Algoritmen en Datastructuren

Lucien Sina

28.11.2025

ISBN: 9789403831749

Gedrukt door Bookmundo

Inhoudsopgave

1 Inleiding **1**

2 Algoritmen en hun Analyse **4**

2.1 Abstractieniveaus van Probleembeschrijvingen 4

2.1.1 Voorbeeld: Lidmaatschapstest in een Verzameling . . . 5

2.2 Beoordeling van de Kwaliteit van een Algoritme 8

2.2.1 Correctheid 8

2.2.2 Begrijpelijkheid 8

2.2.3 Eenvoud van Implementatie . 8

2.2.4 Efficiëntie (Uitvoeringstijd en Geheugengebruik) 9

2.2.5 Afwegingen tussen Criteria . 9

2.3 Uitvoeringstijd en Geheugengebruik van Algoritmen 10

2.3.1 Meten van de Uitvoeringstijd 10
2.4 Constructie van een Looptijdfunctie 14
2.4.1 Inleiding 14
2.4.2 Abstractie van Uitvoeringstijd 14
2.4.3 Case Study: Zoeken in een Array 16
2.4.4 Gebruik van O-Notatie 17
2.4.5 Oefeningen en Oplossingen . 18
2.4.6 Samenvatting 21
2.5 Analyse van Controlestructuren . . . 22
2.5.1 Elementaire Operaties en Sequenties 22
2.5.2 Looptijden van Basisbesturingselementen 24
2.6 Het Verbeteren van een Algoritme . 27
2.6.1 Tijdcomplexiteitsanalyse . . . 27
2.6.2 Verbeterd Algoritme met Vroege Terminatie 28
2.6.3 Tijdcomplexiteitsanalyse . . . 28
2.6.4 Geoptimaliseerd Algoritme met Binaire Zoekopdracht . . 30
2.6.5 Tijdcomplexiteitsanalyse . . . 31
2.6.6 Demonstratie 31
2.7 Algemene O-notatie 32
2.7.1 Definities 32
2.7.2 Relaties Tussen Functies . . . 33

2.7.3 Voorbeeld: Vergelijken van $\log n$ en $\sqrt{n}$ 35
2.7.4 Voorbeeld: Runtime van `contains` en $\texttt{contains}_2$ . . . 36

3 Algebra's en abstracte datatypen 40
3.1 Contexten 40
3.2 Abstractieniveaus 41
3.2.1 Algebra als Datatype 41
3.2.2 Formaliseren van Datatypen en Algebra's 43
3.3 Algebra's en Datatypen 43
3.3.1 Voorbeeld Handtekening . . . 44
3.3.2 Twee Benaderingen voor Specificatie 46
3.4 Oefening: Algebra voor een Teller . . 50
3.5 Polymorfe en Monomorfe Datatypes 52

4 Basisconcepten 55
4.1 Algoritme 55
4.1.1 Belangrijke Concepten 57
4.2 Definities van Algebra, ADT, en Model 60
4.3 Oefeningen en Oplossingen 62
4.3.1 Oefening 1.1: Algoritme . . . 62
4.3.2 Oefening 1.2: Signatuur, Algebra, en ADT 63
4.3.3 Oefening 1.3: Datatypes en Gegevensstructuren 64

4.3.4 Oefening 1.4: Monomorfe en Polymorfe ADT's 65
4.3.5 Oefening 1.5: Implementatie van Gegevensstructuren . . . 66

5 Programmeren en Gegevensstructuren 68
5.1 Constructie van Gegevensstructuren 69
5.2 Programmeertalen 70
5.3 Gegevenstypes in Java 71
5.3.1 Primitieve Gegevenstypes . . 71
5.4 Primitieve Gegevenstypes in Java . . 73
5.5 Arrays 75
5.5.1 Voorbeeld 75
5.5.2 Waardebereik van een Array Type 76
5.5.3 Adresberekening 76
5.5.4 Efficiëntie van Array Toegang 77
5.5.5 Arrays als Representatietools 77
5.5.6 Voorbeeld: Itereren door een Array 77
5.5.7 Specifieke Array Bereiken . . 77
5.5.8 Java-Specifiek Gedrag 78
5.6 Klassen 78
5.6.1 Structuur van een Klassedefinitie 80
5.6.2 Methoden in Klassen 81

5.6.3 Realiseren van Algebra of Abstracte Gegevenstypes . . 81
5.7 Records 82
5.7.1 Definitie van Records 82
5.7.2 Voorbeelden van Record Definities 83
5.7.3 Operaties op Records 84
5.7.4 Waardebereik van een Recordtype 85
5.7.5 Representatie en Adresberekening 85
5.7.6 Toegang in Constante Tijd . 86

