

Hoofdstuk 15

Extern Sorteren

Motivatie

Extern Sorteren bestaat in het sorteren van hoeveelheden data die te groot zijn om nog in het centraal geheugen te kunnen.

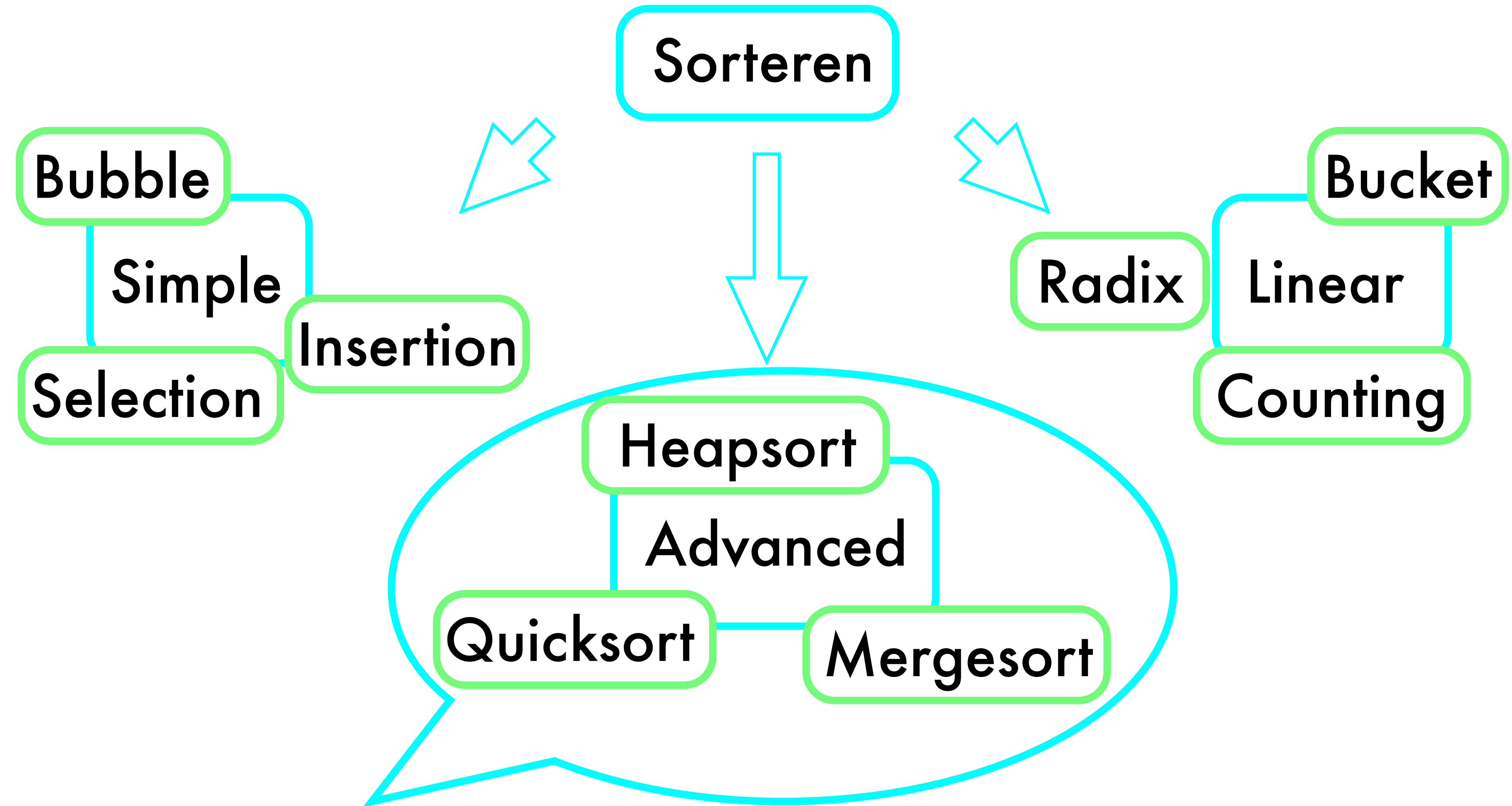
Gegevens afkomstig
van Hubble

Patiëntengegevens
van UZBrussel

Het aanbod van
amazon.com

De performantie wordt volledig bepaald
door het aantal block transfers!

Intern Sorteren: Overzicht



Enkel mergesort heeft aanvaardbaar caching-behaviour

Ter herinnering: sequentiële files

Lange rijen van Scheme waarden op disk

ADT output-file

```
new
  ( disk string → output-file )
open-write!
  ( disk string → output-file )
close-write!
  ( output-file → ∅ )
reread!
  ( output-file → input-file )
write!
  ( output-file any → ∅ )
delete!
  ( output-file → ∅ )
```

ADT input-file

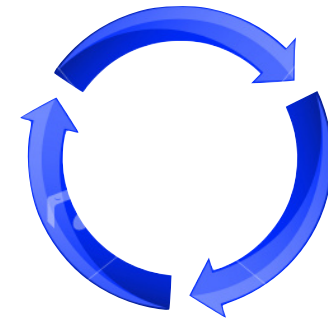
```
open-read!
  ( disk string → input-file )
rewrite!
  ( input-file → output-file )
close-read!
  ( input-file → ∅ )
has-more?
  ( input-file → boolean )
read
  ( input-file → any )
peek
  ( input-file → any )
delete!
  ( input-file → ∅ )
```

reread! en rewrite!
"spoelen de file terug"

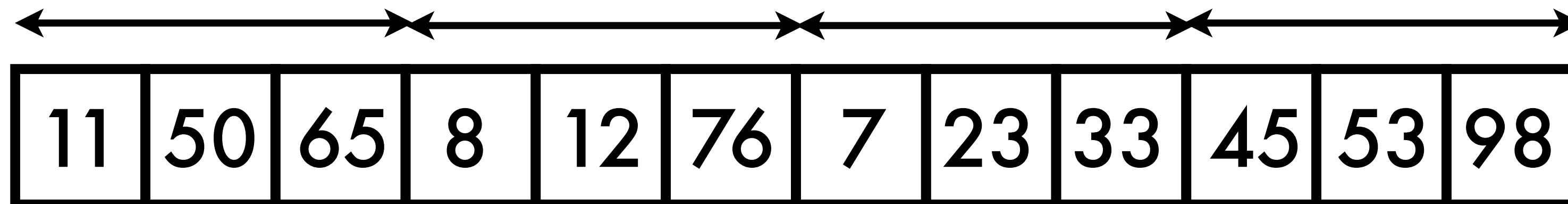
Extern Sorteren: Basisidee

We verdelen de te-sorteren file in een aantal hulpfiles.
De hulpfiles bestaan uit gesorteerde "zones".

We mergen de
gesorteerde hulpfiles



Een gesorteerde zone heet een run

A horizontal array of 12 numbers: 11, 50, 65, 8, 12, 76, 7, 23, 33, 45, 53, 98. Above the array, a long double-headed arrow spans the entire length. Four 'X' marks are placed above the array at positions corresponding to the end of the first, second, and third runs (after 65, 8, and 12).

11	50	65	8	12	76	7	23	33	45	53	98
----	----	----	---	----	----	---	----	----	----	----	----

Runs van lengte 3

Een abstractie boven output files

ADT output-file-with-counted-runs

new

(disk string number → output-file-with-counted-runs)

reread!

(output-file-with-counted-runs → input-file-with-counted-runs)

close-write!

(output-file-with-counted-runs → ∅)

file

(output-file-with-counted-runs → output-file)

run-length

(output-file-with-counted-runs → number)

name

(output-file-with-counted-runs → string)

delete!

(output-file-with-counted-runs → ∅)

new-run!

(output-file-with-counted-runs → ∅)

write!

(output-file-with-counted-runs any → ∅)

run-length

**new-run! start
een verse run**

**write! mislukt op
een volle run**

Implementatie Triviaal

Een abstractie boven input files

ADT input-file-with-counted-runs

rewrite!

(input-file-with-counted-runs → output-file-with-counted-runs)

close-read!

(input-file-with-counted-runs → \emptyset)

file

(input-file-with-counted-runs → input-file)

run-length

(input-file-with-counted-runs → number)

name

(input-file-with-counted-runs → string)

delete!

(input-file-with-counted-runs → \emptyset)

new-run!

(input-file-with-counted-runs → \emptyset)

has-more?

(input-file-with-counted-runs → boolean)

run-has-more?

(input-file-with-counted-runs → boolean)

read

(input-file-with-counted-runs → any)

peek

(input-file-with-counted-runs → any)

**new-run! start
een verse run**

**read faalt
indien de run
opgebruikt is**

Implementatie triviaal

We bestuderen 3 algoritmen

Multiway Balanced Merge Sort

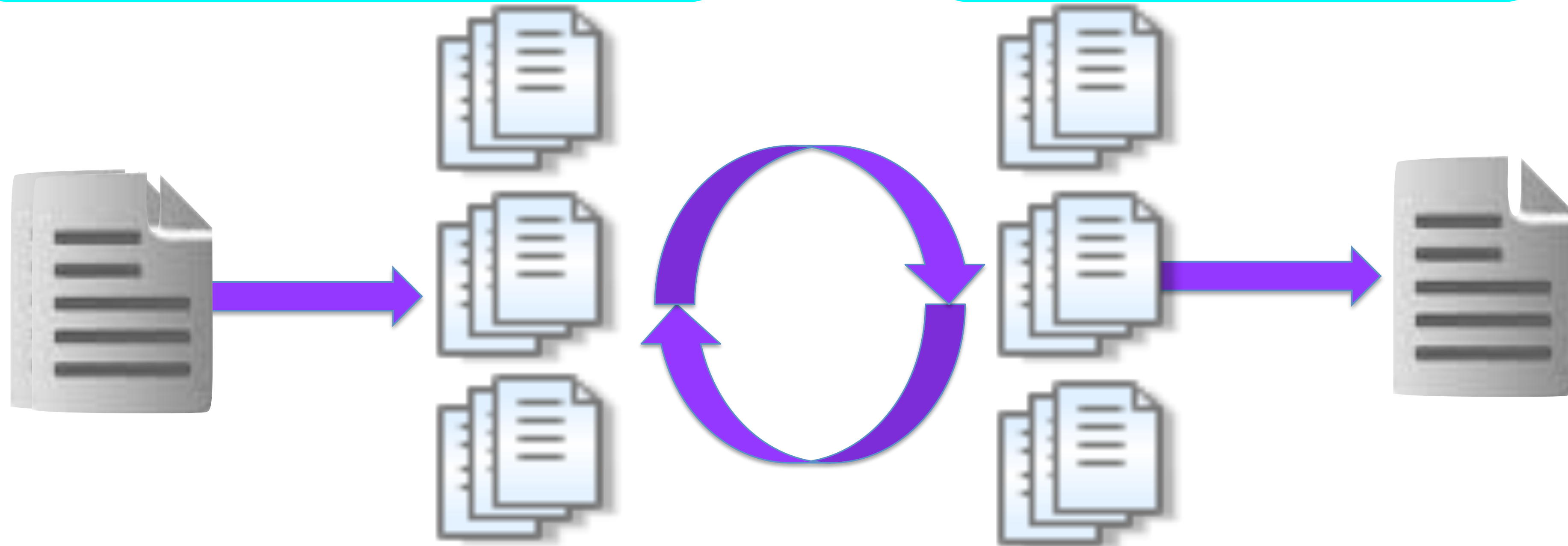
Polyphase Sort

p-Polyphase Sort

Multiway Balanced Merge Sort

De input file wordt verdeeld over p hulpfiles met runs van lengte r

We gebruiken $2 \times p$ hulpfiles in een p -way merge sort



In elke iteratie mergen we p files met runs van lengte r naar p files met runs van lengte $p \times r$

