Eerste college complexiteit

4 februari 2008

Introductie

- docent: dr. J.M. (Jeannette) de Graaf
- werkgroepleider: Thijs van Ommen
- website: http://www.liacs.nl/home/graaf/COMP/
- college: dinsdag 11.15 – 13.00 in zaal 174
- werkcollege: maandag 11.15 – 13.00 in zaal 174

- eerste college: maandag 4 februari 2008
- tweede college: dinsdag 5 februari 2008
- eerste werkcollege: maandag 11 februari 2008
- tentamen: donderdag 5 juni, 14.00–17.00
- hertentamen: dinsdag 12 augustus, 10.00–13.00

Huiswerk:

- Er komen drie of vier huiswerkopgaven
- Deze dienen individueel ingeleverd te worden
- Samenwerken mag (met mate), overschrijven niet
- Voor elke huiswerkopgave krijg je een cijfer
- Het gemiddelde hiervan wordt opgeteld bij het aantal punten dat je voor het tentamen haalt
- Delen door 10 levert je eindcijfer

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

Opmerking: we zullen hier altijd sequentiële algoritmen bekijken.
Vraag: hoe meten we de hoeveelheid werk die een algoritme doet?

Complexiteit = tijdcomplexiteit = hoeveelheid werk

- een maat voor de hoeveelheid werk moet iets vertellen over de efficiëntie van de methode

- die maat moet onafhankelijk zijn van de gebruikte computer, programmeertaal, implementatiedetails etc.

- de complexiteit hangt gewoonlijk af van de grootte van de invoer: hoe groter de invoer, hoe hoger de complexiteit

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 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;
```

(1)	$max := A[1];$	1 x
(2)	$index := 2;$	1 x
(3)	**while** $index \leq n$ **do**	n x
(4)	**if** $max < A[index]$ **then**	$n-1$ x
(5)	$max := A[index];$	$\leq n-1$ x
(6)	**fi**	
(7)	$index := index + 1;$	$n-1$ x
(8)	**od**	
(9)	**return** $max;$	1 x
		$(\leq)\ 4n$

aantal vergelijkingen $(max < A[index]) = n - 1 \approx n =$ aantal vergelijkingen $(index \leq n) \approx 4n$ $\Theta(n)$

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) aan het aantal basisoperaties, ofwel:
- alle andere operaties worden (in orde van grootte) hooguit even vaak uitgevoerd als de basisoperatie

Basisoperatie: voorbeelden

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

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

- bepaal een basisoperatie

- tel het aantal basisoperaties $\Longrightarrow f(n)$

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

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

probleem	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
graafprobleem	aantal knopen en/of takken

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

- worst case complexiteit is $g(n)$: het algoritme doet voor elke mogelijke invoer hooguit $g(n)$ stappen
- best case complexiteit is $h(n)$: het algoritme doet voor elke mogelijke invoer minstens $h(n)$ stappen
- worst case geeft dus een bovengrens, best case een ondergrens
- average case complexiteit is de complexiteit gemiddeld over alle mogelijke invoeren
- verschillende algoritmen voor hetzelfde probleem: vergelijk complexiteiten $\Longrightarrow$ optimaliteit

- bestaat er een efficiënter algoritme voor het probleem?

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

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

- bewijs stellingen die een ondergrens opleveren voor het aantal (basis)operaties dat nodig is om het probleem op te lossen, d.w.z.

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

- om een ondergrens te bewijzen kunnen we bijvoorbeeld een beslissingsboomargument of een adversary-argument gebruiken

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

Complexiteit van recursieve algoritmen:

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

- een en ander leidt tot recurrente betrekkingen

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

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

werk per aanroep ($n > 1$)

Laat $C(n)$ = aantal recursieve aanroepen op n elementen, dan geldt:

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

Oplossing: $C(n) = n$

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

a. Leg uit waarom het vergelijken van array-elementen (regel 3, tweede test) een goede basisoperatie is.
b. Bekijk worst case en best case.

Tweede college complexiteit

5 februari 2008

Wiskundige achtergrond

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

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

  Er geldt: $x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1$ en $\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n$ voor elk geheel getal $n \geq 0$

- $\log_b x = y \iff b^y = x$

  Notatie: $\lg x = \log_2 x$ en $\ln x = \log_e x$

  Er geldt: $\log_b x = \frac{\log_c x}{\log_c b}$

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

- $\lceil \frac{n}{2} \rceil = \begin{cases} \frac{n}{2} & \text{als } n \text{ even} \\ \frac{n+1}{2} & \text{als } n \text{ oneven} \end{cases}$
- $\lfloor \frac{n}{2} \rfloor = \begin{cases} \frac{n}{2} & \text{als } n \text{ even} \\ \frac{n-1}{2} & \text{als } n \text{ oneven} \end{cases}$

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

- $\sum_{i=1}^{n} i^2 = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$

- $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

- $\sum_{i=a}^{b} q^i = \frac{q^a - q^{b+1}}{1-q}$ voor $q \neq 1$

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

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

1. $f = O(g)$: er bestaan constanten c en n_0 (beide > 0) zodat $0 \leq f(n) \leq cg(n)$ voor alle $n \geq n_0$: asymptotische bovengrens

2. $f = \Omega(g)$: er bestaan constanten c' en n'_0 (beide > 0) zodat $0 \leq c'g(n) \leq f(n)$ voor alle $n \geq n'_0$: asymptotische ondergrens

3. $f = \Theta(g)$: er bestaan constanten c_1, c_2 en n''_0 (alle > 0) zodat $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ voor alle $n \geq n''_0$: asymptotisch gedrag

Voor bijbehorende plaatjes: zie college.

$f = \Theta(g)$: neem uit f de hoogste orde term, negeer alle lagere orde termen en de constante die voor de hoogste orde term staat $\longrightarrow g$.

$\Theta(1)$: constant

$\Theta(\lg n)$: logaritmisch

$\Theta(n)$: lineair

$\Theta(n^k)$ met $k > 0$: polynomiaal

$\Theta(2^n)$, $\Theta(a^n)$ met $a > 1$: exponentieel

$\frac{1}{2}n^2 - 3n = \Theta(n^2)$

$3n^3 + 6n^2 + 9 = \Theta(n^3)$

$42n = O(n^2)$, maar NIET $42n = \Omega(n^2)$

$\log_7 n = \Theta(\lg n)$

$\sum_{i=1}^{n} i = \Theta(n^2)$

$\sum_{i=1}^{n} \frac{1}{i} = \Theta(\lg n)$

$\sum_{i=1}^{n} \lg i = \Theta(n \lg n)$

$2^n = O(3^n)$, maar NIET $2^n = \Omega(3^n)$

$n! = 1 * 2 * 3 * \cdots * n$

$n! = O(n^n)$; $n! = \Omega(2^n)$

$\lg(n!) = \sum_{i=1}^{n} \lg i$

$n! \geq (\frac{n}{2})^{\frac{n}{2}}$ als n even is; $n! \geq (\frac{n}{2})^{\frac{n+1}{2}}$ als n oneven is

Dus $n! \geq (\frac{n}{2})^{\frac{n}{2}}$ als $n \geq 2$

En derhalve: $\lg(n!) \geq \frac{n}{2}(\lg n - 1)$

Formule van Stirling: $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$

Stelling

1. $f = \Theta(g) \iff f = O(g)$ en $f = \Omega(g)$
2. $f = O(g) \iff g = \Omega(f)$

Stelling

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \alpha$ met $0 < \alpha < \infty \Longrightarrow f = \Theta(g)$

2. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow f = O(g)$, maar $g \neq O(f)$

   (ofwel $f \neq \Omega(g)$)

3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Longrightarrow f = \Omega(g)$, maar $f \neq O(g)$

Gedrag voor grote n

In onderstaande tabel is te zien hoe snel enkele veel voorkomende functies toenemen als functie van n.

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	125000	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

Vergelijk:

het aantal protonen in het heelal is een getal met 79 cijfers

het aantal microseconden sinds de oerknal is een getal met 24 cijfers

Polynomiaal: (meestal) binnen redelijke tijd klaar
Exponentieel: zeker niet in acceptabele tijd klaar

N	10	20	50	100	300
N^2	$\frac{1}{10000}$ sec	$\frac{1}{2500}$ 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

executietijd tijd bij miljoen stappen per seconde

Vergelijk: de oerknal was ongeveer 15 miljard jaar geleden

Enige voorbeelden van recurrente betrekkingen.

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

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

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

   Dan geldt: $T(n) = O(n \lg n)$

Enige voorbeelden van recurrente betrekkingen.

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

   Oplossing: $T(n) = n$

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

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

De volgende stelling geeft een verband aan tussen de hoogte van een binaire boom en het aantal knopen (resp. bladeren).

Stelling

Gegeven een binaire boom met n knopen (en b bladeren) en hoogte h. Dan geldt:

1. $h \geq \lceil \lg b \rceil$

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

Derde college complexiteit

12 februari 2008

Zoeken en beslissingsbomen

Probleem
Zoek X in een (willekeurig) array A, bestaande uit n elementen $A[1], \ldots, A[n]$.

Complexiteit van het probleem (opgave 3.e.)
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.

Algoritme en complexiteit
Lineair zoeken (zie opgave 3) doet in de worst case n vergelijkingen, en is dus optimaal.
Overigens worden gemiddeld (average case) $q * \frac{1}{2} * (n+1) + (1-q) * n$ sleutelvergelijkingen gedaan, met q de kans dat X in A voorkomt.

Probleem

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

Algoritme (zie opgave 4):

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

De complexiteit van geordend lineair zoeken*.

Worst case:
n sleutelvergelijkingen ($A[index] < X$)

Average case:

$$\frac{n}{2} + \frac{n}{n+1} + q * (\frac{1}{2} - \frac{n}{n+1}) = \Theta(\frac{n}{2}),$$

met q de kans dat X in A voorkomt, en:

1. als X in A: alle n posities in A even waarschijnlijk
2. als X niet in A: alle $n+1$ gaten even waarschijnlijk

*zie ook opgave 4

Neem weer aan dat A oplopend gesorteerd is en kies k met $1 \leq k < n$

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

Worst case:

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

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

Vraag: kan zoeken in een geordend array nog beter?

Antwoord: Ja, namelijk binair zoeken.

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

We tellen de achtereenvolgende tests $X = A$[Midden] en $X < A$[Midden] als 1 vergelijking.

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

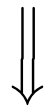
Average case (voor $n = 2^k - 1$): gemiddeld aantal vergelijkingen nodig om X te vinden in A is

- $\frac{1}{n}\Sigma_{i=0}^{k-1}(i+1)2^i$ als X zeker in A zit*
- k als X niet aanwezig†
- in totaal dus $\frac{q}{n}\Sigma_{i=0}^{k-1}(i+1)2^i + (1-q)k = \Theta(\lg(n+1))$, met q de kans dat X in A voorkomt.

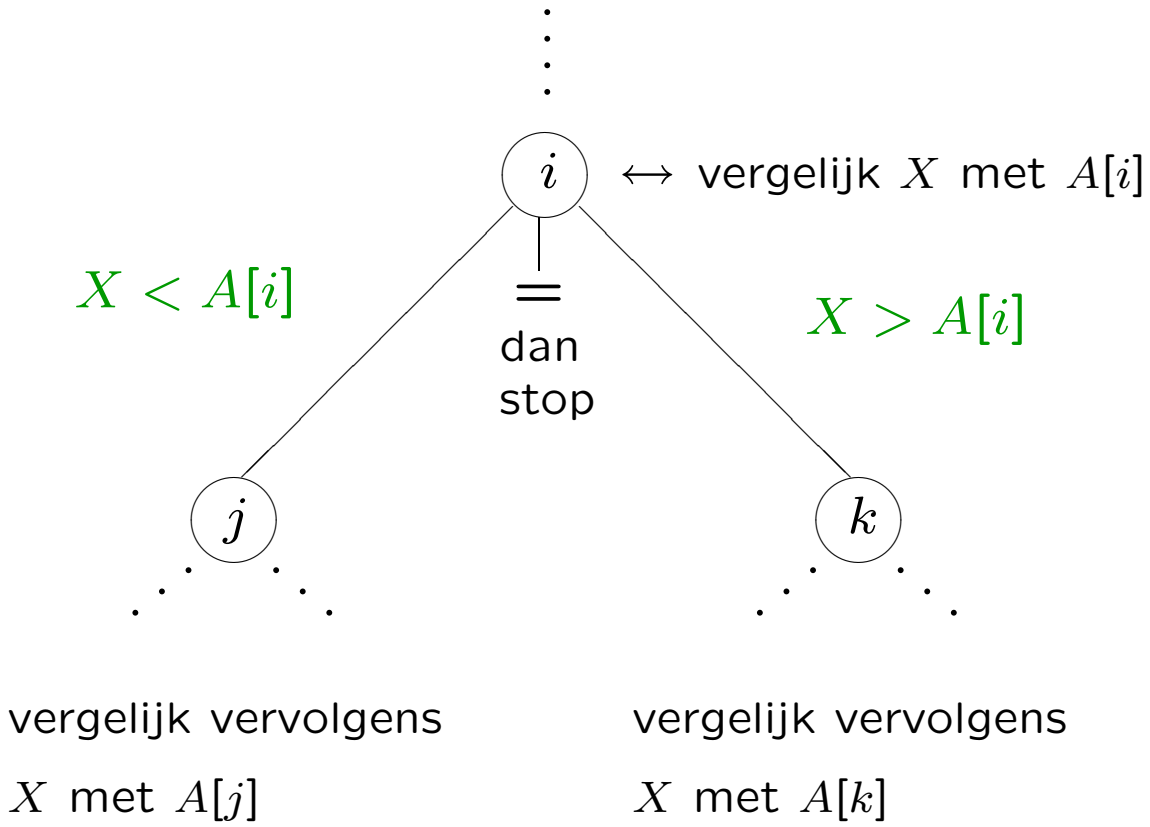
*onder de aanname dat elke positie even waarschijnlijk is

†het kost altijd k vergelijkingen om dat te constateren

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



beslissingsboom: binaire boom waarin knopen corresponderen met sleutelvergelijkingen; een pad vanaf de wortel naar een willekeurige knoop correspondeert met een executie van het algoritme



Beslissingsboom voor algoritmen gebaseerd op sleutelvergelijkingen

Stelling

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

Gevolg

Binair zoeken is optimaal (voor wat betreft de worst case)

*van de vorm $X =, < A[i]$

Vierde college complexiteit

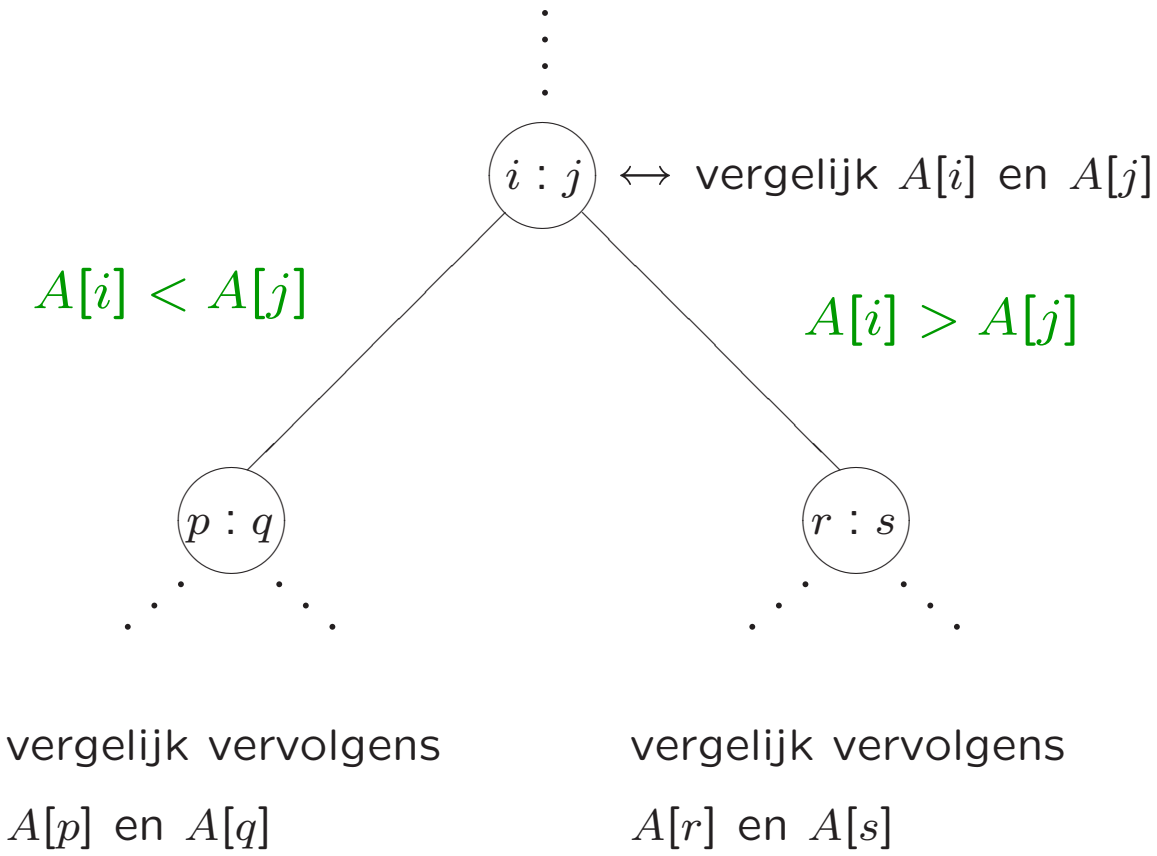
19 februari 2007

Selectie en adversary argument

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

beslissingsboom*: binaire boom waarin de interne knopen corresponderen met arrayvergelijkingen en de bladeren/externe knopen met het eindresultaat; een pad vanaf de wortel naar een blad correspondeert met een executie van het algoritme

*alle $A[i]$ zijn verschillend



Beslissingsboom voor algoritmen gebaseerd op arrayvergelijkingen

Stelling

Elk algoritme dat de grootste (resp. kleinste) bepaalt uit een array met n elementen, en dat uitsluitend gebaseerd is 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/minimum al een scherpere ondergrens gevonden, namelijk $n - 1$.

Probleem

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

Klasse van algoritmen

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

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).

Bovengrens

Het probleem kan worden opgelost door het array eerst aflopend 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

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

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

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

2. (Variant) Het maximum en minimum beide opsporen.

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

   Voor deze variant moet natuurlijk $n \geq 2$ gelden.

3. $k = 2$: de op een 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 (ook via 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 over de optimaliteit van dat algoritme zeggen.

Een optimaal algoritme:

```
if A[1] > A[2] then // n ≥ 2
    grootste := A[1]; kleinste := A[2];
else
    grootste := A[2]; kleinste := A[1];
i := 3;                                  // voor het gemak: n even
while i < n do                           // anders kleine aanpassing
    if A[i] > A[i + 1] then                                        *
        gr := A[i]; kl := A[i + 1];
    else
        gr := A[i + 1]; kl := A[i];
    fi
    if gr > grootste then                                          *
        grootste := gr;
    fi
    if kl < kleinste then                                          *
        kleinste := kl;
    fi
    i := i + 2;
od
```

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

Terminologie:

wedstrijd $\longleftrightarrow$ vergelijking;

winnaar $\longleftrightarrow$ grootste van de twee;

speler $\longleftrightarrow$ array-element; etcetera.

Complexiteit:

De toernooimethode doet $n + \lceil \lg n \rceil - 2$ arrayvergelijkingen. Dit is optimaal.

Geschikte implementatie:

met behulp van een soort heap (zie opgave 26).

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 ℓ rondes nodig ($\ell = \lg n$).

Algoritme (vervolg):

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

Als $n = 2^\ell$ doet de toernooimethode dus $n + \lg n - 2$ array-vergelijkingen in totaal. Dit is optimaal.

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

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

- algoritme: wil zo veel mogelijk informatie krijgen om met zo weing 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:

- datum raden (maand + dag)
- galgje: woord raden binnen 10 stappen
- X zoeken in een rij met n verschillende elementen

- 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 een bad-case invoer op
- de adversary(-strategie) zorgt ervoor dat hij op elk moment een invoer kan geven die consistent is met de op de gestelde vragen gegeven antwoorden
- het aantal stappen (=vragen) dat 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.

Stelling

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

Bewijs

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

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

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	$n-2$	

Merk op dat de uitslagen altijd kunnen, want de waarde van een V (resp. W) mag altijd omlaag (resp. omhoog).

vergelijking x-y	actie van de adversary (constructie bad case)
N-N	kies "frisse" waarden zodat x>y
W-N	kies "frisse" waarde voor y (<x) je mag de waarde van x ook ongestraft verhogen; dat beïnvloedt eerdere uitslagen namelijk niet
W-V	x mag ongestraft verhoogd worden (of y verlaagd) zodat x > y wordt
W-W	laat winnen welk van de twee groter is
WV-V	y mag ongestraft verlaagd worden

Vijfde college complexiteit

26 februari 2008

Selectie $O(n)$ en Insertion sort

Adversary argument

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

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

Het doel is een ondergrens $f(n)$ af te leiden voor het aantal stappen dat elk algoritme tegen de adversary nodig heeft. 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 $\geq f(n)$.

Bepaal of een string bestaande uit n bits twee opeenvolgende nullen bevat.

Basisoperatie: het bekijken van één bitpositie

Laat zien:

1. als $n = 2, 3, 5$ moeten in de worst case alle n bits bekeken worden (*)
2. voor $n = 4$ is er een algoritme dat altijd maar 3 bits hoeft te bekijken

(*) met behulp van een adversary argument

Probleem

Gegeven n verschillende getallen, opgeslagen in een array A: $A[1], A[2], \ldots, A[n]$. Laat verder een geheel getal k met $1 \leq k \leq n$ gegeven zijn. Gevraagd het k-de element in grootte.

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.

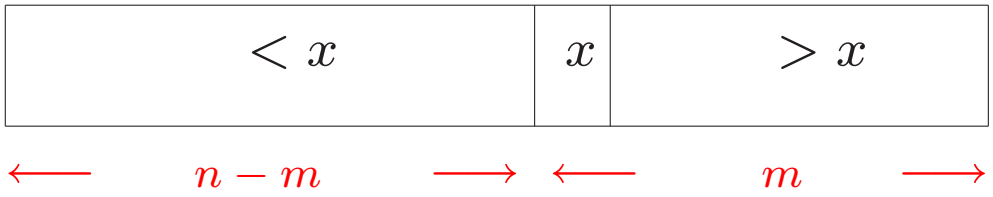
Algoritme:

1. Verdeel de getallen in $\lfloor \frac{n}{5} \rfloor$ groepjes van 5 elementen, en 1 groepje met de resterende n mod 5.

2. Vind de mediaan van elk van de $\lceil \frac{n}{5} \rceil$ groepjes van 5 (of minder), bijvoorbeeld met behulp van Bubblesort of opgave 21.

3. Vind de mediaan x van de in stap 2 gevonden $\lceil \frac{n}{5} \rceil$ medianen: recursie.

4. Partitioneer (ongeveer zoals bij Quicksort) alle elementen rond x. Stel dat m getallen $\geq x$ 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

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

Na stap 3. van het algoritme geldt:

- Ten minste $3 * (\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$ elementen zijn groter (respectievelijk kleiner) dan x.
- Dus er zijn hooguit $\frac{7n}{10} + 6$ elementen $\leq x$ (respectievelijk $\geq x$) (*).

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

(*) om preciezer te zijn: $\lceil \frac{7n}{10} \rceil + 6$

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

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

$< n$ (als $n > 20$) $\quad \leq dn \quad \uparrow$ (bijv. 80)

De term $O(n)$ komt van stap 1, 2 en 4 samen.

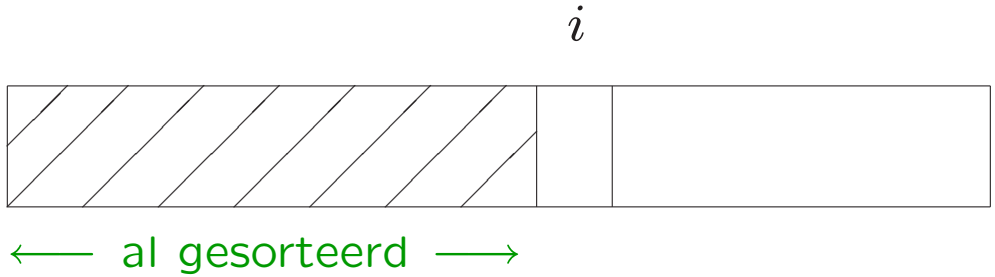
Bewering:

$T(n) \leq cn$ voor geschikte c en $n \geq \cdots$

Probleem

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

Insertion sort



$A[i]$ op de juiste plek in het gesorteerde stuk $A[1] \cdots A[i-1]$ invoegen.

Het algoritme is gebaseerd op het doen van arrayvergelijkingen.

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

- Het aantal 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 wordt neergezet.

- De verwisselingen zijn steeds buursverwisselingen.

We tellen het aantal vergelijkingen $A[j] > x$.

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

(*) onder de aanname dat alle $A[i]$'s verschillend zijn en dat alle $n!$ permutaties (ordeningen) van $A[1]$ t/m $A[n]$ even waarschijnlijk zijn. We middelen dan over alle mogelijke permutaties en dat zijn in essentie alle mogelijke invoerrijtjes.

Zesde college complexiteit

4 maart 2008

Inversies, mergesort en merge

Definitie: een inversie van de permutatie $A[1], A[2], \ldots, A[n]$ is een paar $(A[i], A[j])$ waarvoor $i < j$ en $A[i] > A[j]$. 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.

Bovendien: een buursverwisseling (zoals bij Insertion sort) heft altijd precies één inversie op (indien de buurelementen verkeerd om staan).

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

Gevolg. Elk sorteeralgoritme (gebaseerd op arrayvergelijkingen) dat hooguit één inversie opheft per vergelijking doet ten minste $\frac{1}{2}n(n-1)$ vergelijkingen in de worst case.

Conclusie: Insertion sort is optimaal voor wat betreft de worst case, binnen de klasse van algoritmen gebaseerd op het doen van arrayvergelijkingen, waarbij per vergelijking hooguit één inversie wordt opgeheven (bijvoorbeeld via buursverwisselingen).

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 ten hoogste één inversie opheft, moet ten minste $\frac{1}{4}n(n-1)$ vergelijkingen doen in de average case.

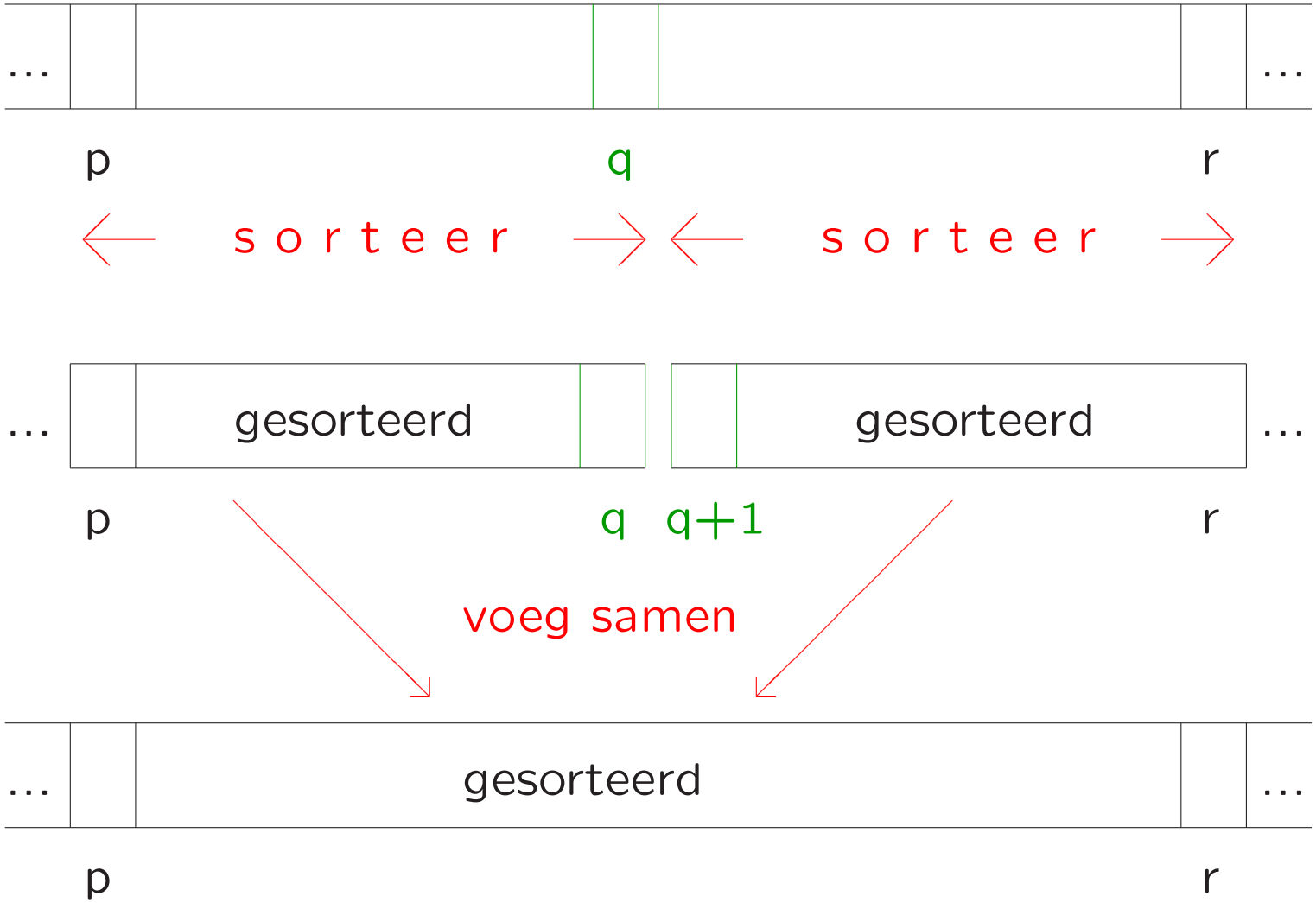
Insertion sort doet gemiddeld $\frac{1}{4}n(n-1)+n-\sum_{i=1}^{n}\frac{1}{i}$ vergelijkingen, dus Insertion sort is in de average case in orde van grootte optimaal (binnen de betreffende klasse van algoritmen), namelijk $\Theta(n^2)$.

Het (recursieve) algoritme:

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

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

Aanroep: MergeSort$(A, 1, n)$.


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

Merge(A, p, q, r)::

$i := p$; $j := q + 1$; $k := p$;
while $i \leq q$ **and** $j \leq r$ **do**
  if $A[i] < A[j]$ **then**
    $hulp[k] := A[i]$; $i := i + 1$; $k := k + 1$;
  else
    $hulp[k] := A[j]$; $j := j + 1$; $k := k + 1$;
  fi
od
if $i > q$ **then** // eerste helft is op
  kopieer $A[j], \ldots, A[r]$ naar $hulp$;
else // tweede helft is op
  kopieer $A[i], \ldots, A[q]$ naar $hulp$;
fi
kopieer $hulp[p], \ldots, hulp[q]$ terug naar A;

- 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)

- Voor het bepalen van de complexiteit van Merge tellen we het aantal vergelijkingen van de vorm: $A[i] < A[j]$

- Is het aantal arrayvergelijkingen hier wel een goede maat voor de complexiteit?