6 Dynamische Gegevensstructuren 87
6.1 Pointertypes 88
6.1.1 Definitie van Pointertypes . . 88
6.1.2 Declaratie en Gebruik van Pointervariabelen 88
6.1.3 Grafische Weergave van Pointers 89
6.1.4 Belang van Pointers in Dynamische Gegevensstructuren 90
6.2 Dereferencing en Garbage Collection 90
6.3 Referentietypen in Java 95
6.3.1 Toewijzingen 95
6.4 Methode-aanroepen in Java 98

7 Aanvullende datastructuren 103
7.1 Enumeratie Types 103
7.2 Subbereik Types 107
7.3 Verzamelingen 110

8 Elementaire Datatypes 114
8.1 Lijsten 114
8.1.1 Lijsten met Eerste, Rest, VoegToe en Concateneer . . . 115
8.2 Lijsten met Expliciete Posities 117

9 Lijstimplementaties 123
9.1 Dubbel-Gelinkte Lijst 123
9.2 Implementatie van Dubbel-Gelinkte Lijsten 126
9.2.1 Een Lege Lijst Maken 126
9.2.2 Toegang tot de Eerste Positie 127
9.2.3 Invoegen van Elementen . . . 127
9.2.4 Lijsten Samenvoegen 129
9.2.5 Elementen Zoeken 130
9.2.6 Elementen Verwijderen . . . 131
9.3 Eenvoudige Gelinkte Lijst 132
9.4 Gekoppelde Lijsten in Arrays 136

10 Stapel 141
10.1 Specificatie van een Stack 142
10.2 Stack Implementatie 143

11 Queues **147**
11.1 Queue-operaties en uitbreidingen . . 148
11.2 Algebraïsche specificatie van queues 148
11.3 Queue-handtekening 149
11.3.1 Diagramvoorstelling 149
11.4 Toepassingen van Queues 150
11.5 Implementatie van Queues in Java . 151
11.5.1 Implementatie van een Queue met een Cyclische Array 151
11.5.2 Implementatie van een Queue met Twee Stacks . . . 154
11.5.3 Vergelijking van de Twee Implementaties 157

12 Mappings **158**
12.1 Kenmerken van Mappings 159
12.2 Mappings met Eindig en Scalaire Domeinen 159
12.3 Grote of Spaarzame Domeinen . . . 160
12.4 Samenvatting van Mapping-Implementaties 162

13 Binaire Bomen **164**
13.1 Definitie 164
13.1.1 Binaire Bomen 165
13.1.2 Bladeren en Deelbomen . . . 166
13.1.3 Knopen en Hun Classificatie 167

13.1.4 Paden, Voorouders en Afstammelingen 168
13.1.5 Subtrees en Padlengtes . . . 169
13.1.6 Hoogte en Diepte 169
13.2 Kenmerken van Binaire Bomen . . . 170
13.2.1 Maximale Hoogte van een Binaire Boom 170
13.2.2 Minimale Hoogte van een Binaire Boom 171
13.2.3 Exacte Minimale Hoogte van een Binaire Boom 172
13.2.4 Relatie Tussen Bladeren en Interne Knoten 173
13.2.5 Samenvatting van Kenmerken 174
13.3 Boomalgebra 175
13.3.1 Soorten 175
13.3.2 Operaties 175
13.3.3 Setdefinitie 175
13.3.4 Functiedefinities 175
13.3.5 Recursieve Structuren 176
13.3.6 Boomdoorlopen 176
13.3.7 Oefening: Berekening van de Hoogte van een Boom 177
13.4 Binaire Boom Java Implementaties . 178
13.4.1 Met Pointers 178
13.4.2 Met Array Inbedding 180
13.4.3 Complexiteit Vergelijking . . 183

14 Algemene Bomen **184**
14.1 Definitie 184
14.1.1 Illustratie 185
14.2 Implementaties van Algemene Bomen 187
14.2.1 Implementatie met Arrays . . 187
14.2.2 Implementatie met Binaire Bomen 189

15 Oefeningen **191**

16 Eindwoorden **209**
16.1 Aanbevolen Boeken door Lucien Sina 210
16.1.1 Logica 210
16.1.2 Programmeren 210
16.1.3 Theoretische informatica . . . 211
16.1.4 Complexiteitstheorie 211
16.1.5 Zoek-, sorteer- en graafalgoritmen 212
16.1.6 De leereeks 212
16.2 Literatuur 213
16.3 Dankbetuiging 214

Voorwoord

Efficiënte algoritmen en datastructuren vormen het hart van de informatica en zijn de basis voor het oplossen van complexe problemen binnen en buiten de programmeerwereld. Het vermogen om algoritmen te ontwerpen, analyseren en implementeren is essentieel om de uiteenlopende uitdagingen aan te pakken waarmee programmeurs worden geconfronteerd. Dit boek heeft als doel lezers de nodige tools en inzichten aan te reiken om zich in dit boeiende vakgebied te kunnen bewegen.

