

Hoofdstuk 5

Sorteren

Sorteren

Sorteren = een aantal data waarden ordenen volgens een zekere “orde”

<<?

Er bestaan *interne* sorteeralgoritmen en externe sorteeralgoritmen

Sorteren is zeker $\Omega(n)$

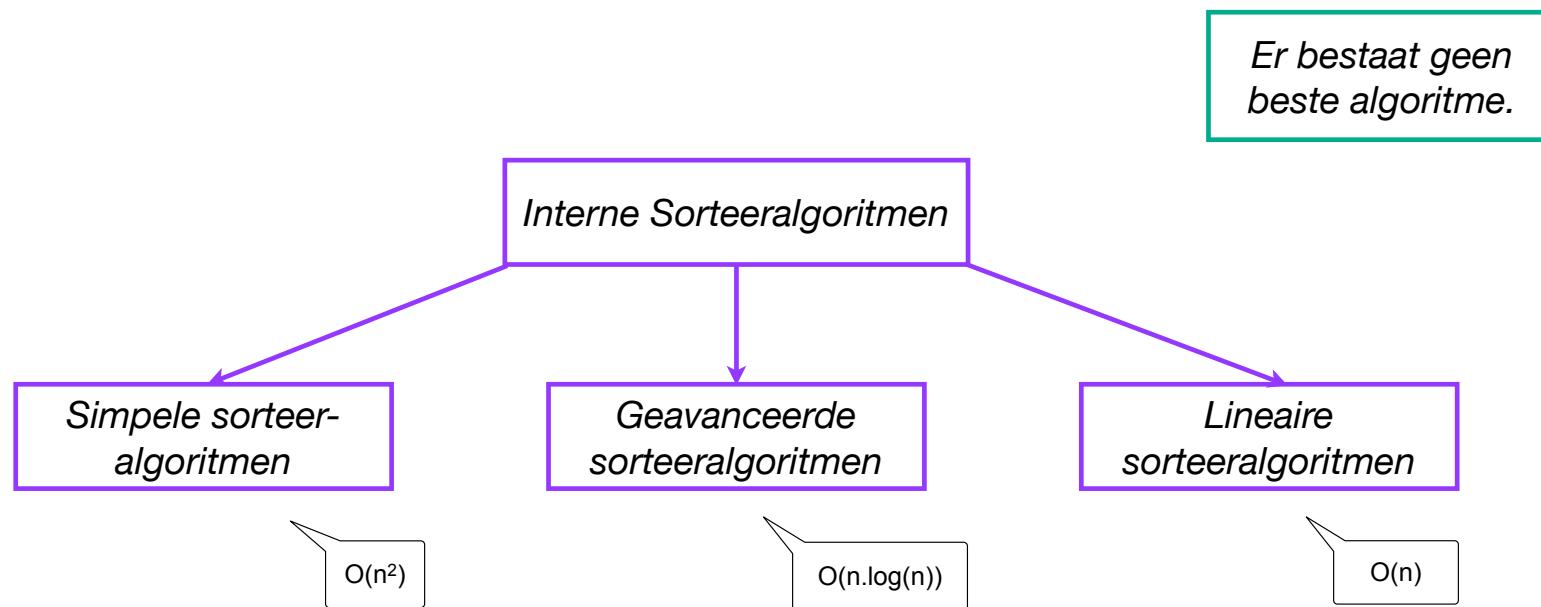
Liedjes in Spotify

Treinen in de uurroosters

Boeken op Amazon

Studenten in CALI

Taxonomie van Sorteeralgoritmen



Terminologie

Wolfgang De Meuter	10F743	Algoritmen en Datastructuren 1	25
Joeri De Koster	10F727	Besturingssystemen en Systeemfundamenten	19
Joeri De Koster	10F727	Cloud Computing and Big Data Processing	19
Coen De Roover	10F744	Interpretatie van Computerprogramma's 1	18
Theo D'Hondt	10F746	Programming Language Engineering	50
Coen De Roover	10F744	Software Quality	18
Wolfgang De Meuter	10F743	Structuur van Computerprogramma's 1	25
Viviane Jonckers	10F746	Structuur van Computerprogramma's 1	34

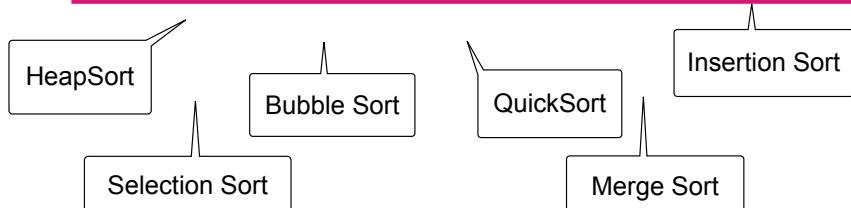
Bijvoorbeeld sorteer op vak

We sorteren “records” die bestaan uit “velden”. De velden waarop we sorteren heten **sleutelvelden** of **sorteervelden**. De andere noemen we **satellietvelden**.

Sorteeralgoritmen

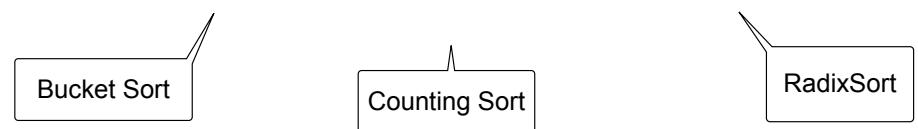
Comparatieve Sorteeralgoritmen

Ingewikkelde combinaties van “vergelijken van data” en “verplaatsen van data”. De sleutelvelden worden vergeleken. Alle velden worden verplaatst. Verplaatsen neemt meestal de vorm van “swap” aan: verwisselen



Niet-comparatieve Sorteeralgoritmen

De velden wordt in detail beschouwd en soms ontleed in onderdelen. Er wordt uitgerekend (bvb op basis van de onderdelen) waar ze moeten staan, of er wordt geturfd hoeveel records er groter of kleiner zijn



Comparatieve Sorteeralgoritmen: Vergelijken

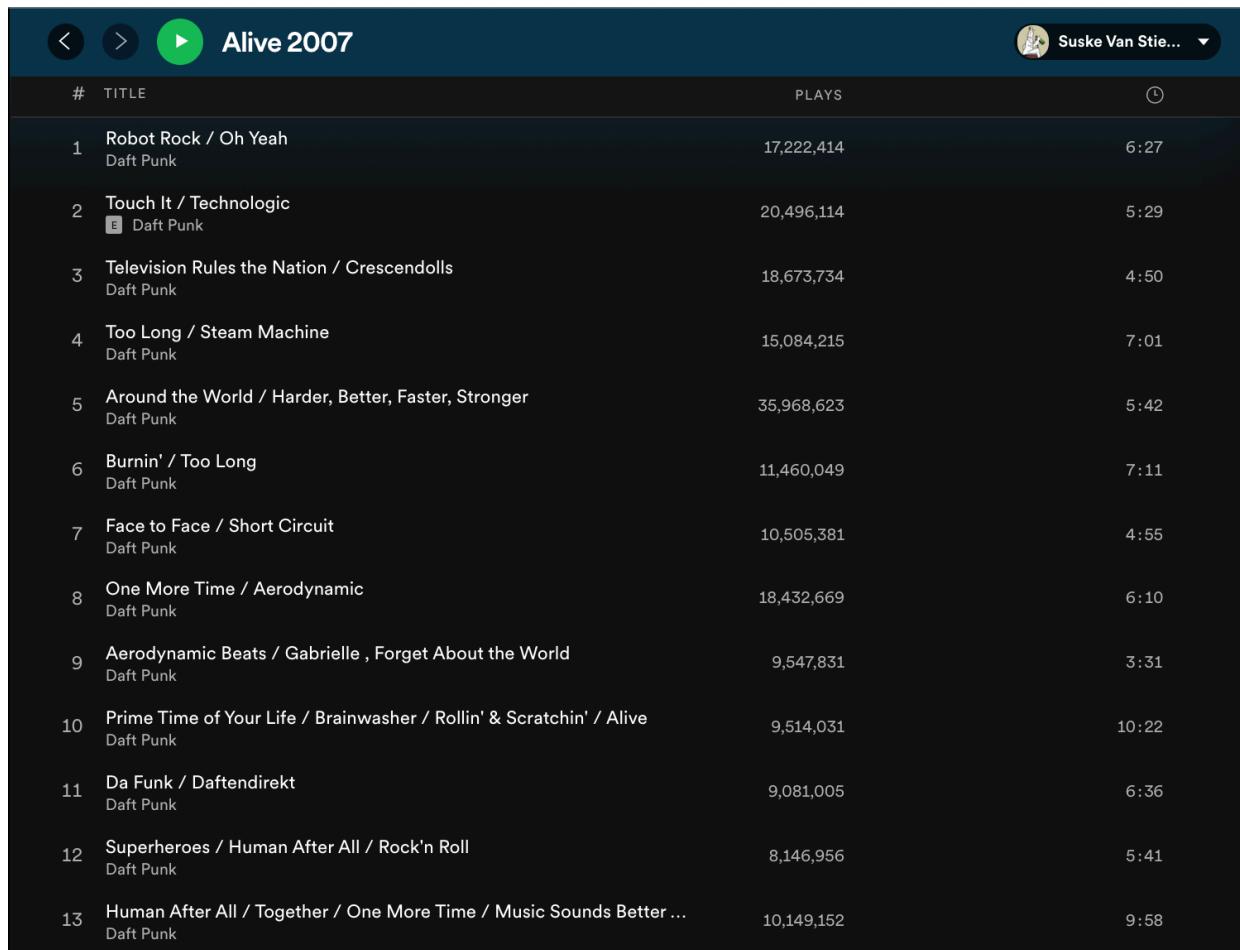
```
> (define people (vector '("Paul" 41) '("Anna-Mae" 44) '("Selma" 23)
                           '("Kenny" 68) '("Roger" 41) '("Paul" 22)))
> (sort people (lambda (p1 p2) (string<? (car p1) (car p2))))
> people
#{"Anna-Mae" 44)
 ("Kenny" 68)
 ("Paul" 41)
 ("Paul" 22)
 ("Roger" 41)
 ("Selma" 23)}
> (set! people (vector '("Paul" 41) '("Anna-Mae" 44) '("Selma" 23)
                           '("Kenny" 68) '("Roger" 41) '("Paul" 22)))
> (sort people (lambda (p1 p2) (< (cadr p1) (cadr p2))))
> people
#{"Paul" 22)
 ("Selma" 23)
 ("Paul" 41)
 ("Roger" 41)
 ("Anna-Mae" 44)
 ("Kenny" 68)}
```

We gebruiken een totale orde <<?

