Zoeken en sorteren

Classic Computer Science Algorithms

Stijn Lievens Pieter-Jan Maenhaut Koen Mertens Lieven Smits

AJ 2021–2022

HO GENT

Zoeken en sorteren

HO
GENT

Inhoud

HO GENT

Inleiding

Zoeken in een array:

- Klassieke array met elementen
- Effect van gesorteerde array? vb. woordenboek
- Aanname array: ophalen element in constante tijd onafhankelijk van positie

Effect sortering: je hoeft niet meer te zoeken in deel van array dat langs de verkeerde kant van het beschouwde element ligt

HO GENT

Linear of sequentieel zoeken

Invoer Een item zoekItem dat moet gevonden worden, een array van items genaamd rij met lengte n.

Uitvoer de index van het eerste element in rij dat gelijk is aan zoekItem wordt teruggegeven of −1 indien zoekItem niet voorkomt in rij.

```
1:  function ZoekSequentieel(zoekItem, rij)
2:      i ← 0                                   # overloopt de posities
3:      while i < n and rij[i] ≠ zoekItem do
4:          i ← i + 1
5:      if i = n then                           # niet gevonden
6:          index ← −1
7:      else
8:          index ← i                           # gevonden
9:      return index
10: end function
```

HO GENT

Binair zoeken

We kunnen ook binair zoeken. Dit werkt enkel in een gesorteerde array.
Zo zoeken we enkel verder in het relevante deel van de array.

- Beschouw een element, bv. het middelste
- Indien het gezochte element kleiner is dan het beschouwde: zoek links verder
- Indien het gezochte element groter is dan het beschouwde: zoek rechts verder

Merk op dat dit algoritme zowel iteratief als recursief kan werken.
In geval van recursie:

- Basisstap: rij met 1 element
- Recursieve stap: alle andere gevallen: de rij wordt gehalveerd

HO GENT

Binair zoeken (iteratief)

Invoer Een item `zoekItem` dat moet gevonden worden, een *gesorteerde* array genaamd `rij` van items van lengte n.

Uitvoer de index van het eerste element in `rij` dat gelijk is aan `zoekItem` wordt teruggegeven of `-1` indien `zoekItem` niet voorkomt in `rij`.

HO GENT

Binair zoeken (iteratief)

```
1:  function ZoekBinair(zoekItem, rij)
2:      l ← 0
3:      r ← n − 1
4:      while l ≠ r do          # herhalen totdat slechts één element overblijft
5:          m ← ⌊(l+r)/2⌋
6:          if rij[m] < zoekItem then
7:              l ← m + 1                    # in de rechterhelft zoeken
8:          else
9:              r ← m                        # in de linkerhelft zoeken
10:     if rij[l] = zoekItem then
11:         index ← l
12:     else
13:         index ← −1
14:     return index
15: end function
```

HO GENT

Binair zoeken (recursief)

Invoer Een item zoekItem dat moet gevonden worden, een *gesorteerde* array genaamd rij van items van lengte n.
Uitvoer de index van het eerste element in rij dat gelijk is aan zoekItem wordt teruggegeven of –1 indien zoekItem niet voorkomt in rij.

```
1: function ZOEKBINAIR(zoekItem, rij)
2:     return ZOEKRECURSIEF(zoekItem, rij, 0, n – 1)
3: end function
```

HO GENT

Binair zoeken (recursief)

Invoer Een item `zoekItem` dat moet gevonden worden, een *gesorteerde* array genaamd `rij` van items van lengte n, twee natuurlijke getallen l en r die het gedeelte van de array aangeven waarin gezocht moet worden.

Uitvoer de index van het eerste element in `rij` dat gelijk is aan `zoekItem` wordt teruggegeven indien het element voorkomt tussen $\text{rij}[l], \text{rij}[l+1], \ldots, \text{rij}[r]$. De teruggeven index ligt dan tussen l en r. Er wordt -1 teruggegeven indien `zoekItem` niet voorkomt tussen de elementen $\text{rij}[l], \text{rij}[l+1], \ldots, \text{rij}[r]$.

