

# Sorteeralgoritmen

Bachelor IT

Sven Mariën

(sven.marien01@ap.be)

# Wat is en waarom ?

- Wat is ?

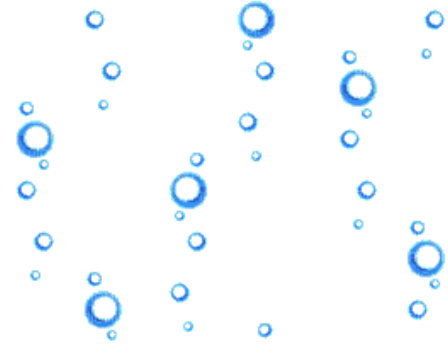
Een **sorteer algoritme** is een **algoritme** om elementen van een **lijst** in een bepaalde volgorde te zetten. In de geschiedenis van het **programmeren** zijn vele algoritmen voor deze taak bedacht die zich onderscheiden door verschillende snelheid, geheugengebruik en gedrag bij toename van het aantal te sorteren elementen. Het sorteren van bijvoorbeeld een pak **speelkaarten** stelt andere eisen dan het sorteren van het **telefoonboek** van **New York**.

- Waarom nodig ?

- We willen een gesorteerde lijst op het scherm,..
- We willen de mediaan vinden van een lijst getallen
- Sommige **zoek**algoritmen (zie later) verwachten dat de elementen reeds gesorteerd zijn.
- ...

# Bubble sort

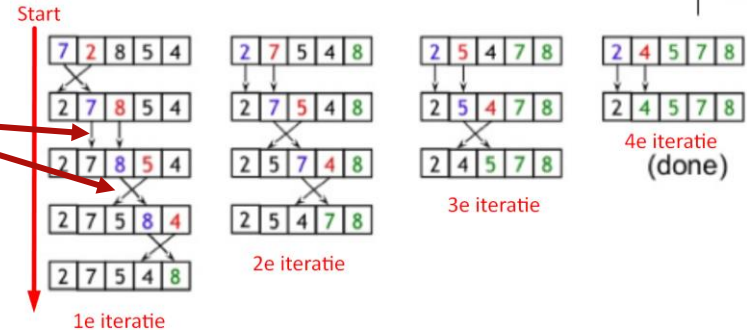
1. Loop door de te sorteren rij van n elementen
  - a. Vergelijk daarbij elk element met het volgende.
  - b. Verwissel beide als ze in de verkeerde volgorde staan.
  - c. Schuif dan een stapje op tot het einde is bereikt.
2. Loop opnieuw door de rij, maar ga nu door tot het voorlaatste element, omdat het laatste element reeds het grootste in de rij was.
3. Nog een keer, maar negeer dan de twee laatste elementen, enz...
4. Ga zo door tot de hele reeks gesorteerd is



5 2 4 6 1 3

# Bubble sort (2)

- Indien  $n$  = aantal elementen
- Dan zijn er  $n-1$  iteraties nodig
- Elke iteratie omvat  $n-i$  “compare and swap” bewerkingen
- Zijn er nog optimalisatie(s) mogelijk ?
  - => “**adaptief**” algoritme: stopt als de lijst reeds gesorteerd is



5 2 4 6 1 3

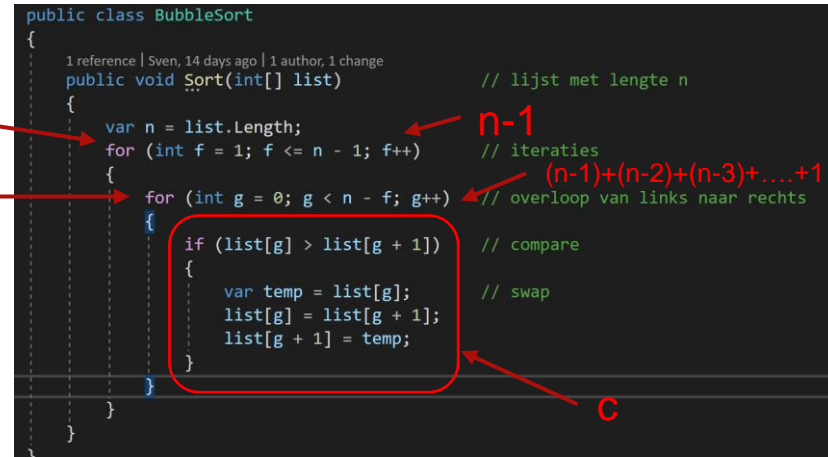
# Bubble sort

- Omvormen naar c# ... ?