We gebruiken een totale orde <<?



Comparatieve Sorteeralgoritmen: Verplaatsen



The screenshot shows a dark-themed music player interface. At the top, there are navigation icons (back, forward, play/pause) and the album title "Alive 2007". On the right, there is a user profile picture and the name "Suske Van Stie...". Below the title, the table has three columns: "#", "TITLE", and "PLAYS". The "TITLE" column includes the artist name in parentheses. The "PLAYS" column shows the number of plays, and the last column shows the duration of each song.

#	TITLE	PLAYS	
1	Robot Rock / Oh Yeah Daft Punk	17,222,414	6:27
2	Touch It / Technologic Daft Punk	20,496,114	5:29
3	Television Rules the Nation / Crescendolls Daft Punk	18,673,734	4:50
4	Too Long / Steam Machine Daft Punk	15,084,215	7:01
5	Around the World / Harder, Better, Faster, Stronger Daft Punk	35,968,623	5:42
6	Burnin' / Too Long Daft Punk	11,460,049	7:11
7	Face to Face / Short Circuit Daft Punk	10,505,381	4:55
8	One More Time / Aerodynamic Daft Punk	18,432,669	6:10
9	Aerodynamic Beats / Gabrielle , Forget About the World Daft Punk	9,547,831	3:31
10	Prime Time of Your Life / Brainwisher / Rollin' & Scratchin' / Alive Daft Punk	9,514,031	10:22
11	Da Funk / Daftendirekt Daft Punk	9,081,005	6:36
12	Superheroes / Human After All / Rock'n Roll Daft Punk	8,146,956	5:41
13	Human After All / Together / One More Time / Music Sounds Better ... Daft Punk	10,149,152	9:58

hangt af van de
representatie

Verplaatsen: 1'ste methode

```
(define playlist-1
  (vector
    (vector 1 "One More Time- Aerodynamic" "8:03" "Daft Punk")
    (vector 2 "Face to Face- Harder Better Faster Stronger" "4:55" "Daft Punk")
    (vector 3 "Too Long" "5:10" "Daft Punk ")
    (vector 4 "Around the World- Harder Better Faster Stronger" "7:27" "Daft Punk")
    (vector 5 "Steam Machine" "1:39" "Daft Punk")
    (vector 6 "Crescendolls-Too Long- High Life" "7:41" "Daft Punk")
    (vector 7 "Television Rules the Nation" "2:47" "Daft Punk")
    (vector 8 "Technologic" "5:29" "Daft Punk ")
    (vector 9 "Superheroes- Human After All" "6:13" "Daft Punk")
    (vector 10 "Da Funk" "5:59" "Daft Punk")
    (vector 11 "The Brainwasher- The Primetime of Your Life- Steam Machine" "12:37" "Daft Punk")
    (vector 12 "Robot Rock-Oh Yeah" "6:36" "Daft Punk")))
```

```
(let ((song (vector-ref playlist-1 i)))
  (vector-set! playlist-1 i (vector-ref playlist-1 j))
  (vector-set! playlist-1 j song))
```

by reference

Verplaatsen: 2'de methode

```
(define playlist-2
  (vector
    (vector 1 2 3 4 5 6 7 8 9 10 11 12)
    (vector "One More Time- Aerodynamic" "Face to Face- Harder Better Faster Stronger"
            "Too Long" "Around the World- Harder Better Faster Stronger"
            "Steam Machine" "Crescendolls-Too Long- High Life" "Television Rules the Nation"
            "Technologic" "Superheroes- Human After All" "Da Funk"
            "The Brainwasher- The Primetime of Your Life- Steam Machine" "Robot Rock-Oh Yeah")
    (vector "8:03" "4:55" "5:10" "7:27" "1:39" "7:41" "2:47" "5:29" "6:13" "5:59" "12:37" "6:36")
    (vector "Daft Punk" "Daft Punk" "Daft Punk" "Daft Punk" "Daft Punk" "Daft Punk"
            "Daft Punk" "Daft Punk" "Daft Punk" "Daft Punk" "Daft Punk" "Daft Punk")))
```

```
(let ((nr (vector-ref (vector-ref playlist-2 0) i))
      (title (vector-ref (vector-ref playlist-2 1) i)))
      (time (vector-ref (vector-ref playlist-2 2) i)))
      (artist (vector-ref (vector-ref playlist-2 3) i)))
  (vector-set! (vector-ref playlist-2 0) i (vector-ref (vector-ref playlist-2 0) j))
  (vector-set! (vector-ref playlist-2 1) i (vector-ref (vector-ref playlist-2 1) j))
  (vector-set! (vector-ref playlist-2 2) i (vector-ref (vector-ref playlist-2 2) j))
  (vector-set! (vector-ref playlist-2 3) i (vector-ref (vector-ref playlist-2 3) j))
  (vector-set! (vector-ref playlist-2 0) j nr)
  (vector-set! (vector-ref playlist-2 1) j title)
  (vector-set! (vector-ref playlist-2 2) j time)
  (vector-set! (vector-ref playlist-2 3) j artist)))
```

by copy

Verplaatsen: een alternatief

```
(define playlist-1
  (vector
    (vector 1 "One More Time- Aerodynamic" "8:03" "Daft Punk")
    (vector 2 "Face to Face- Harder Better Faster Stronger" "4:55" "Daft Punk")
    (vector 3 "Too Long" "5:10" "Daft Punk ")
    (vector 4 "Around the World- Harder Better Faster Stronger" "7:27" "Daft Punk")
    (vector 5 "Steam Machine" "1:39" "Daft Punk")
    (vector 6 "Crescendolls-Too Long- High Life" "7:41" "Daft Punk")
    (vector 7 "Television Rules the Nation" "2:47" "Daft Punk")
    (vector 8 "Technologic" "5:29" "Daft Punk ")
    (vector 9 "Superheroes- Human After All" "6:13" "Daft Punk")
    (vector 10 "Da Funk" "5:59" "Daft Punk")
    (vector 11 "The Brainwasher- The Primetime of Your Life- Steam Machine" "12:37" "Daft Punk")
    (vector 12 "Robot Rock-Oh Yeah" "6:36" "Daft Punk")))
```

```
#(5 7 2 3 8 10 9 12 4 6 1 11)
```

creëren van een
index-vector

Stabiliteit van Sorteeralgoritmen

Een sorteeralgoritme heet *stabiel* als het de *relatieve orde van velden met gelijke sleutel* bewaart

... 5 ... "Word Up" ... "Korn" ...
... 6 ... "Brothers" ... "Tiga" ...
... 7 ... "Word Up" ... "Cameo" ...

Sommige algoritmen zijn van nature stabiel

Voor sommige algoritmen hangt het af van <<?: strikt of niet.

Sommige algoritmen zijn niet stabiel tenzij de originele locatie een deel van de sleutel wordt

... 6 ... "Brothers" ... "Tiga" ...
... 5 ... "Word Up" ... "Korn" ...
... 7 ... "Word Up" ... "Cameo" ...

... 6 ... "Brothers" ... "Tiga" ...
... 7 ... "Word Up" ... "Cameo" ...
... 5 ... "Word Up" ... "Korn" ...



In-place-heid van Sorteeralgoritmen

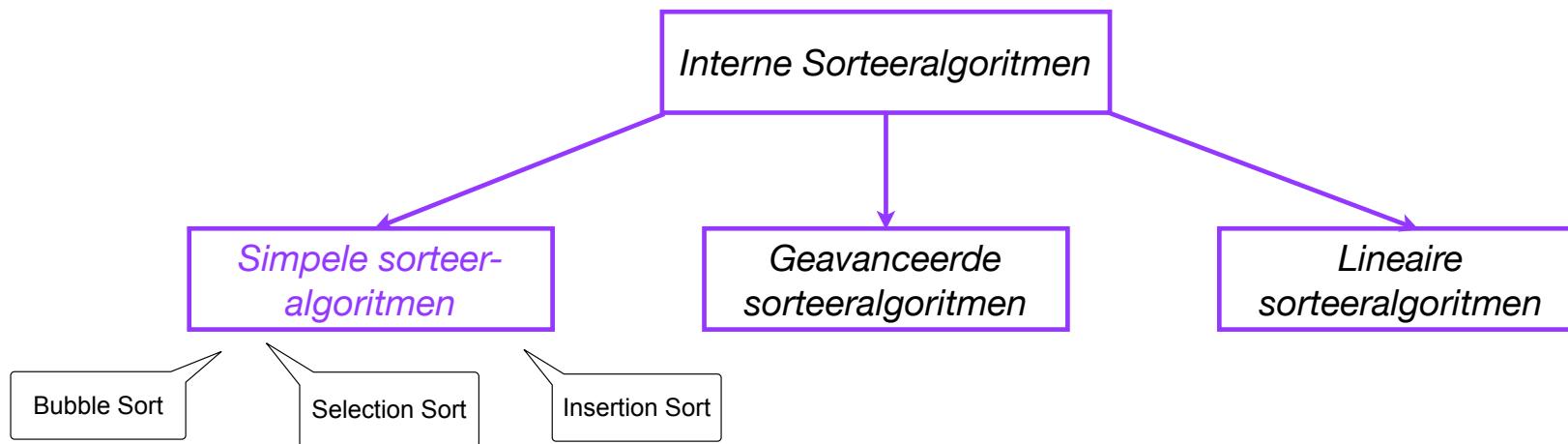
Sommige algoritmen zijn van nature uit niet in-place