HO GENT

Binair zoeken (recursief)

```
1:  function ZOEKRECURSIEF(zoekItem, rij, l, r)
2:      if l = r then                                  # basisstap, rij van lengte 1
3:          if zoekItem = rij[l] then
4:              return l
5:          else
6:              return −1
7:      else                                           # inductiestap
8:          m ← ⌊(l+r)/2⌋
9:          if rij[m] < zoekItem then
10:             return ZOEKRECURSIEF(zoekItem, rij, m + 1, r)    # zoek rechts
11:         else
12:             return ZOEKRECURSIEF(zoekItem, rij, l, m)        # zoek links
13: end function
```

HO GENT

Efficiënt Zoeken

- Binair zoeken is ingewikkelder dan sequentieel zoeken.
- Binair zoeken is beter? Definieer “beter”?
- Focus op twee zaken:
  1. De uitvoeringstijd
  2. Het geheugengebruik (RAM)

HO GENT

Tijdscomplexiteit

In deze cursus: focus op uitvoeringstijd. Geen exacte berekening want:

- Verschillende computers
- Zelfde computer met andere processen actief
- Steeds krachtigere programmeertalen: inschatten aantal instructies wordt complex

HO GENT

Tijdscomplexiteit

Daarom: ASYMPTOTISCHE ANALYSE van de uitvoeringstijd. Dit karakteriseert het gedrag van de uitvoeringstijd voor “grote” waarden van de input. Typisch gedrag is bvb:

- invoer verdubbelt $\implies$ uitvoeringstijd verdubbelt
  - lineaire functie: $T(n) = n$
- invoer verdubbelt $\implies$ uitvoeringstijd $\times$ 4
  - kwadratische functie: $T(n) = n^2$
- invoer + 1 $\implies$ uitvoeringstijd $\times$ 2.
  - exponentiële functie: $T(n) = 2^n$.
- invoer verdubbelt $\implies$ uitvoeringstijd + constante.
  - logaritmische functie: $T(n) = \log(n)$
- ...

HO GENT

Tijdscomplexiteit

Bij de analyse van de zoekalgoritmen zien we dat de asymptotische uitvoeringstijd bepaald wordt door het aantal vergelijkingen dat wordt uitgevoerd.

HO GENT

Tijdscomplexiteit sequentieel zoeken

Bij sequentieel zoeken kunnen we ons afvragen hoeveel keer de volgende vergelijking wordt uitgevoerd:

$$\text{rij}[i] \neq \text{zoekItem}$$

- beste geval? $\Longrightarrow$ 1
- slechte geval? $\Longrightarrow$ n
- gemiddeld geval? $\Longrightarrow$ n ÷ 2 dus $T(n) = O(n)$

We noemen dit **lineare tijdscomplexiteit** of tijdscomplexiteit van orde n.

HO GENT

Tijdscomplexiteit binair zoeken

Bij binair zoeken tellen we hoeveel keer de vergelijking

$$\text{rij}[m] < \text{zoekItem}$$

wordt uitgevoerd. Stel: $n = 2^k$, met k een natuurlijk getal .

- $n = 1$, i.e. $k = 0 \implies$ 0 keer
- $n = 2$, i.e. $k = 1 \implies$ 1 keer
- $n = 4$, i.e. $k = 2 \implies$ 2 keer
- $n = 8$, i.e. $k = 3 \implies$ 3 keer

Dus:

$$n = 2^k \iff k = \log_2(n).$$

Dit noemen we **logaritmische tijdscomplexiteit**:

$$T(n) = O\big(\log_2(n)\big).$$

HO GENT

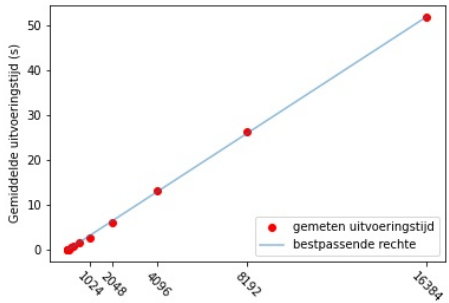
Experiment tijdscomplexiteit

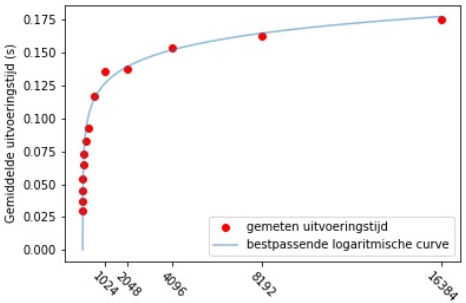
Voer een experiment uit met hetzelfde algoritme, met n variërend en meet de uitvoeringstijd bvb.