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 $\min\{k, m\}$

Zij $T(n) =$ aantal vergelijkingen in de worst case van Mergesort 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: $T(n) = n \lg n - n + 1 = \Theta(n \lg n)$

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

Mergesort is in orde van grootte optimaal voor wat betreft de worst case (zie later). Er is echter extra geheugenruimte ter grootte $\Theta(n)$ nodig.

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

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

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

Opmerking: het aantal vergelijkingen in de worst case is $\Theta(n \lg n)$, evenals het aantal vergelijkingen in de best case.
Gevolg: het aantal vergelijkingen in de average case is ook $\Theta(n \lg n)$.

Stelling.

1. Elk algoritme, alleen 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 ter 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 Mergealgoritme optimaal.

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

Kies de rijtjes $A = (a_1, a_2, \ldots, a_m)$ en $B = (b_1, b_2, \ldots, b_m)$ zo dat

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

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

Het bewijs van deze bewering gaat uit het ongerijmde.

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

Aannames:

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

Zo'n algoritme correspondeert (voor elke n) met een beslissingsboom die de series vergelijkingen representeert die het algoritme uitvoert voor elke mogelijke invoer.

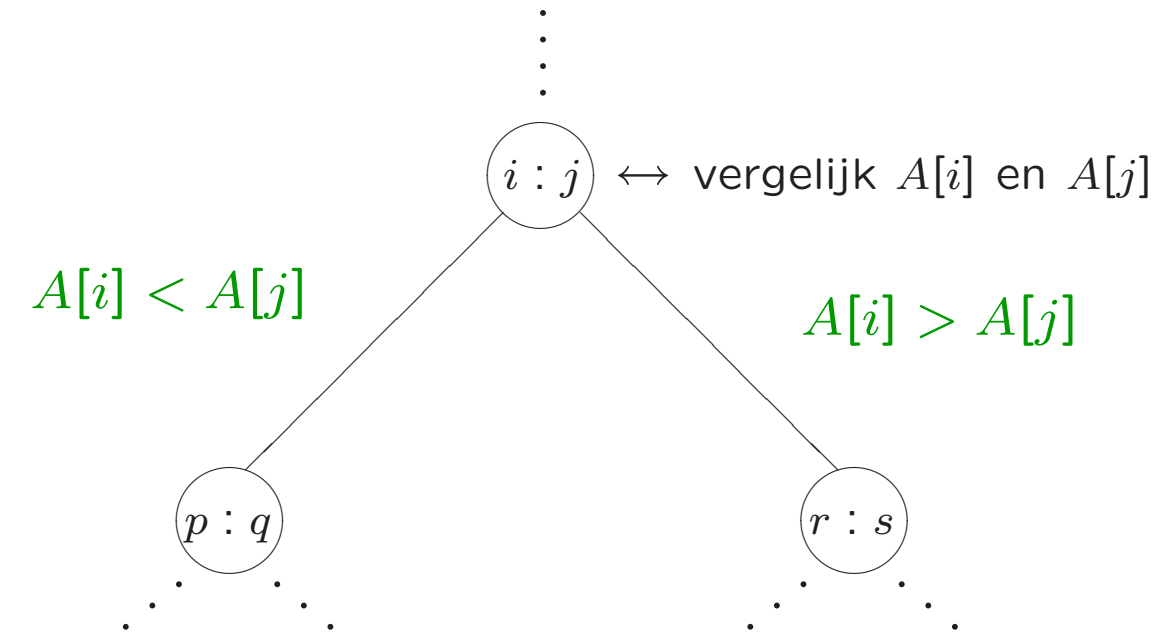
algoritme gebaseerd op het doen van arrayvergelijkingen $A[i] < A[j]$

$$\Downarrow$$

beslissingsboom*: binaire boom waarin de interne knopen corresponderen met arrayvergelijkingen en de bladeren/externe knopen met het eindresultaat†; een pad vanaf de wortel naar een blad correspondeert met een executie van het algoritme

*alle $A[i]$ zijn verschillend

†eindresultaat = de gevonden sorteringen/ordeningen (in dit geval)



Beslissingsboom voor algoritmen gebaseerd op arrayvergelijkingen

1. - alleen de onderlinge volgorde van de array-elementen is van belang; niet de waarde
   - het rijtje $1, 5, 3, 4, 2$ wordt bijvoorbeeld precies zo behandeld door het sorteeralgoritme als het rijtje $3, 15, 8, 11, 6$
   - ze volgen dan ook precies hetzelfde pad in de beslissingsboom
   - er zijn in essentie $n!$ mogelijke te onderscheiden invoeren
2. - sorteren = vind de oplopende ordening
   - er zijn dus $n!$ verschillende eindantwoorden (=ordeningen) mogelijk
   - een sorteeralgoritme moet die allemaal kunnen bereiken
   - de bijbehorende beslissingsboom moet dus minstens $n!$ bladeren hebben
3. conclusie uit het voorgaande: een beslissingsboom corresponderend met een sorteeralgoritme heeft precies $n!$ bladeren (n = aantal array-elementen)

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

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.

Zevende college complexiteit

17 maart 2008

Ondergrens sorteren, Quicksort

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

Aannames:

- A bevat n verschillende waarden.
- het sorteeralgoritme stopt zodra de sortering (onderlinge ordening) gevonden is.

Zo'n algoritme correspondeert (voor elke n) met een beslissingsboom die de series vergelijkingen representeert die het algoritme uitvoert voor elke mogelijke invoer.

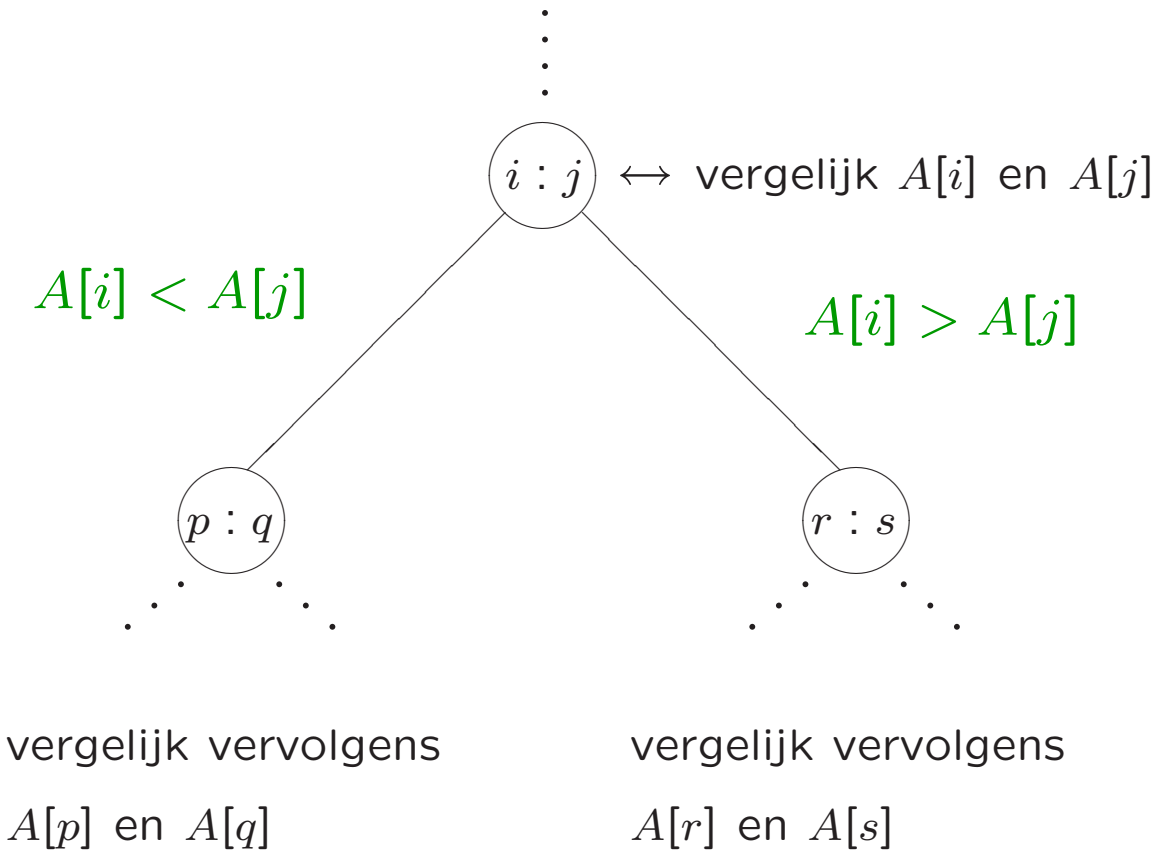
algoritme gebaseerd op het doen van arrayvergelijkingen $A[i] < A[j]$

$$\Downarrow$$

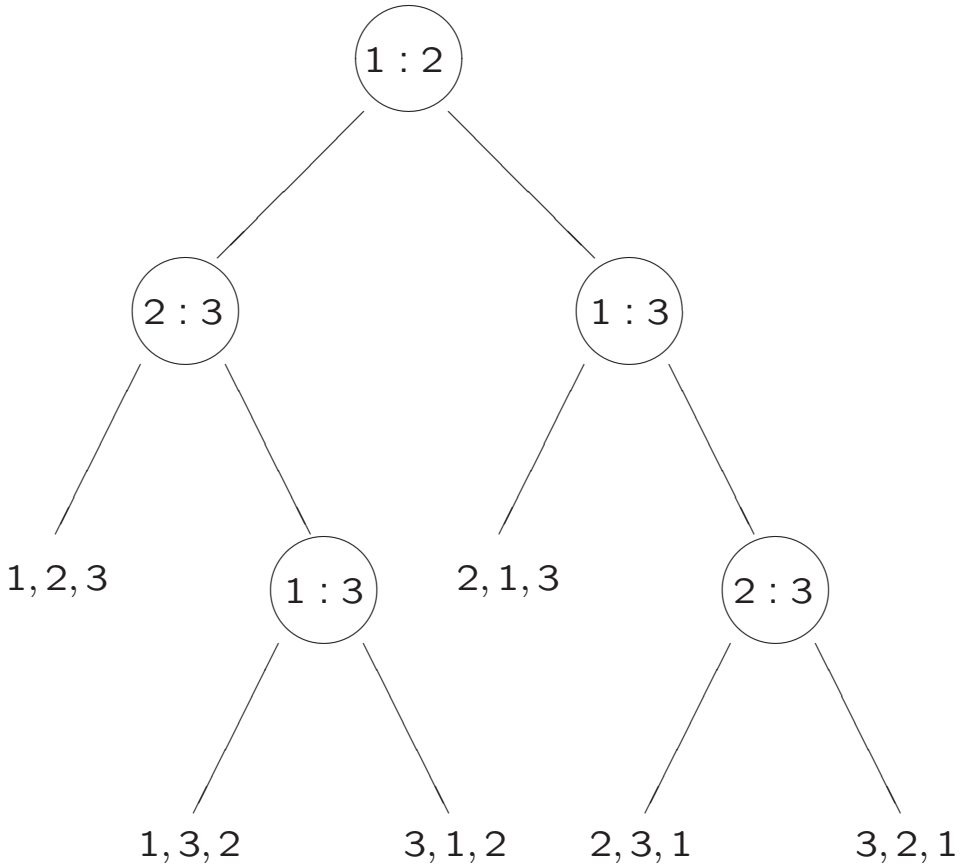
beslissingsboom*: binaire boom waarin de interne knopen corresponderen met arrayvergelijkingen en de bladeren/externe knopen met het eindresultaat†; een pad vanaf de wortel naar een blad correspondeert met een executie van het algoritme

*alle $A[i]$ zijn verschillend

†eindresultaat = de gevonden sorteringen/ordeningen (in dit geval)



Beslissingsboom voor algoritmen gebaseerd op arrayvergelijkingen



Beslissingsboom voor Insertion sort met $n = 3$

1. - alleen de onderlinge volgorde van de array-elementen wordt onderscheiden; niet de waarde
   - het rijtje $3, 15, 8, 11, 6$ wordt bijvoorbeeld precies zo behandeld door het sorteeralgoritme als het rijtje $1, 5, 3, 4, 2$
   - 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

2. - sorteren = vind 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 heeft precies $n!$ bladeren (n = aantal array-elementen)

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

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.

Gegeven een binaire boom $\mathcal{B}$ met b bladeren.

Definitie. De externe padlengte E van $\mathcal{B}$ is de som van de lengtes van alle paden van de wortel naar een blad:

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

Lemma. Zij E de externe padlengte van $\mathcal{B}$. Dan geldt:

$$E \geq b(\lg b - 1)$$

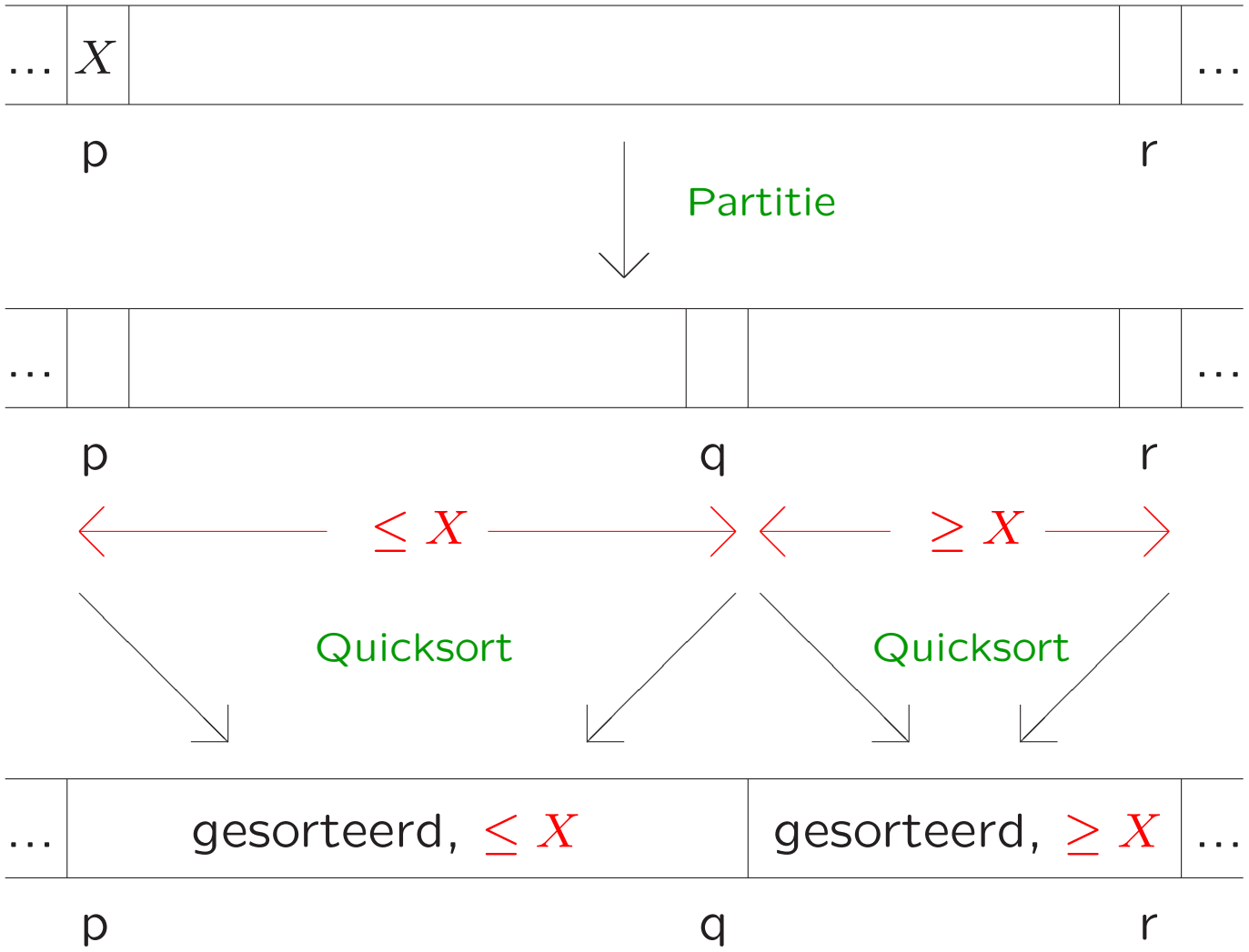
Gevolg. De gemiddelde lengte van een pad van de wortel naar een blad $= \frac{E}{b} \geq \lg b - 1$.

Quicksort (Hoare 1962)

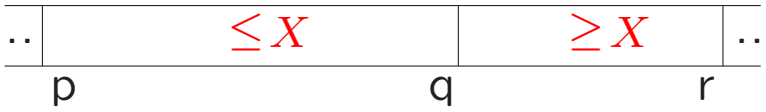
```
QuickSort(A, p, r)::
// sorteert A[p], ..., A[r] oplopend
        if p < r then
                q := Partitie(A, p, r);
                QuickSort(A, p, q);
                QuickSort(A, q + 1, r);
        fi
```

Aanroep:
Quicksort(A,1,n)

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


... X ...
p
r
Partitie
...
...
p
q
r
≤ X
≥ X
Quicksort
Quicksort
... gesorteerd, ≤ X gesorteerd, ≥ X ...
p
q
r

Partitie (A, p, r)::
// reorganiseert het (deel)array $A[p], \ldots, A[r]$ als volgt:
//

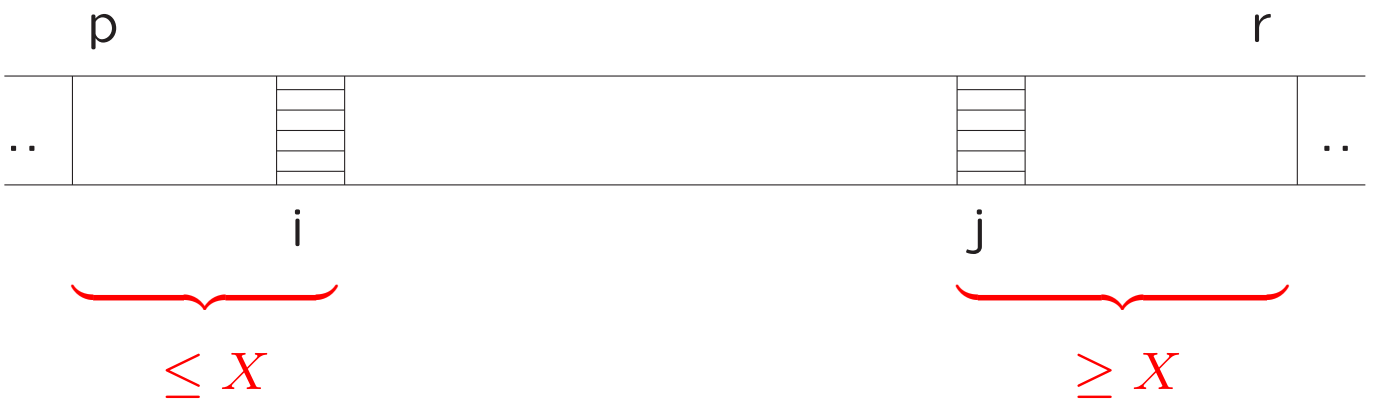


(*) // hier komt nog wat
$x = A[p]$; $i := p - 1$; $j := r + 1$;
while $i < j$ **do**
    $j := j - 1$; // loop met j naar links
    while $A[j] > x$ **do**
        $j := j - 1$;
    od // tot je een waarde $A[j] \leq x$ vindt
    i := i + 1; // loop met i naar rechts
    while $A[i] < x$ **do**
   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

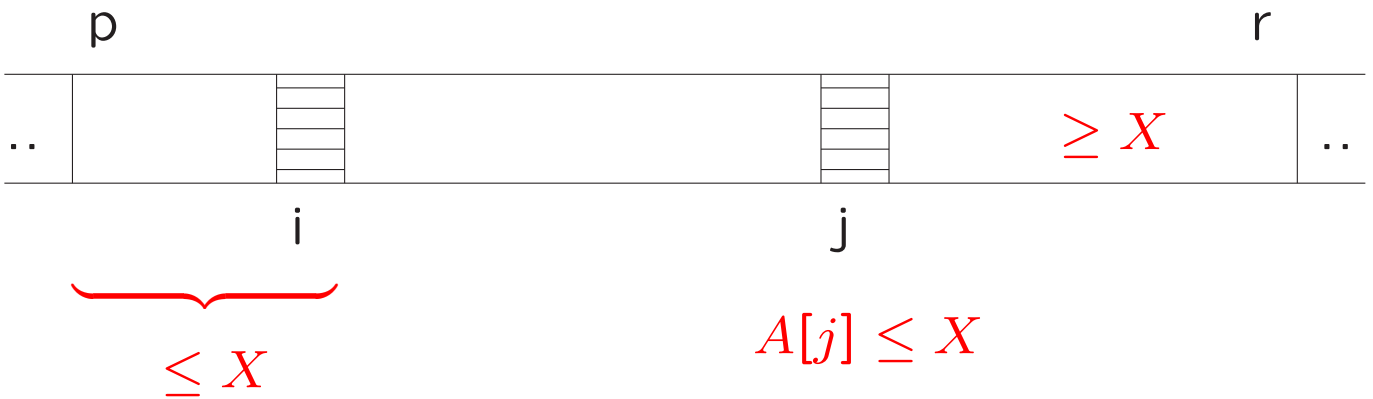
Opmerkingen bij Partitie

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

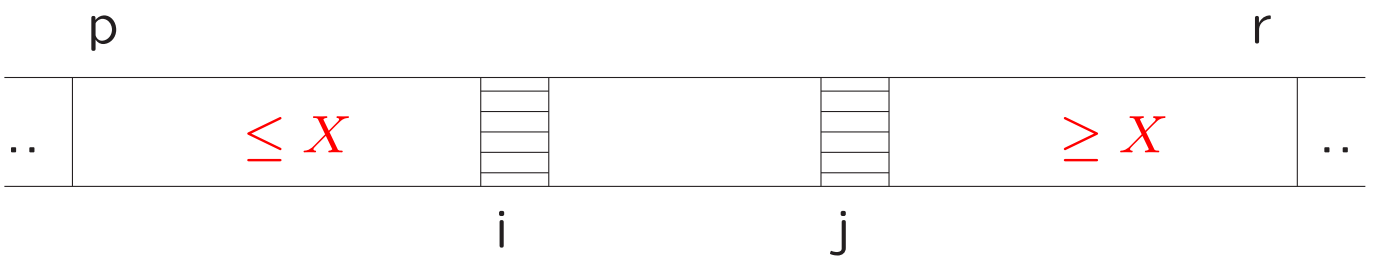
1. Na een volledige ronde:



2. Na de j-loop:

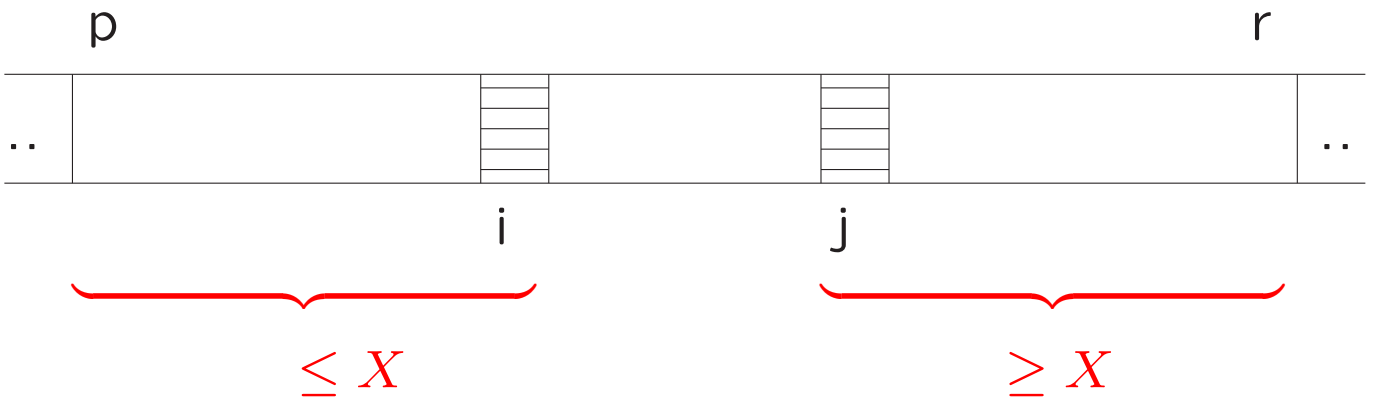


3. Na de i-loop, vóór de verwisseling:



$A[i] \geq X$ $A[j] \leq X$

4. Na de verwisseling:



1. Wat gebeurt er met gelijke array-elementen?

2. Wat gebeurt er met een rijtje met allemaal verschillende elementen?

3. Stel dat we $A[i]$ en $A[i+k]$, die verkeerd om staan, verwisselen. Hoeveel inversies worden dan ten minste resp. ten hoogste opgeheven?

1. Bad case voor Quicksort:

   Op het reeds gesorteerde rijtje (!), bijvoorbeeld $1, 2, \ldots, n$, doet Quicksort $\sum_{k=3}^{n+1} k = \frac{1}{2}n(n+3) - 2 = \Theta(n^2)$ vergelijkingen.

2. Good case voor Quicksort (n een tweemacht):

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

   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.

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

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

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

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

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

*Randomized Quicksort

Partitie (A, p, r)::
// reorganiseert het (deel)array $A[p], \ldots, A[r]$ als volgt:

Kies *random* array-element en wissel met $A[p]$.
$x = A[p]$; $i := p - 1$; $j := r + 1$;
while $i < j$ **do**
  $j := j - 1$; // loop met j naar links
  while $A[j] > x$ **do**
    $j := j - 1$;
  od // tot je een waarde $A[j] \leq x$ vindt
  i := i + 1; // loop met i naar rechts
  while $A[i] < x$ **do**
    $i := i + 1$;
  od // tot je een waarde $A[i] \geq x$ vindt
  if $i < j$ **then**
    wissel$(A[i], A[j])$;
  fi // $A[i]$ en $A[j]$ staan nu weer in het goede stuk
od
return j; // dit wordt dus q

Algorithm	Worst case	Average	Space usage
Insertion sort	$n^2/2$	$\Theta(n^2)$	In place
Quicksort	$n^2/2$	$\Theta(n \lg n)$	Extra space proportional to $\lg n$
Mergesort	$n \lg n$	$\Theta(n \lg n)$	Extra space proportional to n for merging
