

Hoofdstuk 1

Inleiding

Inhoud

1. Terminologie
2. ADTs
3. Opslagdatastructuren en Genericiteit
4. Snelheid en geheugengebruik van algoritmen

1.1 Terminologie

Primitieve Data vs. Samengestelde Data

Heet ook
datastructuur

```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
> #t
#t
> 3.14
3.14
> 22/7
3 1/7
> 4
4
> #\a
#\a
> #\newline
#\newline
> #\space
#\space
> abs
#<procedure:abs>
> 'a-symbol
a-symbol
```

boolean

number

character

procedure

symbol




```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512
MB.
> (cons 1 2)
(1 . 2)
> (vector "one" "two" "three")
#("one" "two" "three")
> "this string contains many characters"
"this string contains many characters"
```

pair

vector

string



*Samengestelde data is data die met behulp van een **dataconstructor** geconstrueerd wordt. De samenstellende delen zijn primitieve data waarden of — op hun beurt — samengestelde data. Samengestelde data is dus data die afbreekbaar is (in dezelfde programmeertaal)*

Primitieve datawaarden zijn datawaarden die niet afbreekbaar zijn (in dezelfde programmeertaal)

Procedures om datastructuren te bewerken

```
> (cons 1 2)
(1 . 2)
> (vector "one" "two" "three")
#("one" "two" "three")
> "this string contains many characters"
"this string contains many characters"
> (make-vector 10)
#(0 0 0 0 0 0 0 0 0 0)
> (make-string 5 #\space)
"      "
```

Letterlijke
Constructoren

Procedurele
Constructoren



*Dataconstructor =
Geheugen reserveren + Onderdelen initialiseren*

Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.

```
> (define c (cons 1 2))
> (car c)
1
> (cdr c)
2
> (define v (vector 5 4 3 2 1))
> (vector-ref v 2)
3
> (define v (vector "one" "two" "three" "four"))
> (vector-ref v 2)
"three"
> (vector-set! v 3 "cinq")
> v
#("one" "two" "three" "cinq")
> (set-car! c "one")
> (set-cdr! c "два")
> c
("one" . "два")
```

Accessoren
(aka getters)

Accessoren
(aka getters)

Mutatoeren
(aka setters)




Datatypes

Elke datawaarde behoort tot een verzameling met een zekere naam: het *datatype* van de waarde. Datatypes van primitieve datawaarden heten *primitieve datatypes*.

boolean number

procedure

```
> (number? 3.14)
#t
> (number? 22/7)
#t
> (number? abs)
#f
> (number? number?)
#f
> (procedure? number?)
#t
> (procedure? "abs")
#f
> (string? "abs")
#t
> (symbol? 'abs)
#t
> (boolean? "waar")
#f
> (pair? 3.14)
#f
```



Datatypes van samengestelde datawaarden heten *samengestelde datatypes*

vector

pair

string

We hebben (en maken) een
predicaat per datatype


Datatype = Naam + Operaties

Ieder datatype bepaalt welke operaties toepasbaar zijn op zijn waarden!

Raadpleeg de standaard van R7RS voor de details

pair
car
cdr
set-car!
set-cdr!
eq?
eqv?
...

vector
vector-ref
vector-set!
vector-length
vector->list
eq?
...
...

> (+ 3.14 22/7)
6.282857142857143
> (+ "3.14" 22/7)
✗ ✗: contract violation
expected: number?
given: "3.14" 

2	Revised ⁷ Scheme	
	SUMMARY	CONTENTS
	The report gives a defining description of the programming language Scheme. Scheme is a statically scoped and properly tail recursive dialect of the Lisp programming language [23] invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and object-oriented styles, find convenient expression in Scheme.	Introduction 3
	The introduction offers a brief history of the language and of the report.	1 Overview of Scheme 5
	The first three chapters present the fundamental ideas of the language and describe the notational conventions used for describing the language and for writing programs in the language.	1.1 Semantics 5
	Chapters 4 and 5 describe the syntax and semantics of expressions, definitions, programs, and libraries.	1.2 Syntax 5
	Chapter 6 describes Scheme's built-in procedures, which include all of the language's data manipulation and input/output primitives.	1.3 Notation and terminology 5
	Chapter 7 provides a formal syntax for Scheme written in extended BNF, along with a formal denotational semantics. An example of the use of the language follows the formal syntax and semantics.	2 Lexical conventions 7
	Appendix A provides a list of the standard libraries and the identifiers that they export.	2.1 Identifiers 7
	Appendix B provides a list of optional but standardized implementation feature names.	2.2 Whitespace and comments 8
	The report concludes with a list of references and an alphabetic index.	2.3 Other notations 8
	<i>Note:</i> The editors of the R ⁵ RS and R ⁶ RS reports are listed as authors of this report in recognition of the substantial portions of this report that are copied directly from R ⁵ RS and R ⁶ RS. There is no intended implication that those editors, individually or collectively, support or do not support this report.	2.4 Datum labels 9
		3 Basic concepts 9
		3.1 Variables, syntactic keywords, and regions 9
		3.2 Disjointness of types 10
		3.3 External representations 10
		3.4 Storage model 10
		3.5 Proper tail recursion 11
		4 Expressions 12
		4.1 Primitive expression types 12
		4.2 Derived expression types 14
		4.3 Macros 21
		5 Program structure 25
		5.1 Programs 25
		5.2 Import declarations 25
		5.3 Variable definitions 25
		5.4 Syntax definitions 26
		5.5 Record-type definitions 27
		5.6 Libraries 28
		5.7 The REPL 29
		6 Standard procedures 30
		6.1 Equivalence predicates 30
		6.2 Numbers 32
		6.3 Booleans 40
		6.4 Pairs and lists 40
		6.5 Symbols 43
		6.6 Characters 44
		6.7 Strings 45
		6.8 Vectors 48
		6.9 Bytevectors 49
		6.10 Control features 50
		6.11 Exceptions 54
		6.12 Environments and evaluation 55
		6.13 Input and output 55
		6.14 System interface 59
		7 Formal syntax and semantics 61
		7.1 Formal syntax 61
		7.2 Formal semantics 65
		7.3 Derived expression types 68
		A Standard Libraries 73
		B Standard Feature Identifiers 77
		Language changes 77
		Additional material 80
		Example 81
		References 81
		Alphabetic index of definitions of concepts, keywords, and procedures 84

Proceduretypes

Zo'n proceduretype
is geen Scheme!

Iedere Scheme procedure werkt op *invoer/argumenten* van een zeker datatype en produceert *uitvoer* van een zeker datatype. De combinatie van deze types is *het proceduretype van de procedure*.

car
(pair → any)

append
(pair pair → pair)

+
(number ... number → number)

sin
(number → number)

(argtype₁ ... argtype_n → result-type)

Procedure-types voor Hogere-Orde Procedures

```
(define (zero f a b epsilon)
  (define c (/ (+ a b) 2))
  (cond ((< (abs (f c)) epsilon) c)
        ((< (* (f a) (f c)) 0) (zero f a c epsilon))
        (else (zero f c b epsilon))))
```



Zo'n proceduretype
is geen Scheme!

```
zero
  ( (number → number) number number number → number )
```

Algoritmen

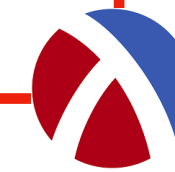


Genoemd naar Abu Abdullah
Muhammad ibn Musa al-Khwarizmi
("de zoon van Khwarizmi")

Een *algoritme* is een *algemene* en
berekenbare procedure om een
probleem op te lossen

```
(define (fac n)
  (define (fac-rec n)
    (if (= n 0)
        1
        (* n (fac-rec (- n 1)))))
  (if (odd? n)
      (error "It's odd!")
      (fac-rec n)))
```

Scheme procedure
(berekenbaar), maar **niet**
voldoende algemeen




~~"Om de code van een bankkaart te
kraken bel je het orakel van Delfi!"~~

Algemene procedure,
maar **niet berekenbaar**
door computer

Primitieve vs. Samengestelde Algoritmen

Een *primitief algoritme* is een algoritme dat niet afbreekbaar is (in dezelfde programmeertaal)

```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
> +
#<procedure:+>
> *
#<procedure:*>
> =
#<procedure:=>
> abs
#<procedure:abs>
```



	letrec	lambda	
cond		let*	if
	do		

Samengestelde algoritmen zijn algoritmen die met behulp van een algoritmische constructoren geconstrueerd wordt. De samenstellende delen zijn primitieve algoritmen of — op hun beurt — samengestelde algoritmen. Samengestelde algoritmen zijn dus afbreekbaar (in dezelfde programmeertaal)

define

define stelt geen
code samen

Procedurele Abstractie & Data Abstractie

Waarom zo belangrijk?

According to the official Google employee report, **27,169 software engineers** work at Google (i.e. research & development) (source). 23 sep. 2021

Procedurele abstractie bestaat erin een (betekenisvolle) **naam** te geven aan een samengestelde procedure

Verdeelt programmeurs in 2 groepen: **gebruikers** en **implementator**

Gebruikerscode **leesbaar** en korter. Implementatorcode **makkelijk aan te passen**

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```



```
(define (fib n)
  (define (iter n a b)
    (if (= n 0)
        a
        (iter (- n 1) b (+ a b))))
  (iter n 1 1))
```

Welke???

```
> (fib 10)
89
```



Data abstractie bestaat erin een (betekenisvolle) **naam** te geven aan samengestelde data en aan alle **operaties** die op die data inwerken.

