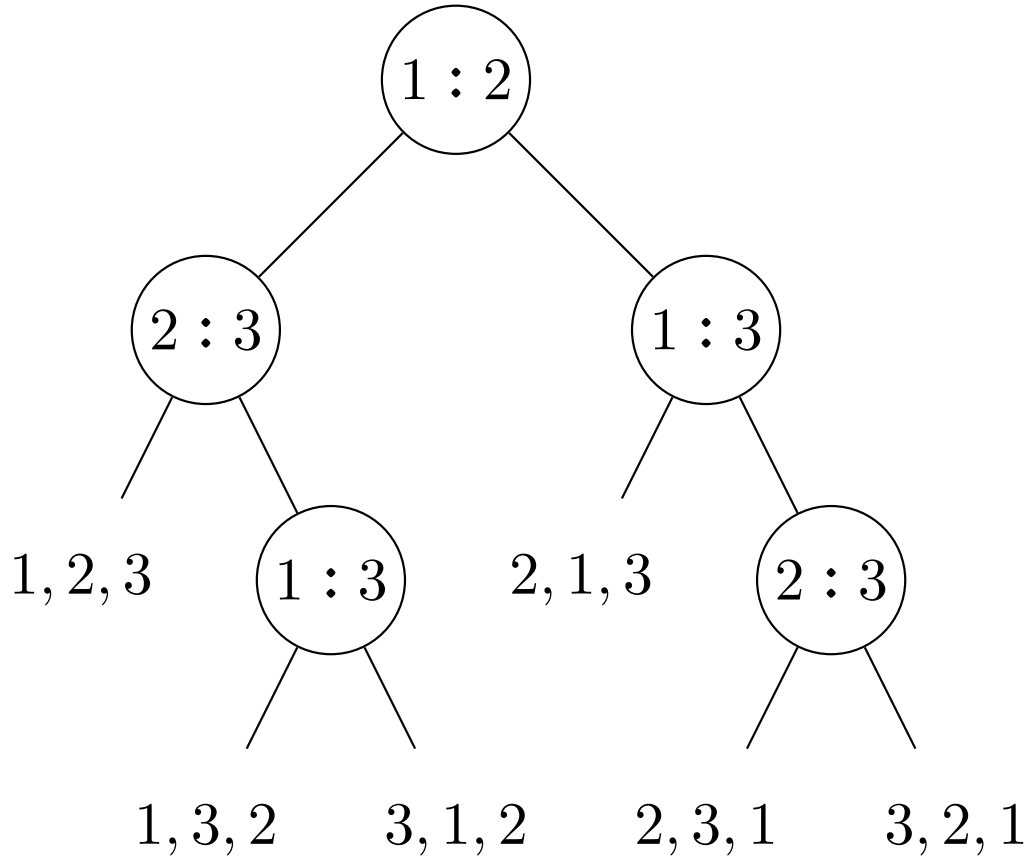



vergelijk vervolgens
$A[p]$ en $A[q]$

vergelijk vervolgens
$A[r]$ met $A[s]$

Beslissingsboom voor algoritmen gebaseerd op **arrayvergelijkingen**

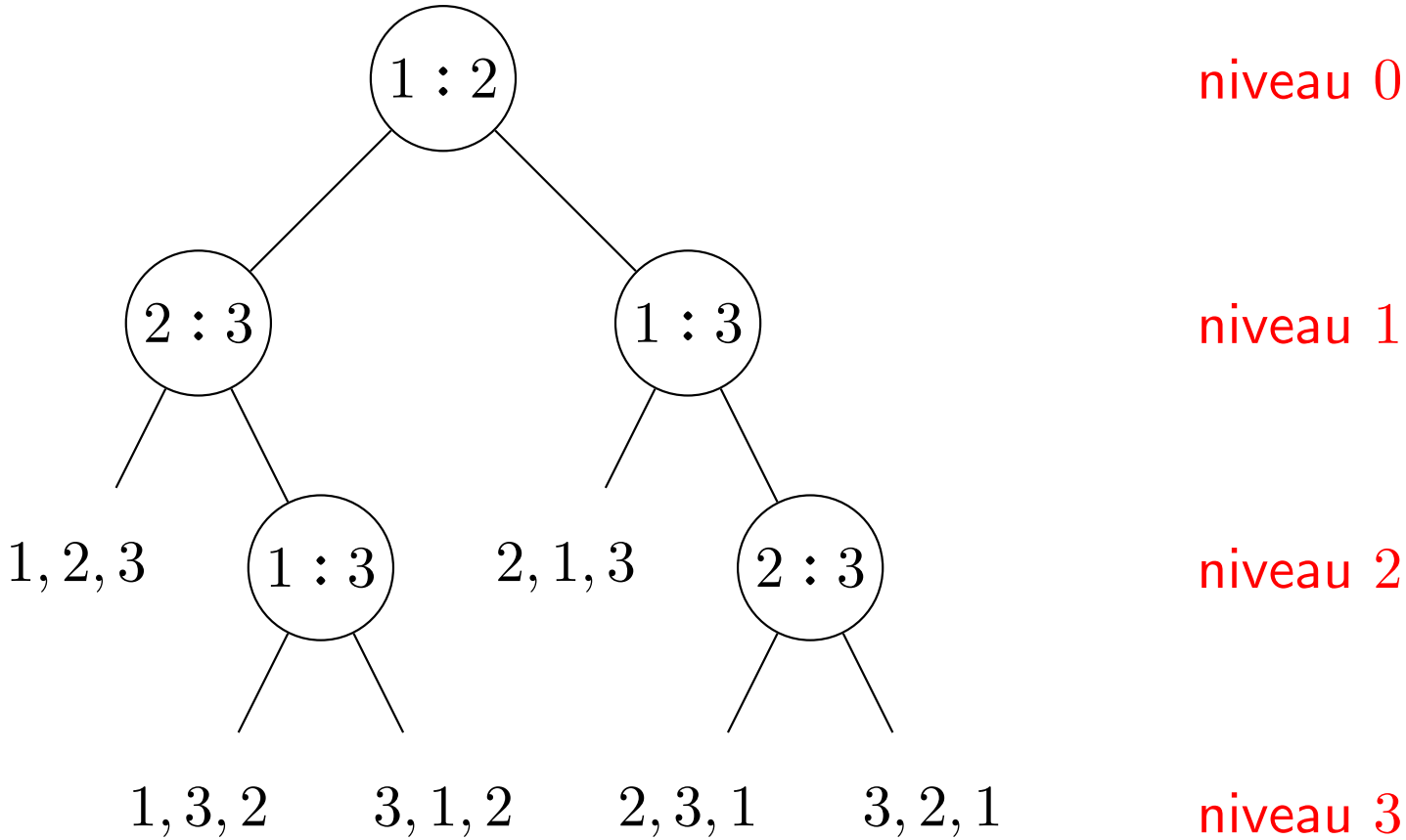
# Insertion sort




Beslissingsboom voor Insertion sort met $n = 3$

$2, 3, 1$ betekent: $A[2] < A[3] < A[1]$ (andere bladeren analoog)

# Hoogte $\Longleftrightarrow$ worst case




In een beslissingsboom voor algoritmen gebaseerd op arrayvergelijkingen geeft de hoogte van de boom precies het aantal vergelijkingen aan in de worst case.

# Bladeren*: maximaal

1. 
   - alleen de onderlinge volgorde van de array-elementen wordt onderscheiden; niet de waarde
   - het rijtje $6, 11, 15, 8, 3$ wordt bijvoorbeeld precies zo behandeld door het sorteeralgoritme als het rijtje $2, 4, 5, 3, 1$
   - ze volgen dan ook precies hetzelfde pad in de beslissingsboom
   - er zijn in essentie $n!$ mogelijke te onderscheiden invoeren, die elk één pad volgen in de boom $\Rightarrow$ er zijn maximaal $n!$ bladeren

*algemeen

# Bladeren*: minimaal

2. 
   - sorteren komt neer op het vinden van de oplopende ordening
   - er zijn dus $n!$ verschillende eindantwoorden (ordeningen) mogelijk
   - een sorteeralgoritme moet die allemaal kunnen vinden
   - de bijbehorende beslissingsboom moet dus minstens $n!$ bladeren hebben
3. conclusie: een beslissingsboom corresponderend met een sorteeralgoritme gebaseerd op arrayvergelijkingen, heeft precies $n!$ bladeren (waar $n$ het aantal elementen is).

*voor sorteren

# Ondergrens sorteren

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

**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 $n!$ mogelijke volgordes als invoerrijtje even waarschijnlijk zijn.
De stelling volgt direct uit de resultaten op de volgende sheet.

---

[*] om dit te bewijzen heb je alleen nodig dat het aantal bladeren $\geq n!$ is

[†] hiervoor gebruik je dat het aantal bladeren precies gelijk is aan $n!$

# Intermezzo

Gegeven een binaire boom $B$ met $b$ bladeren.

Definitie De externe padlengte $E$ van $B$ is de som van de lengtes van alle paden van de wortel naar een blad:

$$E = \sum_{\text{bladeren}} (\text{lengte pad wortel} \rightarrow \text{blad})$$

Lemma Zij $E$ de externe padlengte van $B$. Dan geldt:

$$E \geq b \times (\lceil \lg b \rceil - 1)$$

Gevolg De gemiddelde lengte van een pad van de wortel naar een blad is $\frac{E}{b} \geq \lceil \lg b \rceil - 1$.

# Verdeel en heers

Mergesort en Quicksort zijn sorteermethoden die allebei zijn gebaseerd op de verdeel-en-heers-strategie:

```
1 Sorteer(rij)::
2 |   if de rij heeft meer dan één element then
3 |   |   Verdeel de rij in twee stukken: linkerrij en rechterrij;
4 |   |   Sorteer(linkerrij);
5 |   |   Sorteer(rechterrij);
6 |   |   Combineer linkerrij en rechterrij;
7 |   fi
```

Mergesort stopt het meeste werk in de combineerstap, Quicksort in de verdeelstap. Beide sorteermethoden zijn gebaseerd op arrayvergelijkingen, maar doen niet uitsluitend buurverwisselingen, zoals Insertion sort en Bubblesort.

# Quicksort*

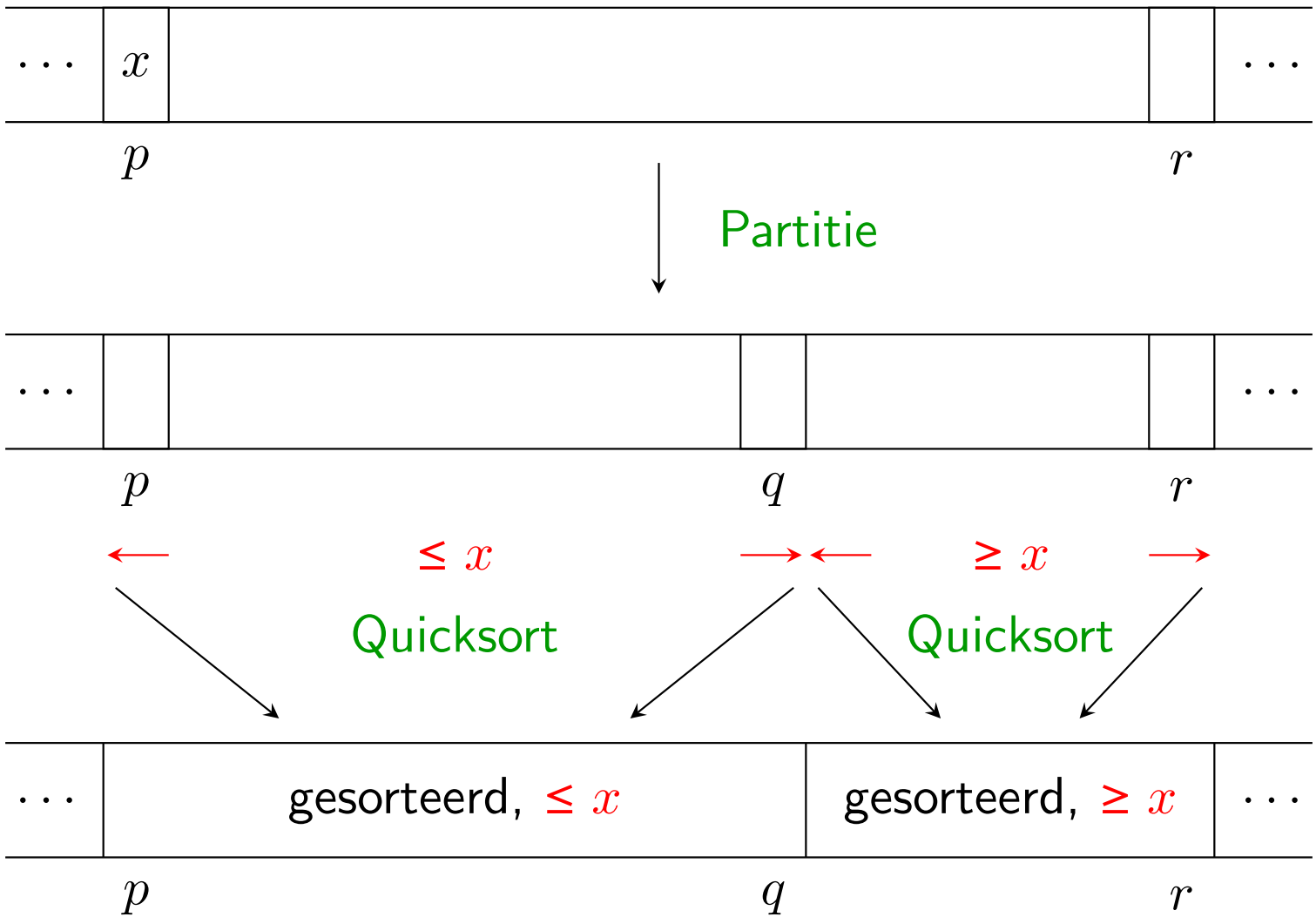
```
1 Quicksort(A, p, r)::
  |  // sorteert A[p], ..., A[r]
2 |  if p < r then
3 |  |  q := Partitie(A, p, r);
4 |  |  Quicksort(A, p, q);
5 |  |  Quicksort(A, q + 1, r);
6 |  fi
```

Aanroep: Quicksort($A$, $1$, $n$)

- recursief
- alleen interne verwisselingen
- geen extra geheugenruimte
- in de praktijk een van de snelste

*Tony Hoare, 1962; Hoare-logica, CSP (en veel meer); Turing Award 1980

# Quicksort


$\cdots$
$x$
$\cdots$
$p$
$r$
Partitie
$\cdots$
$\cdots$
$p$
$q$
$r$
$\leq x$
$\geq x$
Quicksort
Quicksort
$\cdots$
gesorteerd, $\leq x$
gesorteerd, $\geq x$
$\cdots$
$p$
$q$
$r$

# Partitie

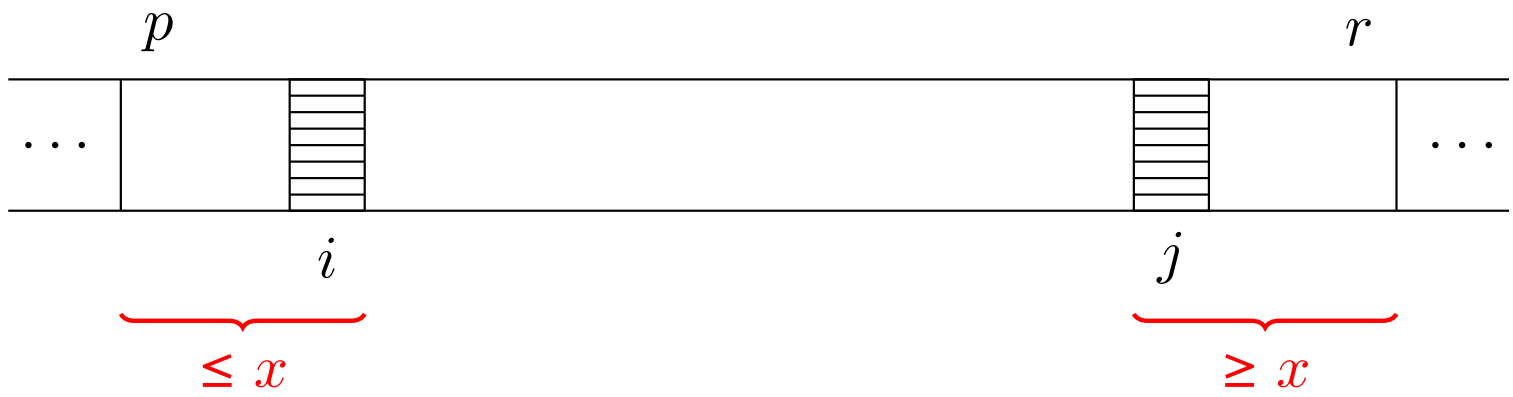
```
1  Partitie(A, p, r)::
       // (*) hier komt nog wat
2      x := A[p]; i := p − 1; j := r + 1;
3      while i < j do
4          j := j − 1;                    // loop met j naar links
5          while A[j] > x do
6              j := j − 1;
7          od                  // tot je een waarde A[j] ≤ x vindt
8          i := i + 1;                    // loop met i naar rechts
9          while A[i] < x do
10             i := i + 1;
11         od                  // tot je een waarde A[i] ≥ x vindt
12         if i < j then
13             wissel(A[i], A[j]);
14         fi  // A[i] en A[j] staan nu in het goede stuk
15     od
16     return j;                                  // dit wordt dus q
```

# Opmerkingen bij Partitie

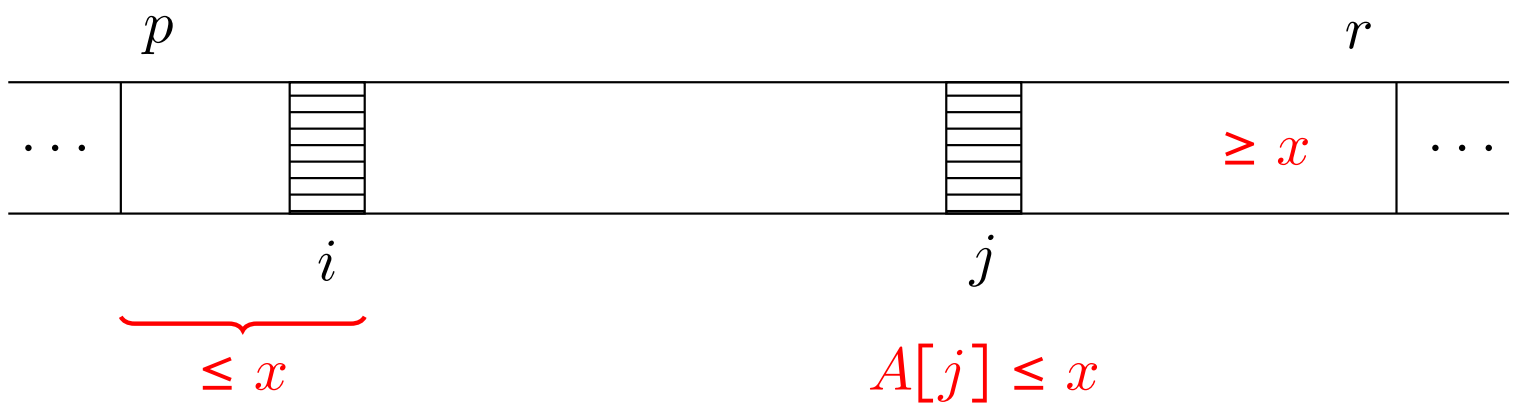
1. in ronde 1 stopt $i$ op positie $p$ en $j$ op een positie $\geq p$
2. de *basisoperatie* is het vergelijken van array-elementen: $A[j] > x$ en $A[i] < x$
3. partitie stopt uiteindelijk met $i = j$ of $i = j + 1$
4. na afloop is altijd $p \leq j \leq r - 1$, dus $p \leq q \leq r - 1$. Quicksort wordt dus echt recursief aangeroepen op kleinere rijtjes.
5. elk array-element wordt precies één keer vergeleken met $x$, behalve $A[q]$ (twee keer) en eventueel $A[q + 1]$ (soms twee)
6. Partitie doet altijd $\Theta(m)$ vergelijken, namelijk $m + 1$ of $m + 2$, met $m$ het aantal elementen van het (deel)array $A[p], \ldots, A[r]$
7. er worden elementen verwisseld die ver uit elkaar kunnen liggen. Per vergelijking worden dus wellicht $> 1$ inversies opgeheven.

# Werking Partitie

1. Na een volledige ronde:

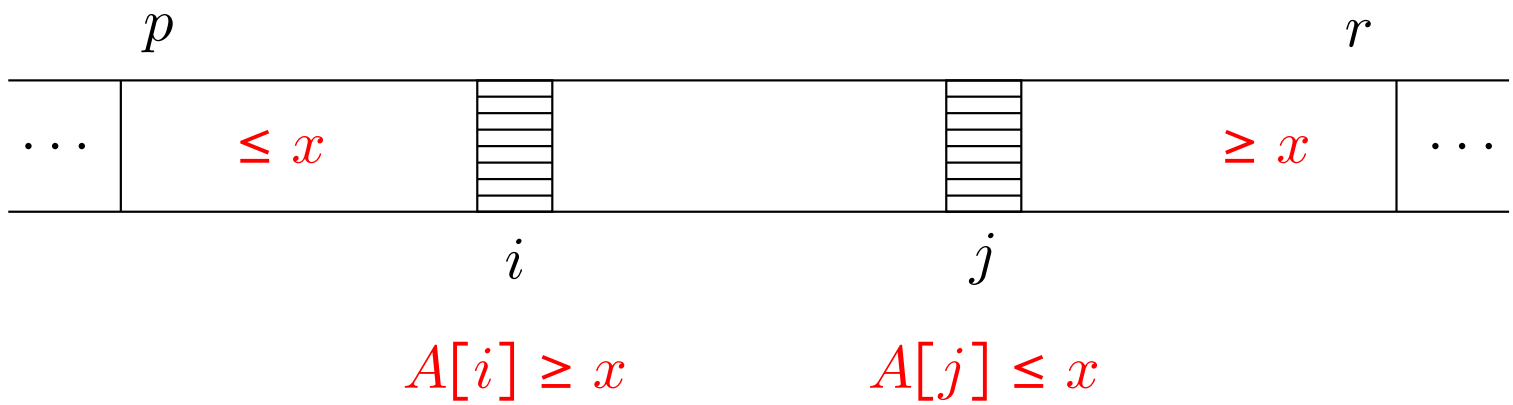



2. Na de $j$-loop:

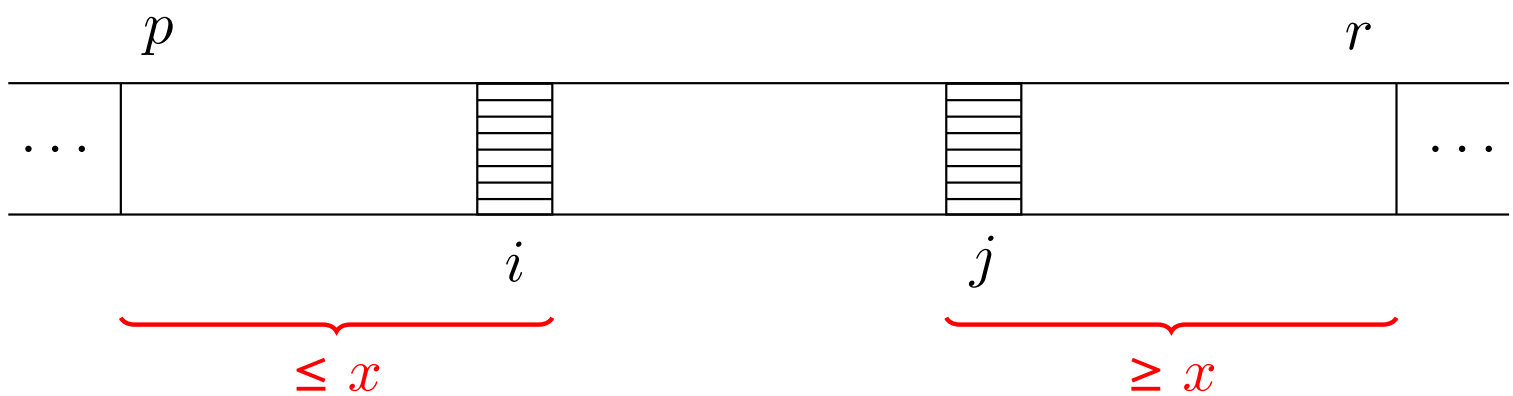


# Werking Partitie

3. Na de $i$-loop, vóór de verwisseling:




4. Na de verwisseling:



# Vragen bij Partitie

- wat gebeurt er bij gelijke array-elementen? Bijvoorbeeld: $5, 3, 5, 5, 6, 5, 7$ of $5, 6, 3, 4, 3, 8, 3$ of een array bestaande uit allemaal dezelfde elementen.
- wat gebeurt er met een rijtje met allemaal verschillende elementen? Bijvoorbeeld: $3, 2, 7, 4, 6, 8, 5, 1$, of een reeds gesorteerd rijtje of een omgekeerd gesorteerd rijtje.
- zie ook opgaven 40 en 41 uit het dictaat.

# Bad case

Bad case voor Quicksort:

Op een reeds gesorteerd rijtje (!) bestaande uit verschillende waardes, bijvoorbeeld $1, 2, \ldots, n$, doet Quicksort

$$\sum_{k=3}^{n+1} k = \frac{1}{2} n(n+3) - 2 \in \Theta(n^2)$$

vergelijkingen. Partitie deelt het array telkens zeer onevenwichtig in tweeën.

# Good case

Good case voor Quicksort (met $n = 2^k$):

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

Het aantal vergelijkingen $B(n)$ is dan $\Theta(n \lg n)$. Dit komt voor als Partitie het array bij elke aanroep precies in tweeën deelt. Dit is zelfs het *beste* geval voor Quicksort.

# Worst case

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

$$W(n) = \begin{cases} 0 & n = 1 \\ \max_{1 \leq q \leq n-1}(W(q) + W(n-q)) + \Theta(n) & n > 1 \end{cases}$$

Er geldt dus dat $W(n) \leq dn^2$ voor zekere $d > 0$, en vanaf zekere $n_0$. Dus: $W(n) \in O(n^2)$.

Samen met de bad case hebben we dus:

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

*i.p.v. $\Theta(n)$ kun je in de recurrente betrekking ook $\leq n + 2$ zetten

# Spil (pivot)

De keuze van de spil ($x$ dus) heeft grote invloed op de complexiteit van Quicksort. Standaard het eerste array-element ($A[p]$) kiezen als spil is een slechte keuze.

Voeg ter verbetering op plek (*) in Partitie toe:

**Kies een slim array-element en wissel dat met $A[p]$.**

*Slim* kan zijn: kies een willekeurig* array-element. De worst case blijft dan uiteraard $\Theta(n^2)$, maar onder de aanname dat alle $A[i]$ verschillend zijn hebben we nu:

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

*Randomised Quicksort

# Randomised Quicksort

```
1  Partitie(A, p, r)::
2      wissel willekeurig array-element met A[p];
3      x := A[p]; i := p − 1; j := r + 1;
4      while i < j do
5          j := j − 1;                    // loop met j naar links
6          while A[j] > x do
7              j := j − 1;
8          od                  // tot je een waarde A[j] ≤ x vindt
9          i := i + 1;                    // loop met i naar rechts
10         while A[i] < x do
11             i := i + 1;
12         od                  // tot je een waarde A[i] ≥ x vindt
13         if i < j then
14             wissel(A[i], A[j]);
15         fi  // A[i] en A[j] staan nu in het goede stuk
16     od
17     return j;                                  // dit wordt dus q
```

# Sorteermethoden

| Algoritme | worst case | average case | extra ruimte |
|---|---|---|---|
| Insertion sort | $\frac{1}{2}n^2$ | $\Theta(n^2)$ | "in place" (d.w.z. geen) |
| Quicksort | $\frac{1}{2}n^2$ | $\Theta(n \lg n)$ | proportioneel aan $\lg n$ |
| Mergesort | $n \lg n$ | $\Theta(n \lg n)$ | proportioneel aan $n$ |
| Heapsort | $2n \lg n$ | $\Theta(n \lg n)$ | in place |

Heapsort: zie dictaat, opgave 38

# Vergelijken

Vergelijking van verschillende sorteeralgoritmen
(average case; tijd in seconden)

| $n$ | Insertion $O(n^2)$ | Shellsort $O(n^{7/6})$ | Heapsort $O(n \lg n)$ | Quicksort $O(n \lg n)$ | Quicksort* $O(n \lg n)$ |
|---|---|---|---|---|---|
| 10 | 0,00044 | 0,00041 | 0,00057 | 0,00052 | 0,00046 |
| 100 | 0,00675 | 0,00171 | 0,00420 | 0,00284 | 0,00244 |
| 1000 | 0,59564 | 0,02927 | 0,05565 | 0,03153 | 0,02587 |
| 10 000 | 58,864 | 0,42998 | 0,71650 | 0,36765 | 0,31532 |
| 100 000 | — | 5,7298 | 8,8591 | 4,2298 | 3,5882 |
| 1 000 000 | — | 71,164 | 104,68 | 47,065 | 41,282 |

*geoptimaliseerd

# Om te onthouden van vandaag

- Recurrente betrekkingen: oefenen (ook de bewijzen!)
- Mergesort
  - verdeel-en-heers, sorteert recursief twee helften en voegt ze weer samen
  - best case, worst case, average case $\in \Theta(n \lg n)$
- Quicksort
  - verdeel-en-heers, verdeelt in twee delen en sorteert deze recursief
  - best case, average case $\in \Theta(n \lg n)$, worst case $\in \Theta(n^2)$
- Worst case en average case sorteren (met arrayvergelijkingen) allebei $\in \Omega(n \lg n)$

# (Werk)college

**Volgende college**: dinsdag 14 maart, 9u00–10u45, zaal C1 (Gorlaeus)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304, 306–308 en 303 (Snellius)
Opgaven uit het dictaat: 15, 16, 31, 23, 24, 18

# Huiswerk

**Huiswerk 2**

Verschijnt vanmiddag op de website.

Inleveren via Brightspace, uiterlijk maandag**ochtend** 27 maart, **11u59**.

# Complexiteit 2023 — college 6

14 maart 2023

---

Shellsort

Optimaal sorteren

Sorteren in $O(n)$

# Vorige keer

- Mergesort
  - verdeel-en-heers, sorteert recursief twee helften en voegt ze weer samen
  - best case, worst case, average case $\in \Theta(n \lg n)$
- Quicksort
  - verdeel-en-heers, verdeelt in twee delen en sorteert deze recursief
  - best case, average case $\in \Theta(n \lg n)$, worst case $\in \Theta(n^2)$
- Worst case en average case sorteren (met arrayvergelijkingen) allebei $\in \Omega(n \lg n)$

# Vandaag

- Shellsort
- Optimaal sorteren
- Sorteren in $O(n)$

# Inversies

Voor sorteeralgoritmen gebaseerd op arrayvergelijkingen, waarbij per arrayvergelijking hooguit één inversie wordt opgeheven*, geldt:

- # arrayvergelijkingen ≥ # inversies invoerarray
- # arrayvergelijkingen in de worst case ≥ $\frac{1}{2}n(n-1)$

Als je een beter sorteeralgoritme wilt (gebaseerd op arrayvergelijkingen), moet je dus elementen verwisselen die verder van elkaar liggen, zoals gebeurt bij Mergesort, Quicksort en Shellsort (en veel meer).

*zoals bij algoritmen die gebruikmaken van buurverwisselingen, zoals Insertion sort en Bubblesort

# Inversies opheffen

Stel dat $A[i]$ en $A[i+k]$ $(k > 0)$ verkeerd om staan en dat we die verwisselen. Hoeveel inversies worden dan ten minste respectievelijk ten hoogste opgeheven?

Situatie:


met $A[i] > A[i+k]$. Verwissel nu $A[i]$ en $A[i+k]$.

# Shellsort*

Shellsort was een van de eerste sorteeralgoritmen met een beter dan kwadratische worst case-complexiteit.

Shellsort sorteert in elke ronde deelrijtjes. In het begin veel korte rijtjes, waarbij de elementen uit een rijtje ver van elkaar liggen. Later weinig lange rijtjes, waarbij de elementen uit een rijtje dicht bij elkaar liggen. De rij wordt zo als het ware voorgesorteerd. In de laatste ronde wordt de rij dan als geheel gesorteerd. Shellsort sorteert met behulp van vergelijk-verwissel-operaties (compare-exchange, indien nodig).

*Donald Shell, 1959

# $k$-gesorteerd

**Definitie**

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

**Voorbeeld**: het rijtje

$$5, 8, 24, 13, 7, 18, 31, 19, 44, 63, 82, 29$$

is 4-gesorteerd (en overigens ook 6-gesorteerd).

Merk op: 1-gesorteerd is gesorteerd.

# De methode

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 $i = t, \ldots, 1$. Aangezien $h_1 = 1$ sorteert Shellsort correct.

Merk op: bij het $k$-sorteren worden $k$ deelrijtjes van elk maximaal $\lceil \frac{n}{k} \rceil$ elementen gesorteerd.