Heapsort	$2n \lg n$	$\Theta(n \lg n)$	In place
Accel. Heapsort	$n \lg n$	$\Theta(n \lg n)$	In place

Vergelijking van verschillende sorteeralgoritmen

(tijd in seconden)

n	Insertion sort $O(n^2)$	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
10000	58.864	0.42998	0.71650	0.36765	0.31532
100000	NA	5.7298	8.8591	4.2298	3.5882
1000000	NA	71.164	104.68	47.065	41.282

Quicksort* = optimized Quicksort

Achtste college complexiteit

18 maart 2008

Opgave 37/38 & Shellsort

a. Hoe werkt *Quicksort* op het rijtje 5 5 5 5 5 5 5 5 ? Hoeveel vergelijkingen en hoeveel verwisselingen worden gedaan?

b. Laat $n = 2^k$. Hoeveel vergelijkingen doet Quicksort voor een rijtje van n dezelfde getallen? En hoeveel verwisselingen?

c. Is Quicksort stabiel?

d. Laat n even zijn. Hoeveel vergelijkingen doet Quicksort op het rijtje $n, n-1, \ldots, 2, 1$? En hoeveel verwisselingen? Hoeveel verwisselingen worden gedaan op het reeds gesorteerde rijtje $1, 2, \ldots, n-1, n$?

a. Zoals bekend doet *Quicksort* $\frac{1}{2}n(n+3)-2 = \Theta(n^2)$ vergelijkingen op reeds gesorteerde rijtjes zoals $1, 2, 3, \ldots, n$. Bekijk wat er gebeurt met het gesorteerde rijtje als nu niet steeds het eerste array-element $A[p]$ wordt gekozen als pivot, maar de mediaan van $A[p]$, $A[r]$ en $A[\lceil \frac{p+r}{2} \rceil]$.

b. Neem aan dat we een rijtje met n verschillende getallen hebben. Hoe wordt het rijtje opgesplitst door Partitie als de pivot X het i-de element in grootte is (van klein naar groot: het 1-e element in grootte is de kleinste, het n-de in grootte de grootste)? M.a.w. wat is de q die Partitie oplevert?

c. Laat $n = 10$. We bekijken weer de "gewone" versie van Quicksort, dus waarbij als pivot steeds het eerste array-element wordt gekozen. Hoeveel vergelijkingen doet Quicksort als Partitie om en om de slechtste en de beste splitsing oplevert? Neem aan dat Partitie in elke stap $m + 1$ vergelijkingen doet voor een (deel)rijtje van m elementen. Vergelijk dit aantal met het aantal vergelijkingen dat gedaan wordt voor het gesorteerde rijtje. Idem met het aantal vergelijkingen dat Quicksort doet voor het rijtje waarbij in elke stap de (deel)rij in twee ongeveer gelijke stukken wordt verdeeld (best case). Ligt het aantal vergelijkingen dichter bij de best case of bij de worst case?

d. Maak aannemelijk dat een invoerrijtje waarbij afwisselend slecht en goed wordt gesplitst (zie **c.**) in orde van grootte evenveel vergelijkingen doet als het rijtje waarbij in elke ronde in twee gelijke delen wordt gesplitst ($\Theta(n \lg n)$ dus). Gebruik hiertoe dat in het eerste geval in *twee* rondes drie deelrijtjes zijn "ontstaan" van resp. 1, $\frac{m-1}{2}$ en $\frac{m-1}{2}$ (geval m is oneven) elementen, terwijl we in het tweede geval na *een* ronde een rijtje ter lengte $\frac{m-1}{2}$ en een rijtje ter lengte $\frac{m+1}{2}$ hebben. Hierin is m de lengte van het (deel)rijtje waarop we Quicksort (dus Partitie) loslaten.

Een en ander suggereert dat Quicksort het meestal goed ($= \Theta(n \lg n)$) doet: het gemiddelde zal "dus" rond $\Theta(n \lg n)$ liggen.

Shellsort* was een van de eerste sorteeralgoritmen met worst case complexiteit minder dan kwadratisch.

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 / compare-exchange (indien nodig) operaties.

*Donald Shell, 1959

Definitie

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

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 = gesorteerd.

Shellsort gebruikt een rijtje stapgroottes (increments) $h_t, h_{t-1}, \ldots, h_2, h_1 = 1$. De rij A met n elementen wordt gesorteerd door achtereenvolgens subrijen te sorteren van elementen die telkens op afstand h_i van elkaar liggen. Met andere woorden: A wordt h_i-gesorteerd voor $i = t, \ldots, 1$. 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.

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

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

$\downarrow$ 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

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

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

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

↓ 6-sorteren

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

↓ 3-sorteren

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

↓ 1-sorteren

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

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

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

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

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

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

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

De complexiteit van Shellsort (= 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 $t = \lfloor \lg n \rfloor$. Voor het gemak nemen we $n = 2^k$, dus $t = k$.

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

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

Negende college complexiteit

25 maart 2008

Shellsort, Optimaal sorteren, O(n)-sorteren

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

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

Shellsort: 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

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

Insertion sort doet hier **52** vergelijkingen;
Shellsort doet **44** vergelijkingen

De complexiteit van Shellsort (= 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 $t = \lfloor \lg n \rfloor$. Voor het gemak nemen we $n = 2^k$, dus $t = k$.

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

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

Stelling B. Het aantal arrayvergelijkingen dat Shellsort met deze increments doet is in de worst case $O(n^2)$.

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

2. Stapgroottes (Hibbard): $2^k-1, 2^{k-1}-1, \ldots, 7, 3, 1$ ($k = \lfloor \lg n \rfloor$). Merk op dat opeenvolgende increments hier relatief priem zijn (volgt uit: $h_{i+1} = 2h_i + 1$).

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

3. Het kan nog beter!

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

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

Beter:

- Twee doorgangen: $h_2 \approx 1,72 \cdot \sqrt[3]{n},\ h_1 = 1$.
  Gemiddeld $O(n^{5/3})$

- Heel veel doorgangen: increments 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 $O(n(\lg n)^2)$.

- Increments van de vorm $4^{j+1} + 3 \cdot 2^j + 1$:
  $1, 8, 23, 77, 281, 1073, 4193, 16577, \ldots$
  Worst case $O(n^{4/3})$.

- $\ldots$

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

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

Tabel:

n	2	3	4	5	6	7	8	9	10	11	12	...	21
$\lceil \lg n! \rceil$	1	3	5	7	10	13	16	19	22	26	29	...	66
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

M(n) = aantal vergelijkingen dat Mergesort doet in de worst case voor een rij met n elementen.
B(n)= aantal vergelijkingen dat Binary Insertion sort in de worst case doet 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 31.)

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

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

Er geldt zelfs:

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

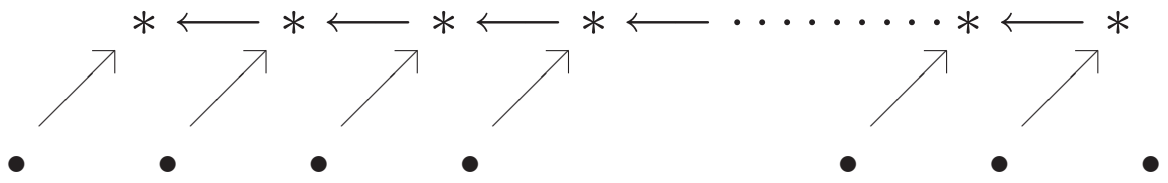
en dat is ook precies het aantal vergelijkingen dat Mergesort doet. Dus $B(n) = M(n)$.

Vraag: Kan het nog beter?
Antwoord: Ja !
Voorbeeld: 5 elementen kunnen met 7 vergelijkingen gesorteerd worden (Demuth, 1956).

De methode van Demuth kan gegeneraliseerd worden tot het algoritme Merge Insertion sort (Ford & Johnson).

1. Vergelijk de n elementen twee aan twee.

2. Sorteer de $\lfloor \frac{n}{2} \rfloor$ winnaars $*$ (de grootsten dus) recursief. Dit levert iets op als:



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

Merge Insertion sort sorteert bijvoorbeeld:

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

Vraag: is Merge Insertion sort optimaal?
Antwoord: nee, bijvoorbeeld $n = 47$.

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

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