Om de kernproblemen binnen de informatica aan te pakken, moeten programmeurs leren om nieuwe algoritmen te creëren door bestaande op een vaardige manier te combineren. Even belangrijk is het vermogen om deze algoritmen te analyseren op het gebied van uitvoeringstijd en geheugengebruik. Datastructuren, die informatie organiseren om

efficiënte algoritmische bewerkingen mogelijk te maken, vormen een ander cruciaal aandachtspunt van dit boek.

Er zal een duidelijk onderscheid worden gemaakt tussen **datatypen**—de abstracte, specificatiegerichte kijk op het organiseren van gegevens—en **datastructuren**, die de concrete implementaties van deze datatypen vertegenwoordigen. Lezers zullen ontdekken dat één datatype meerdere implementaties kan hebben, elk met zijn eigen voor- en nadelen. Door deze verschillen te verkennen, helpt dit boek bij het ontwikkelen van een diepgaand begrip van hoe men efficiënte oplossingen voor uiteenlopende situaties kan kiezen en ontwerpen.

Om optimaal van dit boek te profiteren, is een basisvaardigheid in programmeren aan te bevelen. Bekendheid met een taal als Java zal bijzonder nuttig zijn. Voor lezers die nog basiskennis van programmeren nodig hebben, verwijs ik graag naar mijn eerdere boek over programmeren, waarin de essentiële concepten voor effectief coderen worden geïntroduceerd. Hoewel enige wiskundige vaardigheid de begripsvorming kan bevorderen, heb ik er bewust voor gekozen om alle benodigde wiskundige concepten in dit boek te behandelen.

Om het leren te versterken, bevat dit boek talrijke oefeningen, compleet met oplossingen. Deze

oefeningen zijn bedoeld om de theoretische concepten te verankeren en praktische ervaring op te doen met implementaties.

Of je nu student bent, programmeur, of gewoon nieuwsgierig naar algoritmen en datastructuren, dit boek is zo opgebouwd dat het je stap voor stap door de materie leidt. Ik hoop dat het een waardevolle bron zal zijn tijdens jouw ontdekkingsreis binnen dit essentiële onderdeel van de informatica.

Hoofdstuk 1

Inleiding

Algoritmen werken op datastructuren, en datastructuren bevatten algoritmen als componenten. Daardoor zijn algoritmen en datastructuren onlosmakelijk met elkaar verbonden. In dit boek streven we ernaar om algoritmen en datastructuren te classificeren binnen de context van verwante concepten zoals functies, procedures, abstracte datatypen, datatypen, algebra's, typen, klassen en modules.

Op het meest abstracte niveau gebruiken we de wiskunde om de specificaties van algoritmen en datastructuren te formaliseren. Een algoritme realiseert een functie, die als specificatie voor dat algoritme dient. Evenzo is een algoritme zelf een

specificatie van een procedure, functie of methode die uiteindelijk geïmplementeerd moet worden.

Het is belangrijk te beseffen dat algoritmen doorgaans niet als computerprogramma's worden gepresenteerd. In plaats daarvan worden ze beschreven op een hoger, menselijk leesbaar niveau om de communicatie te vergemakkelijken. Een computerprogramma *kan* echter wel dienen als een manier om een algoritme te beschrijven. Met andere woorden: een computerprogramma **is** een algoritme, maar de beschrijving van een algoritme is meestal niet beperkt tot een computerprogramma.

In het eerste hoofdstuk richten we ons op de analyse van algoritmen, waarbij we specifiek hun uitvoeringstijd en geheugengebruik onder de loep nemen.

Aan de kant van datastructuren beginnen we met abstracte concepten zoals **abstracte datatypen** en **algebra's**, die concrete datatypen definiëren. Een datastructuur is een implementatie van een algebra of een abstract datatype op algoritmisch niveau. Datastructuren kunnen op hun beurt worden geïmplementeerd in programmeertalen. Op programmeerniveau komen we belangrijke concepten tegen zoals datatypen, klassen en modules.

Deze gestructureerde benadering stelt ons in staat om de kloof te overbruggen tussen hoogab-

stracte concepten en hun concrete implementaties in programmeercode, wat leidt tot een omvattend begrip van algoritmen en datastructuren.

Hoofdstuk 2

Algoritmen en hun Analyse

2.1 Abstractieniveaus van Probleembeschrijvingen

In deze sectie verkennen we de verschillende abstractieniveaus waarop algoritmen beschreven kunnen worden. Ter illustratie nemen we het volgende voorbeeld:

2.1.1 Voorbeeld: Lidmaatschapstest in een Verzameling

Gegeven een verzameling S van gehele getallen, bepaal of een specifiek getal c voorkomt in S.

