Zesde college complexiteit

19 maart 2019

Mergesort, Ondergrens sorteren

Quicksort, Shellsort

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

- # arrayvergelijkingen $\geq$ # inversies invoerarray

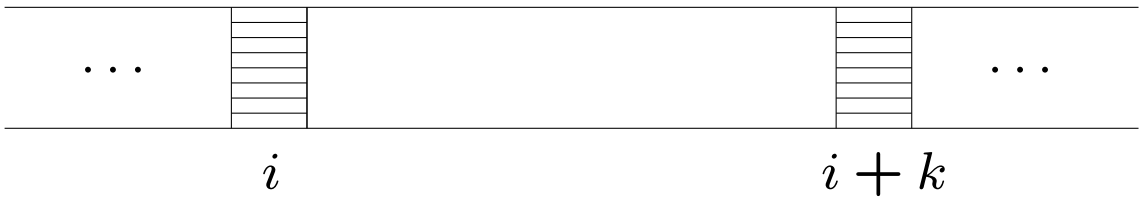
- # arrayvergelijkingen in de worst case $\geq \frac{1}{2}n(n-1)$

Als je een beter sorteeralgoritme (gebaseerd is op arrayvergelijkingen) wilt, moet je elementen verwisselen die verder van elkaar liggen, zoals Mergesort, Quicksort, Shellsort.

*dit is het geval bij algoritmen die gebruikmaken van buurverwisselingen, zoals Insertion sort en Bubblesort

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

Situatie:



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

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

```
Sorteer(rij)::
    if ( de rij heeft meer dan één element ) then
        Verdeel de rij in twee stukken: linkerrij en rechterrij;
        Sorteer(linkerrij);
        Sorteer(rechterrij);
        Combineer linkerrij en rechterrij;
    fi .
```

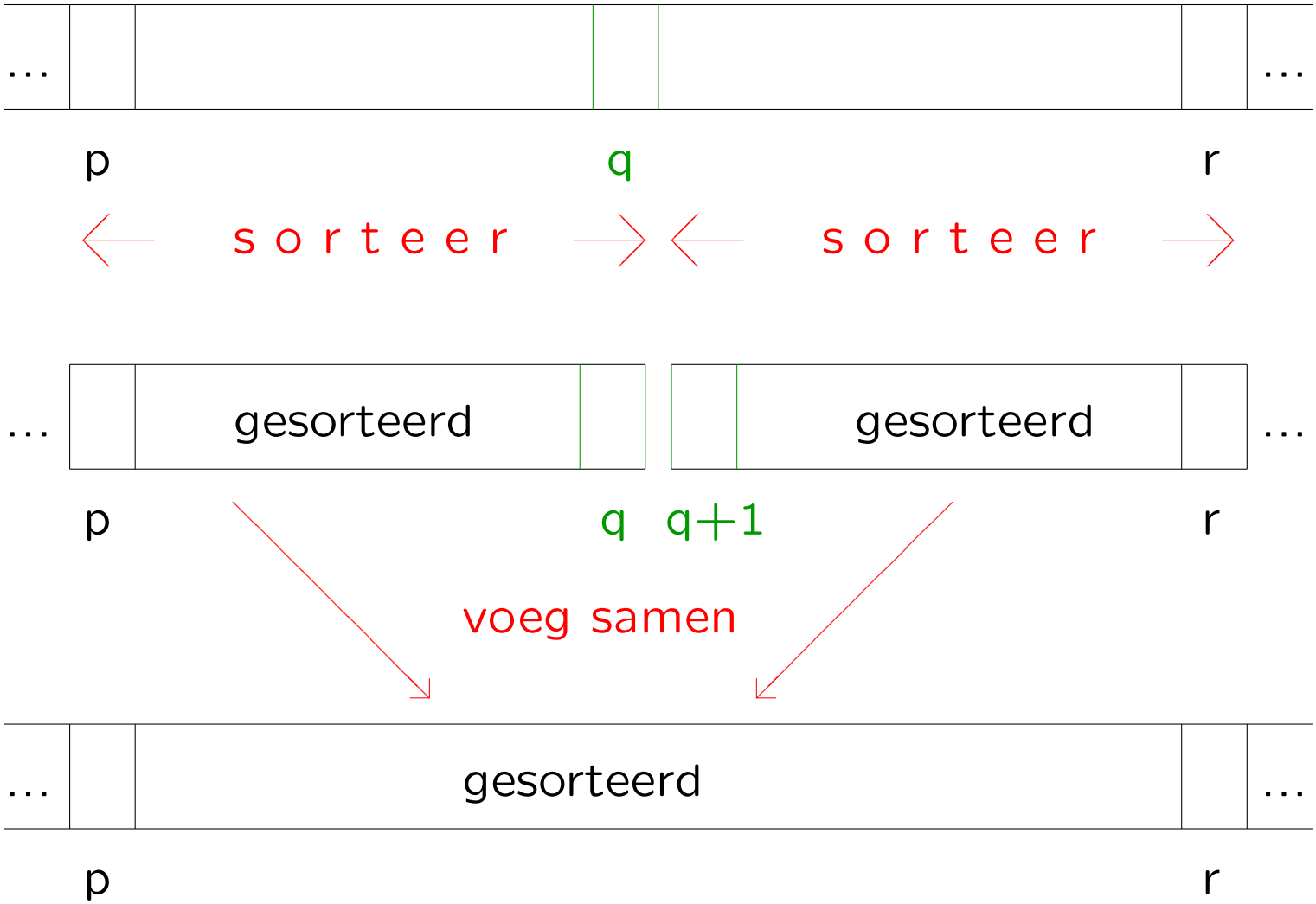
Mergesort stopt het meeste werk in de Combineer-stap, Quicksort in de Verdeel-stap. Beide sorteermethoden zijn gebaseerd op arrayvergelijkingen, maar doen geen buurverwisselingen, zoals Insertion sort en Bubblesort.