Verdeelt programmeurs in 2 groepen: **gebruikers** en **implementator**

Gebruikerscode **leesbaar** en korter. Implementatorcode **makkelijk aan te passen**.

```
> (define wolf (person "wolfgang" "de Meuter" 18 100000))
> (name wolf)
"wolfgang"
> (age wolf)
18
> (age! wolf 25)
> (age wolf)
25
```

record?

pair?

vector?



Data abstractie wordt verwezenlijkt m.b.v. ADT's ...

1.2 ADTs

Abstract Data Types

Is een concept dat niks met Scheme te maken heeft. Verschillende programmeertalen bieden verschillende technieken aan om ADTs technisch in code om te zetten.

Een *ADT* bestaat uit een *naam voor een datatype* en een *reeks proceduretypes* die de procedures beschrijven waarmee data elementen van het ADT gemanipuleerd kunnen worden. Dit zijn proceduretypes voor de constructoren, de accessoren, de mutators en de operaties.

Verdeelt programmeurs in 2 groepen: *gebruikers* en *implementators*



Edsger Dijkstra
1930-2002

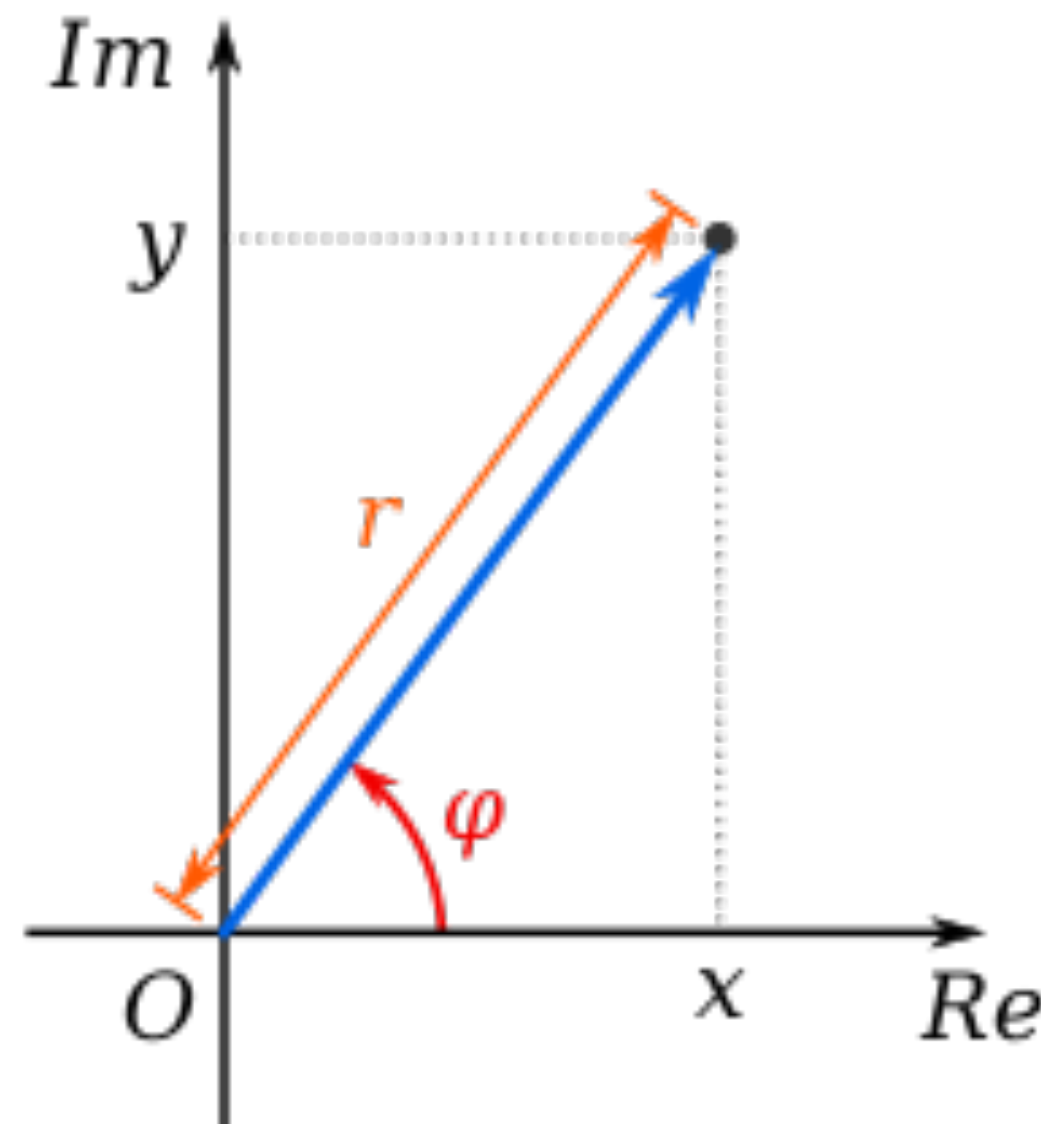
"Separation of Concerns"
(1974)



Barbara Liskov
1939-

"Programming with abstract data types"
(1974)

Voorbeeld: Complexe Getallen



(Bron: wikipedia)

```
ADT complex
  new      ( number number → complex )
  complex? ( any → boolean )
  +        ( complex complex → complex )
  -        ( complex complex → complex )
  /        ( complex complex → complex )
  *        ( complex complex → complex )
  modulus  ( complex → number )
  argument ( complex → number )
  real     ( complex → number )
  imag     ( complex → number )
```

*Een ADT is een abstract concept. Gegeven een programmeertaal, dan kiest een implementator voor een **representatie** en een **implementatie**. De implementator probeert de concrete representatie zo goed mogelijk weg te stoppen voor de gebruikers. (bvb met **libraries** of met **objecten**)*

Procedurele ADT Implementatie: Lijsten (1/3)

```
(define-library (complex)
  (export new complex? real imag + - / * modulus argument)
  (import (scheme inexact) (scheme cxx)
    (rename (except (scheme base) complex?)
      (+ number+) (* number*) (/ number/) (- number-))))
(begin
  (define complex-tag 'complex)

  (define (new r i)
    (list complex-tag r i))

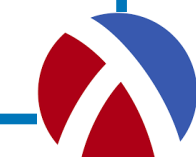
  (define (complex? any)
    (and (pair? any)
      (eq? (car any) complex-tag)))

  (define (real c)
    (cadr c))

  (define (imag c)
    (caddr c))

  (define (+ c1 c2)
    (define result-real (number+ (real c1) (real c2)))
    (define result-imag (number+ (imag c1) (imag c2)))
```

In deze eerste
implementatie van het
ADT complex stellen we
complexe getallen voor
als Scheme **lijstjes**



Procedurele ADT Implementatie: Lijsten (2/3)

```
(define (+ c1 c2)
  (define result-real (number+ (real c1) (real c2)))
  (define result-imag (number+ (imag c1) (imag c2)))
  (new result-real result-imag))
```

```
(define (- c1 c2)
  (define result-real (number- (real c1) (real c2)))
  (define result-imag (number- (imag c1) (imag c2)))
  (new result-real result-imag))
```

```
(define (* c1 c2)
  (define result-real (number- (number* (real c1) (real c2))
                                (number* (imag c1) (imag c2))))
  (define result-imag (number+ (number* (real c1) (imag c2))
                                (number* (imag c1) (real c2))))
  (new result-real result-imag))
```

```
(define (/ c1 c2)
  (define denom (number+ (number* (real c2)
                                    (real c2))
                          (number* (imag c2)
                                    (imag c2))))
  (define result-real (number+ (number* (real c1)
                                          (real c2))
```

We programmeren zoveel mogelijk operaties van het ADT zo **abstract** mogelijk, d.w.z. zo weinig mogelijk gebruik makend van Scheme cxxr procedures

$(a,b) + (c,d) = (a+b, c+d)$

$(a,b) - (c,d) = (a-b, c-d)$



Procedurele ADT Implementatie: Lijsten (3/3)

```
(define (/ c1 c2)
  (define denom (number+ (number* (real c2)
                                    (real c2))
                          (number* (imag c2)
                                    (imag c2))))
  (define result-real (number- (number* (real c1)
                                          (real c2))
                               (number* (imag c1)
                                          (imag c2))))
  (define result-imag (number- (number* (imag c1)
                                          (real c2))
                               (number* (real c1)
                                          (imag c2))))
  (new (number/ result-real denom) (number/ result-imag denom)))

(define (modulus c)
  (sqrt (number+ (number* (real c) (real c))
                 (number* (imag c) (imag c)))))

(define (argument c)
  (atan (imag c) (real c))))
```