Voorbeeld met $p = 2$

3108 5923 7117 8753 9482 8664 2671 7723 2770 7036 2030 5895
9455 2660 7758 6013 5865 8633 3437 9422 9872 6533 2287 2254
1153 2001 5238 5474 8985 7668 8525 9508 7687 4215 3811 6935
7191 6657 3325 8259

Een file van
40 waarden

Stel een buffer van runlengte $r = 5$

3108 5923 7117 8753 9482 | 2030 2660 5895 7758 9455 | 1153 2254
2287 6533 9872 | 3811 4215 7687 8525 9508

en

2671 2770 7036 7723 8664 | 3437 5865 6013 8633 9422 | 2001 5238
5474 7668 8985 | 3325 6657 6935 7191 8259

runlengte 5

2671 2770 3108 5923 7036 7117 7723 8664 8753 9482 | 1153 2001 2254
2287 5238 5474 6533 7668 8985 9872

en

2030 2660 3437 5865 5895 6013 7758 8633 9422 9455 | 3325 3811 4215
6657 6935 7191 7687 8259 8525 9508

runlengte 10

Voorbeeld met $p = 2$

2671 2770 3108 5923 7036 7117 7723 8664 8753 9482 | 1153 2001
2254 2287 5238 5474 6533 7668 8985 9872

en

2030 2660 3437 5865 5895 6013 7758 8633 9422 9455 | 3325 3811
4215 6657 6935 7191 7687 8259 8525 9508

runlengte 10

2030 2660 2671 2770 3108 3437 5865 5895 5923 6013 7036 7117 7723
7758 8633 8664 8753 9422 9455 9482

en

1153 2001 2254 2287 3325 3811 4215 5238 5474 6533 6657 6935 7191
7668 7687 8259 8525 8985 9508 9872

runlengte 20

1153 2001 2030 2254 2287 2660 2671 2770 3108 3325 3437 3811
4215 5238 5474 5865 5895 5923 6013 6533 6657 6935 7036 7117
7191 7668 7687 7723 7758 8259 8525 8633 8664 8753 8985 9422
9455 9482 9505 9872

runlengte 40

Enkele Handige Testhulpstukken

```
(define (dump name lst)
  (define (w-iter out lst)
    (out:write! out (car lst))
    (if (not (null? (cdr lst)))
        (w-iter out (cdr lst))))
  (define out (out:new disk name))
  (w-iter out lst)
  out)
```

**Dump een
lijst op file**

```
(define (coll name)
  (define (coll-rec in)
    (cons (in:read in)
          (if (in:has-more? in)
              (coll-rec in)
              ())))
  (define in (in:open-read! disk name))
  (define res (coll-rec in))
  (in:close-read! in)
  (in:delete! in)
  res)
```

**Recupereer
de lijst**

```
(define (test-sort lst)
  (define disks (vector (disk:new "d0") (disk:new "d1") (disk:new "d2")
                        (disk:new "d3") (disk:new "d4") (disk:new "d5")))
  (vector-for-each (lambda (d) (fs:format! d)) disks)
  (let ((f (dump "f" lst)))
    (out:reread! f)
    (sort! f disks <)
    (display (coll "f")))))
```

p=3

Het Algoritme

Input file
op een disk

Een vector van disks
waarop elke hulpfile
zal worden gemaakt

```
(define (sort! file dsks <<?)  
  (define files (make-aux-bundle dsks))  
  (distribute! file files <<?)  
  (merge!      files <<?)  
  (collect!   files file)  
  (delete-aux-bundle! files))
```

Bundel van Hulpfiles

```
(define (make-bundle p in out)
  (cons p (cons in out)))

(define (order files)
  (car files))

(define (input files indx)
  (vector-ref (cadr files) indx))

(define (output files indx)
  (vector-ref (cddr files) indx))

(define (for-each-input files proc)
  (define nrfs (order files))
  (do ((indx 0 (+ indx 1)))
    ((= indx nrfs)
     (proc (input files indx) indx))))

(define (for-each-output files proc)
  (define nrfs (order files))
  (do ((indx 0 (+ indx 1)))
    ((= indx nrfs)
     (proc (output files indx) indx))))
```


Hulpfiles aanmaken en verwijderen

```
(define (make-aux-bundle disks)
  (define p (floor (/ (vector-length disks) 2)))
  (define in (make-vector p))
  (define out (make-vector p))
  (define name "aux-")
  (do ((i 0 (+ i 1)))
      ((= i p))
      (vector-set! out i (ofcr:new (vector-ref disks i)
                                   (string-append name (number->string i))
                                   rlen))
      (vector-set! in i (ofcr:new (vector-ref disks (+ p i))
                                   (string-append name (number->string (+ p i))
                                   rlen))
      (ofcr:reread! (vector-ref in i) rlen))
    (make-bundle p in out))
```

**Elke file mogelijks
op haar eigen disk**

```
(define (delete-aux-bundle! files)
  (for-each-input files
    (lambda (file indx)
      (ifcr:delete! file)))
  (for-each-output files
    (lambda (file indx)
      (ofcr:delete! file))))
```

Initiële Distributie

te sorteren data

bundel

```
(define (distribute! inpt files <<?)  
  (define p (order files))  
  (let loop  
    ((indx 0))  
    (let ((nubr (read-run! inpt)))  
      (when (not (= nubr 0))  
        (quicksort irun nubr <<?)  
        (write-run! (output files indx) nubr)  
        (ofcr:new-run! (output files indx))  
        (loop (next-file indx p))))))  
  (swap-files!? files))
```

Beperkt door grootte
centraal geheugen

We maken de buffer
zo groot mogelijk
(initiële run-lengte).

run buffer in centraal
geheugen

aantal gelezen
= aantal te
schrijven bytes

1. Vul de buffer
2. Quicksort de buffer
3. Ledig de buffer alternerend
naar output file $i \in [0..p-1]$

```
(define (next-file indx p)  
  (modulo (+ indx 1) p))
```

```
(define rlen 10)  
(define irun (make-vector rlen))  
  
(define (read-run! file)  
  (let loop  
    ((indx 0))  
    (cond ((or (= indx rlen) (not (in:has-more? file)))  
          indx)  
          (else  
            (vector-set! irun indx (in:read file))  
            (loop (+ indx 1))))))  
  
(define (write-run! ofcr imax)  
  (let loop  
    ((indx 0))  
    (ofcr:write! ofcr (vector-ref irun indx))  
    (if (< (+ indx 1) imax)  
        (loop (+ indx 1))))))
```


Swappen van $p \leftrightarrow p$ hulpfiles

```
(define (swap-files!? files)
  (define (switch-refs)
    (define tmp input)
    (set! input output)
    (set! output tmp))
  (define p (order files))
  (define old-run-length (ofcr:run-length (output files 0)))
  (define new-run-length (* p old-run-length))
  (for-each-output files (lambda (outp indx)
                           (ofcr:reread! outp old-run-length)))
  (for-each-input files (lambda (inpt indx)
                          (ifcr:rewrite! inpt new-run-length)))
  (switch-refs)
  (ifcr:has-more? (input files 1)))
```

geeft #f weer als alle te
lezen data in eerste file
zit, anders #t

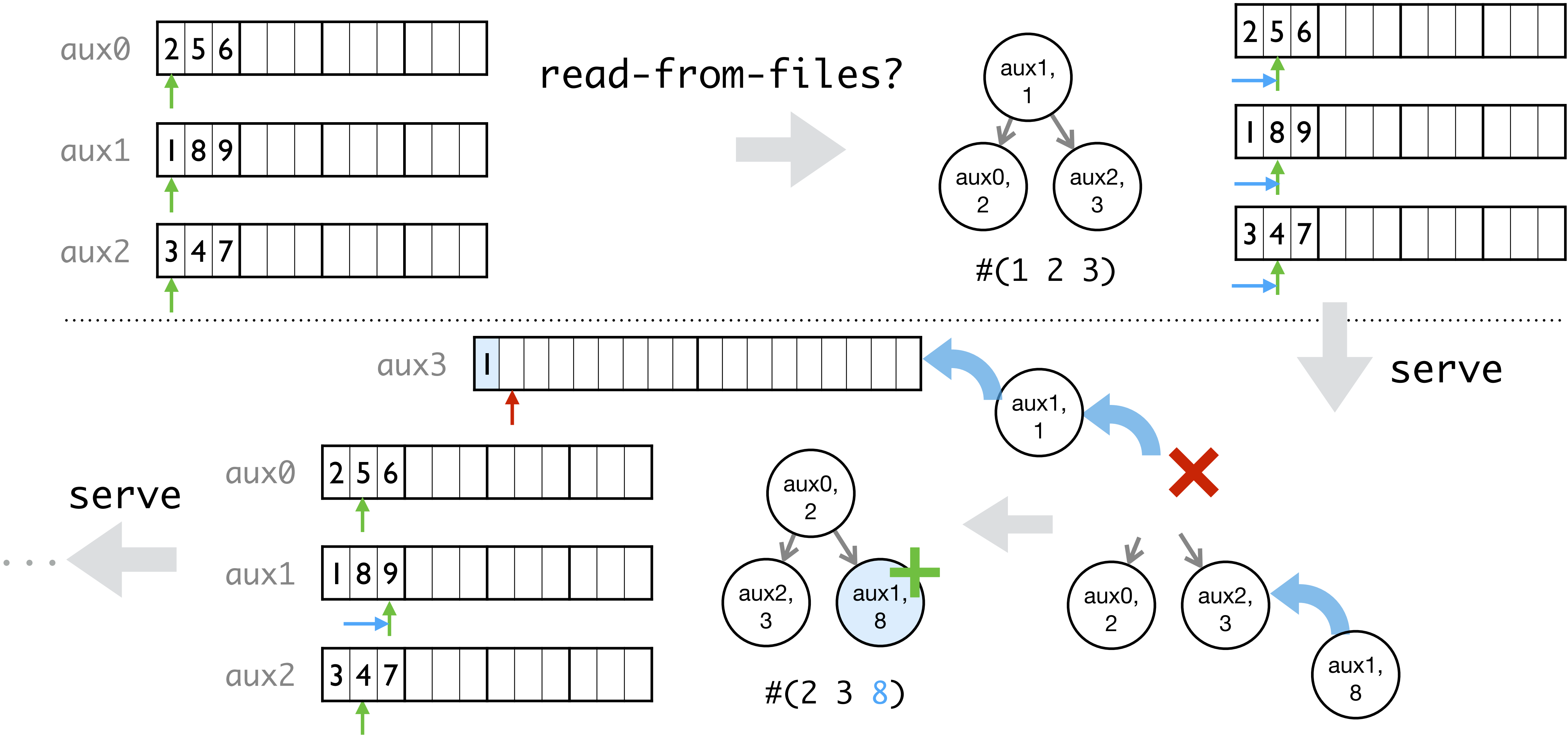
"is er nog werk?"