Sommige algoritmen zijn ronduit in-place

Sommige algoritmen zijn niet in-place maar wel in-place te krijgen door substantieel andere implementatie

Een sorteeralgoritme heet *in-place* als het geen extra geheugen gebruikt buiten het geheugen reeds ingenomen door de te-sorteren data

Taxonomie van Sorteeralgoritmen

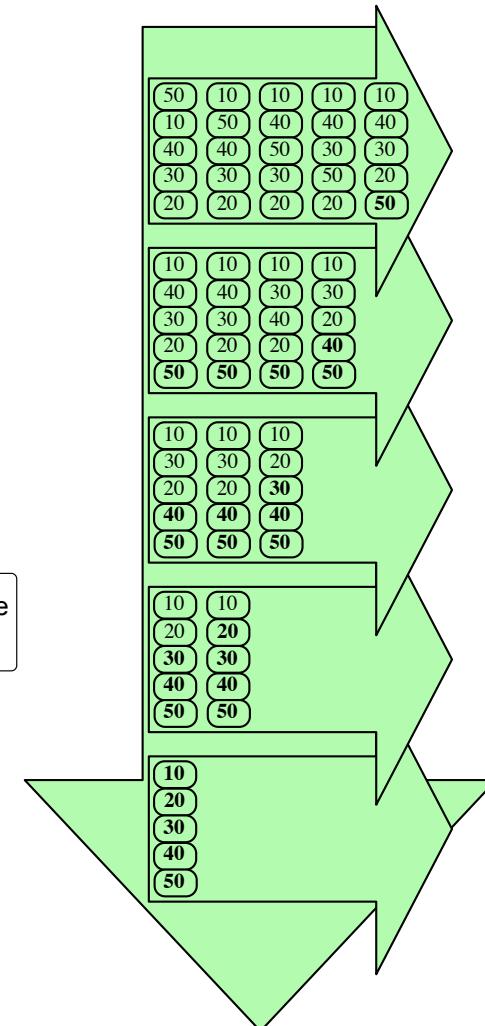
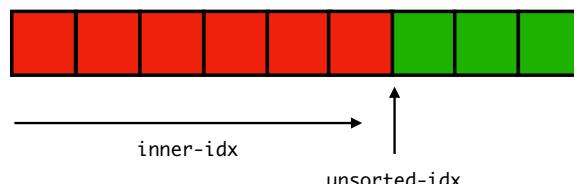


Bubblesort: Basisidee

forall unsorted-idx van $n-2$ naar 0
 forall inner-idx van 0 tot unsorted-idx
 Als elementen op inner-idx en inner-idx+1 fout staan: swap ze

Als er in de inner loop niets geswapt werd is het meteen gedaan!

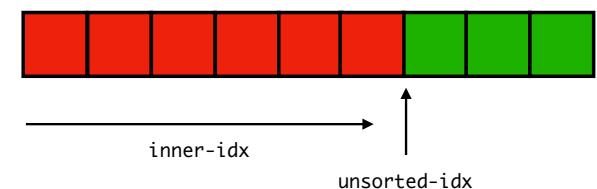
Bubbel telkens het grootste element naar achter



Bubblesort: Implementatie

```
(define (bubble-sort vector <<?)  
  (define (bubble-swap vector idx1 idx2)  
    (let ((keep (vector-ref vector idx1)))  
      (vector-set! vector idx1 (vector-ref vector idx2))  
      (vector-set! vector idx2 keep)  
      #t))  
  (let outer-loop  
    ((unsorted-idx (- (vector-length vector) 2)))  
    (if (>= unsorted-idx 0)  
        (if (let inner-loop  
                  ((inner-idx 0)  
                   (has-changed? #f))  
                  (if (> inner-idx unsorted-idx)  
                      has-changed?  
                      (inner-loop (+ inner-idx 1)  
                                 (if (<<? (vector-ref vector (+ inner-idx 1))  
                                         (vector-ref vector inner-idx))  
                                     (bubble-swap vector inner-idx (+ inner-idx 1))  
                                     has-changed?))))  
            (outer-loop (- unsorted-idx 1))))))
```

\forall unsorted-idx van $n-2$ naar 0
 \forall inner-idx van 0 tot unsorted-idx
 Als elementen op inner-idx en inner-idx+1 fout staan: swap ze
 Als er in de inner loop niets geswapt werd is het meteen gedaan!



Bubbel telkens het grootste element naar achter



Bubblesort: Performantie

De i 'de inner loop doet $n-i$ werk. Dus $b(i) = n-i$.

Best case: inner loop wordt slechts 1 keer uitgevoerd en has-changed? blijft #f. De outer loop stopt dus meteen. Dus $r(n) = 1$.

$$\sum_{i=1}^{r(n)} b(i) = n-1 \in \Omega(n)$$

Best-case: $n-1$ compares and 0 swaps.

Worst case: inner loop wordt n keer uitgevoerd en has-changed? wordt telkens #t. Dus $r(n) = n$.

$$\sum_{i=1}^{r(n)} b(i) = \frac{n(n-1)}{2} \in O(n^2)$$

Worst-case: $\frac{n(n-1)}{2}$ compares and $\frac{n(n-1)}{2}$ swaps.

Bubblesort: Eigenschappen

Bubble sort is stabiel als je een strikte ongelijkheid kiest

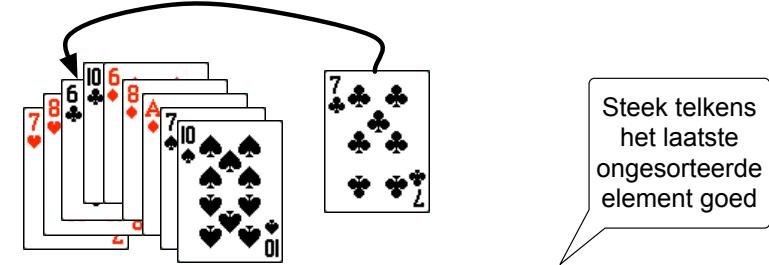
Bubble-sort is makkelijk te herschrijven voor gelinkte lijsten

Bubble sort is duidelijk in-place

Leuke naam maar dat is dan ook het enige!

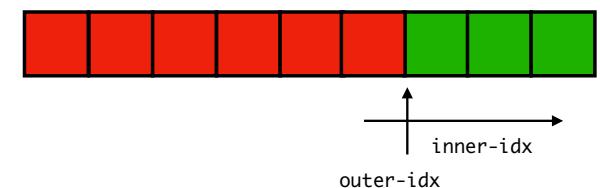
Insertion Sort: Implementatie

```
(define (insertion-sort vector <<?)  
  (define (>=? x y) (not (<<? x y)))  
  (let outer-loop  
    ((outer-idx (- (vector-length vector) 2)))  
    (let  
        ((current (vector-ref vector outer-idx)))  
      (vector-set!  
       vector  
       (let inner-loop  
         ((inner-idx (+ 1 outer-idx)))  
         (cond  
           ((or (>= inner-idx (vector-length vector))  
                 (>=? (vector-ref vector inner-idx)  
                       current))  
            (- inner-idx 1))  
           (else  
            (vector-set! vector (- inner-idx 1)  
                        (vector-ref vector inner-idx))  
            (inner-loop (+ inner-idx 1))))))  
        current)  
      (if (> outer-idx 0)  
          (outer-loop (- outer-idx 1)))))))
```



\forall current op outer-idx van $n-2$ naar 0
Verwissel element current met het element op de volgende positie:

- \forall inner-idx van outer-idx tot n:
- Als current *kleiner* is dan het element op inner-idx of einde vector bereikt: inner-idx-1 gevonden en stop.
 - Anders: schuif element op inner-idx naar inner-idx-1 en doe verder.



Insertion Sort: Performantie

Average Case: $n-1$ keer met in de i 'de keer $\frac{i}{2}$ werk.

Dus $\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$ stappen. Dit geeft $O(n^2)$.

Hoeveel keer wordt de inner loop uitgevoerd?

Best Case (gesorteerde vector): 1 keer.

Dit geeft $\Omega(n)$

Worst Case (ongesorteerde vector): $n-1$ keer met de i 'de keer i werk. Dus $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$.