# Bubble sort (3)

- Time complexity bepalen:
- Input = n elementen
  - Buitenste lus: n-1 iteraties
  - Binnenste lus : n/2 iteraties
  - Compare and swap : tijd “c”
- Totaal:  $(n-1) * n/2 * c$ 
  - $= c.n^2/2 - c.n/2$
  - $\Rightarrow O(n^2)$

```
public class BubbleSort
{
    1 reference | Sven, 14 days ago | 1 author, 1 change
    public void Sort(int[] list) // lijst met lengte n
    {
        var n = list.Length;
        for (int f = 1; f <= n - 1; f++) // iteraties n-1
        {
            for (int g = 0; g < n - f; g++) // (n-1)+(n-2)+(n-3)+....+1
            {
                if (list[g] > list[g + 1]) // overloop van links naar rechts // compare
                {
                    var temp = list[g]; // swap
                    list[g] = list[g + 1];
                    list[g + 1] = temp;
                }
            }
        }
    }
}
```



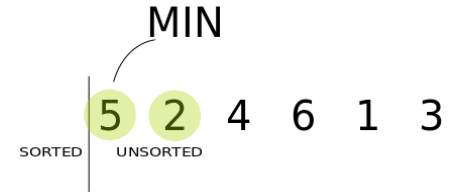
# Bubble sort (3)

- We zien de sortering gebeuren van achter naar voor 
- Algoritme is niet (tijds) efficiënt
- Werkt wel snel (na optimalisatie) als de lijst reeds bijna gesorteerd is.
- Geheugen verbruik is constant.
- Wordt veelal gebruikt als demonstratie voor sorteeralgoritmes

Time complexity	Space Complexity
$O(n^2)$	$O(1)$

# Selection sort

1. Zoek eerst de kleinste waarde in de lijst.
2. Verwissel het met de “eerste” waarde in de lijst. De eerste waarde staat nu op zijn plaats.
3. Herhaal de bovenstaande stappen met de rest van de lijst.



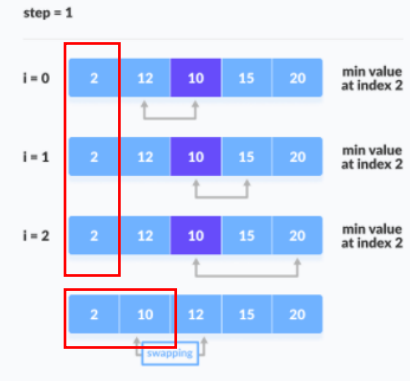


# Selection sort (2)

Start



1<sup>e</sup> iteratie



2<sup>e</sup> iteratie



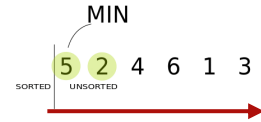
3<sup>e</sup> iteratie



4<sup>e</sup> iteratie  
(Einde)

- Indien  $n$  = aantal elementen
- Dan zijn er  $n-1$  iteraties nodig
- Elke iteratie wordt 1 korter, aangezien telkens het eerste element op de juiste plaats komt te staan
- “Worst case”: Elke iteratie bevat  $n-i$  compares en 1 swap

# Selection sort (3)



- De sortering gebeurt hier van voor naar achter
- Algoritme is niet heel tijdsefficient
- Wel sneller dan “bubble sort” indien de lijst totaal niet gesorteerd is.
  - Selection sort = max.1 swap per iteratie
- = **Niet adaptief** algoritme (houdt geen rekening met een gesorteerde lijst)
- Constant / laag geheugen verbruik (zie ook oefeningen)
  - Interessant voor systemen met weinig beschikbaar geheugen

