Vierde college algoritmiek

27 februari 2015

Complexiteit, Brute Force en Exhaustive Search

Analyseren van algoritmen:

- Hoe werkt het algoritme?
- Werkt het op alle toegestane invoer correct?
- Hoeveel resources gebruikt het algoritme?
  - Processortijd
  - Geheugen

Uiteraard hangt de doorlooptijd van een algoritme af van de invoer. Sommige invoeren kosten meer tijd dan andere. Dit kan verschillende oorzaken hebben. Stel dat we een algoritme hebben dat controleert of een array allemaal verschillende getallen bevat. Dan hangt de doorlooptijd o.a. af van:

- De grootte van de invoer: 3 of 300 getallen

- De instantie (soort invoer): 5,5,5,5,5 of 1,2,3,4,5

Vaak zijn we geïnteresseerd in de orde van grootte van de looptijd: hoe neemt deze toe voor grotere invoer (zeg, langere rijtjes)?

We kunnen een algoritme testen door de doorlooptijd voor een groot aantal instanties te meten en uit te zetten tegen de invoergrootte n

Wat zijn nadelen van een dergelijke benadering?

Een andere methode om de looptijd van een algoritme te analyseren is om de tijdcomplexiteit van dat algoritme te bepalen: een functie die voor een algoritme bepaalt hoeveel werk er gedaan moet worden voor invoer ter grootte n in

- het slechtst mogelijke scenario
- het best mogelijk scenario
- het gemiddelde scenario (we praten dan over de te verwachte hoeveelheid werk; gemiddeld over alle mogelijke invoer)

Complexiteit (= tijdcomplexiteit) van een algoritme:

- = hoeveelheid werk verricht door het algoritme
- hangt meestal af van de grootte van de invoer: hoe groter de invoer, hoe groter de complexiteit
- wordt bepaald door het aantal keer dat een zekere basisoperatie wordt uitgevoerd
- het belangrijkst is de (asymptotische) groei
- wordt vaak uitgedrukt in O-notatie (orde van grootte)
- hangt vaak ook af van het soort invoer: worst case, best case, average case

Voorbeld 1:

```
// invoer: array a[0···n−1] bestaande uit n reals
// uitvoer: de grootste waarde

    max := a[0];
    for i := 1 to n − 1 do
        if ( a[i] > max ) then ←− basisoperatie
            max := a[i];
        fi
    od
    return max;
```

Complexiteit: $C(n) = n - 1 \in \Theta(n)$

Voorbeeld 2:

```
// invoer: array a[0···n−1] bestaande uit n reals
// uitvoer: true als alle n waarden verschillen
```

for $i := 0$ **to** $n-2$ **do**
  for $j := i+1$ **to** $n-1$ **do**
    if ($a[i] = a[j]$) **then** $\longleftarrow$ basisoperatie
      return false; **fi**
  od
od
return true;

Best case complexiteit: $B(n) = 1 \in \Theta(1)$

Worst case complexiteit:

$$W(n) = \sum_{i=0}^{n-2} (n-1-i) = \frac{1}{2} n(n-1) \in \Theta(n^2)$$

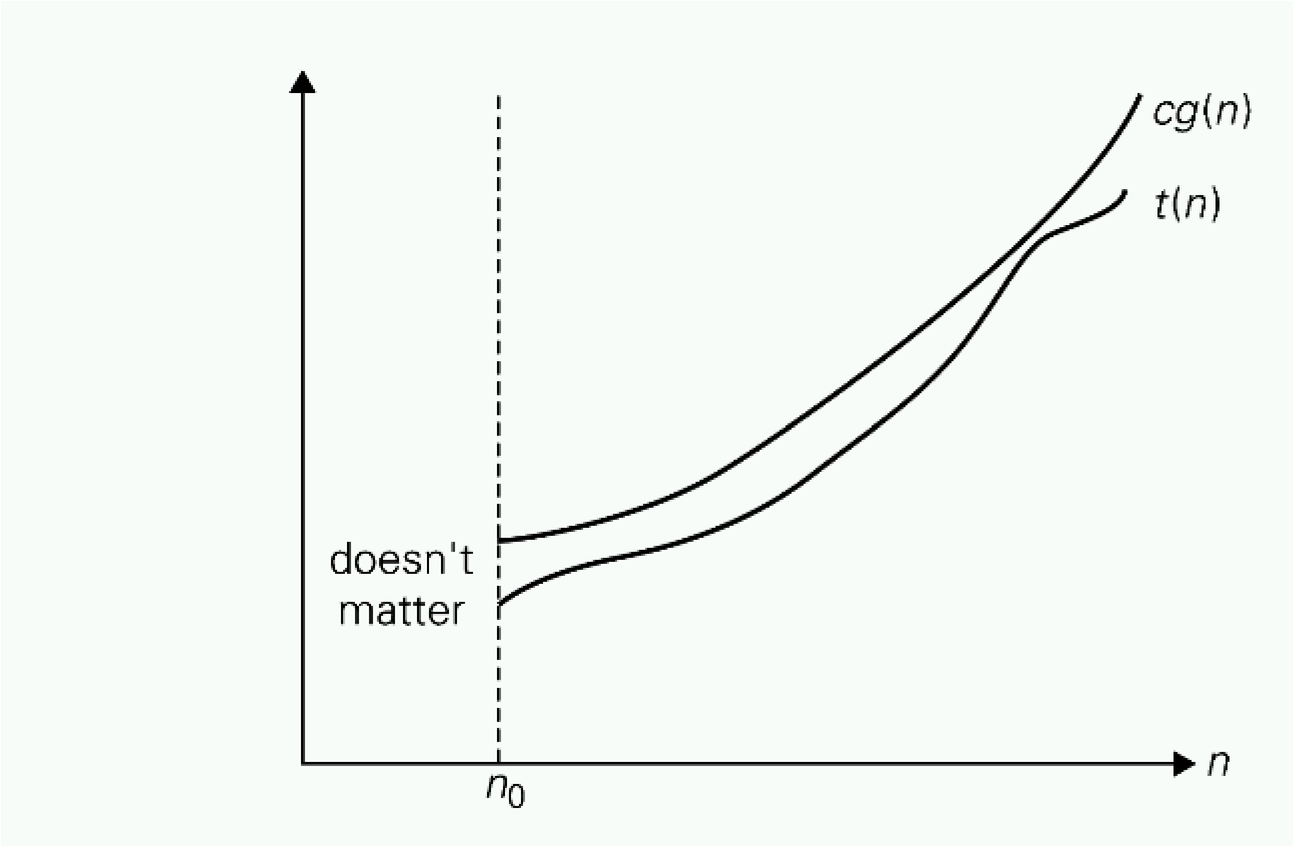
O-notatie beschrijft het asymptotisch gedrag:

$f \in O(g) \iff \exists c > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:
$f(n) \leq c \cdot g(n)$

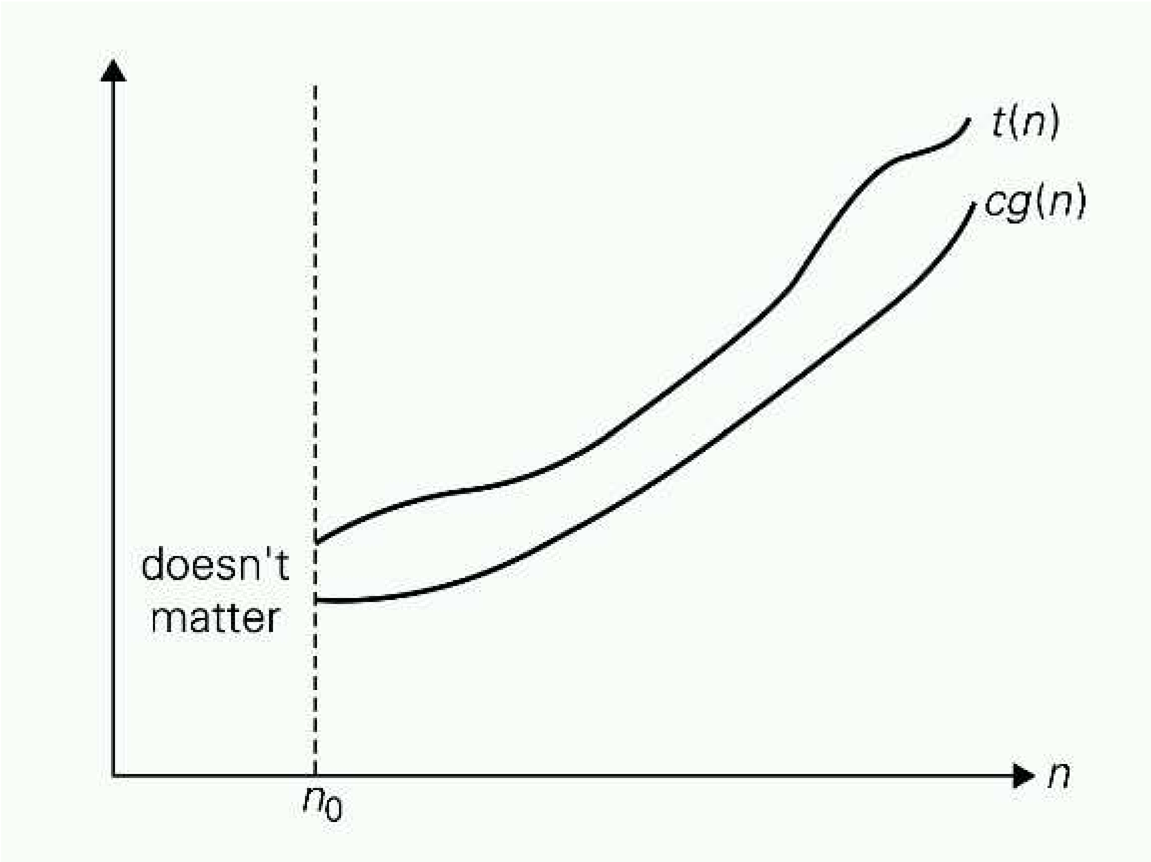
$f \in \Omega(g) \iff \exists c' > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:
$f(n) \geq c' \cdot g(n)$

$f \in \Theta(g) \iff \exists c_1, c_2 > 0$ en $\exists n_0 \geq 0$ zodat $\forall n > n_0$:
$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$