- Laat k de waardes 1 t.e.m. 14 doorlopen.
- Genereer een willekeurige rij van lengte $n = 2^k$ bestaande uit gehele getallen.
- Kies 30 random getallen uit deze rij.
- Voer voor elk van deze 30 random getallen de methode $100\,000$ keer uit en sla de uitvoeringstijd hiervan (apart) op.
- Maak een grafiek van de gemiddelde uitvoeringstijd voor elke n en lees de orde af
- Pas eventueel de juiste schaal aan op de verticale of horizontale as

HO GENT

Experiment tijdscomplexiteit





Uitvoeringstijd van lineair en binair zoeken, resp. in de linker- en rechterfiguur. Merk ook de volledige verschillende schaal op de verticale as bij beide plots.

HO GENT

Experiment tijdscomplexiteit

Aandachtspunten bij de figuren op de vorige slide:

- Merk de verschillen op
- Wat bij kleine n?
- En bij grote n?
- Wat met de variantie?

HO GENT

Sorteren door Selectie

Basisidee:

- Zoek het grootste element en plaats het achteraan.
- Sorteer de rest van de array.

HO GENT

Sorteren door Selectie

Invoer De array a is gevuld met n elementen.
Uitvoer De array a is gesorteerd.

```
1:  function SELECTIONSORT(a)
2:      for i = n − 1 ... 1 by − 1 do                    # achteraan starten
3:          positie ← i
4:          max ← a[i]
5:          for j = i − 1 ... 0 by − 1 do                # j doorloopt de deelrij
6:              if a[j] > max then
7:                  positie ← j
8:                  max ← a[j]
9:          a[positie] ← a[i]                # grootste element wisselen met laatste
10:         a[i] ← max
11: end function
```

HO GENT

Sorteren door Selectie

Voorbeeld:

44	55	12	42	94	18	06	67
44	55	12	42	67	18	06	|94
44	55	12	42	06	18	|67	94
44	18	12	42	06	|55	67	94
06	18	12	42	|44	55	67	94
06	18	12	|42	44	55	67	94
06	12	|18	42	44	55	67	94
06	12	18	42	44	55	67	94

HO GENT

Complexiteitsanalyse

De uitvoeringstijd wordt bepaald door het aantal keer dat de vergelijking

$$a[j] > \max$$

op regel 6 uitgevoerd wordt, of het aantal keer dat de teller j wijzigt

	voor i	wordt j	aantal vergelijkingen
	$n-1$ $n-2$ … 1	$n-2, n-3, \ldots, 1, 0$ $n-3, n-4, \ldots, 1, 0$ 0	$n-1$ $n-2$ … 1
totaal			$\sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$

We besluiten

$$T(n) = O(n^2)\,.$$

HO GENT

Hoe Werd die Som Bepaald?

Voorbeeld: bereken: $1 + 2 + 3 + \cdots + 10$.

stijgend	1	2	3	4	5	6	7	8	9	10
omgekeerd	10	9	8	7	6	5	4	3	2	1
som	11	11	11	11	11	11	11	11	11	11

Dus:

$$1 + 2 + 3 + \cdots + 10 = \frac{11 \times 10}{2}$$

Algemeen:

$$1 + 2 + 3 + \cdots + m = \frac{(m + 1)m}{2}$$

HO GENT

Sorteren door Tussenvoegen

Basisidee:

1. Veronderstel dat er reeds een deel vooraan de array gesorteerd is.
2. Neem het eerste element van het niet gesorteerde deel en voeg dit element toe op de juiste plaats in het gesorteerde deel. Op deze manier wordt het gesorteerde deel uitgebreid.