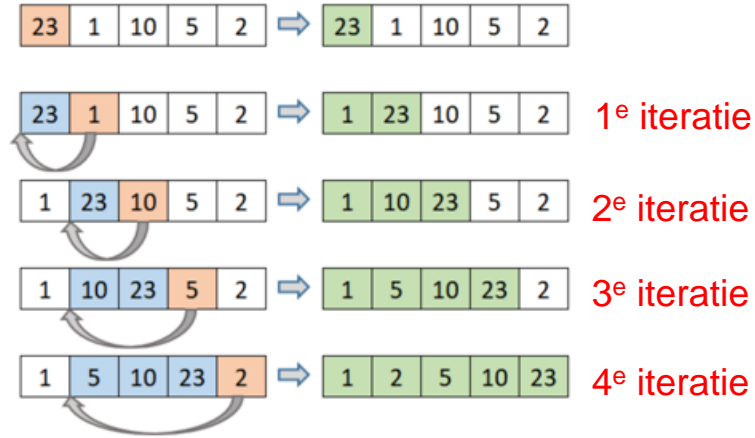
Time complexity	Space Complexity
$O(n^2)$	$O(1)$

# Insertion Sort



- Werkt zoals het sorteren van een kaartspel in je hand
1. Zet de 2<sup>de</sup> kaart tov. kaart 1 op de juiste plaats.
  2. Kijk naar de volgende kaart en zet deze in de reeds gesorteerde rij links op de juiste plaats
  3. Doe dit tot het einde van de reeks

# Insertion sort (2)



- Indien  $n$  = aantal elementen
- Dan zijn er  $n-1$  iteraties nodig
- Elke iteratie wordt langer aangezien er meer compare/swap operaties nodig zijn, per iteratie zijn er max.  $i$  compare en swap operaties nodig

# Insertion sort (3)

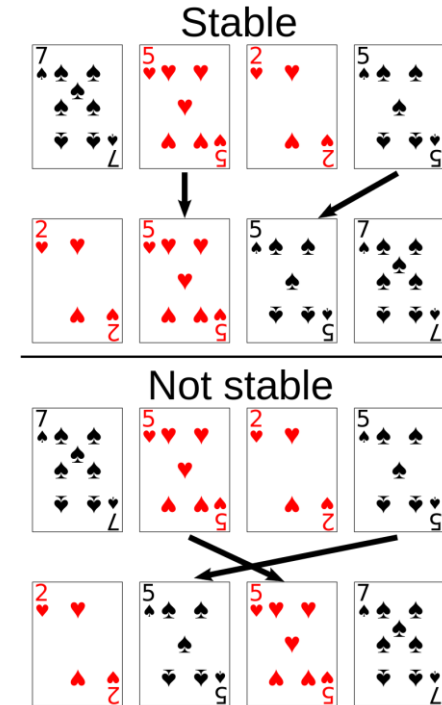


- De sortering gebeurt hier eveneens van vooraan naar achteraan de lijst
- Is “**adaptief**”, zal bij een reeds gesorteerde lijst minder tijd nodig hebben (beste geval  $\Rightarrow O(n)$ )
- Is een “**stabiel**” algoritme (zie verder)
- Constant & laag geheugenverbruik

Time complexity	Space Complexity
$O(n^2)$	$O(1)$

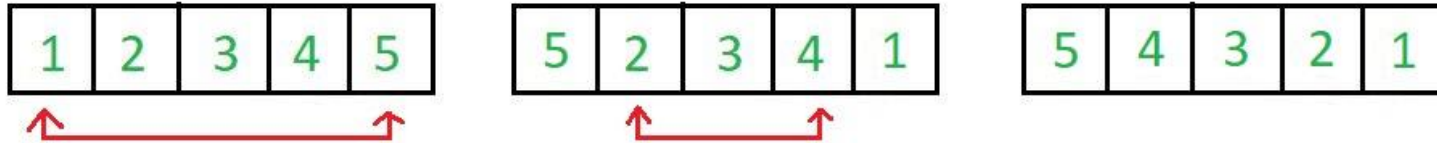
# “Stabiele” algoritmen

- Bij een “**stabiel**” algoritme blijven gegevens met eenzelfde waarde in dezelfde volgorde staan nadat de lijst gesorteerd werd.
- Bij een “**niet-stabiel**” algoritme wordt er hiermee geen rekening gehouden en zou het dus kunnen dat de volgorde na sortering niet meer dezelfde is.