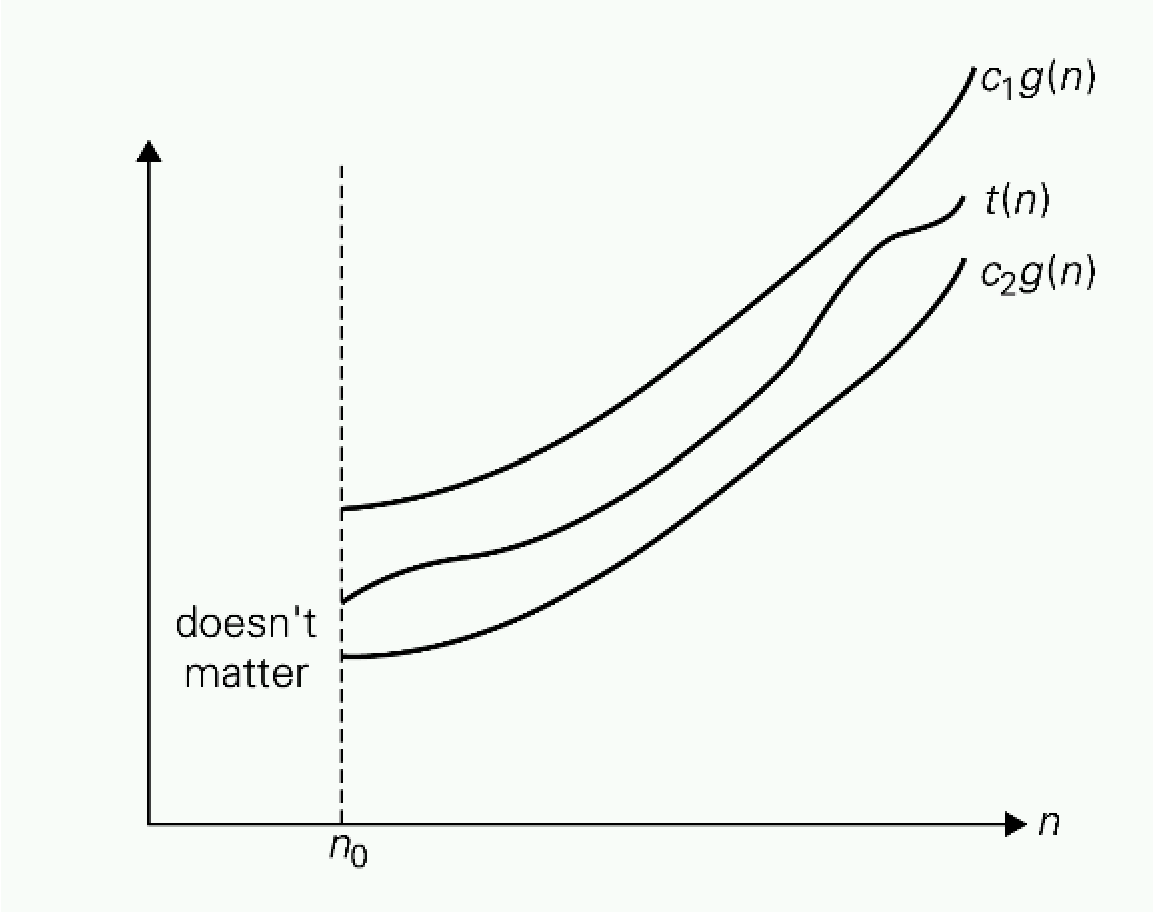
Aangezien zowel f als g in onze toepassingen de complexiteit van een algoritme voorstelt is hier steeds $f > 0$ en $g > 0$.



$t \in O(g)$: $t(n)$ groeit niet sneller dan $g(n)$



$t \in \Omega(g)$: $t(n)$ groeit minstens zo snel als $g(n)$



$t \in \Theta(g)$: $t(n)$ groeit even snel als $g(n)$

Stelling:

(1) $$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \iff f \in \Theta(g)$$

f en g zijn van dezelfde orde (groeien even snel)

(2) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f \in O(g),\ \text{maar } f \notin \Theta(g)$$

f gaat echt minder snel naar ∞ dan g

(3) $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff f \in \Omega(f) \text{ maar } f \notin \Theta(g)$$

f gaat echt sneller naar ∞ dan g

Merk op dat geldt: $g \in O(f) \iff f \in \Omega(g)$

N	10	50	100	300	1000
$\log_2 N$	3	5	6	8	9
$5N$	50	250	500	1500	5000
$N \cdot \log_2 N$	33	282	665	2469	9966
N^2	100	2500	10.000	90.000	7 cijfers
N^3	1000	125.000	7 cijfers	8 cijfers	10 cijfers
2^N	1024	16 cijfers	31 cijfers	91 cijfers	302 cijfers
$N!$	7 cijfers	65 cijfers	161 cijfers	623 cijfers	onvoorstelbaar
N^N	11 cijfers	85 cijfers	201 cijfers	744 cijfers	onvoorstelbaar

Ter vergelijking:

het aantal protonen in het heelal is een getal met 79 cijfers
het aantal microseconden sinds de Big Bang heeft 24 cijfers

(1) $n(n-2) \in O(n^3)$; $n(n-2) \in O(n^2)$; $n(n-2) \in \Omega(n^2)$

(2) $n \in O(n^2)$, maar NIET $n \in \Theta(n^2)$

(3) $2^n \in O(3^n)$, maar NIET $2^n \in \Theta(3^n)$

(4) $(n^2+1)^{10} \in \Theta(n^{20})$

(5) $\sqrt{10n^2+7n+3} \in \Theta(n)$

(6) $2n \log_2(n+2)^2 + (n+2)^2 \log_2(n/2) \in \Theta(n^2 \log_2 n)$

(7) $2^{n+1} + 3^{n-1} \in \Theta(3^n)$

(8) $\lfloor \log_2 n \rfloor \in \Theta(\log_2 n)$; $\log_{10} n \in \Theta(\log_2 n)$

(9) $2^n \in O(n!)$, maar NIET $2^n \in \Theta(n!)$

De volgende naamgeving wordt meestal gehanteerd:

1	constant
$\log n$	logaritmisch
n	lineair
$n \log n$	n-log-n
n^2	kwadratisch
n^α $(\alpha > 1)$	polynomiaal
2^n	exponentieel
$n!$, n^n, ...	superexponentieel

Voorbeeld 3:

```
// invoer: array a[0···n−1] bestaande uit n reals en
//         een reeel getal k
// uitvoer: index i waarvoor a[i] = k; −1 als deze niet
//          bestaat
```

basisoperatie

$i := 0$
while ($i < n$ **and** $a[i] \neq k$) **do**
    $i := i + 1$; **od**
if ($i < n$)
    return i; **fi**
return -1;

best case/worst case/average case: $\cdots$

Recursief algoritme voor de berekening van $n!$:

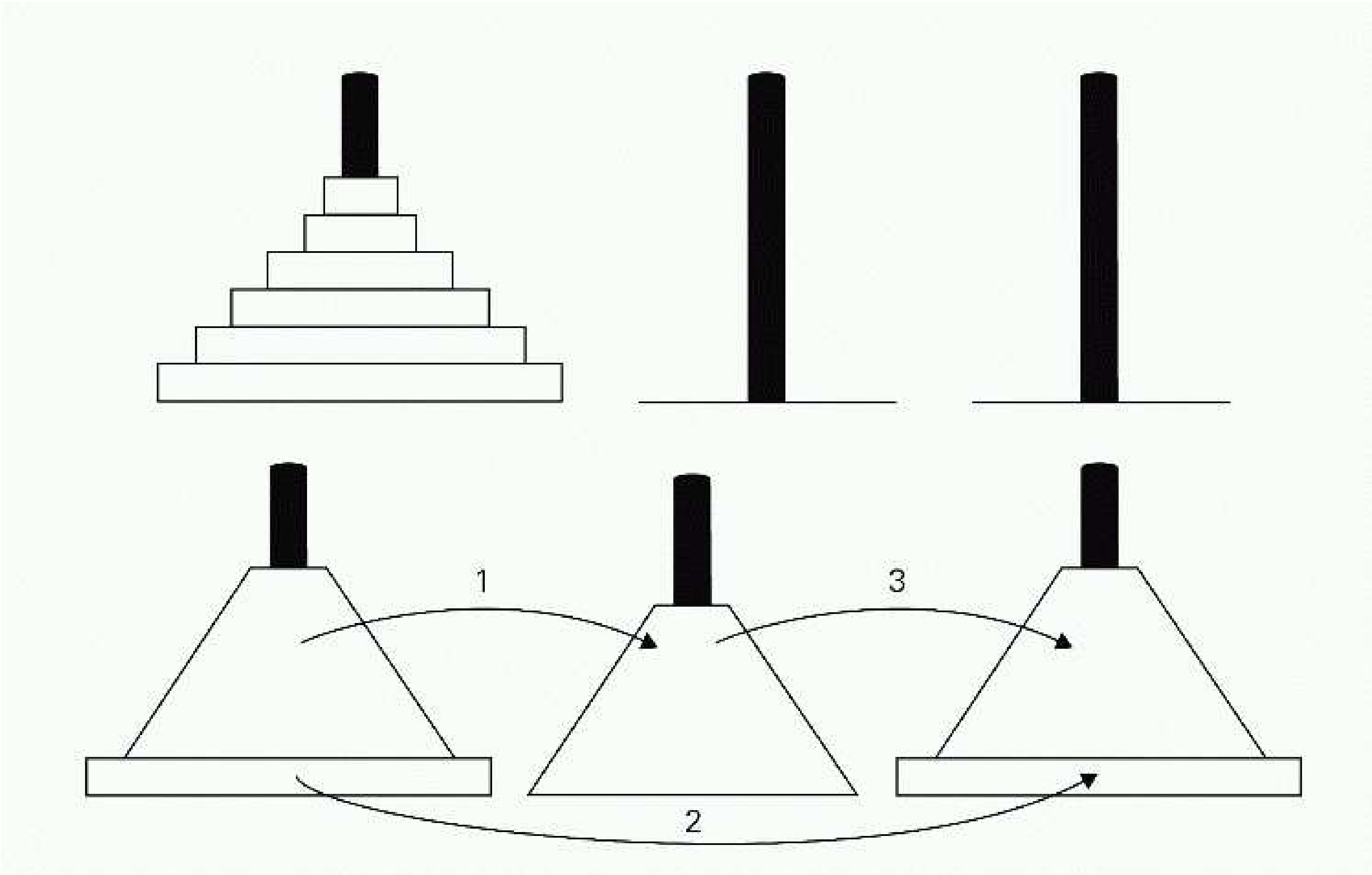
```
int faculteit ( int n ) { // gebruikt: n! = n·(n−1)!
  if ( n == 0 )
     return 1;
  else                      basisoperatie
     return n*faculteit(n-1);
}  // faculteit
```

$M(n)$ = aantal vermenigvuldigingen (= aantal recursieve aanroepen - 1) voldoet aan de recurrente betrekking:

$$\begin{cases} M(0) = 0 \\ M(n) = M(n-1) + 1 \quad \text{voor } n > 0 \end{cases}$$

Oplossing: $M(n) = n \longrightarrow$ complexiteit `faculteit` is $\Theta(n)$.