Dit geeft $O(n^2)$

Het aantal compares is altijd gelijk aan het aantal moves bij insertion sort

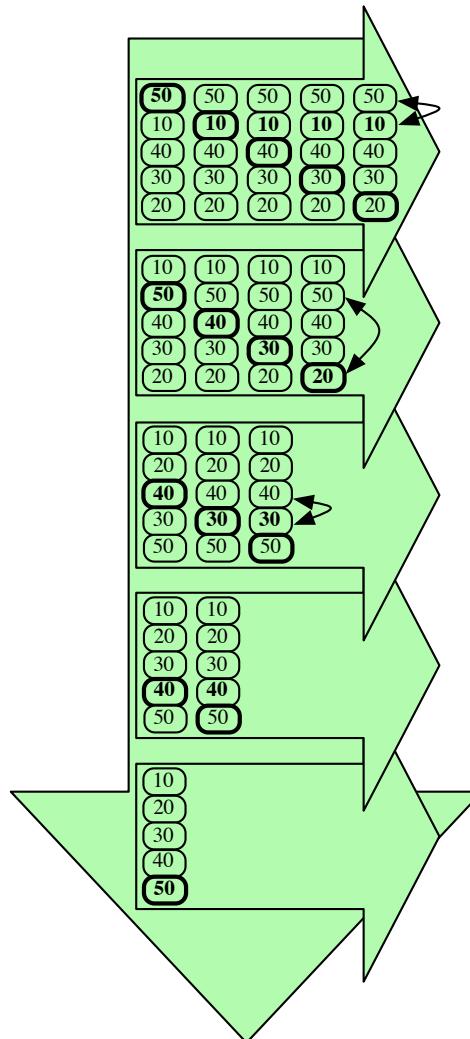
Insertion Sort: Eigenschappen

Insertion-sort is makkelijk te herschrijven voor gelinkte lijsten

Insertion sort is stabiel als je een strikte ongelijkheid kiest

Insertion sort is duidelijk in-place

Selection Sort: Basisidee



\forall outer-idx van 0 naar $n-1$.

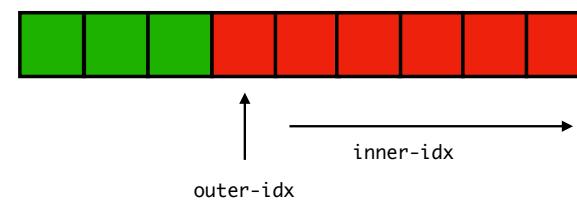
Verwissel dit element met het element dat zit op:

smallest-id is de locatie het kleinste element “tot nu toe”.

Voor elke inner-idx van outer-idx+1 tot n :

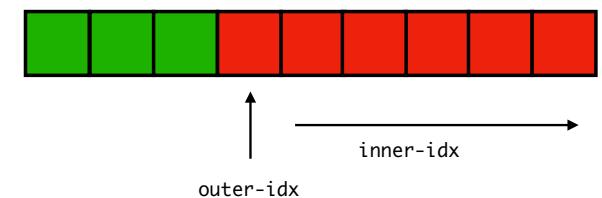
Als het element op inner-idx kleiner is dan het element op smallest-idx dan wordt inner-idx de nieuwe waarde van smallest-idx

Selecteer telkens het kleinste element uit de elementen die nog niet gesorteerd zijn



Selection Sort: Implementatie

```
(define (selection-sort vector <<?)
  (define (swap vector i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (let outer-loop
    ((outer-idx 0))
    (swap vector
          outer-idx
          (let inner-loop
            ((inner-idx (+ outer-idx 1))
             (smallest-idx outer-idx))
            (cond
              ((>= inner-idx (vector-length vector))
               smallest-idx)
              ((<<? (vector-ref vector inner-idx)
                     (vector-ref vector smallest-idx))
               (inner-loop (+ inner-idx 1) inner-idx)))
              (else
               (inner-loop (+ inner-idx 1) smallest-idx))))))
    (if (< outer-idx (- (vector-length vector) 1))
        (outer-loop (+ outer-idx 1))))))
```



Selection Sort: Performantie

De outer loop werkt van 0 naar $n-1$. De inner loop werkt van outer-idx + 1 naar $n-1$. Dus $r(n) = n$ en $b(i) = n-i$. *Altijd!*

$$\sum_{i=1}^{r(n)} b(i) = \sum_{i=1}^n (n - i) = (n^2 - n) = O(n^2). \text{ *Altijd!*}$$

Maar hiervan zijn er slechts $n-1$ swaps

Selection Sort: Eigenschappen

Selection sort is niet stabiel

Waarom?

Selection sort is duidelijk in-place

Selection-sort is makkelijk te herschrijven voor gelinkte lijsten

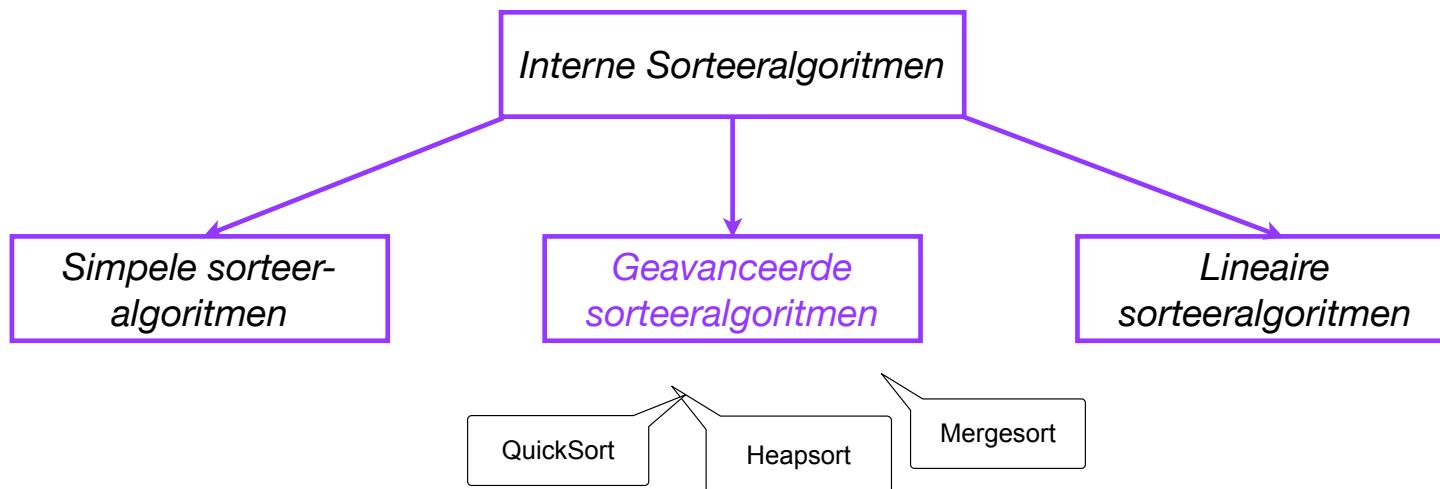
Simple Sort Algorithms: Overview

	worst-case	best-case
bubble-sort	$O(n^2)$	$\Omega(n)$
insertion-sort	$O(n^2)$	$\Omega(n)$
selection-sort	$\Theta(n^2)$	$\Theta(n^2)$

Uitgesplitst

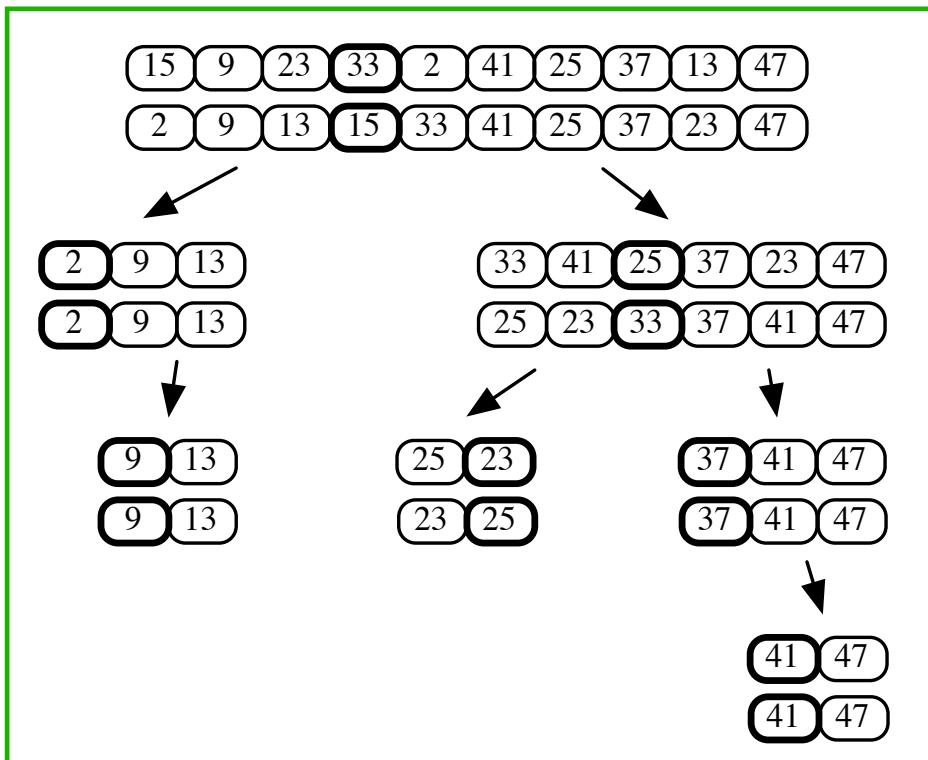
	worst-case	best-case
bubble-sort	M: $O(n^2)$ C: $O(n^2)$	M: 0 C: $\Omega(n)$
insertion-sort	M: $O(n^2)$ C: $O(n^2)$	M: $\Omega(n)$ C: $\Omega(n)$
selection-sort	M: $\Theta(n)$ C: $\Theta(n^2)$	M: $\Theta(n)$ C: $\Theta(n^2)$

Taxonomie van Sorteeralgoritmen



QuickSort: Basisidee

C.A.R. Hoare (1959)



Om de elementen van l tot r te sorteren

Kies het element op plaats l (= de **pivot**)

Ga van l naar r en tegelijk van r naar l . Vergelijk ieder element met de pivot. Gooi grotere elementen naar rechts en kleinere naar links. Stop de pivot in het **midden** m .

