

datastructuren en algoritmen

2011 09 22

college 6

totnutoe

- **lineaire datastructuren**
stacks, queues, vectoren, lijsten, rijtjes, priority queues
- **hierarchische datastructuren**
bomen, binaire bomen, binaire zoekbomen, AVL-bomen, heaps
- **algoritmen**
sorteren, heap-sort, selection sort, insertion sort
- **hashing**

schema

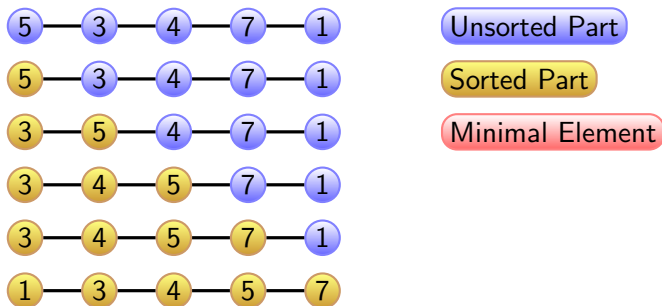
- recap
- insertion sort
- selection sort
- bubble sort
- merge sort
- quick sort
- materiaal

sorteren: specificatie

- **input:**
een eindig rijtje getallen
- **output:**
een geordende permutatie van de input

insertion sort

- het rijtje bestaat uit een gesorteerd deel en een niet-gesorteerd deel
- neem het eerste item uit het niet-gesorteerde deel en voeg het op de juiste plek toe aan het gesorteerde deel



insertion sort: via priority queue

(al gezien:)

insertion sort kan gezien worden als instantie van priority queue sort

met priority queue geïmplementeerd met gesorteerde lijst

insertion sort: pseudo-code

input: array A ter lengte n

output: A (oplopend) gesorteerd

Algorithm insertionSort(A, n):

for $j := 1$ **to** $n - 1$ **do**

$key := A[j]$

$i := j - 1$

while $i \geq 0$ **and** $A[i] > key$ **do**

$A[i + 1] := A[i]$

$i := i - 1$

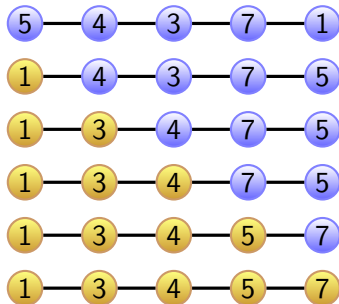
$A[i + 1] := key$

insertion sort: analyse

- **correctheid:**
via **invariant:**
aan het begin van de for-loop is het sub-array $A[0 \dots j - 1]$ een gesorteerde permutatie van het sub-array $A[0 \dots j - 1]$ van het input-array
- **tijdscomplexiteit**
via **sommatie** in $\mathcal{O}(n^2)$

selection sort

- zoek een kleinste key in het rijtje en verwissel dat item met het eerste item
- itereer op de rest van het rijtje



Unsorted Part

Sorted Part

Minimal Element

selection sort: via priority queue

(al gezien:)

selection sort kan gezien worden als instantie van priority queue sort

met priority queue geïmplementeerd met ongesorteerde lijst

selection sort: pseudocode

Algorithm selectionSort(A, n):

for $i := 0$ **to** $n - 2$ **do**

$m := i$

for $j = i$ **to** $n - 1$ **do**

if $A[j] < A[m]$ **then** $m := j$

$x := A[m]$

$A[m] := A[i]$

$A[i] := x$

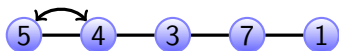
selection sort: analyse

- **correctheid:** via invariant
- **worst-case tijdscomplexiteit:** via sommatie

schema

- recap
- insertion sort
- selection sort
- **bubble sort**
- merge sort
- quick sort
- materiaal

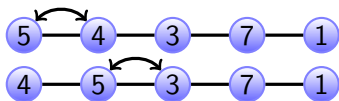
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

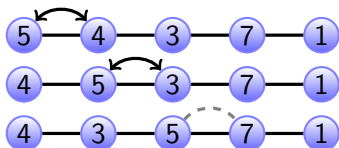
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

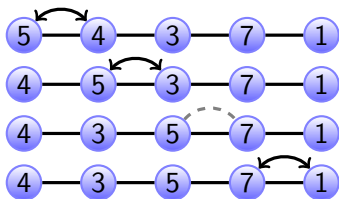
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

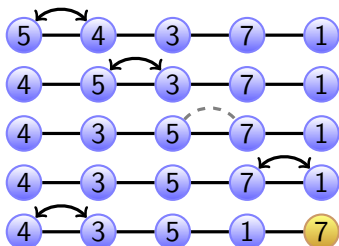
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

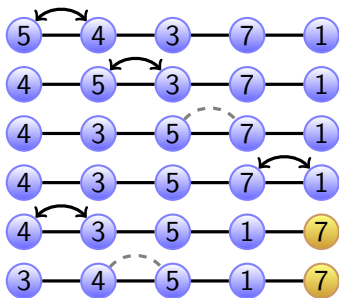
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