$(a,b) * (c,d) = (ac-bd, ad+bc)$

$(a,b) / (c,d) = ((ac+bd)/(c*c+d*d), (bc-ad)/(c*c+d*d))$

Zo **abstract** mogelijk



Procedurele ADT Implementatie: Records (1/2)

```
(define-library
  (complex)
  (export new complex? real imag + - / * modulus argument)
  (import (scheme inexact)
    (rename (except (scheme base) complex?)
      (+ number+) (* number*) (/ number/) (- number-))))

(begin

  (define-record-type complex
    (new r i)
    complex?
    (r real)
    (i imag))

  (define (+ c1 c2)
    (define result-real (number+ (real c1) (real c2)))
    (define result-imag (number+ (imag c1) (imag c2)))
    (new result-real result-imag))

  (define (- c1 c2)
    (define result-real (number- (real c1) (real c2)))
    (define result-imag (number- (imag c1) (imag c2)))
    (new result-real result-imag))
```

In deze tweede
implementatie van het
ADT complex stellen we
complexe getallen voor
als **records**

Rest van de code
ongewijzigd



Procedurile ADT Implementatie: Records (2/2)

```
(define (/ c1 c2)
  (define denom (number+ (number* (real c2)
                                    (real c2))
                          (number* (imag c2)
                                    (imag c2))))
  (define result-real (number+ (number* (real c1)
                                          (real c2))
                               (number* (imag c1)
                                          (imag c2))))
  (define result-imag (number- (number* (imag c1)
                                          (real c2))
                               (number* (real c1)
                                          (imag c2))))
  (new (number/ result-real denom) (number/ result-imag denom)))

(define (modulus c)
  (sqrt (number+ (number* (real c) (real c))
                 (number* (imag c) (imag c)))))

(define (argument c)
  (atan (imag c) (real c)))
```



Gebruik van de Procedurele Implementatie

We importeren de library die het ADT implementeert. Vanaf nu weten we niet meer wat de exacte technische samenstelling van complexe getallen is.

```
(import (prefix (a-d examples complex-1) complex:)
        (scheme base)
        (scheme write))
```

```
(define cpx1 (complex:new 1 4))
(define cpx2 (complex:new 5 3))
(display (complex:+ cpx1 cpx2))(newline)
(display (complex:* cpx1 cpx2))(newline)
(display (complex:/ cpx1 cpx2))(newline)
(display (complex:- cpx1 cpx2))(newline)
(display (complex:real cpx1))(newline)
(display (complex:imag cpx2))(newline)
(display (complex:modulus cpx1))(newline)
(display (complex:argument cpx2))
```

Dit stukje code is geschreven in procedurele stijl. We passen steeds procedures toe op argumenten (bvb. complexe getallen)

```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
(complex 6 7)
(complex -7 23)
(complex 1/2 1/2)
(complex -4 1)
1
3
4.123105625617661
0.5404195002705842
>
```

Aan de uitvoer van print zie je wat er gebeurt. In deze interactie met de REPL werd de lijst-versie gebruikt.



```
Welcome to DrRacket, version 8.1 [cs].
Language: r7rs, with debugging; memory limit: 512 MB.
#(struct:complex 6 7)
#(struct:complex -7 23)
#(struct:complex 1/2 1/2)
#(struct:complex -4 1)
1
3
4.123105625617661
0.5404195002705842
>
```

In deze interactie werd de record-versie gebruikt.



Objectgebaseerde Implementatie (1/2)

```
(define (make-complex r i)
  (define (complex+ c)
    (make-complex (+ r (c 'real))
                  (+ i (c 'imag))))
  (define (complex* c)
    (make-complex (- (* r (c 'real))
                     (* i (c 'imag)))
                  (+ (* r (c 'imag))
                     (* i (c 'real'))))
  (define (complex- c)
    (make-complex (- r (c 'real))
                  (- i (c 'imag))))
  (define (complex/ c)
    (define denom (+ (* (c 'real)
                        (c 'real))
                     (* (c 'imag)
                        (c 'imag))))
    (define real (+ (* r (c 'real)) (* i (c 'imag))))
    (define imag (- (* i (c 'real)) (* r (c 'imag))))
    (make-complex (/ real denom) (/ imag denom)))
  (define (modulus)
    (sqrt (+ (* r r) (* i i))))
  (define (argument)
```

In deze derde implementatie van het ADT complex stellen we complexe getallen voor als **dispatcher functies**



Objectgebaseerde Implementatie (2/2)

```
(define (modulus)
  (sqrt (+ (* r r) (* i i))))
(define (argument)
  (atan i r))
(define (real)
  r)
(define (imag)
  i)
(lambda (message . args)
  (cond ((eq? message '+) (apply complex+ args))
        ((eq? message '-') (apply complex- args))
        ((eq? message '*') (apply complex* args))
        ((eq? message '/') (apply complex/ args))
        ((eq? message 'modulus) (modulus))
        ((eq? message 'argument) (argument))
        ((eq? message 'real) (real))
        ((eq? message 'imag) (imag))
        ((eq? message 'complex->list) (list 'complex r i))
        (else (display "Complex Number Message Not Understood"))))))
```

In deze derde
implementatie van het
ADT complex stellen we
complexe getallen voor
als **dispatcher functies**

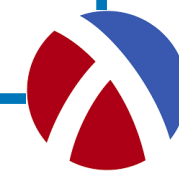


Gebruik van de Objectgerichte Implementatie

```
(import (a-d examples complex-3)
        (scheme base)
        (scheme write))

(define cpx1 (make-complex 1 4))
(define cpx2 (make-complex 5 3))
(display ((cpx1 '+ cpx2) 'complex->list))(newline)
(display ((cpx1 '* cpx2) 'complex->list))(newline)
(display ((cpx1 '- cpx2) 'complex->list))(newline)
(display ((cpx1 '/ cpx2) 'complex->list))(newline)

(display (cpx1 'real))(newline)
(display (cpx2 'imag))(newline)
(display (cpx1 'modulus))(newline)
(display (cpx2 'argument))
```



```
Welcome to DrRacket, version 8.1 [cs].
Language: R5RS; memory limit: 512 MB.
(complex 6 7)
(complex -7 23)
(complex -4 1)
(complex 1/2 1/2)
1
3
4.123105625617661
0.5404195002705842
>
```



Resulteert in een compleet andere programmeerstijl. In plaats van procedures op argumenten toe te passen (= **procedurele programmeerstijl**) sturen we een bericht naar één van de “argumenten” (= **objectgerichte stijl**)

Overzicht van de Terminologie

Deze principes kunnen in elke programmeertaal gebruikt en toegepast worden, zij het misschien op technisch andere manieren.

	Data	Algoritmen
Primitief	Primitieve Data	Ingebouwde Procedures
Samengesteld	Datastructuur	Zelf geschreven Procedures
Abstractie	ADTs	Procedurenamen en Proceduretypes

1.3 Opslagdatastructuren en Genericiteit

Opslagdatastructuren

Een Excel file

Een telefoonboek

Een bibliotheekcatalogus

Een datastructuur die dient om datawaarden op te slaan met als bedoeling die later weer op te kunnen vragen noemen we een opslagdatastructuur (Eng: storage data structure).

Een complex getal

Een breuk

Een matrix

Voorbeeld v/e Opslagdatastructuur

Een “doos” die steeds
“het grootste ding ooit
toegevoegd” onthoudt

```
(define-library (max-o-mem)
  (export new max-o-mem? read write!)
  (import (scheme base))
  (begin

    (define-record-type max-o-mem
      (new sma val)
      max-o-mem?
      (sma <<)
      (val value value!))

    (define (read mom)
      (value mom))

    (define (write! mom new-value)
      (define sma (<< mom))
      (define val (value mom))
      (if (sma val new-value)
          (value! mom new-value))
      mom)))
```

*Doordat de constructor eigenlijk een hogere order procedure is (ze verwacht een “sma” procedure) is deze code **onafhankelijk** van het soort waarden die in één bepaalde max-o-mem kunnen opslaan.*

```
> (define mom1 (mom:new < 0))
> (define mom2 (mom:new string<? ""))
> (define mom3 (mom:new char<? #\space))
> (mom:write! mom1 10)
#(struct:max-o-mem #<procedure:<> 10)
> (mom:write! mom1 5)
#(struct:max-o-mem #<procedure:<> 10)
> (mom:write! mom1 40)
#(struct:max-o-mem #<procedure:<> 40)
> (mom:write! mom2 "wolf")
#(struct:max-o-mem #<procedure:string<?> "wolf")
> (mom:write! mom2 "viviane")
#(struct:max-o-mem #<procedure:string<?> "wolf")
> (mom:write! mom3 "xantippe")
✗ ✗: /examples/max-o-mem.rkt:31:10: char<?: contract
violation
  expected: char?
  given: "xantippe"
```



Genericiteit van een Opslagdatastructuur

```
(define-library (max-o-mem)
  (export new max-o-mem? read write!)
  (import (scheme base))
  (begin

    (define-record-type max-o-mem
      (new sma val)
      max-o-mem?
      (sma <<)
      (val value value!))

    (define (read mom)
      (value mom))

    (define (write! mom new-value)
      (define sma (<< mom))
      (define val (value mom))
      (if (sma val new-value)
          (value! mom new-value))
      mom)))
```