# Voorbeeld

$h_3 = 6, h_2 = 3, h_1 = 1, n = 13$

$$81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15$$

$\downarrow$ 6-sorteren

$$15, 94, 11, 58, 12, 35, 17, 95, 28, 96, 41, 75, 81$$

$\downarrow$ 3-sorteren

$$15, 12, 11, 17, 41, 28, 58, 94, 35, 81, 95, 75, 96$$

$\downarrow$ 1-sorteren

$$11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96$$

# Voorbeeld

Insertion sort op het voorbeeldrijtje: 52 vergelijkingen.

Shellsort met Insertion sort: 44 vergelijkingen:

$81, 94, 11, 96, 12, 35, 17, 95, 28, 58, 41, 75, 15$

\# inversies: 41 $\downarrow$ 6-sorteren: 8 vergelijkingen

$15, 94, 11, 58, 12, 35, 17, 95, 28, 96, 41, 75, 81$

\# inversies: 25 $\downarrow$ 3-sorteren: 15 vergelijkingen

$15, 12, 11, 17, 41, 28, 58, 94, 35, 81, 95, 75, 96$

\# inversies: 11 $\downarrow$ 1-sorteren: 21 vergelijkingen

$11, 12, 15, 17, 28, 35, 41, 58, 75, 81, 94, 95, 96$

# Algoritme

```
1  h := ⌊n/2⌋;                                          // dus h_t = ⌊n/2⌋
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
               // juist invoegen in deelrijtje
6              if temp < A[j − j] then
7                  A[j] := A[j − h];                        // schuif
8                  j := j − h;
9              else
10                 break;
11         A[j] := temp;                                  // zet neer
       // de rij is nu h-gesorteerd
12     h := ⌊h/2⌋;          // oorspronkelijke keuze van Shell
```

Merk op: regels 4 t/m 11 is Insertion sort op deelrijtjes.

# Eigenschap

Een zeer belangrijke eigenschap van Shellsort is de volgende (zonder bewijs):

**Stelling**
Als een $\ell$-gesorteerd array wordt $h$-gesorteerd met behulp van vergelijk-verwissel-operaties, dan blijft het $\ell$-gesorteerd.

# Voorbeeld

Voorbeeld met $n = 12$ en incrementrijtje $6, 4, 3, 2, 1$:

$7, 19, 24, 13, 31, 8, 82, 18, 44, 63, 5, 29$

$\downarrow$ 6-sorteren

$7, 18, 24, 13, 5, 8, 82, 19, 44, 63, 31, 29$ 6-gesorteerd

$\downarrow$ 4-sorteren

$5, 8, 24, 13, 7, 18, 31, 19, 44, 63, 82, 29$ $\{4, 6\}$-gesorteerd

$\downarrow$ 3-sorteren

$5, 7, 18, 13, 8, 24, 31, 19, 29, 63, 82, 44$ $\{3, 4, 6\}$-gesorteerd

$\downarrow$ 2-sorteren

$5, 7, 8, 13, 18, 19, 29, 24, 31, 44, 82, 63$ $\{2, 3, 4, 6\}$-gesorteerd

$\downarrow$ 1-sorteren

$5, 7, 8, 13, 18, 19, 24, 29, 31, 44, 63, 82$ $\{1, 2, 3, 4, 6\}$-gesorteerd

# Complexiteit

De complexiteit van Shellsort (d.w.z. het aantal arrayvergelijkingen) hangt in hoge mate af van de gekozen stapgroottes. De analyse is in het algemeen extreem moeilijk en nog zeer incompleet.

1. Stapgroottes: $h_t = \lfloor \frac{n}{2} \rfloor$, $h_i = \lfloor \frac{h_{i+1}}{2} \rfloor$ voor $i = t-1, \ldots, 1$. Dan $t = \lfloor \lg n \rfloor$ rondes. Voor het gemak nemen we $n = 2^k$. Dan: $t = \lg n = k$ en stapgroottes zijn $2^{k-1}, 2^{k-2}, \ldots, 4, 2, 1$.

   Stelling A
   Het aantal arrayvergelijkingen dat Shellsort doet met deze serie stapgroottes is in de worst case $\Omega(n^2)$.

   Bij het bewijs is het voldoende om een bad case aan te geven waarvoor het aantal vergelijkingen $\Omega(n^2)$ is. Zie college.

# Complexiteit

Stelling B
Het aantal arrayvergelijkingen dat Shellsort doet met deze serie stapgroottes is in de worst case $O(n^2)$. Bewijs: zie college.

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

2. Stapgroottes (Hibbard): $2^k - 1, 2^{k-1} - 1, \ldots, 7, 3, 1$
   ($k = \lfloor \lg n \rfloor$)

   Stelling. In de worst case doet Shellsort met Hibbards serie stapgroottes $O(n\sqrt{n})$ vergelijkingen.

3. Het kan nog beter!

# Bewijs Hibbard — Schets

Stelling. In de worst case doet Shellsort met Hibbards reeks stapgroottes $O(n\sqrt{n})$ arrayvergelijkingen.

**Bewijsschets**. In een ronde met *grote stap* $h$ zijn we snel klaar met Insertion sort: $O(\frac{n^2}{h})$; in een ronde met *kleine stap* gebruiken we dat de rij al is voorgesorteerd en daarmee weinig vergelijkingen doet: $O(nh)$.

Als $n = 2^k$ dan geldt $\sqrt{n} = 2^{k/2}$, halverwege de stapgroottes.

# Bewijs Hibbard — Frobenius

**Feit**. Als $a$ en $b$ geen gemene delers hebben, geldt: alle gehele getallen vanaf $m = (a-1)(b-1)$ kunnen worden uitgedrukt als combinatie $k \times a + \ell \times b$.*

$a = 3, b = 7$: $\quad 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, \ldots$

**Gevolg** voor Shellsort.
Als het array $A$ zowel is $a$-gesorteerd als $b$-gesorteerd, dan geldt dat $A[j] \leq A[i]$ voor alle $j \leq i - m$. Kortom: als we $A[i]$ toevoegen met Insertion sort dan stopt dit proces vóór $A[i-m]$, omdat eerdere array-elementen kleiner zijn.

Dat zijn maximaal $\frac{m}{h}$ vergelijkingen (bij stapgrootte $h$).

*Frobenius coin problem

# Bewijs Hibbard

Laat $n = 2^k$. Hibbard stappen $2^k - 1, 2^{k-1} - 1, \ldots, 7, 3, 1$.

**Kleine stappen**. Als de huidige stap $h$ is, waren de vorige twee $2h + 1$ en $4h + 3$, met Frobeniuswaarde $m = 2h(4h + 2)$. Dan zijn er maximaal $n \times \frac{2h(4h+2)}{h} \approx 8nh$ vergelijkingen nodig bij een ronde met stap $h$.

$$\sum_{i=1}^{k/2} 8n(2^i - 1) \leq 8n \sum_{i=1}^{k/2} 2^i \leq 16n \times 2^{t/2} \in O(n\sqrt{n})$$

**Grote stappen**. Bij stap $h$ maximaal $\frac{n^2}{h}$ vergelijkingen bij Insertion sort.

$$\sum_{i=k/2}^{k} \frac{n^2}{2^i - 1} \leq 2n^2 \sum_{i=t/2}^{t} \frac{1}{2^i} \leq 4n^2 \times \frac{1}{2^{t/2}} \in O(n\sqrt{n})$$

**Samen**. $O(n\sqrt{n})$

# Relatief priem

Voorbeeld. Na 8-sorteren heeft 3-sorteren i.h.a. veel meer effect dan 4-sorteren.

8-gesorteerd, 36 inversies

$$5, 2, 11, 7, 15, 3, 16, 6, 12, 9, 14, 8, 19, 13, 20, 10$$

$\downarrow$ 4-sorteren

31 inversies

$$5, 2, 11, 6, 12, 3, 14, 7, 15, 9, 16, 8, 19, 13, 20, 10$$

8-gesorteerd, 36 inversies

$$5, 2, 11, 7, 15, 3, 16, 6, 12, 9, 14, 8, 19, 13, 20, 10$$

$\downarrow$ 3-sorteren

7 inversies

$$5, 2, 3, 6, 7, 8, 9, 12, 11, 10, 13, 15, 14, 16, 20, 19$$

# Complexiteit

## Betere incrementseries

- Twee (!) doorgangen: $h_2 \approx 1{,}72 \times \sqrt[3]{n}$, $h_1 = 1$
  Gemiddeld $O(n^{5/3})$
- Heel veel doorgangen: stapgroottes van de vorm $2^i \cdot 3^j$:
  $1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, 48, 54, 72, 81, \ldots$
  Worst case $O(n(\lg n)^2)$
- Stapgroottes van de vorm $4^{j+1} + 3 \cdot 2^j + 1$:
  $1, 8, 23, 77, 281, 1073, 4193, 16577, \ldots$
  Worst case $O(n^{4/3})$
- . . . (optimale serie stapgroottes is [nog] onbepaald)

# Oude tentamenopgave

## Opgave

Neem aan dat $n$ een macht van $7$ is, dus $n = 7^k$ met $k \geq 0$ geheel. Bewijs nu dat het aantal vergelijkingen dat Shellsort met stapgroottes $n/7, n/49, n/343, \ldots, 49, 7, 1$ in de worst case doet $\Omega(n^2)$ is.

Het bewijs gaat analoog aan het bewijs voor Shellsort met Shells rijtje stapgroottes.

# Ondergrens

Voor sorteeralgoritmen gebaseerd op arrayvergelijkingen is het aantal arrayvergelijkingen* in de worst case ten minste $\lceil \lg n! \rceil$.

**Vraag**: hoe dicht kan men in de buurt van deze ondergrens komen?

| $n$ | 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 |
| $M(n)$ | 1 | 3 | 5 | 8 | 11 | 14 | 17 | 21 | 25 | 29 | 33 | ... | 74 |
| $B(n)$ | 1 | 3 | 5 | 8 | 11 | 14 | 17 | 21 | 25 | 29 | 33 | ... | 74 |

*voor het vinden van de juiste ordening

# Binary Insertion sort

$M(n)$ is het worst case-aantal vergelijkingen van Mergesort voor een rij met $n$ elementen.
$B(n)$ is het worst case-aantal vergelijkingen van Binary Insertion sort voor een rij met $n$ elementen.
Merk op dat $\lceil \lg 1! \rceil = M(1) = B(1) = 0$.

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. (Zie ook opgave 34.)

Om $A[i]$ op de juiste plek te kunnen invoegen in het deelarray $A[1], \ldots, A[i-1]$ zijn nu in het slechtste geval $\lceil \lg i \rceil$ vergelijkingen nodig*. Dus:

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

*voor het vinden van de juiste positie

# Demuth

Er geldt zelfs:

$$\sum_{i=2}^{n} \lceil \lg i \rceil = n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + 1,$$

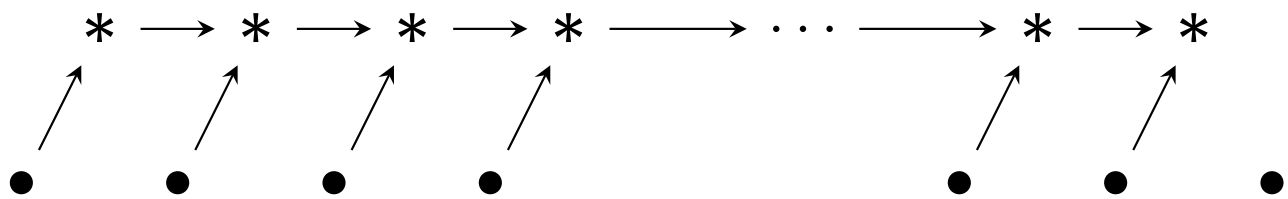
en dat is ook precies het aantal vergelijkingen dat Mergesort doet. Dus $B(n) = M(n)$.

**Vraag**: kan het nog beter?
**Antwoord**: ja!
**Voorbeeld**: 5 elementen kunnen worden gesorteerd met 7 vergelijkingen (Howard Demuth, 1956) (opgave 25b)

De methode van Demuth kan worden gegeneraliseerd tot het algoritme Merge Insertion sort (Lester Ford en Selmer Johnson, 1959).

# Merge Insertion sort

1. vergelijk de $n$ elementen twee aan twee
2. sorteer de $\lfloor \frac{n}{2} \rfloor$ winnaars $*$ (dus de grootsten) recursief. Dit levert iets op als:



Hierin betekent $*_1 \to *_2$ dat $*_1 < *_2$ en $\bullet \to *$ dat $\bullet < *$.

3. voeg nu de $\lfloor \frac{n}{2} \rfloor$ verliezers (en de losse waarde als $n$ oneven is) in op de juiste plek **in een ingenieuze volgorde**.

# Aantal vergelijkingen

Het aantal vergelijkingen dat Merge Insertion sort doet is voor $n \leq 15$ en $n = 20, 21, 22$ gelijk aan de theoretische ondergrens van $\lceil \lg n! \rceil$.

Merge Insertion sort sorteert bijvoorbeeld:

- 10 elementen in 22 vergelijkingen (optimaal)
- 12 elementen in 30 vergelijkingen (optimaal)
- 21 elementen in 66 vergelijkingen (optimaal)

# Optimaal?

Voor kleine $n$ (zoals $n = 12$) is het mogelijk om, via exhaustive search, de echte ondergrens *empirisch* te bepalen: probeer alle mogelijke combinaties van vergelijken op alle permutaties van $n$ getallen en concludeer dat $\ell$ vergelijkingen niet voldoende zijn. De ondergrens is dan dus $\geq \ell + 1$.
Inmiddels is zo ook aangetoond dat Merge Insertion sort optimaal is voor $n = 13, 14, 15, 22$ (Peczarski, 2004/2006).

**Vraag**: is Merge Insertion sort optimaal?

**Antwoord**: nee, bijvoorbeeld voor $n = 47$ is een algoritme bekend (Schulte Mönting, 1981) dat één vergelijking minder nodig heeft (200) dan Merge Insertion sort (201).

# Voorkennis

Als je helemaal niets weet over het array $A$ is de enige manier om $A$ correct te ordenen het stellen van vragen van de vorm $A[i] < A[j]$. In dat geval geldt de theoretische ondergrens $\lceil \lg n! \rceil \in \Theta(n \lg n)$. Soms kun je echter sneller sorteren dan $\Theta(n \lg n)$ (namelijk in $O(n)$ tijd) door handig gebruik te maken van voorkennis over de invoer.

Dit is niet in strijd met de theoretische ondergrens: die geldt immers voor sorteeralgoritmen die *alle* mogelijke invoeren kunnen sorteren (en zijn gebaseerd op arrayvergelijkingen).

We bekijken ter afsluiting van het gedeelte over sorteren twee methoden die efficiënt sorteren als de invoer aan zekere voorwaarden voldoet: Counting sort en Radix sort.

# Counting sort

**Invoer**: een array $A = A[1], \ldots, A[n]$
**Aanname**: $\forall i : 1 \leq A[i] \leq k$ voor een of andere constante $k$
**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 worden gebruikt om $X$ meteen op de juiste positie te zetten in het uitvoerarray. Onder bovenstaande aanname over de invoer kan dat zonder (array)vergelijkingen in $O(n)$ stappen. Pas dit idee aan voor de situatie waarin waarden vaker kunnen voorkomen in $A$.

# Het algoritme

```
1  for i := 1 to k do
2  |  C[i] := 0;                                  // initialisatie
3  od
4  for j := 1 to n do
5  |  C[A[j]] := C[A[j]] + 1;
   |  // telt hoe vaak de waarde A[j] voorkomt in A
6  od
7  for i := 2 to k do
8  |  C[i] := C[i] + C[i − 1];
9  od
   // C[i] bevat nu het aantal getallen ≤ i uit A
10 for j := n down to 1 do                        // !!
11 |  B[C[A[j]]] := A[j];
12 |  C[A[j]] := C[A[j]] − 1;
13 od
```

# Voorbeeld

$A = 3\ 6\ 4\ 1\ 3\ 4\ 1\ 4$ $\qquad n = 8$

$C = 0\ 0\ 0\ 0\ 0\ 0$ $\qquad k = 6$

$C = 2\ 0\ 2\ 3\ 0\ 1$ $\qquad \leftarrow$ na tweede for

$C[i]$ = aantal keer dat $i$ voorkomt in $A$

$C = 2\ 2\ 4\ 7\ 7\ 8$ $\qquad \leftarrow$ na derde for

$C[i]$ = aantal array-elementen $\leq i$

# De complexiteit

De complexiteit van Counting sort wordt bepaald door het aantal toekenningen (:=) aan array-elementen, en dat er zijn $\Theta(n + k)$. Indien $k \in O(n)$ is de complexiteit dus $O(n)$.

Extra geheugenruimte is ook $\Theta(n + k)$.

Counting sort werkt voor invoerrijtjes waarvan de elementen zijn begrensd (bijvoorbeeld tussen $1$ en $k$).

Counting sort is een stabiele sorteermethode, dat wil zeggen: gelijke waarden uit het invoerrijtje $A$ komen in precies dezelfde volgorde in het uitvoerarray $B$ te staan als dat ze stonden in $A$.

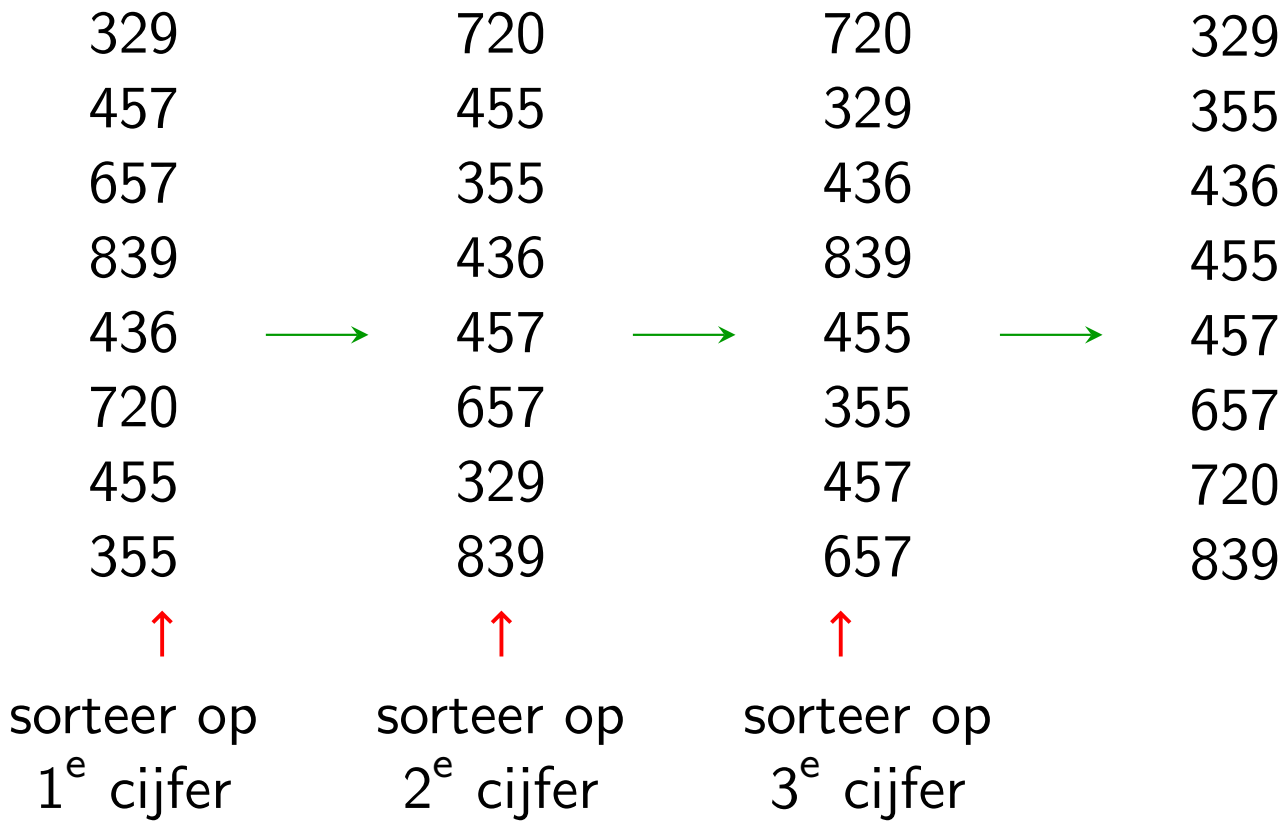
# Radix sort

De sorteermethode Radix sort sorteert $n$ getallen, elk van $d$ cijfers, waarbij elk cijfer een waarde heeft tussen $0$ en $k-1$ (bijvoorbeeld $k = 2$ of $k = 10$).

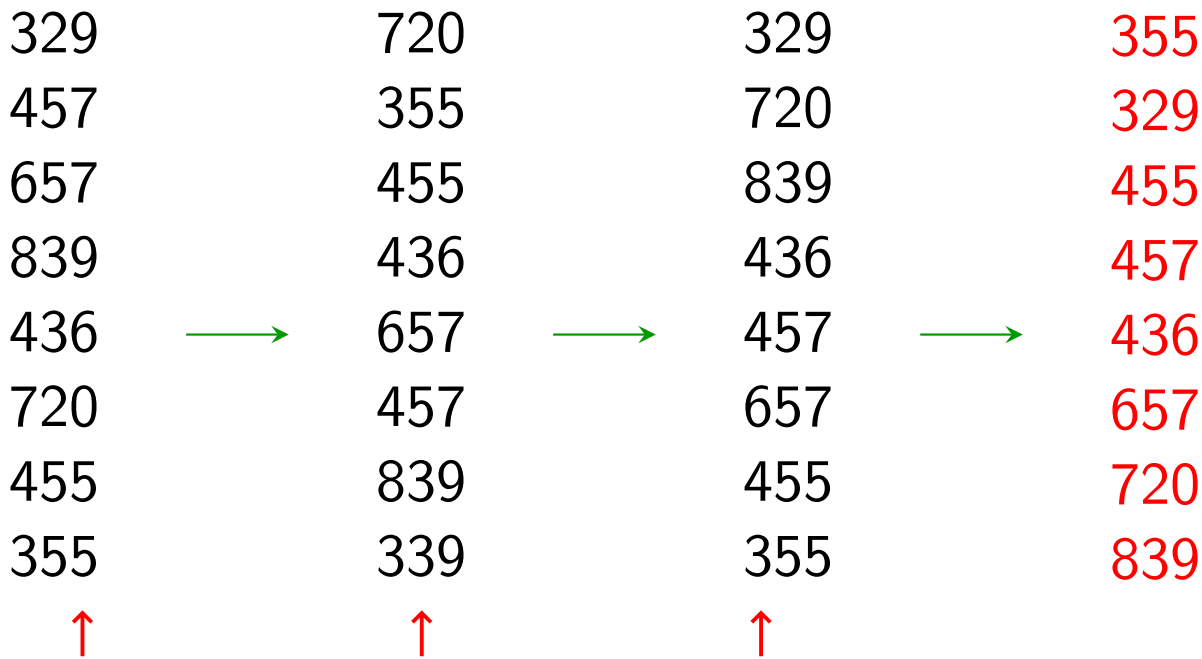
De getallen worden gesorteerd door ze achtereenvolgens op het $i^{\text{de}}$ cijfer te sorteren, te beginnen bij het minst significante cijfer. Het gebruik van een stabiele sorteermethode voor het sorteren op het $i^{\text{de}}$ cijfer is essentieel.

```
1 Radixsort(A, d)::
2 |   for i := 1 to d do
  |   |   // minst significante cijfer eerst, dus van
  |   |      rechts naar links
3 |   |   sorteer A op het i^de cijfer;
  |   |   // met een stabiele (!!) sorteermethode
4 |   od
```

# Voorbeeld


329
457
657
839
436
720
455
355
720
455
355
436
457
657
329
839
720
329
436
839
455
355
457
657
329
355
436
455
457
657
720
839
sorteer op 1e cijfer
sorteer op 2e cijfer
sorteer op 3e cijfer

# Niet stabiel




Gebruikte sorteermethode is hier Mergesort.

# Complexiteit

- als sorteermethode per cijfer kunnen we Counting sort gebruiken, aangezien de cijfers tussen $0$ en $k-1$ zitten.
- bovendien is Counting sort *stabiel* (zie eerder).
- als we Counting sort gebruiken, kost elke ronde $\Theta(k+n)$ stappen. In totaal $d$ rondes, dus $\Theta(dk+dn)$. En dat is $O(n)$ als $d$ een constante is en $k \in O(n)$.
- een nadeel van deze methode is dat er net als bij Counting sort $\Theta(n+k)$ extra geheugenruimte nodig is.
- Radix sort kun je bijvoorbeeld ook gebruiken om woorden alfabetisch te sorteren (met $k = 26$ voor het Nederlandse alfabet).

# Om te onthouden van vandaag

- Shellsort
  - sorteert deelrijtjes in rondes a.d.h.v. een serie (afnemende) stapgroottes: $k$-sorteren
  - complexiteit sterk afhankelijk van serie stapgroottes; Shell-rijtje $O(n^2)$ en Hibbard $O(n\sqrt{n})$
- Binary Insertion sort
- Merge Insertion sort: bereikt de theoretische ondergrens voor een aantal kleine $n$, maar nog steeds niet optimaal
- Counting sort: tellen en in de juiste volgorde overschrijven; $\Theta(n + k)$, of $O(n)$ als $k \in O(n)$.
- Stabiel sorteren: behoudt volgorde gelijke elementen
- Radix sort: sorteert cijfer voor cijfer, van minst significant (rechts) naar meest significant (links); stabiel intern sorteeralgoritme nodig; $\Theta(dk + dn)$, of $O(n)$ als $d$ constant en $k \in O(n)$.

# (Werk)college

**Volgende college**: dinsdag 21 maart, 9u00–10u45, zaal C3 (Gorlaeus)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304 en 303 (dus niet 306–308) (Snellius)
Opgaven uit het dictaat: 39 (minus Heapsort), 40, 41, 42, 43

# Huiswerk

**Huiswerk 1**

Is nagekeken, cijfers en feedback zijn beschikbaar op Brightspace.
Het (einde van het) werkcollege is een goed moment voor vragen.

# Ten slotte

**Ga stemmen!**

# Complexiteit 2023 — college 7

21 maart 2023

---

lets anders dan rijtjes

# Vorige keer

- Shellsort
  - sorteert deelrijtjes in rondes a.d.h.v. een serie (afnemende) stapgroottes: $k$-sorteren
  - complexiteit sterk afhankelijk van serie stapgroottes; Shell-rijtje $O(n^2)$ en Hibbard $O(n\sqrt{n})$
- Binary Insertion sort
- Merge Insertion sort: bereikt de theoretische ondergrens voor een aantal kleine $n$, maar nog steeds niet optimaal
- Counting sort: tellen en in de juiste volgorde overschrijven; $\Theta(n + k)$, of $O(n)$ als $k \in O(n)$.
- Stabiel sorteren: behoudt volgorde gelijke elementen
- Radix sort: sorteert cijfer voor cijfer, van minst significant (rechts) naar meest significant (links); stabiel intern sorteeralgoritme nodig; $\Theta(dk + dn)$, of $O(n)$ als $d$ constant en $k \in O(n)$.

# Vandaag

- Polynoomevaluatie
- Matrixvermenigvuldiging
- Eulerkringen
- Hamiltonkringen

# Polynoomevaluatie

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 ($a_i \in \mathbb{R}$).

Probleem:

**Gegeven**: $a_0, a_1, \ldots, a_n$ en $x$
**Gevraagd**: $p(x)$

Van links naar rechts de termen $a_i x^i$ berekenen en optellen geeft een $\Theta(n^2)$-algoritme.

Als we het polynoom daarentegen van rechts naar links evalueren kunnen we eenvoudig een ordeverbetering bereiken.

# Gewone methode

Algoritme 1: "gewoon"

```
1 pol := a_0 + a_1 × x; // n ≥ 1
2 macht := x;
3 for i := 2 to n do
4 |   macht := macht × x;
  |   // berekent x^2, x^3, ...
5 |   pol := pol + a_i × macht;
6 od
```

Basisoperatie: $\times$ en $+, -$

Complexiteit: aantal $\times$: $2n - 1$
aantal $+, -$: $n$

# Horner

## Algoritme 2: methode van Horner*

```
1 pol := a_n;
2 for i = n − 1 downto 0 do
3 |   pol := pol × x + a_i;
4 od
```

Gebaseerd op:

$$p(x) = a_0 + x \times (a_1 + x \times (a_2 + \ldots x \times (a_{n-1} + x \times a_n) \ldots))$$

Complexiteit: aantal $\times$: $n$
aantal $+, -$: $n$

**Vraag**: kan het met minder $\times$ en $+, -$?
**Antwoord**: nee!

*gepopulariseerd door William Horner (1819); Chinese bronnen bekend uit de derde eeuw

# Schema's

Algoritmen gebaseerd op het doen van vergelijkingen konden we beschrijven met **beslissingsbomen**.
Een model om rekenkundige algoritmen (algoritmen die zijn gebaseerd op $+$, $-$, $\times$ en $/$) te beschrijven: schema's.
Een schema

- is een eindige serie stappen van de vorm $s_i := q \circ r$;
- hierin is $\circ$ een rekenkundige operatie: $\times$, $/$, $+$ of $-$
- $q$ en $r$ zijn constanten (bijvoorbeeld $1, -1, \pi^2, \ldots$) of invoerwaarden (hier $a_i$'s en $x$) of tussenresultaten van eerdere stappen
- de laatste stap uit het schema berekent het eindresultaat (hier dus $p(x)$)

Horner met $n = 2$: $s_1 := a_2 \times x$; $s_2 := s_1 + a_1$; $s_3 := s_2 \times x$; $s_4 := s_3 + a_0$;

# Optimaliteit

Stelling. Elk schema (dat alleen +, − en × gebruikt) om $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ te berekenen moet minstens $n$ $(+,-)$-stappen doen en $n$ $\times$-stappen.
Bewijs voor $(+,-)$-stappen volgt (vervang $x$ door $1$) uit:

Lemma. Een schema (dat alleen +, − en × gebruikt) om $a_0 + a_1 + a_2 + \ldots + a_{n-1} + a_n$ te berekenen moet ten minste $n$ $(+,-)$-stappen doen.
Het bewijs gaat met inductie op $n$, met $n = 0$ als flauw basisgeval.
Vervang overal $a_n$ door $0$. Dit geeft een schema dat $a_0 + a_1 + a_2 + \ldots + a_{n-1}$ berekent. Kijk naar de eerste $(+,-)$-stap die $a_n$ gebruikt (die bestaat zeker): $s_i := q \pm 0$ of $s_i := 0 + r$ of $s_i := 0 - r$; laat de twee eerste weg en vervang overal $s_i$ door $q$ danwel $r$, of $s_i := -1 \times r$ vervangt zo nodig de derde. We hebben zo een $(+,-)$-stap geëlimineerd uit het schema.
Gebruik nu inductie.

# Optimaliteit — en verder

De methode van Horner berekent $p(x)$ met $n$ vermenigvuldigingen en $n$ optellingen $(+, -)$. Er bestaat geen algoritme dat het probleem voor algemene $p$ en $x$ kan oplossen met minder vermenigvuldigingen en optellingen. De methode van Horner is derhalve optimaal.*

Maar misschien kan het wel beter voor polynomen die een heel speciale vorm hebben.

*de schets van het bewijs van de ondergrens (vorige twee slides) is geen tentamenstof

# Met preprocessing

Polynoomevaluatie met preprocessing: bewerk het polynoom tot een polynoom in een speciale vorm waarop een nieuw evaluatiealgoritme sneller werkt.

## Het polynoom

Laat $p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, met $n = 2^k - 1$.

$p(x)$ is dus een monisch polynoom, dat wil zeggen dat $a_n = 1$. We kunnen zonder beperking der algemeenheid aannemen dat het te evalueren polynoom monisch is. Hetzelfde geldt voor de aanname dat $n = 2^k - 1$.

# Speciale vorm

De speciale vorm (recursief geformuleerd):

$$p(x) = (x^j + b) \times q(x) + r(x)$$

waarin $q$ en $r$ ook weer monisch zijn en in de speciale vorm staan, beide van graad $2^{k-1} - 1$ zijn, en $j = 2^{k-1}$.

Voorbeeld (met $n = 7$)

$$p(x) = x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x + 1 =$$

$$(x^4+2)[(x^2+4)(x+6)+(x-20)]+[(x^2-10)(x-10)+(x-107)]$$

Om dit polynoom in $x$ te evalueren gebruikt Horner $7$ vermenigvuldigingen en $7$ optellingen, maar het kan met $5$ en $10$.

# Constructie

Een gegeven monisch polynoom $p$ van graad $n = 2^k - 1$ is eenvoudig om te zetten naar de speciale vorm.

We willen $p$ schrijven als: $p(x) = (x^j + b) \times q(x) + r(x)$ met $q$ en $r$ monisch, beide van graad $2^{k-1} - 1$, en $j = 2^{k-1}$. De waarde van $b$ en de coëfficiënten van $q$ en $r$ zijn hieruit simpel af te lezen. Immers: 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:

$$b + 1 = a_{j-1},\ q_\ell = a_{\ell+j} \text{ en } b \times q_\ell + r_\ell = a_\ell,$$

voor $\ell = 0, 1, \ldots, j-2$ en de $a_i$ de coëfficiënten van $p$.

Vervolgens kunnen $q$ en $r$ op dezelfde manier in de speciale vorm worden gebracht. Algoritme met complexiteit: zie opgave 48.