Sorteren door tussenvoegen of “card sort“ kan m.a.w. het best vergeleken worden met het op volgorde steken van kaarten.

HO GENT

Sorteren door Tussenvoegen

Voorbeeld:

44|	55	12	42	94	18	06	67
44	55|	12	42	94	18	06	67
12	44	55|	42	94	18	06	67
12	42	44	55|	94	18	06	67
12	42	44	55	94|	18	06	67
12	18	42	44	55	94|	06	67
06	12	18	42	44	55	94|	67
06	12	18	42	44	55	67	94

HO GENT

Sorteren door Tussenvoegen

Invoer De array a is gevuld met n elementen.
Uitvoer De array a is gesorteerd.

```
1: function CARDSORT(a)
2:     for i = 1 ... n − 1 do
3:         x ← a[i]                          # x bevat het in te voegen element
4:         j ← i                             # j zoekt de juiste positie voor x
5:         while j > 0 and x < a[j − 1] do   # schuif grotere elementen op
6:             a[j] ← a[j − 1]               # schuif a[j − 1] eentje op
7:             j ← j − 1
8:         a[j] ← x                          # x wordt op de juiste positie tussengevoegd
9: end function
```

HO GENT

Complexiteitsanalyse

Uitvoeringstijd afhankelijk van ordening bij aanvang

1. slechtste geval: binnenste lus $\frac{n(n-1)}{2}$ keer uitgevoerd. Dus

$$T(n) = O(n^2)$$

2. beste geval: buitenste lus $(n - 1)$ keer, maar binnenste lus stopt telkens onmiddellijk. Dus

$$T(n) = O(n)$$

3. gemiddeld geval:

$$T(n) = O(n^2).$$

$\Longrightarrow$ kwadratische tijdscomplexiteit, want steeds worst case analyse.

HO GENT

Sorteren door Mengen

Sorteren door mengen (*mergesort*) is een ingewikkelder algoritme dan de voorgaande twee eenvoudige sorteeralgoritmes maar is ook een heel stuk efficiënter. Basisidee:

1. Sorteer de eerste helft van de array.
2. Sorteer de tweede helft van de array.
3. Meng de twee gesorteerde deelrijen samen tot één gesorteerde array.

De eerste twee stappen in dit proces gebeuren op een *recursieve* manier. Merk op dat het eigenlijk sorteren gebeurt bij het mengen van de twee gesorteerde rijen.

HO
GENT

Sorteren door Mengen

Voorbeeld: Beschouw de rij

44 55 12 42 94 18 06 67 .

Deze wordt opgesplitst in twee deelrijen

44 55 12 42
94 18 06 67.

Beide deelrijen worden vervolgens recursief gesorteerd tot

12 42 44 55
06 18 67 94.

Tot slot worden de gesorteerde deelrijen samengevoegd tot de gesorteerde rij

06 12 18 42 44 55 67 94.

HO GENT

Mengen van twee gesorteerde (deel)rijen

12	42	44	55	06	18	67	94	06							
12	42	44	55	06	18	67	94	06	12						
12	42	44	55	06	18	67	94	06	12	18					
12	42	44	55	06	18	67	94	06	12	18	42				
12	42	44	55	06	18	67	94	06	12	18	42	44			
12	42	44	55	06	18	67	94	06	12	18	42	44	55		
12	42	44	55	06	18	67	94	06	12	18	42	44	55	67	94

HO GENT

Mengen van twee gesorteerde (deel)rijen

De functie MERGESORT roept de recursieve functie MERGESORTRECURSIVE aan. In deze methode wordt een deel van de rij gesorteerd door het te sorteren deel op te splitsen in twee deelrijen van halve lengte. Vervolgens worden de gesorteerde deelrijen gemengd. Het samenvoegen van de beide deelrijen gebeurt in de functie MERGE.

Invoer de array a is gevuld met n elementen.
Uitvoer de array a is gesorteerd

```
1: function MERGESORT(a)
2:     MERGESORTRECURSIVE(a, 0, n − 1)
3: end function
```

HO GENT

Mengen van twee gesorteerde (deel)rijen

Invoer de array a is gevuld met n elementen, *begin* en *einde* wijzen naar geldige posities in de array a.