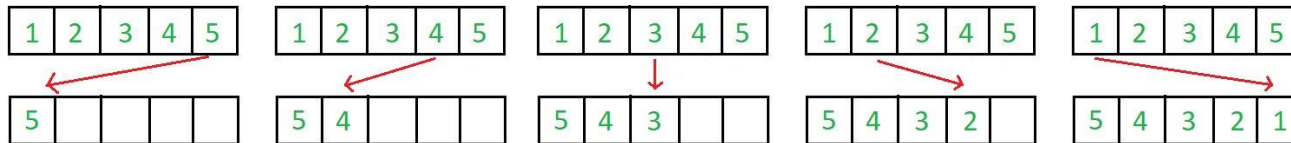


# “In place” algoritmen

- Er is geen extra geheugen nodig om te sorteren
  - Voorbeeld: omkeren van een array



- Not in place: extra geheugen gebruiken
  - Voorbeeld: ... omkeren van een array



# Quick sort

- “**Divide and conquer**” algoritme (verdeel en heers)
  - Het probleem opdelen in 1 of meerdere deelproblemen van hetzelfde type, tot ze eenvoudig genoeg zijn.
- 1. Pivot selectie: Selecteer een element uit de reeks. Dit wordt het **pivot** element.
- 2. Partioneren: Alle elementen **kleiner dan de pivot** worden aan de **ene zijde** bij elkaar geplaatst, **groter dan de pivot** aan de **andere zijde**.
- 3. Herhaal dit recursief

Unsorted Array





Say we have

8	1	6	9	6	3	5	2	7	0
---	---	---	---	---	---	---	---	---	---

And 6 is chosen as the pivot.

1. Swap pivot with the last element, we get

8	1	6	9	0	3	5	2	7	6
---	---	---	---	---	---	---	---	---	---

2	1	6	9	0	3	5	8	7	6
---	---	---	---	---	---	---	---	---	---

2	1	6	9	0	3	5	8	7	6
---	---	---	---	---	---	---	---	---	---

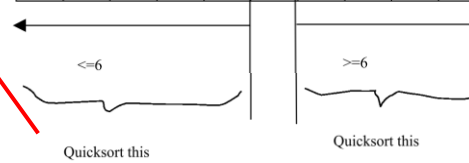
2	1	5	9	0	3	6	8	7	6
---	---	---	---	---	---	---	---	---	---

2	1	5	3	0	9	6	8	7	6
---	---	---	---	---	---	---	---	---	---

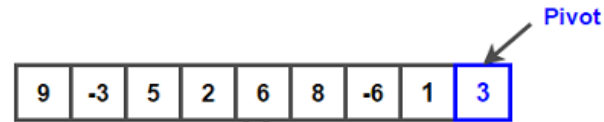


Now swap i with pivot, we get

2	1	5	3	0	6	6	8	7	9
---	---	---	---	---	---	---	---	---	---



# Quick sort (2)



# Quicksort

- **Pivot selectie** kan op verschillende manieren, neem steeds:
  - het eerste element
  - het laatste element
  - het middelste element
  - een **willekeurig** element
  - de **mediaan**
- Een goede keuze van pivot kan het algoritme versnellen.