*Een **generische datastructuur** is een datastructuur die onafhankelijk is van het datatype van de elementen die in de datastructuur zitten. In Scheme wordt dit met een hogere-orde constructor bewerkstelligd.*



Genericiteit van een ADT

DT<V> wil zeggen dat DT de naam van het ADT is en dat het ADT gebruik maakt van “waarden van soort V” zonder dat V gefixt wordt op een type.

```
ADT max-o-mem< T >  
  
new  
  ( ( T T → boolean ) T → max-o-mem< T > )  
max-o-mem?  
  ( any → boolean )  
write!  
  ( max-o-mem< T > T → max-o-mem< T > )  
read  
  ( max-o-mem< T > → T )
```

```
Welcome to DrRacket, version 8.1 [cs].  
Language: r7rs, with debugging; memory limit: 512 MB.  
> (define complex-mom (mom:new complex< (complex:new 0 0)))  
> (define number-mom (mom:new < 0))  
> complex-mom  
#(struct:max-o-mem #<procedure:complex<> (complex 0 0))  
> number-mom  
#(struct:max-o-mem #<procedure:<> 0)
```

T = complex

T = number



Gebruik van Generische ADTs

```
(import (prefix (a-d examples max-o-mem) mom:)
        (prefix (a-d examples complex-1) complex:)
        (scheme base)
        (scheme write)
        (scheme inexact))
```

```
(define (greatest lst << init)
  (define max (mom:new << init))
  (define (iter lst)
    (mom:write! max (car lst))
    (if (not (null? (cdr lst)))
        (iter (cdr lst))))
  (iter lst)
  (mom:read max))
```

; Example with complex numbers

```
(define (complex< c1 c2)
  (define (square x) (* x x))
  (< (sqrt (+ (square (complex:real c1))
             (square (complex:imag c1))))
    (sqrt (+ (square (complex:real c2))
             (square (complex:imag c2))))))
```

```
> (define complex-mom (mom:new complex< (complex:new 0 0)))
> (define number-mom (mom:new < 0))
> (define complex-list (list (complex:new 1 0) (complex:new 0 1)
                             (complex:new 3 4) (complex:new 4 3)))
> (define number-list (list 1 2 3 4 5))
> (display (greatest number-list < 0))
5
> (display (greatest complex-list complex< (complex:new 0 0)))
(complex 3 4)
>
```

T = complex

T = number



Een Belangrijk Generisch ADT: dictionary

ADT dictionary< K V >

Key-value paren met
keys van type K en
values van type V

new

((K K → boolean) → dictionary< K V >)

dictionary?

(any → boolean)

insert!

(dictionary< K V > K V → dictionary< K V >)

delete!

(dictionary< K V > K → dictionary< K V >)

find

(dictionary< K V > K → V ∪ {#f})

empty?

(dictionary< K V > → boolean)

full?

(dictionary< K V > → boolean)

Associatief
geheugen (met
elke key wordt
een value
geassocieerd)

*Een dictionary is een datastructuur die “**associaties**” beheert. Elke associatie bestaat uit een **key** en een **value**. De belangrijkste operaties zijn het toevoegen, verwijderen en opzoeken van een associatie. Het snel implementeren van deze 3 operaties is één van de centrale vraagstukken van de cursus.*

Voorbeeldgebruik van Dictionaries

```
(define english-dutch (new string=?))

(insert! (insert! english-dutch "algorithm" "algoritme") "ADT" "Abstract Datatype")
(insert! english-dutch "key" "sleutel")
(insert! english-dutch "value" "waarde")
(insert! english-dutch "accessor" "uitlezer")
(insert! english-dutch "mutator" "overschrijver")
(insert! english-dutch "operation" "operatie")

(delete! english-dutch "mutator")
(if (find english-dutch "mutator")
    (error "It is still there!" "mutator"))
```

K = string en
V = string



1.4 Snelheid en geheugengebruik van algoritmen

Performantie van Algoritmen

Mensen vinden computers en software “goed” als die snel hun werk doen en weinig kosten.

Geheugen kost geld. Time is money.

Het bestuderen van tijdsgebruik en geheugenverbruik van verschillende algoritmen en datastructuren is één van de centrale vraagstukken van de computerwetenschappen.

Performantie van Algoritmen

De *experimentele* benadering

We zouden de snelheid van een algoritme kunnen *meten* met een stopwatch of met de `(time-it (algoritme))` expressie.

seconden,
nanoseconden

```
> (time-it (fib 10))  
(0 . 0)  
> (time-it (fib 20))  
(0 . 5000000)  
> (time-it (fib 30))  
(0 . 260000000)  
> (time-it (fib 40))  
(19 . 986000000)  
>
```



chronometer.rkt

Nadeel = dit geeft absolute getallen die je niet kan veralgemenen

De *analytische* benadering

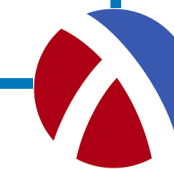
We *tellen* het aantal computationele stappen die nodig zijn bij een *invoer van grootte n*.

Dit levert een functie op, $f_A(n)$, de *performantiekarakteristiek* van algoritme A

Voordeel = deze methode is technologie-onafhankelijk en algemeen. Je kan van algoritme A en algoritme B $f_A(n)$ en $f_B(n)$ vergelijken door eenvoudige wiskundige analyse.

Voorbeeld

```
(define (greatest lst)
  (define (iter result l)
    (cond
      ((null? l) result)
      ((< result (car l)) (iter (car l) (cdr l)))
      (else (iter result (cdr l)))))
  (iter 0 lst))
```



In het **ergste geval** is:
 $f_{\text{greatest}}(n) = 7n + 3$

In het **beste geval** is:
 $f_{\text{greatest}}(n) = 6n + 3$



Soorten Analyses van Algoritmen

Erger dan dit zal het algoritmen nooit performen.

*Bij een **worst-case analyse** onderzoeken we de performantiekenarakteristiek van een algoritme en gaan we uit van de slechtst mogelijke invoer.*

Beter dan dit zal het algoritmen nooit performen.

*Bij een **best-case analyse** onderzoeken we de performantiekenarakteristiek van een algoritme en gaan we uit van de best mogelijke invoer.*

Wat is “gemiddeld”?
Statistiek nodig.

*Bij een **average-case analyse** onderzoeken we de performantiekenarakteristiek van een algoritme en gaan we uit van de gemiddelde invoer.*

Algoritmen en datastructuren 2.

*Bij een **amortised analyse** onderzoeken we de performantiekenarakteristiek van een algoritme voor een rij van achtereenvolgende toepassingen ervan.*

Een Experimentje in Excel

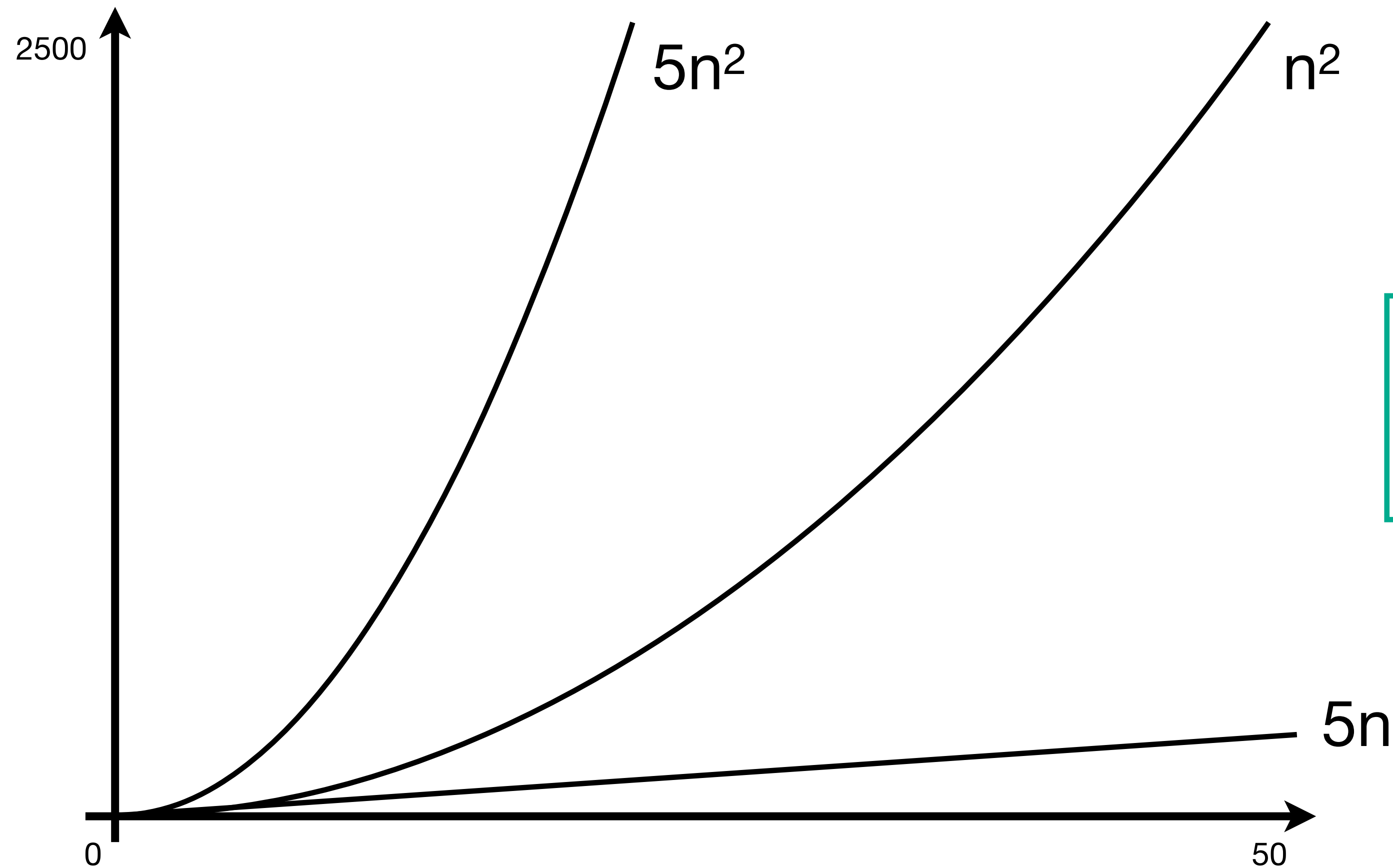
n	$\log(n)$	\sqrt{n}	n	$n.\log(n)$	n^2	n^3	2^n
2	1	2	2	2	4	8	4
4	2	2	4	8	16	64	16
8	3	3	8	24	64	512	256
16	4	4	16	64	256	4096	65536
32	5	6	32	160	1024	32768	4294967296
64	6	8	64	384	4096	262144	1.85×10^{19}
128	7	12	128	896	16384	2097152	3.40×10^{38}
256	8	16	256	2048	65536	16777216	1.16×10^{77}
512	9	23	512	4608	262144	134217728	1.34×10^{154}
1024	10	32	1024	10240	1048576	1073741824	1.79×10^{308}