Het (recursieve) Mergesort algoritme:

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

if $p < r$ **then**
$q := \lfloor \frac{p+r}{2} \rfloor$;
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`$[r]$ 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)

- Geheel analoog kan een functie Merge(A, B, C, k, m) geschreven worden die twee gesorteerde rijen A (k elementen) en B (m elementen) samenvoegt tot de gesorteerde rij C ($n = k + m$ elementen)

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

- Er worden altijd $2n$ verplaatsingen van array-elementen gedaan

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

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

1. Het aantal vergelijkingen in de worst case is $n - 1$

2. Het aantal vergelijkingen in de best case is $\min\{k, m\}$

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

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

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

Mergesort is in orde van grootte optimaal voor wat betreft de worst case (immers: de ondergrens voor sorteren via arrayvergelijkingen is $\Omega(n \lg n)$). 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: $B(n) = \frac{n}{2} \lg n \in \Theta(n \lg n)$.

Stelling

1. *Elk* algoritme gebaseerd op arrayvergelijkingen dat twee gesorteerde arrays (rijen) van lengte m samenvoegt tot één gesorteerd array, doet in het slechtste geval ten minste $2m-1$ van zulke vergelijkingen.
   Voor $m = \frac{n}{2}$ (n even) is dit dus ten minste $n-1$.

2. Voor het samenvoegen van twee rijtjes 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 Merge-algoritme optimaal, althans voor twee ongeveer even lange rijtjes.

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

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

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

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.

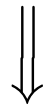
We bekijken sorteeralgoritmen gebaseerd op het doen van vergelijkingen van de vorm $A[i] < A[j]$.

Aannames (z.b.d.a.):

- 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 (ter grootte n). Elk pad van de wortel tot een blad correspondeert met een executie van het algoritme.