# Evaluatie

Als het polynoom $p$ in de juiste vorm staat kan $p(x)$ als volgt worden geëvalueerd:

1. evalueer $q(x)$ en $r(x)$ recursief
2. bereken de $x^j$'s; nodig hiervoor zijn $x, x^2, x^4, \ldots, x^{2^{k-1}}$. Bereken deze alle van tevoren: $k - 1$ vermenigvuldigingen
3. vermenigvuldig $(x^j + b)$ met $q(x)$ en tel er $r(x)$ bij op: $1$ vermenigvuldiging en $2$ optellingen

# Aantal vermenigvuldigingen

Zij $M(k)$ het aantal vermenigvuldigingen dat wordt gedaan om een monisch polynoom (in de speciale vorm) van graad $2^k - 1$ te evalueren, zonder de berekening van de $x^j$ mee te tellen.

Dan voldoet $M(k)$ aan de volgende recurrente betrekking:

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

**Oplossing**: $M(k) = 2^{k-1} - 1 \rightarrow \frac{n-1}{2}$

# Aantal optellingen

Zij $A(k)$ het aantal optellingen dat wordt gedaan om een monisch polynoom (in de speciale vorm) van graad $2^k - 1$ te evalueren.

Dan voldoet $A(k)$ aan de volgende recurrente betrekking:

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

**Oplossing**: $A(k) = 3 \times 2^{k-1} - 2 \rightarrow \frac{3n-1}{2}$

# Matrixvermenigvuldiging

Zij $A$ en $B$ twee $n \times n$ matrices met elementen $a_{ij}$ en $b_{ij}$ $(1 \leq i, j \leq n)$. De elementen $c_{ij}$ van het (matrix-)product $C = A \times B$ zijn dan gedefinieerd als volgt: $c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$.

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

```
1 for i := 1 to n do
2 |   for j := 1 to n do
3 |   |   c_ij := 0;
4 |   |   for k := 1 to n do
5 |   |   |   c_ij := c_ij + a_ik × b_kj;
6 |   |   od
7 |   od
8 od
```

# Voorbeeld $2 \times 2$

Voorbeeld: product van twee $2 \times 2$-matrices algemeen.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

$$\begin{aligned} c_{11} &= a_{11} \times b_{11} + a_{12} \times b_{21} \\ c_{12} &= a_{11} \times b_{12} + a_{12} \times b_{22} \\ c_{21} &= a_{21} \times b_{11} + a_{22} \times b_{21} \\ c_{22} &= a_{21} \times b_{12} + a_{22} \times b_{22} \end{aligned}$$

Het product van twee matrices is overigens ook gedefinieerd voor niet-vierkante matrices $A$ en $B$, mits maar geldt dat het aantal kolommen van $A$ gelijk is aan het aantal rijen van $B$.

# Voorbeeld $2 \times 2$

We berekenen $c_{21}$

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

via $c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$.

We kunnen $c_{21}$ trouwens ook berekenen als:
$(a_{21} + a_{22}) \times b_{11} + a_{22} \times (b_{21} - b_{11})$. Dit lijkt weinig zinvol, maar toch...

# Voorbeeld $2 \times 2$: aantal vermenigvuldigingen

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

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

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

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

Algemeen: $M_2 = (a_{21} + a_{22}) \times b_{11}$; $M_4 = a_{22} \times (b_{21} - b_{11})$; dan $M_2 + M_4 = c_{21}$, etc.

# Recursief

Neem aan dat $n = 2^k$. We kunnen dan de matrices $A$, $B$ en $C$ iedere opsplitsen in vier $\frac{n}{2} \times \frac{n}{2}$-deelmatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Terzijde: de complexiteit van matrixvermenigvuldiging is in $\Omega(n^2)$, want ieder algoritme moet in ieder geval de $n \times n$ matrices $A$ en $B$ lezen en het resultaat (alle $c_{ij}$) berekenen / schrijven.

# Recursief

We kunnen $C = A \times B$ recursief berekenen via*:

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

$$\begin{aligned} C_{11} &= A_{11} \times B_{11} + A_{12} \times B_{21} \\ C_{12} &= A_{11} \times B_{12} + A_{12} \times B_{22} \\ C_{21} &= A_{21} \times B_{11} + A_{22} \times B_{21} \\ C_{22} &= A_{21} \times B_{12} + A_{22} \times B_{22} \end{aligned}$$

Dit zijn $8$ vermenigvuldigingen en $4$ optellingen van $\frac{n}{2} \times \frac{n}{2}$-matrices. Voor $M(k)$, het aantal vermenigvuldigingen van elementen, geldt: $M(k) = 8M(k-1)$ en $M(0) = 1$, wat $M(k) = 8^k = n^3$ geeft. Hetzelfde als de "gewone" methode.

*ga voor $n = 4$ na dat dit correct is door een en ander uit te schrijven

# Strassen

Analoog aan het $2 \times 2$-geval kunnen we de berekening van de vier $C_{ij}$'s herschrijven zodat 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 zijn van twee deelmatrices van $A$ en $B$:

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

# Strassen

$P_1, \ldots, P_7$ zijn $\frac{n}{2} \times \frac{n}{2}$-matrices die alle het product zijn van twee matrices $S$ en/of deelmatrices van $A$ en $B$:

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

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

# Strassen

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

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

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.

Denk eraan dat matrixvermenigvuldiging niet commutatief is: doorgaans geldt $A \times B \neq B \times A$. Merk op dat bij het herschrijven op deze en de vorige twee slides nergens stiekem $X \times Y$ door $Y \times X$ is vervangen; alle gelijkheden zijn dan ook geldig. Controleer zelf dat het allemaal klopt.

# Recurrente betrekking

Dit levert de volgende recurrente betrekkingen op voor het aantal vermenigvuldigingen $M(k)$ en het aantal optellingen $A(k)$ van elementen, met $n = 2^k$:

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

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

Oplossing: $M(k) = 7^k = 2^{\lg 7^k} = 2^{k \lg 7} = n^{\lg 7} \approx n^{2,81}$
$A(k) = 6 \times 7^k - 6 \times 4^k \in \Theta(n^{\lg 7})$.

# Nog sneller

We hebben dus: Strassen (1969) met $\Theta(n^{2,81})$.

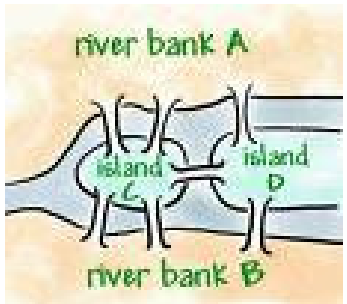
Het kan nog sneller!

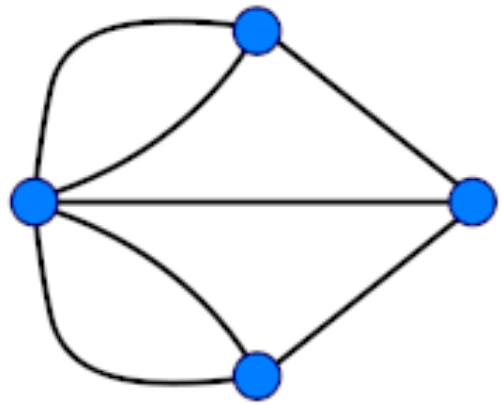
| | |
|---|---|
| Pan (1978) | $\Theta(n^{2,796})$ |
| Coppersmith-Winograd (1990) | $\Theta(n^{2,376})$ |
| Stothers (2010) | $\Theta(n^{2,3737})$ |
| Williams (2012) | $\Theta(n^{2,3729})$ |
| Le Gall (2014) | $\Theta(n^{2,3728639})$ |

Dit soort algoritmen heeft echter zulke grote constanten verstopt in de $\Theta$ dat ze alleen asymptotisch sneller zijn dan Strassen voor matrices die te groot zijn om met de huidige hardware te berekenen. Vermoeden: $\Theta(n^{2+\epsilon})$ voor elke $\epsilon > 0$.

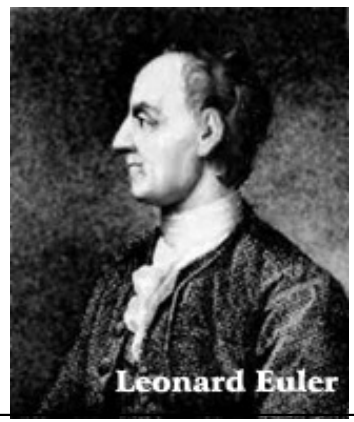
# Euler

Koningsberger bruggenprobleem:



Kun je een wandeling door de stad maken waarbij je elke brug precies één keer beloopt en je weer terugkeert in het beginpunt?




Leonard Euler, 1736*

*en.wikipedia.org/wiki/List_of_things_named_after_Leonhard_Euler

# Definities

Gegeven een samenhangende, ongerichte graaf $\mathcal{G} = (V, E)$.

- een pad van $u$ naar $v$ is een rij knopen waarvoor geldt dat tussen elk tweetal opeenvolgende knopen uit die rij een tak (lijn) zit.
- de lengte van een pad is het aantal takken op dat pad is het aantal knopen $- 1$.
- een kring in een ongerichte graaf is een pad dat begint en eindigt in dezelfde knoop (een *gesloten* pad dus) en dat geen enkele tak meer dan één keer bevat.* Een kring bestaat dus uit allemaal verschillende takken.
- een gesloten pad dat geen enkele *knoop* meer dan één keer bevat† is een speciaal geval van een kring. Zo'n pad bestaat dus uit allemaal verschillende knopen.

*Foundations of Computer Science (FoCS): *circuit*

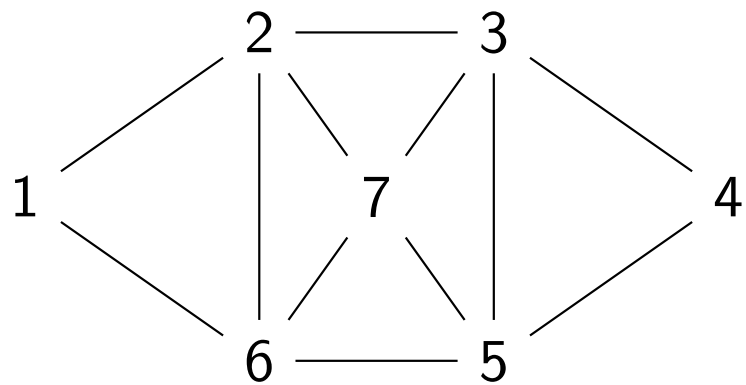
†FoCS: *cykel*

# Eulerkringen

Gegeven een samenhangende, ongerichte graaf $\mathcal{G} = (V, E)$.

**Definitie**: een kring in $\mathcal{G}$ die alle takken van $\mathcal{G}$ bevat heet een Eulerkring. Deze bevat dus *elke* tak precies één keer.

**Voorbeeld**:




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

# Eulerkringprobleem

Eulerkringprobleem. Gegeven een samenhangende ongerichte graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Eulerkring?

Dit is een voorbeeld van een beslissingsprobleem: het antwoord is ja of nee.

Stelling
Een samenhangende ongerichte graaf heeft een Eulerkring $\Leftrightarrow$ de graad (d.w.z. het aantal buren) van iedere knoop is even.

Het is dus heel gemakkelijk (d.w.z. polynomiaal) na te gaan of een graaf een Eulerkring heeft: $O(|E|)$.

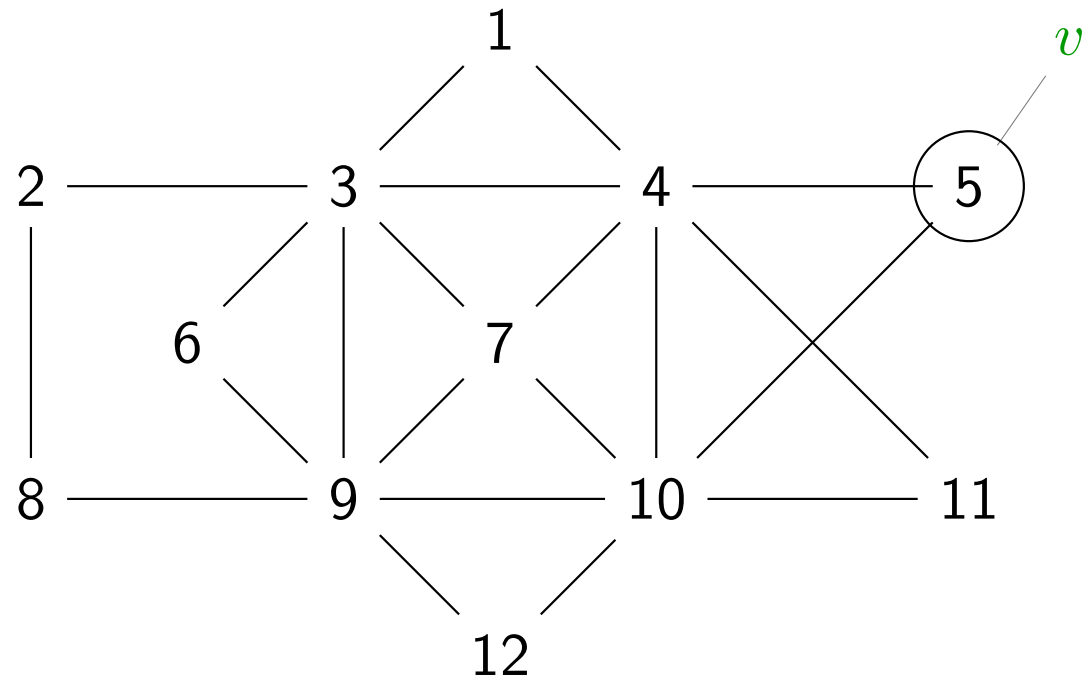
Opmerking
Het bewijs van "$\Leftarrow$" is constructief: het geeft je meteen een (overigens) $O(|E| + |V|)$ algoritme om zo'n Eulerkring te vinden.

# Constructie

**Algoritme** voor constructie Eulerkring (Hierholzer, 1873):

```
1  controleer de samenhangendheid en controleer of elke knoop
     even graad heeft;
2  if controle positief then
3  |   kies een knoop v;
4  |   construeer een kring C (van v naar v);
   |   // gewoon lopen en steeds andere, ongebruikte
   |      takken bewandelen tot je terug bent in v
5  |   while er nog onbewandelde takken zijn do
6  |   |   zoek/onthoud een knoop w van C die nog
   |   |     onbewandelde takken heeft;
7  |   |   construeer een kring (van w naar w) uit ongebruikte
   |   |     takken;
8  |   |   voeg deze kring in in C op de plek van w;
9  |   od
10 fi
```

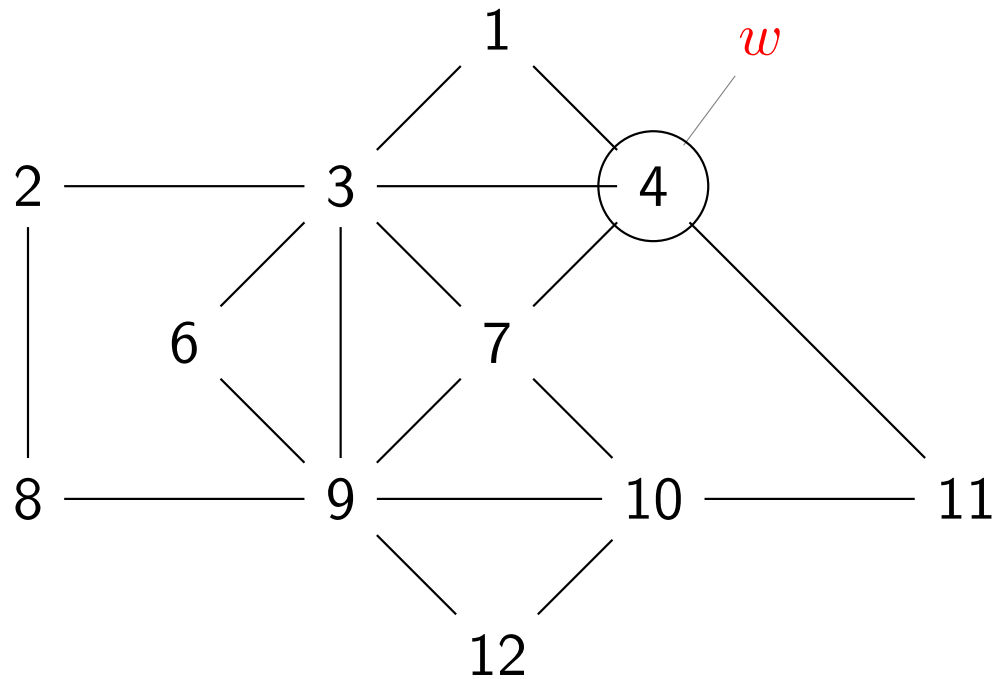
# Opbouw Eulerkring




Kring vanuit $v$: 5, 4, 10, 5

$\mathcal{C}$ = 5, 4, 10, 5

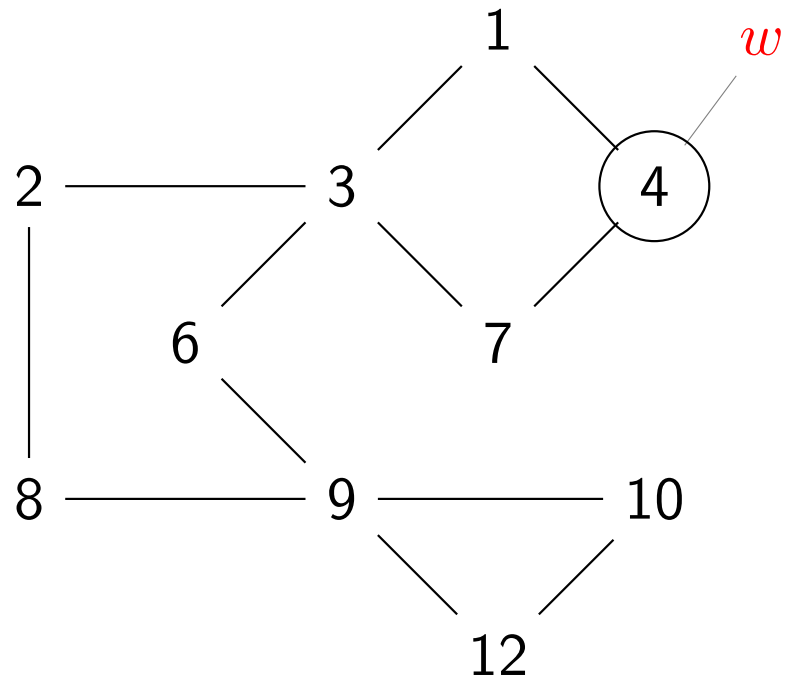
# Opbouw Eulerkring




Kring vanuit $w$: 4, 11, 10, 7, 9, 3, 4

$\mathcal{C}$ = 5, 4, 11, 10, 7, 9, 3, 4, 10, 5

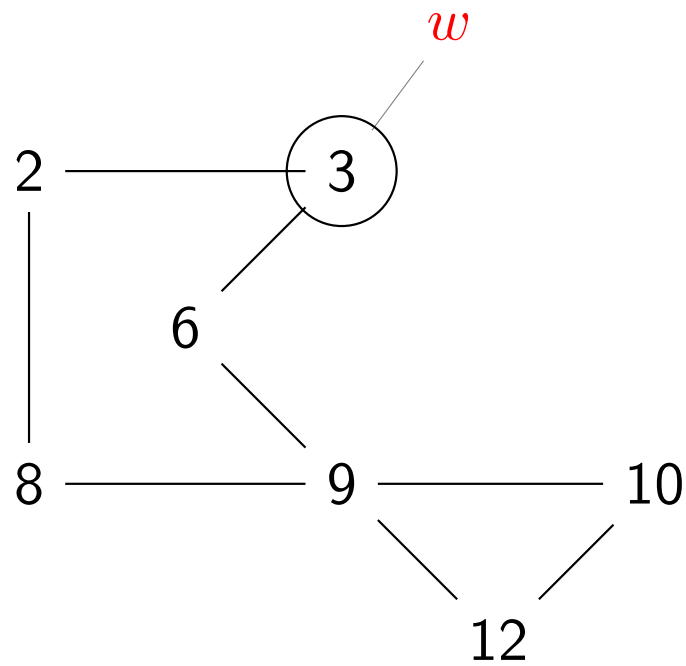
# Opbouw Eulerkring




Kring vanuit $w$: 4, 1, 3, 7, 4

$\mathcal{C}$ = 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

# Opbouw Eulerkring




Kring vanuit $w$: 3, 2, 8, 9, 10, 12, 9, 6, 3
(of eerst 3, 2, 8, 9, 6, 3 en dan 9, 10, 12, 9)

$\mathcal{C}$ = 5, 4, 1, 3, 2, 8, 9, 10, 12, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5

# Complexiteit

**Complexiteit** van dit algoritme: $O(|V| + |E|)$

Stap 1: samenhangendheid controleren met behulp van DFS en tegelijk de graden bepalen: $O(|V| + |E|)$.

Stap 4 t/m 8: kan in $O(|E|)$ stappen

- gebruik een kopie van $\mathcal{G}$ ($O(|V| + |E|)$ om die te maken).
- haal tijdens de constructie van $\mathcal{C}$ een tak weg (uit de kopie) zodra die is gebruikt.
- houd de kring $\mathcal{C}$ bij als enkelverbonden lijst, met een pointer naar de eerste knoop die nog onbewandelde takken heeft. We kiezen in stap 6 dus de eerste de beste knoop op $\mathcal{C}$ die nog onbewandelde takken heeft.

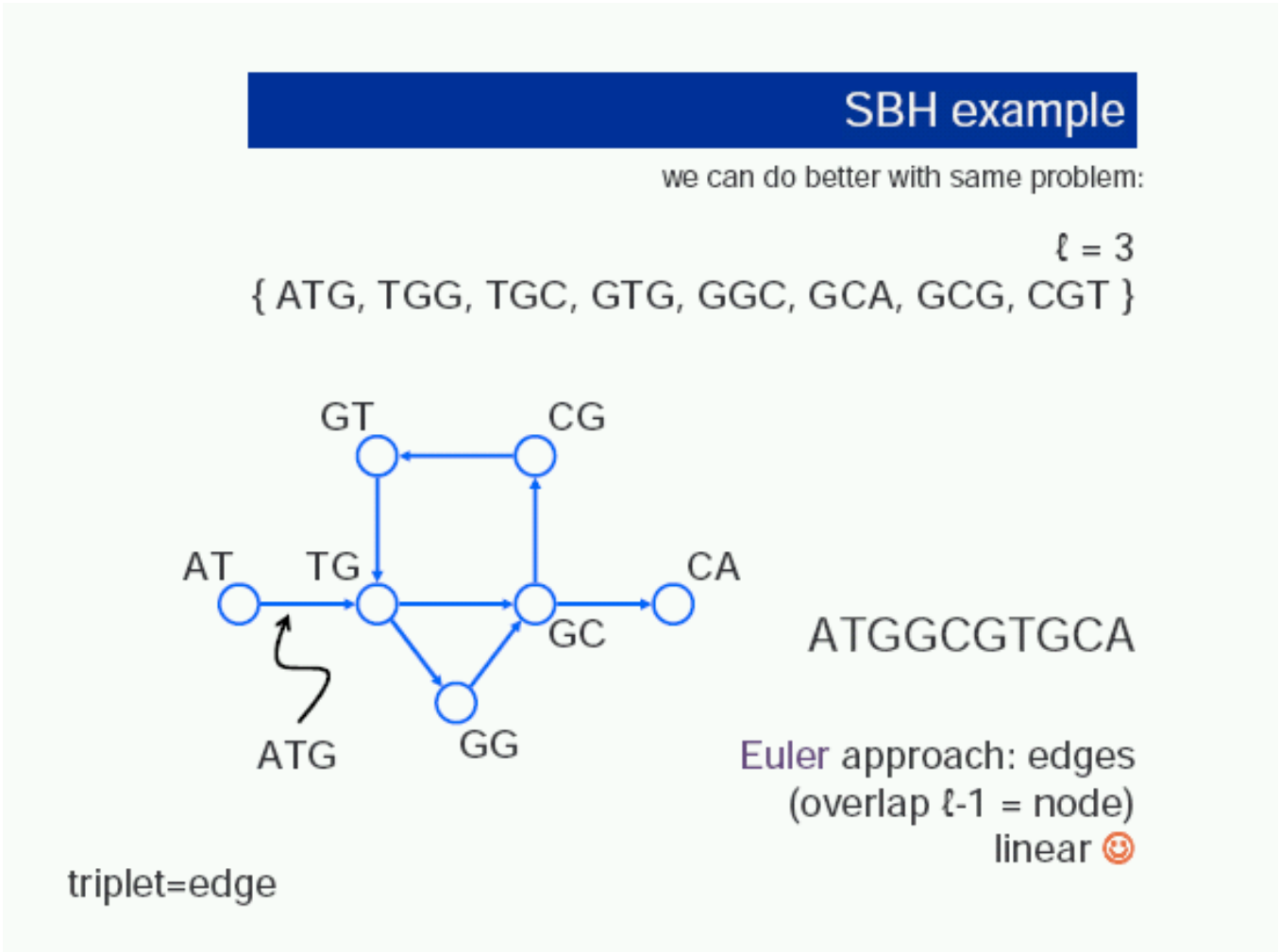
# Opmerkingen

Opmerkingen bij implementatie en complexiteit:

- gebruik een adjacencylist als datastructuur en laat de buurlijsten oplopend zijn gesorteerd
- als een tak $(v, w)$ is gebruikt, halen we alleen $w$ uit de buurlijst van $v$ maar niet andersom (dat komt later)
- de twee implementatie-opmerkingen hierboven zijn bedoeld om complexiteit $O(|E|)$ te verkrijgen i.p.v. $O(|E|^2)$
- zie opgave 50
- zie opgave 52 voor een alternatief (langzamer) algoritme (Fleury, 1883)
- om de complexiteit te beschrijven hebben we verschillende soorten operaties geteld en op één hoop gegooid

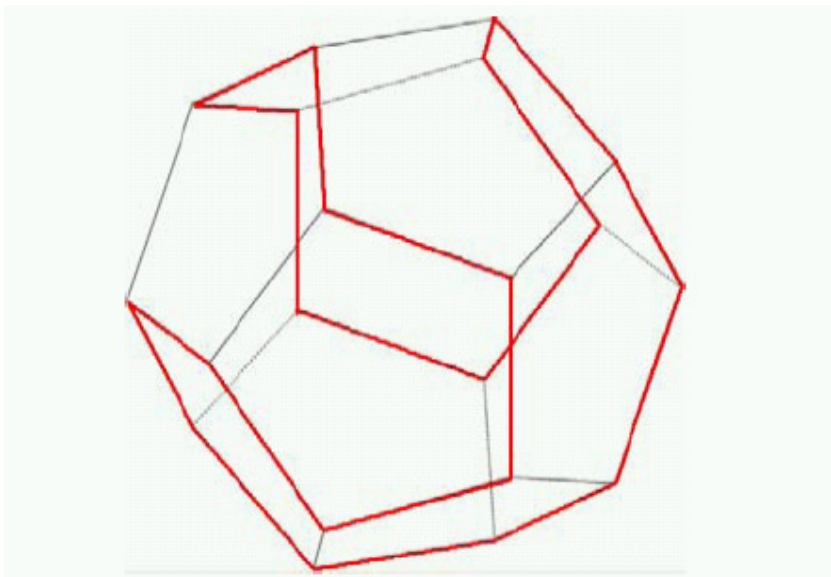
# Toepassing (DNA)

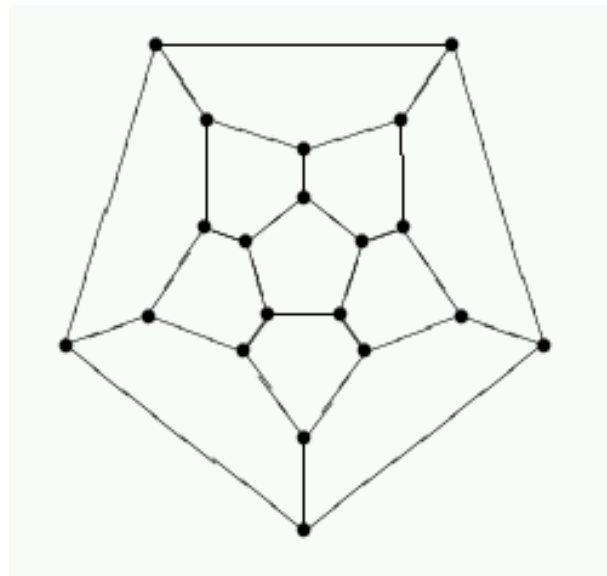
Uit: Molecular Computational Biology (oud mastercollege)



# Hamilton

Kun je een wandeling door de graaf rechtsonder maken waarbij elke *knoop* precies één keer wordt bezocht en je weer eindigt in het startpunt?





Sir William Rowan Hamilton, 1859*

*https://en.wikipedia.org/wiki/List_of_things_named_after_William_Rowan_Hamilton

# Hamiltonkringprobleem

Gegeven een ongerichte (of gerichte) graaf $\mathcal{G} = (V, E)$.

**Definitie**: een Hamiltonkring in $\mathcal{G}$ is een kring die elke knoop precies één keer bevat.

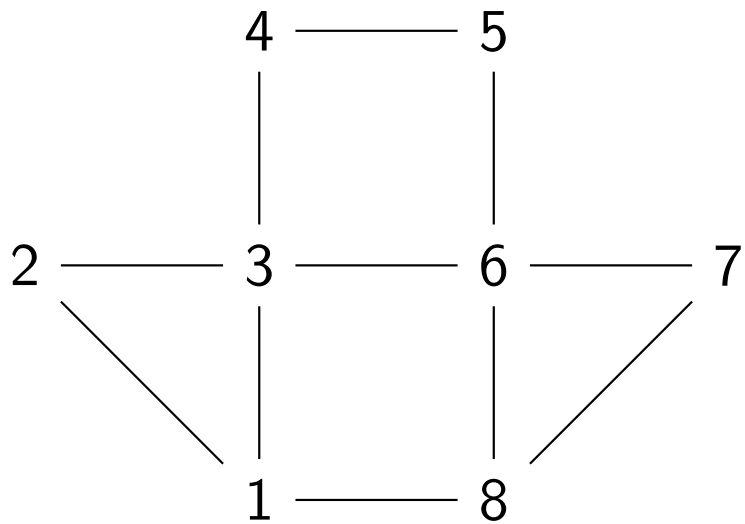
Bijbehorend beslissingsprobleem: Hamiltonkringprobleem (HC) voor ongerichte grafen*. Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Hamiltonkring?

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

*analoog voor gerichte grafen

# Voorbeeld 1

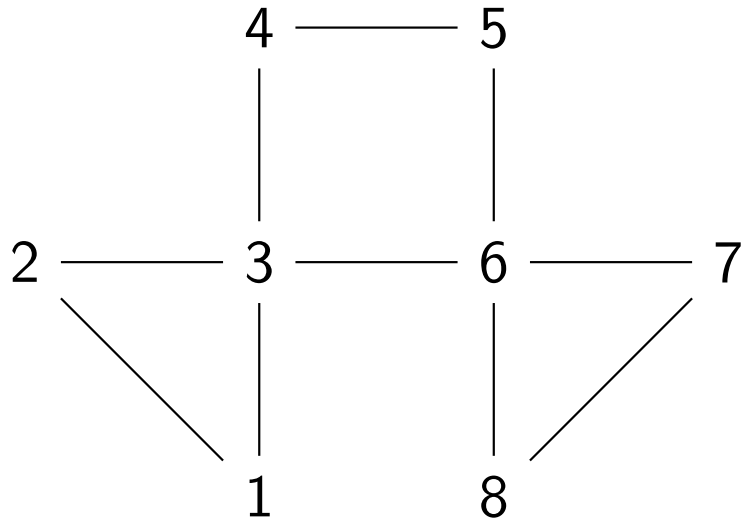
**Voorbeeld 1**:




Deze graaf heeft een Hamiltonkring: 1, 2, 3, 4, 5, 6, 7, 8, 1.

# Voorbeeld 2

**Voorbeeld 2**:




Deze graaf heeft geen Hamiltonkring, maar wel een Hamiltonpad: 1, 2, 3, 4, 5, 6, 7, 8.

# Toepassing

Vermomde Hamiltonproblemen:

1. kan een paard over een $n \times n$-schaakbord een serie sprongen maken zodanig dat het alle $n^2$ velden precies één keer bezoekt en ten slotte weer terugkeert in zijn beginveld?
2. gegeven een groep van $n$ mensen. Van elk tweetal is bekend of ze elkaar wel of niet kennen. Kunnen deze mensen zo aan een ronde tafel worden geplaatst, dat iedereen twee bekenden heeft als buren?
3. (Gray-code) er zijn $2^n$ binaire getallen van $n$ bits. Kunnen deze zo (cyclisch) achter elkaar worden geplaatst dat opeenvolgende getallen slechts op één plaats verschillen?

# Moeilijkheid HC

1. een (super)exponentieel algoritme is snel gevonden: genereer alle $n!$ mogelijke Hamiltonkringen ($n$ is het aantal knopen van $\mathcal{G}$) en controleer voor elk of het een Hamiltonkring is.
2. het *controleren* of een kandidaat-Hamiltonkring echt een Hamiltonkring is, is *polynomiaal*.
3. het *vinden* van een Hamiltonkring daarentegen is/lijkt *héééééél moeilijk* (d.w.z. (super)exponentieel).
4. er is geen polynomiaal algoritme bekend voor dit probleem; zelfs niet voor het beslissingsprobleem HC, dat alleen vraagt *of* $\mathcal{G}$ een Hamiltonkring heeft.
5. er is ook niet bewezen dat een exponentieel algoritme *nodig* is om HC op te lossen.