# Quicksort

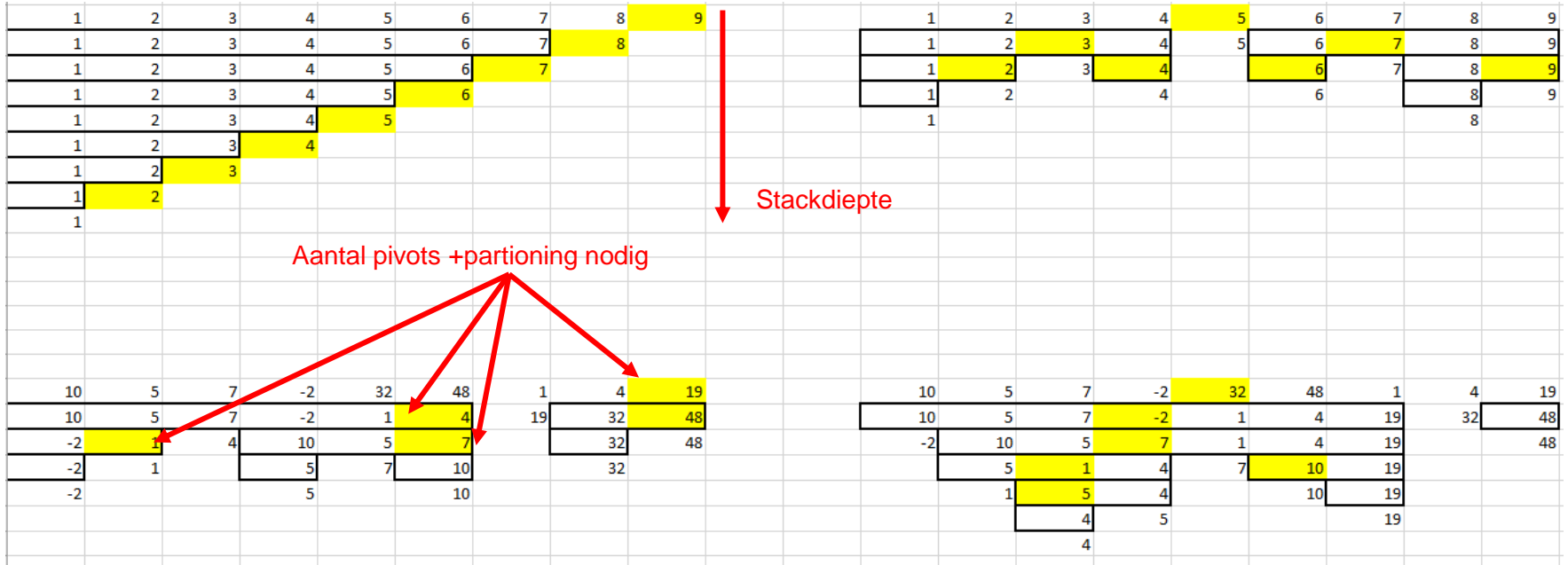
- **Partitioneren** / opdelen in 2 groepen (kleiner en groter dan pivot)
  - Kan op verschillende manieren gebeuren
    - Van buiten naar binnen
    - Van links naar rechts
    - ...
  - Zorgt ervoor dat de pivot zelf reeds op de juiste plaats komt te staan.

# Quicksort

- Time complexity:
  - bij een slechte pivot selectie kan het voorkomen dat de volledige rij naar de andere zijde van de pivot moet worden verschoven of dat er steeds 1 partitie leeg is.
- Space complexity:
  - We hebben **geen extra variabelen** nodig aangezien we telkens 2 waarden omwisselen van plaats in de array. Het is dus een “**in place**” algoritme
  - We doen echter wel recursieve calls, in het slechte geval zijn dat er **n**, dus het geheugen verbruik hiervoor zal lineair oplopen
- Het is **geen “stabiel” algoritme**, de relatieve plaats van gelijkwaardige elementen blijft niet behouden.
- Alhoewel de “worst case” complexity nog steeds  $O(n^2)$  bedraagt wordt Quicksort toch aanzien als een snel algoritme, gemiddeld gezien scoort het immers veel beter