outputfiles worden
inputfiles met runlengte
old-run-length

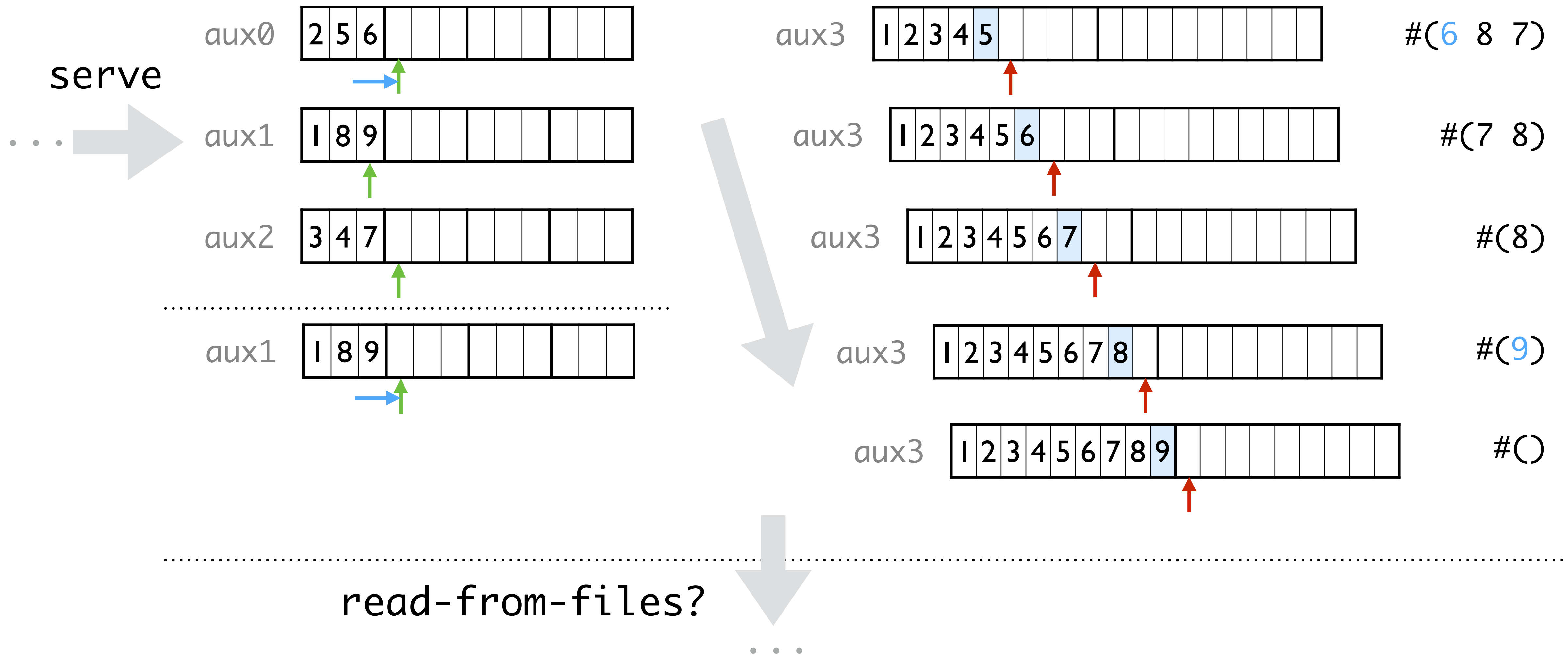
inputfiles worden
outputfiles met runlengte
 $p \times \text{old-run-length}$

Het hart van het algoritme

$p = 3, m = 3$



p = 3, m = 3



Het hart van het algoritme

```
(define (merge! files <<?)  
  (define heap (heap:new (order files)  
    (lambda (c1 c2)  
      (<<? (cdr c1) (cdr c2))))))  
  
  (let merge-files  
    ((out-idx 0))  
    (cond ((read-from-files? heap files)  
      (let merge-p-runs  
        ((rcrd (serve heap files)))  
        (ofcr:write! (output files out-idx) rcrd)  
        (if (not (heap:empty? heap))  
            (merge-p-runs (serve heap files))))  
      (ofcr:new-run! (output files out-idx))  
      (merge-files (next-file out-idx (order files))))  
      ((swap-files!? files)  
       (merge-files 0))))))
```

heap bevat pairs van
file-index en waarde

bepaal kleinste
en lees
volgende in run

lees van p input files
eerste waarde

De **outer loop** merget de p
files en blijft dit doen
zolang swap-files #t geeft

De **inner loop** merget
één run van iedere file
naar een nieuwe run

Heap met de kop van elke file

We gebruiken een heap om het kleinste van p elementen te bepalen

Lees het eerste element van elke nog niet-lege file

```
(define (read-from-files? heap files)
  (for-each-input
    files
    (lambda (file indx)
      (when (ifcr:has-more? file)
        (ifcr:new-run! file)
        (heap:insert! heap (cons indx (ifcr:read file))))))
  (not (heap:empty? heap)))
```

Geef kleinste element terug, voeg volgende uit zelfde run toe aan heap

kleinste element

```
(define (serve heap files)
  (define el (heap:delete! heap))
  (define indx (car el))
  (define rcrd (cdr el))
  (when (ifcr:run-has-more? (input files indx))
    (heap:insert! heap (cons indx (ifcr:read (input files indx))))
    rcrd)
```

lezen uit run waaruit kleinste element komt

De originele file herstellen

De gesorteerde
data bleef achter
op de 1ste inputfile

```
(define (collect! files inpt)
  (define last (input files 0))
  (in:rewrite! inpt)
  (let loop
    ((rcrd (ifcr:read last)))
    (out:write! inpt rcrd)
    (if (ifcr:run-has-more? last)
        (loop (ifcr:read last))))
  (out:close-write! inpt))
```

Performantie

We zijn geïnteresseerd in het aantal blocktransfers.
We tellen dus het aantal “passes” door de hulpfiles .

Zij n het aantal waarden op de input file en zij m de grootte van de buffer.
We hebben dus n/m runs na distributie.

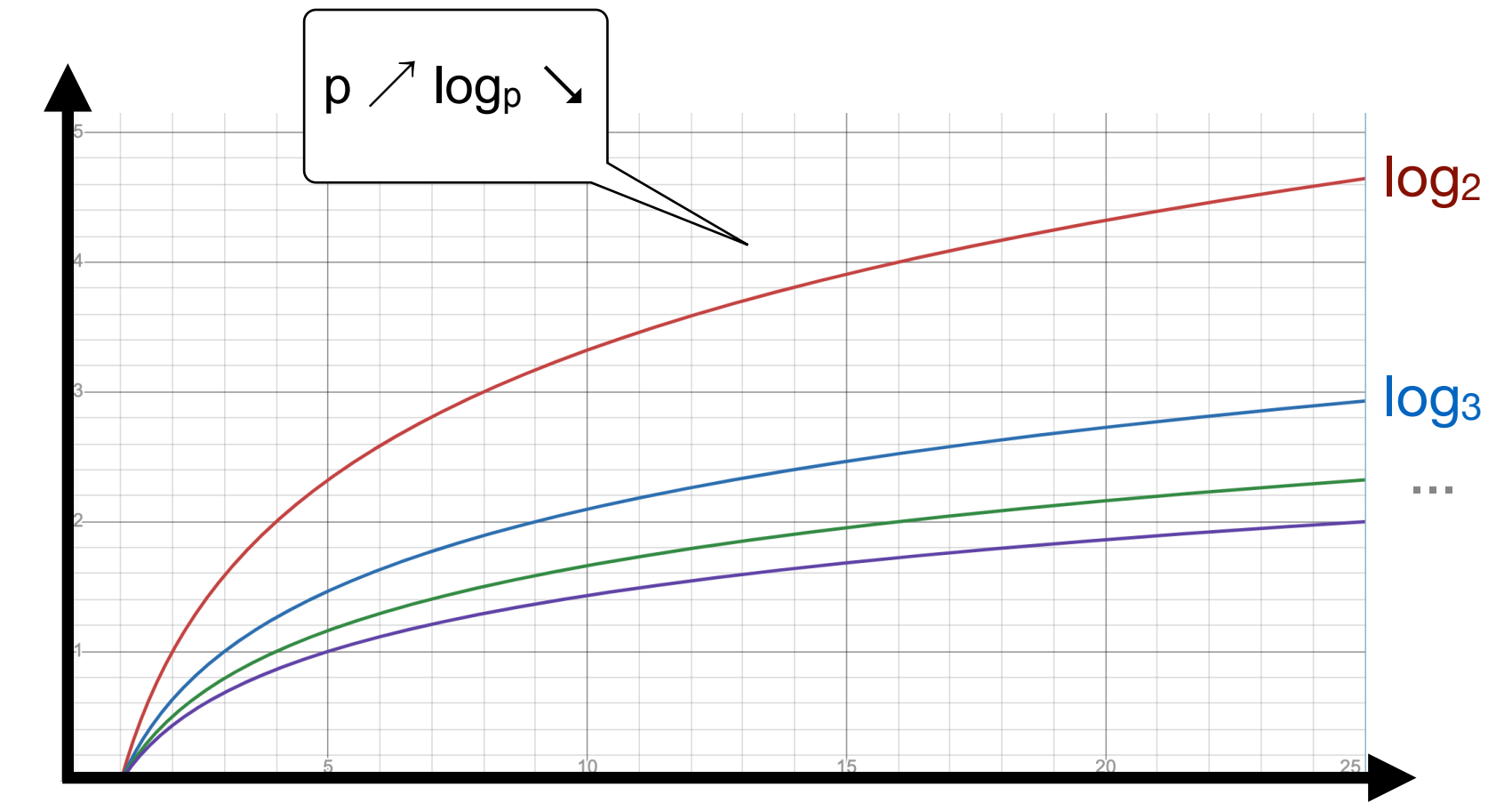
“pass”: alles een keer lezen en schrijven

De runlengte ver p -voudigt in iedere pass.
Het aantal runs op de hulpfiles deelt door p in iedere pass. Dus: hoe dikwijls kunnen we n/m door p delen tot we 1 run krijgen?

p -way balanced merge sort doet dus $O(\log_p(n/m))$ passes door de data.

Nadelen van Mergesort

n ligt vast (hoeveelheid data). m eveneens (grootte centraal geheugen). Enkel het grondtal p van de logaritme is variabel.



p kan niet willekeurig hoog:

- ofwel: het aantal disks ligt hardwarematig vast
- ofwel: het aantal files is beperkt door de buffers in sequentiële files: caching-gedrag!