# Instanties

6. als $\mathcal{G}$ een ja-instantie is van HC, dus een Hamiltonkring heeft, is er een eenvoudige (d.w.z. polynomiale) manier om dat aan te tonen met behulp van de juiste hint (certificaat). Immers, in dit geval is het certificaat zo'n Hamiltonkring: controle of deze echt een Hamiltonkring is, is polynomiaal.
7. de enige manier om te laten zien dat $\mathcal{G}$ een nee-instantie is, dus geen Hamiltonkring bevat, lijkt: som alle $n!$ kandidaat-Hamiltonkringen op en laat zien dat ze alle geen Hamiltonkring zijn. Dat is (super)exponentieel.
8. er zijn eenvoudige en lastige instanties... → NP-volledigheid

# Om te onthouden van vandaag

- Polynoomevaluatie
  - Methode van Horner
  - Met preprocessing en monische polynomen
- Matrixvermenigvuldiging
  - Strassen: $\Theta(n^{\lg 7})$
- Eulerkringen
  - d.e.s.d.a. graad van elke knoop even
  - Constructiealgoritme
- Hamiltonkringen
  - kandidaat-oplossing polynomiaal te controleren
  - gewoon oplossen vermoedelijk hééééééél moeilijk

# (Werk)college

28 maart: géén hoor- of werkcollege

**Volgende college**: dinsdag 4 april, 9u00–10u45, zaal 412 (Snellius)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304 en 303 (Snellius)
Opgaven uit het dictaat: 49, 51, 52a, 47, 48

# Huiswerk

**Huiswerk 2**

Inleveren op Brightspace, vóór maandagmiddag 27 maart. D.w.z.: uiterlijk maandag 27 maart 11u59. Daarna nog wel feedback maar geen cijfer.

# Complexiteit 2023 — college 8

4 april 2023

---

## Inleiding complexiteitstheorie

# De eerste helft van het semester, héél kort samengevat

- Analyse van algoritmen
- "Hoeveel werk" doet een gegeven algoritme?
  - Tellen basisoperaties
  - In orde van grootte: $O$, $\Omega$, $\Theta$, best/worst/average case
- "Hoeveel werk" kost een probleem minstens
  - Ondergrens voor worst case van elk algoritme (op basis van specifieke basisoperatie)
  - Beslissingsbomen, adversary-argumenten, ad hoc
- Problemen en algoritmen voor:
  - Zoeken en selectie
  - Sorteren
  - Polynoomevaluatie en matrixvermenigvuldiging
  - Eulerkringen en Hamiltonkringen

# Complexiteitstheorie

# De tweede helft van het semester

**Complexiteitstheorie**

(meer overzicht volgt aan het eind van het college)

# Vandaag

- Inleiding complexiteitstheorie
- Handelbare, onhandelbare en onbeslisbare problemen
- De complexiteitsklassen $\mathcal{P}$ (formeel) en $\mathcal{NP}$ en $\mathcal{NPC}$ (informeel)

# Handelbaar/onhandelbaar

| $n$ | $10$ | $100$ | $1\,000$ | $100\,000$ | $10\,000\,000$ |
|---|---|---|---|---|---|
| $\lg n$ | $3$ | $6$ | $9$ | $16$ | $23$ |
| $\sqrt{n}$ | $3$ | $10$ | $32$ | $316$ | $3\,162$ |
| $n$ | $10$ | $100$ | $1\,000$ | $100\,000$ | $10\,000\,000$ |
| $n \lg n$ | $33$ | $664$ | $9\,966$ | $1\,660\,964$ | $\approx 10^8$ |
| $n^2$ | $100$ | $10\,000$ | $1\,000\,000$ | $10^{10}$ | $10^{14}$ |
| $n^{10}$ | $10^{10}$ | $10^{20}$ | $10^{30}$ | $10^{50}$ | $10^{70}$ |
| $2^n$ | $1\,024$ | $\approx 10^{30}$ | $\approx 10^{301}$ | $\approx 10^{30\,102}$ | veel |
| $n!$ | $3\,628\,800$ | $\approx 10^{157}$ | $\approx 10^{2\,567}$ | $\approx 10^{456\,573}$ | veel |
| $n^n$ | $10^{10}$ | $10^{200}$ | $10^{3\,000}$ | $10^{500\,000}$ | $10^{70\,000\,000}$ |

Ter vergelijking:

- het aantal protonen in het heelal heeft 79 cijfers
- het aantal microseconden sinds de Oerknal heeft 24 cijfers

# Handelbaar/onhandelbaar

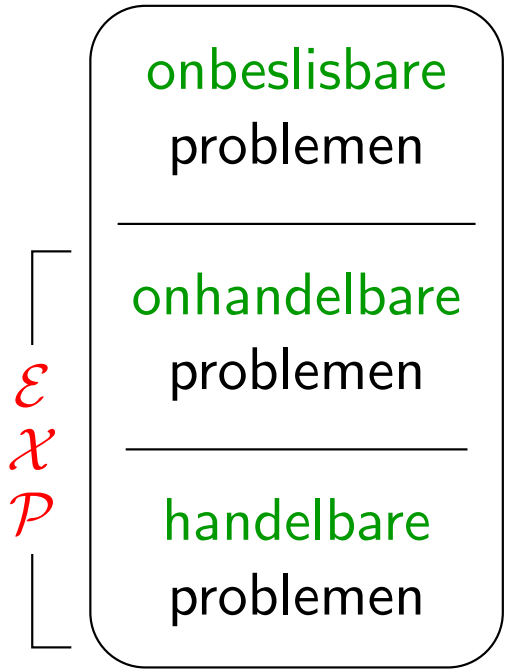
Met één miljoen ($10^6$) instructies per seconde:

| $n$ | 10 | 20 | 50 | 100 | 300 |
|---|---|---|---|---|---|
| $n^2$ | $\frac{1}{10\,000}$ sec | $\frac{1}{2\,500}$ sec | $\frac{1}{400}$ sec | $\frac{1}{100}$ sec | $\frac{9}{100}$ sec |
| $n^5$ | $\frac{1}{10}$ sec | 3,2 sec | 5,2 min | 2,8 uur | 28,1 dag |
| $2^n$ | $\frac{1}{1000}$ sec | 1 sec | 35,7 jaar | 400 biljoen eeuwen | 75-cijfers veel eeuwen |
| $n^n$ | 2,8 uur | 3,3 biljoen jaar | 70-cijfers veel eeuwen | 185-cijfers veel eeuwen | 728-cijfers veel eeuwen |

Ter vergelijking: de Oerknal was ongeveer 15 miljard jaar geleden.
($\approx 2 \times 13!$ en $\approx 2^{34}$)

# Indeling problemen

We kunnen problemen globaal indelen in drie klassen:




Problemen waarvoor geen algoritme mogelijk is (stop/halting-probleem)

Problemen waarvoor geen polynomiaal algoritme mogelijk is

Problemen waarvoor een polymiaal algoritme bestaat: $\mathcal{P}$

Beslisbare problemen:

- in acceptabele tijd oplosbaar (deterministisch)
- in niet-acceptabele tijd oplosbaar (deterministisch)

# $\mathcal{P}$ en $\mathcal{EXP}$

$\mathcal{EXP}$ is de klasse van problemen waarvoor een exponentieel algoritme bestaat, dus met worst case-complexiteit $O(2^{p(n)})$ voor een of ander polynoom $p$ en $n$ een maat voor de invoergrootte. Er bestaan ook nog $2\mathcal{EXP}$, $3\mathcal{EXP}$, ...

$\mathcal{P}$ is de klasse van problemen* waarvoor een polynomiaal algoritme bestaat, dus met worst case-complexiteit $O(p(n))$.

Onder $\mathcal{EXP}$ vallen uiteraard ook alle polynomiaal oplosbare problemen: $\mathcal{P} \subset \mathcal{EXP}$.

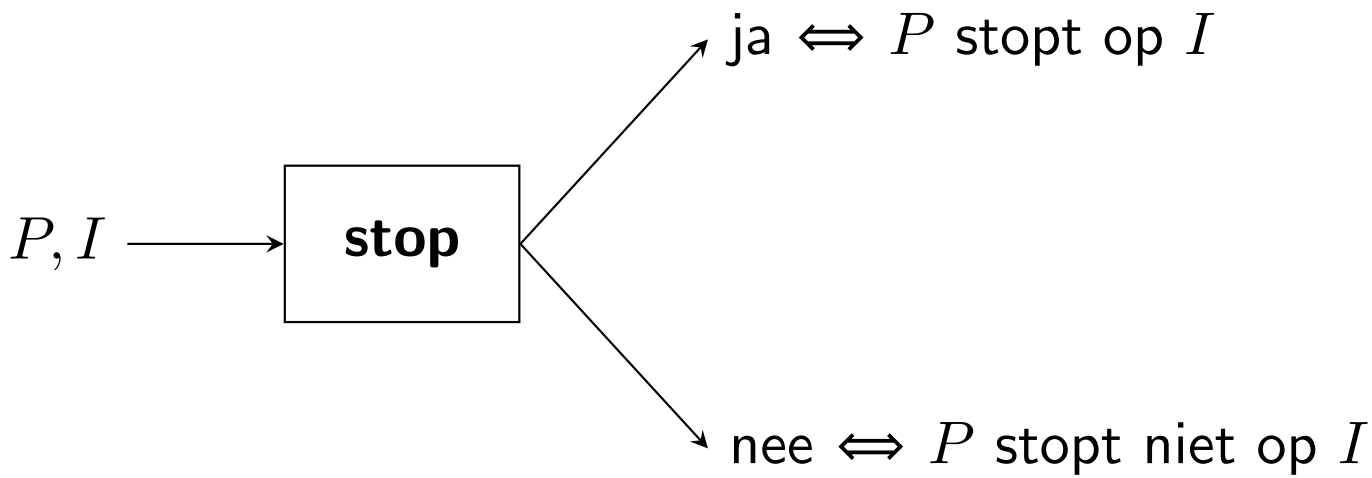
De moeilijkste problemen uit $\mathcal{EXP}$ zijn problemen die echt een exponentieel algoritme nodig hebben, dus problemen met $\Theta(2^{p(n)})$ als ondergrens voor de complexiteit. Problemen die (minstens) exponentieel zijn noemen we onhandelbaar.

*later beperken we ons tot beslissingsproblemen

# Halting-probleem

Stopprobleem:

Bestaat er een algoritme **stop** dat voor een willekeurig programma $P$ en invoer $I$ kan bepalen of $P$ wel of niet stopt op invoer $I$?



# De klasse $\mathcal{P}$

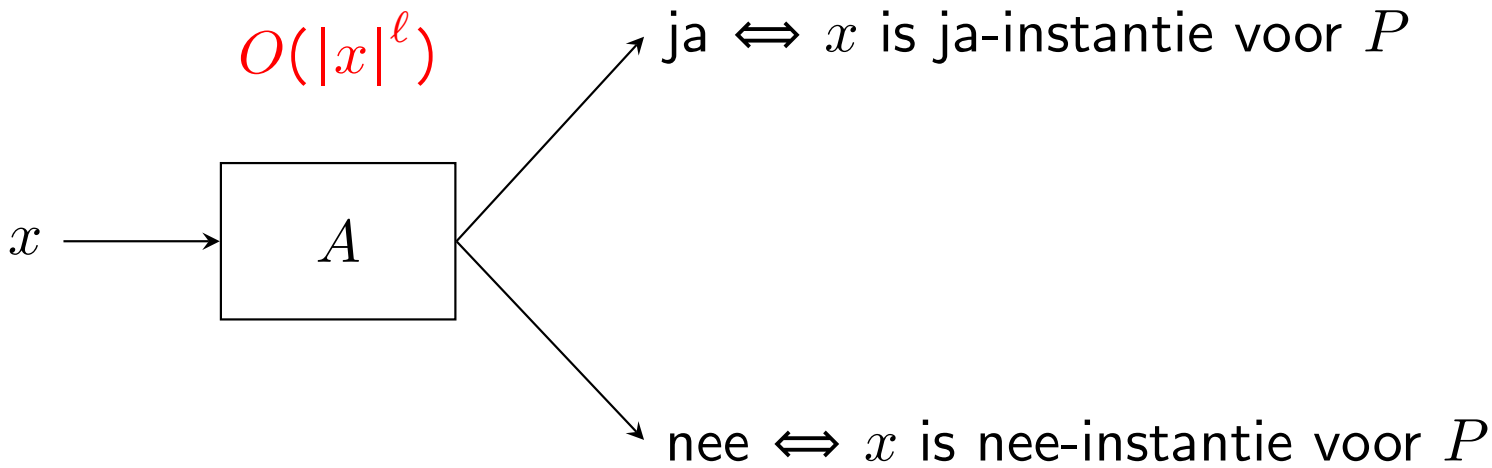
**Definitie**
Een algoritme heet polynomiaal begrensd als zijn worst case-complexiteit van boven is begrensd door een functie die polynomiaal is in de lengte van de invoer. Dus: worst case is $O(p(n))$ voor een zeker polynoom $p$, en met $n$ (een maat voor) de lengte van de invoer.
Eenvoudiger geformuleerd: worst case is $O(|x|^{\ell})$ met $|x|$ de lengte van de invoer $x$ en $\ell \geq 0$.

**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 $\mathcal{P}$.

# $P \in \mathcal{P}$

Gegeven een willekeurig beslissingsprobleem $P \in \mathcal{P}$. Dan is er een polynomiaal deterministisch algoritme $A$ dat $P$ oplost voor elke invoer $x$.




Een algoritme is deterministisch als het elke keer dat het wordt uitgevoerd op dezelfde invoer (instantie) hetzelfde doet, en dus dezelfde uitvoer oplevert.

# Handelbaar $\leftrightarrow \mathcal{P}$

**Vraag**: waarom handelbaar = polynomiaal begrensd?

- als een probleem niet in $\mathcal{P}$ zit, is het zeker onhandelbaar
- de klasse $\mathcal{P}$ heeft mooie afsluitingseigenschappen: een algoritme dat bestaat uit een eindige opeenvolging en/of samenstelling van polynomiaal begrensde algoritmen is zelf ook weer polynomiaal begrensd (zie de opgaven)
- de klasse $\mathcal{P}$ is onafhankelijk van het berekeningsmodel en van gebruikte coderingen: als een probleem polynomiaal is begrensd in het ene model, dan ook in een ander model (voor alle redelijke/praktische berekeningsmodellen)

# $\mathcal{EXP}$ en $\mathcal{NPC}$

Het blijkt buitengewoon moeilijk te zijn om te bewijzen dat voor een probleem ieder algoritme complexiteit $\Omega(2^{p(n)})$ heeft (dus minstens exponentieel is in $n$).

Er zijn maar relatief weinig niet-triviale problemen waarvoor zo'n ondergrens is bewezen. Bijvoorbeeld: gegeneraliseerd schaken op een $n \times n$ bord (is er een winnende strategie?), en een aantal problemen uit de formele talen en logica.

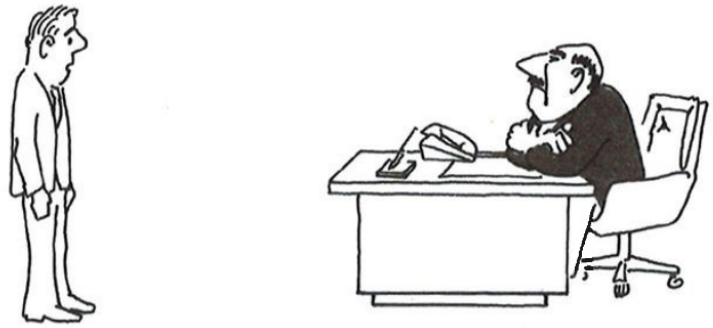
Er zijn echter heel veel problemen waarvoor we geen polynomiaal algoritme hebben, maar ook geen exponentiële ondergrens $\Rightarrow \mathcal{NPC}$.

# $\mathcal{NPC}$

De klasse van NP-volledige problemen $\mathcal{NPC}$ (Engels: NP-complete) heeft enkele interessante eigenschappen, zoals:

1. voor geen enkel NP-volledig probleem is tot dusver een polynomiaal algoritme gevonden. Men vermoedt dus dat ze onhandelbaar zijn, maar dat heeft tot dusver ook nog niemand kunnen bewijzen.
2. als er een polynomiaal algoritme bestaat voor ook maar één NP-volledig probleem, dan is meteen elk NP-volledig probleem in polynomiale tijd oplosbaar.
3. omgekeerd: als er van één enkel NP-volledig probleem wordt bewezen dat het onhandelbaar is, dan zijn alle NP-volledige problemen onhandelbaar.

:)

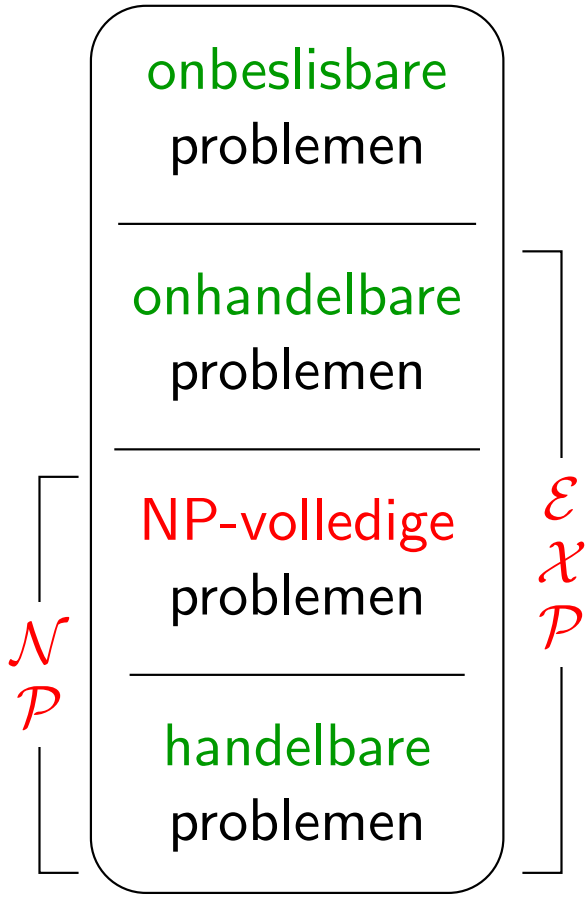


"I can't find an efficient algorithm, I guess I'm just too dumb."

"I can't find an efficient algorithm, but neither can all these famous people."

M.R. Garey en D.S. Johnson, Computers and intractability, Freeman, 1979

# Uitgebreidere indeling




Problemen waarvoor geen algoritme mogelijk is (stop/halting-probleem)

Problemen waarvoor geen **deterministisch** polynomiaal algoritme mogelijk is

**???**

Problemen waarvoor een **deterministisch** polynomiaal algoritme bestaat: $\mathcal{P}$

$\mathcal{NP}$ is de klasse van problemen waarvoor een niet-deterministisch polynomiaal algoritme bestaat. We zullen dit allemaal nog preciezer maken.

# $\mathcal{NP}$ informeel

$\mathcal{P}$ is de klasse van problemen waarvoor een deterministisch polynomiaal algoritme bestaat.
$\mathcal{NP}$ is de klasse van problemen waarvoor een niet-deterministisch polynomiaal algoritme bestaat (precieze definitie later).

**Niet-deterministisch**: je mag een oplossing gokken.
**Polynomiaal**: deze kan daarna in polynomiale tijd worden gecontroleerd.

$\mathcal{P}$: beslissingsproblemen die polynomiaal oplosbaar zijn.
$\mathcal{NP}$: beslissingsproblemen die polynomiaal verifieerbaar zijn (althans, de ja-instanties; laat je niet foppen).
$\mathcal{NPC}$: bevat de moeilijkste problemen uit $\mathcal{NP}$.

M.a.w.: voor $\mathcal{P}$ is oplossing in polynomiale tijd te *vinden*, voor $\mathcal{NP}$ is oplossing in polynomiale tijd te *controleren*.

# $\mathcal{P}$, $\mathcal{NP}$ en $\mathcal{NPC}$

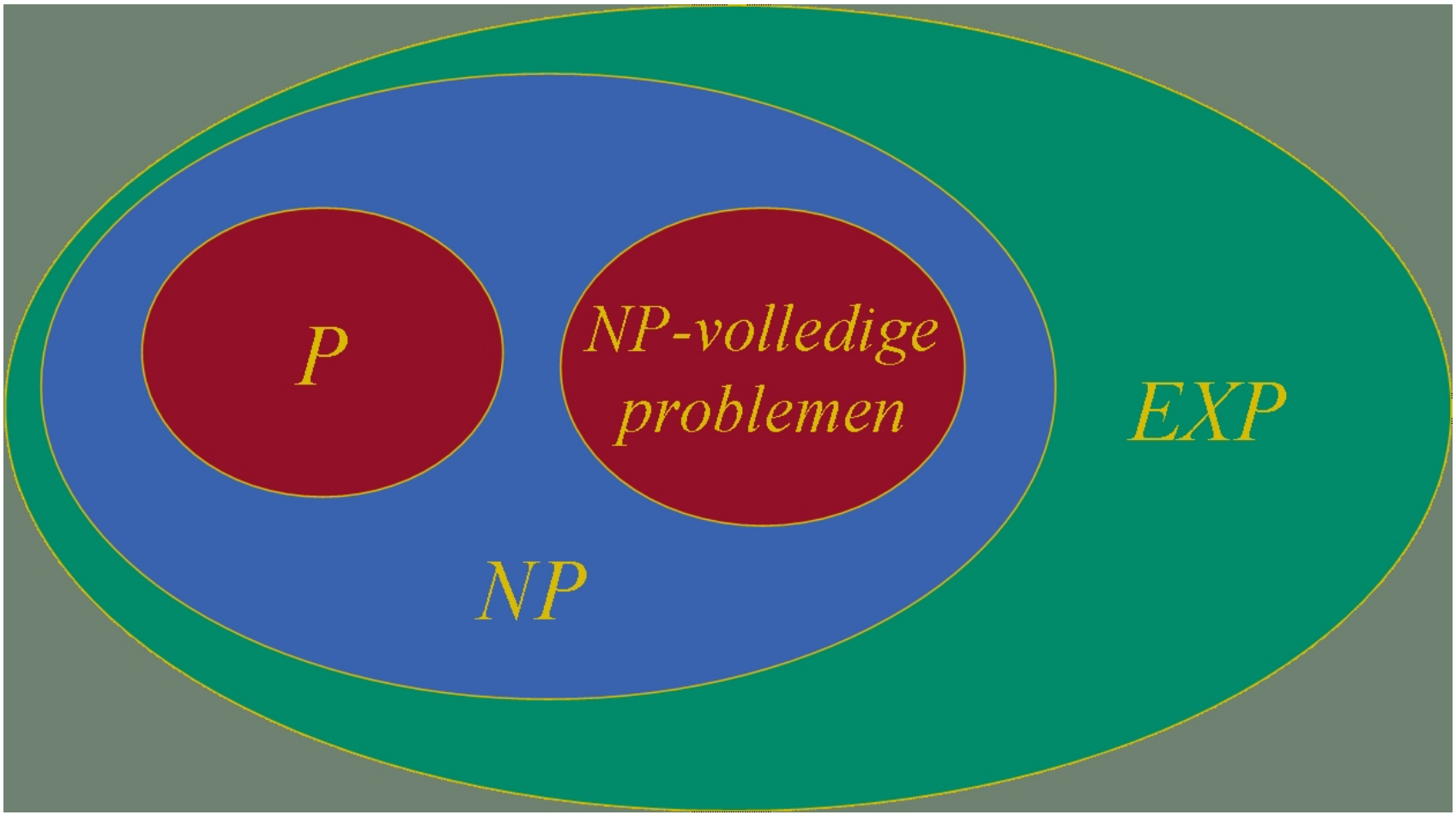
We weten dat $\mathcal{P} \subseteq \mathcal{NP}$ en $\mathcal{NPC} \subseteq \mathcal{NP}$. In theorie kan het zijn dat $\mathcal{P} = \mathcal{NP}$, maar dat is onwaarschijnlijk (zie later).

Wat niet kan, is dat $\mathcal{P} \neq \mathcal{NP}$ en $\mathcal{P} \cup \mathcal{NPC} = \mathcal{NP}$. Met andere woorden, als $\mathcal{P}$ ongelijk is aan $\mathcal{NP}$, dan zijn er problemen in $\mathcal{NP}$ die *noch* in $\mathcal{P}$ zitten, noch in $\mathcal{NPC}$ (Ladner, 1975). Deze problemen worden ook wel $\mathcal{NPI}$ (NP-intermediate) genoemd.

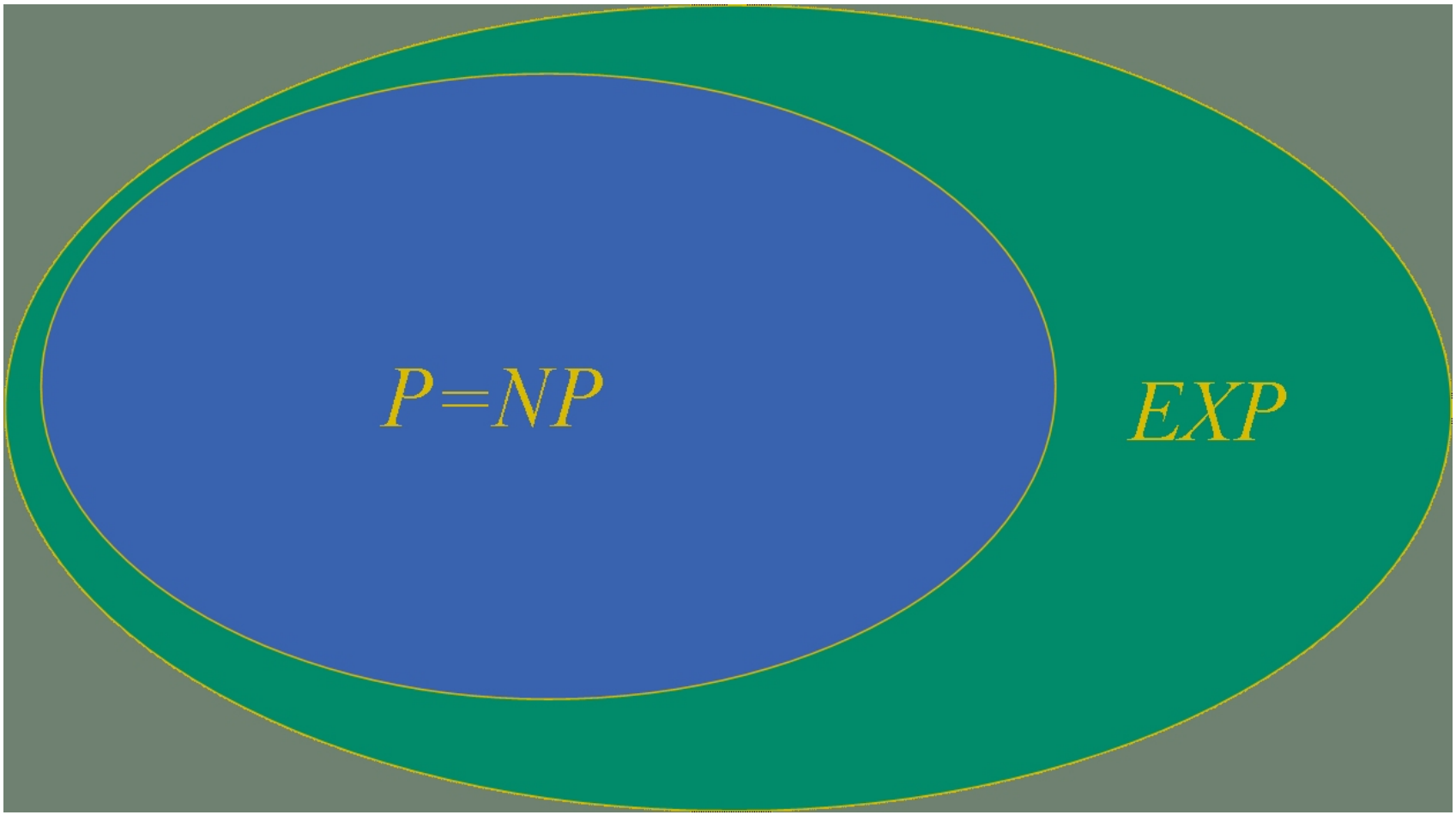
Het bestaan van $\mathcal{NPI}$-problemen is hypothetisch. Ze bestaan onder de aanname dat $\mathcal{P} \neq \mathcal{NP}$, dat weten we zeker, maar we hebben nog geen probleem kunnen identificeren. Er zijn wel enkele veelbelovende kandidaten, zoals het graafisomorfismeprobleem*.

*László Babai (2015–2017): oplosbaar in $O(2^{p(\lg n)})$, quasi-polynomiaal

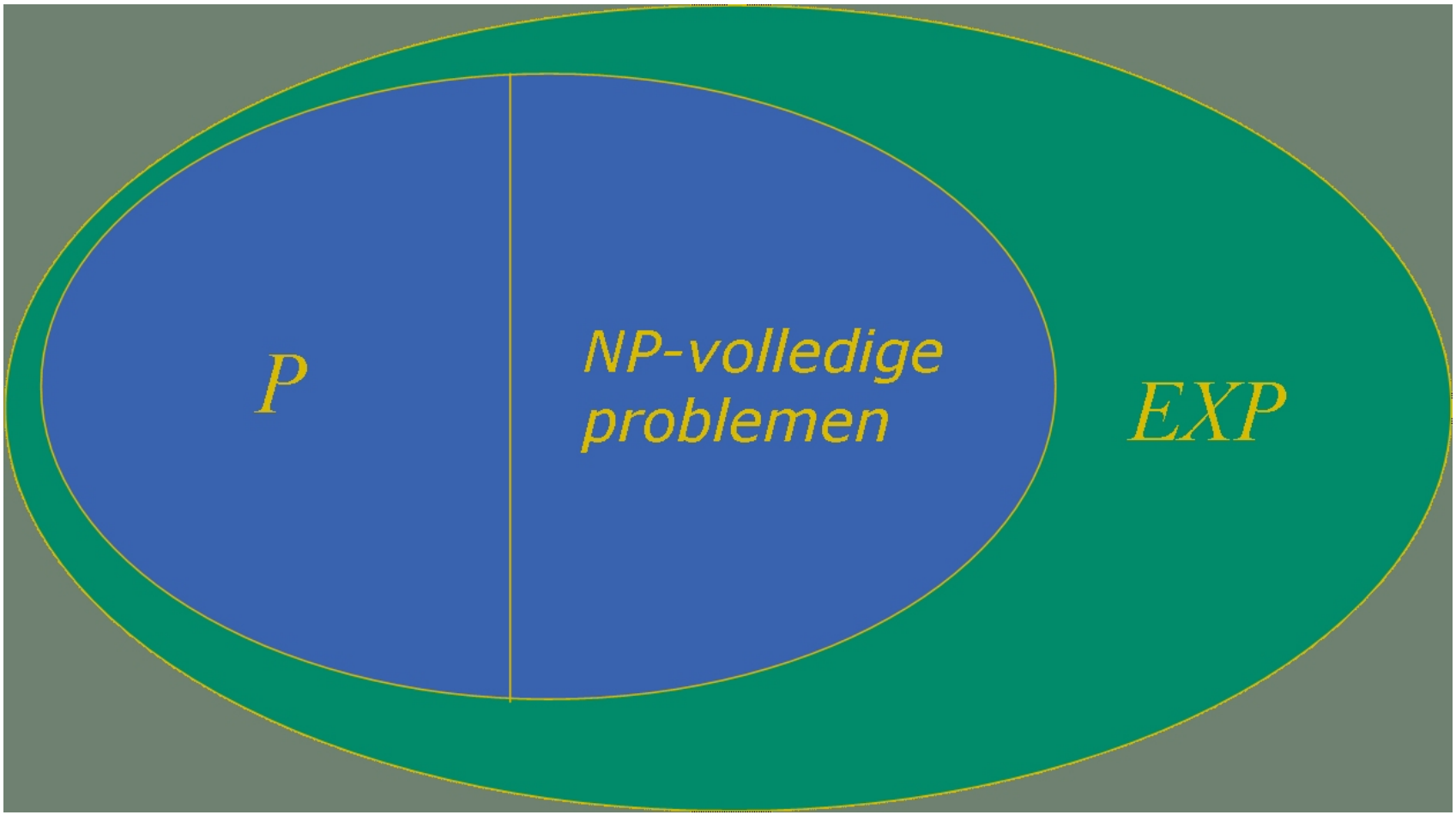
# De meest waarschijnlijke relatie



# Onwaarschijnlijk, maar mogelijk



# Elegant, maar onmogelijk



# Beslissingsproblemen

De theorie van NP-volledigheid beperkt zich tot beslissingsproblemen. Bij een beslissingsprobleem zijn slechts twee antwoorden mogelijk: ja of nee.

Probleeminvoer waarop het antwoord *ja* is noemen we ja-instanties; als het antwoord *nee* is spreken we van nee-instanties.