```
for i := 1 to k do
    C[i] := 0;        // initialisatie
od
for j := 1 to n do
    C[A[j]] := C[A[j]] + 1;
    // telt aantal keer dat A[j] in A voorkomt
od
for i := 2 to k do
    C[i] := C[i] + C[i − 1];
od
// C[i] bevat nu het aantal getallen ≤ i uit A
for j := n downto 1 do      // !!
    B[C[A[j]]] := A[j];
    C[A[j]] := C[A[j]] − 1;
od
```

Voorbeeld

A	$=$	3	6	4	1	3	4	1	4	$n = 8$
C	$=$	0	0	0	0	0	0			$k = 6$
C	$=$	2	0	2	3	0	1			$\longleftarrow$ na 2^e for

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

C	$=$	2	2	4	7	7	8			$\longleftarrow$ na 3^e for

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

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

Extra geheugenruimte is ook $O(n+k)$.

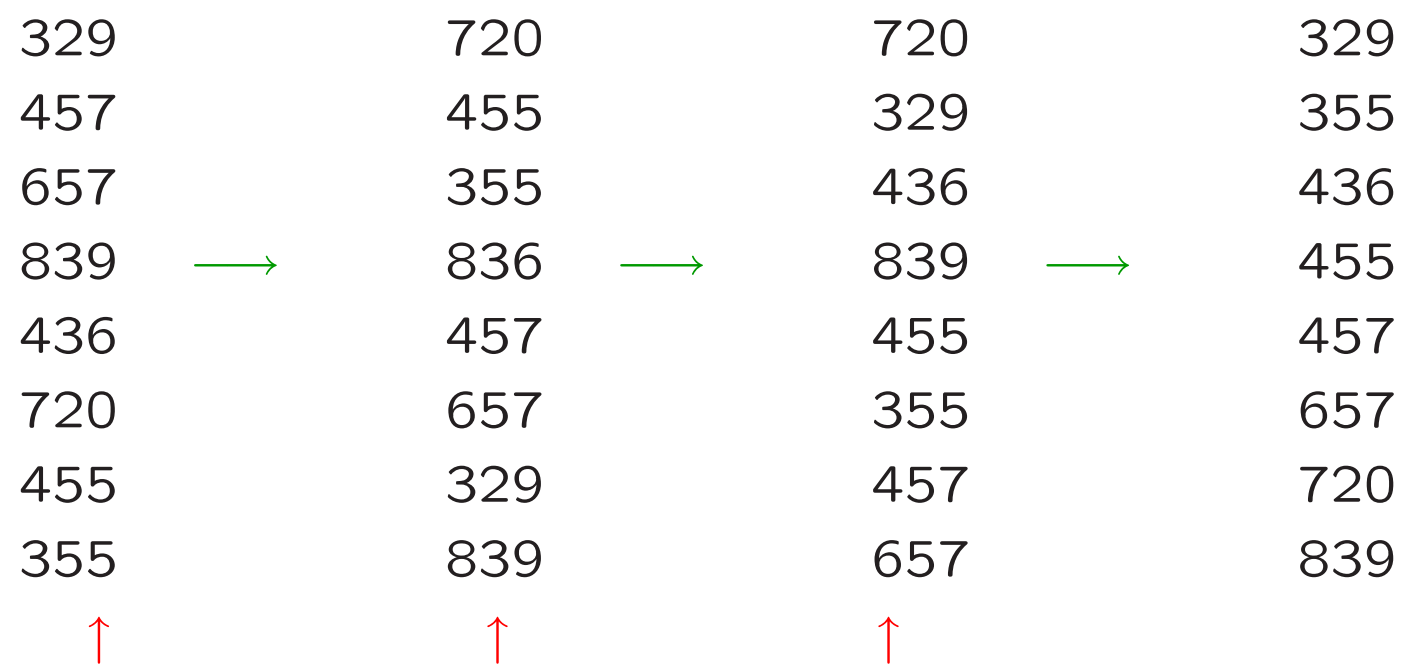
Counting sort is een stabiele sorteermethode, dat wil zeggen: gelijke waarden uit het invoerarray A komen in precies dezelfde volgorde in het uitvoerarray B te staan als ze in A stonden.

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-de cijfer te sorteren, te beginnen bij het minst significante cijfer. Het gebruik van een stabiele sorteermethode voor het sorteren op het i-de cijfer is essentieel.

Radixsort(A, d):

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

Voorbeeld


329
457
657
839
436
720
455
355
sorteer op
1e cijfer
720
455
355
836
457
657
329
839
sorteer op
2e cijfer
720
329
436
839
455
355
457
657
sorteer op
3e cijfer
329
355
436
455
457
657
720
839

Indien de gebruikte sorteermethode voor het sorteren per cijfer niet stabiel is kan het fout gaan:

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

- Als sorteermethode per cijfer kunnen we Counting sort gebruiken, aangezien de cijfers tussen 0 en $k-1$ zitten.

- In dat geval kost elke ronde $O(k+n)$ stappen. In totaal (d rondes) dus $O(dk+dn)$. En dat is $O(n)$ als d een constante is en $k = O(n)$.

- Een nadeel van deze methode is dat er net als bij Counting sort $O(n+k)$ extra geheugenruimte nodig is.

Tiende college complexiteit

1 april 2008

Polynoomevaluatie

NP-volledigheid I

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: $a_0, a_1, \ldots, a_{n-1}, a_n$ en x
Gevraagd: $p(x)$

Van links naar rechts de termen $a_i x^i$ berekenen en optellen levert een $O(n^2)$-algoritme.
Als we het polynoom daarentegen van rechts naar links evalueren kunnen we eenvoudig een ordeverbetering bereiken.

Algoritme 1: "gewoon"

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

Basisoperatie: $*$ en $+, -$

Complexiteit: aantal $* = 2n - 1$

aantal $+, - = n$

Algoritme 2: methode van Horner

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

Gebaseerd op:

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

Complexiteit:

aantal $* = n$

aantal $+, - = n$

Vraag: kan het met minder $*$ en $+, -$?

Antwoord: nee !

Algoritmen gebaseerd op het doen van vergelijkingen konden we beschrijven met **beslissingsbomen**.
Een model om rekenkundige algoritmen (algoritmen die gebaseerd zijn op $+, -, *$ en $/$) mee 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: $*, /, +$ of $-$
- q en r zijn constanten (bijvoorbeeld $1, -1, \pi^2, \ldots$) of invoerwaarden (hier a_k's of x) of tussenresultaten van eerdere stappen
- de laatste stap uit het schema berekent het eindresultaat (hier dus $p(x)$)

Stelling. Een schema (dat alleen $+, -$ en $*$ gebruikt) om $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ te berekenen moet ten minste n $(+, -)$ stappen doen en n $*$ stappen.
Het bewijs gaat met inductie naar n.

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 met minder $*$'s en +/-'s kan oplossen. De methode van Horner is derhalve optimaal.

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

Polynoomevaluatie met preprocessing: bewerk het polynoom tot een polynoom in een speciale vorm waarop een nieuw evaluatie-algoritme 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 dit zonder beperking der algemeenheid aannemen.

De speciale vorm

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

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 $*$'s en 7 $+/-$'s, maar het kan met 5 $*$'s en 10 $+/-$'s.

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

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

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

Vervolgens kunnen q en r op dezelfde manier in de speciale vorm gebracht worden, etcetera. Algoritme: zie opgave 44.

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

1. Evalueer $q(x)$ en $r(x)$ recursief

2. Bereken de x^j's: nodig hiervoor zijn $x, x^2, x^4, \ldots, x^{2^{k-1}}$.
   Bereken deze alle van tevoren: $k-1$ $*$'s

3. Vermenigvuldig $(x^j + b)$ met $q(x)$ en tel er $r(x)$ bij op:
   1 $*$ en 2 $+/-$'s

Zij $M(k)$ = het aantal $*$'s dat gedaan wordt om een monisch polynoom (in de speciale vorm) van graad $2^k - 1$ te eva- lueren, 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: $M(k) = 2^{k-1} - 1 \longrightarrow \frac{n-1}{2}$

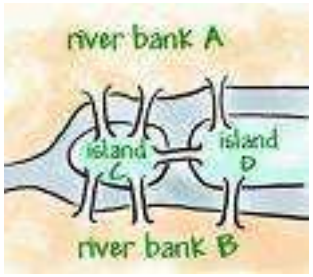
Zij $A(k)$ = het aantal $+/-$'s dat gedaan wordt om een monisch polynoom (in de speciale vorm) van graad $2^k - 1$ te evalueren.

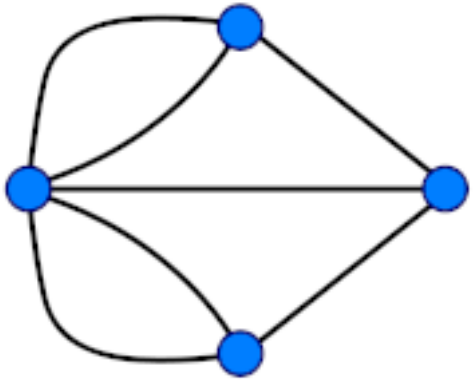
Dan voldoet $A(k)$ aan de volgende recurrente betrekking:

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

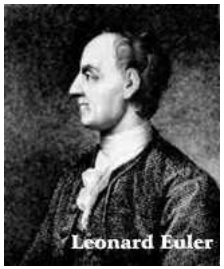
Oplossing: $A(k) = 3 * 2^{k-1} - 2 \longrightarrow \frac{3n-1}{2}$

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?



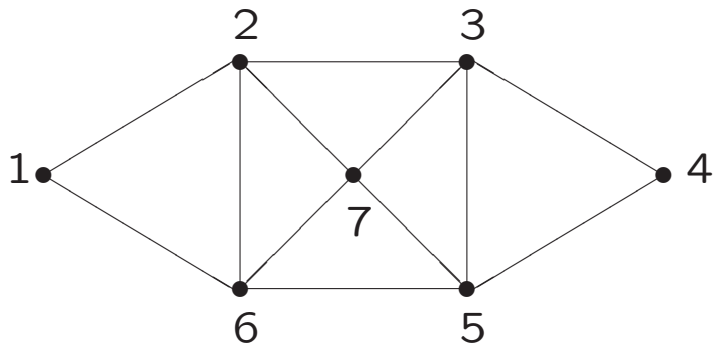
Leonard Euler, 1736

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.

* Een kring bestaat per definitie uit allemaal verschillende takken.

Voorbeeld:



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

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

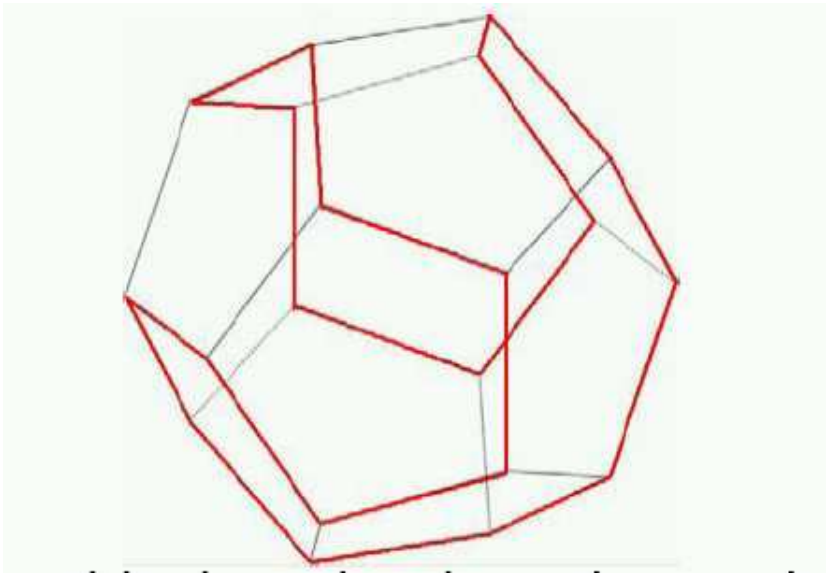
Dit is een voorbeeld van een beslissingsprobleem: het antwoord is ja of nee.

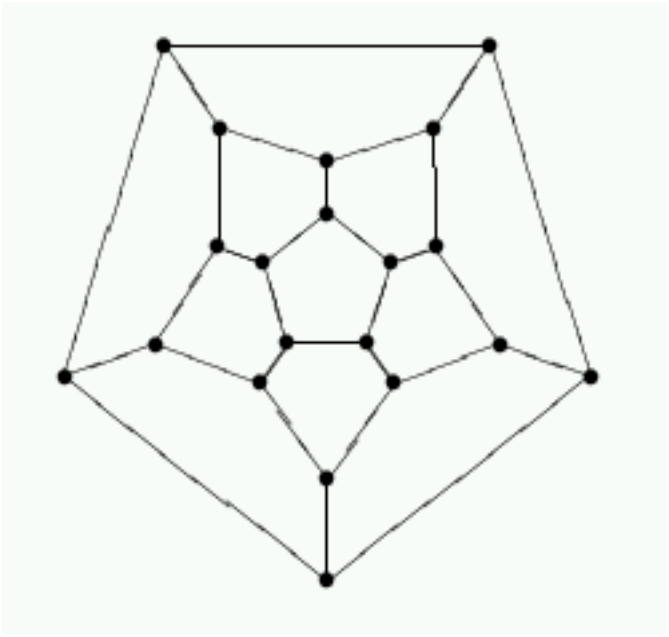
Stelling
Een samenhangende ongerichte graaf heeft een Eulerkring $\Longleftrightarrow$ de graad van iedere knoop is even.

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

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

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





Sir William Rowan Hamilton, 1859

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

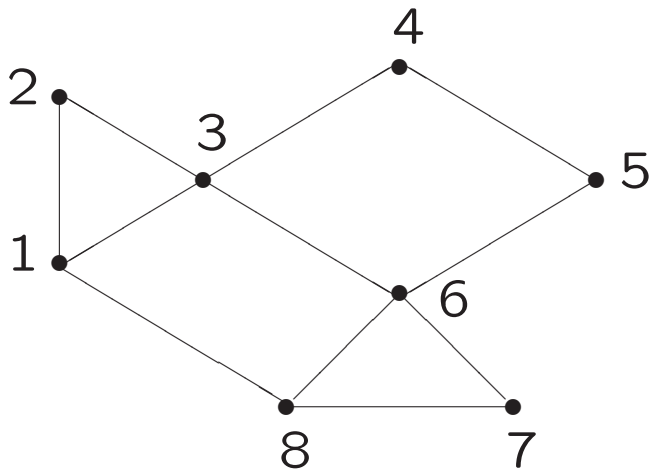
Definitie: een Hamiltonkring in $\mathcal{G}$ is een kring die elke knoop precies één keer bevat.

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 is $\mathcal{G}$ een nee-instantie.

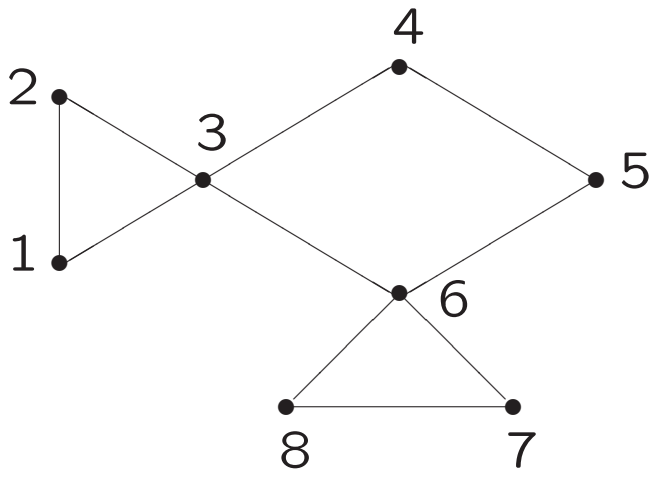
* analoog voor gerichte grafen

Voorbeeld 1:



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

Voorbeeld 2:



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

.

1. Een exponentieel algoritme is snel gevonden.
2. Er is geen polynomiaal algoritme voor dit probleem bekend.
3. Er is ook niet bewezen dat een exponentieel algoritme nodig is.
4. Als $\mathcal{G}$ een Hamiltonkring heeft is er een makkelijke (=polynomiaal) manier om dat aan te tonen (certificaat*).
5. De enige manier om te laten zien dat $\mathcal{G}$ geen Hamiltonkring bevat lijkt te zijn: som alle $n!^{\dagger}$ kandidaat Hamiltonkringen op en laat zien dat ze geen Hamiltonkring zijn.

* voor dit probleem: certificaat = Hamiltonkring

† $n = |V|$ = aantal knopen van $\mathcal{G}$

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 Oerknal heeft 24 cijfers

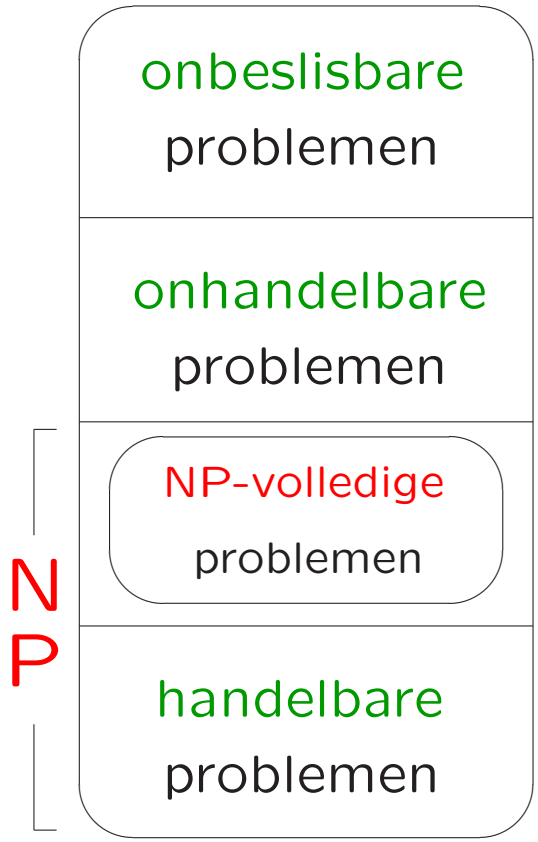
Handelbaar/onhandelbaar -2-

Stel dat de computer 1 instructie doet per microseconde (10^{-6} sec).

N	10	20	50	100	300
N^2	$\frac{1}{10000}$ sec	$\frac{1}{2500}$ 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.

Problemen kunnen als volgt in klassen worden ingedeeld:



Problemen waarvoor geen algoritme mogelijk is (stop-probleem)

Problemen waarvoor geen polynomiaal algoritme mogelijk is

????

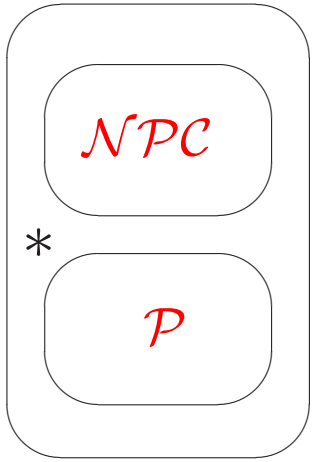
Problemen waarvoor een polynomiaal algoritme bestaat: P

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

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

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

Uit de theorie van NP-volledigheid:



$\mathcal{NP}$

Nondeterministic

Polynomial