<4 hulpfiles
niet bruikbaar

Oneven aantal
niet bruikbaar

We bestuderen 3 algoritmen

Multiway Balanced Merge Sort

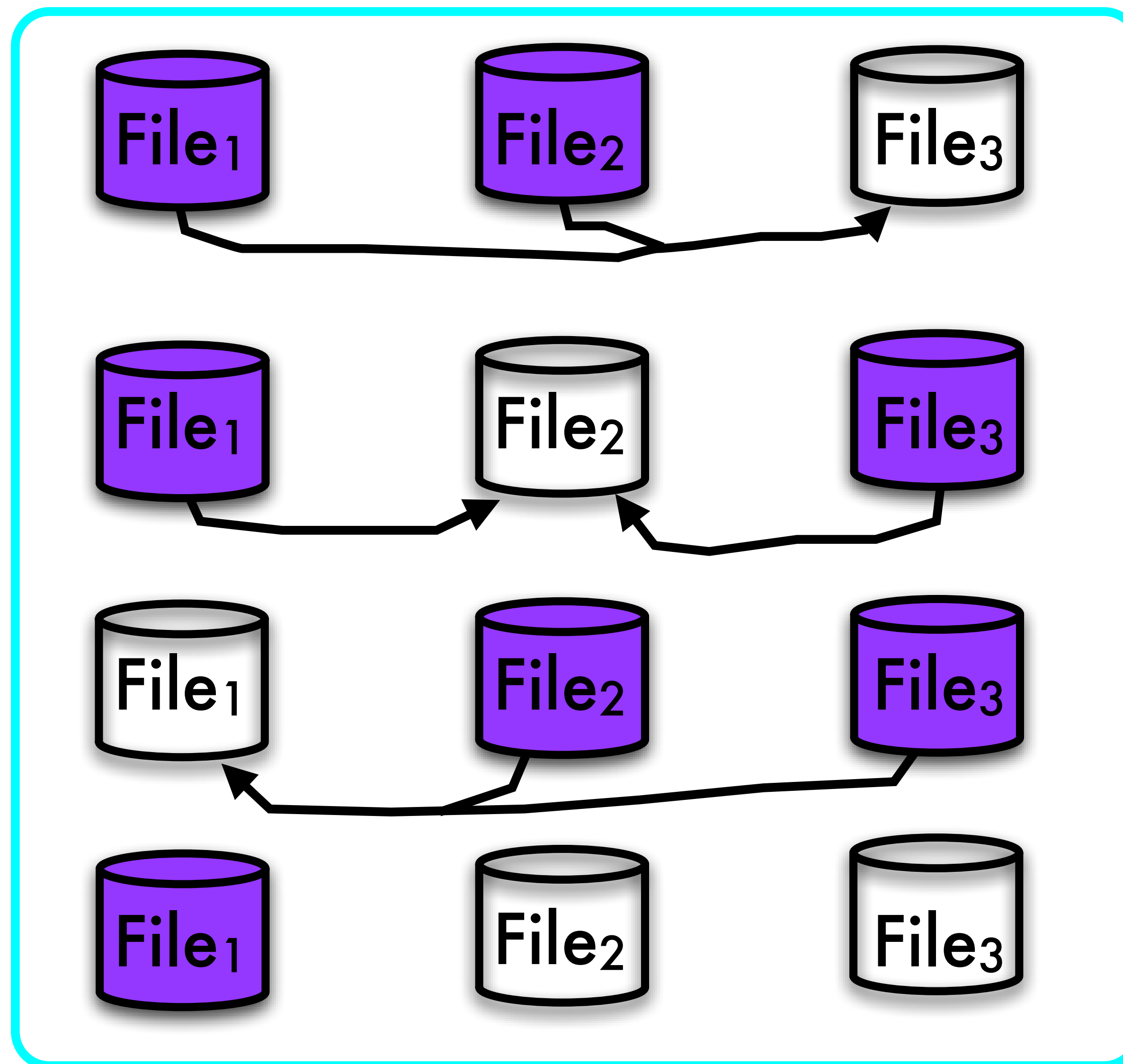
Polyphase Sort

p-Polyphase Sort

Idee van Polyphase Sort ($p+1=3$)

"phases"

wij nemen $p=2$

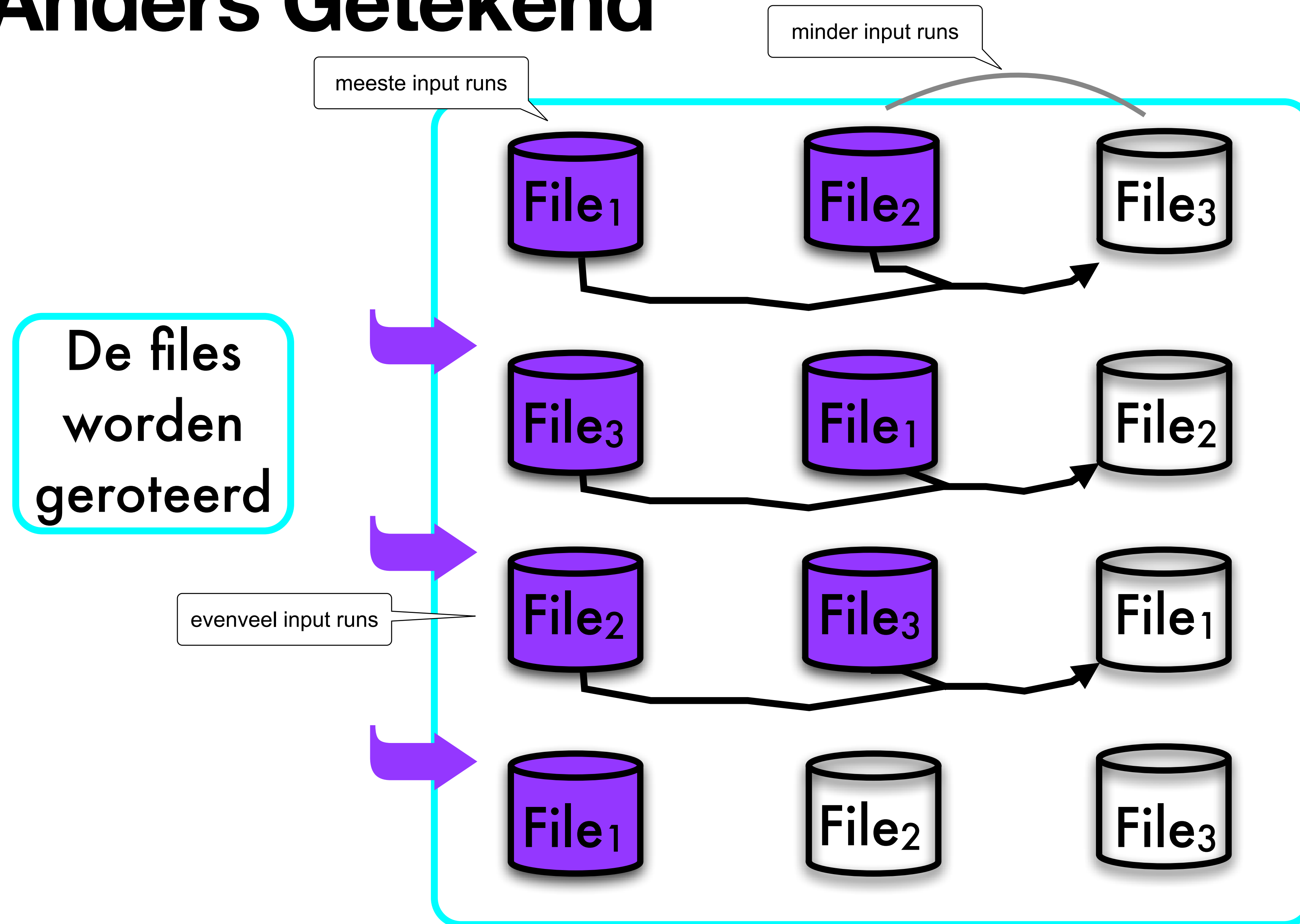


Telkens van p
hulpfiles naar 1
hulpfile mergen

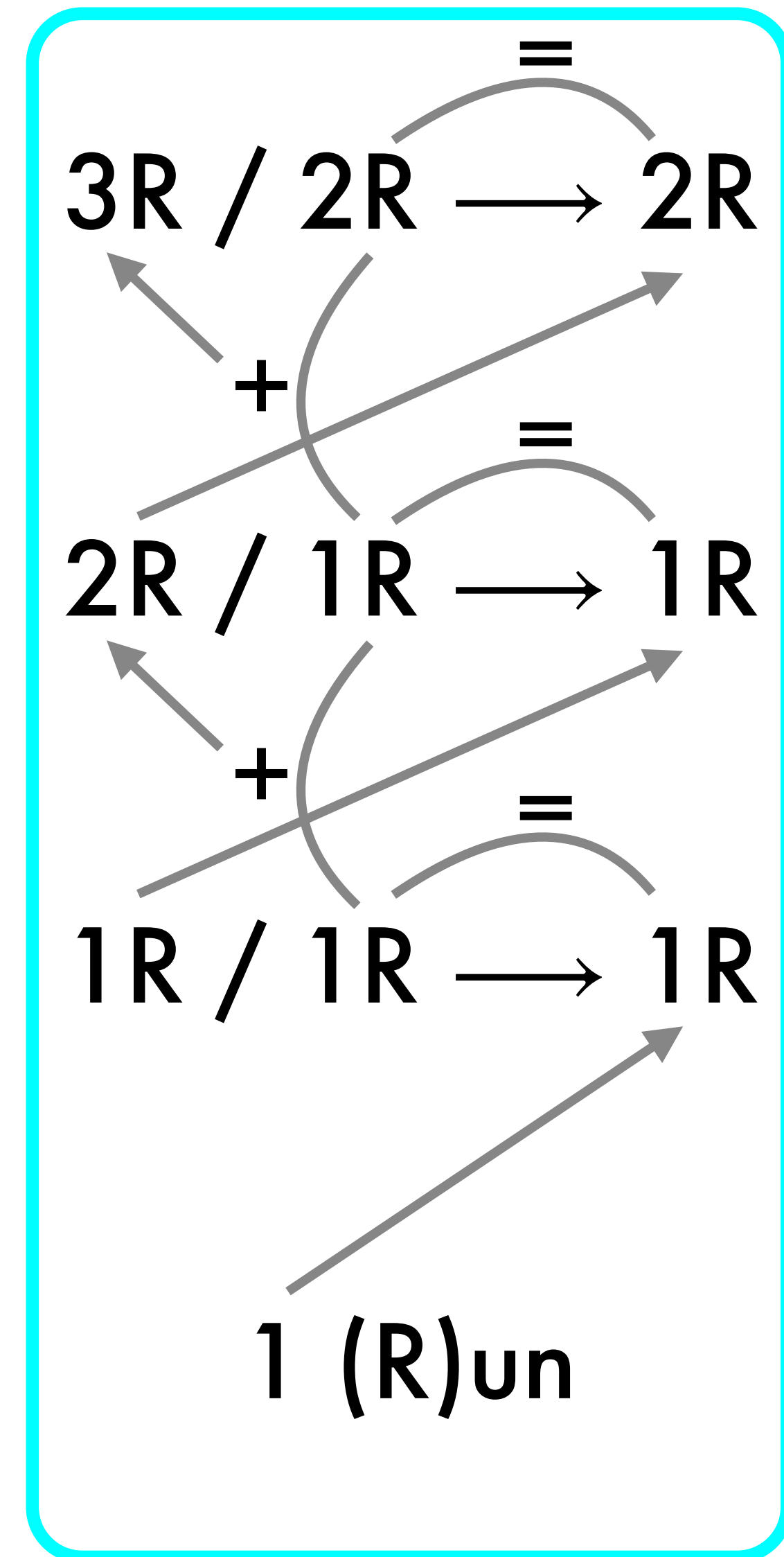
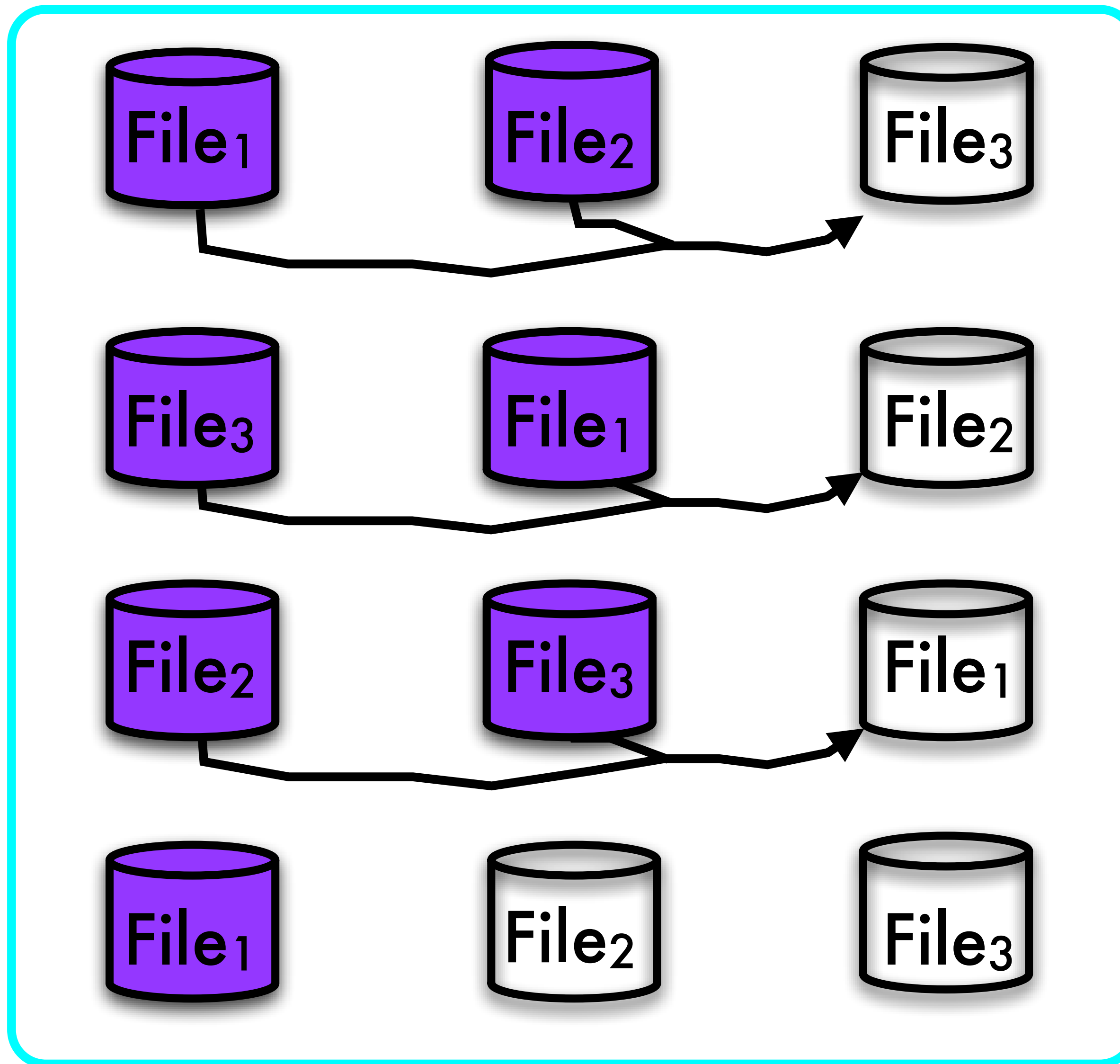
De ene file wordt
geledigd; op de
andere blijft wat
data achter.