Optimalisatieproblemen worden omgezet naar beslissingsproblemen, en wel zo dat geldt: als het optimalisatieprobleem handelbaar is, dan is het corresponderende beslissingsprobleem dat ook. En dus ook omgekeerd: **als het beslissingsprobleem onhandelbaar is, dan is het corresponderende optimalisatieprobleem dat ook**.

# Berekeningsmodel

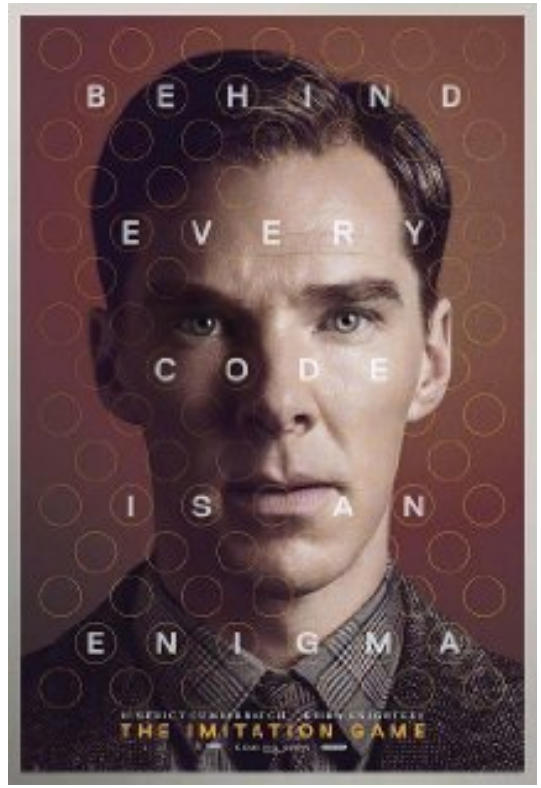
De klassen $\mathcal{P}$ en $\mathcal{NP}$ worden formeel gedefinieerd met behulp van Turing machines. Een Turing machine is een uiterst simpel maar krachtig model van een 'computer'.

Elk probleem dat met een 'normaal' computerprogramma in polynomiale tijd is op te lossen, blijkt ook polynomiaal oplosbaar te zijn met behulp van een Turing machine, en omgekeerd.

We mogen dus gewoon C++-achtige algoritmen blijven gebruiken, maar het is nuttig om iets te weten over de werking van Turing machines. Zie ook het vak Computability.

# Alan Turing

1912–1954

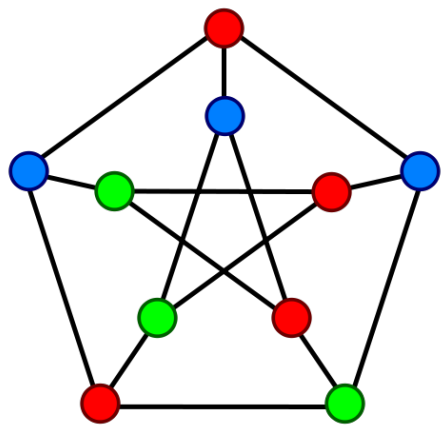



2015

# Voorbeelden $\mathcal{NPC}$-problemen

Van honderden problemen is bekend dat ze NP-volledig zijn. We bekijken hier zes zeer bekende $\mathcal{NPC}$-problemen:

- CNF-satisfiability
- Subset Sum
- Hamiltonkring
- Handelsreizigersprobleem
- Kliek
- Graafkleuring



# CNF-satisfiability

- een literal is een Boolese variabele of de negatie daarvan (dus $x$ of $\neg x$).
- een clause (clausule) is een disjunctie ($\vee$) van literals.
  Voorbeeld: $x_1 \vee \neg x_3 \vee x_4 \vee \neg x_6$.
- een logische formule $\phi$ staat in CNF (conjunctieve normaalvorm) als hij bestaat uit een conjunctie ($\wedge$) van clausules.
  Voorbeeld: $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge \neg x_1$.
- een waardering (waarheidstoekenning) van een verzameling logische variabelen $\{x_1, x_2, \ldots, x_n\}$ is een toekenning van de waarde True of False aan elk van de logische variabelen uit die verzameling.

# SAT

## Beslissingsprobleem SAT

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

**Voorbeeld**

De waardering $w$ met $w(x_1)$ = False, $w(x_2)$ = True en $w(x_3)$ = False maakt de logische formule

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge \neg x_1$$

waar. Deze $\phi$ is dus een ja-instantie voor SAT.

# Subset Sum

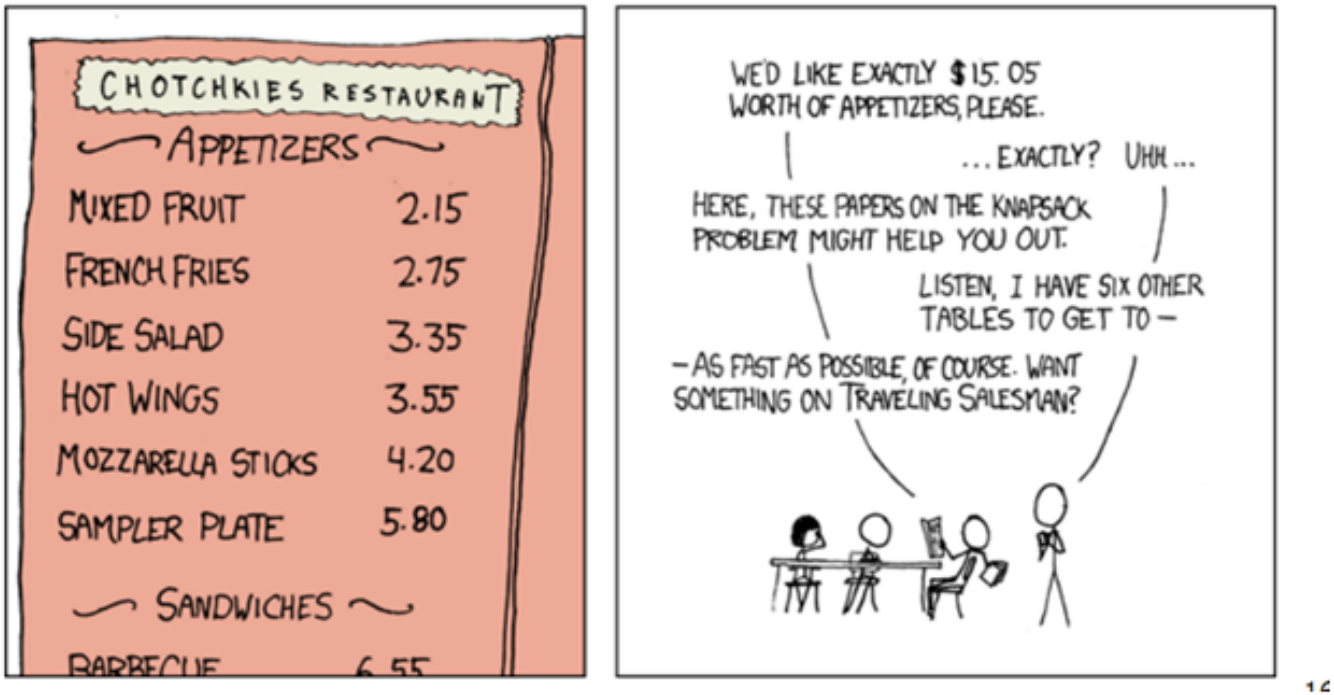
## Beslissingsprobleem SUM

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

**Voorbeeld 1**
Neem $S = \{1, 7, 28, 3, 2, 5, 9, 32, 41, 11, 8\}$ en $t = 30$, dan is dit een ja-instantie voor het probleem. Immers voldoet $S' = \{7, 3, 9, 11\}$.

**Voorbeeld 2**
Neem $S = \{1, 4, 16, 64, 256, 1040, 1093, 1284, 1344\}$ en $t = 3754$, dan is dit een ja-instantie voor het probleem. Immers voldoet $S' = \{1, 4, 16, 256, 1040, 1093, 1344\}$.

:)




https://xkcd.com/287/

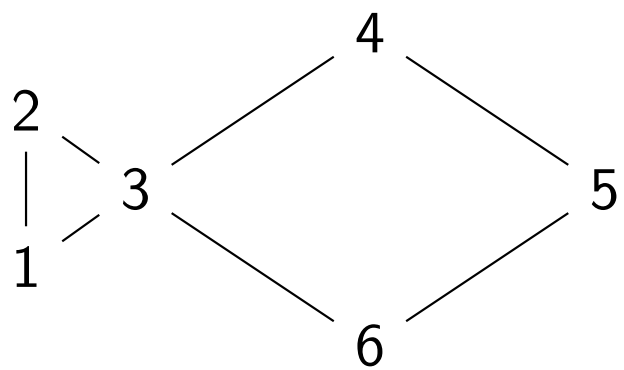
# Hamiltonkring

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

Beslissingsprobleem HC

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

**Voorbeeld**




is een nee-instantie voor HC.

# Handelsreizigersprobleem

Handelsreizigersprobleem of Travelling Salesperson Problem (TSP)

## Optimalisatieprobleem

Gegeven een volledige*, ongerichte graaf $\mathcal{G} = (V, E)$ met gewichten op de takken. Geef een Hamiltonkring in $\mathcal{G}$ met minimaal totaalgewicht.

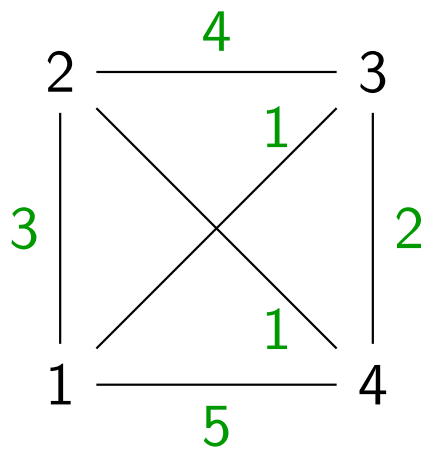
## Beslissingsprobleem TSP

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 $\leq k$?

*tussen elk tweetal knopen van $\mathcal{G}$ zit een tak

# TSP: voorbeeld

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



# 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 $\mathcal{G} = (V, E)$. Geef een kliek met zo veel mogelijk knopen (een maximale kliek).

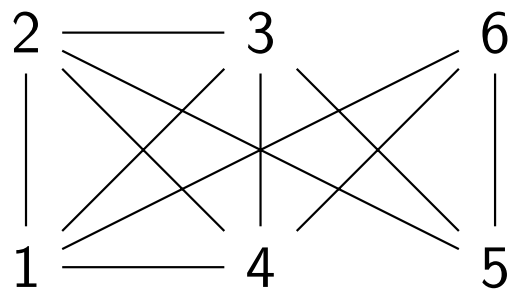
## Beslissingsprobleem Kliek

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 $k$?

# Kliek: vervolg

**Opmerking**
Het is equivalent om te vragen naar een kliek ter grootte gelijk aan $k$. Immers: elke deelverzameling van een kliek is zelf weer een kliek.

**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: $\{1, 2, 3, 4\}$.



# Graafkleuring

Een kleuring van (de knopen van) een ongerichte graaf $\mathcal{G} = (V, E)$ is een afbeelding $c : V \mapsto S$, waarin $S$ een eindige verzameling (van kleuren) is, met de eigenschap 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 $\mathcal{G} = (V, E)$. Vind een kleuring van $\mathcal{G}$ met zo weinig mogelijk kleuren.

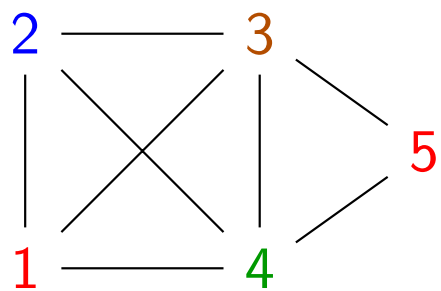
## Beslissingsprobleem Kleur

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)?

# Kleur: vervolg

**Opmerking**
Het is equivalent om te vragen naar een kleuring met precies $k$ kleuren: ze hoeven niet allemaal te worden gebruikt.

**Voorbeeld**
Onderstaande graaf kan worden gekleurd met 4 kleuren, maar niet met minder dan 4 (waarom niet)?
Kleur bijvoorbeeld 1 en 5 rood, 2 blauw, 3 bruin en 4 groen.



# Gemeenschappelijk

1. voor alle zes voorbeeldproblemen is eenvoudig een exponentieel algoritme op te schrijven.
2. in alle gevallen kan in polynomiale tijd worden gecontroleerd of een kandidaatoplossing een echte oplossing (= "dat wat je zoekt") is.
3. voor al deze problemen geldt: het lijkt extreem moeilijk (exponentieel) te zijn om voor gegeven invoer $x$ te bepalen of het antwoord "ja" of "nee" moet zijn.
4. echter, *als* $x$ een ja-instantie is, dan is er een eenvoudige (polynomiale) manier om iemand daarvan te overtuigen.

# Gemeenschappelijk

5. voor ja-instanties bestaat er een zogenaamd certificaat dat kan worden gebruikt om te laten zien dat het antwoord inderdaad "ja" is. In de voorbeelden is dit steeds de gezochte oplossing, d.w.z. dat wat de invoer zou moeten bezitten.
6. bovendien is dit certificaat kort (polynomiaal) en kan verifiëren / controleren ervan in polynomiale tijd.
7. eigenschap 2 t/m 6: de genoemde problemen zitten in $\mathcal{NP}$. Later wordt alles formeler gemaakt.
8. ja-instanties zijn eenvoudig te verifiëren met de juiste hint (certificaat). Hoe zit het met nee-instanties?

# Om te onthouden van vandaag

- handelbare en onhandelbare problemen: wel/geen *deterministisch* polynomiaal algoritme
- $\mathcal{P}$: oplossing in polynomiale tijd te *vinden*, handelbaar; $\mathcal{EXP} \setminus \mathcal{P}$: oplossing niet in polynomiale tijd te vinden; onhandelbaar
- $\mathcal{NP}$: *niet-deterministisch* polynomiaal algoritme; oplossing in polynomiale tijd te *controleren*
- $\mathcal{NPC}$: moeilijkste problemen in $\mathcal{NP}$; aantal welbekende voorbeelden; handelbaarheid onbekend
- beslissingsproblemen en optimalisatieproblemen: zelfde handelbaarheid

# De komende weken

- de klasse $\mathcal{NP}$ formeler gemaakt
- polynomiale reducties en de klasse $\mathcal{NPC}$
- de stelling van Cook, iets over MIP- en SAT-solvers
- benaderingsalgoritmen
- gastcollege Hendrik Jan Hoogeboom: complexiteit en spellen*

*details onder voorbehoud

# (Werk)college

**Volgende college**: dinsdag 11 april, 9u00–10u45, zaal 412 (Snellius)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304 en 303 (Snellius, onderaan de trap)
Opgaven uit het dictaat: 55, 10, 45, 46, 33

# Complexiteit 2023 — college 9

11 april 2023

---

## De klasse $\mathcal{NP}$

# Vorige keer

- handelbare en onhandelbare problemen: wel/geen *deterministisch* polynomiaal algoritme
- $\mathcal{P}$: oplossing in polynomiale tijd te *vinden*, handelbaar; $\mathcal{EXP} \setminus \mathcal{P}$: oplossing niet in polynomiale tijd te vinden; onhandelbaar
- $\mathcal{NP}$: *niet-deterministisch* polynomiaal algoritme; oplossing in polynomiale tijd te *controleren*
- $\mathcal{NPC}$: moeilijkste problemen in $\mathcal{NP}$; aantal welbekende voorbeelden; handelbaarheid onbekend
- beslissingsproblemen en optimalisatieproblemen: zelfde handelbaarheid

# Vandaag

- de klasse $\mathcal{NP}$, hoe deze formeel te definiëren en hoe aan te tonen dat een probleem erin zit
- een voorproefje van reducties en de klasse $\mathcal{NPC}$

# Voorbeeldproblemen

Vorige week besproken:

- CNF-satisfiability (SAT)
  Gegeven een logische formule $\phi$ in CNF. Bestaat er een waardering van de voorkomende variabelen die $\phi$ waar maakt?
- Subset Sum (SUM)
  Gegeven een eindige verzameling $S \subset \mathbb{N}$ en een getal $t \in \mathbb{N}$. Bestaat er een deelverzameling $S' \subseteq S$ zodat $\sum_{s \in S'} s = t$?
- Hamiltonkring (HC)
  Gegeven een graaf $\mathcal{G} = (V, E)$. Heeft $\mathcal{G}$ een Hamiltonkring?
- Handelsreizigersprobleem (TSP)
  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 $\leq k$?

# Voorbeeldproblemen

Vorige week besproken:

- 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 $\geq k$?
  (of equivalent: ter grootte $k$?)
- Graafkleuring (Kleur)
  Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$ en een geheel getal $k > 0$. Bestaat er een kleuring van $\mathcal{G}$ met $\leq k$ kleuren?
  (of equivalent: met $k$ kleuren?)

# Gemeenschappelijk

1. voor alle zes voorbeeldproblemen is eenvoudig een exponentieel algoritme op te schrijven.
2. in alle gevallen kan in polynomiale tijd worden gecontroleerd of een kandidaatoplossing een echte oplossing (= "dat wat je zoekt") is.
3. voor al deze problemen geldt: het lijkt extreem moeilijk (exponentieel) te zijn om voor gegeven invoer $x$ te bepalen of het antwoord "ja" of "nee" moet zijn.
4. echter, *als* $x$ een ja-instantie is, dan is er een eenvoudige (polynomiale) manier om iemand daarvan te overtuigen.

# Gemeenschappelijk

5. voor ja-instanties bestaat er een zogenaamd certificaat dat kan worden gebruikt om te laten zien dat het antwoord inderdaad "ja" is. In de voorbeelden is dit steeds de gezochte oplossing, d.w.z. dat wat de invoer zou moeten bezitten.
6. bovendien is dit certificaat kort (polynomiaal) en kan verifiëren / controleren ervan in polynomiale tijd.
7. eigenschap 2 t/m 6: de genoemde problemen zitten in $\mathcal{NP}$. Later wordt alles formeler gemaakt.
8. ja-instanties zijn eenvoudig te verifiëren met de juiste hint (certificaat). Hoe zit het met nee-instanties?
9. Er zijn ook makkelijke instanties / ingeperkte versies van het probleem.

# $\mathcal{NP}$ informeel

$\mathcal{P}$: beslissingsproblemen die polynomiaal oplosbaar zijn.

$\mathcal{NP}$: beslissingsproblemen die polynomiaal verifieerbaar zijn,

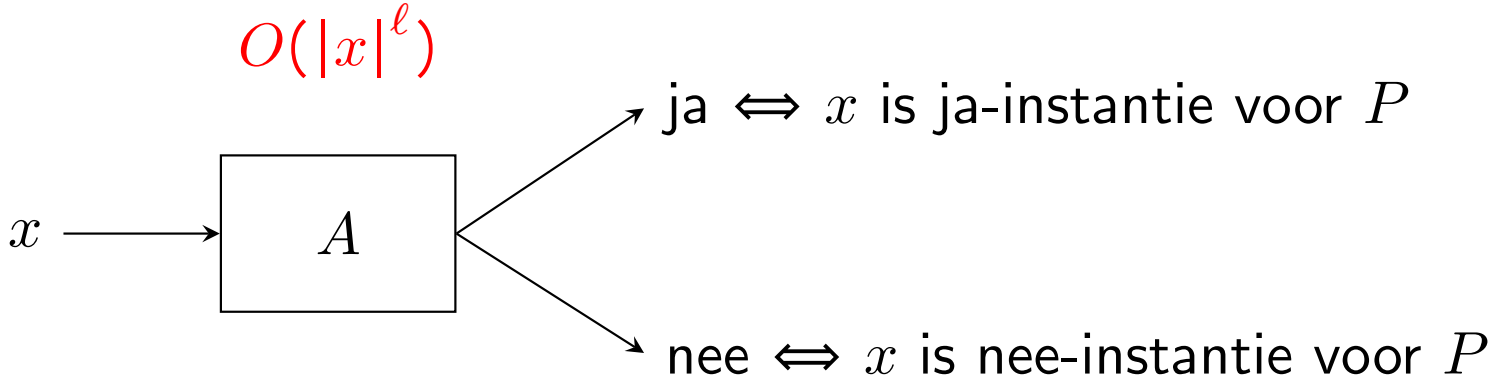
- althans, de ja-instanties
- en met de juiste hint/certificaat
- voor ja-instanties bestaat er altijd zo'n certificaat

$\mathcal{NPC}$: bevat de moeilijkste problemen uit $\mathcal{NP}$.

Voor problemen in $\mathcal{NP}$ duurt het *vinden* van een oplossing lang (tenzij ze in $\mathcal{P}$ zitten); het *verifiëren/controleren* kan echter in polynomiale tijd.

# $P \in \mathcal{P}$

Gegeven een willekeurig beslissingsprobleem $P \in \mathcal{P}$. Dan is er een polynomiaal deterministisch algoritme $A$ dat $P$ oplost voor elke invoer $x$.




Het algoritme is polynomiaal in de lengte van de invoer, dus worst case is $O(|x|^{\ell})$ ($\ell \geq 0$). Een algoritme is deterministisch als het elke keer dat het wordt uitgevoerd op dezelfde invoer (instantie) hetzelfde doet, en dus dezelfde uitvoer oplevert.

# Certificaat

NP-problemen: de *ja-instanties* zijn eenvoudig (polynomiaal) te verifiëren met behulp van het juiste certificaat, de nee-instanties niet.

Veel beslissingsproblemen zijn geformuleerd als "is-er-een"-vragen. In dat geval kun je bij een certificaat denken aan een oplossing voor het probleem, dus datgene wat wordt gezocht en wat een ja-instantie een ja-instantie maakt.

Daarvan moet dan onder andere worden gecontroleerd dat deze voldoet aan de criteria van het probleem, dus inderdaad een *echte* oplossing is, en derhalve een ja-antwoord rechtvaardigt.

Als deze verificatie/controle in polynomiale tijd kan, zit het probleem in $\mathcal{NP}$.

# Voorbeeld

Gegeven een logische formule $\phi$ in CNF, bestaat er een waardering die $\phi$ waarmaakt?

In geval van een ja-instantie kunnen we als certificaat een waarmakende waardering nemen. Er kan nu in polynomiale tijd worden geverifieerd dat deze waardering $\phi$ inderdaad waarmaakt, en dus dat $\phi$ een ja-instantie is.
Probleem SAT zit derhalve in $\mathcal{NP}$ (en overigens ook in $\mathcal{NPC}$).

Ook voor de andere vijf voorbeeldproblemen geldt: als $x$ een ja-instantie is voor het probleem, dan bestaat er een certificaat waarmee je dat eenvoudig (polynomiaal) kunt aantonen.

# Certificaten

| Probleem | Invoer | Certificaat |
|---|---|---|
| HC | $\langle \mathcal{G} \rangle$ | Hamiltonkring |
| SAT | $\langle \phi \rangle$ | waarmakende waardering |
| Kliek | $\langle \mathcal{G}, k \rangle$ | kliek met ($\geq$) $k$ knopen |
| Kleur | $\langle \mathcal{G}, k \rangle$ | kleuring met ($\leq$) $k$ kleuren |
| Sum | $\langle S, t \rangle$ | deelverzameling met som $t$ |
| TSP | $\langle \mathcal{G}, k \rangle$ | Hamiltonkring met totaalgewicht $\leq k$ |

# Lineair?

1. Invoer: een array $A$ met $n > 0$ gehele getallen
   Algoritme:

```
1 for i := 1 to n do
2 |   print A[i];
3 od
```

   Complexiteit: $\Theta(n)$

2. Invoer: een geheel getal $n > 0$
   Algoritme:

```
1 for i := 1 to n do
2 |   print '1';
3 od
```

   Complexiteit: $\Theta(n) = \Theta(2^{\lg n})$

# En nu nog even dit...

**Vraag**: wat is eigenlijk de lengte van de invoer?

**Voorbeeldprobleem**

Gegeven een geheel getal $n > 1$. Heeft $n$ echte delers, m.a.w., is $n = a \times b$ voor zekere $a, b > 1$?

**Algoritme**:

```
// gewoon alle mogelijke delers proberen
1 gevonden := False;
2 m := 2;
3 while not gevonden and m < n do
4 |   if n mod m = 0 then
5 |   |   gevonden := True;
6 |   else
7 |   |   m := m + 1;
8 |   fi
9 od
// m is nu de kleinste deler > 1 van n
```

# Lengte invoer

De worst case-complexiteit van dit algoritme is $\Theta(n)$ (indien we de berekening van $n \bmod m$ tellen als één stap, 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 $\ell$-aire codering met $\ell > 1$, dan is het algoritme exponentieel in de lengte van de invoer (voor elke $\ell > 1$ dus).

**Vraag**: is het algoritme polynomiaal of exponentieel?

# Primes is in $\mathcal{P}$

In 2002 is bewezen door Manindra Agrawal en zijn PhD-studenten Neeraj Kayal en Nitin Saxena dat het probleem in $\mathcal{P}$ zit. De zogenaamde AKS-test is een priemgetaltest die polynomiaal is in het aantal cijfers (dus de lengte) van $n$.

In 2006 ontvingen zij hiervoor de Fulkerson Prize (discrete wiskunde) en de Gödel Prize (theoretische informatica).

# Ander voorbeeld

## Knapzakprobleem

**Gegeven** een knapzak met capaciteit $S$ (een geheel getal $> 0$) en $n$ objecten met gewichten $s_1, s_2, \ldots, s_n$ en met waarde $w_1, w_2, \ldots, w_n$. (Alle $s_i$ en $w_i$ zijn geheel en $> 0$.)
**Gevraagd** een deelverzameling van de objecten met totaalgewicht $\leq S$ en maximale totaalwaarde.

Het knapzakprobleem kan worden opgelost met een algoritme met complexiteit $O(n \times S)$ (dynamisch programmeren, zie Algoritmiek).

Dit is niet polynomiaal, maar exponentieel als functie van de lengte van de invoer!

# $\mathcal{NP}$ iets formeler

$\mathcal{NP}$ is de klasse van beslissingsproblemen waarvoor een niet-deterministisch polynomiaal algoritme bestaat:

- niet-deterministisch:
  je mag een oplossing gokken → certificaat
- polynomiaal:
  deze kan daarna in polynomiale tijd worden geverifieerd (gecontroleerd)
- ja-instantie:
  voor een ja-instantie bestaat er een oplossing, en dus een certificaat waarmee je aantoont dat het inderdaad een ja-instantie is

# Niet-deterministisch: parallel met Automata Theory

Voor een niet-deterministische eindige automaat (NFA) $\mathcal{A}$ geldt: een woord $w$ zit in de taal van $\mathcal{A}$ d.e.s.d.a. er minstens één executie van $\mathcal{A}$ bestaat waarin $w$ wordt geaccepteerd.

M.a.w.:

- er mogen ook executies bestaan waarin $w$ wordt verworpen.
- een woord zit *niet* in de taal van $\mathcal{A}$ d.e.s.d.a. er *geen enkele* executie van $\mathcal{A}$ bestaat waarin $w$ wordt geaccepteerd.

Hetzelfde geldt voor niet-deterministische algoritmen:

- voor ja-instanties moet minstens één geldige executie bestaan die een oplossing vindt, maar er mogen ook allerlei ongeldige executies bestaan.
- voor nee-instanties mag geen enkele geldige executie bestaan.

# Niet-deterministisch algoritme

We kunnen een niet-deterministisch algoritme omschrijven als bestaande uit drie fases: gokken, verifiëren en uitvoer. We gebruiken een dergelijke omschrijving in de (precieze) definitie van $\mathcal{NP}$.

We bekijken de drie fasen in detail:

1. Niet-deterministische gokfase
   Er wordt een willekeurige string $s$ in het geheugen geschreven. Elke keer dat het algoritme executeert kan dit een andere string zijn.

   Deze string is vaak een soort gok van de oplossing van het probleem: het beoogde certificaat; $s$ kan echter ook een onzinstring blijken te zijn.

# Niet-deterministisch algoritme

2. Deterministisch verificatiefase
   Zowel de invoer $x$ van het probleem als de string $s$ mogen hier worden gebruikt. Er wordt True of False geretourneerd, of het programma stopt nooit (het kan bijvoorbeeld belanden in een oneindige loop).

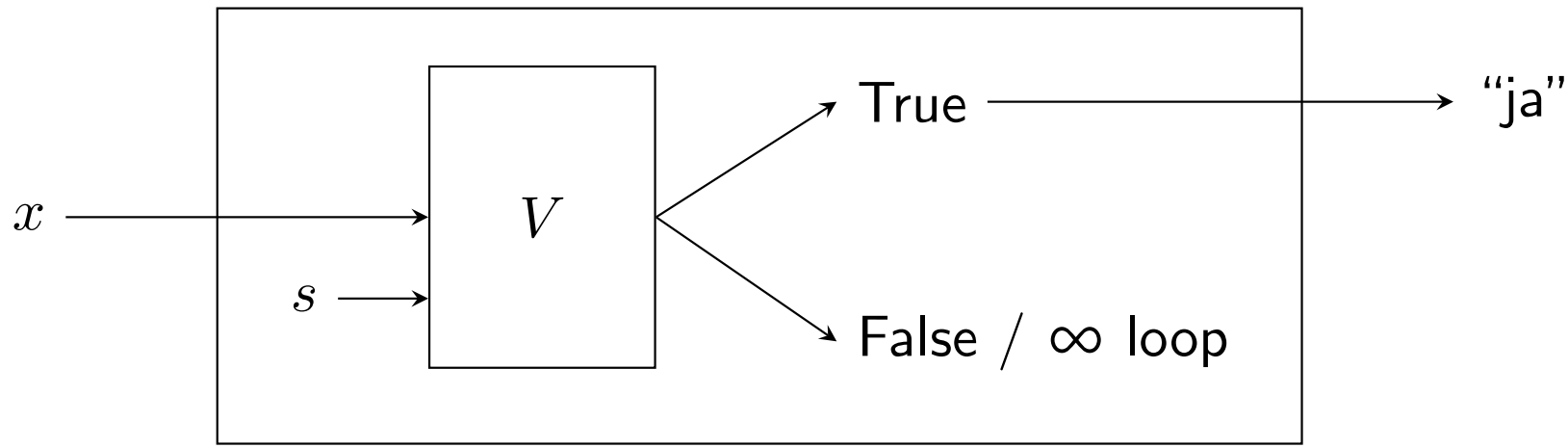
   Hier wordt gecontroleerd of $s$ een oplossing is van het probleem bij de gegeven invoer $x$, m.a.w. er wordt gecontroleerd of $s$ een ja-antwoord rechtvaardigt.

# Niet-deterministisch algoritme

3. Uitvoerfase
   Als fase 2 (verificatie) 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 dat in de gokfase nodig is om $s$ te schrijven (dus het aantal karakters waaruit $s$ bestaat) plus het aantal stappen dat wordt gedaan in de verificatiefase (plus één stap voor de uitvoerfase).

# Schematisch: een executie




fase 1: $s$ wordt gegenereerd (niet-deterministisch)

fase 2: verificatiefase (deterministisch)

fase 3: uitvoerstap

# Vragen

Bij een niet-deterministisch algoritme zijn verschillende executies mogelijk voor dezelfde invoer $x$, afhankelijk van de gegokte $s$. Dus:

(a) Wat is *het* antwoord (ja/nee) van het algoritme voor invoer $x$?

(b) Wanneer noemen ze zo'n algoritme polynomiaal?

# Antwoord

**Definitie**

Het antwoord van een niet-deterministisch algoritme $A$ voor invoer $x$ is "ja" $\iff$ er is een executie* van $A$ die "ja" geeft als uitvoer $\iff$ er is een string s waarvoor fase 2 True oplevert. 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 zo'n niet-deterministisch algoritme $A$ voor invoer $x$ is "ja"† $\iff$ er bestaat een string $s$ (certificaat) waarmee je kunt aantonen (in fase 2) dat $x$ een ja-instantie is.