Quicksort van l tot $m-1$

Quicksort van $m+1$ tot r

QuickSort: Implementatie

```
(define (quicksort vector <<?)
  (define (swap i j)
    (let ((keep (vector-ref vector i)))
      (vector-set! vector i (vector-ref vector j))
      (vector-set! vector j keep)))
  (define (shift-to-right i x)
    (if (<<? (vector-ref vector i) x)
        (shift-to-right (+ i 1) x)
        i))
  (define (shift-to-left j x)
    (if (<<? x (vector-ref vector j))
        (shift-to-left (- j 1) x)
        j)))
```



```
(define (partition pivot i j)
  (let ((shifted-i (shift-to-right i pivot))
        (shifted-j (shift-to-left j pivot)))
    (cond ((< shifted-i shifted-j)
           (swap shifted-i shifted-j)
           (partition pivot shifted-i (- shifted-j 1)))
          ((>= shifted-i shifted-j)
           shifted-j))))
(define (quicksort-main l r)
  (if (< l r)
      (begin
        (if (<<? (vector-ref vector r)
                  (vector-ref vector l))
            (swap l r))
        (let ((m (partition (vector-ref vector l) (+ l 1) r)))
          (swap l m)
          (quicksort-main l (- m 1))
          (quicksort-main (+ m 1) r))))))
(quicksort-main 0 (- (vector-length vector) 1)))
```

Sentinels



Verdeel&Heers Algoritmen



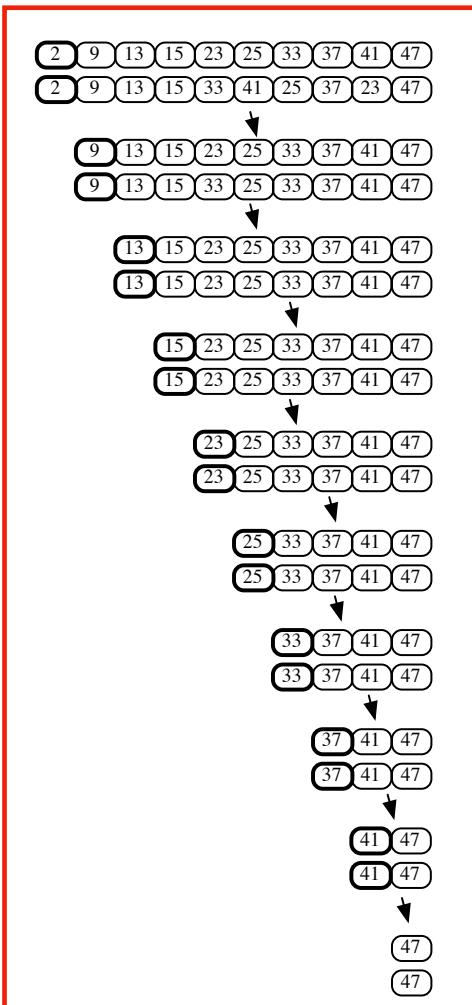
1) **Divide**: Verdeel het op-te-lossen probleem in deelproblemen

2) **Conquer**: Bereken de oplossing voor de deelproblemen

3) **Combine**: Combineer de deeloplossingen tot een totaaloplossing voor het op-te-lossen probleem

Dikwijls weggelaten

QuickSort kan Ontaarden



De pivot komt toevallig altijd bij een uiteinde

Quicksort degenerereert!

QuickSort: Performantie

De pivot valt telkens precies in het midden

Best Case. De recursiediepte is $\log_2(n)$

De pivot zit telkens precies aan een uiteinde

Worst Case. De recursiediepte is n

Zij $W_i(n)$ de hoeveelheid werk gedaan op recursiediepte i

$$\begin{aligned}W_0(n) &= n \\W_1(n) &= n-1 \\W_2(n) &= n-3 \\W_3(n) &= n-7\end{aligned}$$

$$W_i(n) = n - (2^i - 1)$$

$$\begin{aligned}\text{De totale hoeveelheid werk} \\&= \sum_{i=0}^{\log_2(n)} W_i(n) = \sum_{i=0}^{\log_2(n)} (n - (2^i - 1)) \\&= n \cdot \log_2(n) - (2n - 1) + \log_2(n) \\&\in \Omega(n \cdot \log_2(n))\end{aligned}$$

$$\begin{aligned}W_0(n) &= n \\W_1(n) &= n-1 \\W_2(n) &= n-2 \\W_3(n) &= n-3\end{aligned}$$

$$W_i(n) = n - i$$

$$\begin{aligned}\text{De totale hoeveelheid werk} \\&= \sum_{i=0}^n W_i(n) = \sum_{i=0}^n (n - i) \\&= n^2 - \sum_{i=0}^n i \\&\in O(n^2)\end{aligned}$$

Het “gemiddelde” geval

Soms eens een goede pivot. Soms eens een slechte. Evenveel goede als slechte.

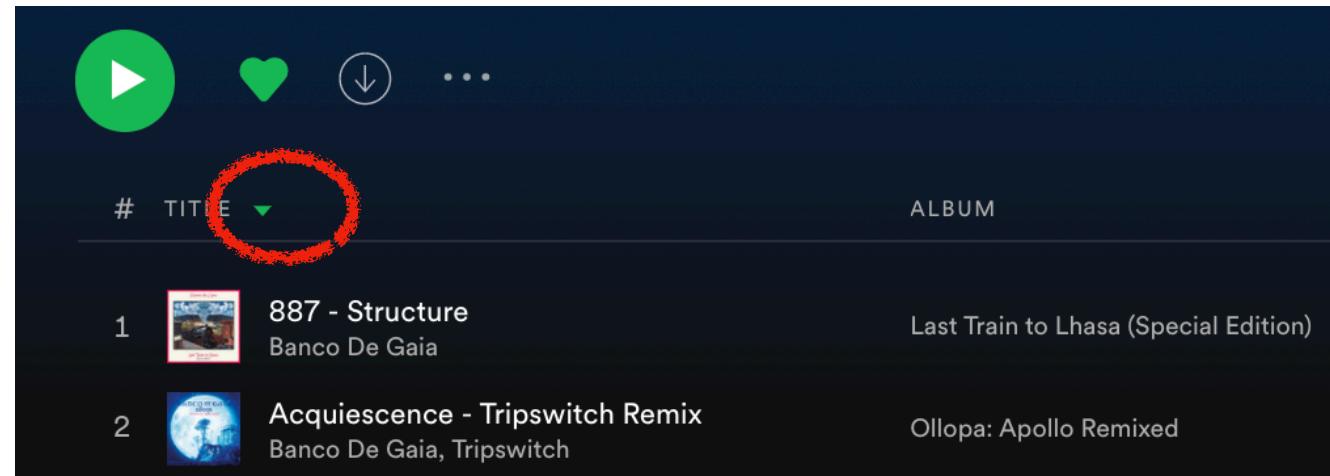
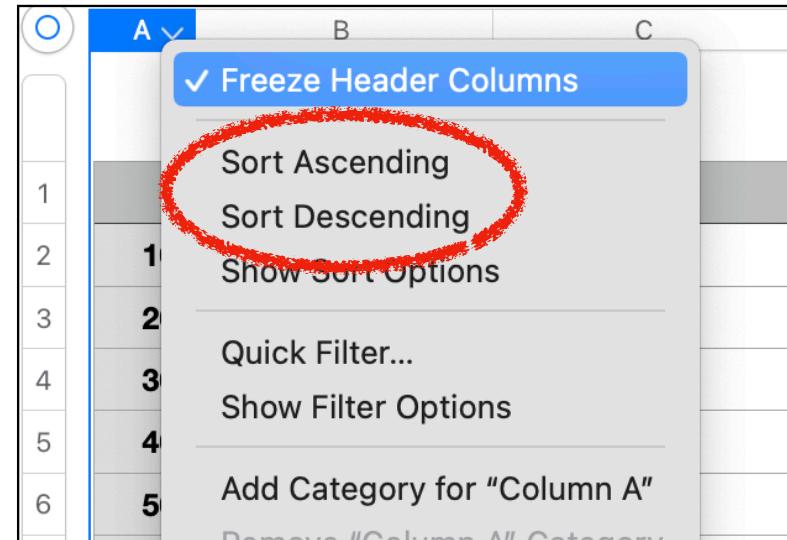
Een goede na een slechte levert 3 delen van lengte 1, lengte $\frac{n-1}{2}$ en lengte $\frac{n-1}{2}$

Dit levert dus een recursieboom die **dubbel** zo diep is als het beste geval. Onafhankelijk van n!

“In het echt” krijgen we 80% goed en 20% slechte. Dus 1 5’de dieper

Het ergste geval

Het ergste geval is helaas
geen esoterisch geval



Marginaliseren van de Worst-Case

```
(define (quicksort vector <<?)  
  (define (random-inbetween l r)  
    (+ l (random-integer (+ (- r l) 1))))  
  (define (swap i j)  
    ...)  
  (define (shift-to-right i x)  
    ...)  
  (define (shift-to-left j x)  
    ...)  
  (define (partition pivot i j)  
    ...)  
  (define (randomized-partition l r)  
    (swap l (random-inbetween l r))  
    (if (<<? (vector-ref vector r)  
               (vector-ref vector l))  
        (swap l r))  
    (partition (vector-ref vector l) (+ l 1) r)))  
  (define (quicksort-main vector l r)  
    (if (< l r)  
        (let ((m (randomized-partition l r)))  
          (swap l m)  
          (quicksort-main vector l (- m 1))  
          (quicksort-main vector (+ m 1) r)))  
    (quicksort-main vector 0 (- (vector-length vector) 1))))
```

Randomized Quicksort



Steek het geluk een handje toe

```
(define (quicksort vector <<?)  
  ...  
  (define (m3-partition l r)  
    (let ((middle (div (+ l r) 2)))  
      (swap middle (+ l 1))  
      (if(<<? (vector-ref vector l)  
               (vector-ref vector (+ l 1)))  
          (swap l (+ l 1)))  
      (if (<<? (vector-ref vector r)  
                 (vector-ref vector (+ l 1)))  
          (swap r (+ l 1)))  
      (if (<<? (vector-ref vector r)  
                 (vector-ref vector l))  
          (swap l r))  
      (partition (vector-ref vector l) (+ l 1) (- r 1))))  
  (define (quicksort-main vector l r)  
    (if (< l r)  
        (let ((m (m3-partition l r)))  
          (swap l m)  
          (quicksort-main vector l (- m 1))  
          (quicksort-main vector (+ m 1) r))))  
  (quicksort-main vector 0 (- (vector-length vector) 1)))
```

Mediaan van 3 QuickSort

Beschouw l, middle en r.
Het kleinste gaat in l+1, **mediaan in l**, het grootste in r en de overschot in middle.