Oplossing Hanoi algemeen (*)



Recursieve oplossing van de Torens van Hanoi

Een recursief algoritme voor het probleem van de Torens van Hanoi (zie Programmeermethoden):

```
// zet toren van n stuks (optimaal) van a naar b via c
// print de zetten
void zet (int n, int a, int b, int c) {
  if ( n > 0 ) {
    zet (n-1, a, c, b);
    cout << "zet van " << a << "naar " << b << endl;
    zet (n-1, c, b, a);
  }  // if
}  // zet
```

- n (het aantal schijven) is een maat voor de grootte van de invoer

- het verzetten van een schijf is de basisoperatie

Laat $M(n)$ = aantal zetten (= aantal recursieve aanroepen - 1), dan voldoet $M(n)$ aan de recurrente betrekking:

$$\begin{cases} M(0) = 0 \\ M(n) = 2M(n-1) + 1 \quad \text{voor } n > 0 \end{cases}$$

Oplossing (zie college of Levitin 2.4):

$M(n) = 2^n - 1 \longrightarrow$ complexiteit `zet` is $\Theta(2^n)$.

Brute force: a straightforward approach, usually directly based on the problem statement and definitions.

Ofwel: los een probleem op via de meest voor de hand liggende (recht-toe-recht-aan) methode, meestal door eenvoudigweg de definitie van een oplossing te gebruiken. Vaak ook: alle mogelijkheden proberen.

Voorbeeld 1: vind de grootste gemene deler van twee getallen m en n door van alle mogelijke integers ≥ 2 (en $\leq \min(m, n)$) te proberen of ze zowel m als n delen. (Zie college 1.)

Voorbeeld 2: los de DONALD + GERALD = ROBERT puzzel op door alle 9! (er was al gegeven dat D = 5) mogelijke antwoorden te proberen. (Zie college 1.)

Voorbeeld 3: zoek een gegeven X in een array van n stuks door er van links naar rechts doorheen te lopen en X met alle n te vergelijken.

Zoek herhaald de volgende in grootte (hier de kleinste van de rest) en zet die op de juiste positie in het array. Dit sorteert het array oplopend.

for $i := 0$ **to** $n-2$ **do**
  $\texttt{min} := i$;
  for $j := i+1$ **to** $n-1$ **do**
    if $A[j] < A[\texttt{min}]$ **then**
      $\texttt{min} := j$;
    fi
  od
  wissel$(A[i], A[\texttt{min}])$;
od

Aantal vergelijkingen: $\frac{1}{2}n(n-1)$.

Probleem: bereken de waarde van het polynoom $p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ in het punt $x = x_0$ (Levitin, opgave 3.1.4)

Brute force algoritme (uit de definitie):

```
p := 0;
for i := n downto 0 do
    macht := 1;
    for j := 1 to i do
        macht := macht * x; // bereken x^i
    od
    p := p + a[i] * macht;
od
return p;
```

Complexiteit (aantal $*/+$): $\Theta(n^2)$

Polynoomevaluatie: slimmer

Slimmer: we kunnen de efficiëntie eenvoudig flink verbeteren door van rechts naar links te evalueren en de x^i handiger te berekenen:

$p := a[0]$;
`macht` $:= 1$;
for $i := 1$ **to** n **do**
  `macht` $:=$ `macht` $* \; x$;
  $p := p + a[i] \; *$ `macht`;
od
return p;

Complexiteit: $\Theta(n)$;
Preciezer: $\#(*) = 2$n; $\#(+) = n$

Dit kan nog beter (methode van Horner), echter niet in orde van grootte.

Gegeven een patroon (= string van m karakters) en een tekst (= string van $n \geq m$ karakters). **Gevraagd** de index van de beginpositie in de tekst waar het patroon voorkomt.

Brute force algoritme: patroon v.l.n.r. langs de tekst schuiven en steeds de overeenkomstige karakters uit tekst en patroon vergelijken

```
for i := 0 to n − m do
    j := 0;
    while j < m and patroon[j] = tekst[i + j] do
        j := j + 1;
    od
    if j = m then
        return i;
    fi
od
return −1; // geen match gevonden
```

De werking van het algoritme geïllustreerd aan de hand van het volgende voorbeeld:

N	O	B	O	D	Y	-	N	O	T	I	C	E	D	-	H	I	M
N	O	T															
	N	O	T														
		N	O	T													
			N	O	T												
				N	O	T											
					N	O	T										
						N	O	T									
							N	O	T								

Het aantal vergelijkingen dat dit algoritme doet hangt af van de tekst en het patroon. In de worst case worden $m*(n-m+1)$ vergelijkingen gedaan. Dit komt voor wanneer in elke i-stap het patroon helemaal (dus m vergelijkingen) vergeleken wordt met de tekst. De (worst case) complexiteit van het algoritme is dus $O(n * m)$.

Opgave: geef een tekst en een patroon waarvoor het algoritme $m * (n - m + 1)$ vergelijkingen doet.

Opmerking: het kan beter (Boyer-Moore, Knuth-Morris-Pratt), maar voor "gewone-taal" teksten is het algoritme zo slecht nog niet.