"onbalanceerde" merge

Anders Getekend



Redenering “Achterstevoren”



Naar een generalisatie toe

$13R / 8R \longrightarrow 8R$

$8R / 5R \longrightarrow 5R$

$5R / 3R \longrightarrow 3R$

$3R / 2R \longrightarrow 2R$

$2R / 1R \longrightarrow 1R$

$1R / 1R \longrightarrow 1R$

$1 (R)_{un}$

Het aantal runs
op beide files zijn
opeenvolgende
Fibonaccigetallen

De hoeveelheid data
moet dus een
veelvoud van een
Fibonaccigetel zijn

Als file geen “Fib-veelvoud” lang is

(1) We “padden”
de laatste run met
voldoende $+\infty$

sentinel

(2) We voegen
voldoende “dummy
runs” met $+\infty$ toe
aan het einde

Zullen we straks
wegoptimaliseren

Initiële Runs met Padding

**Aanvullen
met $+\infty$**

```
(define (padded-read-run! file sent)  
  (define bsiz (read-run! file))  
  (if (and (not (= bsiz rlen))  
          (not (in:has-more? file)))  
      (do ((pad bsiz (+ pad 1)))  
          ((= pad rlen) bsiz)  
          (vector-set! irun pad sent))  
      bsiz))
```

Bundel van 3 Hulpfiles

```
(define (make-aux-bundle disks)
  (define files (make-vector 3))
  (vector-set! files 0 (ofcr:new (vector-ref disks 0)
                                "aux-0" rlen))
  (vector-set! files 1 (ofcr:new (vector-ref disks 1)
                                "aux-1" rlen))
  (vector-set! files 2 (ofcr:new (vector-ref disks 2)
                                "aux-2" (+ rlen rlen)))
  files)
```

```
(define (delete-aux-bundle! files)
  (fwrs:delete! (vector-ref files 0))
  (fwrs:delete! (vector-ref files 1))
  (fwrs:delete! (vector-ref files 2)))
```

```
(define (output files)
  (vector-ref files 2))

(define (input files i)
  (vector-ref files i))
```


Fibonaccidistributie

```

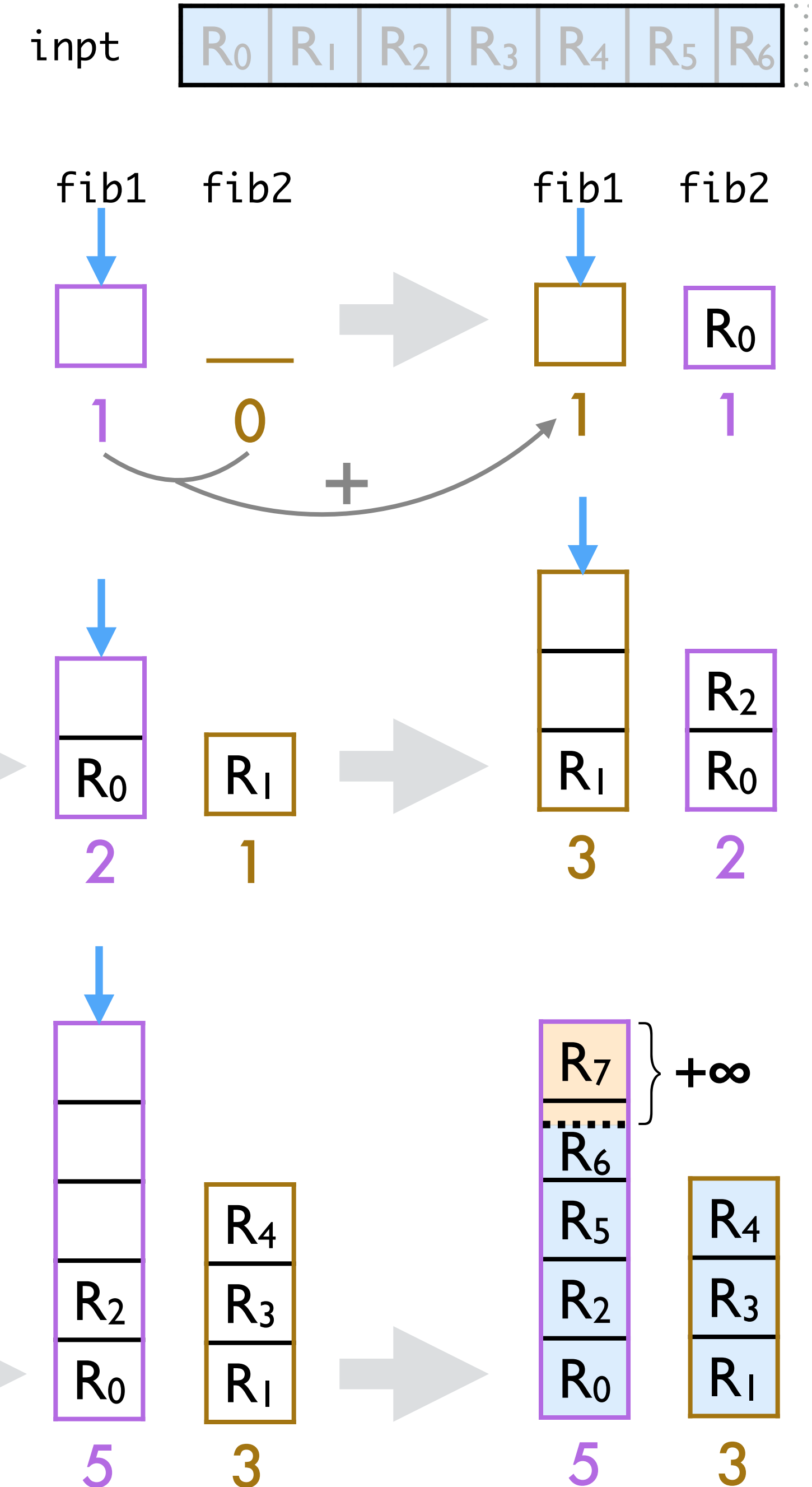
(define (distribute! inpt files <<? sent)
  (define (swap-input files)
    (define temp (vector-ref files 0))
    (vector-set! files 0 (vector-ref files 1))
    (vector-set! files 1 temp))
  (let loop
    ((fib1 1)
     (fib2 0)
     (out-ctr 0)
     (nmb (padded-read-run! inpt sent)))
    (cond ((< out-ctr fib1)
           (cond ((= nmb 0)
                  (write-run! (input files 0) rlen)
                  (ofcr:new-run! (input files 0))
                  (loop fib1 fib2 (+ out-ctr 1) nmb))
                (else
                 (quicksort irun nmb <<?)
                 (write-run! (input files 0) rlen)
                 (ofcr:new-run! (input files 0))
                 (loop fib1 fib2 (+ out-ctr 1) (padded-read-run!
                                                    inpt sent))))))
    ((in:has-more? inpt)
     (swap-input files)
     (loop (+ fib1 fib2) fib1 fib2 nmb)))
  (ofcr:reread! (input files 0) (ifcr:run-length (input files 0)))
  (ofcr:reread! (input files 1) (ifcr:run-length (input files 1))))
  
```

aantal (opgevulde) runs in linkse torentje

Blijf dummies schrijven

$+\infty$

13R / 8R
 8R / 5R
 5R / 3R
 3R / 2R
 2R / 1R
 1R / 1R



Het hart van het algoritme

Nagenoeg identiek aan multiway mergesort

nu heap met
gegarandeerd 2
elementen

```
(define (merge! files <<?)  
  (define heap (heap:new 2  
    (lambda (c1 c2)  
      (<<? (cdr c1) (cdr c2)))))  
  
  (let merge-files  
    ()  
    (cond ((read-from-files? heap files)  
      (let merge-2-runs  
        ((rcrd (serve heap files)))  
        (ofcr:write! (output files) rcrd)  
        (if (not (heap:empty? heap))  
            (merge-2-runs (serve heap files)))))  
      (ofcr:new-run! (output files))  
      (merge-files)  
      ((next-phase!? files)  
       (merge-files))))))
```

```
(define (merge! files <<?)  
  (define heap (heap:new (order files)  
    (lambda (c1 c2)  
      (<<? (cdr c1) (cdr c2)))))  
  
  (let merge-files  
    ((out-idx 0))  
    (cond ((read-from-files? heap files)  
      (let merge-p-runs  
        ((rcrd (serve heap files)))  
        (ofcr:write! (output files out-idx) rcrd)  
        (if (not (heap:empty? heap))  
            (merge-p-runs (serve heap files)))))  
      (ofcr:new-run! (output files out-idx))  
      (merge-files (next-file out-idx (order files))))  
    ((swap-files!? files)  
     (merge-files 0))))))
```