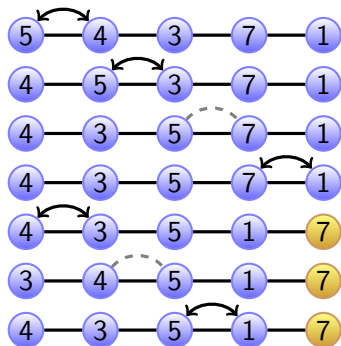
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

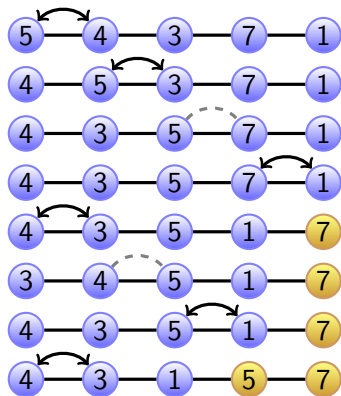
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

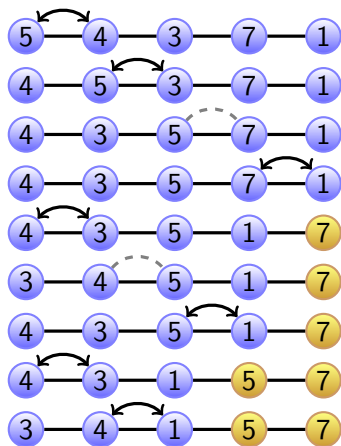
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

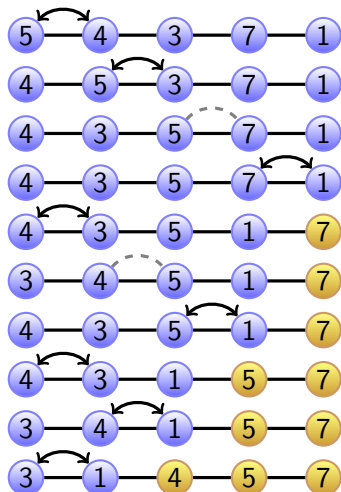
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

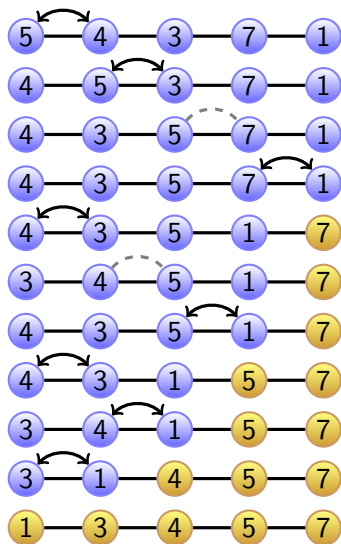
bubble-sort: voorbeeld



Unsorted Part

Sorted Part

bubble-sort: voorbeeld



Unsorted Part

Sorted Part

bubble-sort: algoritme

verwissel buren die in verkeerde volgorde staan

input: array A terlengte l

Algorithm bubbleSort(A, l):

$n := l$

$swapped := \mathbf{true}$

while $swapped$ **do**

$swapped := \mathbf{false}$

for $i := 0$ **to** $n - 2$ **do**

if $A[i] > A[i + 1]$ **then**

$\text{swap}(A[i], A[i + 1])$

$swapped := \mathbf{true}$

$n := n - 1$

bubble sort: analyse

- correctheid:
- worst-case tijdscomplexiteit

schema

- recap
- insertion sort
- selection sort
- bubble sort
- **merge sort**
- quick sort
- materiaal

merge sort: idee

- **probleem:** sorteer rij getallen
- **algoritme:** sorteer links, sorteer rechts, merge twee gesorteerde lijsten
- **bedenker:** John von Neumann 1945



merge: idee

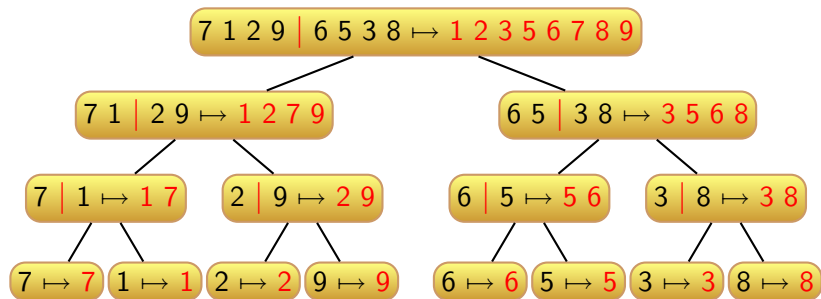
input: twee gesorteerde rijtjes

output: de gesorteerde merge van de twee input-rijtjes

idee algoritme: voeg achteraan in de outputrij S
het kleinste element van de inputrijtjes S_1 en S_2 toe

het werk van merge sort zit in deze stap

merge sort: voorbeeld



elke knoop representeert een recursive call

merge sort: algoritme

Algorithm `mergeSort(S)`:

Input: een rijtje S van n elementen

Output: het rijtje S gesorteerd

if `size(S) > 1` **then**

$(S_1, S_2) = \text{partition } S \text{ into size } \lfloor n/2 \rfloor \text{ and } \lceil n/2 \rceil$

`mergeSort(S1)`

`mergeSort(S2)`

$S = \text{merge}(S_1, S_2)$

merge: algoritme

Algorithm `merge(A, B)`:

Input: gesorteerde rijtjes A , B

Output: gesorteerd rijtje met precies items uit A en B

$S =$ empty list

while $\neg A.isEmpty()$ and $\neg B.isEmpty()$ **do**

if $A.first().element < B.first().element$ **then**

$S.insertLast(A.remove(A.first()))$

else

$S.insertLast(B.remove(B.first()))$

done

while $\neg A.isEmpty()$ **do** $S.insertLast(A.remove(A.first()))$

while $\neg B.isEmpty()$ **do** $S.insertLast(B.remove(B.first()))$

return S

we gebruiken methoden uit het ADT voor lijsten

merge: tijdscomplexiteit

aannames:

- A bevat p items en B bevat q items
- rijtjes met:
vinden, toevoegen, verwijderen vooraan of achteraan in $\mathcal{O}(1)$
(circular array of singly/doubly linked list)

dan:

- dan: merge in $\mathcal{O}(p + q)$
want operaties in de loop in $\mathcal{O}(1)$
totaal aantal iteraties $p + q$. dus running time in $\mathcal{O}(p + q)$.

merge sort: complexiteit via boom-analyse

- input rijtje van n elementen met n een twee-macht
- hoogte van merge sort boom is $\mathcal{O}(\log n)$ eigenlijk $\lceil \log(n) \rceil$
dus $\lceil \log(n) \rceil + 1$ laagjes met op elk laagje $\mathcal{O}(n)$
dus als comparison in $\mathcal{O}(1)$ dan merge sort in $\mathcal{O}(n \log n)$
- per laagje: werk = merge = aantal items
- totaal: $\mathcal{O}(n \cdot \log n)$

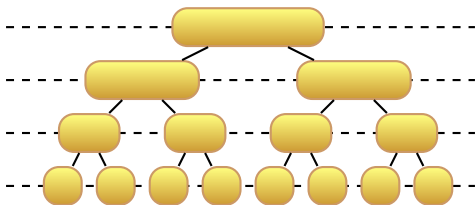
depth	nodes	size
-------	-------	------

0	1	n
---	---	-----

1	2	$n/2$
---	---	-------

i	2^i	$n/2^i$
-----	-------	---------

...
-----	-----	-----



merge: complexiteit via recurrente betrekking

functie $T(n)$ voor de worst-case running time van merge sort met recurrente betrekking:

$$T(n) = \begin{cases} 1 & \text{als } n = 1 \\ 2T(\frac{n}{2}) + n & \text{als } n > 1 \end{cases}$$

levert $T(n) \in \mathcal{O}(n \log(n))$

verdeel en heers



merge sort is een voorbeeld van een verdeel en heers algoritme

- **divide:** verdeel probleem S in deelproblemen
- **recur:** los met recursie deelproblemen op
- **conquer:** voeg oplossingen van deelproblemen samen tot oplossing van S

verdeel en heers: merge sort



merge sort is een voorbeeld van een verdeel en heers algoritme

- **divide:** verdeel probleem S in deelproblemen
split rijtje S in twee rijtjes S_1 en S_2
- **recur:** los met recursie deelproblemen op
sorteer S_1 en S_2
- **conquer:** voeg oplossingen van deelproblemen samen
tot oplossing van S
merge gesorteerde S_1 en S_2 tot gesorteerde S

verdeel en heers: voorbeeld tiling

- **probleem:** betegel een $2^k \times 2^k$ vierkant met L -blokjes laat precies één vooraf gekozen hokje vrij
- **idee algoritme:** verdeel en heers: betegel de vier kwarten
- **correctheid:** met inductie
- **complexiteit:** met recurrenente betrekking

$$T(n) = \begin{cases} 1 & \text{als } n = 1 \\ 4T(\frac{n}{2}) & \text{als } n > 1 \end{cases}$$

verdeel en heers: voorbeeld binary search

ook binary search is een voorbeeld van een verdeel en heers algoritme

schema

- recap
- insertion sort
- selection sort
- bubble sort
- merge sort
- quick sort
 - algoritme zoals in het boek
 - in-place algoritme zoals oa in probleemoplossen
- materiaal

quicksort

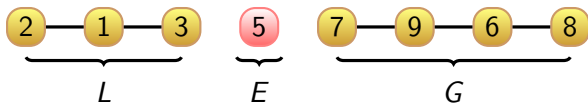
- **probleem:** sorteer rijtje
- **algoritme:**
 - kies pivot en splits de rij in drie stukken:
 - kleiner dan pivot
 - gelijk aan pivot
 - groter dan pivot
 - sorteer delen en voeg samen
- **het werk zit m in de partitie**

quick-sort: verdeel en heers

- divide:** verdeel de lijst in L ($<$ pivot), E ($=$ pivot), G ($>$ pivot)



- recur:** sorteer (recursieve aanroep) L en G



- conquer:** voeg de gesorteerde L en E en G samen



quick-sort: partitie

Algorithm `partition(S, p)`:

Input: rijtje S van n elementen, positie p van de pivot

Output: partitie van S in L , E , G kleiner, gelijk, groter pivot

$L, E, G =$ empty lists

$x = S.\text{elementAtRank}(p)$

while $\neg S.\text{isEmpty}()$ **do**

$y = S.\text{remove}(S.\text{first}())$

if $y < x$ **then** $L.\text{insertLast}(y)$

if $y == x$ **then** $E.\text{insertLast}(y)$

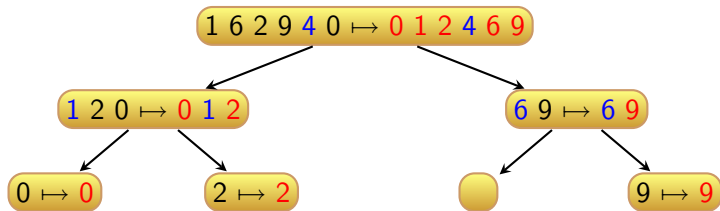
if $y > x$ **then** $G.\text{insertLast}(y)$

done

return L, E, G

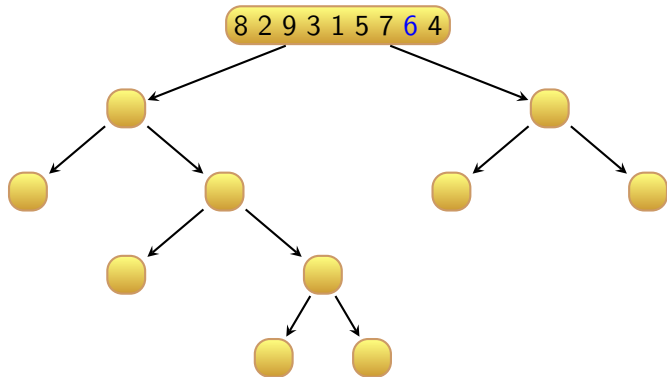
- **aanname:** toevoegen en verwijderen in $\mathcal{O}(1)$
- **opmerking:** we moeten elke key met de pivot vergelijken
- **dan:** totaal in $\mathcal{O}(n)$

quick-sort: boom

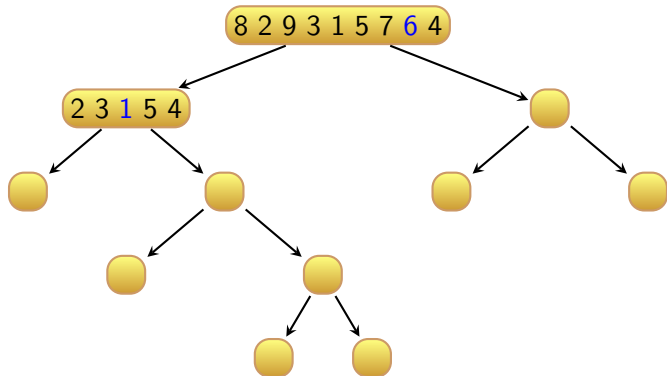


- elke knoop representeert recursieve aanroep
- bladeren: sorteren van rijtje lengte 0 of 1