Sentinels + mediaan



Nog een Technische verbetering

De “computationele overhead” van Quicksort is zeer groot. Recursie, bindingen, procedure oproepen, etc.

Waarde platform per platform bepalen door experimentatie

Ga vanaf een bepaalde grootte (bvb. $n=10$) over naar insertion sort of selection sort

QuickSort: Eigenschappen

Quicksort is
helemaal niet stabiel

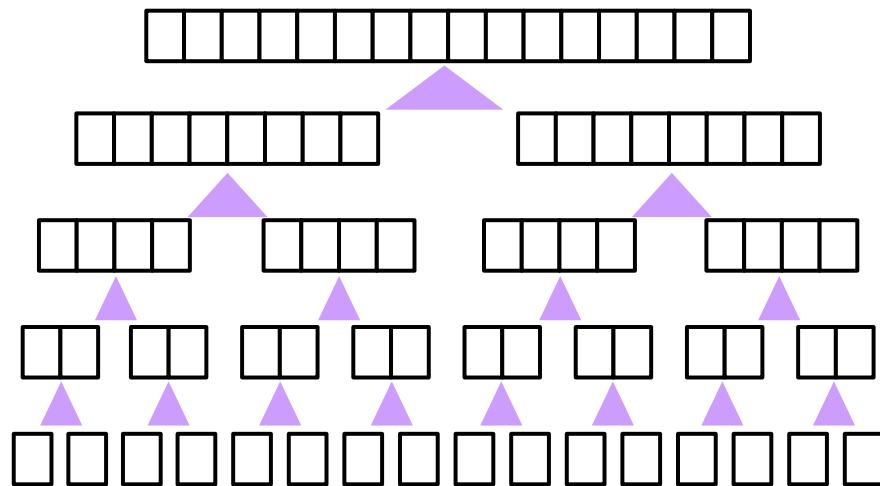
```
#(20 ... <20... 50 ... 15 ... 15' ... >20...)
```

Quicksort (basisvariante) werkt
wel goed op dubbel gelinkte lijsten
maar niet op andere lijsten

Quicksort is duidelijk niet in-place:
minstens $\Omega(\log(n))$ geheugen

Mergesort: Basisidee

von Neumann (1945)



We dalen recursief af en doen al het eigenlijke werk tijdens het **backtracken**

Mergesort: Implementatie

```
(define (merge-sort vector <<?)  
  (define (merge vector p q r)  
    ...)  
  (define (merge-sort-rec vector p r)  
    (if (< p r)  
        (let ((q (div (+ r p) 2)))  
          (merge-sort-rec vector p q)  
          (merge-sort-rec vector (+ q 1) r)  
          (merge vector p q r))))  
  
(merge-sort-rec vector 0 (- (vector-length vector) 1))  
vector)
```



Mergesort: Implementatie

```
(define (merge vector p q r)
  (let ((working-vector (make-vector (+ (- r p) 1))))
    (define (copy-back a b)
      ...)
    (define (flush-remaining k i until)
      ...)
    (define (merge-iter k i j)
      (cond ((and (<= i q) (<= j r))
              (let ((low1 (vector-ref vector i))
                    (low2 (vector-ref vector j)))
                (if (<<? low1 low2)
                    (begin
                      (vector-set! working-vector k low1)
                      (merge-iter (+ k 1) (+ i 1) j))
                    (begin
                      (vector-set! working-vector k low2)
                      (merge-iter (+ k 1) i (+ j 1))))))
            ((<= i q)
             (flush-remaining k i q))
            (else
             (flush-remaining k j r))))
      (merge-iter 0 p (+ q 1))))
```

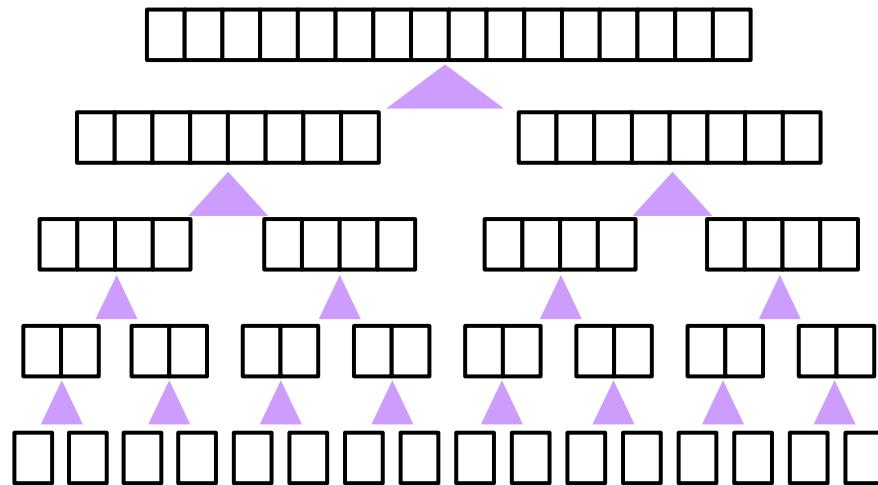
```
(define (copy-back a b)
  (vector-set! vector b (vector-ref working-vector a))
  (if (< a (- (vector-length working-vector) 1))
      (copy-back (+ a 1) (+ b 1)))
```



```
(define (flush-remaining k i until)
  (vector-set! working-vector k (vector-ref vector i))
  (if (< i until)
      (flush-remaining (+ k 1) (+ i 1) until)
      (copy-back 0 p)))
```



Mergesort: Performantie



$W_i(n) =$ de hoeveelheid werk op backtrackniveau i

$i \in [1.. \log_2(n)]$

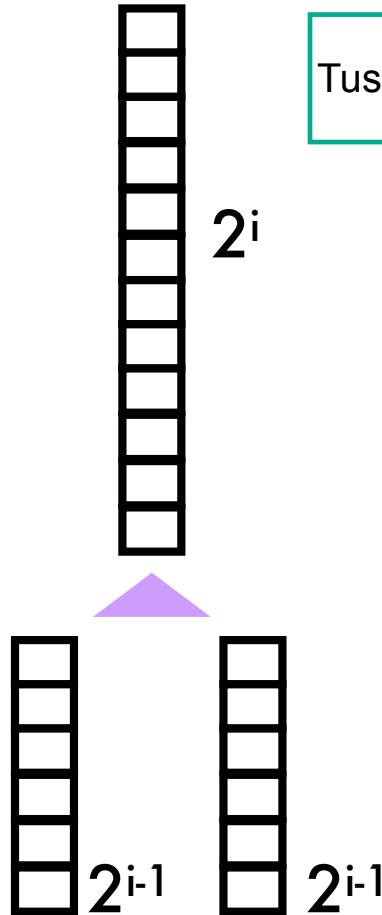
$$W_i(n) = 2^{\log(n)-i} M_i(n)$$

Dus is de totale hoeveelheid werk

$$= \sum_{i=1}^{\log_2(n)} W_i(n)$$

$M_i(n) =$ Hoeveelheid werk om 2 “regio’s” van lengte 2^{i-1} te mergen naar 1 van lengte 2^i

Het mergen van 2 “regio’s”



Tussen 2^{i-1} en 2^i compares

Dus het aantal stappen
van $M_i(n)$

$$\begin{aligned} &\leq 2^{i+1} + 2^i - 1 \\ &\leq 2^{i+1} + 2^i \\ &\leq 2^{i+1} + 2^{i+1} \\ &= 2^{i+2} \end{aligned}$$

Een keer lezen
en schrijven
Elk element moet moven: dus 2^{i+1} moves

Dus is de totale hoeveelheid werk

$$\begin{aligned} &= \sum_{i=1}^{\log_2(n)} W_i(n) \\ &= \sum_{i=1}^{\log_2(n)} 2^{\log_2(n)-i} M_i(n) \\ &\leq \sum_{i=1}^{\log_2(n)} 2^{\log_2(n)-i} 2^{i+2} \\ &\in O(n \cdot \log_2(n)) \end{aligned}$$

Mergesort: Eigenschappen

Mergesort is stabiel als je een niet-strikte ongelijkheid kiest

Mergesort is duidelijk niet in-place

Mergesort werkt extreem goed op lijsten. Het is de basis van externe sorteeralgoritmen.

Heapsort: Basisidee

Floyd&Williams (1964)

Gooi alle elementen op een heap



Pas n keer delete! toe op de heap en vul zo
de vector opnieuw Pas n keer delete! toe op
de heap en vul zo de vector opnieuw

Op eerste zicht niet in-place, maar...

Heapsort: Implementatie

```
(define (heapsort vector <<?)  
  (define >>? (lambda (x y) (not (<<? x y))))  
  (define heap (from-scheme-vector vector >>?))  
  (define (extract idx)  
    (vector-set! vector idx (delete! heap))  
    (if (> idx 0)  
        (extract (- idx 1)))  
    (extract (- (vector-length vector) 1))))
```

Het algoritme is in-place omdat
from-scheme-vector in-place is
(let op de omgekeerde orde >>?)