Uitvoer de elementen met index *begin* tot en met index *einde* werden gesorteerd.

```
1: function MERGESORTRECURSIVE((a, begin, einde)
2:     if begin < einde then
3:         midden ← ⌊(begin + einde)/2)⌋
4:         MERGESORTRECURSIVE(a, begin, midden)
5:         MERGESORTRECURSIVE(a, midden + 1, einde)
6:         MERGE(a, begin, midden, einde)
7: end function
```

HO GENT

De functie merge

Invoer de array *a* is gevuld met *n* elementen; de elementen van de deelrij gaande van de *begin*-positie tot en met de *midden*-positie zijn gesorteerd; de elementen van de deelrij gaande van de (*midden*+1)-positie tot en met de *eind*-positie zijn gesorteerd.

Uitvoer de elementen met index *begin* t.e.m. index *einde* werden gesorteerd.

HO GENT

```
1:  function MERGE(a, begin, midden, einde)
2:      i ← begin                                  # de teller i doorloopt de linkse deelrij
3:      j ← midden + 1                             # de teller j doorloopt de rechtse deelrij
4:      k ← i                                      # de teller k doorloopt de hulparray hulpa
5:      hulpa ← nieuwe array[n]                    # tijdelijke hulpopslag
6:      while i ≤ midden and j ≤ einde do          # totdat een deelrij leeg is
7:          if a[i] ≤ a[j] then                    # het kleinste element komt eerst
8:              hulpa[k] ← a[i]   ;   i ← i + 1
9:          else
10:             hulpa[k] ← a[j]   ;   j ← j + 1
11:         k ← k + 1                              # er is een element in hulpa opgeslagen
12:     if i > midden then                         # de 2de deelrij moet leeggemaakt worden
13:         while j ≤ einde do
14:             hulpa[k] ← a[j]   ;   j ← j + 1   ;   k ← k + 1
15:     else                                       # de 1ste deelrij moet leeggemaakt worden
16:         while i ≤ midden do
17:             hulpa[k] ← a[i]   ;   i ← i + 1   ;   k ← k + 1
18:     for k = begin ... einde do                 # kopieer gesorteerde deelrijen in a
19:         a[k] ← hulpa[k]
20: end function
```

Complexiteitsanalyse

- Merge methode: mengen van deelrijen met lengte $n/2 \implies$ lineair in n
- Stel nu $n = 2^k$ met k een natuurlijk getal
- MergeSortRecursive methode: als $n = 1$ dan is begin = einde $\implies$ vgl wordt 1 keer doorlopen

$$T(1) = 1,$$

Als $n = 2$ dan

$$T(2) = T(1) + T(1) + 2(mengen) = 4$$

$$T(4) = 2T(2) + 4 = 2 \times 4 + 4 = 12,$$

$$T(8) = 2T(4) + 8 = 2 \times 12 + 8 = 32.$$

Hieruit kunnen we herkennen dat het algemene patroon is:

$$T(n) = T(2^k) = n \times (k + 1).$$

Dus tijdscomplexiteit $T(n)$ van sorteren door mengen:

$$T(n) = O\big(n \log_2(n)\big).$$

HO GENT

Geheugengebruik

- Mergesort is dus tijdsefficiënter dan voorgaande
- doch minder geheugen efficiënt
- MERGE-algoritme: extra hulprij nodig

HO GENT

Oefeningen 1 t.e.m. 4

1. Ga in de array (1 2 3 4 6 7 8 9 10) achtereenvolgens op zoek naar de waarden 3, 5 en 10. Maak hierbij gebruik van het algoritme ZOEKBINAIR. Houd tijdens je simulatie de waarden bij van de variabelen *l*, *r* en *m*.
2. (**Dodona**) Implementeer de iteratieve methode voor binair zoeken.
3. (**Dodona**) Herwerk en implementeer de oplossingsmethode voor sorteren door selectie zodat niet langer het grootste element naar achteren wordt gebracht maar het kleinste element naar voor. Het reeds gesorteerde deel van de array bevindt zich m.a.w. steeds vooraan.
4. Pas de oplossingsmethode beschreven in het opgegeven algoritme aan zodat de elementen niet langer van klein naar groot worden gesorteerd maar van groot naar klein.
   4.1 sorteren door tussenvoegen
   4.2 sorteren door mengen

HO GENT

Oefening 5

(**Dodona**) Bubble sort is een oplossingsmethode voor het sorteren van een array a van lengte n. Het algoritme hieronder beschrijft deze methode in pseudo-code.