*dus een string $s$

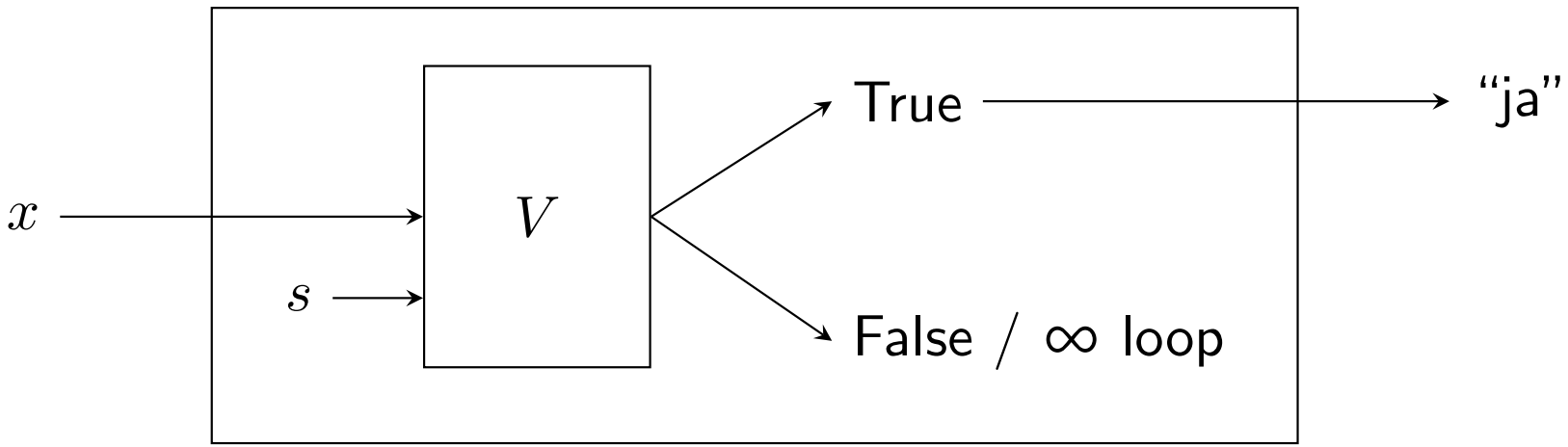
†ofwel: $x$ is een ja-instantie

# De klasse $\mathcal{NP}$

- een niet-deterministisch algoritme heet polynomiaal begrensd als voor elke invoer $x$ waarvoor het antwoord "ja" is, er een executie is van het algoritme die "ja" oplevert in hooguit $O(|x|^{\ell})$ stappen (voor zekere $\ell \geq 0$).
  Dientengevolge mag in dat geval de string $s$ niet te lang zijn (polynomiaal in $|x|$), en het verificatiealgoritme uit fase 2 moet polynomiaal zijn begrensd in $|x|$ (en $|s|$).
- $\mathcal{NP}$ is nu de klasse van beslissingsproblemen waarvoor er een polynomiaal begrensd niet-deterministisch algoritme bestaat.
- $\mathcal{NP}$ betekent: $\mathcal{N}$on-deterministisc $\mathcal{P}$olynomial time

# $P \in \mathcal{NP}$

Niet-deterministisch en polynomiaal:




voor zekere ja-executie:

fase 1: $O(|s|) \subseteq O(|x|^{\ell})$

fase 2: $O(|s|^{p} \times |x|^{q}) \subseteq O(|x|^{r})$

fase 3: $O(1)$

# Voorbeeldproblemen

**Stelling**. Alle zes voorbeeldproblemen zitten in $\mathcal{NP}$.

**Voorbeeld 1**: Kleur
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)?

**Voorbeeld 2**: HC
Gegeven een (gerichte of ongerichte) graaf $\mathcal{G} = (V, E)$. Heeft deze graaf een Hamiltonkring?
Laat hierna $V = \{1, 2, \ldots, n\}$ en dus $|V| = n$.

**Voorbeeld 3**: SAT
Gegeven een logische formule $\phi$ in CNF. Bestaat er een waardering van de in $\phi$ voorkomende logische variabelen die $\phi$ waar maakt?

# Kleur $\in \mathcal{NP}$

Laat $V = \{1, 2, \ldots, n\}$ en de mogelijke kleuren $1, 2, \ldots, k$. Een polynomiaal begrensd *niet-deterministisch algoritme* $A$ voor Kleur, met als invoer $x = \langle \mathcal{G}, k \rangle$, is:

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 goede kleuring voorstelt ($s_i$ wordt geïnterpreteerd als de kleur van knoop $i$):

   (1) controleer of er precies $n = |V|$ integers staan (elke knoop een kleur): kan polynomiaal, $O(|s|)$

   (2) controleer of elke integer tussen $1$ en $k$ is (er worden $k$ kleuren gebruikt): kan polynomiaal, $O(|s|)$

   (3) controleer of aangrenzende knopen verschillend zijn gekleurd. Takken $(v, w)$ aflopen en in $s$ de kleur van $v$ en $w$ opzoeken en vergelijken: $O(|E| \times |s|) \subseteq O(|\mathcal{G}| \times |s|) \subseteq O(|x| \times |s|)$.

# Kleur $\in \mathcal{NP}$

Als de drie tests positief zijn wordt True geretourneerd. Zodra een test negatief uitvalt wordt False geretourneerd (of wordt een oneindige loop begonnen).

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

**Merk op**. Tests (1) en (2) controleren of $s$ een kleuring van de knopen voorstelt, test (3) controleert of de kleuring correct is.

Er geldt: het antwoord van $A$ op invoer $x = \langle \mathcal{G}, k \rangle$ is "ja" $\Longleftrightarrow$ er bestaat een goede string $s$ waarop fase 2 True oplevert $\Longleftrightarrow$ er bestaat een correcte kleuring van de knopen van $\mathcal{G} \Longleftrightarrow x = \langle \mathcal{G}, k \rangle$ is een ja-instantie voor Kleur.

Verder: voor een ja-executie, dus met $s$ een correcte kleuring, is $|s| \in O(|V|) \subseteq O(|x|)$. Ergo: $A$ is polynomiaal begrensd.

# HC $\in \mathcal{NP}$

Een polynomiaal begrensd *niet-deterministisch algoritme* voor HC (invoer $\mathcal{G}, n = |V|, V = \{1, 2, \ldots, n\}$) (zie ook dictaat):

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: $O(|s|)$
   (2) controleer dat elke integer tussen $1$ en $n$ is: $O(|s|)$
   (3) controleer dat alle knopen uit $s$ verschillen: $O(|s|^2)$*
   (4) controleer dat tussen opeenvolgende knopen uit $s$ een tak zit in de graaf (en tussen de eerste en de laatste): $O(|s| \times |E|) \subseteq O(|s| \times |\mathcal{G}|)$.

*kan ook in bijvoorbeeld $O(n \lg n)$, maar dat maakt nu niet uit

# HC $\in \mathcal{NP}$

Als de vier tests positief zijn wordt True geretourneerd. Zodra een test negatief uitvalt wordt False geretourneerd (of wordt een oneindige loop begonnen).

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

**Merk op**. Tests (1) t/m (3) controleren dat $s$ een rij van $n$ verschillende knopen van $\mathcal{G}$ voorstelt (syntactische controle), test (4) controleert dat het een Hamiltonkring is.

Voor ja-instanties bestaat er een string $s$ die een Hamiltonkring voorstelt, en dus een executie die "ja" oplevert, waarbij $|s| \in O(|\mathcal{G}|)$. Zo'n ja-executie is dus polynomiaal in $|\mathcal{G}|$. Voor nee-instanties bestaat zo'n string $s$ niet.

# SAT $\in \mathcal{NP}$

**SAT**: Gegeven een logische formule $\phi$ in CNF. Bestaat er een waardering van de in $\phi$ voorkomende logische variabelen zodat $\phi$ de waarde True krijgt (dus een waardering die $\phi$ waar maakt)?

Een logische formule $\phi$ staat in CNF (conjunctieve normaalvorm) als hij bestaat uit een conjunctie ($\wedge$) van clausules (disjuncties, $\vee$).

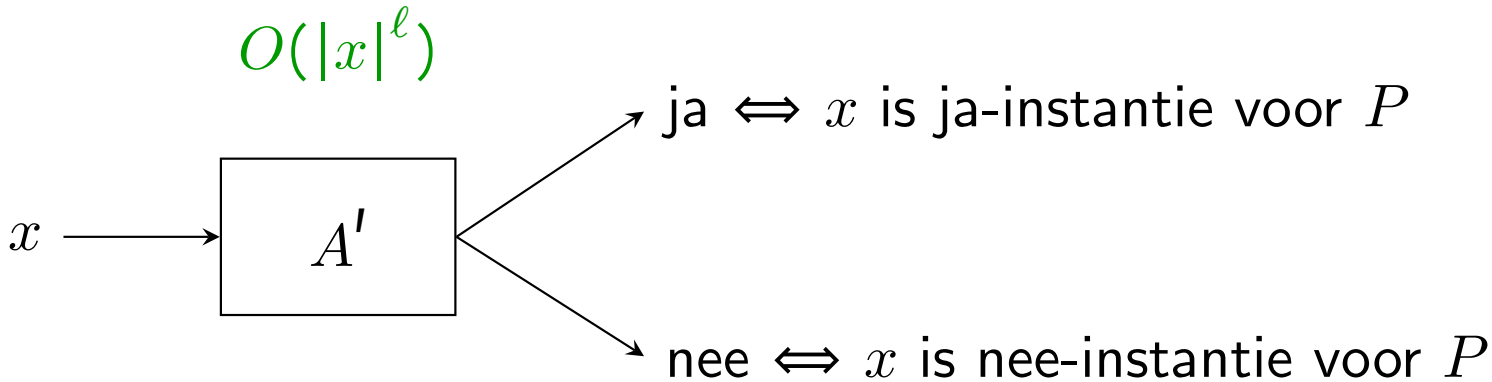
Voorbeeld: $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge \neg x_1$.

**Opgave**
Geef een niet-deterministisch algoritme voor SAT.

# $\mathcal{P} \subseteq \mathcal{NP}$

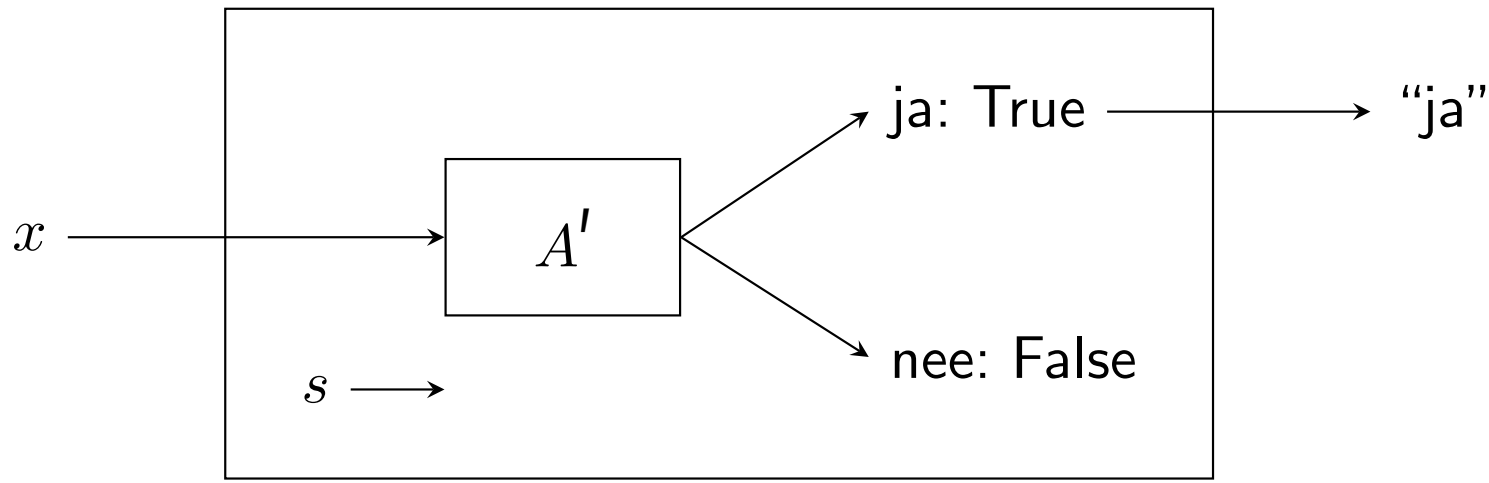
Stelling: $\mathcal{P} \subseteq \mathcal{NP}$

**Bewijs**:
Neem een willekeurig probleem $P \in \mathcal{P}$. Dan is er een polynomiaal deterministisch algoritme $A'$ dat $P$ oplost.




Het volgende niet-deterministische algoritme $A$ is dan een polynomiaal begrensd algoritme voor $P$.

# $\mathcal{P} \subseteq \mathcal{NP}$




fase 1: $s$ wordt gegenereerd; $O(|s|)$

fase 2: negeer $s$ en voer $A'$ uit op $x$; $O(|x|^{\ell})$

fase 3: $O(1)$

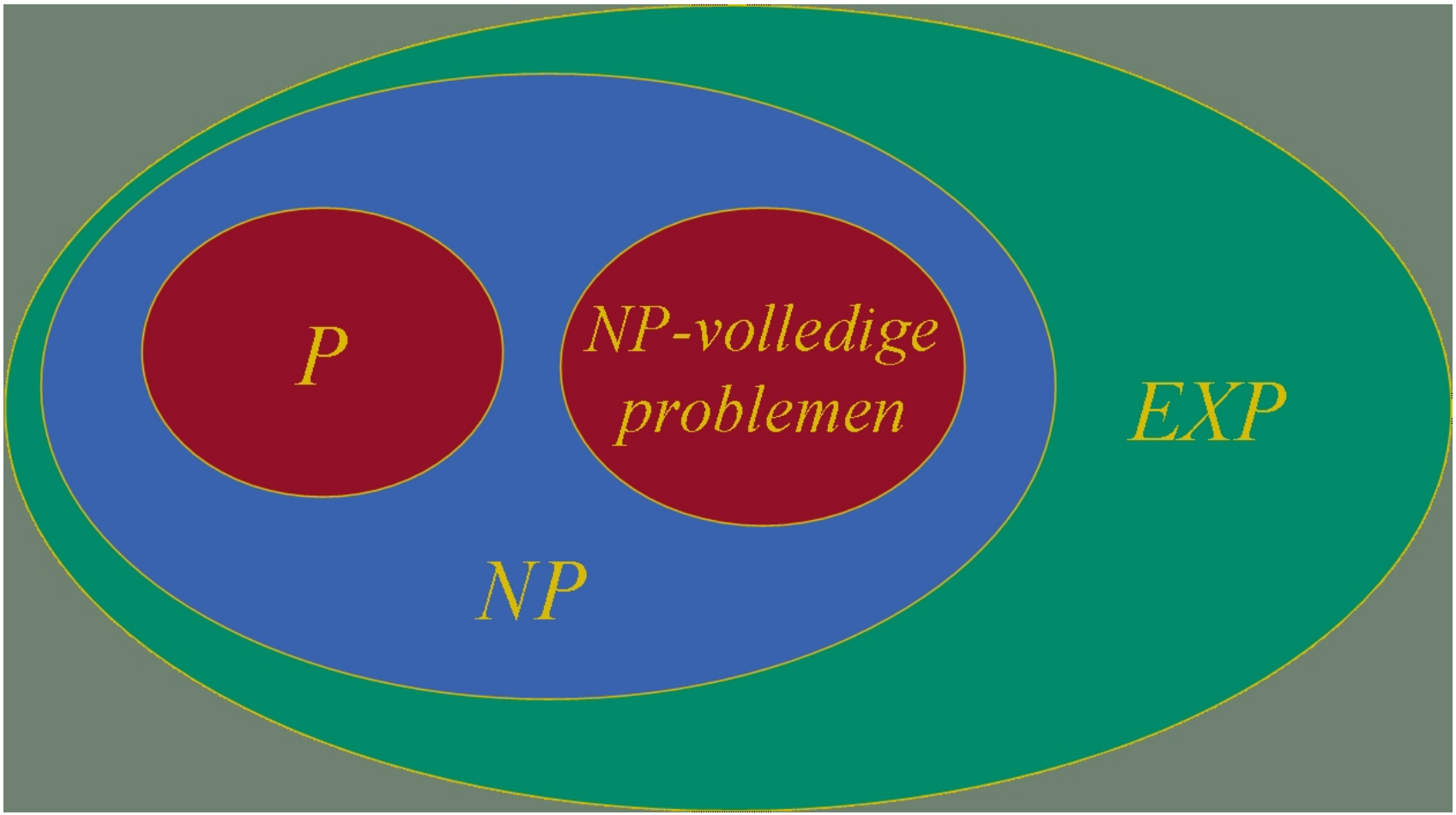
# $\mathcal{NPC}$

De klasse van NP-volledige problemen $\mathcal{NPC}$ (Engels: NP-complete) bevat de *moeilijkste* problemen in $\mathcal{NP}$ en heeft de volgende interessante eigenschap:

1. als er een polynomiaal algoritme bestaat voor ook maar één NP-volledig probleem, dan is meteen elk NP-volledig probleem oplosbaar in polynomiale tijd.*
2. omgekeerd: als er van één enkel NP-volledig algoritme wordt bewezen dat het onhandelbaar is, dan zijn alle NP-volledige problemen onhandelbaar.

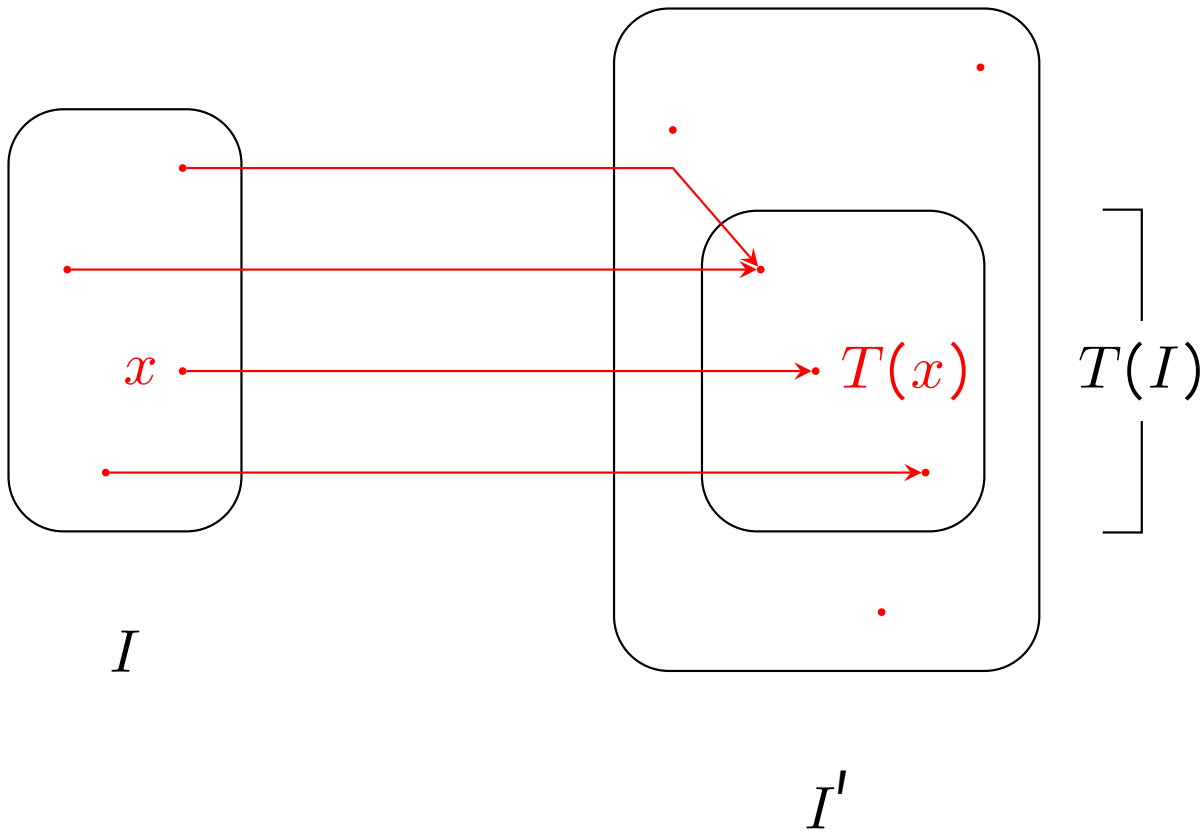
*sterker nog: dan is elk probleem in $\mathcal{NP}$ oplosbaar in polynomiale tijd, dus $\mathcal{P} = \mathcal{NP}$

# De meest waarschijnlijke relatie



# 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 $T(x) \in I'$.



# Reducties

**Definitie**

$T$ heet een polynomiale reductie (of *polynomiale transformatie*) van $P$ naar $Q$ als geldt:

1. $T$ kan worden berekend in polynomiaal begrensde tijd (als functie van $|x|$). D.w.z.: de constructie van $T(x)$ uit $x$ kan in $O(|x|^k)$ stappen in de worst case ($k \geq 0$).
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 $Q$.

3'. voor elke $x$ uit $I$ geldt: als $T(x)$ een ja-instantie is voor $Q$ dan is $x$ een ja-instantie voor $P$. (Dit is equivalent met 3.)

# Reduceerbaar

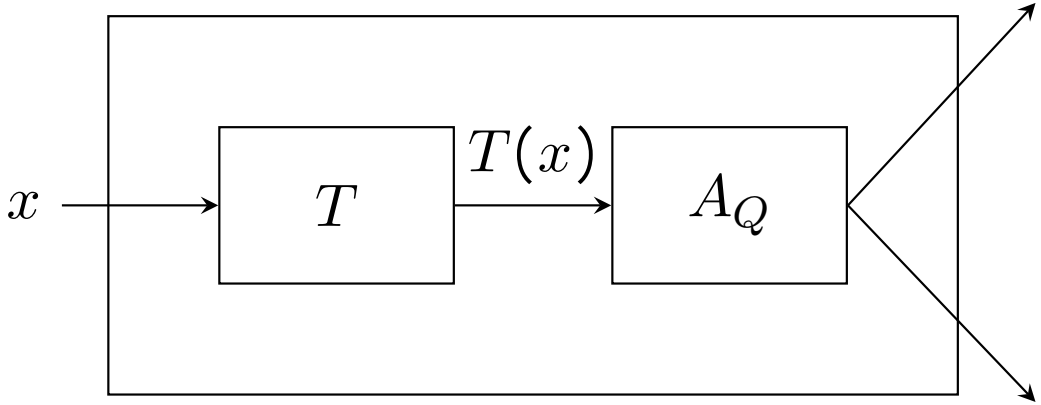
**Definitie**
Een probleem $P$ is polynomiaal reduceerbaar (of polynomiaal transformeerbaar) naar $Q$ als er een polynomiale reductie bestaat van $P$ naar $Q$.

**Notatie**: $P \leq_{\mathsf{P}} Q$.

**Stelling**
Als $P \leq_{\mathsf{P}} Q$ en $Q \in \mathcal{P}$, dan ook $P \in \mathcal{P}$.

# Bewijs stelling

Laat $A_Q$ een polynomiaal deterministisch algoritme zijn voor $Q$.
Een polynomiaal deterministisch algoritme voor $P$ is dan:




ja $\Leftrightarrow$ $T(x)$ is ja-instantie van $Q$ $\Leftrightarrow$ $x$ is ja-instantie van $P$

nee $\Leftrightarrow$ $T(x)$ is nee-instantie van $Q$ $\Leftrightarrow$ $x$ is nee-instantie van $P$

$O(|x|^k) \cdot O(|T(x)|^\ell)$

$O(|x|^{k\ell})$

reductie-eigenschap

# Samengevat: reducties

$P \leq_{\mathsf{P}} Q$ betekent dat er een polynomiale reductie $T$ bestaat van $P$ naar $Q$:

1. $T$ beeldt elke invoer $x$ van beslissingsprobleem $P$ af op een invoer $T(x)$ van beslissingsprobleem $Q$.
2. de constructie van $T(x)$ uit $x$ is polynomiaal ($O(|x|^k)$).
3. reductie-eigenschap: voor elke $x$ uit $I$ (de invoerverzameling van $P$) geldt: $x$ is een ja-instantie voor $P \Leftrightarrow T(x)$ is een ja-instantie voor $Q$.

# NP-hard en NP-volledig

**Definitie**
Een probleem $Q$ is NP-hard (ook wel: NP-moeilijk) als elk probleem $P$ in $\mathcal{NP}$ polynomiaal reduceerbaar is tot $Q$: dus $P \leq_{\mathsf{P}} Q$ 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).

# $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$

**Stelling**

Als een of ander willekeurig NP-volledig probleem in $\mathcal{P}$ zit, dan is $\mathcal{P} = \mathcal{NP}$.

Dit betekent dus: als één enkel NP-volledig probleem $P$ polynomiaal begrensd is, dan zijn alle problemen uit $\mathcal{NP}$ polynomiaal begrensd.

Omgekeerd: als *een* willekeurig probleem in $\mathcal{NP}$ zeker niet polynomiaal begrensd is, dan zijn *alle* NP-volledige problemen niet polynomiaal begrensd.

# Om te onthouden van vandaag

- de klasse $\mathcal{NP}$: beslissingsproblemen waarvoor een niet-deterministisch polynomiaal algoritme bestaat
  - niet-deterministisch: je mag een oplossing gokken
  - polynomiaal: oplossing kan in polynomiale tijd worden geverifieerd (gecontroleerd)
  - ja-instantie: alle invoeren waarvoor *een* executie bestaat die "ja" uitvoert
- niet-deterministisch polynomiaal algoritme:
  1. gokfase: genereer string $s$
  2. verificatiefase: controleer of $s$ een correcte oplossing is in $O(|s|^p \times |x|^q)$
  3. uitvoerfase: antwoord "ja" als de controler slaagt, anders geen uitvoer
  4. als "ja": $|s| \in O(|x|^k)$ en fase 2 dus in $O(|x|^r)$

# (Werk)college

**Volgende college**: dinsdag 18 april, 9u00–10u45, zaal 412 (Snellius)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304 en 303 (Snellius, onderaan de trap)
Opgaven uit het dictaat: 54, 59, 61

# Huiswerk

**Huiswerk 2**

Is nagekeken, cijfers en feedback staan op Brightspace.

**Huiswerk 3**

Komt volgende week.

**Huiswerk 4**

Vervalt, wordt samengevoegd met huiswerk 3.

# Complexiteit 2023 — college 10

18 april 2023

---

## Reducties en $\mathcal{NPC}$

# Vorige keer

- de klasse $\mathcal{NP}$: beslissingsproblemen waarvoor een niet-deterministisch polynomiaal algoritme bestaat
  - niet-deterministisch: je mag een oplossing gokken
  - polynomiaal: oplossing kan in polynomiale tijd worden geverifieerd (gecontroleerd)
  - ja-instantie: alle invoeren waarvoor *een* executie bestaat die "ja" uitvoert
- niet-deterministisch polynomiaal algoritme:
  1. gokfase: genereer string $s$
  2. verificatiefase: controleer of $s$ een correcte oplossing is in $O(|s|^p \times |x|^q)$
  3. uitvoerfase: antwoord "ja" als de controle slaagt, anders geen uitvoer
  4. als "ja": $|s| \in O(|x|^k)$ en fase 2 dus in $O(|x|^r)$

# Vandaag

- Reducties: definitie en voorbeelden
- De klasse $\mathcal{NPC}$ en NP-volledigheidsbewijzen

# TSP en $\mathcal{NP}$

Korte herhaling: $\mathcal{NP}$

Als voorbeeld bekijken we het handelsreizigersprobleem **TSP**.
Hiervoor geldt:

$$\text{TSP} \in \mathcal{NP}$$

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 $\leq k$?
De invoer van het probleem is dus $x = \langle \mathcal{G}, k \rangle$.

*tussen elk tweetal knopen van $\mathcal{G}$ zit een tak

# Invoer

Terzijde: het maakt niet uit voor de polynomialiteit van het algoritme welke representatie is gekozen voor de invoer.

- Neem aan dat $V = \{1, 2, \ldots, n\}$. We nemen nu voor $\mathcal{G}$ de adjacency-list-representatie, en voor $k$ de binaire of decimale representatie (bijvoorbeeld). Dan is zoeken van een tak met bijbehorend gewicht $O(|V|) \subseteq O(|x|)$.
- Hetzelfde als hierboven, maar nu nemen we voor $\mathcal{G}$ de adjacency-matrix-representatie. Dan is zoeken van een tak met bijbehorend gewicht $O(1)$.
- (denkend aan een Turingmachine) Representeer $\langle \mathcal{G}, k \rangle$ als een rij van knopen, gevolgd door de takken met gewichten, gevolgd door $k$, alles binair of decimaal. Dan is zoeken van een tak met bijbehorend gewicht $O(|x|)$.

We gaan hier uit van de eerste optie.

# TSP $\in \mathcal{NP}$

Een polynomiaal begrensd niet-deterministisch algoritme $A$ voor TSP:

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 met gewicht $\leq k$:

   (1) controleer of er precies $|V|$ integers staan: $O(|s|)$

   (2) controleer of elke integer tussen $1$ en $|V|$ zit: $O(|s|)$

   (3) controleer of alle knopen uit $s$ verschillen: $O(|s|^2)$

   (4) controleer of het totale gewicht van de Hamiltonkring (dat stelt $s$ voor als is voldaan aan (1)–(3)) $\leq k$ is: $O(|s| \times |x|)$ (want . . . )

# TSP $\in \mathcal{NP}$

Als de vier tests positief zijn wordt True geretourneerd. Zodra een test negatief uitvalt wordt False teruggegeven (of wordt er een oneindige loop begonnen of . . . ).

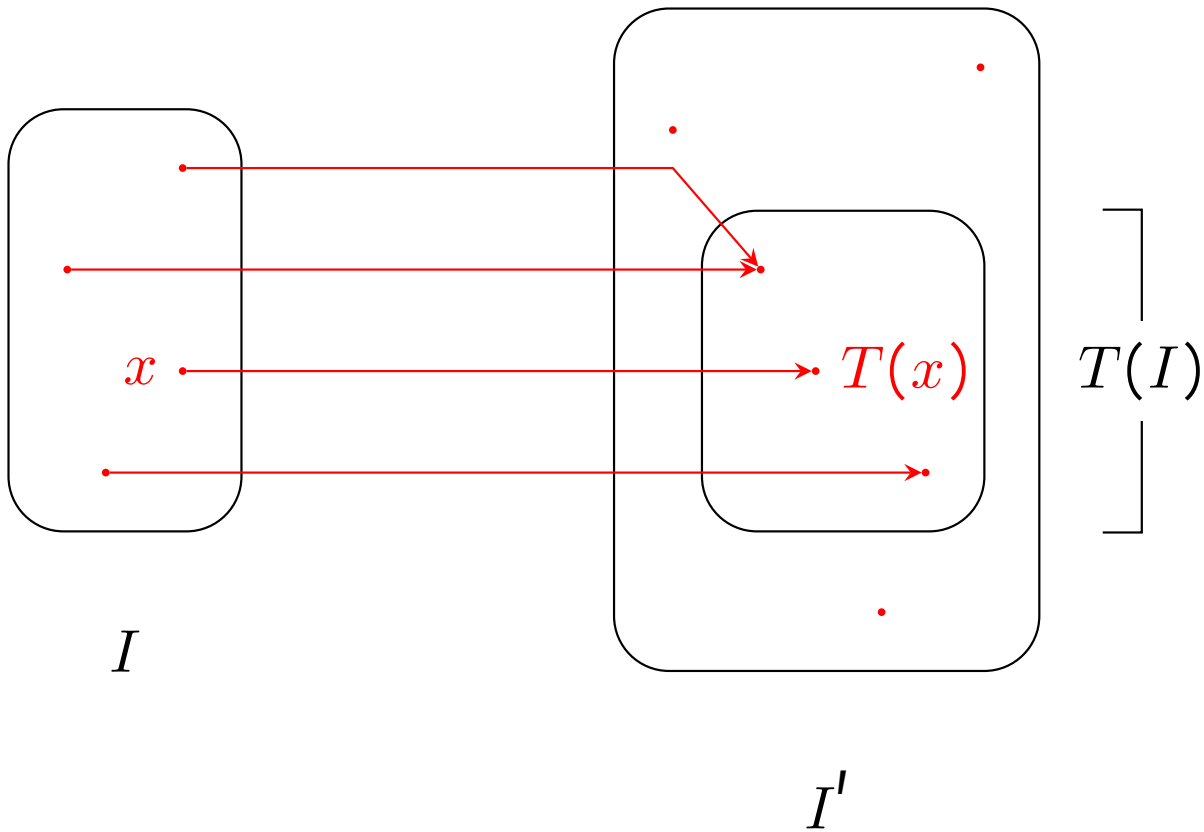
3. Fase 3 (uitvoerfase)
   Als fase 2 True oplevert wordt "ja" uitgevoerd, anders niets.

Voor het hierboven beschreven algoritme geldt:

- Het antwoord van $A$ op invoer $x = \langle \mathcal{G}, k \rangle$ is "ja" $\iff$ er bestaat een string $s$ waarvoor fase 2 True geeft $\iff$ er bestaat een string $s$ die een Hamiltonkring voorstelt met gewicht $\leq k$ $\iff$ $\mathcal{G}$ heeft een Hamiltonkring met gewicht $\leq k$ $\iff$ $x$ is een ja-instantie voor TSP.
- Het algoritme is polynomiaal, want voor een ja-instantie stelt $s$ een (goede) Hamiltonkring voor, dus dan is $|s| \in O(|V|) \subseteq O(|x|)$ en derhalve is fase 2 dan $O(|x|^2)$.

# 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 $T(x) \in I'$.



# Reducties

**Definitie**

$T$ heet een polynomiale reductie (of *polynomiale transformatie*) van $P$ naar $Q$ als geldt:

1. $T$ kan worden berekend in polynomiaal begrensde tijd (als functie van $|x|$). D.w.z.: de constructie van $T(x)$ uit $x$ kan in $O(|x|^k)$ stappen in de worst case ($k \geq 0$).
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 $Q$.

3'. voor elke $x$ uit $I$ geldt: als $T(x)$ een ja-instantie is voor $Q$ dan is $x$ een ja-instantie voor $P$. (Dit is equivalent met 3.)

# Reduceerbaar

**Definitie**
Een probleem $P$ is polynomiaal reduceerbaar (of polynomiaal transformeerbaar) naar $Q$ als er een polynomiale reductie bestaat van $P$ naar $Q$.