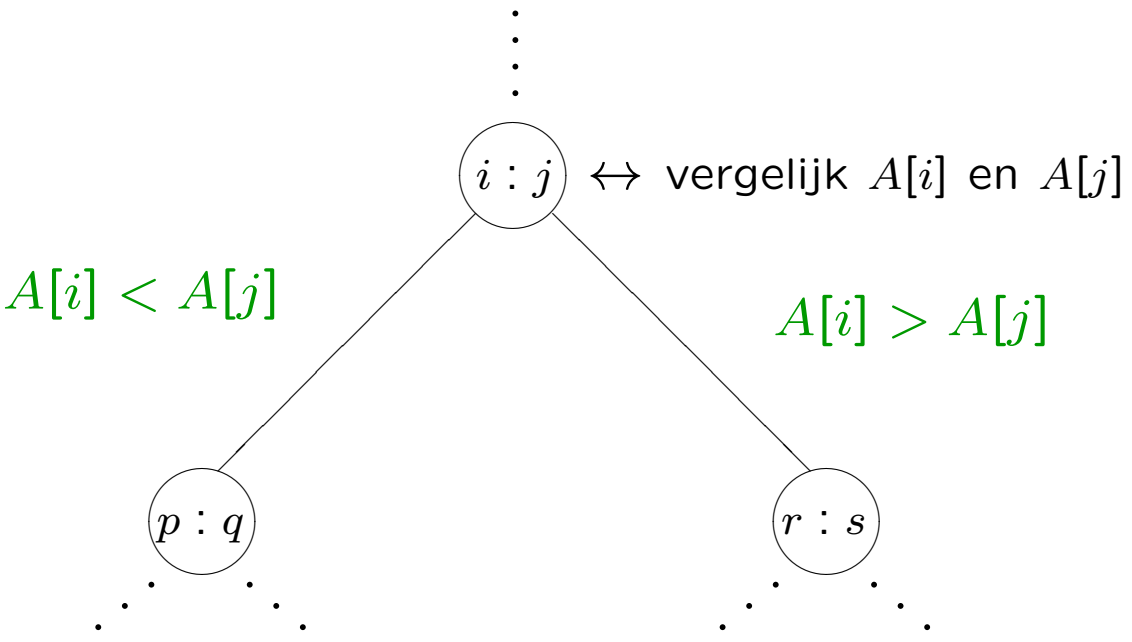
algoritme gebaseerd op het doen van arrayvergelijkingen $A[i] < A[j]$



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

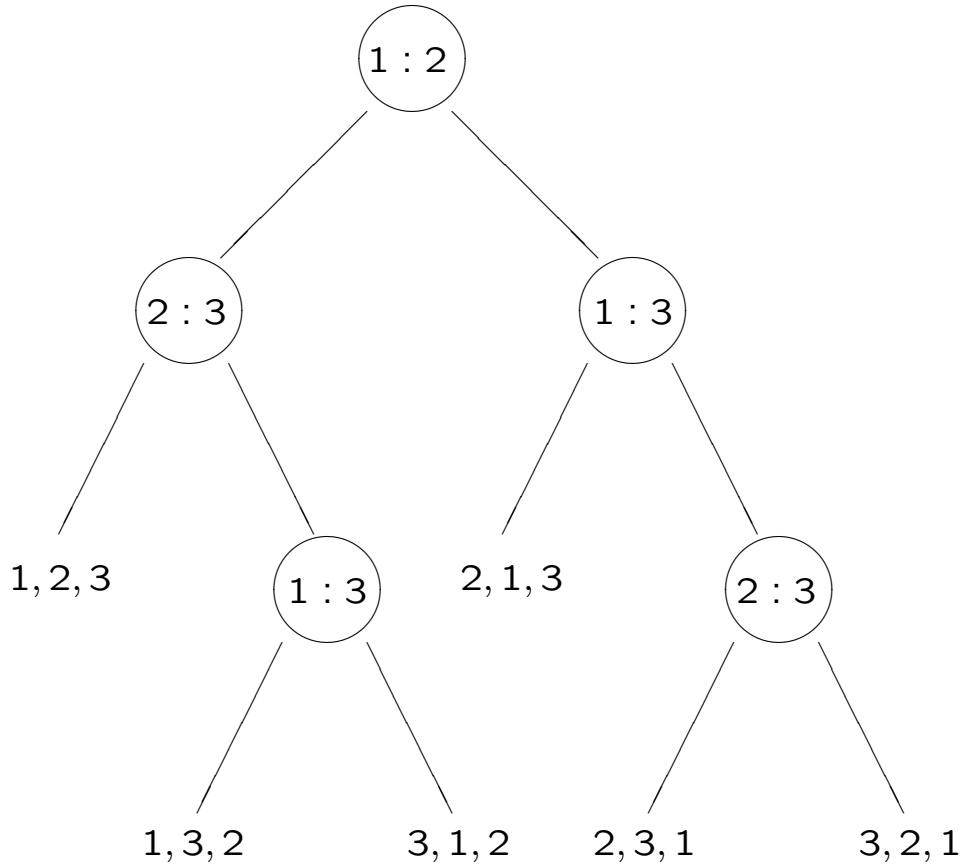
†eindresultaat = de gevonden sortering/ordening (in dit geval)