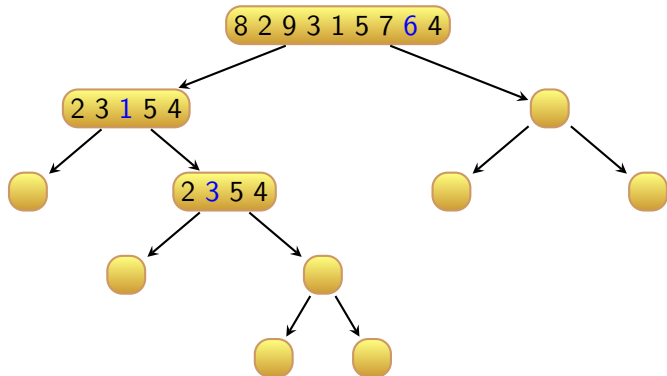
quick-sort: voorbeeld



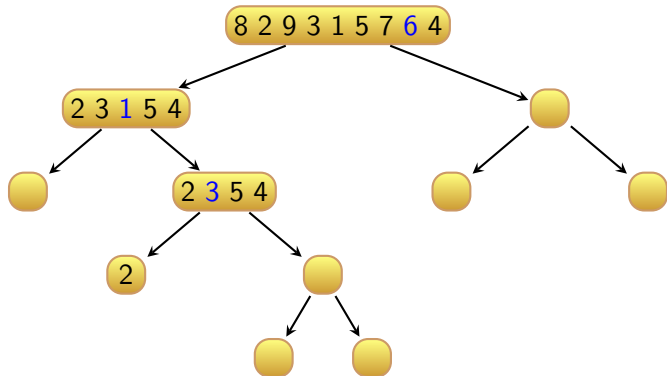
quick-sort: voorbeeld



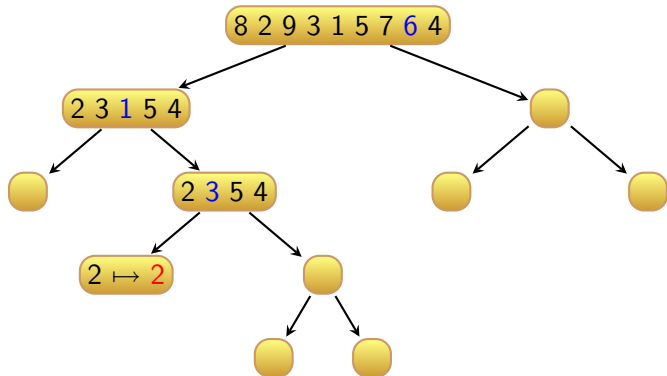
quick-sort: voorbeeld



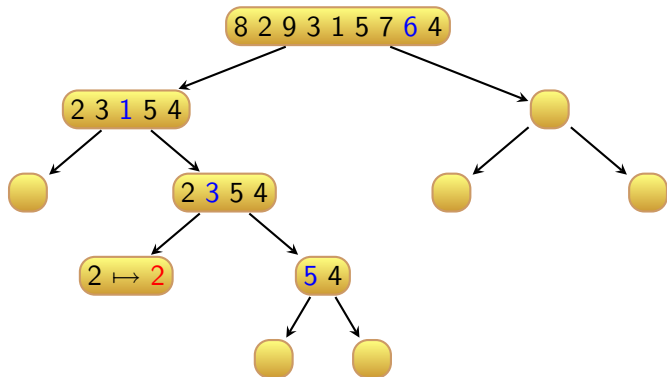
quick-sort: voorbeeld



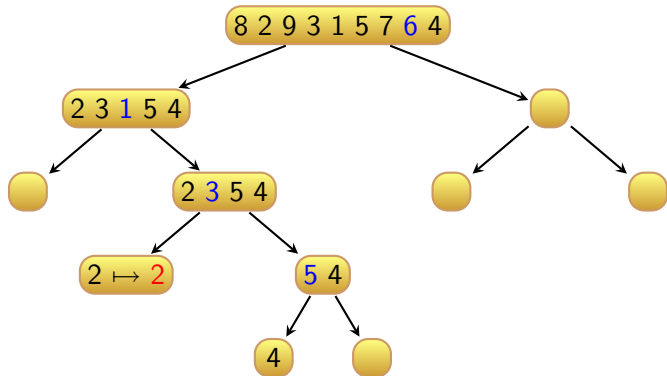
quick-sort: voorbeeld



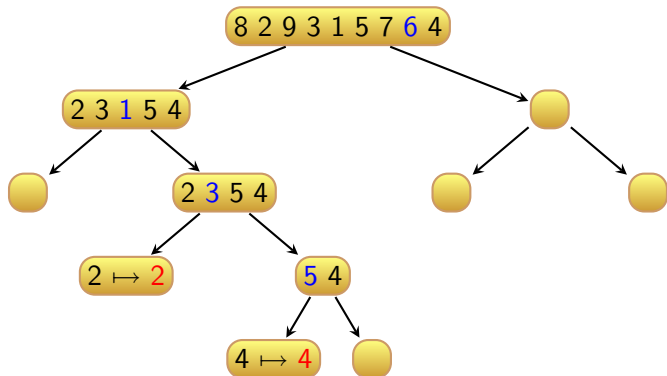
quick-sort: voorbeeld



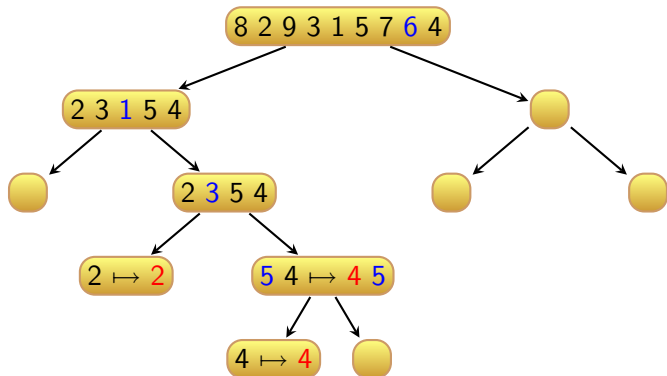
quick-sort: voorbeeld



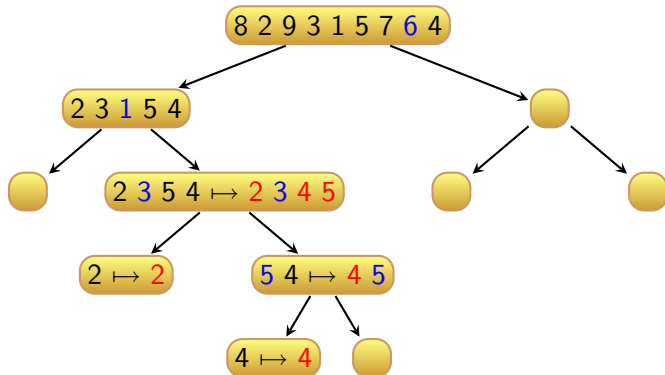
quick-sort: voorbeeld



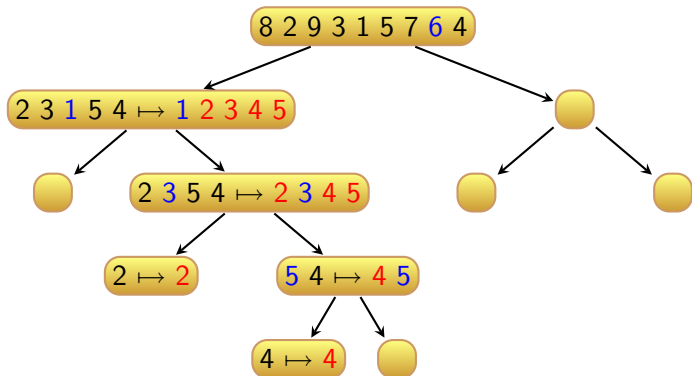
quick-sort: voorbeeld



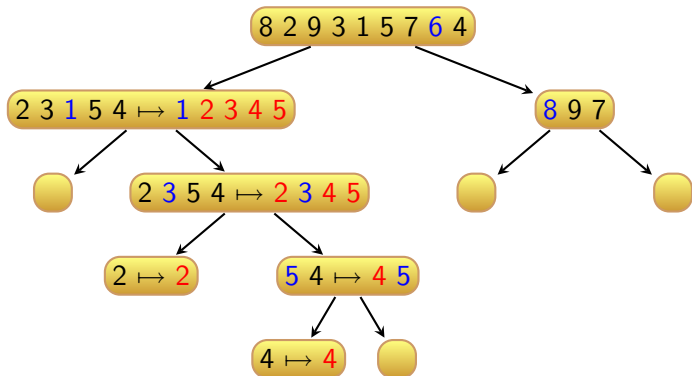
quick-sort: voorbeeld



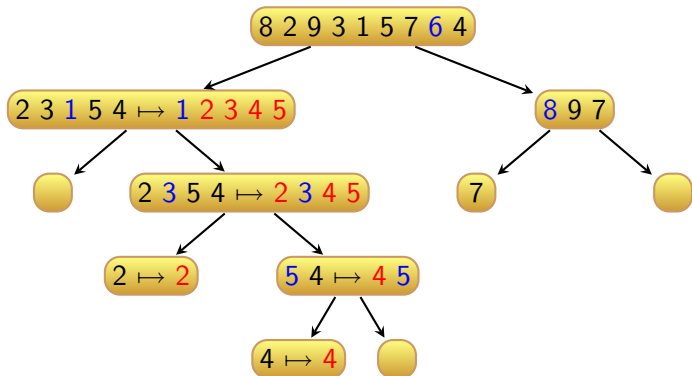
quick-sort: voorbeeld



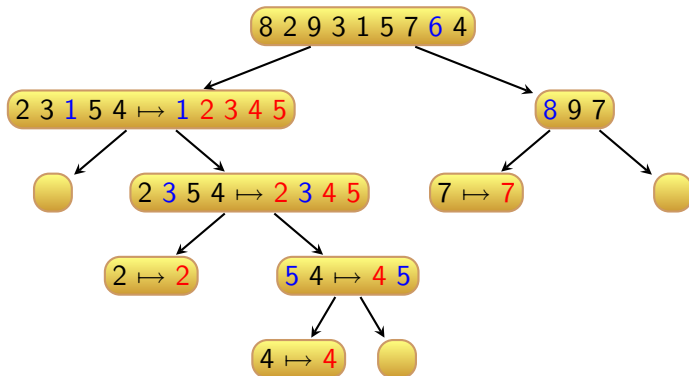
quick-sort: voorbeeld



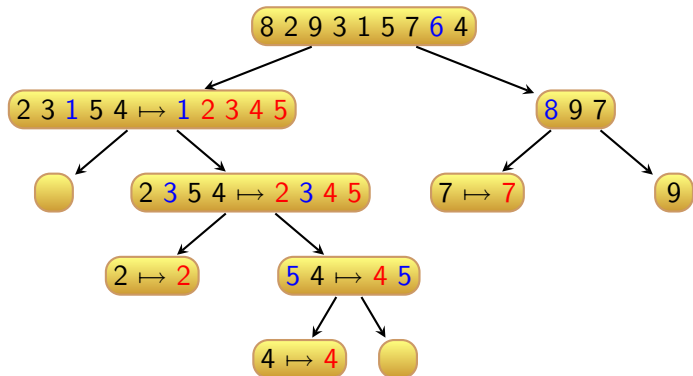
quick-sort: voorbeeld