**Notatie**: $P \leq_{\mathsf{P}} Q$.

**Stelling**
Als $P \leq_{\mathsf{P}} Q$ en $Q \in \mathcal{P}$, dan ook $P \in \mathcal{P}$.

Bewijs: zie college 9

# Samengevat: reducties

$P \leq_{\mathsf{P}} Q$ betekent dat er een polynomiale reductie $T$ bestaat van $P$ naar $Q$:

1. $T$ beeldt elke invoer $x$ van beslissingsprobleem $P$ af op een invoer $T(x)$ van beslissingsprobleem $Q$.
2. de constructie van $T(x)$ uit $x$ is polynomiaal ($O(|x|^k)$).
3. reductie-eigenschap: voor elke $x$ uit $I$ (de invoerverzameling van $P$) geldt: $x$ is een ja-instantie voor $P \Leftrightarrow T(x)$ is een ja-instantie voor $Q$.

# HC1 $\leq_P$ HC2

HC1: **gegeven** een *gerichte* graaf $\mathcal{G} = (V, E)$.
**Vraag**: heeft $\mathcal{G}$ een Hamiltonkring?

HC2: **gegeven** een *ongerichte* graaf $\mathcal{G} = (V, E)$.
**Vraag**: heeft $\mathcal{G}$ een Hamiltonkring?

**Bewering**: HC1 $\leq_P$ HC2: zie volgende sheet.

**Opmerking**: er geldt ook: HC2 $\leq_P$ HC1. Bedenk zelf een eenvoudige reductie.

# Een reductie van HC1 naar HC2

Transformatie $T$ die een gerichte graaf $\mathcal{G} = (V, E)$ afbeeldt op een ongerichte graaf $T(\mathcal{G}) = \mathcal{G}' = (V', E')$:

- $V' = \{v_1, v_2, v_3 \mid v \in V\}$: elke knoop $v \in V$ wordt afgebeeld op een drietal knopen $v_1, v_2, v_3$.
- $E' = \{(v_1, v_2), (v_2, v_3) \mid v \in V\} \cup \{(v_3, w_1) \mid (v, w) \in E\}$: 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 (pijl) van $v$ naar $w$ in $\mathcal{G}$ komt een tak in $\mathcal{G}'$ tussen $v_3$ en $w_1$.

Dan geldt

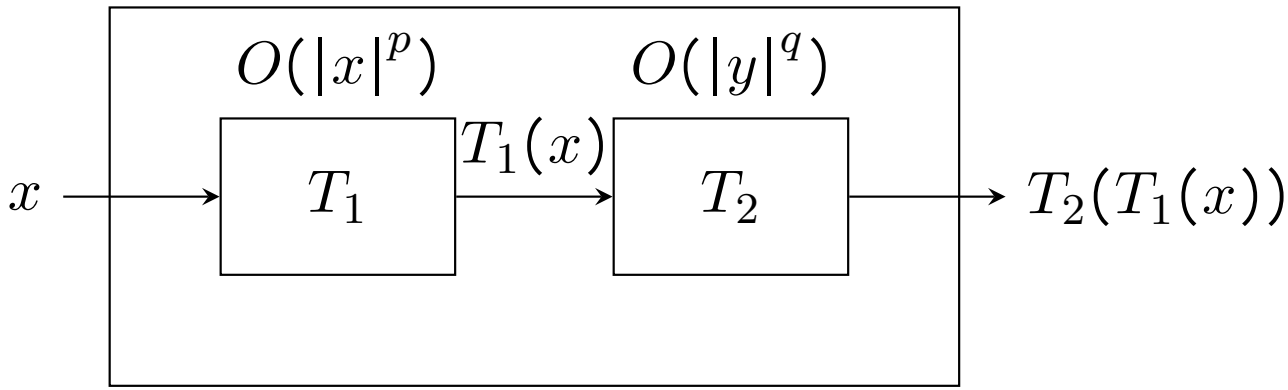
1. $T$ kan in polynomiaal begrensde tijd worden berekend (constructie van $T(\mathcal{G})$ kan zeker in $O(|\mathcal{G}|^q)$, bijv. met $q = 2$)
2. $\mathcal{G}$ is een ja-instantie voor HC1 $\Leftrightarrow$ $T(\mathcal{G})$ is een ja-instantie voor HC2, ofwel: $\mathcal{G}$ heeft een gerichte Hamiltonkring $\Leftrightarrow$ $\mathcal{G}'$ heeft een ongerichte Hamiltonkring

# Transitief

**Lemma**

$\leq_P$ is transitief, d.w.z.: als $P_1 \leq_P P_2$ en $P_2 \leq_P P_3$ dan ook $P_1 \leq_P P_3$.

Het bewijs:




- De samenstelling van $T_1$ en $T_2$ is polynomiaal begrensd omdat $T_1$ en $T_2$ dat zijn: $O(|x|^{pq})$ (let op: **niet** $O(|x|^{p+q})$!)
- $T_2 \circ T_1(x) = T_2(T_1(x))$ is een ja-instantie van $P_3 \iff T_1(x)$ is een ja-instantie van $P_2 \iff x$ is een ja-instantie van $P_1$ (volgt uit de reductie-eigenschap van $T_2$ en $T_1$)

# NP-hard en NP-volledig

**Definitie**
Een probleem $Q$ is NP-hard (ook wel: NP-moeilijk) als elk probleem $P$ in $\mathcal{NP}$ polynomiaal reduceerbaar is tot $Q$: dus $P \leq_{\mathsf{P}} Q$ 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).

# NP-volledigheid bewijzen

**Stelling**

Stel $Q$ is een probleem waarvoor geldt dat $P \leq_{\mathsf{P}} Q$ 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 volgt uit de transitiviteit van $\leq_{\mathsf{P}}$ (twee slides terug) en de definitie van NP-hard (vorige slide).

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

# Reductiemethode: $Q \in \mathcal{NPC}$

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

   Stap 3 valt 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 $T(x)$ van $I'$

   3b. Laat zien dat $T(x)$ uit $x$ kan worden geconstrueerd in polynomiaal begrensde tijd ($O(|x|^k)$ voor zekere $k \geq 0$)

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

# Maar...

Deze reductiemethode is **inductief**:

$$\frac{P \in \mathcal{NPC} \quad P \leq_{\mathsf{P}} Q \quad Q \in \mathcal{NP}}{Q \in \mathcal{NPC}}$$

Wat ontbreekt hier nog?

# Maar...

Deze reductiemethode is **inductief**:

$$\frac{P \in \mathcal{NPC} \quad P \leq_{\mathsf{P}} Q \quad Q \in \mathcal{NP}}{Q \in \mathcal{NPC}}$$

Wat ontbreekt hier nog?

Een **basisgeval**:

$$\frac{}{P_0 \in \mathcal{NPC}}$$

# Stephen Cook

In 1971 bewees Stephen Cook op een directe manier (dus door een reductie te geven van alle problemen uit $\mathcal{NP}$) dat SAT NP-volledig is. (Details in volgend college.)

Stelling

Gegeven een *arbitrair* probleem $P \in \mathcal{NP}$. Dan is $P$ reduceerbaar tot SAT: $P \leq_P$ SAT.

Sindsdien is met behulp van de reductiemethode van veel bekende problemen aangetoond dat ze NP-volledig zijn. Bijvoorbeeld:

$$\text{SAT} \leq_P \text{3SAT} \leq_P \text{Kliek} \leq_P \text{VC}$$
$$\text{3SAT} \leq_P \text{HC2} \leq_P \text{TSP}$$
$$\text{SAT} \leq_P \text{3Kleur} \leq_P \text{4Kleur}$$

# 3SAT en Kliek

## 3SAT

Gegeven een logische formule $\phi$ in 3-CNF. Bestaat er een waardering die $\phi$ waar maakt?

**Definitie** Een logische formule $\phi$ staat in 3-CNF als $\phi$ een conjunctie is van clausules, waarbij elke clausule een disjunctie is van drie verschillende* literals. variabele: $x_5, x_7, \ldots$; literal: $x_3, \neg x_6, \ldots$; clausule: $x_5 \vee \neg x_6 \vee x_8, \ldots$

## Kliek

Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$ en een geheel getal $k$ $(0 \leq k \leq |V|)$. Is er in $\mathcal{G}$ een kliek ter grootte $k$?

**Definitie**

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 $(u, v) \in E$. (Oftewel: $V'$ is volledig.)

*deze eis wordt ook wel weggelaten bij de definitie van 3SAT

# 3SAT $\leq_P$ Kliek

Er geldt: 3SAT $\leq_P$ Kliek. Om dit aan te tonen moeten we een logische formule in 3-CNF afbeelden op een invoer voor Kliek, dus op een ongerichte graaf en een geheel getal.

Zij $\phi$ een logische formule in 3-CNF, met zeg $m$ clausules: $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$. Hierin is $C_r = \ell_1^r \vee \ell_2^r \vee \ell_3^r$ $(r = 1, \ldots, m)$ en $\ell_1^r$, $\ell_2^r$ en $\ell_3^r$ steeds verschillend bij vaste $r$.

Construeer nu een ongerichte graaf $\mathcal{G}_\phi = (V, E)$ als volgt.

Voor elke clausule $C_r$ uit $\phi$ doen we drie knopen $v_1^r$, $v_2^r$ en $v_3^r$ in $V$ (deze corresponderen met $\ell_1^r$, $\ell_2^r$ en $\ell_3^r$). $\mathcal{G}_\phi$ heeft dus $3m$ knopen.

# 3SAT $\leq_P$ Kliek

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 $r \neq s$), en
- de bijbehorende $\ell_i^r$ en $\ell_j^s$ zó zijn dat $\ell_i^r \neq \neg\ell_j^s$, met andere woorden: $\ell_i^r$ en $\ell_j^s$ zijn niet elkaars negatie.

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

- De constructie van $T(\phi)$ uit $\phi$ kan in $O(|\phi|^q)$ stappen voor zekere $q \geq 0$ (dus polynomiaal).
- Er is een waardering die $\phi$ waarmaakt $\iff$ $\mathcal{G}_\phi$ heeft een kliek ter grootte $m$.

# 3SAT $\leq_P$ Kliek: voorbeeld

Laat $\phi = C_1 \wedge C_2 \wedge C_3$, met $C_1 = x_1 \vee \neg x_2 \vee \neg x_3$,
$C_2 = \neg x_1 \vee x_2 \vee x_3$ en $C_3 = x_1 \vee x_2 \vee x_3$. Hier is dus $m = 3$.

Dan $v_1^1 = x_1$, $v_2^1 = \neg x_2$ en $v_3^1 = \neg x_3$; alle uit clausule $C_1$, enz.

- een waardering $w$ die $\phi$ waarmaakt is bijvoorbeeld: $w(x_1) = w(x_2) =$ False en $w(x_3) =$ True. Een bijbehorende kliek in $\mathcal{G}_\phi$ ter grootte 3 is dan $\{v_2^1, v_3^2, v_3^3\}$*.
- een kliek ter grootte 3 in $\mathcal{G}_\phi$ is bijvoorbeeld $\{v_1^1, v_2^2, v_2^3\}$. Een bijbehorende waardering is $w(x_1) = w(x_2) = w(x_3) =$ True. Deze maakt $\phi$ waar.

*De bovenindex komt overeen met de clausule. Uit elk drietal (clausule) één knoop (literal).

# SAT en 3SAT

SAT

Gegeven een logische formule $\phi$ in CNF. Bestaat er een waardering die $\phi$ waar maakt?

**Definitie**

Een logische formule $\phi$ staat in CNF als $\phi$ een conjunctie is van clausules, waarin elke clausule een disjunctie is van literals.

3SAT

Gegeven een logische formule $\phi$ in 3-CNF. Bestaat er een waardering die $\phi$ waar maakt?

**Definitie**

Een logische formule $\phi$ staat in 3-CNF als $\phi$ in CNF staat en elke clausule bestaat uit precies drie (doorgaans verschillende) literals.

# SAT $\leq_P$ 3SAT

Er geldt: SAT $\leq_P$ 3SAT. Om dit aan te tonen moeten we een logische formule $\phi$ in CNF afbeelden op een logische formule $\phi'$ in 3-CNF. We gaan er voor het gemak van uit dat de $\ell_1, \ell_2, \ldots, \ell_k$ per clausule al verschillend zijn (kan anders worden bewerkstelligd in $O(|\phi|^2)$). Op clausuleniveau werkt de transformatie $T$ als volgt:

$$\ell_1 \mapsto (\ell_1 \vee \widetilde{\ell_2} \vee \widetilde{\ell_3}) \wedge (\ell_1 \vee \widetilde{\ell_2} \vee \neg\widetilde{\ell_3}) \wedge (\ell_1 \vee \neg\widetilde{\ell_2} \vee \widetilde{\ell_3}) \wedge (\ell_1 \vee \neg\widetilde{\ell_2} \vee \neg\widetilde{\ell_3})$$
$$\ell_1 \vee \ell_2 \mapsto (\ell_1 \vee \ell_2 \vee \widetilde{\ell_3}) \wedge (\ell_1 \vee \ell_2 \vee \neg\widetilde{\ell_3})$$
$$\ell_1 \vee \ell_2 \vee \ell_3 \mapsto \ell_1 \vee \ell_2 \vee \ell_3$$
$$\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4 \mapsto (\ell_1 \vee \ell_2 \vee \widetilde{\ell_5}) \wedge (\ell_3 \vee \ell_4 \vee \neg\widetilde{\ell_5})$$
$$\ell_1 \vee \ell_2 \vee \ell_3 \vee \ell_4 \vee \ell_5 \mapsto (\ell_1 \vee \ell_2 \vee \widetilde{\ell_6}) \wedge (\ell_3 \vee \neg\widetilde{\ell_6} \vee \widetilde{\ell_7}) \wedge (\ell_4 \vee \ell_5 \vee \neg\widetilde{\ell_7})$$

# SAT $\leq_P$ 3SAT

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

$$\ell_1 \vee \ell_2 \vee \ldots \vee \ell_{k-1} \vee \ell_k \mapsto (\ell_1 \vee \ell_2 \vee \widetilde{\ell_{k+1}}) \wedge (\ell_3 \vee \neg\widetilde{\ell_{k+1}} \vee \widetilde{\ell_{k+2}}) \wedge (\ell_4 \vee \neg\widetilde{\ell_{k+2}} \vee \widetilde{\ell_{k+3}}) \wedge \ldots \wedge (\ell_{k-2} \vee \neg\widetilde{\ell_{2k-4}} \vee \ell_{2k-3}) \wedge (\ell_{k-1} \vee \ell_k \vee \neg\widetilde{\ell_{2k-3}})$$

Hierin zijn $\widetilde{\ell_{k+1}}, \widetilde{\ell_{k+2}}, \ldots, \widetilde{\ell_{2k-3}}$ 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. Het beeld van een conjunctie van clausules definiëren we als een conjunctie van de beelden van de samenstellende clausules:

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

# SAT $\leq_P$ 3SAT

Voor deze transformatie $T$ geldt:

- de constructie van $T(\phi)$ uit $\phi$ kan met een polynomiaal begrensd ($O(|\phi|^q)$) algoritme.
- $\phi$ is een ja-instantie van SAT $\Leftrightarrow$ $T(\phi)$ is een ja-instantie van 3SAT.
- ofwel: er is een waardering die $\phi$ waarmaakt $\Leftrightarrow$ er is een waardering die $T(\phi)$ waarmaakt.
- conclusie uit de vorige punten: SAT $\leq_P$ 3SAT.

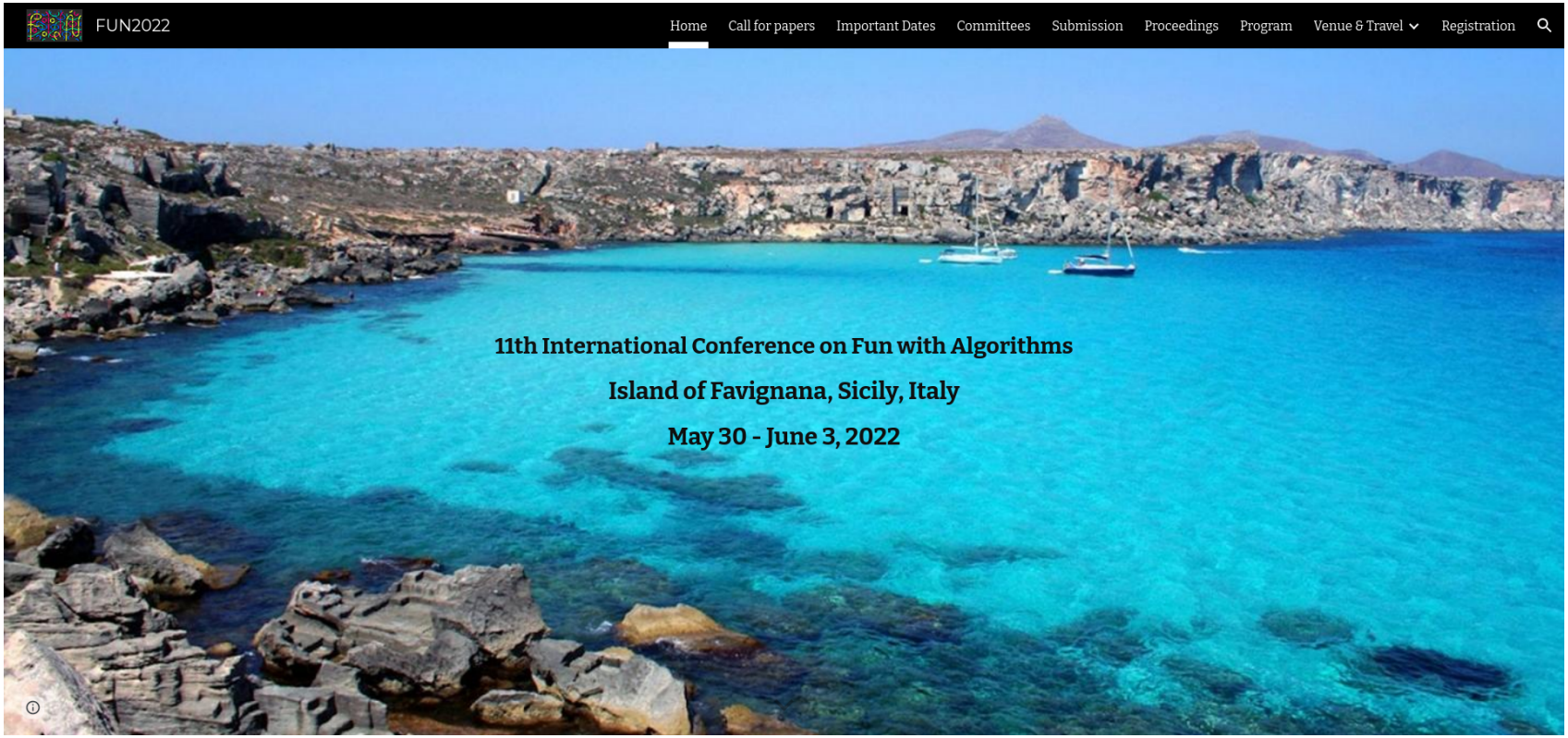
# Vragen

We hebben nu: SAT $\leq_P$ 3SAT (*)

Verder is 1SAT: gegeven een logische formule $\phi$ in 1-CNF. Bestaat er een waardering die $\phi$ waar maakt? Een logische formule $\phi$ in 1-CNF heeft de volgende vorm: $\phi = \ell_1 \wedge \ell_2 \wedge \ldots \wedge \ell_n$.

1. Stel dat we weten dat 3SAT $\in \mathcal{NPC}$ en SAT $\in \mathcal{NP}$. Volgt dan uit (*) dat SAT $\in \mathcal{NPC}$?
2. Stel dat we weten dat SAT $\in \mathcal{NPC}$ en 3SAT $\in \mathcal{NP}$. Volgt dan uit (*) dat 3SAT $\in \mathcal{NPC}$?
3. Stel dat we weten dat 3SAT $\in \mathcal{NPC}$. Is 1SAT $\leq_P$ 3SAT?
4. Stel dat we weten dat 3SAT $\in \mathcal{NPC}$. Is 3SAT $\leq_P$ 1SAT?

# FUN




**FUN 2022**: `https://sites.google.com/view/fun2022/home`

# FUN 2016: Two-Dots

**FUN 2016**: https://www2.idsia.ch/cms/fun16/

**Proceedings**: https://drops.dagstuhl.de/opus/portals/lipics/index.php?semnr=16004

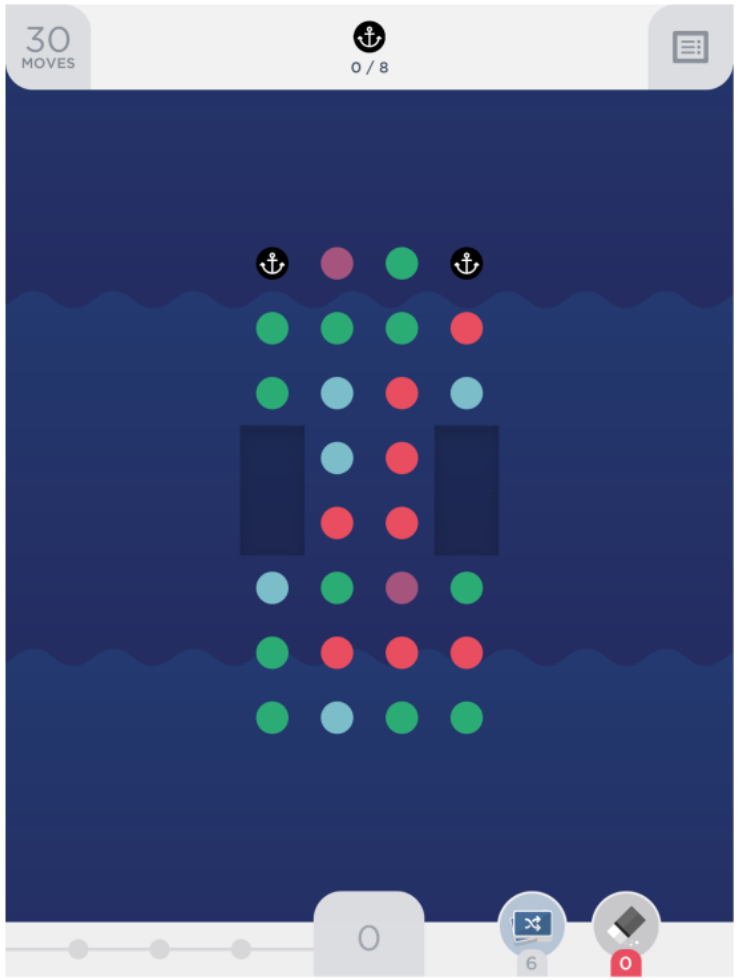
**Two-Dots is NP-Complete**: https://drops.dagstuhl.de/opus/frontdoor.php?source_opus=5883

**Two-Dots**: https://www.dots.co/twodots/

**Online alternatief**: https://plays.org/game/two-dots/

# Two-Dots


**(a)** Level One of Two Dots.




**(b)** A more advanced level of Two Dots.

**Doel**: gegeven $k$ zetten, verzamel $x$

# Two-Dots

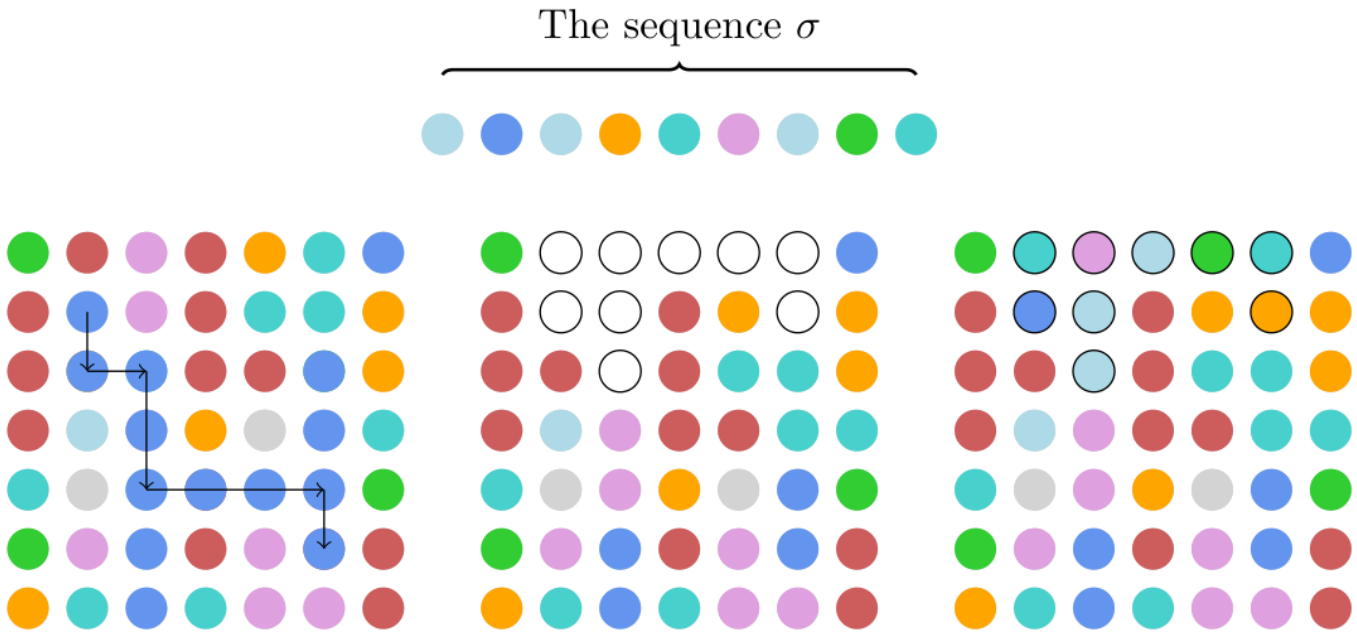



**Figure 2** A depiction of the regular move. The first panel shows the set of locations that are committed to the move, the second panel shows the voids created, and the third panel shows how the voids are filled in accordance with $\sigma$.

Reguliere zet: orthogonaal pad, verzamelt pad

# Two-Dots

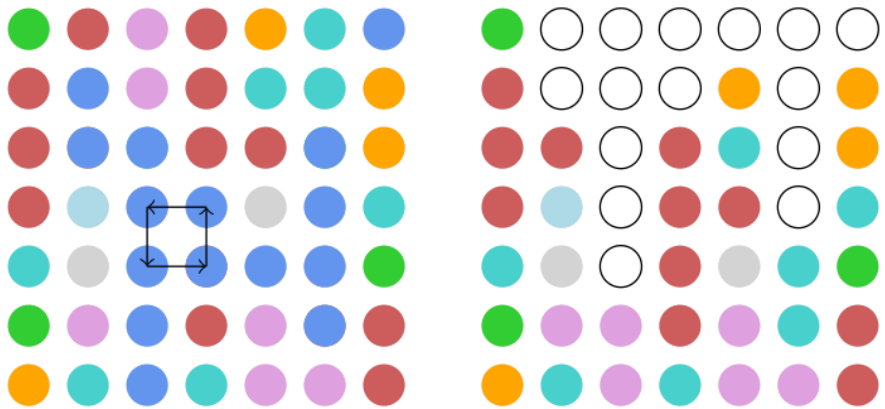


**Figure 3** A depiction of the square move, which has the effect of eliminating all the blue dots from the board. The example is rather similar to the above, but note the difference in the number of voids created. The process for filling up the voids is identical, and is therefore omitted.

Speciale zet: $2 \times 2$ vierkant, verzamelt hele kleur

# Two-Dots en Kliek

**Kliek**: gegeven een graaf $\mathcal{G} = (V, E)$ en een getal $k$. Bevat $\mathcal{G}$ een kliek (volledige deelgraaf) van grootte $k$?


**(a)** The Vertex Gadgets. **(b)** A Tester Gadget for the edges $(u, v)$ and $(w, v)$.

**Figure 4** The vertex and tester gadgets used in the reduction from CLIQUE. The colors $\alpha$ and $\beta$ are depicted, respectively, by blue and red.

**Two-Dots**: $2k + \binom{k}{2}$ zetten, doel is $2k$ blauw en $4\binom{k}{2}$ rood

# Two-Dots en Exact Cover: zonder de vierkante zet

**Exact Cover**: gegeven een verzameling $U = \{u_1, \ldots, u_n\}$ en een verzameling van verzamelingen $F = \{S_1, \ldots, S_m\}$ (waar $|S_i| = 3$ voor alle $i$). Bestaat er een deelverzameling $G \subseteq F$ zodat elk element van $U$ precies één keer voorkomt in de elementen van $G$?

(Dit is NP-volledig.)

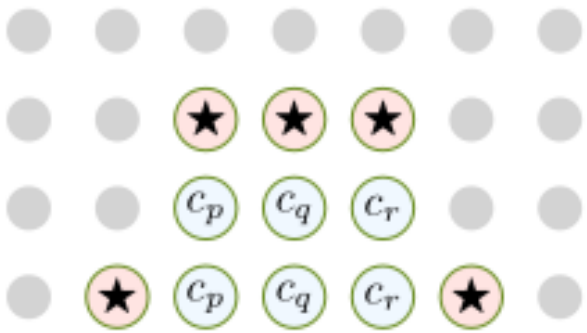
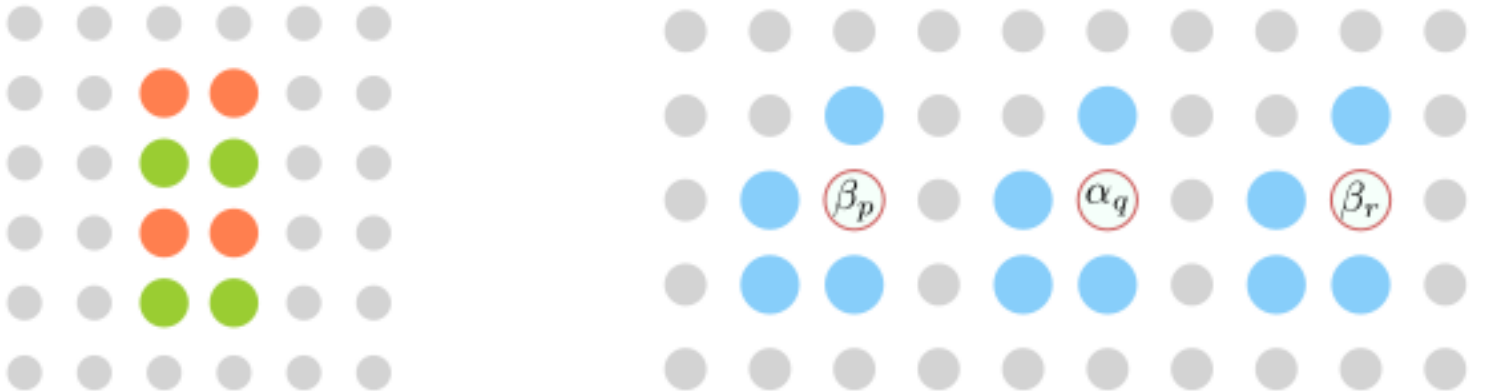



**Figure 5** The gadget corresponding to a set $S = \{u_p, u_q, u_r\}$, from an instance of X3C.

**Two-Dots**: $n + \frac{n}{3}$ zetten, doel is $2$ van elke $c_i$ en $\frac{5n}{3}$ $\star$

# Two-Dots en 3-SAT: doel van constante grootte

**3-SAT**: gegeven een formule $\phi$ in 3-CNF met clausules $C_1, \ldots, C_m$ en variabelen $x_1, \ldots, x_n$, bestaat er een waardering die $\phi$ waar maakt?




**(a)** The variable gadget, where the two colors shown correspond to the two possible assignments.

**(b)** A Clause gadget corresponding to the clause $(\overline{x_p}, x_q, \overline{x_r})$. The color corresponding to the clause is depicted in Blue.

**Figure 6** The variable and clause gadgets used in the reduction from 3-SAT.

**Two-Dots**: $2n + m + 1$ zetten, doel is $2$ ⋆
(buiten beeld: twee ⋆ gescheiden door een stapel overeenkomend met de kleuren van de clausules)

# Om te onthouden van vandaag

- polynomiale reductie $T$ van $P$ naar $Q$: $P \leq_{\mathsf{P}} Q$
  - $T$ beeldt elke invoer $x$ van $P$ af op een invoer $T(x)$ van $Q$
  - constructie van $T(x)$ uit $x$ is polynomiaal (in $|x|$)
  - reductie-eigenschap: $x$ is ja-instantie voor $P \iff T(x)$ is ja-instantie voor $Q$
  - hoop voorbeelden
- $Q$ is NP-hard (NP-moeilijk) $\iff P \leq_{\mathsf{P}} Q$ voor alle $P \in \mathcal{NP}$
- $Q$ is NP-volledig ($\mathcal{NPC}$) als $Q$ NP-hard is en $Q \in \mathcal{NP}$
- NP-volledigheidsbewijs voor $Q$: bewijs dat $Q \in \mathcal{NP}$ en dat $P \leq_{\mathsf{P}} Q$ voor $P \in \mathcal{NPC}$
- SAT is het oer-$\mathcal{NPC}$-probleem

# (Werk)college

**Volgende college**: dinsdag 25 april, 9u00–10u45, zaal 407-409 (Snellius)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304 en 303 (Snellius, onderaan de trap)
Opgaven uit het dictaat: 58, 63, 64, 65

# Huiswerk

**Huiswerk 3**

Verschijnt vanmiddag op de website.

Inleveren via Brightspace, uiterlijk dinsdag 9 mei, 23u59.

**Huiswerk 4**

Vervalt.