Heap met de kop van elke file

```
(define (read-from-files? heap files)
  (define ifcr1 (input files 0))
  (define ifcr2 (input files 1))
  (ifcr:new-run! (input files 1))
  (ifcr:new-run! (input files 0))
  (when (ifcr:has-more? ifcr2)
    (heap:insert! heap (cons 0 (ifcr:read ifcr1)))
    (heap:insert! heap (cons 1 (ifcr:read ifcr2))))
  (not (heap:empty? heap)))
```

Quasi identiek

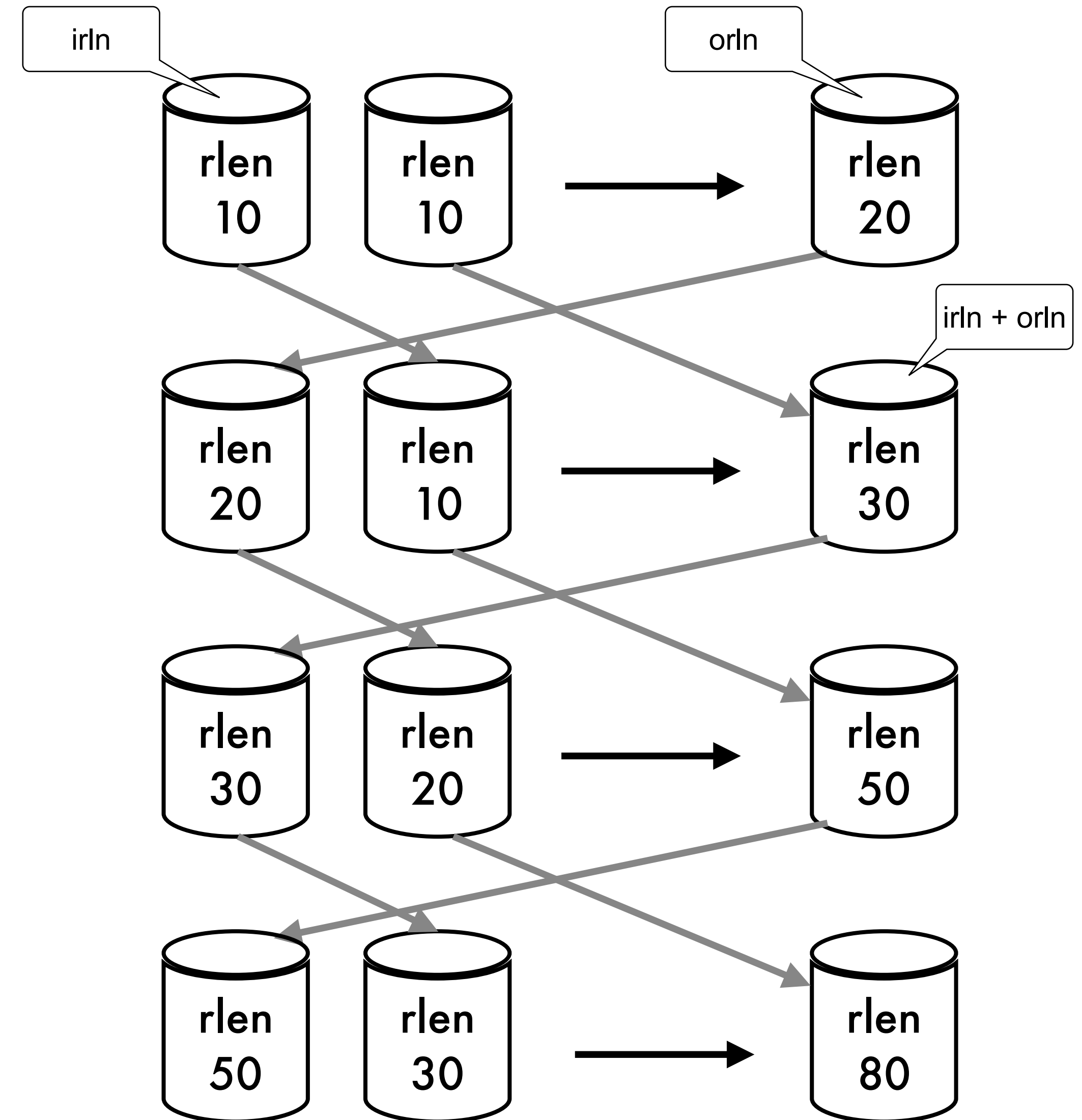
```
(define (serve heap files)
  (define el (heap:delete! heap))
  (define indx (car el))
  (define rcrd (cdr el))
  (if (ifcr:run-has-more? (input files indx))
      (heap:insert! heap (cons indx (ifcr:read (input files indx)))))
  rcrd)
```

Identiek

Roteren van de Files

```
(define (next-phase!? files)
  (define irln (ifcr:run-length (input files 0)))
  (define orln (ofcr:run-length (output files)))
  (define last (vector-ref files 2))
  (vector-set! files 2 (vector-ref files 1))
  (vector-set! files 1 (vector-ref files 0))
  (vector-set! files 0 last)
  (ifcr:rewrite! (output files) (+ irln orln))
  (ofcr:reread! (input files 0) orln)
  (ifcr:has-more? (input files 1)))
```

Runlengte is óók
een Ficonaccirij



De originele file herstellen

```
(define (collect! files inpt sent)
  (define last (input files 0))
  (in:rewrite! inpt)
  (let loop
    ((rcrd (ifcr:read last)))
    (out:write! inpt rcrd)
    (if (ifcr:run-has-more? last)
        (let ((rcrd (ifcr:read last)))
          (if (not (eq? rcrd sent))
              (loop rcrd))))))
  (out:close-write! inpt))
```

Identiek, behalve nu
 $+\infty$ niet meeschrijven!

Alles Samen: Polyphase Sort

```
(define (sort! file dsk <<? sent)
  (define files (make-aux-bundle dsk))
  (distribute! file files <<? sent)
  (merge!      files <<?)
  (collect!   files file sent)
  (delete-aux-bundle! files))
```

Identiek op
sentinel na

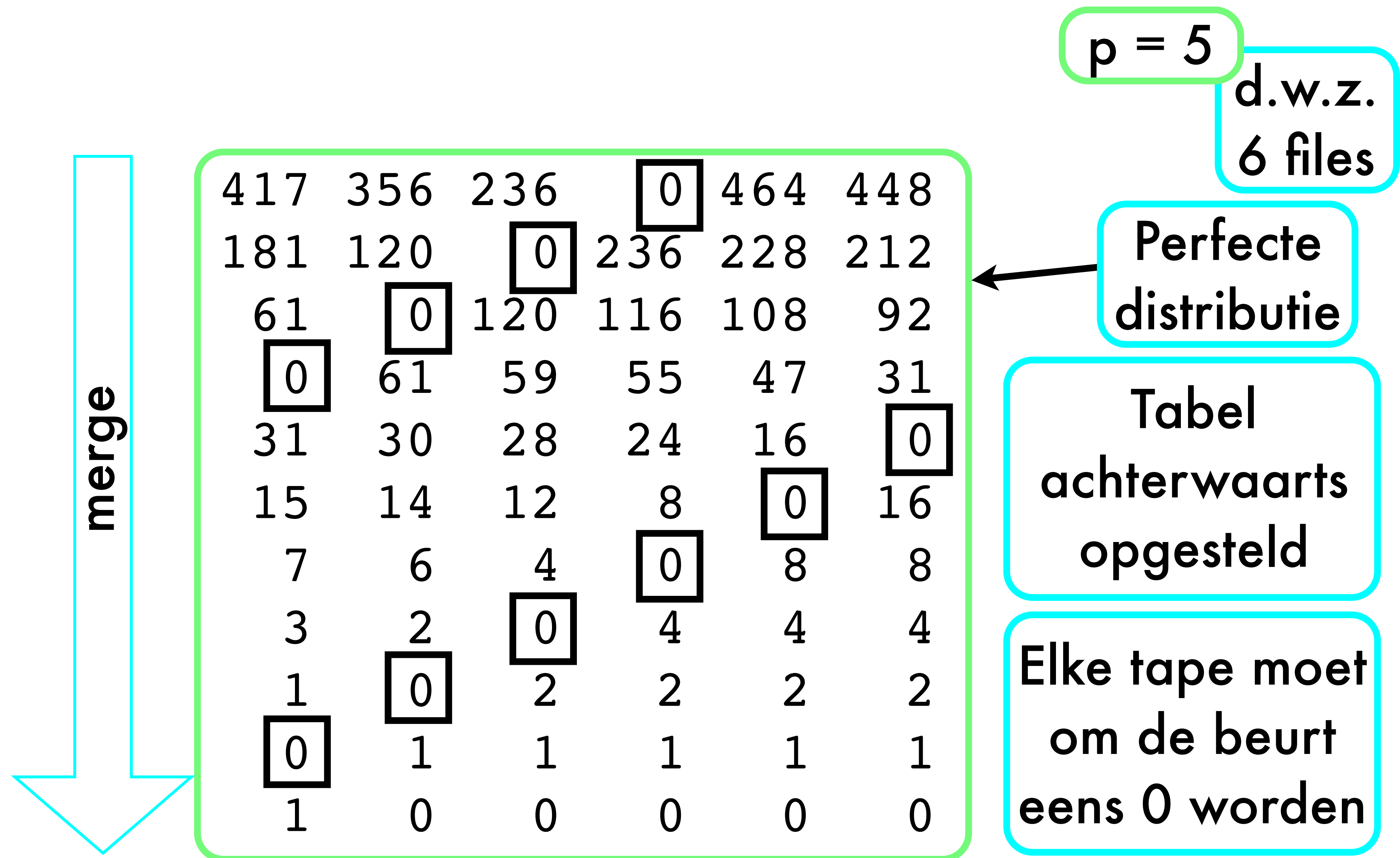
We bestuderen 3 algoritmen

Multiway Balanced Merge Sort

Polyphase Sort

p-Polyphase Sort

Veralgemening naar $p > 2$



Na Rotatie

$p = 5$

T.t.z.
6 files

merge

464	448	417	356	236
236	228	212	181	120
120	116	108	92	61
61	59	55	47	31
31	30	28	24	16
16	15	14	12	8
8	8	7	6	4
4	4	4	3	2
2	2	2	2	1
1	1	1	1	1
1	0	0	0	0

Som = max
runs met
deze rij

1922
977
497
389
253
80
33
17
9
5
1

Redenering “Achterstevoren”

464	448	417	356	236
236	228	212	181	120
120	116	108	92	61
61	59	55	47	31
31	30	28	24	16
16	15	14	12	8
8	8	7	6	4
4	4	4	3	2
2	2	2	2	1
1	1	1	1	1
1	0	0	0	0

Bepalen van de $(n+1)^{\text{de}}$ rij f_i^{n+1} op basis van de n^{de} rij f_i^n