Time complexity	Space Complexity
$O(n^2)$	$O(n)$

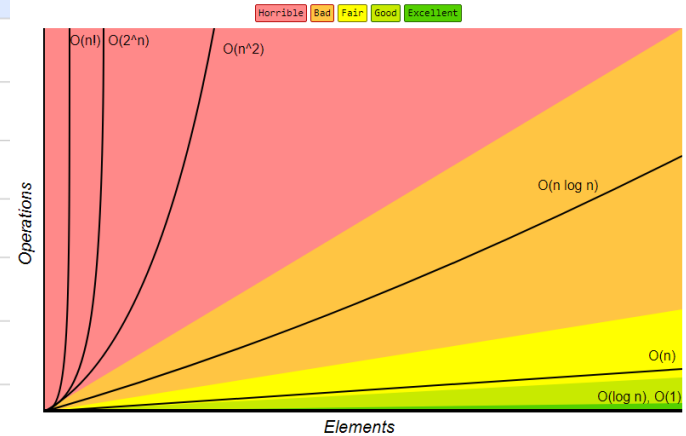
# Kiezen van een “goede” pivot



Meer info: <https://en.wikipedia.org/wiki/Quicksort>

# Tussentijdse vergelijking

Name	Best	Average	Worst	Memory
Quicksort	$n \log n$	$n \log n$	$n^2$	$\log n$ (average)
Mergesort	$n \log n$	$n \log n$	$n \log n$	$n$ (worst case)
Heapsort	$n \log n$	$n \log n$	$n \log n$	1
Timsort	$n$	$n \log n$	$n \log n$	$n$
Bubble sort	$n$	$n^2$	$n^2$	1
Selection sort	$n^2$	$n^2$	$n^2$	1
Insertion sort	$n$	$n^2$	$n^2$	1



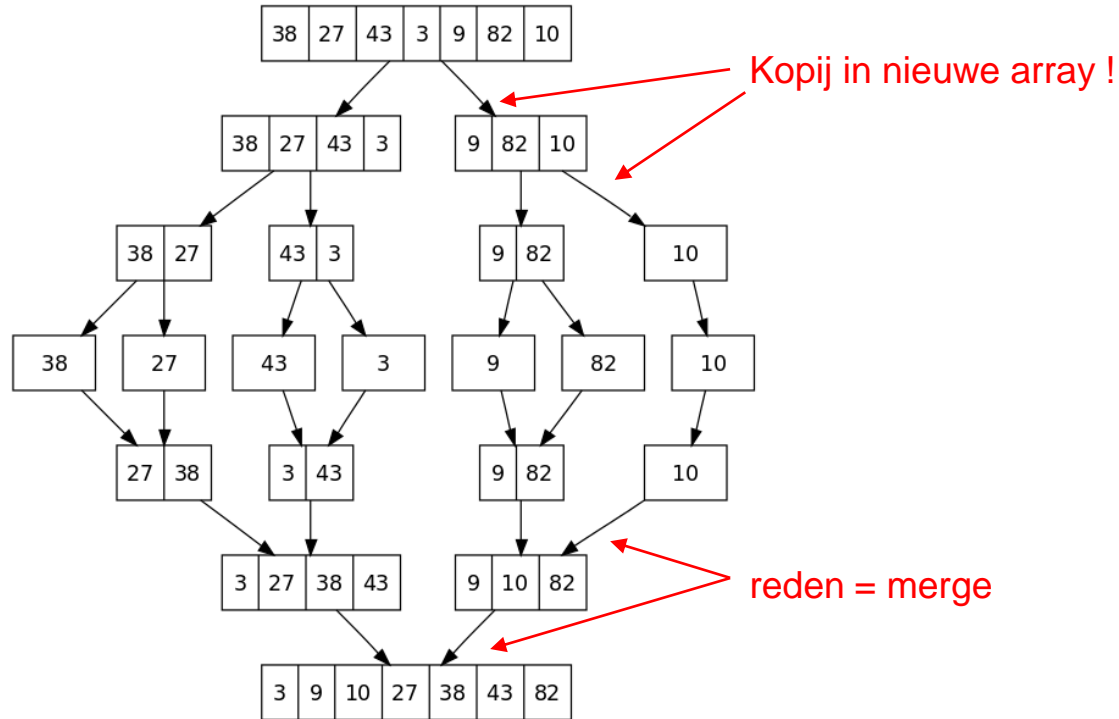
# Merge sort

1. Deel de reeks op in 2 groepen
2. Herhaal dit tot je groepjes van max. 2 elementen overhoudt
3. Sorteer elk groepje
4. Combineer 2 groepjes terug en sorteer ze tegelijkertijd
5. Herhaal dit tot de volledige reeks terug is opgebouwd.

6 5 3 1 8 7 2 4



# Merge sort (2)



# Merge Sort (3)

- Stabiel algoritme
- Niet “in place” algoritme
- Extra geheugen nodig  $O(n)$

Time complexity	Space Complexity
$O(n \cdot \log n)$	$O(n)$

## Extra resources:

- Quick sort filmpje (adhv. hongaarse volksdans)
- Veel info over quicksort en de verschillende mogelijkheden (partitionering, pivot selectie)