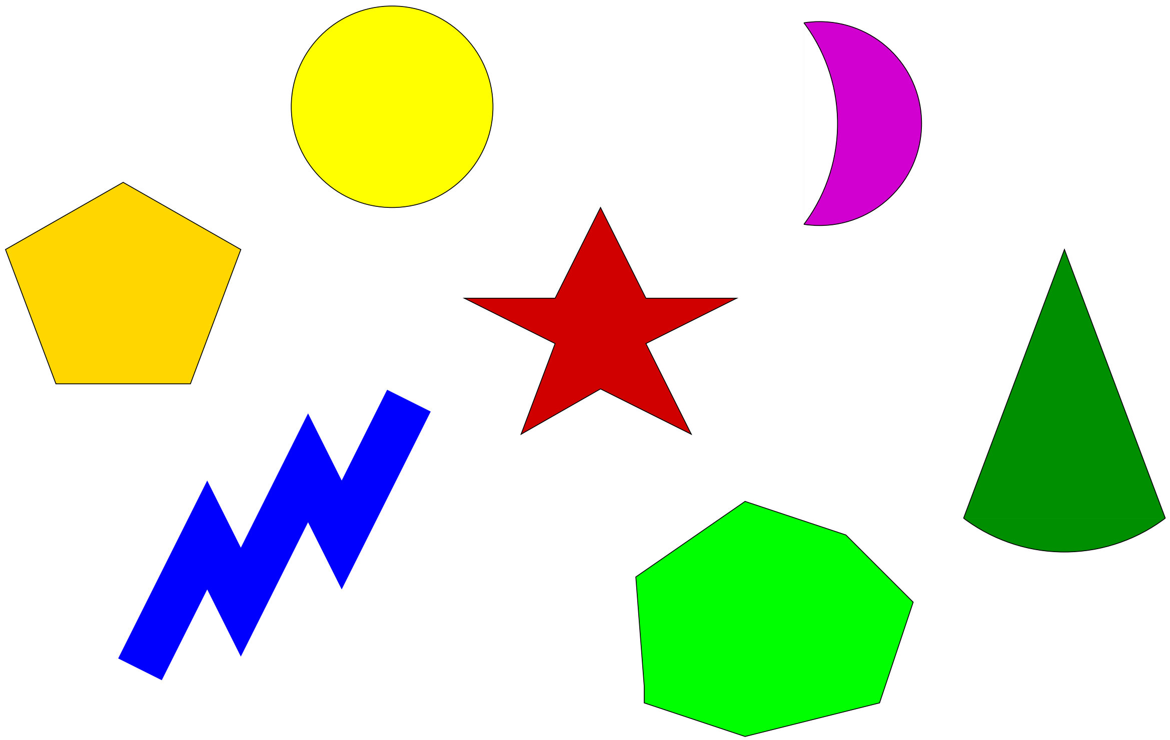
Gegeven n punten $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$.
Gevraagd het/een tweetal punten dat het dichtst bij elkaar ligt. Afstandsmaat: $\mathrm{d}(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Brute force algoritme: alle paren (p_i, p_j) (met $i < j$) aflopen en hun onderlinge afstanden $\mathrm{d}(p_i, p_j)$ vergelijken.

$\texttt{dmin} := \infty$;
for $i := 1$ **to** $n - 1$ **do**
    for $j := i + 1$ **to** n **do**
        $d := (x_i - x_j)^2 + (y_i - y_j)^2$;
        if $d < \texttt{dmin}$
            $\texttt{dmin} := d; k := i; l := j$;
        fi // (p_k, p_l) voorlopig closest pair
    od
od

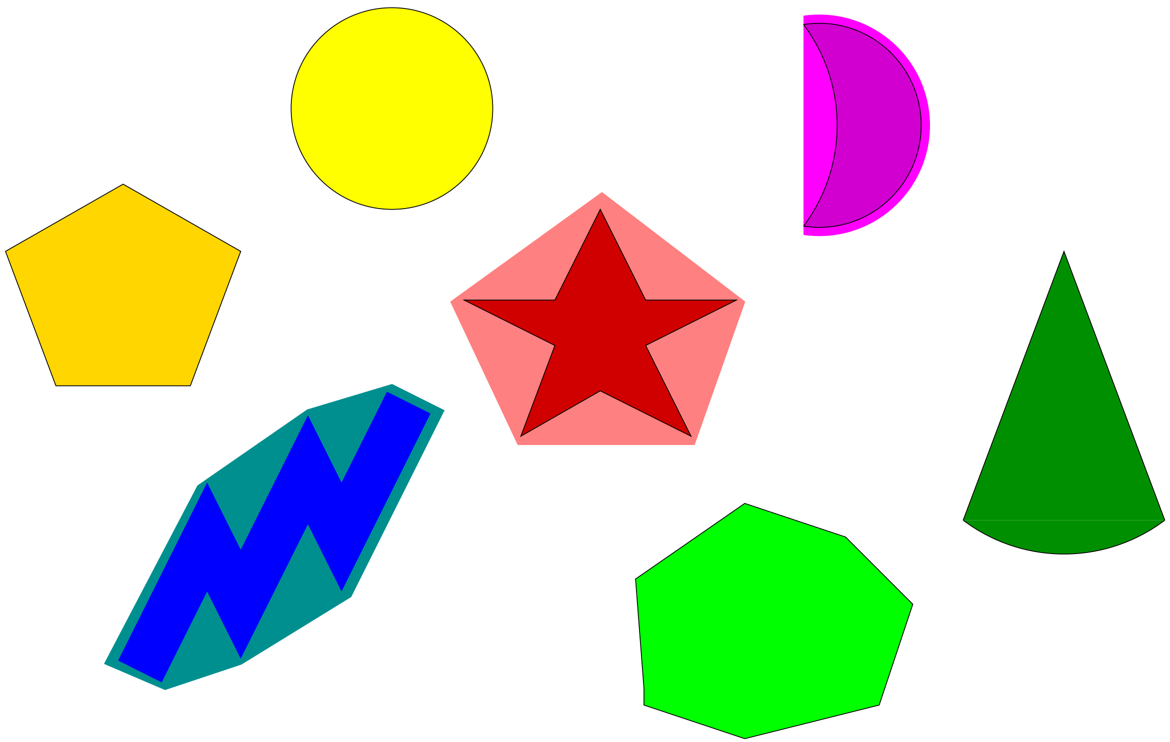
Complexiteit: $\frac{1}{2}n(n-1) \in \Theta(n^2)$

Een verzameling punten in het platte vlak heet convex als voor elk tweetal punten uit die verzameling geldt dat het verbindend lijnstuk ook weer in die verzameling ligt.