Pak f_0^n , tel hem overal bij. De laatste keer bij 0

$f_0^n + f_1^n$ ← $f_0^n + f_2^n$ ← $f_0^n + f_3^n$ ← $f_0^n + f_4^n$ ← f_0^n
 f_0^n ← f_1^n ← f_2^n ← f_3^n ← f_4^n

f_i^{n+1} uitgeschreven

$$f_i^{n+1} = f_0^n + f_{i+1}^n \quad \forall \text{ kolom } i$$

$$f_4^{n+1} = f_0^n$$

$$f_3^{n+1} = f_0^n + f_4^n = f_0^n + f_0^{n-1}$$

$$f_2^{n+1} = f_0^n + f_3^n = f_0^n + f_0^{n-1} + f_0^{n-2}$$

$$f_1^{n+1} = f_0^n + f_2^n = f_0^n + f_0^{n-1} + f_0^{n-2} + f_0^{n-3}$$

$$f_0^{n+1} = f_0^n + f_1^n = f_0^n + f_0^{n-1} + f_0^{n-2} + f_0^{n-3} + f_0^{n-4}$$

Wiskundige Betekenis

De p-de orde Fibonaccirij is gedefinieerd door:

$$F_p(n) = F_p(n-1) + F_p(n-2) + \dots + F_p(n-p) \text{ als } n \geq k$$

$$F_p(p-1) = 1$$

$$F_p(i) = 0 \text{ als } i < p-1$$

0 0 0 1 1 2 4 8 16 31 61 120 236 464

“Ieder Fibonaccigetal is de som van de p vorige”

Gewone
Fibonaccirij
voor $p=2$

Wij nemen $f_0 = 1$
en $f_p = 0$ voor $p < 0$

Op naar een beter dummy-systeem

10 10
20 10
30 20
50 30
80 50
130 80

Bvb: een file van
131 waarden lang

Een lange staart dummies die
steeds weer gekopieerd wordt

(1) Dummies
gelijker verdelen

(2) Dummies tellen
i.p.v. wegschrijven

Fibonaccis/Dummies Berekenen

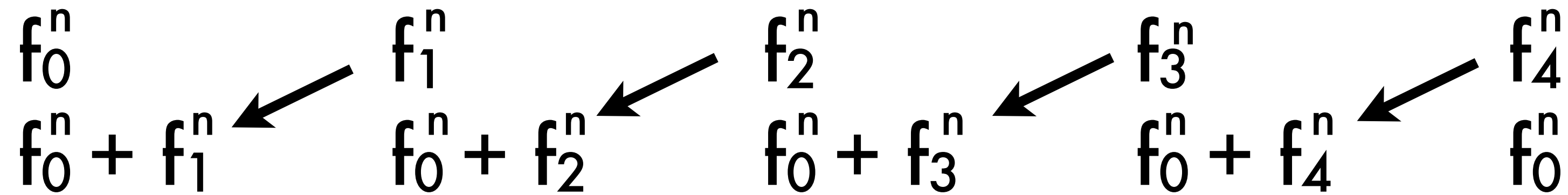
236 228 212 181 120

Stel deze rij
helemaal vol

464 448 417 356 236

Berekening van
de volgende rij

f_0^n f_1^n f_2^n f_3^n f_4^n
 $f_0^n + f_1^n$ $f_0^n + f_2^n$ $f_0^n + f_3^n$ $f_0^n + f_4^n$ f_0^n



228 220 205 175 116

Vrije plaats = Initiële
dummy hoeveelheden

Algemeen voor p files

$$f_i^{n+1} = f_0^n + f_{i+1}^n \quad \forall \text{ kolom } i$$

We beginnen met alle nieuwe voorziene plaatsen als dummy runs te beschouwen:

$$d_i^{n+1} = f_0^n + f_{i+1}^n - f_i^n \quad \forall \text{ kolom } i$$

Elke keer we een run wegschrijven verlagen we de "dummy count d_i " voor die file

Dummies Verspreiden

run count

dummy
count

#(1 1 1 1 1 0)	#(1 1 1 1 1 0)
#(1 1 1 1 1 0)	#(0 1 1 1 1 0)
#(1 1 1 1 1 0)	#(0 0 1 1 1 0)
#(1 1 1 1 1 0)	#(0 0 0 1 1 0)
#(1 1 1 1 1 0)	#(0 0 0 0 1 0)
#(2 2 2 2 1 0)	#(1 1 1 1 0 0)
#(2 2 2 2 1 0)	#(0 1 1 1 0 0)
#(2 2 2 2 1 0)	#(0 0 1 1 0 0)
#(2 2 2 2 1 0)	#(0 0 0 1 0 0)
#(4 4 4 3 2 0)	#(2 2 2 1 1 0)
#(4 4 4 3 2 0)	#(1 2 2 1 1 0)
#(4 4 4 3 2 0)	#(1 1 2 1 1 0)
#(4 4 4 3 2 0)	#(1 1 1 1 1 0)
#(4 4 4 3 2 0)	#(0 1 1 1 1 0)
#(4 4 4 3 2 0)	#(0 0 1 1 1 0)
#(4 4 4 3 2 0)	#(0 0 0 1 1 0)
#(4 4 4 3 2 0)	#(0 0 0 0 1 0)
#(8 8 7 6 4 0)	#(4 4 3 3 2 0)
#(8 8 7 6 4 0)	#(3 4 3 3 2 0)
#(8 8 7 6 4 0)	#(3 3 3 3 2 0)
#(8 8 7 6 4 0)	#(2 3 3 3 2 0)
#(8 8 7 6 4 0)	#(2 2 3 3 2 0)
#(8 8 7 6 4 0)	#(2 2 2 3 2 0)
#(8 8 7 6 4 0)	#(2 2 2 2 2 0)
#(8 8 7 6 4 0)	#(1 2 2 2 2 0)
...	...

Er zijn allerlei manieren.
De optimale manier is nog
niet gekend. We zullen
een manier gebruiken die
beter lijkt te zijn dan alle
andere manieren die even
simpel zijn [D. Knuth]

Horizontaal, file per
file, tot een file met
gelijk aantal dummies
wordt tegengekomen

De Runlengte is variabel!

run count

dummy
count

```
#(1 1 1 1 1 0) #(1 1 1 1 1 0)
#(1 1 1 1 1 0) #(0 1 1 1 1 0)
#(1 1 1 1 1 0) #(0 0 1 1 1 0)
#(1 1 1 1 1 0) #(0 0 0 1 1 0)
#(1 1 1 1 1 0) #(0 0 0 0 1 0)
#(2 2 2 2 1 0) #(1 1 1 1 0 0)
#(2 2 2 2 1 0) #(0 1 1 1 0 0)
#(2 2 2 2 1 0) #(0 0 1 1 0 0)
#(2 2 2 2 1 0) #(0 0 0 1 0 0)
#(4 4 4 3 2 0) #(2 2 2 1 1 0)
#(4 4 4 3 2 0) #(1 2 2 1 1 0)
#(4 4 4 3 2 0) #(1 1 2 1 1 0)
#(4 4 4 3 2 0) #(1 1 1 1 1 0)
#(4 4 4 3 2 0) #(0 1 1 1 1 0)
#(4 4 4 3 2 0) #(0 0 1 1 1 0)
#(4 4 4 3 2 0) #(0 0 0 1 1 0)
#(4 4 4 3 2 0) #(0 0 0 0 1 0)
#(8 8 7 6 4 0) #(4 4 3 3 2 0)
#(8 8 7 6 4 0) #(3 4 3 3 2 0)
#(8 8 7 6 4 0) #(3 3 3 3 2 0)
#(8 8 7 6 4 0) #(2 3 3 3 2 0)
#(8 8 7 6 4 0) #(2 2 3 3 2 0)
#(8 8 7 6 4 0) #(2 2 2 3 2 0)
#(8 8 7 6 4 0) #(2 2 2 2 2 0)
#(8 8 7 6 4 0) #(1 2 2 2 2 0)
... ..
```

De dummies zijn “gelijkmatig” verspreid met de Knuth-distributie. Maar niet perfect gelijk! Dus zal een merge niet steeds van elke file eten. De runlengte wordt dus variabel!

Een abstractie boven output files

ADT output-file-with-varying-runs

new

(disk string sent → output-file-with-varying-runs)

reread!

(output-file-with-varying-runs → input-file-with-varying-runs)

close-write!

(output-file-with-varying-runs → ∅)

file

(output-file-with-varying-runs → output-file)

name

(output-file-with-varying-runs → string)

delete!

(output-file-with-varying-runs → ∅)

new-run!

(output-file-with-varying-runs → ∅)

write!

(output-file-with-varying-runs any → ∅)

end-of-run-marker

**new-run! markt
een verse run**

**write! mislukt
nooit**

Implementatie: Triviaal

Een abstractie boven input files

```
ADT input-file-with-varying-runs

rewrite!
  ( input-file-with-varying-runs → output-file-with-varying-runs )
close-read!
  ( input-file-with-varying-runs → ∅ )
file
  ( input-file-with-varying-runs → input-file )
name
  ( input-file-with-varying-runs → string )
delete!
  ( input-file-with-varying-runs → ∅ )
new-run!
  ( input-file-with-varying-runs → ∅ )
has-more?
  ( input-file-with-varying-runs → boolean )
run-has-more?
  ( input-file-with-varying-runs → boolean )
read
  ( input-file-with-varying-runs → any )
peek
  ( input-file-with-varying-runs → any )
```

**new-run! markt
een verse run**

**read faalt
indien de run
opgebruikt is**

Implementatie: Triviaal

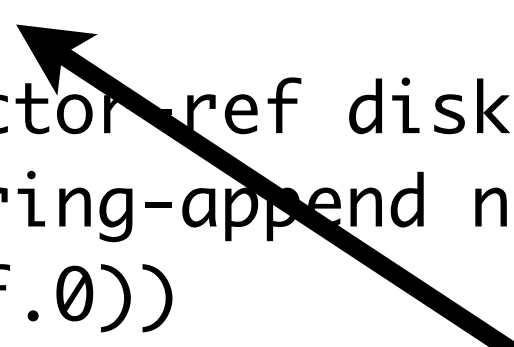
De Hulpfiles

```
(define (make-aux-bundle disks)
  (define p (- (vector-length disks) 1))
  (define rslt (make-bundle (make-vector (+ p 1))
                              (make-vector (+ p 1))
                              (make-vector (+ p 1))
                              (make-vector (+ p 1))))

  (define name "aux-")
  (do ((indx 0 (+ indx 1)))
      ((= indx p))
    (file! rslt indx (ofvr:new (vector-ref disks indx)
                              (string-append name (number->string indx))
                              +inf.0))

    (run!   rslt indx 1)
    (dummy! rslt indx 1)
    (selected! rslt indx #f))
  (file! rslt p (ofvr:new (vector-ref disks p)
                          (string-append name (number->string p))
                          +inf.0))

  (run!   rslt p 0)
  (dummy! rslt p 0)
  rslt)
```



Speelt de file mee in de huidige run-merge? (niet indien dummy geconsumeerd!)