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

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

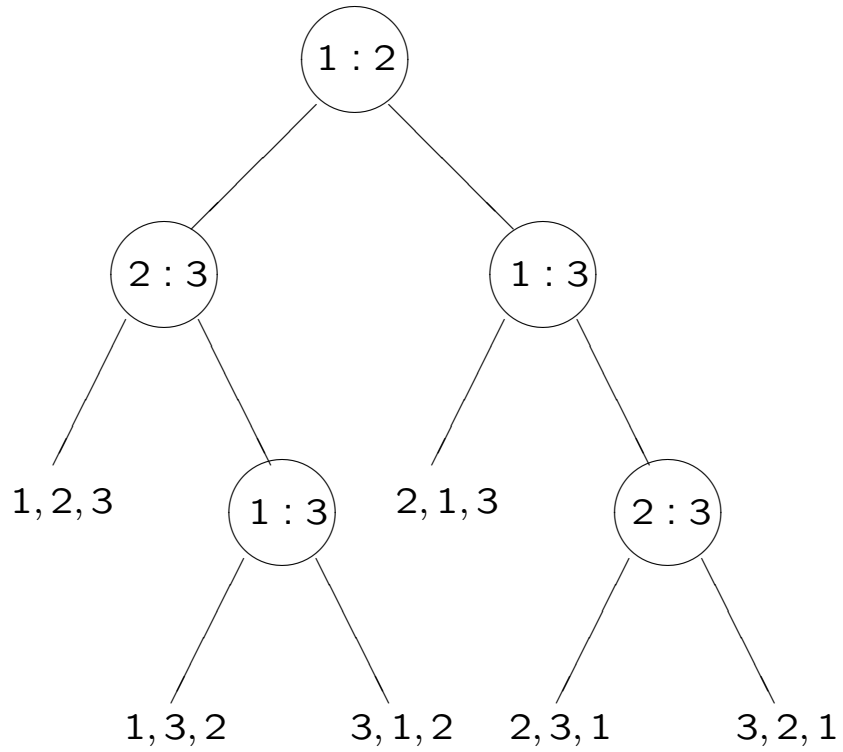
Beslissingsboom voor algoritmen gebaseerd op arrayvergelijkingen



Beslissingsboom voor Insertion sort met $n = 3$

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

Hoogte $\leftrightarrow$ worst case



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

Bladeren*: maximaal

1. - alleen de onderlinge volgorde van de array-elementen wordt onderscheiden; niet de waarde

   - het rijtje $6, 11, 15, 8, 3$ wordt bijvoorbeeld precies zo behandeld door het sorteeralgoritme als het rijtje $2, 4, 5, 3, 1$

   - ze volgen dan ook precies hetzelfde pad in de beslissingsboom

   - er zijn in essentie $n!$ mogelijke te onderscheiden invoeren, die elk één pad volgen in de boom $\Rightarrow$ er zijn maximaal $n!$ bladeren

*algemeen

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 gebaseerd op arrayvergelijkingen heeft precies $n!$ bladeren (n = aantal array-elementen)

*voor *sorteren*

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.