5n of 100n maakt geen
verschil in vergelijking met 2^n

We zijn *enkel geïnteresseerd* in grote n 'en
en we zijn enkel geïnteresseerd in *de*
kolom waarin een algoritme zich bevindt.

5n of 100n maakt geen
verschil in vergelijking met 2^n

Voorbeeld: Lineaire vs. Kwadratische Groei



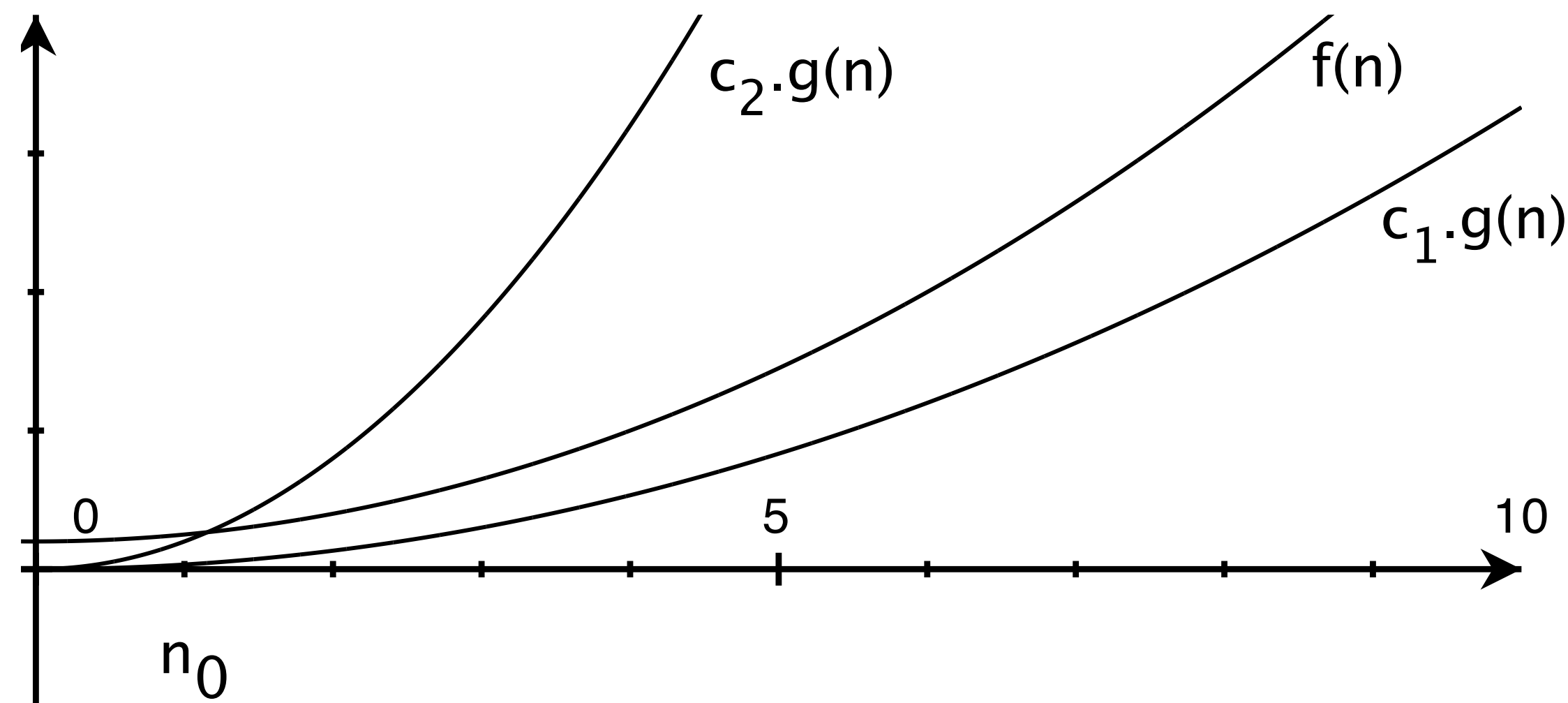
We hebben een precieze taal nodig die ons toelaat te zeggen dat bvb. n , $2n$, $10n+5$ allen *van de soort* n zijn en dat bvb. n^2 en $3n^2+4n+3$ *van de soort* n^2 zijn

Wiskundige Taal#1: Grote Θ

Elke f die je tussen 2 veelvouden van g kan "sandwichen" vanaf een bepaalde n_0

$$\Theta(g(n)) = \{ f \mid \exists c_1, c_2 > 0, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$

$5n \in \Theta(n)$
 $5n^2 \notin \Theta(n)$
 $n/34 \in \Theta(n)$
 $6n+2.3 \in \Theta(n)$
 $1 \notin \Theta(n^2)$
 $5n \notin \Theta(n^2)$
 $10n^3+3n^2 \notin \Theta(n^2)$



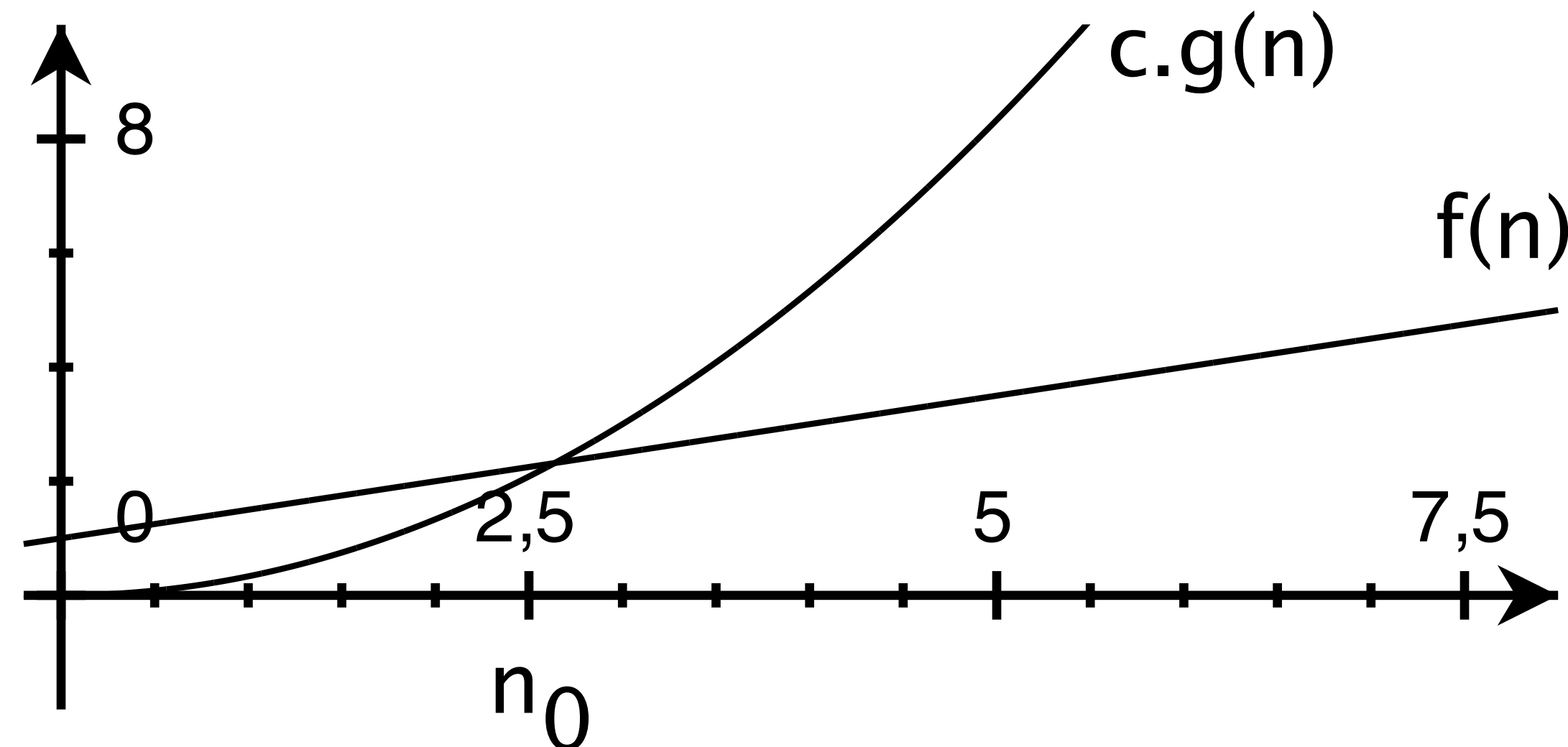
Deze taal is nuttig als het algoritme geen worst-case en geen best-case gedrag heeft maar altijd hetzelfde aantal stapjes doet.