Bundeling van alle fileinfo

```
(define-record-type bundle  
  (make-bundle s d r f)  
  bundle?  
  (s selecteds)  
  (d dummies)  
  (r runs)  
  (f files))
```

**Accessoren houden
de code leesbaar**

```
(define (output fils)  
  (vector-ref (files fils) (order fils)))  
(define (input fils i)  
  (vector-ref (files fils) i))  
  
(define (dummy fils indx)  
  (vector-ref (dummies fils) indx))  
(define (dummy! fils indx cnt)  
  (vector-set! (dummies fils) indx cnt))  
(define (run fils indx)  
  (vector-ref (runs fils) indx))  
(define (run! fils indx cnt)  
  (vector-set! (runs fils) indx cnt))  
(define (file fils indx)  
  (vector-ref (files fils) indx))  
(define (file! fils indx file)  
  (vector-set! (files fils) indx file))  
(define (selected fils indx)  
  (vector-ref (selecteds fils) indx))  
(define (selected! fils indx sltd)  
  (vector-set! (selecteds fils) indx sltd))  
  
(define (order fils)  
  (- (vector-length (files fils)) 1))
```

Fibonaccidistributie van orde p

```
(define (distribute! inpt fils <<?)  
  (define p (order fils))  
  (let loop  
    ((indx 0)  
     (nmbr (read-run! inpt)))  
    (when (> nmbr 0)  
      (quicksort irun nmbr <<?)  
      (write-run! (input fils indx) nmbr)  
      (ofvr:new-run! (file fils indx))  
      (dummy! fils indx (- (dummy fils indx) 1))  
      (cond ((< (dummy fils indx) (dummy fils (+ indx 1)))  
             (loop (+ indx 1) (read-run! inpt)))  
            ((= (dummy fils indx) 0)  
             (let ((rmax (run fils 0)))  
               (do ((i 0 (+ i 1)))  
                 ((= i p)  
                  (dummy! fils i (+ rmax (run fils (+ i 1)) (- (run fils i))))  
                  (run!   fils i (+ rmax (run fils (+ i 1))))))  
               (loop 0 (read-run! inpt))))  
            (else  
             (loop 0 (read-run! inpt))))))  
    (do ((indx 0 (+ indx 1)))  
      ((= indx p)  
       (ofvr:reread! (input fils indx))  
       (eat-dummies fils)))
```

Bereken de
volgende rij

Knuth distributieregel

Doe alsof dummies vóóraan staan

Bepaal kleinste # dummies,
trek die overal af en tel ze
bij de resultaatfile bij

```
(define (eat-dummies fils)
  (define p (order fils))
  (define skip
    (let ((dmin +inf.0))
      (do ((indx 0 (+ indx 1)))
          ((= indx p)
           dmin)
          (if (< (dummy fils indx) dmin)
              (set! dmin (dummy fils indx))))))
  (dummy! fils p (+ (dummy fils p) skip))
  (do ((indx 0 (+ indx 1)))
      ((= indx p))
      (dummy! fils indx (- (dummy fils indx) skip))
      (run! fils indx (- (run fils indx) skip)))
  (run! fils p (+ (run fils p) skip)))
```

#(4 4 3 3 2 0)

#(2 2 1 1 0 2)

Rotatie van de files

```
(define (next-phase!? fils)
  (define p (order fils))
  (define last-fil (file fils p))
  (define last-run (run fils p))
  (define last-dum (dummy fils p))
  (do ((indx p (- indx 1)))
      ((= indx 0))
      (file! fils indx (file fils (- indx 1)))
      (dummy! fils indx (dummy fils (- indx 1)))
      (run! fils indx (run fils (- indx 1)))
      (selected! fils indx (selected fils (- indx 1)))))
  (file!      fils 0 last-fil)
  (dummy!     fils 0 last-dum)
  (run!       fils 0 last-run)
  (selected!  fils 0 #f)
  (ofvr:reread! (input fils 0))
  (ifvr:rewrite! (output fils))
  (eat-dummies fils)
  (> (run fils (- p 1)) 0))
```

**We gaan door
zolang er runs
voorhanden zijn**

Zet alle files klaar

```
(define (start-new-runs! fils)
  (define p (order fils))
  (do ((indx 0 (+ indx 1)))
      ((= indx p)
       (run! fils p (+ (run fils p) 1)))
      (run! fils indx (- (run fils indx) 1)))
  (do ((indx 0 (+ indx 1)))
      ((= indx p))
      (cond ((and (ifvr:has-more? (file fils indx))
                  (= (dummy fils indx) 0))
             (if (selected fils indx)
                 (ifvr:new-run! (file fils indx))
                 (selected! fils indx #t))
             (else
              (selected! fils indx #f)
              (if (> (dummy fils indx) 0)
                  (dummy! fils indx (- (dummy fils indx) 1))))))))))
```

Bepaal welke
files data
leveren en
welke dummies

(data als # dummies = 0)

Heap met de kop van elke file

Vul heap enkel
van files die
data leveren

```
(define (read-from-files? heap fils)
  (define p (order fils))
  (cond ((> (run fils (- p 1)) 0)
        (start-new-runs! fils)
        (do ((indx 0 (+ indx 1)))
            ((= indx p) #t)
            (if (selected fils indx)
                (heap:insert! heap (cons indx (ifvr:read (file fils indx))))))
        (else #f)))
```

Hervul van
dezelfde file

```
(define (serve heap files)
  (define el (heap:delete! heap))
  (define indx (car el))
  (define rcrd (cdr el))
  (if (ifvr:run-has-more? (input files indx)); try where we read
      (heap:insert! heap (cons indx (ifvr:read (input files indx))))
      rcrd)
```

De rest van het verhaal

```
(define (merge! fils <<?)  
  (define heap (heap:new (order fils)  
                          (lambda (c1 c2)  
                            (<<? (cdr c1) (cdr c2))))))  
  
  (let merge-files  
    ()  
    (cond ((read-from-files? heap fils)  
           (let merge-p-runs  
             ((rcrd (serve heap fils)))  
             (ofvr:write! (output fils) rcrd)  
             (if (not (heap:empty? heap))  
                 (merge-p-runs (serve heap fils))))  
           (ofvr:new-run! (output fils))  
           (merge-files))  
          ((next-phase!? fils)  
           (merge-files))))))
```

Quasi
identiek

```
(define (sort! file dsks <<?)  
  (define fils (make-aux-bundle dsks))  
  (distribute! file fils <<?)  
  (merge!   fils <<?)  
  (collect! fils file)  
  (delete-aux-bundle! fils))
```

```
(define (collect! files inpt)  
  (define last (input files 0))  
  (in:rewrite! inpt)  
  (let loop  
    ((rcrd (ifvr:read last)))  
    (out:write! inpt rcrd)  
    (if (ifvr:run-has-more? last)  
        (loop (ifvr:read last))))  
  (out:close-write! inpt))
```

Aantal passes door data [Knuth]

Vanaf 7 nauwelijks
nog winst

$$S = \# \text{ initiële runs} = n/m$$

logaritmische
schaal!

Vrij langdradige
wiskunde nodig

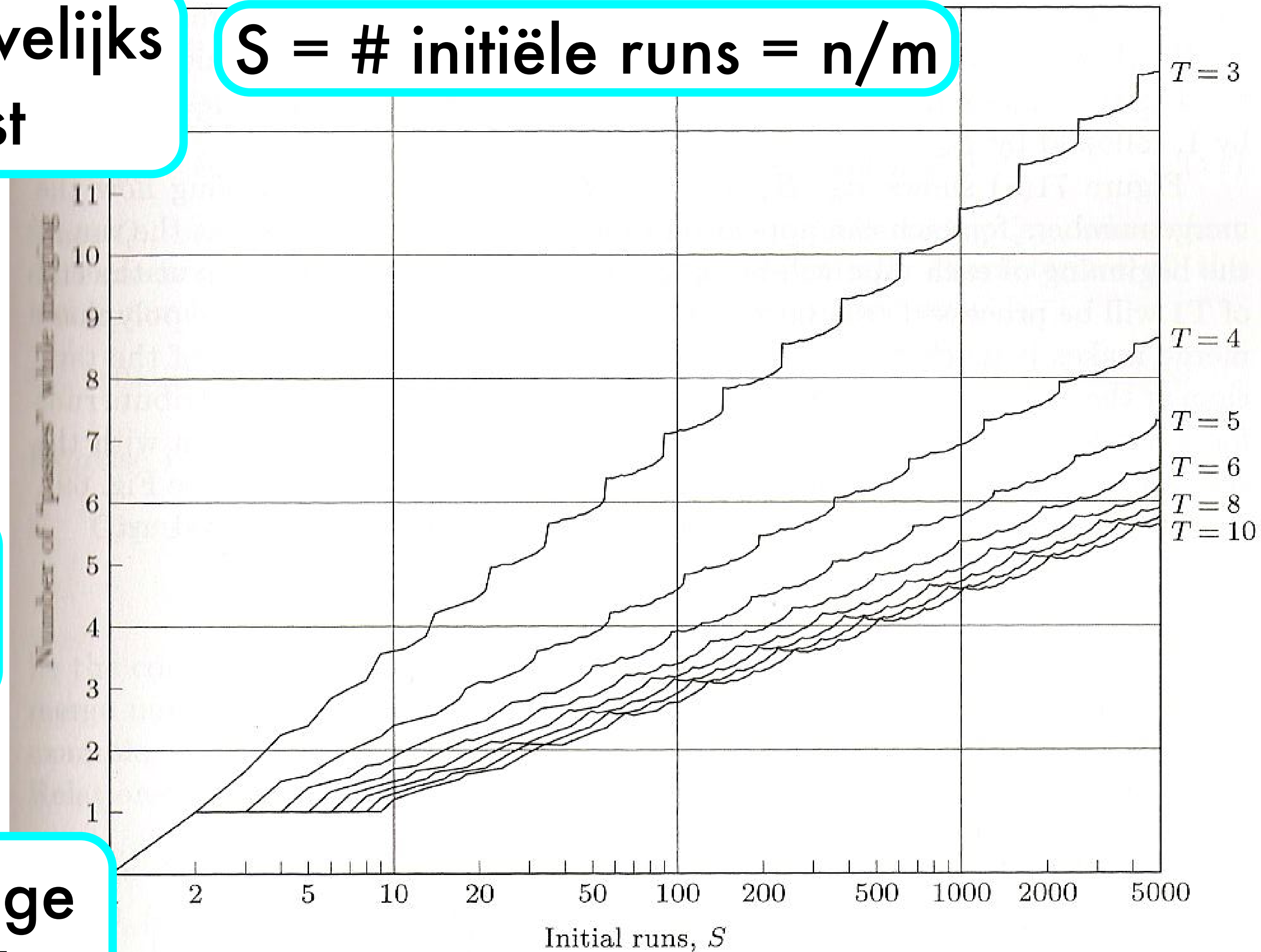


Fig. 70. Efficiency of polyphase merge using Algorithm D.

Theoretische Ondergrens

Stelling: Laat m de grootte van het centraal geheugen zijn en b de grootte van een blok. Het sorteren van n waarden vereist

$$\Omega \left(\frac{n}{b} \log_{\frac{m}{b}} \left(\frac{n}{b} \right) \right) \text{ blok transfers.}$$

n/b is de prijs om de data 1 keer te vast te pakken

$\log(n/b)$ keer de data vastpakken (c.f. intern sorteren)

m/b is het aantal blokken dat in centraal geheugen past

Hoofdstuk 15

15.1 Extern Sorteren: Inleiding

15.2 Multiway balanced merge sort

15.3 Polyphase Sort

15.4 p-Polyphase Sort

15.4 Theoretische Ondergrens Extern Sorteren