# Universitaire verkiezingen

**Universiteitsraad**:
https://www.organisatiegids.universiteitleiden.nl/medezeggenschap/universiteitsraad

**Faculteitsraad**: https://www.organisatiegids.universiteitleiden.nl/faculteiten-en-instituten/wiskunde-en-natuurwetenschappen/medezeggenschap/faculteitsraad

**Verkiezingen**:
https://www.organisatiegids.universiteitleiden.nl/medezeggenschap/universitaire-verkiezingen

**Partijen en standpunten**: https://www.student.universiteitleiden.nl/organisatie/medezeggenschap/universitaire-verkiezingen/partijen/wiskunde-en-natuurwetenschappen/informatica-bsc

Stemmen kan tot **donderdag 20 april 16u00**

# Complexiteit 2023 — college 11

25 april 2023

---

## Satisfiability

# Vorige keer

- polynomiale reductie $T$ van $P$ naar $Q$: $P \leq_{\mathsf{P}} Q$
  - $T$ beeldt elke invoer $x$ van $P$ af op een invoer $T(x)$ van $Q$
  - constructie van $T(x)$ uit $x$ is polynomiaal (in $|x|$)
  - reductie-eigenschap: $x$ is ja-instantie voor $P \iff T(x)$ is ja-instantie voor $Q$
  - hoop voorbeelden
- $Q$ is NP-hard (NP-moeilijk) $\iff P \leq_{\mathsf{P}} Q$ voor alle $P \in \mathcal{NP}$
- $Q$ is NP-volledig ($\mathcal{NPC}$) als $Q$ NP-hard is en $Q \in \mathcal{NP}$
- NP-volledigheidsbewijs voor $Q$: bewijs dat $Q \in \mathcal{NP}$ en dat $P \leq_{\mathsf{P}} Q$ voor $P \in \mathcal{NPC}$
- SAT is het oer-$\mathcal{NPC}$-probleem

# Vandaag

- Stelling van Cook–Levin: SAT is NP-moeilijk
- kort over de werking van SAT-solvers (geen tentamenstof)
- (Strong) Exponential Time Hypothesis (geen tentamenstof)

# Nog even tussendoor

On the Complexity of Two Dots for Narrow Boards and Few Colors
`https://drops.dagstuhl.de/opus/volltexte/2018/8798/`
(FUN 2018)

Nog twee NP-volledigheidsbewijzen:

- een met maar drie kleuren (maar veel kolommen)
- een met maar twee kolommen (maar veel kleuren)

# Two Dots is NP-volledig met drie kleuren

**Exact Cover**: gegeven een verzameling $U = \{u_1, \ldots, u_n\}$ en een verzameling van verzamelingen $F = \{S_1, \ldots, S_m\}$ (waar $|S_i| = 3$ voor alle $i$). Bestaat er een deelverzameling $G \subseteq F$ zodat elk element van $U$ precies één keer voorkomt in de elementen van $G$?

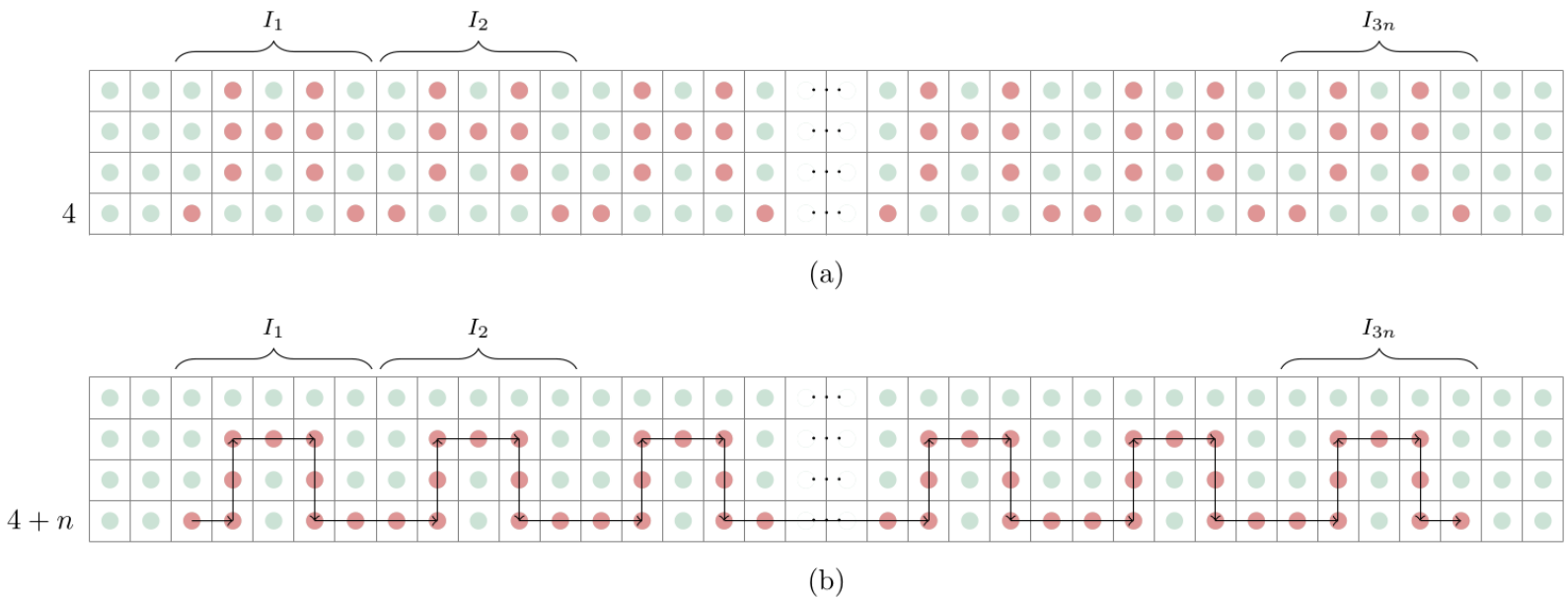



**Figure 4** (a) Initial setup of the check-wire. (b) The check-wire once it gets aligned.

# Two Dots is NP-volledig met drie kleuren

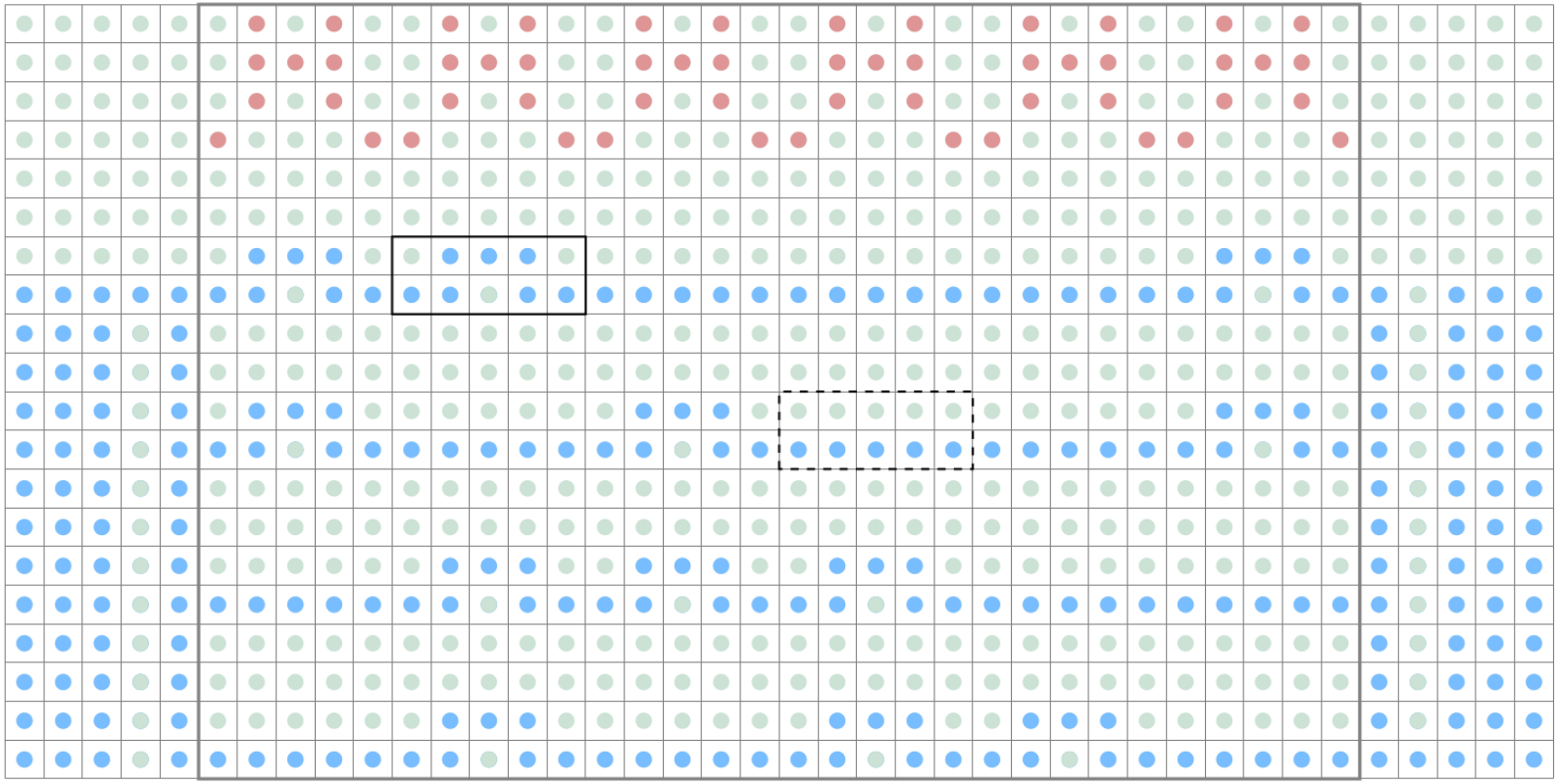


**Figure 3** Overview of the reduction.

Doel: twee zetten voor een hoop blauw en al het rood.

# Two Dots is NP-volledig met twee kolommen

Reductie vanaf **3SAT**.

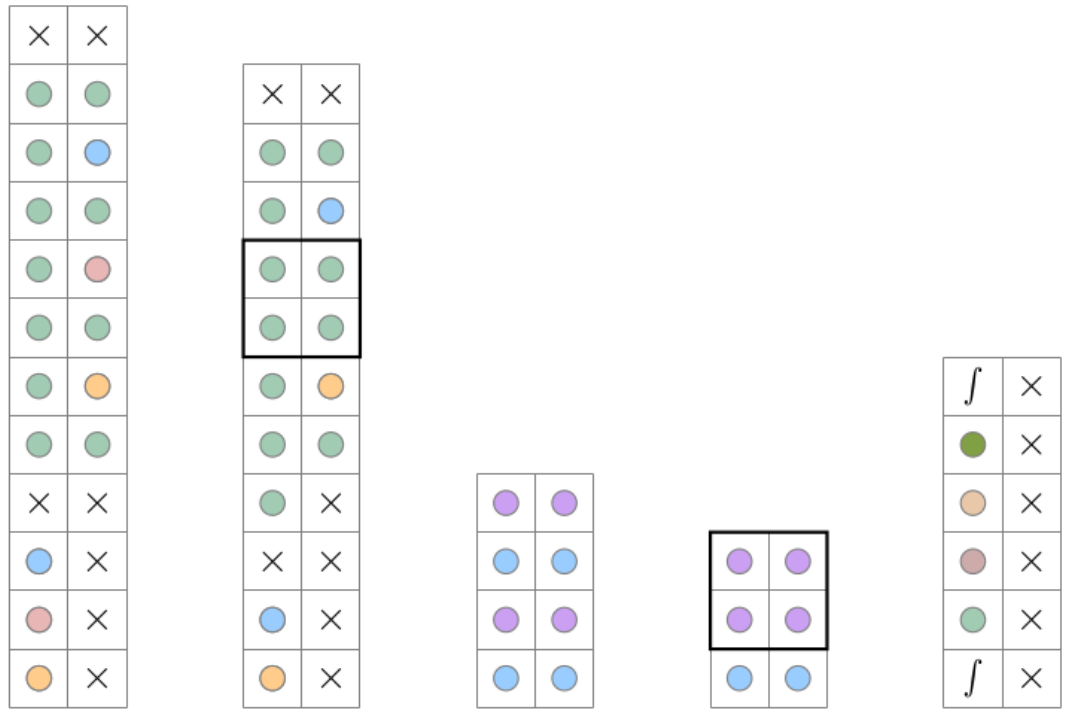



**Figure 6** (a) Initial setup of the clause gadget. (b) The clause gadget in its aligned state. (c) Initial setup of the variable gadget. (d) The state of the variable gadget after one move on one of the literals. (e) The formula-check gadget.

# Two Dots is NP-volledig met twee kolommen

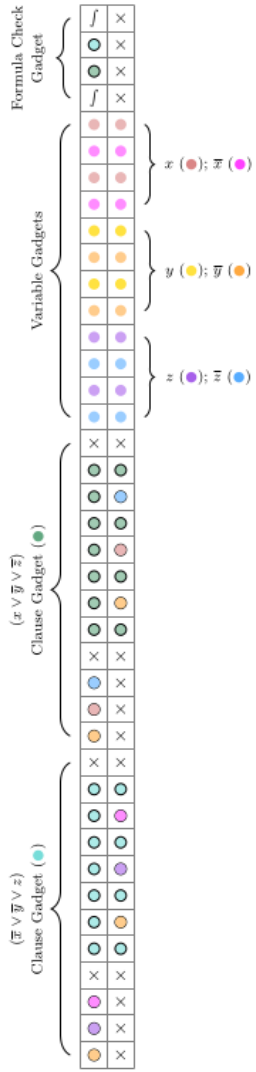



**Figure 5** Overview of the reduction. The grid cells marked × are filled with distinct colors different from the ones used to represent variables and clauses. The goal of the game is to hit two dots of color ∫ and the number of moves are unbounded.

# NP-volledigheid bewijzen

**Stelling**

Stel $Q$ is een probleem waarvoor geldt dat $P \leq_{\mathsf{P}} Q$ 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 volgt uit de transitiviteit van $\leq_{\mathsf{P}}$ (twee slides terug) en de definitie van NP-hard (vorige slide).

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

# Maar...

Deze reductiemethode is **inductief**:

$$\frac{P \in \mathcal{NPC} \quad P \leq_{\mathsf{P}} Q \quad Q \in \mathcal{NP}}{Q \in \mathcal{NPC}}$$

Wat ontbreekt hier nog?

# Maar...

Deze reductiemethode is **inductief**:

$$\frac{P \in \mathcal{NPC} \quad P \leq_{\mathsf{P}} Q \quad Q \in \mathcal{NP}}{Q \in \mathcal{NPC}}$$

Wat ontbreekt hier nog?

Een **basisgeval**:

$$\frac{}{P_0 \in \mathcal{NPC}}$$

# Stephen Cook

In 1971 bewees Stephen Cook op een directe manier (dus door een reductie te geven van alle problemen uit $\mathcal{NP}$) dat SAT NP-volledig is. (Details in volgend college.)

Stelling
Gegeven een *arbitrair* probleem $P \in \mathcal{NP}$. Dan is $P$ reduceerbaar tot SAT: $P \leq_{\mathsf{P}}$ SAT.

Sindsdien is met behulp van de reductiemethode van veel bekende problemen aangetoond dat ze NP-volledig zijn. Bijvoorbeeld:

$$\begin{aligned} \text{SAT} \leq_{\mathsf{P}} \text{3SAT} \leq_{\mathsf{P}} \text{Kliek} \leq_{\mathsf{P}} \text{VC} \\ \text{3SAT} \leq_{\mathsf{P}} \text{HC2} \leq_{\mathsf{P}} \text{TSP} \\ \text{SAT} \leq_{\mathsf{P}} \text{3Kleur} \leq_{\mathsf{P}} \text{4Kleur} \end{aligned}$$

# Stelling van Cook–Levin

Stephen Cook

Leonid Levin

Stelling (Cook 1971; Levin ±1973)
Gegeven een *willekeurig* probleem $P \in \mathcal{NP}$. Dan is $P$ reduceerbaar tot SAT: $P \leq_{\mathsf{P}} \mathsf{SAT}$.

Nu: het bewijs van Cook. *The complexity of theorem proving procedures*, `https://doi.org/10.1145/800157.805047`.
Of: `https://www.cs.toronto.edu/~sacook/` (zoek zelf verder)

# Even tussendoor: LaTeX

LaTeX stamt uit 1984, TeX uit 1978.

The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of determining whether a given propositional formula is a tautology. Here "reduced" means, roughly speaking, that the first problem can be solved deterministically in polynomial time provided an oracle is available for solving the second. From this notion of reducible, polynomial degrees of difficulty are defined, and it is shown that the problem of determining tautologyhood has the same polynomial degree as the certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will give evidence that {tautologies} is a difficult set to recognize, since many apparently difficult problems can be reduced to determining tautologyhood. By reduced we mean, roughly speaking, that if tautologyhood could be decided instantly (by an "oracle") then these problems could be decided in polynomial time. In order to make this notion precise, we introduce query machines, which are like Turing machines with oracles

# Even tussendoor: LaTeX

Definition: We will denote deg ({0}) by $\mathcal{L}_*$, where 0 denotes the zero function.

Thus $\mathcal{L}_*$ is the class of sets recognizable in polynomial time. $\mathcal{L}_*$ was discussed in [2], p. 5, and is the string analog of Cabham's class of functions [3].

We now define the following special sets of strings.

# Even tussendoor: LaTeX

predicate calculus. Further, if M halts in s steps, then

$\phi(A(M)) \leq s^2$. Thus, if, contrary to (2), $T_Q(k) = 0(\sqrt{k}/\log^2 k)$, then a modification of $M_Q$ could verify in only

$$0(\sqrt{s^2}/\log^2 s^2) = 0(s/\log^2 s)$$

steps that M halted in s steps

# Bewijs van Cook

**Schets van het bewijs**

1. Omdat $P \in \mathcal{NP}$, is er een niet-deterministisch algoritme $A$ (een niet-deterministische Turingmachine) voor $P$. Verder is $A$ polynomiaal begrensd ($O(|x|^k)$ op invoer $x$).
2. Dit algoritme zal voor elke invoer $x$ van $P$ worden gemodelleerd als een logische formule $\phi = T(x)$ in CNF: deze $\phi$ beschrijft a.h.w. de berekening van $A$, werkend op $x$.
3. De formule $\phi$ is weliswaar lang, maar kan worden geconstrueerd in hooguit $O(|x|^\ell)$ stappen.
4. Voor een ja-instantie $x$ vergt de executie van $A$ hooguit $N = N(x) \leq c \times |x|^k$ stappen.
5. Een waarmakende waardering voor $\phi$ correspondeert precies met een executie van $A$ die een "ja" produceert. Dus er is een waardering die $\phi$ waarmaakt $\Leftrightarrow$ $x$ is een ja-instantie voor $P$.

# Niet-deterministische Turingmachine

Een NDTM-programma (algemene beschrijving*) bevat:

- $\Gamma$: een eindige verzameling tape-symbolen (waaronder invoeralfabet $\Sigma$ 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: $Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$.
- $\delta \subseteq (Q \setminus \{q_Y, q_N\}) \times \Gamma \times Q \times \Gamma \times \{-1, 0, 1\}$ een transitierelatie die bepaalt wat er kan gebeuren als in een bepaalde toestand een bepaald karakter wordt gelezen.

In de begintoestand staat de invoerstring $x$ op plek 1 tot en met $|x|$ en de rest van de tape is blanco. Het programma start in toestand $q_0$ met de lees- en schrijfkop op positie 1.

*equivalent met ons model, waarin het niet-deterministische gedeelte is samengebracht in Fase 1)

# Bewijs van Cook

Boolese variabelen in $\phi$:

$Q_i^q$ : op tijdstip $i$ is de machine in toestand $q \in Q$; $0 \leq i \leq N$

$H_{ij}$ : op tijdstip $i$ scant de machine cel $j$; $0 \leq i \leq N$, $-N \leq j \leq N$

$S_{ij}^a$ : op tijdstip $i$ bevat cel $j$ symbool $a \in \Gamma$; $0 \leq i \leq N$, $-N \leq j \leq N$

De formule $\phi$ is een conjunctie van:

- $Q_0^{q_0} \wedge H_{01}$

  de machine start in toestand $q_0$, op positie 1 — 2 clausules

- $S_{0j}^{x_j}$, $1 \leq j \leq |x|$ $\quad$ $S_{0j}^{B}$, $-N \leq j \leq 0$ of $|x| < j \leq N$

  $x$ op de posities 1 t/m $|x|$; rest bevat $B$ — $2N + 1$ clausules

# Bewijs van Cook

- $Q_N^{q_Y}$

  op tijdstip $N$ stopt de berekening in de ja-toestand*

- $(\bigvee_{q \in Q} Q_i^q)$, $(\neg Q_i^p \vee \neg Q_i^q)$ $\quad 0 \leq i \leq N$, $p, q \in Q, p \neq q$

  altijd precies één toestand $\quad (N+1)\frac{|Q|(|Q|+1)}{2}$ clausules

- $(\bigvee_{a \in \Gamma} S_{ij}^a)$, $(\neg S_{ij}^a \vee \neg S_{ij}^b)$ $\quad 0 \leq i \leq N, -N \leq j \leq N$, $a, b \in \Gamma, a \neq b$

  cel precies één symbool $\quad (N+1)(2N+1)\frac{|\Gamma|(|\Gamma|+1)}{2}$ clausules

- $(\bigvee_{0 \leq j \leq N} H_{ij})$, $(\neg H_{ij} \vee \neg H_{ik})$ $\quad 0 \leq i \leq N$, $-N \leq j < k \leq N$

  scant precies één cel $\quad (N+1)(1+N(2N+1))$ clausules

*we mogen aannemen dat de machine in $q_Y$ blijft als hij daar al eerder komt

# Bewijs van Cook

- $Q_i^p \wedge H_{ij} \wedge S_{ij}^a \rightarrow \bigvee_{(p,a,q,b,d)\in\delta}(Q_{i+1}^q \wedge H_{i+1,j+d} \wedge S_{i+1,j}^b)$

  elke stap van de machine verloopt volgens de transitierelatie $\delta$
  (nog wel "even" omschrijven naar CNF)

- $S_{ij}^a \wedge \neg H_{ij} \rightarrow S_{i+1,j}^a$

  een cel die op tijdstip $i$ niet wordt gescand, bevat op tijdstip $i + 1$ hetzelfde symbool

Een waarmakende waardering komt zo precies overeen met een echte executie van de niet-deterministische Turingmachine die eindigt in "ja" na een polynomiaal (nl. $N$) aantal stappen.

# “even” omschrijven en tellen

Voor de liefhebber.

$a \wedge b \to c \iff \neg a \vee \neg b \vee c$

$a \wedge b \wedge c \to \bigvee_{i \in I}(d_i \wedge e_i \wedge f_i) \iff (\neg a \vee \neg b \vee \neg c \vee \bigvee_{i \in I} z_i) \wedge \bigwedge_{i \in I}(\neg z_i \vee d_i) \wedge \bigwedge_{i \in I}(\neg z_i \vee e_i) \wedge \bigwedge_{i \in I}(\neg z_i \vee f_i)$ met verse variabelen $z_i$ voor $i \in I$.

Als $a$, $b$ en $c$ alle drie waar zijn, moet er een $z_i$ waar zijn. Vanwege de andere clausules moeten nu $d_i$, $e_i$ en $f_i$ waar zijn.

# Aantal variabelen en clausules

Merk op dat $N \in O(|x|^k)$. Polynomiaal in $N$ impliceert dus polynomiaal in $|x|$.

Verder zijn $Q$, $\Gamma$ en $\delta$ deel van de invoer, dus polynomiaal in $|Q|, |\Gamma|en|\delta|$ is ook polynomiaal in $|x|$.

Verdere details: zie college.

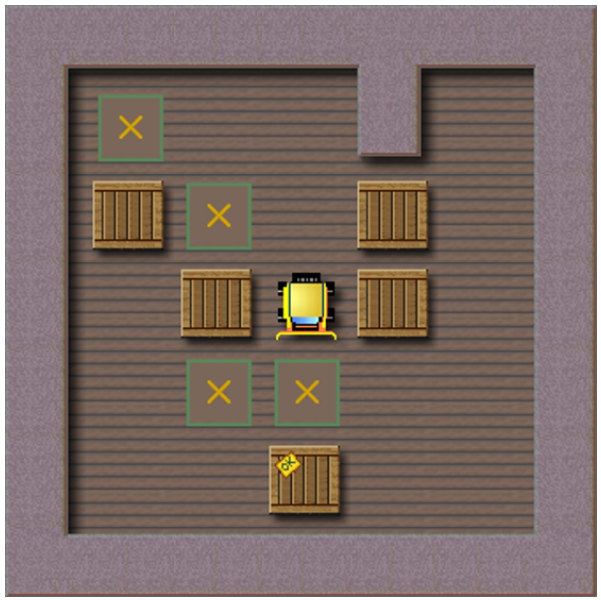
# Reducties naar SAT: Sudoku (statisch)

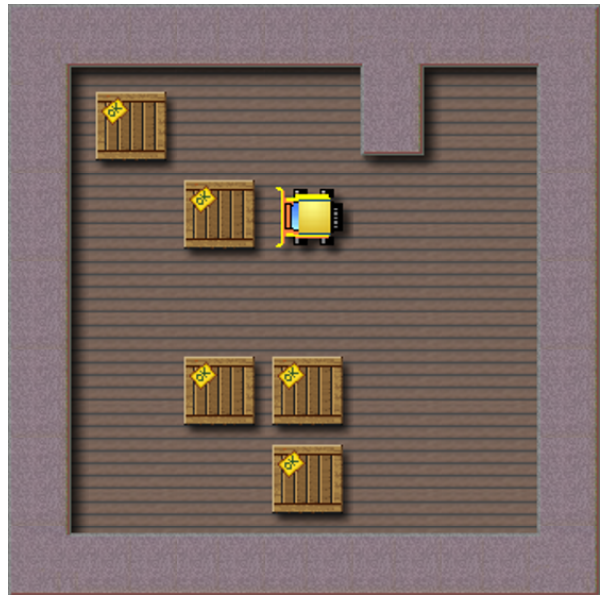
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | 6 | | 9 | | | | | |
| | | | 2 | 5 | | | 4 | |
| | 3 | | 4 | | 8 | 6 | | |
| 4 | | | | | | 5 | | 8 |
| | 8 | | 3 | 4 | 5 | | 9 | |
| 1 | | 9 | | | | | | 4 |
| | | 6 | 7 | | 3 | | 2 | |
| | 1 | | | 9 | 2 | | | |
| | | | | | 4 | | 7 | 5 |

Variabelen: $x_{ij}^{k} \iff$ vakje $(i, j)$ bevat cijfer $k$

- elk vakje bevat precies één cijfer
- rijen en kolommen mogen geen dubbelen bevatten
- de negen groepjes van $3 \times 3$ mogen geen dubbelen bevatten
- definieer een beginconfiguratie

# Reducties naar SAT: Sokoban (dynamisch)





Idee (cf. reductie van Cook):

- definieer begintoestand en eindtoestand
- definieer geldige transitie tussen toestanden
- gebruik een of andere bovengrens voor het aantal stappen
- `https://cspsat.gitlab.io/copris-puzzles/sokoban/`

# SAT-solvers

SAT-solvers zijn algoritmen of programma's die SAT-instanties oplossen, d.w.z., bepalen of ze ja- of nee-instanties zijn.

Een triviaal algoritme (exhaustive search) is te geven in $O(2^n \times |x|)$. Maar: kan het beter?

Handbook of Satisfiability (2009). Armin Biere, Marijn Heule, Hans van Maaren en Toby Walsh.
`https://books.google.nl/books?isbn=160750376X`

# DPLL

Martin Davis, Hilary Putnam, George Logemann, Donald W. Loveland (1960/1961)

- Een van de meest bekende algoritmen, nog steeds de basis voor vele moderne
- Backtracking
- Kiest een literal, probeert een waardering, propageert de waarde door de formule en kijkt recursief of de vereenvoudige formule is waar te maken. Zo nee, probeer de andere waardering.
- *Unit propagation*: als een clausule nog maar één literal heeft, staat de waarde daarvan vast
- *Pure literal elimination*: literals die maar op één manier voorkomen krijgen de waarde die die waar maakt

# Niet-chronologisch backtracken

- DPLL backtrackt *chronologisch*: bij het vinden van een tegenspraak, gaat het algoritme één stap terug in de zoekboom.
- Als die voorgaande stap echter niet relevant was voor de gevonden tegenspraak, is het onnodig om daarvan de andere waardering ook te proberen. Dat kan handiger.
- *Niet-chronologisch* backtracken is hierop een verbetering: bij het vinden van een tegenspraak, kijkt het algoritme welke vorige stappen deze tegenspraak hebben veroorzaakt. Dan wordt teruggesprongen naar de laatste stap uit deze verzameling.

# Conflict-driven clause learning

- Conflict-driven clause learning (CDCL) is hierop nog een verbetering.
- Bij het vinden van een tegenspraak, kijkt het algoritme welke vorige stappen deze tegenspraak hebben veroorzaakt. Hieruit wordt een nieuwe clausule afgeleid en toegevoegd aan het probleem, om die fout voortaan direct te voorkomen.
- Vervolgens wordt doorgaans teruggesprongen naar de laatste stap waarin *unit propagation* iets doet met deze nieuwe clausule.
- Soms kunnen toegevoegde clausules op een later moment weer worden verwijderd.

# SAT-conferenties en -wedstrijden


`http://satisfiability.org/SAT23/`


`https://satcompetition.github.io/2023/`

# Ondergrens SAT?

Gigantische verbeteringen, zowel algoritmisch als implementatietechnisch, maar:

Wat is de ondergrens?

# Even tussendoor: niet-polynomiaal en exponentieel

Polynomiaal: $O(n^k)$ ($k \geq 0$)

Exponentieel: $O(c^n)$ ($c > 1$)

Als $\mathcal{P} \neq \mathcal{NP}$: NP-volledige problemen niet polynomiaal oplosbaar.
Wil echter niet zeggen dat ze per se exponentieel zijn.

Quasi-polynomiaal: groter dan polynomiaal, maar nog niet exponentieel. $2^{O(\log^c n)}$ (constante $c > 0$)
Bijvoorbeeld: graafisomorfisme (Babai, 2017)

Subexponentieel: meerdere definities, maar bijv. $2^{O(\sqrt{n})}$.

# Exponential Time Hypothesis

Russell Impagliazzo en Ramamohan Paturi (1999/2001):

Hypothese (Exponential Time Hypothesis (ETH))
Er bestaat een constante $c > 1$ zodat 3SAT $\in \Omega(c^n)$.

M.a.w.: 3SAT heeft in de worst case een exponentiële ondergrens.

Hypothese (Strong Exponential Time Hypothesis (SETH))
Voor elke constante $c < 2$ bestaat er een $k$ zodat $k$SAT $\in \Omega(c^n)$.

M.a.w.: er bestaat (asymptotisch gezien) geen beter algoritme voor algemene SAT-instanties dan exhaustive search.

SETH $\rightarrow$ ETH $\rightarrow \mathcal{P} \neq \mathcal{NP}$, maar ook interessante gevolgen als ¬SETH of ¬ETH

Hypotheses niet algemeen aangenomen als waar; beide nog niet bewezen of ontkracht.

# Vandaag gezien

- het basisgeval in $\mathcal{NPC}$-bewijzen: de stelling van Cook–Levin
  - reductie van arbitrair $\mathcal{NP}$-probleem naar SAT
  - encodeer elke instantie van het probleem in (groot maar) polynomiaal begrensde logische formule
- SAT-solvers: slimme technieken om dit probleem op te lossen
  - DPLL: backtracking met slimme extra's (unit propagation, pure literal elimination)
  - conflict-driven clause learning (CDCL): leer uit tegenspraken en voeg gaandeweg nieuwe clausules toe
- Exponential Time Hypothesis: hypothese dat 3SAT een exponentiële ondergrens heeft in de worst case
- Strong Exponential Time Hypothesis: hypothese dat SAT algemeen niet beter kan dan $2^n$

# (Werk)college

**Volgende college**: dinsdag 2 mei, 09u00–10u45, zaal 407-409 (Snellius)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304 en 303 (Snellius, onderaan de trap)
Opgaven uit het dictaat: 66, 67, 68

# De komende weken

- 2 mei: herhaling
  - hoorcollege: korte samenvatting semester, uitgebreider waar gewenst
  - werkcollege: representatieve opgaven uit het hele semester
- 9 mei
  - hoorcollege: gastcollege Hendrik Jan Hoogeboom, verdere complexiteitsklassen en spellen
  - werkcollege: huiswerk 3 en desgewenst herhaling
- 16 mei: niets
- 23 mei: oefententamen (hoorcollege), geen werkcollege
- 5 juni: tentamen

Tentamenstof komt binnenkort duidelijker op de website.

# (Werk)college

**Volgende college**: dinsdag 9 mei, 09u00–10u45, zaal 407-409 (Snellius)

**Werkcollege**: zodadelijk van 11u00 tot 12u45, computerzalen 302–304 en 303 (Snellius, onderaan de trap)
Opgaven uit het dictaat: 21, 26, 28, 36, 60, 68

# Huiswerk

**Huiswerk 3**

Deadline is dinsdag 9 mei (23u59), kan tijdens dat werkcollege aan worden gewerkt.