Heapsort: Performantie

Heapify is in $O(n)$. Nadien doet extract n keer delete! wat in $O(\log(n))$ is.

Dus: $O(n) + O(n.\log(n)) = O(n.\log(n))$

Heapsort is niet stabiel

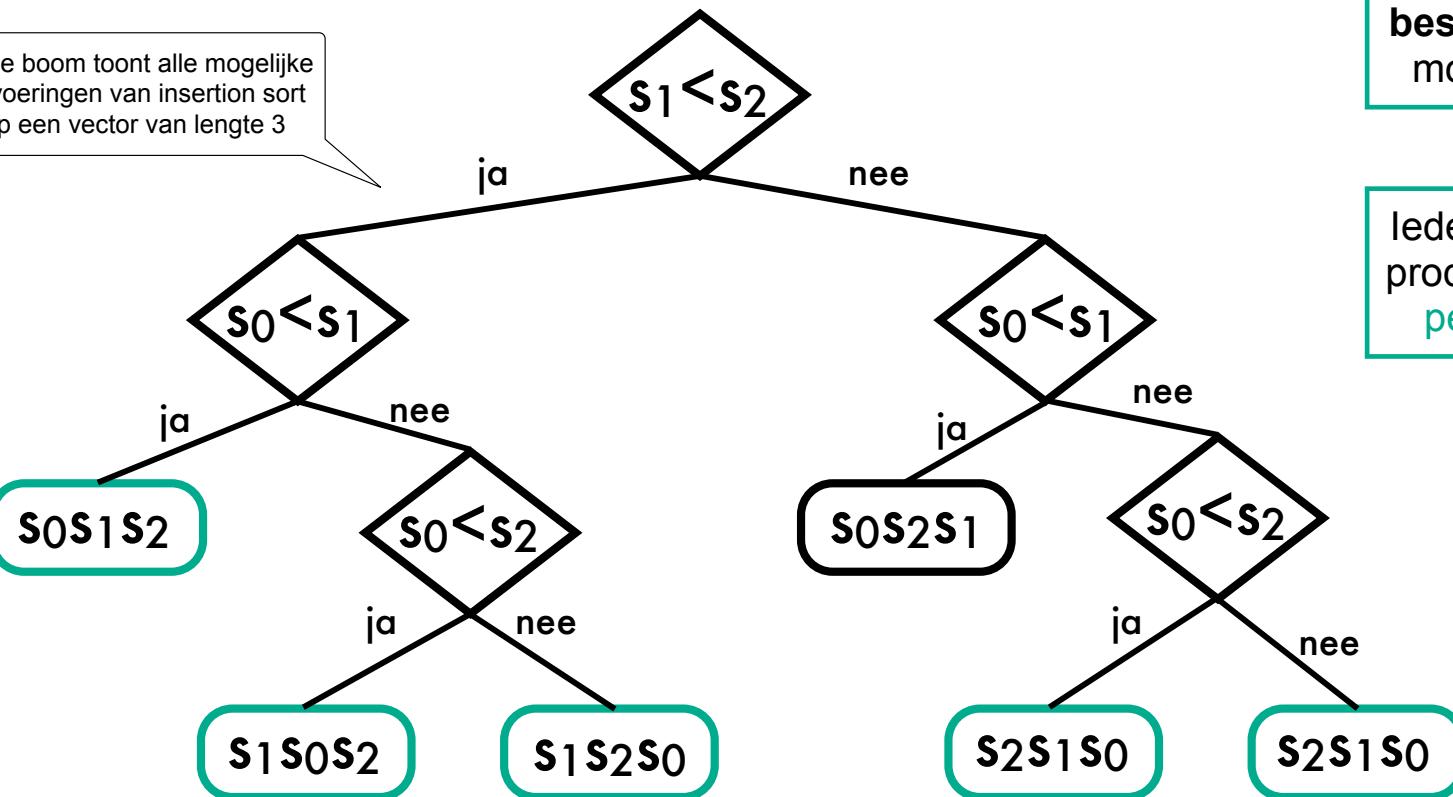
Waarom?

Slecht Nieuws...

Comparatief sorteren

Stelling: geen enkel algoritme dat gebaseerd is op vergelijkingen alleen kan sneller dan $n \log(n)$

Deze boom toont alle mogelijke uitvoeringen van insertion sort op een vector van lengte 3



Voor ieder algoritme kan je een **beslissingsboom** tekenen die alle mogelijke executions visualiseert.

Iedere uitvoering van het algoritme produceert één van de $n!$ mogelijke permutaties van de inputvector

Bewijs

Stelling: geen enkel algoritme dat gebaseerd is op vergelijkingen alleen kan sneller dan $n \cdot \log(n)$

Bewijs: $n!$ permutaties

Dus: $n!$ leaves

Dus: minstens $n!$ knopen

Dus: hoogte $h > \log(n!)$

Dus $h > n \cdot \log(n)$ want
 $\log(n!) \in \Omega(n \cdot \log(n))$

h is de hoogte van de beslissingsboom.
Dus minstens $\Omega(n \cdot \log(n))$ vergelijkingen!

Lemma: $\log(n!) \in \Omega(n \cdot \log(n))$

Bewijs:

$$\begin{aligned}
& \log(n!) \\
&= \log(1) + \log(2) + \dots + \log\left(\frac{n}{2}\right) + \dots + \log(n) \\
&\geq \log\left(\frac{n}{2}\right) + \dots + \log(n) \\
&\geq \log\left(\frac{n}{2}\right) + \dots + \log\left(\frac{n}{2}\right) \\
&= \frac{n}{2} \log\left(\frac{n}{2}\right) \\
&= \frac{n}{2} \log(n-1) \\
&> \frac{n}{2} \log(n) - \frac{n}{6} \log(n) \text{ voor } n > 8 = 2^3 (= n_0)
\end{aligned}$$

Dus $\log(n!) > \frac{1}{3}n \cdot \log(n)$ voor $n > 8$

Comparatief Sorteren: Algemene Conclusie

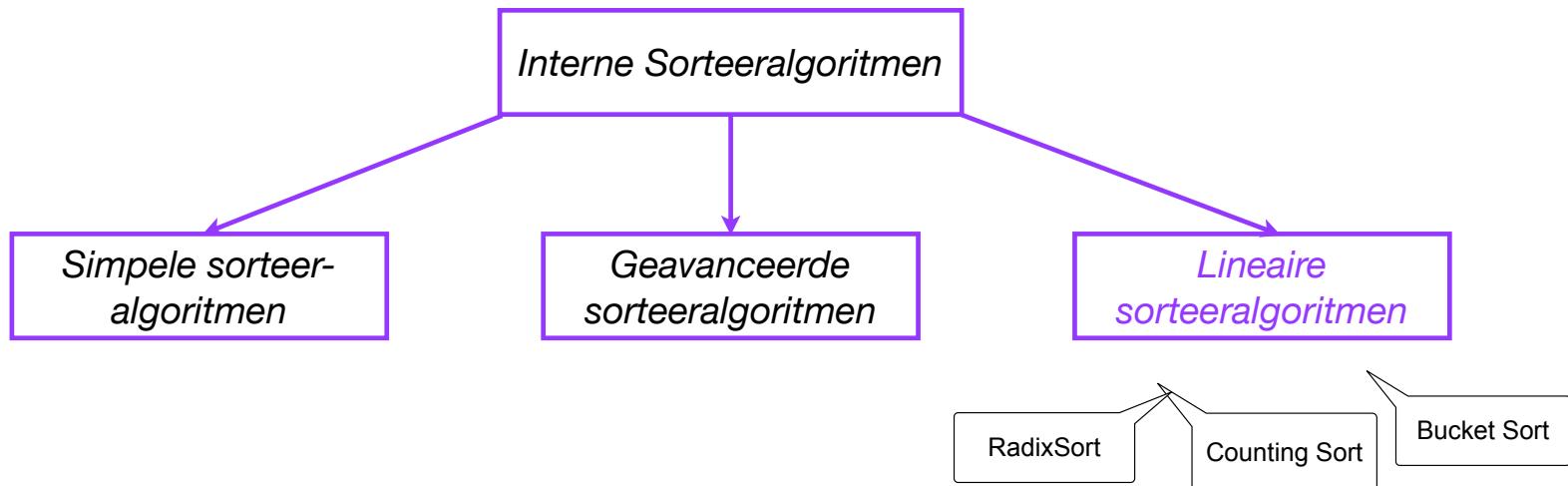
Vermijd bubble sort

Insertion sort is iets beter
dan selection sort, behalve
als moven duur is.

Quicksort wordt onperformant
voor kleine vectoren.

Mergesort is de beste om lijsten te sorteren.
Zie sorteren van “Big Data” in 2BA

Taxonomie van Sorteeralgoritmen



Deze algoritmen eisen meer kennis over de keys. Het zijn **niet langer comparatieve** sorteeralgoritmen.

Radixsort: basisidee

From Computer Desktop Encyclopedia
© 2000 The Computer Language Co., Inc.