Invoer De array a is gevuld met n elementen.
Uitvoer De array a is gesorteerd.

```
1: function BUBBLESORT(a)
2:     for i = 0 ... n − 2 do
3:         for j = n − 1 ... i + 1 by − 1 do
4:             if a[j − 1] > a[j] then
5:                 wissel de elementen a[j − 1] en a[j]
6: end function
```

HO GENT

Oefening 5 (vervolg)

- Implementeer deze methode in Python.
- Tel het exact aantal keer, in functie van n, dat de controle van regel 4, nl. $a[j-1] > a[j]$ wordt uitgevoerd. Gebruik een gesloten formule om je resultaat weer te geven. Leid hieruit de O-notatie voor de uitvoeringstijd van het algoritme bubbleSort af.
- Stel $a = (13, 7, 8, 1)$. Volg in een overzichtelijke tabel de uitwerking van BUBBLESORT(a). Noteer in de tabel minstens de verschillende waarden die i, j en a doorlopen.

HO GENT

Oefening 6

Hieronder volgen een aantal codefragmenten f_i. Bepaal voor elk van deze fragmenten de waarde van x in functie van de invoerwaarde n. Leid hieruit de asymptotische uifvoeringstijd, t.t.z. de O-notatie, af van deze verschillende codefragmenten. Tracht je resultaten experimenteel te verifiëren door een programma te schrijven dat de verhouding berekent van de $f_i(n)$ over je theoretisch voorspelde uitvoeringstijd. Wanneer je de juiste uitvoeringstijd hebt dan zou deze verhouding zo goed als constant moeten zijn (voor grote waarden van n).

HO
GENT

Oefening 6 (vervolg)

```
1: function SOM1(n)
2:     x ← 0
3:     for i = 1 ... n do
4:         x ← x + 1
5:     return (x)
6: end function
```

HO GENT

Oefening 6 (vervolg)

```
1: function SOM2(n)
2:     x ← 0
3:     for i = 1 ... 2n do
4:         for j = 1 ... n do
5:             x ← x + 1
6:     return (x)
7: end function
```

HO GENT

Oefening 6 (vervolg)

```
1: function SOM3(n)
2:     x ← 0
3:     for i = 1 ... n do
4:         for j = 1 ... i do
5:             for k = 1 ... j do
6:                 x ← x + 1
7:     return (x)
8: end function
```

HO GENT

Oefening 6 (vervolg)

```
1: function SOM4(n)
2:     x ← 0
3:     for i = 1 ... n do
4:         for j = 1 ... i do
5:             for k = 1 ... i do
6:                 x ← x + 1
7:     return (x)
8: end function
```

HO GENT

Oefening 6 (vervolg)

```
1: function SOM5(n)
2:     x ← 0
3:     i ← n
4:     while i ≥ 1 do
5:         x ← x + 1
6:         i ← ⌊i/2⌋
7:     return (x)
8: end function
```

HO GENT

Oefening 6 (vervolg)

```
1: function SOM6(n)
2:     x ← 0
3:     i ← n
4:     while i ≥ 1 do
5:         for j = 1 ... i do
6:             x ← x + 1
7:         i ← ⌊i/2⌋
8:     return (x)
9: end function
```

HO GENT

Oefening 6 (vervolg)

```
1: function SOM7(n)
2:     x ← 0
3:     i ← n
4:     while i ≥ 1 do
5:         for j = 1 ... n do
6:             x ← x + 1
7:         i ← ⌊i/2⌋
8:     return (x)
9: end function
```

HO GENT