Wiskundige Specificatie

Op het abstractieniveau van de wiskunde kunnen we het probleem als volgt specificeren:

Laat $\mathbb{Z}$ de verzameling van gehele getallen zijn en $F(\mathbb{Z})$ de verzameling van alle eindige deelverzamelingen van $\mathbb{Z}$. Laat $\text{bool} = \{\text{waar}, \text{onwaar}\}$. Definieer de functie

$$\text{bevat} : F(\mathbb{Z}) \times \mathbb{Z} \to \text{bool}$$

als:

$$\text{bevat}(S, c) = \begin{cases} \text{waar} & \text{als } c \in S, \\ \text{onwaar} & \text{anders.} \end{cases}$$

Algoritmische Beschrijving

Op algoritmisch niveau moeten we een representatie kiezen voor de verzameling S. Laten we S representeren als een array. Een algoritme om het probleem op te lossen kan dan als volgt worden beschreven:

Algoritme bevat(S,c)
Invoer: S als een array van lengte n, en een geheel getal c. **Uitvoer: waar**, als $c \in S$; anders, **onwaar**.

```
1  var b: bool;
2  b := false;
3  for i := 1 to n do
4      if S[i] = c then
5          b := true;
6      end if;
7  end for;
8  return b;
```

Beschrijving op Programmeerniveau

Op programmeerniveau moeten we een specifieke programmeertaal kiezen. Laten we Java gebruiken om het algoritme te beschrijven:

```
1  public boolean bevat(int[] S, int c) {
2      boolean b = false;
3      for (int i = 0; i < S.length; i++) {
4          if (S[i] == c) {
5              b = true;
6          }
7      }
8      return b;
```

```
9 }
```

Bespreking

Zoals aangetoond, beïnvloedt het abstractieniveau hoe een probleem wordt beschreven:

1. Wiskundig Niveau: Op dit niveau specificeren we **wat** bepaald of berekend moet worden, zonder te beschrijven **hoe** dit gebeurt. De wiskundige specificatie van een functie kan op meerdere manieren worden geïmplementeerd.

2. Algoritmisch Niveau: Hier is het doel om te communiceren **hoe** iets bepaald of berekend wordt. Het algoritme is bedoeld voor menselijk begrip, niet voor een compiler. Taal-specifieke details worden weggelaten, zodat de focus op de essentiële logica blijft. De ontwerper van het algoritme gaat uit van enige basiskennis bij de lezer.

3. Programmeerniveau: Op dit niveau drukken we het algoritme uit in een specifieke programmeertaal. De syntaxis en details van de gekozen taal worden belangrijk voor de implementatie.

Door deze abstractieniveaus te scheiden, kunnen we algoritmen systematisch ontwerpen, analyseren en implementeren, terwijl we duidelijkheid en focus behouden op de essentiële aspecten van het probleem.

2.2 Beoordeling van de Kwaliteit van een Algoritme

De kwaliteit van een algoritme wordt beoordeeld aan de hand van verschillende belangrijke criteria, die elk bijdragen aan het nut en de effectiviteit van het algoritme. Deze criteria zijn als volgt:

2.2.1 Correctheid

De primaire vereiste voor elk algoritme is correctheid. Een algoritme moet het probleem waarvoor het ontworpen is, nauwkeurig en betrouwbaar oplossen onder alle gespecificeerde voorwaarden.

2.2.2 Begrijpelijkheid

Een algoritme moet gemakkelijk te begrijpen zijn. Dit verhoogt de kans op correctheid, vereenvoudigt de implementatie en maakt het makkelijker om in de toekomst aanpassingen te doen. Een helder en begrijpelijk algoritme vermindert het risico op fouten tijdens gebruik en onderhoud.

2.2.3 Eenvoud van Implementatie

Het schrijven en debuggen van een algoritme kan het meest complexe deel van zijn levenscyclus zijn.

Daarom moet een algoritme eenvoudig en snel te implementeren zijn. Dit zorgt ervoor dat het efficiënt in de praktijk gebracht kan worden.

2.2.4 Efficiëntie (Uitvoeringstijd en Geheugengebruik)

Een algoritme moet zijn uitvoeringstijd en geheugengebruik minimaliseren. Efficiëntie is vooral belangrijk wanneer het algoritme wordt gebruikt in omgevingen met beperkte middelen of wanneer het grote datasets moet verwerken.

2.2.5 Afwegingen tussen Criteria

Het is belangrijk op te merken dat deze criteria vaak onderling afgewogen moeten worden. Zo kan het prioriteren van begrijpelijkheid leiden tot een minder efficiënt algoritme, terwijl een sterke focus op efficiëntie het algoritme moeilijker te begrijpen of te implementeren kan maken. Het vinden van een goede balans tussen deze afwegingen is een essentieel aspect van het ontwerpen en evalueren van algoritmen, en vereist zorgvuldige afweging van de specifieke vereisten en beperkingen van het probleem.