68234 ASHLEY COMPANY		2911 S. TREMONT ST.		AUSTIN TX.		092740	98766	82509
CUST ID	CUSTOMER NAME	STREET ADDRESS		CITY AND STATE		INVOICE DATE	INVOICE NO.	INVOICE AMOUNT
1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
6	7	8	9	0	1	2	3	4
5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
7	8	9	0	1	2	3	4	5
6	7	8	9	0	1	2	3	4
5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
8	9	0	1	2	3	4	5	6
7	8	9	0	1	2	3	4	5
6	7	8	9	0	1	2	3	4
5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
9	0	1	2	3	4	5	6	7
8	9	0	1	2	3	4	5	6
7	8	9	0	1	2	3	4	5
6	7	8	9	0	1	2	3	4
5	6	7	8	9	0	1	2	3
4	5	6	7	8	9	0	1	2
3	4	5	6	7	8	9	0	1
2	3	4	5	6	7	8	9	0
1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8

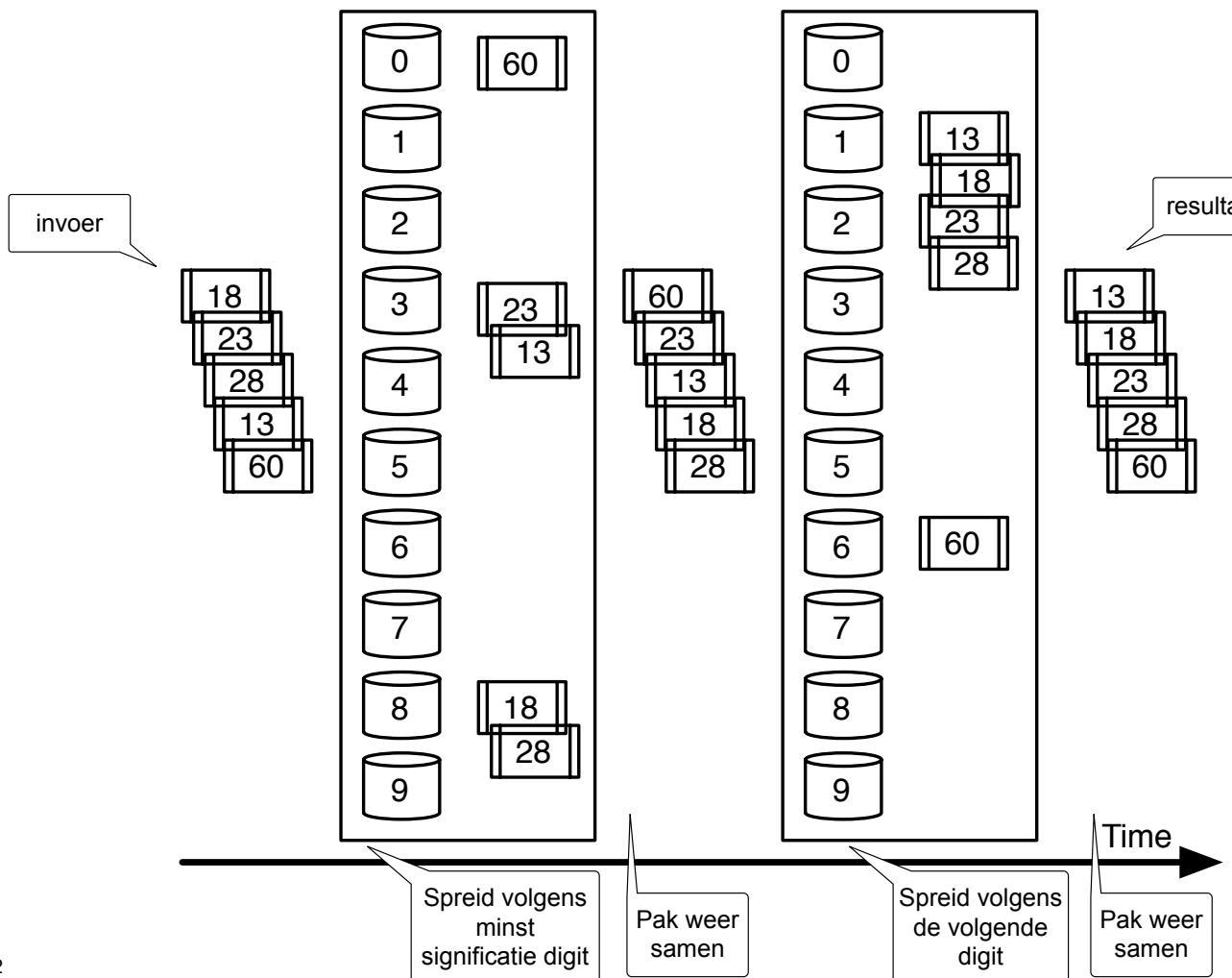


Radix = Grondtal

Getallen: radix = 10

Namen: radix = 26

Basisidee



Alle keys bestaan uit k “digits”

Alle digits kunnen N mogelijke waarden aannemen. Dit is de radix.

Er is een orde tussen deze N mogelijke waarden gedefinieerd. De geïnduceerde orde op de keys is de lexicografische orde.

Je moet met de minst significante digit starten.

Radix Sort: Het algoritme

```
(define (radix-sort slst key key-size)
  (define sort-bins (make-vector 10 '()))
  (define (spread lst digit-k)
    ...)
  (define (collect)
    (define (collect-iter index acc)
      (define (collect-list lst acc)
        (if (null? lst)
            (if (> index 0)
                (collect-iter (- index 1) acc)
                acc)
            (collect-list (cdr lst) (cons (car
              (let ((l-index (vector-ref sort-bins in
                (vector-set! sort-bins index '())
                (collect-list l-index acc)))))
              (collect-iter 9 '())))
            (collect-iter 9 '())))
      (define (radix-iter digit-k slst)
        (spread slst digit-k)
        (if (= digit-k key-size)
            (collect)
            (radix-iter (+ 1 digit-k) (collect))))
      (radix-iter 0 slst))
```

```
(define (spread lst digit-k)
  (define (digit item)
    (mod (div item (expt 10 digit-k)) 10))
  (define (spread-iter lst)
    (let ((idx (digit (key (car lst)))))
      (vector-set! sort-bins
        idx
        (cons (car lst)
          (vector-ref sort-bins idx)))
      (if (not (null? (cdr lst)))
          (spread-iter (cdr lst))))))
  (spread-iter lst))
```



Radix Sort: Eigenschappen

radix-iter wordt k
keer herhaald

In elke iteratie genereren
spread en collect $O(n)$ werk

Dus: $O(k \times n)$

Radix Sort is stabiel. Stabiliteit
van elke fase is cruciaal.

Radix Sort is niet in-place. De bins zijn
nodig en de lijsten. Voor die laatste zou
je cdr pointers kunnen hergebruiken.

Bucket Sort

De bins heten ook “buckets”. Bucket Sort is 1 pass van de Radix Sort

$O(n)$

Dit kan gebruikt worden:

- als je geen perfecte sort nodig hebt.
- als *preprocessing fase* voor een *insertion sort*.

(is goed op “mostly sorted data”)

Counting Sort: Basisidee

Counting Sort telt hoeveel keer iedere key voorkomt en stopt de getallen in een **vector entry per key**.

Keys zijn dus getallen!

Dan worden de accumulatieve sommen berekend van die getallen die dan nadien per key als index dienen.

Sorteer: 1 4 5 4 3 5 7 4 5 3 2 1

12 getallen. We tellen 2 keer 1, 1 keer 2, 2 keer 3, 3 keer 4, 3 keer 5, 0 keer 6 en 1 keer 7.

Accumulatieve sommen: 2 3 5 8 11 11 12

Sommen gebruiken als index: 2 3 5 8 11 11 12

1 → 2 en 1 3 5 8 11 11 12
2 → 3 en 1 2 5 8 11 11 12
3 → 5 en 1 2 4 8 11 11 12
5 → 11 en 1 2 4 8 10 11 12
4 → 8 en 1 2 4 7 10 11 12
7 → 12 en 1 2 4 7 10 11 11
5 → 10 en 1 2 4 7 9 11 11
3 → 4 en 1 2 3 7 9 11 11
4 → 7 en 1 2 3 6 9 11 11
5 → 9 en 1 2 3 6 8 11 11
4 → 6 en 1 2 3 5 8 11 11
1 → 1 en 0 2 3 5 8 11 11

1 1 2 3 3 4 4 4 5 5 5 7

Counting Sort: Het algoritme

```
(define (counting-sort in out max-key key)
  (let ((count (make-vector max-key 0))
        (size (vector-length in)))
    (define (fill-count-vector i)
      (let ((k (key (vector-ref in i))))
        (vector-set! count
                     k (+ (vector-ref count k) 1))
        (if (< (+ i 1) size)
            (fill-count-vector (+ i 1)))))
    (define (sum-vector i)
      (vector-set! count
                   i
                   (+ (vector-ref count (- i 1)) (vector-ref count i)))
      (if (< (+ i 1) max-key)
          (sum-vector (+ i 1))
          count))
    (define (spread-out-again i)
      ...)
    (fill-count-vector 0)
    (sum-vector 1)
    (spread-out-again (- size 1))))
```

```
(define (spread-out-again i)
  (let* ((data (vector-ref in i))
         (k (- (vector-ref count (key data)) 1)))
    (vector-set! out k data)
    (vector-set! count (key data) k)
    (if (<= i 0)
        out
        (spread-out-again (- i 1)))))
```

Counting Sort: Eigenschappen

Het tellen duurt $O(n)$

De sommen berekenen duurt $O(k)$

Het spreiden duurt opnieuw $O(n)$

Alles samen
 $O(n + k)$

Counting Sort
is stabiel

Spreiden van
rechts naar links

Counting Sort is
niet in-place

Hoofdstuk 5

5.1 Terminologie

5.2 Simpele Sorteeralgoritmen

5.2.1 Bubble Sort

5.2.2 Insertion Sort

5.2.3 Selection Sort

5.2.4 Vergelijking

5.3 Geavanceerde Sorteeralgoritmen

5.3.1 Quicksort

5.3.2 Mergesort

5.3.3 Heapsort

5.4 Beperkingen van Vergelijkend Sorteren

5.5 Vergelijking van Vergelijkend Sorteren

5.6 Lineair Sorteren

5.6.1 Radix Sort

5.6.2 Bucket Sort

5.6.3 Counting Sort