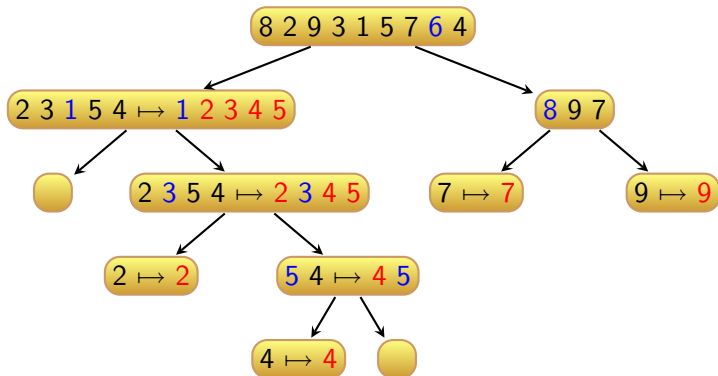
quick-sort: voorbeeld



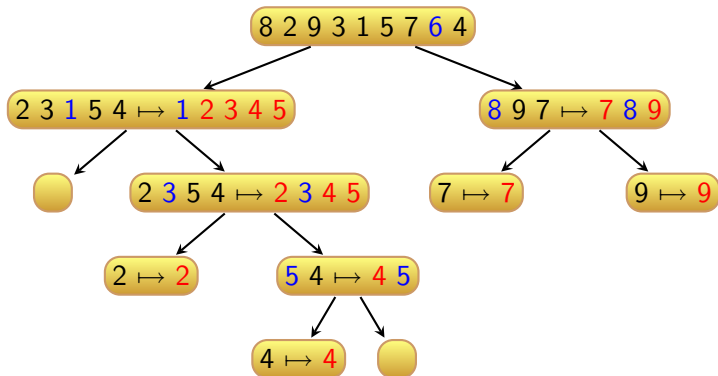
quick-sort: voorbeeld



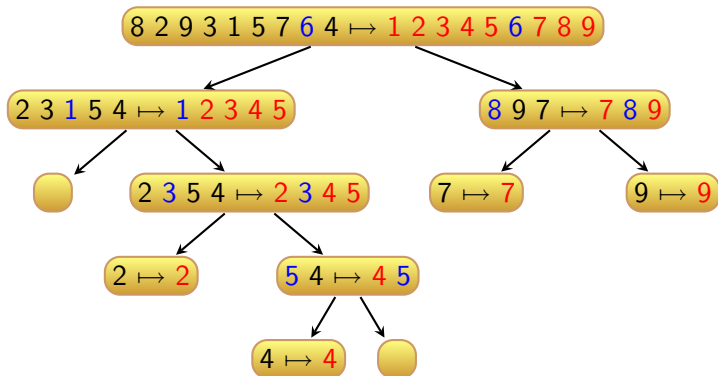
quick-sort: voorbeeld



quick-sort: voorbeeld

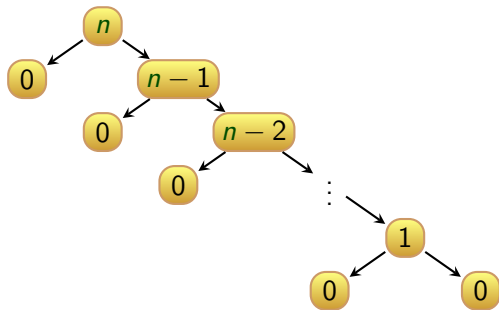


quick-sort: voorbeeld



quick-sort: worst-case tijdscomplexiteit

ongunstigst als pivot grootste of kleinste key
dan: running time $n + (n - 1) + \dots + 1 \in \mathcal{O}(n^2)$
quick??

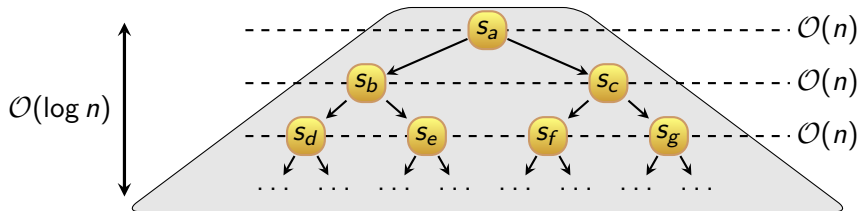


randomized quick-sort

- pivot is random key uit input rijtje
- pivot GOED als L en G alletwee $< \frac{3}{4}$ van input-grootte
- pivot SLECHT als L of $G \geq \frac{3}{4}$ van input-grootte
- kans op goede pivot: $\frac{1}{2}$
- wat zijn de goede en slechte pivots in $1, \dots, 16$?

randomized quick-sort: verwachte running time

- verwachting voor knoop op diepte i : $\frac{i}{2}$ voorgangers zijn goede calls
- dus lengte rijtje op diepte i is $\leq \left(\frac{3}{4}\right)^{\frac{i}{2}} \cdot n$
- lengte 1 voor $i = 2 \log_{\frac{4}{3}} n$, dus hoogte in $\mathcal{O}(\log n)$
- werk per laagje in $\mathcal{O}(n)$



quick-sort: in place

Algorithm `inPlaceQuickSort(A, l, r)`:

Input: list A , indices l and r

Output: list A where elements from index l to r are sorted

if $l \geq r$ **then return**

$p = A[r]$ (take rightmost element as pivot)

$l' = l$ and $r' = r$

while $l' \leq r'$ **do**

while $l' \leq r'$ and $A[l'] \leq p$ **do** $l' = l' + 1$ (find $> p$)

while $l' \leq r'$ and $A[r'] \geq p$ **do** $r' = r' - 1$ (find $< p$)

if $l' < r'$ **then** `swap(A[l'], A[r'])` (swap $< p$ with $> p$)

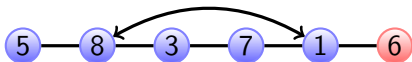
done

`swap(A[r], A[l'])` (put pivot into the right place)

`inPlaceQuickSort(A, l, l' - 1)` (sort left part)