De stelling volgt direct uit de resultaten van de volgende sheet.

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

†Hiervoor gebruik je dat het aantal bladeren gelijk is aan $n!$

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:

$$E = \Sigma_{\text{bladeren}}(\text{lengte pad wortel} \longrightarrow \text{blad})$$

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

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

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

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

```
Sorteer(rij)::
    if ( de rij heeft meer dan één element ) then
        Verdeel de rij in twee stukken: linkerrij en rechterrij;
        Sorteer(linkerrij);
        Sorteer(rechterrij);
        Combineer linkerrij en rechterrij;
    fi .
```

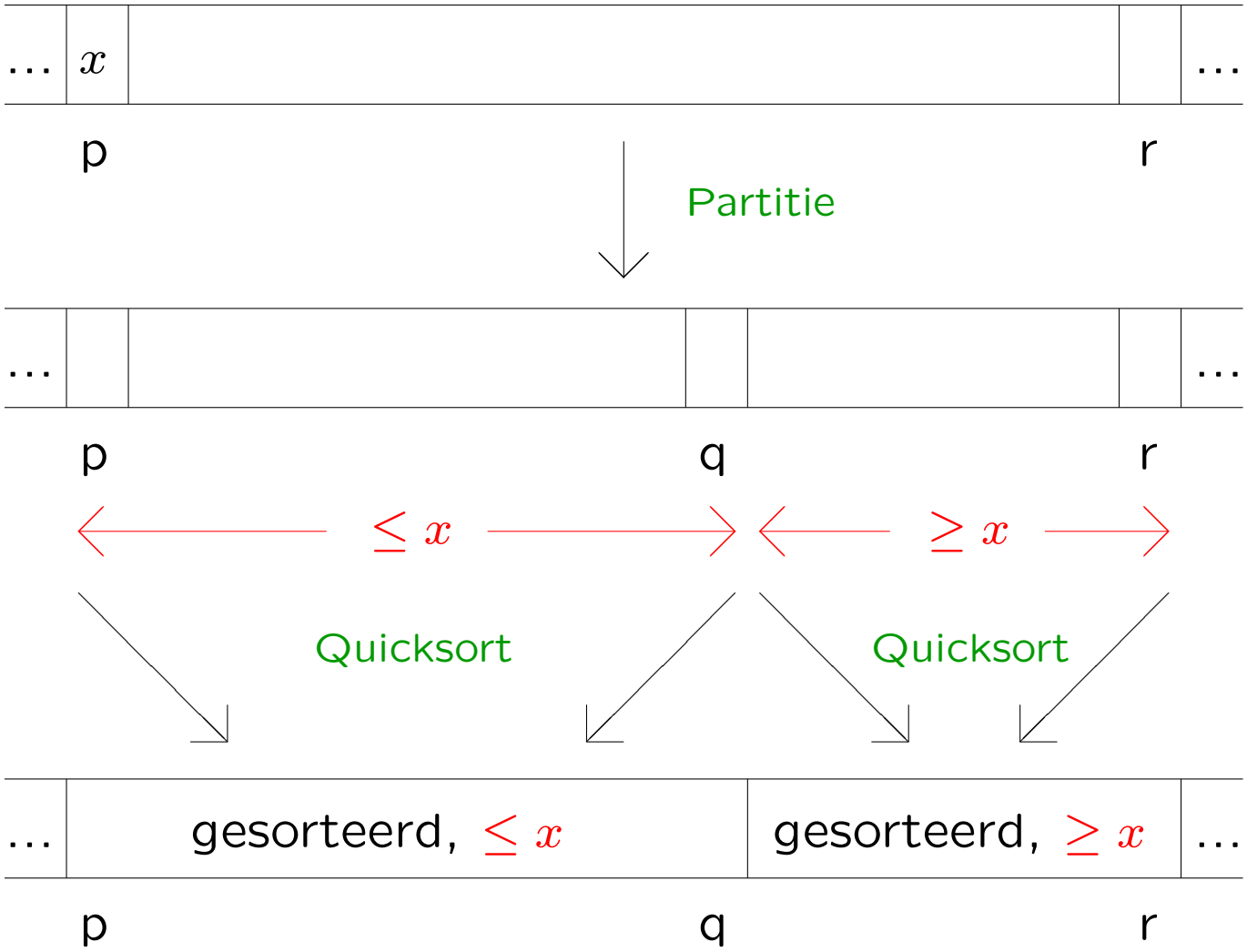
Mergesort stopt het meeste werk in de Combineer-stap, Quicksort in de Verdeel-stap. Beide sorteermethoden zijn gebaseerd op arrayvergelijkingen, maar doen geen buurverwisselingen, zoals Insertion sort en Bubblesort.