Wiskundige Taal#2: Grote O

Elke f tussen die je onder een veelvoud van g kan houden vanaf een bepaalde n_0

$$O(g(n)) = \{ f \mid \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq f(n) \leq c g(n) \}$$

$5n \in O(n)$
 $5n^2 \notin O(n)$
 $n/34 \in O(n)$
 $6n+2.3 \in O(n)$
 $1 \in O(n^2)$
 $5n \in O(n^2)$
 $10n^3+3n^2 \notin O(n^2)$



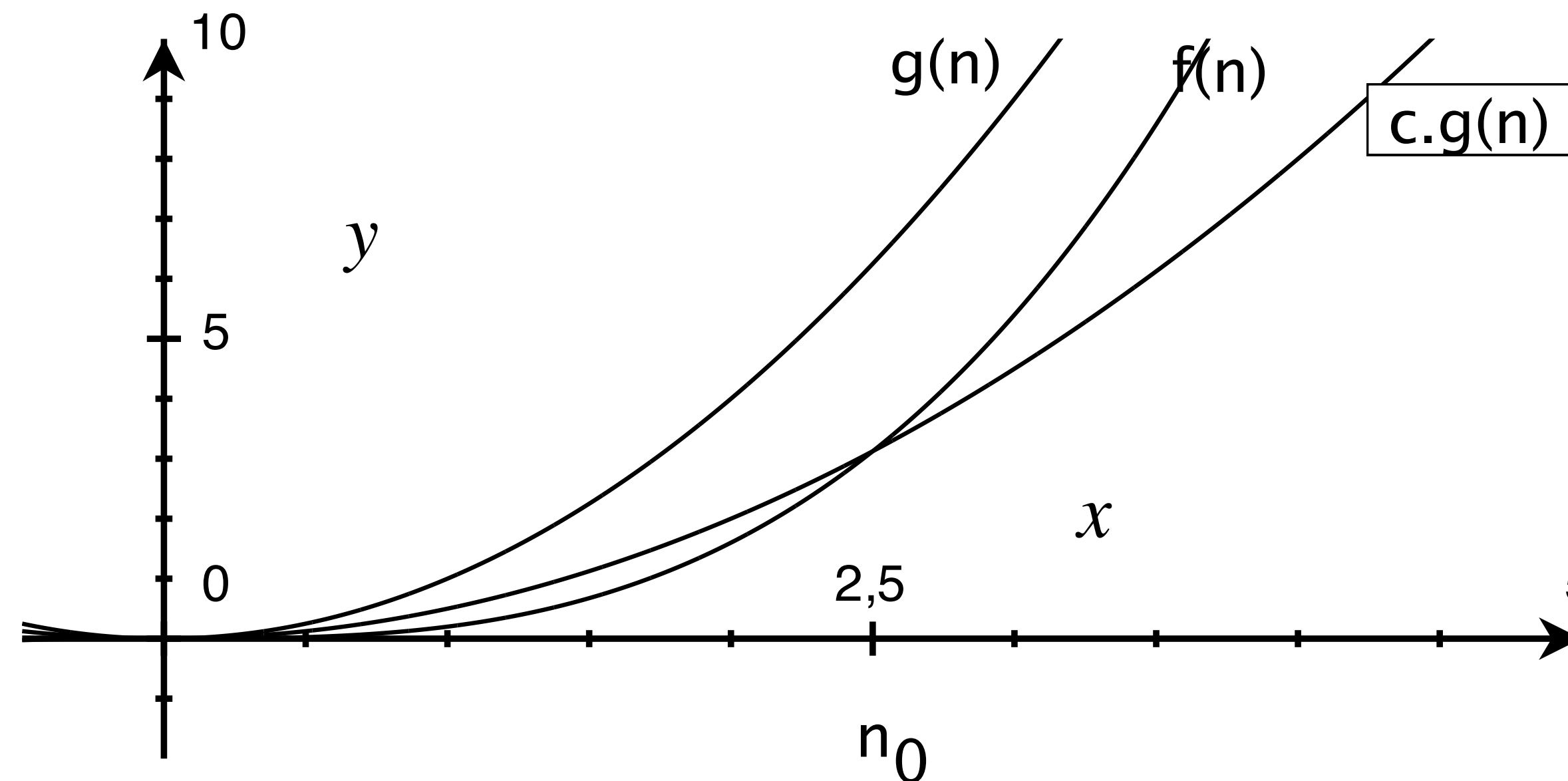
Deze taal is nuttig als het algoritme een worst-case gedrag heeft dat “onder g” zit.

Wiskundige Taal#3: Grote Ω

Elke f tussen die je boven een veelvoud van g kan houden vanaf een bepaalde n_0

$$\Omega(g(n)) = \{ f \mid \exists c, n_0 \geq 0 : \forall n \geq n_0 : 0 \leq c g(n) \leq f(n) \}$$

$5n \in \Omega(n)$
 $5n^2 \in \Omega(n)$
 $n/34 \in \Omega(n)$
 $6n+2.3 \in \Omega(n)$
 $1 \notin \Omega(n^2)$
 $5n \notin \Omega(n^2)$
 $10n^3+3n^2 \in \Omega(n^2)$



Deze taal is nuttig als het algoritme een best-case gedrag heeft dat “boven g” zit.

Eigenschappen, Rekenregels & Opmerkingen

Voor alle functies f en g geldt:

$f \in \Theta(g)$ als en slechts als $f \in \Omega(g)$ én $f \in O(g)$

Enkel dominante termen tellen mee:

$$O(t_1(n) + t_2(n)) = O(\max(t_1(n), t_2(n)))$$

$$\Omega(t_1(n) + t_2(n)) = \Omega(\min(t_1(n), t_2(n)))$$

waarbij:

$$1 < \log(n) < \sqrt{n} < n < n \cdot \log(n) < n^{k-1} < n^k < 2^n < n!$$

Idem voor Ω

$O(c \cdot f) = O(f)$ voor alle constanten c (kies gewoon een andere constante in de definitie)

- Gevolg: $O(c) = O(1)$
- Gevolg: $O(\log_a(n)) = O(\log_b(n))$

Let soms op:

- In zeldzame gevallen zitten er **heel grote constanten** in Θ , Ω en O -expressies verborgen
- Θ , Ω en O zijn enkel nuttig om functies van **verschillende orde** te vergelijken. Eén algoritme van $O(n)$ kan toch véél sneller zijn dan een ander van $O(n)$.

Analyse van 1 enkele Scheme Expressie

$$O(f_{(\text{define } n \ e)}(n)) = O(f_{(\text{set! } v \ e)}(n)) = O(1 + f_e(n))$$

$$O(f_{(\text{set-car! } p \ e)}(n)) = O(f_{(\text{set-cdr! } p \ e)}(n)) = O(1 + f_p(n) + f_e(n))$$

$$O(f_{(\text{let } ((v1 \ e1) (v2 \ e2) \dots (vn \ en)) b1 \ b2 \dots bk)}(n)) = O(1 + f_{e1}(n) + \dots + f_{en}(n) + f_{b1}(n) + \dots + f_{bk}(n))$$

$$O(f_{(\text{begin } e1 \dots en)}(n)) = O(1 + f_{e1}(n) + \dots + f_{en}(n))$$

$$O(f_{(\text{cond } ((c1 \ a1) \dots (cn \ an)))}(n)) = O(1 + f_{c1}(n) + \dots + f_{cn}(n) + f_{a1}(n) + \dots + f_{an}(n))$$

$$O(f_{(\text{if } c \ t \ e)}(n)) = O(1 + f_c(n) + f_t(n) + f_e(n))$$

$$O(f_{(\text{lambda } args \ body)}(n)) = O(1)$$

$$O(f_{(p \ a1 \dots an)}(n)) = O(1 + f_p(n) + f_{a1}(n) + \dots + f_{an}(n))$$

$O(f_p(n)) \in O(1)$ voor $p \in \{+, -, \text{eq?}, \text{vector-ref}, \text{vector-set}, \text{car}, \text{cdr}, \dots\}$

$O(f_p(n)) \in O(n)$ voor $p \in \{\text{list} \rightarrow X, X \rightarrow \text{list}, \text{string-append}, \dots\}$

$$O(f_{(f \ a1 \dots an)}(n)) = O(1 + f_f(n) + f_{a1}(n) + \dots + f_{an}(n) + f_{\text{body}}(n))$$

Analyse van Scheme Procedures

$f_{\text{weird}} \in O(n^2)$

```
(define (weird vector)
  (define len (vector-length vector))
  (if (odd? len)
      (a-linear-function len)
      (a-quadratic-function len)))
```

Voor *niet-recursieve Scheme procedures*:
Bepaal O/Ω voor alle deelexpressies van
de body en neem de som, d.w.z. het
maximum/minimum.