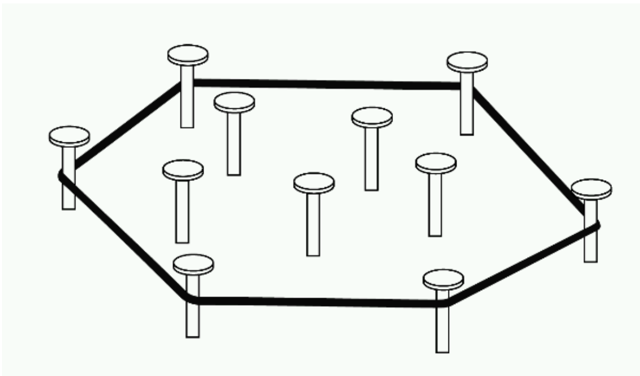
Convexe en niet-convexe vormen

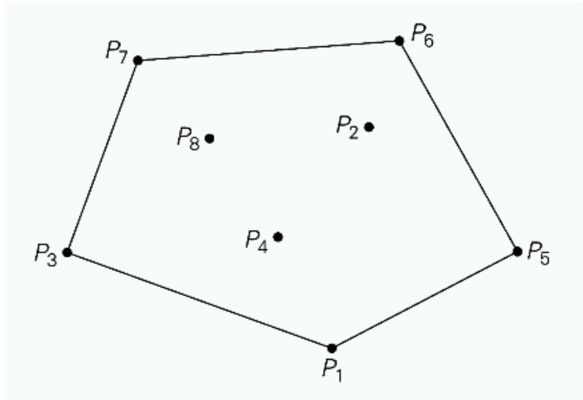
De convex hull (convex omhulsel) van een verzameling S van punten in $\mathbf{R}^2$ is de kleinste convexe verzameling die S bevat.



Convexe omhulsels

Stelling: de convex hull van een verzameling S van $n > 2$ punten in $\mathbf{R}^2$ (niet alle op één lijn) is een convexe veelhoek (polygoon) met als hoekpunten enkele punten uit S.





De convex hull van de verzameling $\{P_1, P_2, \ldots, P_8\}$ is de convexe veelhoek met hoekpunten P_1, P_5, P_6, P_7 en P_3

We baseren ons brute force algoritme op de volgende observatie: een lijnstuk P_iP_j maakt deel uit van de rand van de convex hull van $\{P_1, P_2, \ldots, P_8\}$ d.e.s.d.a. alle andere punten van de verzameling aan een en dezelfde kant van de lijn door P_i en P_j liggen.

Brute force: Ga *voor elk tweetal* punten $P_i = (x_i, y_i)$ en $P_j = (x_j, y_j)$ na of *alle andere punten* aan dezelfde kant van de lijn $(y_j - y_i)x + (x_i - x_j)y = x_iy_j - y_ix_j$ liggen. Zo ja, dan is P_iP_j dus deel van de convex hull.

Complexiteit: $O(n^3)$

Opmerking: het kan veel beter, namelijk $O(n \lg n)$

Brute Force: los een probleem op via de meest voor de hand liggende (recht-toe-recht-aan) methode, meestal door eenvoudigweg de definitie van een oplossing te gebruiken.