Quicksort (Hoare 1962)

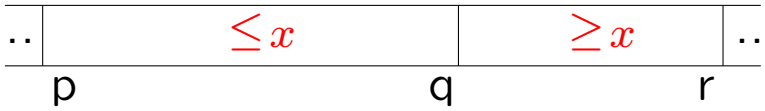
```
QuickSort(A, p, r)::
// sorteert A[p], ..., A[r] oplopend
        if p < r then
                q := Partitie(A, p, r);
                QuickSort(A, p, q);
                QuickSort(A, q + 1, r);
        fi
```

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
$\leq x$
$\geq x$
Quicksort
Quicksort
...
gesorteerd, $\leq x$
gesorteerd, $\geq x$
...
p
q
r

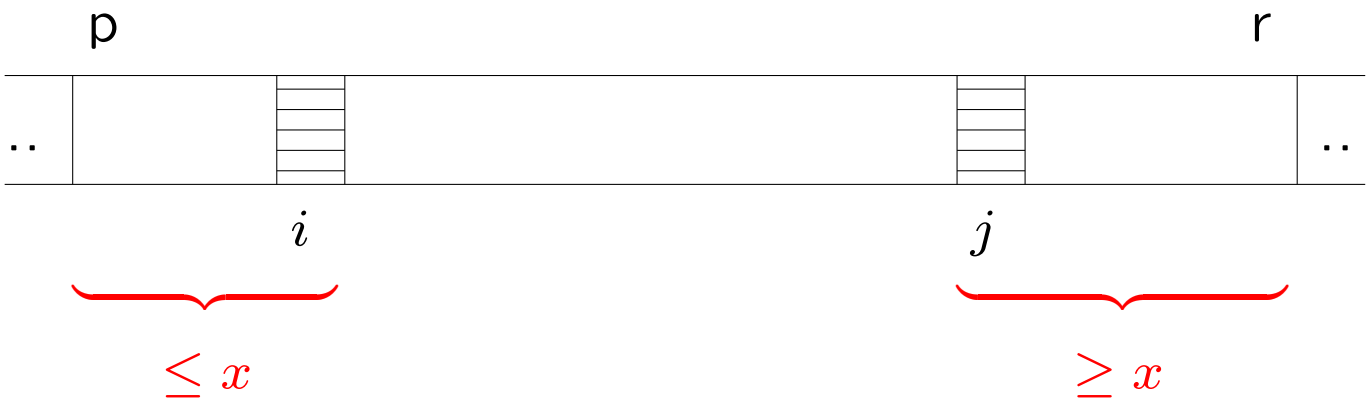
Partitie (A, p, r)::
// reorganiseert het (deel)array $A[p], \ldots, A[r]$ als volgt:



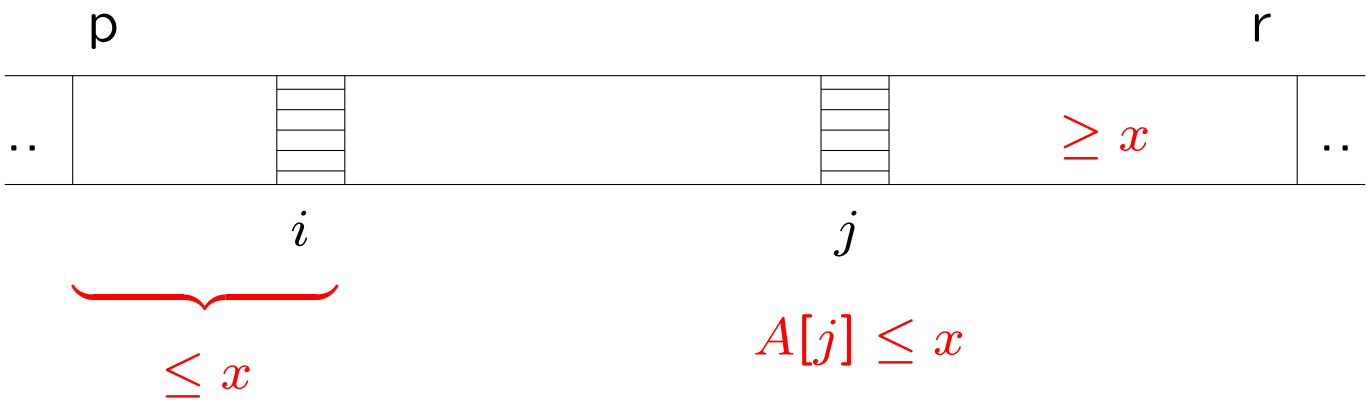
(*) // hier komt nog wat
$x = A[p]$; $i := p - 1$; $j := r + 1$;
while $i < j$ **do**
    $j := j - 1$; // loop met j naar links
    while $A[j] > x$ **do**
        $j := j - 1$;
    od // tot je een waarde $A[j] \leq x$ vindt
    $i := i + 1$; // loop met i naar rechts
    while $A[i] < x$ **do**
        $i := i + 1$;
    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

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

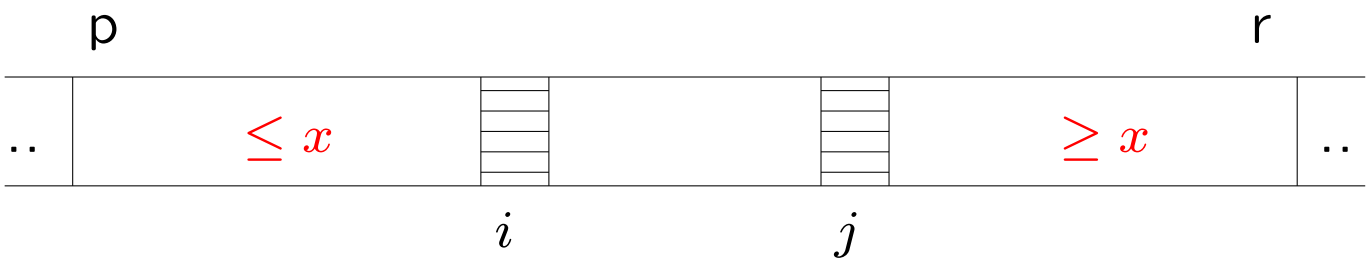
1. Na een volledige ronde:



2. Na de j-loop:

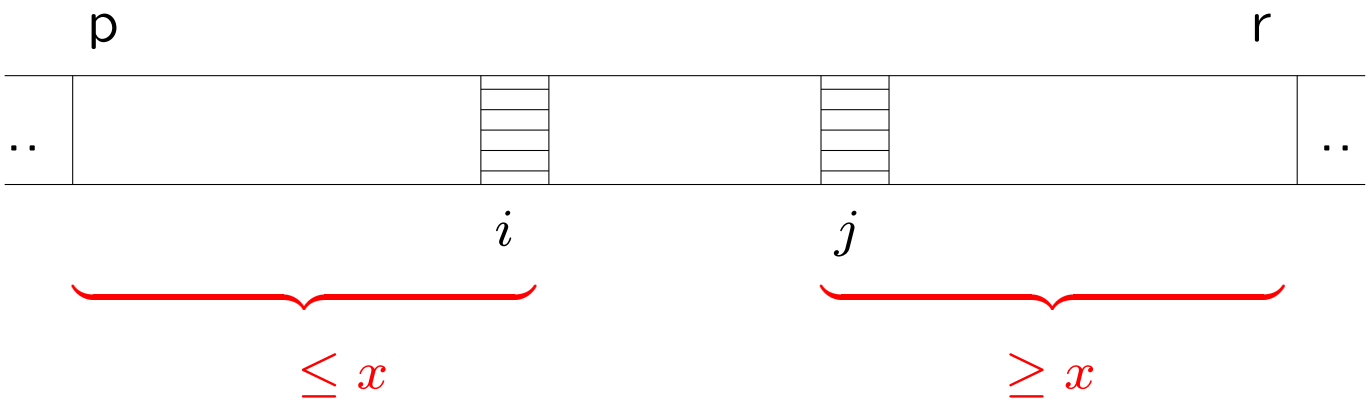


3. Na de i-loop, vóór de verwisseling:



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

4. Na de verwisseling:



1. Wat gebeurt er met gelijke array-elementen? Bijvoorbeeld: $5, 3, 5, 5, 6, 5, 7$ of $5, 6, 3, 4, 3, 8, 3$ of een array bestaande uit allemaal dezelfde elementen.

2. Wat gebeurt er met een rijtje met allemaal verschillende elementen? Bijvoorbeeld: $3, 2, 7, 4, 6, 8, 5, 1$ of een reeds gesorteerd rijtje of een omgekeerd gesorteerd rijtje.

3. Zie ook opgave 40 en 41 uit het opgavendictaat.

1. Bad case voor Quicksort:

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

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

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

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: $W(n) \in O(n^2)$.

Samen met de bad case hebben we dus:

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

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

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

Heapsort: zie dictaat, opgave 38

Vergelijking van verschillende sorteeralgoritmen

(average case; 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

Shellsort (Shell 1959)

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

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

↓ 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 in je eigen deelrijtje
(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 aantal rondes $t = \lfloor \lg n \rfloor$. Voor het gemak nemen we $n = 2^k$. Nu $t = \lg n = k$ en stapgroottes: $2^{k-1}, 2^{k-2}, \ldots, 4, 2, 1$.

   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. Zie verder college.

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

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

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

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

3. Het kan nog beter!

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

Bewijsschets: Je kunt inzien dat je bij stapgrootte h_i voor elk van de $n - h_i$ getallen maar met hoogstens $8h_i + 4$ getallen hoeft te vergelijken*: met $h_i = 3$, $h_{i+1} = 7$ en $h_{i+2} = 15$ weet je bijvoorbeeld al dat $A[100] \leq A[107] \leq A[114] \leq A[129]$. Dat is dus $O(nh_i)$ werk. We gebruiken dit voor $h_i < \sqrt{n}$. Voor $h_i > \sqrt{n}$ als voorheen: $O(n^2/h_i)$. Totaal geeft dat uiteindelijk $O(n\sqrt{n})$.

*Dit volgt uit de speciale eigenschappen van de h_i.

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

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

Betere incrementseries:

- 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 $O(n^{4/3})$

- $\cdots$ (optimale serie stapgroottes is [nog] onbepaald)

- Volgende college:
  dinsdag 26 maart, 11.00 – 12.45, zaal 174

- Eerstvolgende werkcollege:
  dinsdag 19 maart, 13.30 – 15.15, zaal 174
  Opgaven 14, 15, 16, 23, 24

- Huiswerkopgave 2:

  * deadline: dinsdag 2 april; LaTeX, print → college

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