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

Plaatje

Open vraag

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.

Handelsreizigersprobleem of Travelling Salesperson Problem

Optimalisatieprobleem

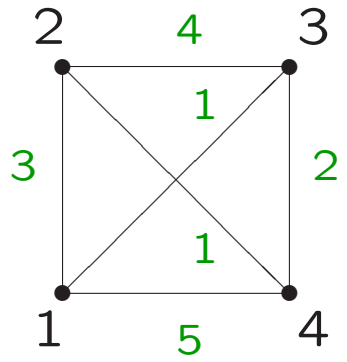
Gegeven een volledige *, ongerichte graaf $\mathcal{G} = (V, E)$ met gewichten op de takken. Geef een Hamiltonkring in $\mathcal{G}$ met minimaal totaalgewicht.

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.

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



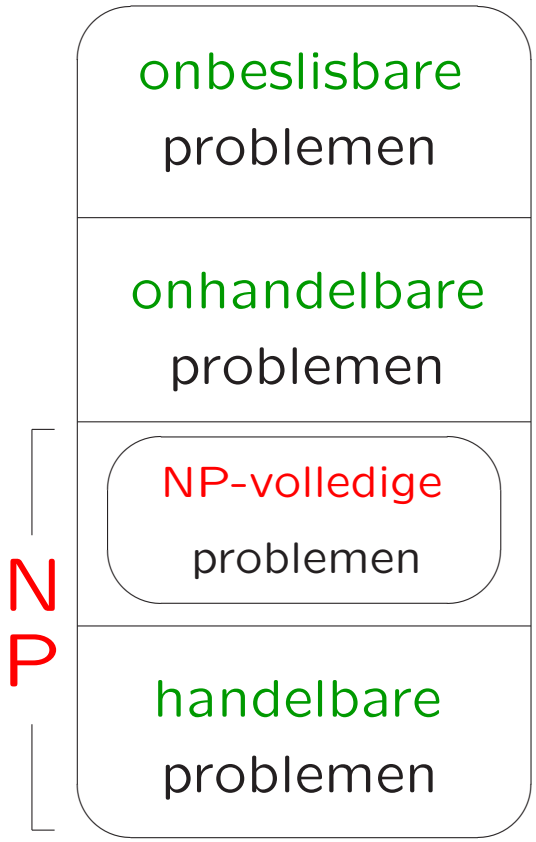
Elfde college complexiteit

8 april 2008

NP-volledigheid II

Indeling problemen

Problemen kunnen als volgt in klassen worden ingedeeld:



Problemen waarvoor geen algoritme mogelijk is (stop-probleem)

Problemen waarvoor geen polynomiaal algoritme mogelijk is

????

Problemen waarvoor een polynomiaal algoritme bestaat: P

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

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

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

De theorie van NP-volledigheid beperkt zich tot beslissingsproblemen. 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.

- 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 ϕ 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 is een toekenning van de waarde True of False aan elk van de logische variabelen uit die verzameling.

Beslissingsprobleem SAT

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

Voorbeeld.
De waardering w met $w(x_1) =$ False, $w(x_2) =$ True en $w(x_3) =$ False maakt de logische formule $\phi = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge \neg x_1$ waar.
Deze ϕ is dus een ja-instantie voor SAT.

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 $\Sigma_{s \in S'} s = t$?

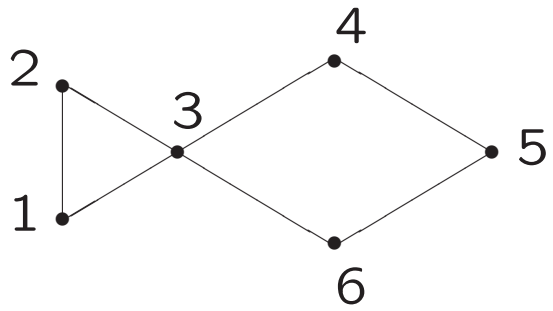
Voorbeeld.
Neem $S = \{1, 4, 16, 64, 256, 1040, 1093, 1284, 1344\}$ en $t = 3754$, dan is dit een ja-instantie voor het probleem. Immers $S' = \{1, 16, 64, 256, 1040, 1093, 1344\}$ voldoet.

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.



Nevenstaande graaf is een nee-instantie voor HC.

Handelsreizigersprobleem of Travelling Salesperson Problem

Optimalisatieprobleem

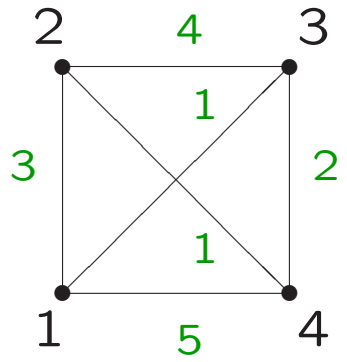
Gegeven een volledige *, ongerichte graaf $\mathcal{G} = (V, E)$ met gewichten op de takken. Geef een Hamiltonkring in $\mathcal{G}$ met minimaal totaalgewicht.

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.

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



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

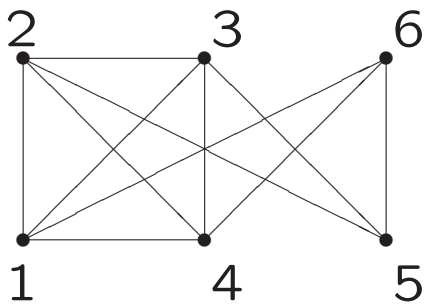
Gegeven een ongerichte graaf $\mathcal{G} = (V, E)$. Geef een kliek met zo veel mogelijk knopen (een maximale kliek).

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?

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

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



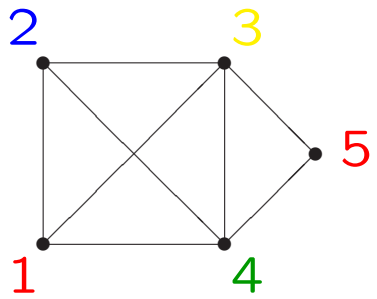
Een kleuring van (de knopen van) een ongerichte graaf $\mathcal{G} = (V, E)$ is een afbeelding $c : V \to S$, waarin S een eindige verzameling (van kleuren) is, en wel zó dat als $(v, w) \in E$ dan $c(v) \neq c(w)$. Met andere woorden: aangrenzende knopen hebben niet dezelfde kleur.

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

Opmerking.
Het is equivalent om te vragen naar een kleuring met precies k kleuren.

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



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

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

3. Voor al deze problemen geldt: het lijkt extreem moeilijk (exponentieel) om voor gegeven invoer x te bepalen of het antwoord "ja" of "nee" moet zijn.

4. Echter, *als* x een ja-instantie is, dan is er een eenvoudige (polynomiale) manier om iemand daarvan te overtuigen.

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

6. Bovendien is dit certificaat kort (polynomiaal).

7. Eigenschap 2 t/m 6 betekenen dat de genoemde problemen in $\mathcal{NP}$ zitten. Later wordt alles wat formeler gemaakt.

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

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

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

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

Vraag: waarom handelbaar = polynomiaal begrensd?

- 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 van polynomiaal begrensde algoritmen is zelf ook weer polynomiaal begrensd (zie opgaven)

- de klasse $\mathcal{P}$ is onafhankelijk van het berekeningsmodel en van gebruikte coderingen: als een probleem polynomiaal begrensd is in het ene model, dan ook in een ander model

Vraag: wat is eigenlijk de lengte van de invoer?

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

Algoritme:

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

Lengte invoer: voorbeeld

De worst case complexiteit van dit algoritme is $O(n)$ (indien we de berekening van n % m voor 1 stap tellen, anders $O(n^2)$).

De lengte van de invoer is het aantal karakters van de codering, in dit geval van het getal n. Als we de unaire codering gebruiken is het algoritme dus lineair in de lengte van de invoer. Nemen we de binaire codering, of in het algemeen de l-aire codering met $l > 1$, dan is het algoritme exponentieel in de lengte van de invoer (voor elke $l > 1$ dus).

Vraag: is het algoritme nu polynomiaal of exponentieel?

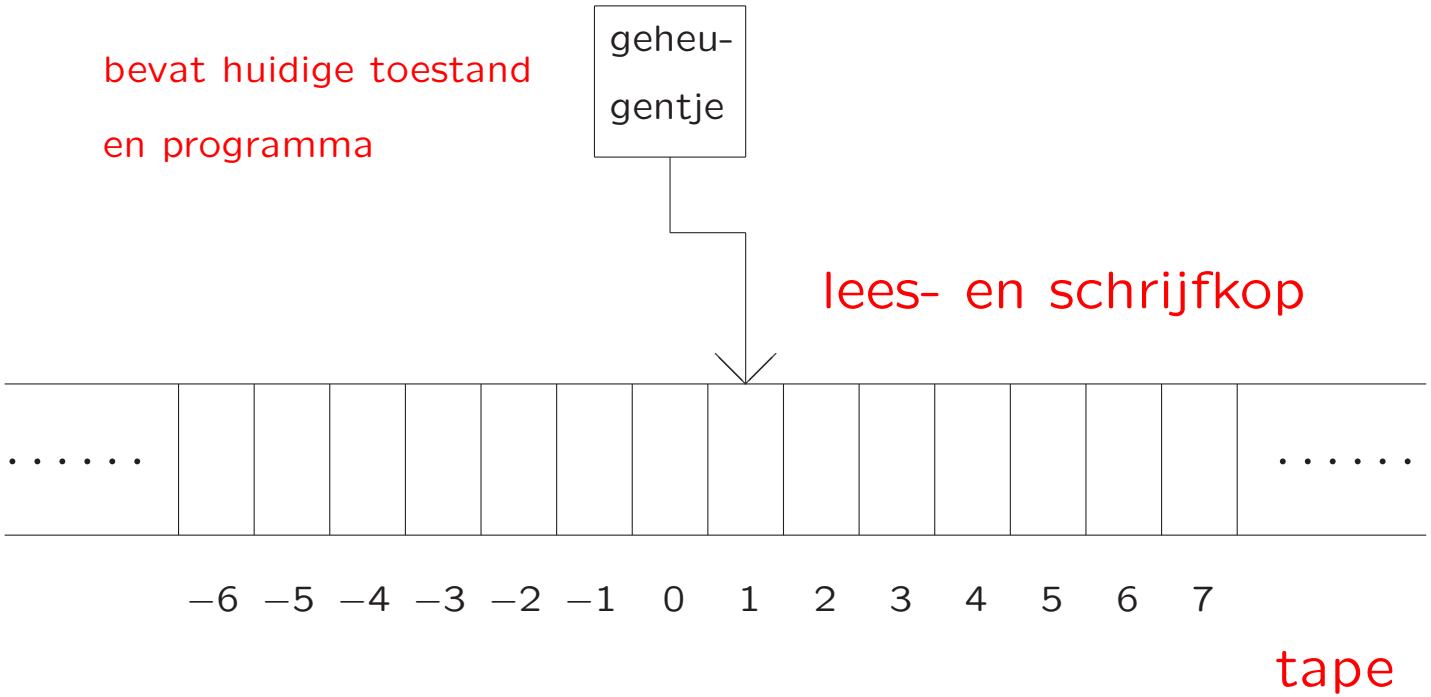
Knapzakprobleem

Gegeven een knapzak met capaciteit S (een geheel getal > 0) en n objecten met gewicht $s_1, s_2, \ldots s_n$ 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 $O(n * S)$ (dynamisch programmeren, zie Algoritmiek).

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

Deterministische one-tape Türing machine (DMT)



Een DTM-programma bevat:

- Γ: een eindige verzameling tape-symbolen (inputsymbolen Σ en blanco). Voorbeeld: $\Gamma = \{0, 1, b\}$.

- Q: een eindige verzameling toestanden, waaronder een begintoestand q_0 en twee eindtoestanden q_Y en q_N. Voorbeeld: $Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$.

- $\delta : Q - \{q_Y, q_N\} \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 0, 1\}$ een transitiefunctie die bepaalt wat er gebeurt als in een bepaalde toestand een bepaald karakter wordt gelezen.

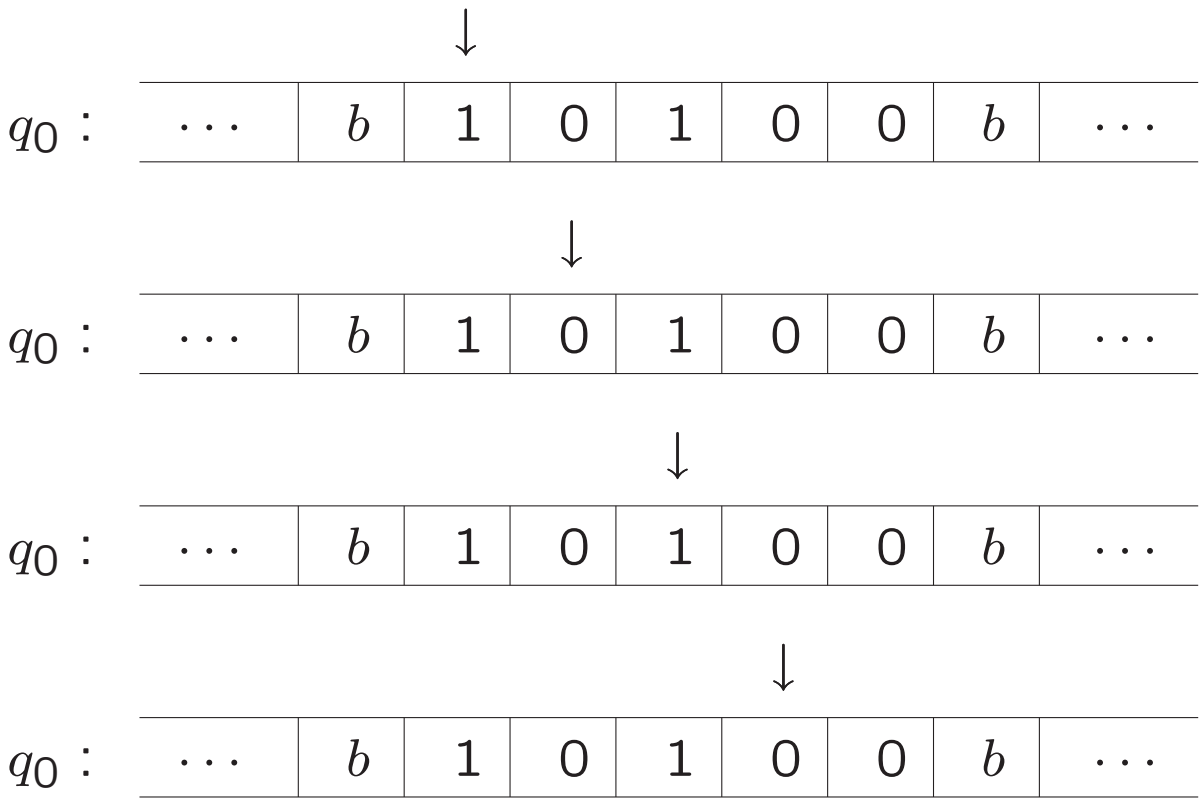
In de begintoestand staat de invoerstring x op plek 1 t/m $|x|$ en de rest van de tape is blanco. Het programma start in toestand q_0 met de lees- en schrijfkop op positie 1 en stopt als de toestand q_y (yes) of q_N (no) bereikt wordt.

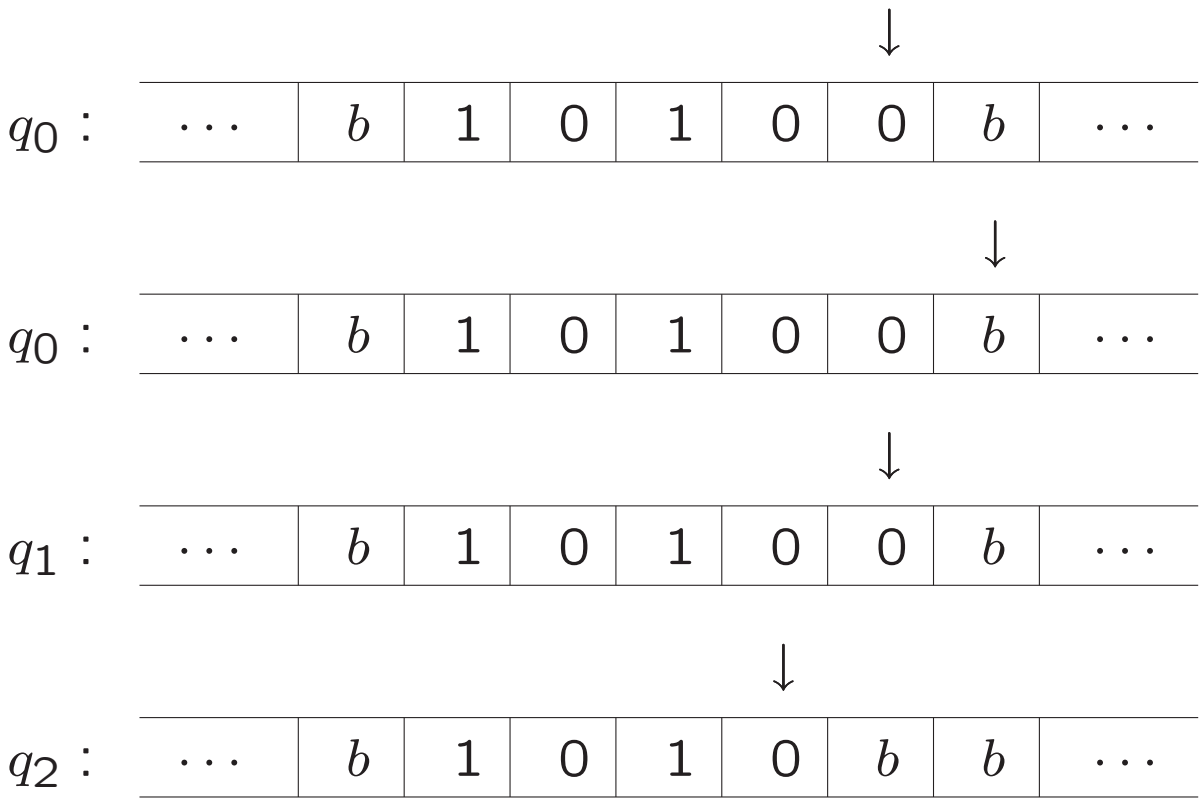
$$\Gamma = \{0, 1, b\}, \Sigma = \{0, 1\}$$

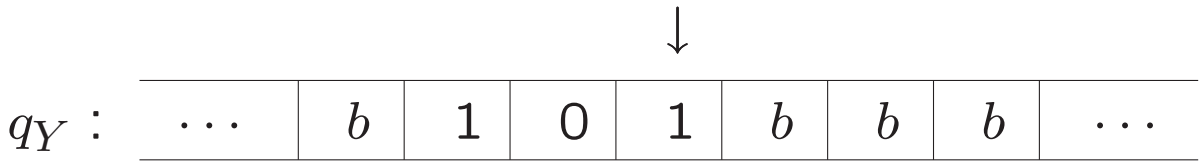
$$Q = \{q_0, q_1, q_2, q_3, q_Y, q_N\}$$

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

$$\delta(q, s)$$


q_0 : ⋯ b 1 0 1 0 0 b ⋯
q_0 : ⋯ b 1 0 1 0 0 b ⋯
q_0 : ⋯ b 1 0 1 0 0 b ⋯
q_0 : ⋯ b 1 0 1 0 0 b ⋯


q_0 : ⋯ b 1 0 1 0 0 b ⋯
q_0 : ⋯ b 1 0 1 0 0 b ⋯
q_1 : ⋯ b 1 0 1 0 0 b ⋯
q_2 : ⋯ b 1 0 1 0 b b ⋯



Vraag: welk beslissingsprobleem lost dit programma op?

Een certificaat kun je gebruiken om aan te tonen dat een ja-instantie van een probleem inderdaad een ja-instantie is.

Veel beslissingsproblemen zijn geformuleerd als "er-is-een"-vragen. In dat geval kun je bij een certificaat denken aan een kandidaat-oplossing (voorgestelde oplossing) van het probleem. Daarvan moet dan onder andere geverifieerd worden dat hij aan de criteria van het probleem voldoet, dus een *echte* oplossing is.

Probleem	Certificaat
HC	Hamiltonkring
SAT	waarmakende waardering
Kliek	kliek met k knopen
Kleur	kleuring met $\leq k$ kleuren
Subset Sum	deelverzameling met som $= t$
TSP	Hamiltonkring met totaalgewicht $\leq k$

Een niet-deterministisch algoritme bestaat uit 3 fasen:

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 het certificaat, het is een soort gok
   // van de oplossing van het probleem;
   // s kan echter ook een onzinstring zijn.

2. Deterministische verificatiefase
   Zowel de invoer van het probleem als de string s mogen hier gebruikt worden. Er wordt True of False geretourneerd of het programma gaat in een oneindige loop en stopt nooit.

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

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