Recurrentievergelijkingen

$$f_{\text{fac}}(n) = 1 + f_{\text{fac}}(n-1)$$

```
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
```



*Vuistregel: voor een recursieve Scheme
procedure bepaal je $O(b(n))$ voor de body en
bepaal je $O(r(n))$ voor het aantal recursieve
oproepen. De procedure is dan $O(b(n).r(n))$*

$$b_{\text{fac}}(n) \in O(1)$$


$$r_{\text{fac}}(n) \in O(n)$$

Dus:

$$f_{\text{fac}}(n) \in O(n)$$

Voorbeelden


```
(define (fib1 n)
  (if (< n 2)
      1
      (+ (fib1 (- n 1)) (fib1 (- n 2)))))
```



$b_{\text{fib1}}(n) \in O(1)$
 $r_{\text{fib1}}(n) \in O(2^n)$
Dus:
 $f_{\text{fib1}}(n) \in O(2^n)$

Eigenlijk is:
 $f_{\text{fib1}}(n) \in \Theta(\Phi^n)$

```
(define (fib2 n)
  (define (iter n a b)
    (if (= n 0)
        a
        (iter (- n 1) b (+ a b))))
  (iter n 0 1))
```



$b_{\text{iter}}(n) \in O(1)$
 $r_{\text{iter}}(n) \in O(n)$
Dus:
 $f_{\text{iter}}(n) \in O(n)$

Addendum #1: Named Let

```
(define (fac n)
  (let fac-loop
    ((result 1)
     (factor n))
    (if (= factor 0)
        result
        (fac-loop (* result factor) (- factor 1)))))
```



```
(define (fac n)
  (define (fac-loop result factor)
    (if (= factor 0)
        result
        (fac-loop (* result factor) (- factor 1))))
  (fac-loop 1 n))
```

*Dus de performantiekarakteristiek van **een named let** zoeken is equivalent aan de analyse van een recursieve procedure.*

Addendum #2: Meerdere Argumenten

```
(define (sum n m)
  (if (= n 0)
      m
      (+ 1 (sum (- n 1) m))))
```

```
(define (times n m)
  (if (= m 1)
      n
      (sum n (times n (- m 1)))))
```

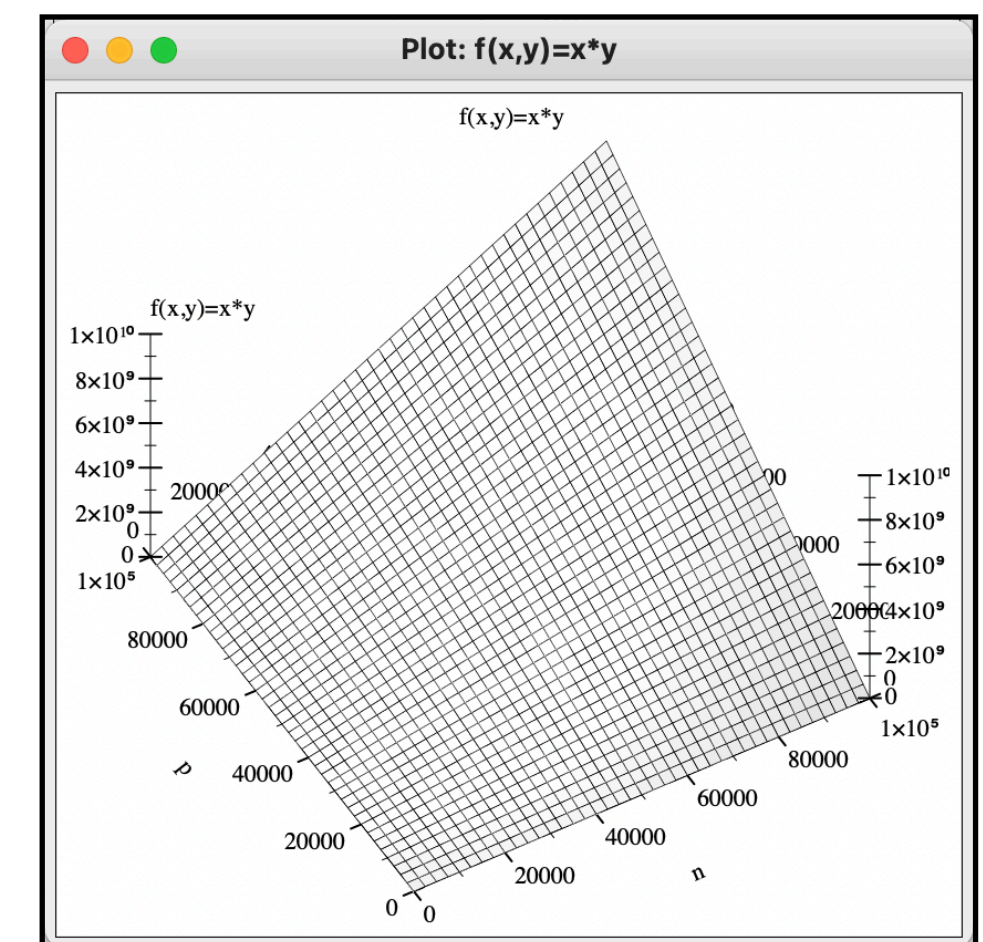
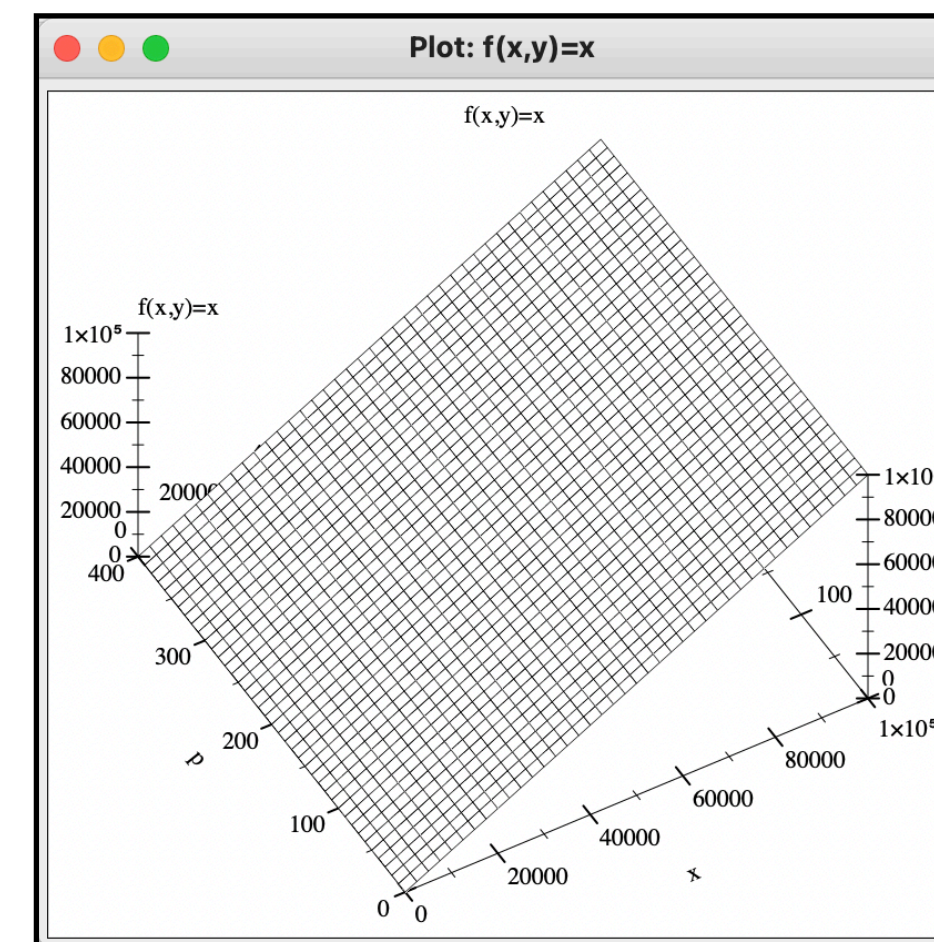
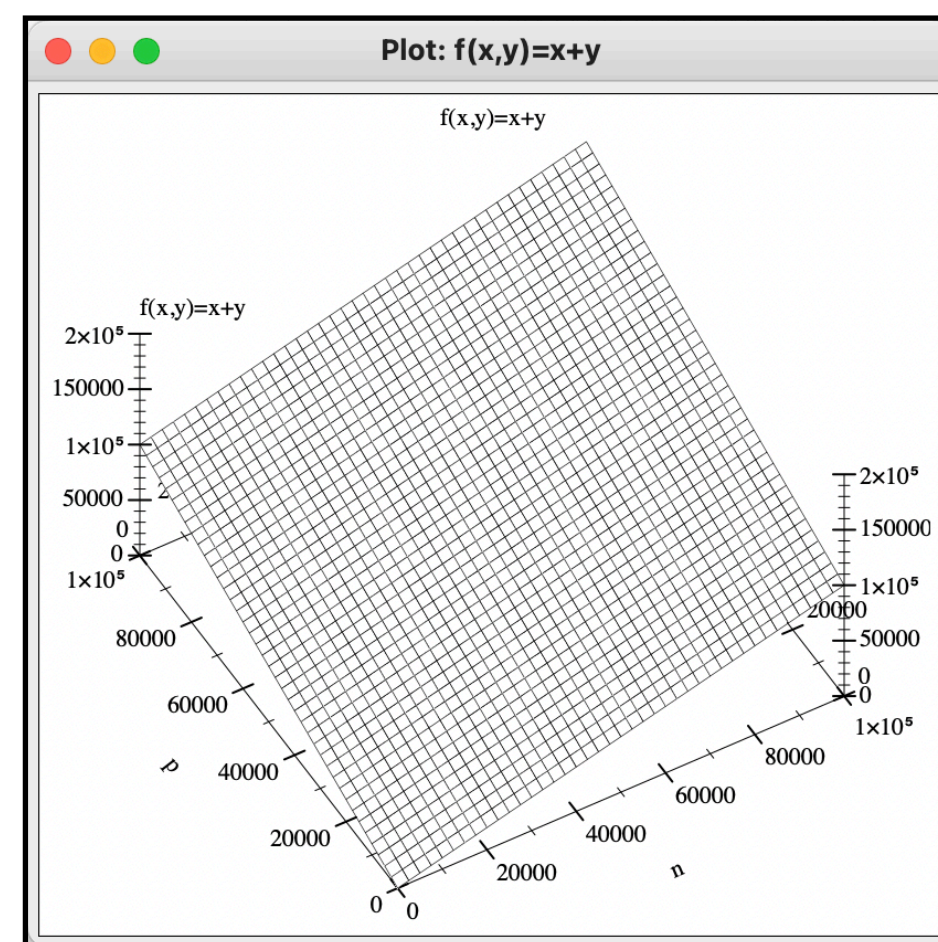
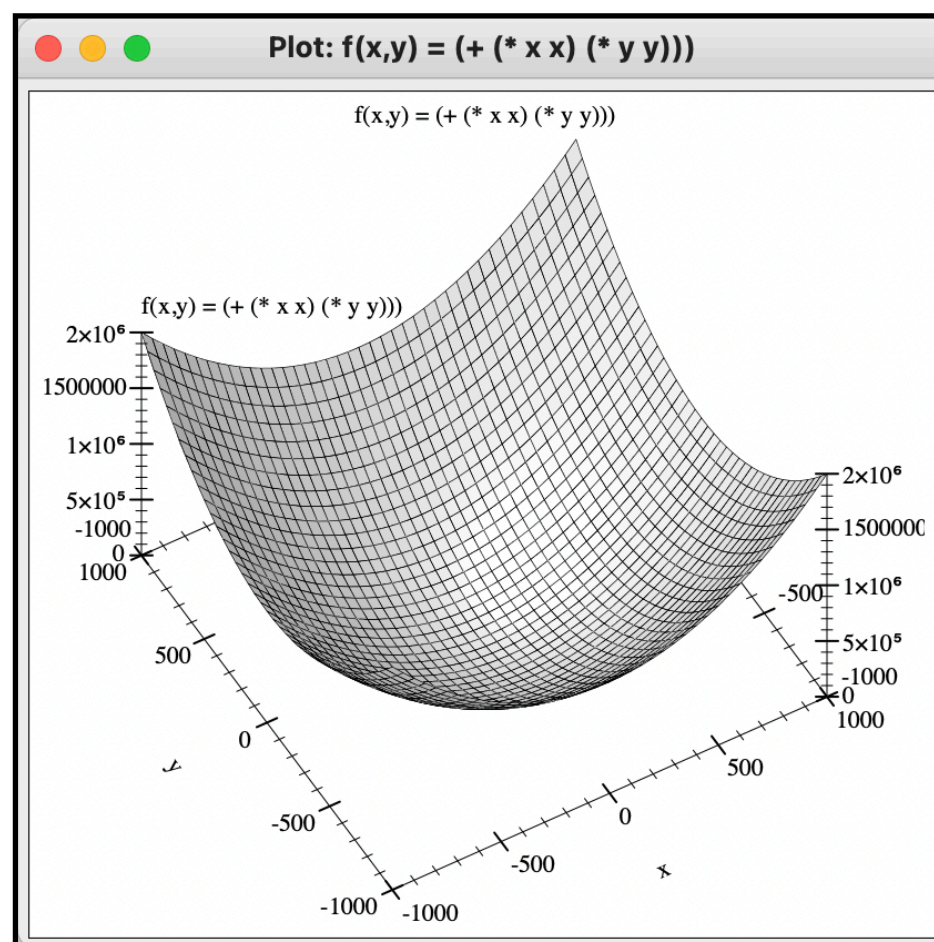