`inPlaceQuickSort(A, l' + 1, r)` (sort right part)

in-place quick-sort: voorbeeld

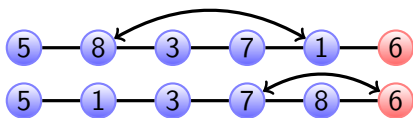


Unsorted Part

Sorted Part

Pivot

in-place quick-sort: voorbeeld

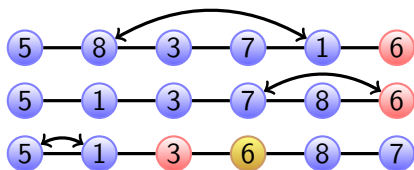


Unsorted Part

Sorted Part

Pivot

in-place quick-sort: voorbeeld

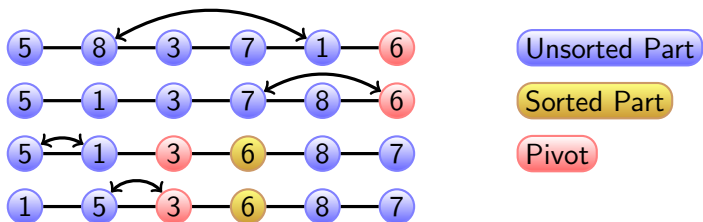


Unsorted Part

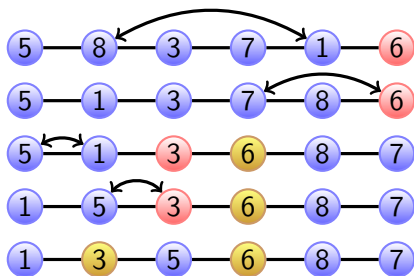
Sorted Part

Pivot

in-place quick-sort: voorbeeld



in-place quick-sort: voorbeeld

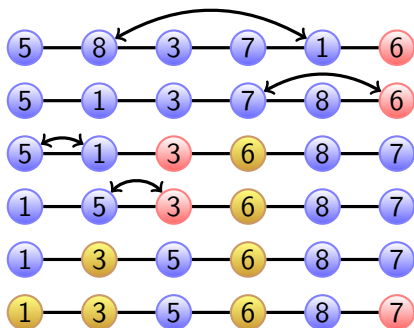


Unsorted Part

Sorted Part

Pivot

in-place quick-sort: voorbeeld

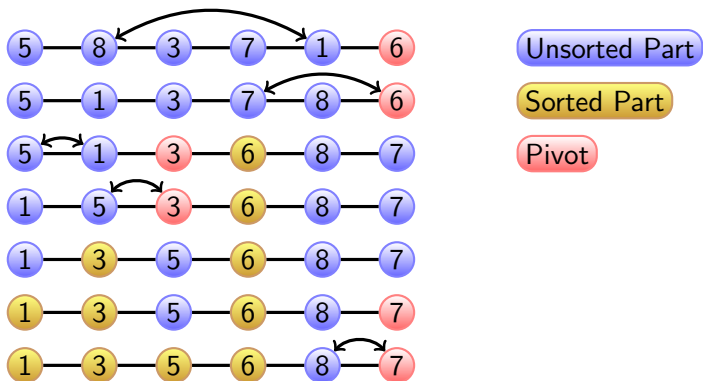


Unsorted Part

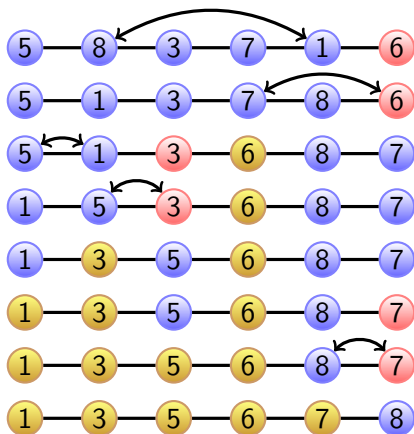
Sorted Part

Pivot

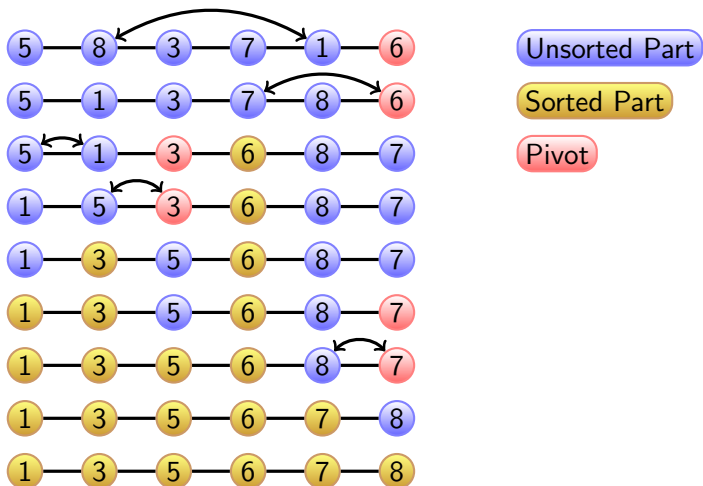
in-place quick-sort: voorbeeld



in-place quick-sort: voorbeeld



in-place quick-sort: voorbeeld



schema

- recap
- insertion sort
- selection sort
- bubble sort
- merge sort
- quick sort
 - algoritme zoals in het boek
 - in-place algoritme zoals oa in probleemplossen
- **materiaal**

materiaal

- boek 4.1, 4.3

extra materiaal

- John von Neumann
- bedenker van quicksort: Tony Hoare
- verdeel en heers
- loop invariant