Het aantal stappen dat een niet-deterministisch algoritme doet is het aantal stappen nodig om s te schrijven (dus het aantal karakters waaruit s bestaat) + het aantal stappen dat gedaan wordt in de verificatiefase (+1 voor fase 3).

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

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

* dus een string s

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

Definitie
$\mathcal{NP}$ is de klasse van beslissingsproblemen waarvoor er een polynomiaal begrensd niet-deterministisch algoritme bestaat.

$\mathcal{NP}$ betekent: $\mathcal{N}$on-deterministic $\mathcal{P}$olynomial time.

Twaalfde college complexiteit

14/15 april 2008

NP-volledigheid III

Definitie.
Een algoritme heet polynomiaal begrensd als zijn worst case complexiteit van boven begrensd is door een functie die polynomiaal is in de lengte van de invoer.
Dus: worst case is $O(|x|^k)$ met $k \geq 0$ en met $|x|$ de lengte van de invoer x (of $O(n^k)$ met n een maat voor de lengte van de invoer).

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

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

Een niet-deterministisch algoritme bestaat uit 3 fasen:

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 (hopelijk) het certificaat, het is een
   // soort gok van de oplossing van het probleem;
   // s kan echter ook een onzinstring zijn.

2. Deterministische verificatiefase
   Zowel de invoer van het probleem als de string s mogen hier gebruikt worden. Er wordt True of False geretourneerd of het programma gaat in een oneindige loop en stopt nooit.

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

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

Het aantal stappen dat een niet-deterministisch algoritme doet is het aantal stappen nodig om s te schrijven (dus het aantal karakters waaruit s bestaat) + het aantal stappen dat gedaan wordt in de verificatiefase (+1 voor fase 3).

Een niet-deterministisch algoritme A antwoordt "ja" op invoer $x \Longleftrightarrow$ er bestaat een executie (dus een string s) van A op x die "ja" als uitvoer geeft. De invoer x is dan een ja-instantie. Als er geen ja-executie bestaat is het antwoord "nee" (x is dan een nee-instantie).

A is polynomiaal (begrensd) als er een ja-executie bestaat* die polynomiaal is in $|x|$. Dit is het geval als voor zo'n ja-executie $|s| = O(|x|^l)$ en Fase 2 polynomiaal is in $|x|$ en $|s|$ (dus $O(|x|^k \cdot |s|^m)$).

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

*die kan dus alleen bestaan voor ja-instanties

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 precies k kleuren gebruikt (ofwel, is $\mathcal{G}$ k-kleurbaar)?
Probleeminvoer: $x =< \mathcal{G}, k >$.

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

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

Een polynomiaal begrensd *niet-deterministisch algoritme* voor HC:

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

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

   (1) controleer dat er precies n integers staan: $O(|s|)$

   (2) controleer dat elke integer tussen 1 en n is: $O(|s|)$

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

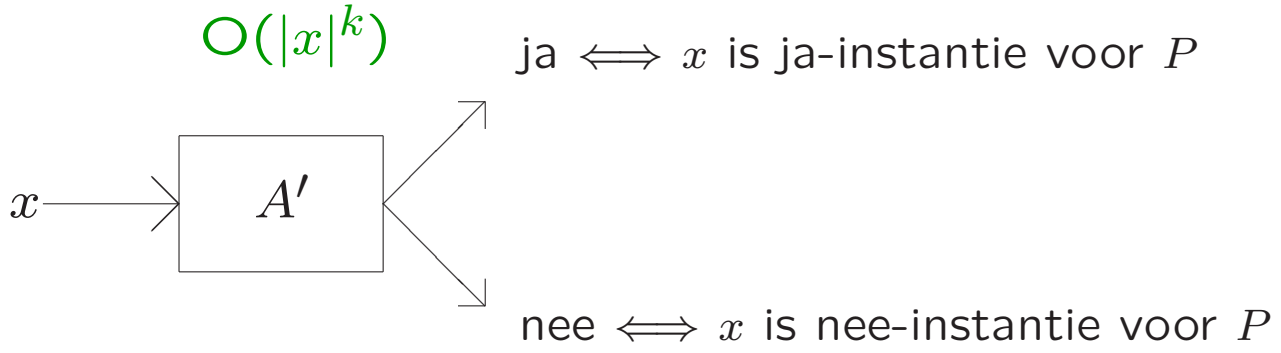
(4) controleer dat tussen opeenvolgende knopen uit s een tak zit in de graaf (en tussen de laatste en de eerste): $O(|s| \cdot |E|) = O(|s| \cdot |\mathcal{G}|)$.

Als de vier tests positief zijn wordt True geretourneerd, anders False (of er wordt in een oneindige loop gegaan).

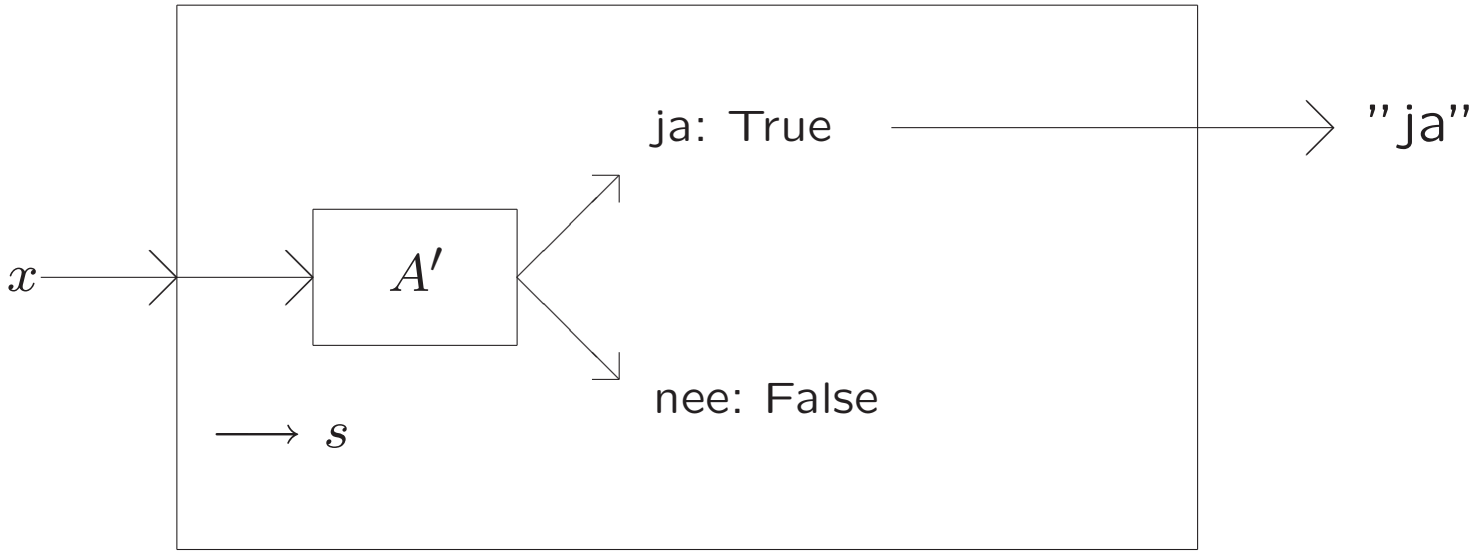
3. Fase 3 (uitvoerfase)
   Als fase 2 True oplevert wordt "ja" uitgevoerd, anders geen uitvoer.

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

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.



fase 1: genereer s;

fase 2: negeer de string s en voer A' uit op x;

fase 3:

Laat $V = \{1, 2, \ldots, n\}$, en de mogelijke kleuren $1, 2, \ldots, k$. Een polynomiaal begrensd *niet-deterministisch algoritme* voor Kleur 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 de kleur van knoop i):

   (1) controleer dat er precies n integers staan (elke knoop een kleur): kan p[olynomiaal, O($|s|$)

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

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

Als de vier tests positief zijn wordt True geretourneerd, zodra een test negatief uitvalt wordt False teruggegeven (of er wordt in een oneindige loop gegaan).

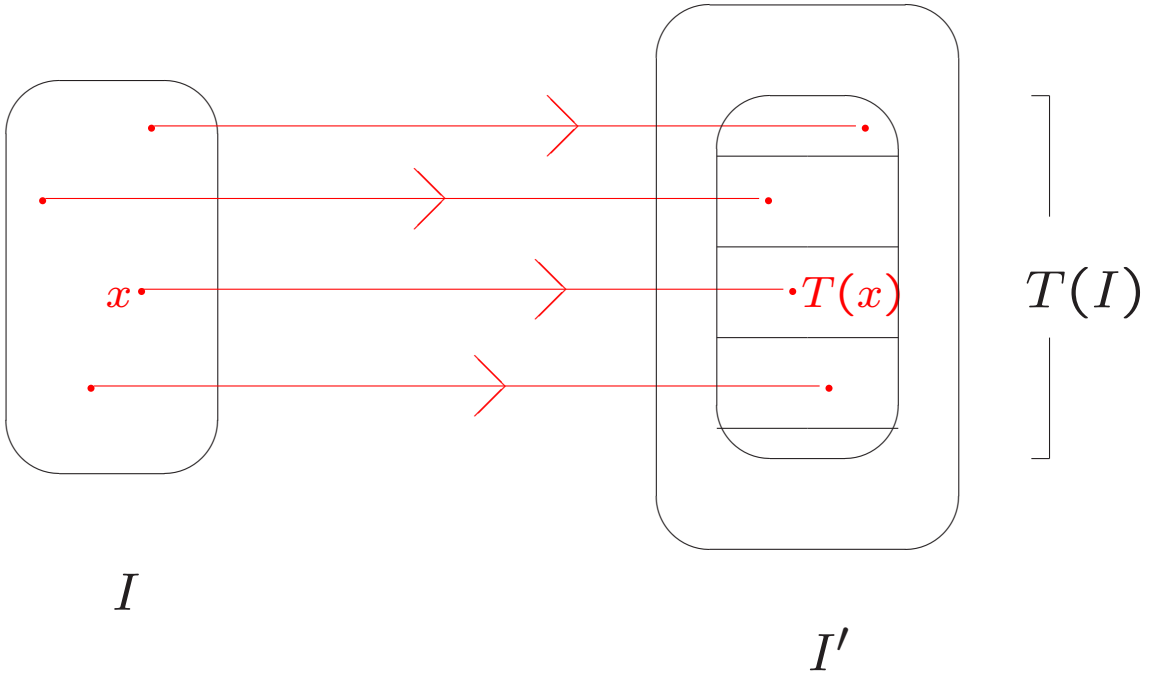
3. Fase 3 (uitvoerfase)
   Als fase 2 True oplevert wordt "ja" uitgevoerd, anders geen uitvoer.

Merk op. Stap (1) t/m (2) controleert dat s een kleuring van de knopen voorstelt stap (3) controleert dat het een correcte kleuring is.

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

Verder: voor een ja-executie, dus met s een goede kleuring, is $|s| = \mathrm{O}(|V|) = \mathrm{O}(|x|)$. Ergo: A is polynomiaal begrensd.

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



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

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

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.)

Definitie

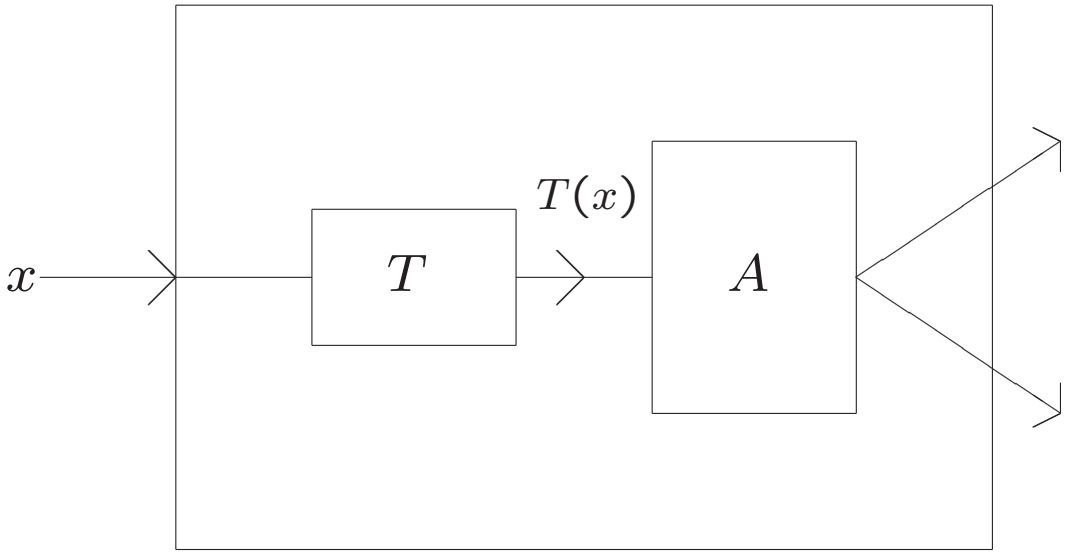
Een probleem P is polynomiaal reduceerbaar (of polynomiaal transformeerbaar) naar Q als er een polynomiale reductie bestaat van P naar Q.

Notatie: $P \leq_P Q$.

Stelling

Als $P \leq_P Q$ en Q zit in $\mathcal{P}$, dan zit P ook in $\mathcal{P}$.

Bewijs stelling



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

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

$O(|x|^k)$ $\qquad$ $O(|y|^l)$

$O(|x|^{kl})$

Definitie
Een probleem Q is NP-hard als elk probleem P in $\mathcal{NP}$ polynomiaal reduceerbaar is tot Q, dat wil dus zeggen dat $P \leq_P Q$ 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).

HP: **gegeven** een ongerichte graaf $\mathcal{G} = (V, E)$.
Vraag: heeft $\mathcal{G}$ een Hamiltonpad?

HC: **gegeven** een ongerichte graaf $\mathcal{G} = (V, E)$.
Vraag: heeft $\mathcal{G}$ een Hamiltonkring?

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

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

Opmerking: er geldt ook: HC2 $\leq_P$ HC1. Bedenk zelf een eenvoudige reductie.

Transformatie T die een gerichte graaf $\mathcal{G} = (V, E)$ op een ongerichte graaf $T(\mathcal{G}) = \mathcal{G}' = (V', E')$ afbeeldt:

- $V' = \{v_1, v_2, v_3 : v \in V\}$: elke knoop $v \in V$ wordt afgebeeld op een drietal knopen v_1, v_2, v_3.
- $E' = \{(v_1, v_2), (v_2, v_3) : v \in V\} \cup \{(v_3, w_1) : (v, w) \in E\}$: 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 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 berekend worden (constructie van $T(\mathcal{G})$ uit $\mathcal{G}$ is $O(|\mathcal{G}|)$)
2. $\mathcal{G}$ is een ja-instantie voor HC1 $\iff$ $\mathcal{G}'$ is een ja-instantie voor HC2, ofwel: $\mathcal{G}$ heeft een gerichte Hamiltonkring $\iff$ $\mathcal{G}'$ heeft een ongerichte Hamiltonkring

Laatste week

- maandag 21 april: 11.15-13.00 college
- maandag 21 april: 13.45-15.30 laatste werkcollege
- dinsdag 22 april: 11.15-13.00 laatste college
- dinsdag 6 mei: 11.15-13.00 KI, dus geen wCom
- tentamen: donderdag 5 juni 2008, 14.00-17.00
- vragenuur: ?????

Dertiende college complexiteit

21 april 2008

NP-volledigheid IV

$P \leq_P Q$ betekent dat er een polynomiale reductie T van P naar Q bestaat:

1. T beeldt elke invoer x van P af op een invoer $T(x)$ van Q.

2. De constructie van $T(x)$ uit x is polynomiaal ($O(|x|^k)$).

3. Voor elke x uit I (= invoerverzameling van P) geldt: x is een ja-instantie voor $P \iff T(x)$ is een ja-instantie voor Q.

Stelling
Als $P \leq_P Q$ en $Q \in \mathcal{P}$, dan $P \in \mathcal{P}$.

Definitie
Een probleem Q is NP-hard als elk probleem P in $\mathcal{NP}$ polynomiaal reduceerbaar is tot Q, dat wil dus zeggen dat $P \leq_P Q$ voor alle $P \in \mathcal{NP}$.

Definitie
Een probleem Q is NP-volledig als

1. $Q \in \mathcal{NP}$

2. Q is NP-hard

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

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

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 $\mathcal{NP}$ niet polynomiaal begrensd is, dan zijn *alle* NP-volledige problemen niet polynomiaal begrensd.