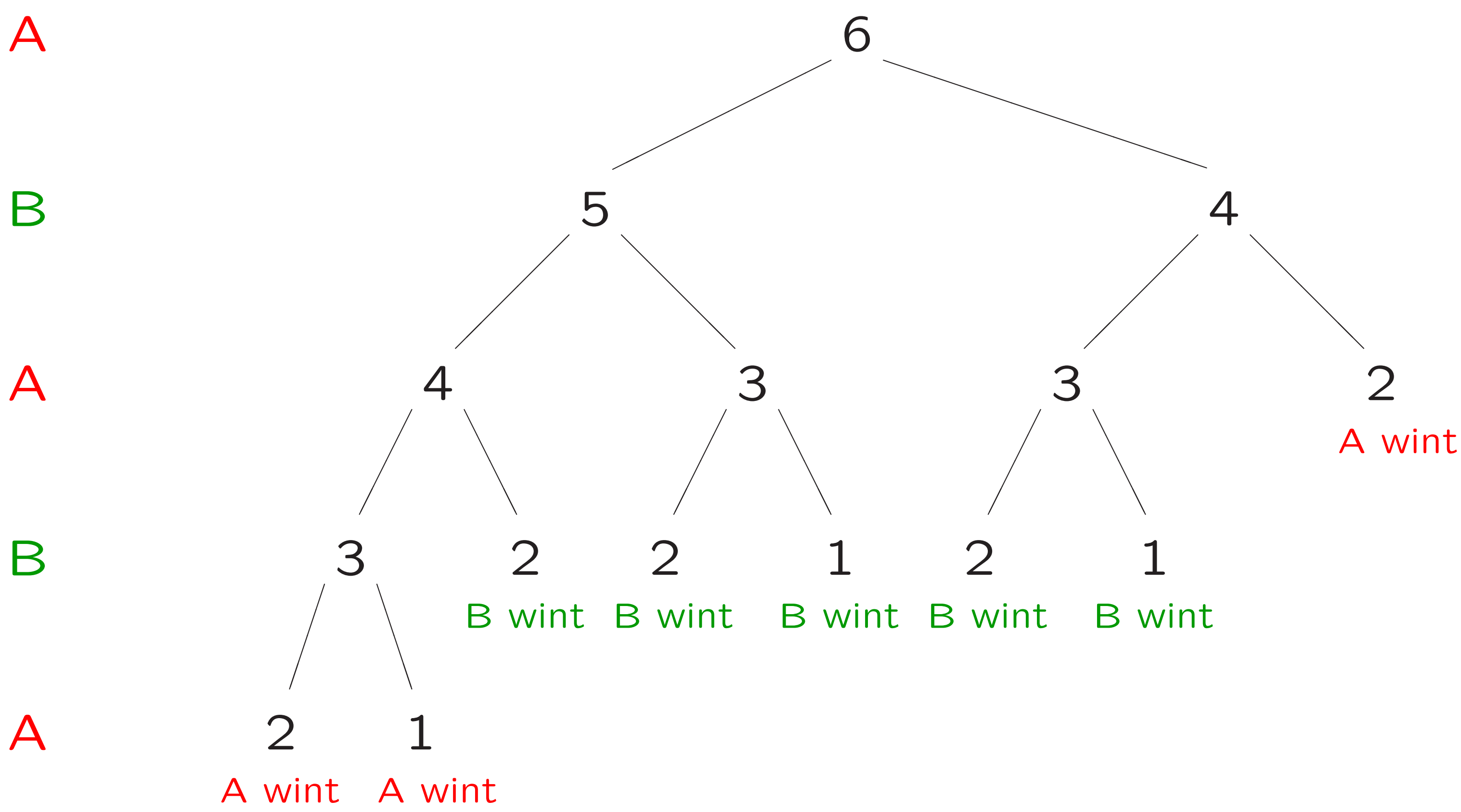
Bekeken voorbeelden:

- Lineair zoeken
- Bepaling van de ggd van twee getallen
- Selection sort
- Polynoomevaluatie
- Patroonherkenning
- Closest Pair
- Convex hull

Brute force:

- **Voordelen**:
  - algemeen toepasbaar
  - eenvoudig
  - levert voor een aantal belangrijke problemen (zoeken, patroonherkenning) een zeer behoorlijk algoritme op

- **Nadelen**:
  - levert meestal geen efficiënt algoritme op
  - soms onacceptabel langzaam



Brute Force: doorloop (*als het ware*) de hele spelboom om te bepalen of een stand winnend is. Alle toestanden/alle spelverlopen worden zo bekeken. Aldus alle mogelijkheden onderzoeken heet ook wel exhaustive search.

Exhaustive search: brute force benadering voor problemen die te maken hebben met het vinden van een element met een speciale eigenschap binnen een verzameling van bijv. permutaties of deelverzamelingen of toestanden of ...

Methode:

. construeer op een systematische manier alle kandidaat-oplossingen, bijvoorbeeld alle permutaties van de getallen 1 t/m n

. evalueer elk van deze mogelijke oplossingen

. retourneer de kandidaatoplossing met de gevraagde eigenschap (als die bestaat) (∗)

(∗) soms, zoals bij optimalisatieproblemen, *moet* je daartoe alle kandidaatoplossingen gezien hebben

Belangrijke observatie bij het analyseren van een 2persoonsspel, waarvan je wilt weten of het winnend is voor de speler die begint (en wat een winnende strategie is):

> een stand is winnend voor degene die aan de beurt is, dan en slechts dan als ten minste één van zijn directe vervolgstanden niet winnend is voor de tegenstander

$\Longrightarrow$ RECURSIE

```
winnend(stand)::
   if eindstand(stand) then
      //  makkelijk; bijv return false;
   else
      for alle mogelijke zetten i do
      // met terugzetten !
          doezet(stand,i);
          if not winnend(stand) then
              neemzetterug(stand,i);
              return true;
          fi
          neemzetterug(stand,i);
      od
      return false;
   fi
```

Doe een zet; bekijk de vervolgstand; neem de zet terug

```
bool nimwinst (int stand) {
    int lucifer;
    if ( stand == 0 )  // tegenstander heeft laatste lucifers gepakt
        return false;
    else {
    // directe vervolgstanden aflopen
        for ( lucifer = 1; lucifer <= 2; lucifer++ ) {
             stand -= lucifer; // doe een zet
             if ( !nimwinst (stand) ) {
                 stand += lucifer; // terugzetten
                 return true;
             }
             stand += lucifer; // terugzetten
        }
        return false;
    }  // else
}
```

Je kunt ook telkens een kopie maken waarin je de zetten doet (zie college 3)

- **Lezen/leren bij dit college:**
  Paragraaf 2.1–3, 2.6–7 (lezen), 3 inl., 3.1–3

- **Werkcollege** programmeeropdracht 1:
  donderdag 5 maart 2015, 13:45–15:30, zaal 306/308, 302/304
  deadline maandag 23 maart, 12:00 uur

- **Eerste vragenuur** programmeeropdracht 1:
  donderdag 6 maart, 15:30–17:00, zaal 306/308

- **Volgend college:**
  vrijdag 6 maart 2015, 11:15–13:00, zaal B2