Lemma

$\leq_P$ is transitief, dat wil zeggen: als $P_1 \leq_P P_2$ en $P_2 \leq_P P_3$ dan is $P_1 \leq_P P_3$.

Stelling

Stel P is een probleem waarvoor geldt dat $Q \leq_P P$ voor een of andere $Q \in \mathcal{NPC}$. Dan is P NP-hard.
Als bovendien $P \in \mathcal{NP}$, dan geldt dat $P \in \mathcal{NPC}$.

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

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

Stap 3 valt uiteen in:

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

3b. Laat zien dat $T(x)$ uit x geconstrueerd kan worden 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 Q $\iff$ $T(x) \in I'$ is een ja-instantie voor P

In 1971 bewees Stephen Cook op een directe manier (dus door een reductie te geven van alle problemen uit $\mathcal{NP}$ naar SAT) dat SAT NP-volledig is.

Stelling
Gegeven een willekeurig probleem $P \in \mathcal{NP}$. Dan is P reduceerbaar tot SAT: $P \leq_P$ SAT.

Sindsdien is met behulp van de reductiemethode van zeer veel bekende problemen aangetoond dat ze NP-volledig zijn. Bijvoorbeeld voor enige voorbeeldproblemen:

$$\text{SAT} \leq_P \text{3SAT} \leq_P \text{Kliek} \leq_P \text{VC}$$
$$\text{3SAT} \leq_P \text{HC} \leq_P \text{TSP}$$
$$\text{SAT} \leq_P \text{3Kleur} \leq_P \text{4Kleur}$$

Schets van het bewijs

1. Omdat $P \in \mathcal{NP}$, is er een niet-deterministisch algoritme A (een niet-deterministische Türingmachine) voor P. Verder is A polynomiaal begrensd ($O(|x|^k)$).
2. Dit algoritme zal voor elke invoer x van P gemodelleerd worden als een logische formule $\phi = T(x)$ in CNF: deze ϕ beschrijft a.h.w. de berekening van A, werkend op x.
3. De formule ϕ is weliswaar lang, maar kan in hooguit $O(|x|^l)$ stappen geconstrueerd worden.
4. Een ja-executie vergt voor ja-instanties x hooguit $N = c \cdot |x|^k$ stappen (want A is polynomiaal begrensd).
5. Een waarmakende waardering voor ϕ correspondeert precies met een executie van A die een "ja" produceert (voor zekere string s dus).

Boolese variabelen in ϕ:

${Q_i}^q$: 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, $0 \leq i, j \leq N$

${S_{ij}}^a$: op tijdstip i bevat cel j symbool $a \in \Sigma$, $0 \leq i, j \leq N$

De formule ϕ is een conjunctie van:

- ${Q_0}^{\texttt{start}} \wedge \cdots \wedge H_{00} \wedge {S_{01}}^{x_1} \wedge {S_{02}}^{x_2} \wedge {S_{03}}^{x_3} \cdots$

  de machine start in toestand `start`;

  x bevindt zich op de posities 1 t/m $|x|$; ...

- $Q_N^{q_Y} \wedge H_{N0} \wedge \cdots$

  op tijdstip N stopt de berekening in de ja-toestand

- $\wedge_{0 \leq i \leq N}(\vee_{q \in Q}\ {Q_i}^q) \wedge \wedge_{0 \leq i \leq N}(\wedge_{p,q \in Q, p \neq q}(\neg {Q_i}^p \vee \neg {Q_i}^q))$

  te allen tijde is de machine in precies één toestand

- $\wedge_{0 \leq i,j \leq N}(\vee_{a \in \Sigma}\ {S_{ij}}^a) \wedge \wedge_{0 \leq i,j \leq N}(\wedge_{a,b \in \Sigma, a \neq b}(\neg {S_{ij}}^a \vee \neg {S_{ij}}^b))$

  te allen tijde bevat elke cel precies één symbool

- $\wedge_{0 \leq i \leq N}(\vee_{0 \leq j \leq N}\ H_{ij}) \wedge \wedge_{0 \leq i \leq N}(\wedge_{0 \leq j < k \leq N}(\neg H_{ij} \vee \neg H_{ik}))$

  te allen tijde scant de machine precies één cel

- $Q_i^{\,p} \wedge H_{ij} \wedge S_{ij}^{\,a} \longrightarrow$
  $\vee_{((p,a),(q,b,d))\in\delta}\ (Q_{i+1}^{\,q} \wedge H_{i+1,j+d} \wedge S_{i+1,j}^{\,b})$

  elke stap van de machine verloopt volgens de transitiefunctie δ

- $S_{ij}^{\,a} \wedge \neg H_{ij} \longrightarrow S_{i+1,j}^{\,a}$

  elke cel die op tijdstip i niet gescand wordt bevat op tijdstip $i+1$ hetzelfde symbool

Een waarmakende waardering correspondeert aldus precies met een echte executie van de niet-deterministische Türingmachine die eindigt in "ja" na een polynomiaal (nl. N) aantal stappen.

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

Opmerking: er geldt ook: HC2 $\leq_P$ HC1. Bedenk zelf een eenvoudige reductie.

Transformatie T die een gerichte graaf $\mathcal{G} = (V, E)$ op een ongerichte graaf $T(\mathcal{G}) = \mathcal{G}' = (V', E')$ afbeeldt:

- $V' = \{v_1, v_2, v_3 : v \in V\}$: elke knoop $v \in V$ wordt afgebeeld op een drietal knopen v_1, v_2, v_3.
- $E' = \{(v_1, v_2), (v_2, v_3) : v \in V\} \cup \{(v_3, w_1) : (v, w) \in E\}$: 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 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 berekend worden (constructie van $T(\mathcal{G})$ uit $\mathcal{G}$ is $O(|\mathcal{G}|)$)
2. $\mathcal{G}$ is een ja-instantie voor HC1 $\iff$ $T(\mathcal{G})$ is een ja-instantie voor HC2, ofwel: $\mathcal{G}$ heeft een gerichte Hamiltonkring $\iff$ $\mathcal{G}'$ heeft een ongerichte Hamiltonkring

3SAT

Gegeven een logische formule ϕ in 3-CNF. Bestaat er een waardering die ϕ True maakt?

Definitie

Een logische formule ϕ staat in 3-CNF als ϕ een conjunctie is van clausules, waarbij elke clausule een disjunctie is van drie verschillende literals.

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 $(u, v) \in E$. (M.a.w.: tussen elk tweetal knopen uit V' zit een tak.)

Er geldt: 3SAT $\leq_P$ Kliek. Om dit aan te tonen moeten we een logische formule in 3-CNF afbeelden op een ongerichte graaf.

Zij ϕ een logische expressie (formule) in 3-CNF, met zeg m clausules: $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$. Hierin is $C_r = l_1^r \vee l_2^r \vee l_3^r$ $(r = 1, \ldots, m)$ en l_1^r, l_2^r en l_3^r steeds verschillend bij vaste r.

Construeer nu een ongerichte graaf $\mathcal{G}_\phi = (V, E)$ als volgt.

Voor elke clausule C_r uit ϕ doen we 3 knopen v_1^r, v_2^r en v_3^r in V (deze corresponderen dus met l_1^r, l_2^r en l_3^r). $\mathcal{G}_\phi$ heeft derhalve $3m$ knopen.

Er komt een tak tussen twee knopen v_i^r en v_j^s als:

- v_i^r en v_j^s in verschillende drietallen zitten (dus $r \neq s$)
- de bijbehorende l_i^r en l_j^s zó zijn dat $l_i^r \neq \neg l_j^s$, met andere woorden: l_i^r en l_j^s zijn niet elkaars negatie

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

- T kan in polynomiaal begrensde tijd berekend worden.
- Er is een waardering die ϕ waarmaakt $\Longleftrightarrow$ $\mathcal{G}_\phi$ heeft een kliek ter grootte m.

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 $v_1^1 \leftrightarrow x_1, v_2^1 \leftrightarrow \neg x_2, v_3^1 \leftrightarrow \neg x_3$; alle uit clausule C_1, etcetera

- een waardering w die ϕ 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 ϕ waar.

(*) De bovenindex geeft aan met welke clausule een knoop correspondeert.

SAT

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

Definitie

Een logische formule ϕ staat in Conjunctive Normal Form als ϕ een conjunctie is van clausules, waarin een clausule een disjunctie is van literals.

3SAT

Gegeven een logische formule ϕ in 3-CNF. Bestaat er een waardering die ϕ True maakt?

Definitie

Een logische formule ϕ staat in 3-CNF als ϕ een conjunctie is van clausules, waarbij elke clausule een disjunctie is van drie verschillende literals.

Er geldt: SAT $\leq_P$ 3SAT. Om dit aan te tonen moeten we een logische formule ϕ in CNF afbeelden op een logische formule ϕ' in 3-CNF. We gaan er voor het gemak van uit dat de $l_1, l_2, \ldots, l_k$ per clausule al verschillend zijn (kan in $O(|\phi|^2)$ worden bewerkstelligd). Op clausuleniveau werkt de transformatie T als volgt:

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

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

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

Hierin zijn $\widetilde{l_{k+1}}, \widetilde{l_{k+2}}, \ldots, \widetilde{l_{2k-3}}$ 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:

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

Voor deze transformatie T geldt:

- De constructie van $T(\phi)$ uit ϕ kan met een polynomiaal begrensd ($= O(|\phi|^k)$) algoritme.

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

- Ofwel: er is een waardering die ϕ waarmaakt $\Longleftrightarrow$ er is een waardering die $T(\phi)$ waarmaakt.

- Conclusie uit de vorige punten: SAT $\leq_P$ 3SAT.

- maandag 21 april: 13.45-15.30 laatste werkcollege
- dinsdag 22 april: 11.15-13.00 laatste college: oud tentamen
- dinsdag 6 mei: 11.15-13.00 KI, dus geen wCom
- tentamen: donderdag 5 juni 2008, 14.00-17.00
- vragenuur: ?????