$b_{\text{sum}}(n,m) \in O(1)$
 $r_{\text{sum}}(n,m) \in O(n)$
Dus:
 $f_{\text{sum}}(n,m) \in O(n)$

$b_{\text{times}}(n,m) \in O(n)$
 $r_{\text{times}}(n,m) \in O(m)$
Dus:
 $f_{\text{times}}(n,m) \in O(n.m)$

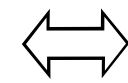
n noch m zijn constanten!

Dus de performantiekarakteristiek van een procedure van meerdere argumenten zal afhangen van elk van die argumenten.



Addendum #3: Geneste Lussen

```
(define (sum-three-to-the n)
  (define (power k)
    (if (= k 0)
        1
        (* 3 (power (- k 1)))))
  (if (= n 0)
      1
      (+ (power n)
         (sum-three-to-the (- n 1)))))
```



```
(define (sum-three-to-the n)
  (let (outer-loop ((n n))
    (if (= n 0)
        1
        (+ (let (inner-loop ((k n))
              (if (= k 0)
                  1
                  (* 3 (inner-loop (- k 1)))))
            (outer-loop (- n 1)))))))
```

$b_{\text{sum-three-to-the}}(n) = n$
 $b_{\text{sum-three-to-the}}(n-1) = n-1$
 ...

Algemeen:

$b_{\text{sum-three-to-the}}(i) = i$

Alles samen:

$$\begin{aligned}
 O\left(\sum_{i=0}^n b(i)\right) &= O\left(\sum_{i=0}^n i\right) \\
 &= O\left(\sum_{i=1}^n i\right) \\
 &= O\left(\frac{n(n+1)}{2}\right) \\
 &= O(n^2)
 \end{aligned}$$

Vuistregel (versie 2): Voor een recursieve Scheme procedure bepaal je $O(r(n))$ voor het aantal recursieve oproepen en $O(b(i))$ voor de i 'de keer dat de body wordt uitgevoerd. De volledige procedure is dan

$$O\left(\sum_{i=1}^{r(n)} b(i)\right)$$

De eerste versie is een speciaal geval: $b(i) = b(n) \forall i$

$$\text{Dus: } \sum_{i=1}^{r(n)} b(i) = \sum_{i=1}^{r(n)} b(n) = b(n) \sum_{i=1}^{r(n)} 1 = b(n)r(n)$$

Analyse van Geheugenverbruik

$O(1)$: vast aantal cellen
onafhankelijk van n

- De meeste algoritmen uit deze cursus zijn “*in-place*”: ze consumeren geen extra geheugen naast het geheugen ingenomen door de invoer.
- Als algoritmen *tóch* extra geheugen vragen, meten we dat op dezelfde manier als tijd: Θ , Ω en O

Uitgedrukt in “*cellen*”. Een paar heeft 2 cellen, een vector van grootte n heeft n cellen. Iedere variabele is ook een cel. Het aantal cellen van een record is het aantal velden van dat record

```
(define (fib1 n)
  (if (< n 2)
      1
      (+ (fib1 (- n 1)) (fib1 (- n 2)))))
```

$\text{fib1} \in \Theta(n)$

recursief
process

```
(define (fib2 n)
  (define (iter n a b)
    (if (= n 0)
        a
        (iter (- n 1) b (+ a b))))
  (iter n 0 1))
```

$\text{fib2} \in \Theta(1)$

iteratief
process

Hoofdstuk 1

1.1 Terminologie

1.1.1 Data en Dataconstructoren

1.1.2 Algoritmen en Algoritmische Constructoren

1.2 Abstractie

1.2.1 Procedurele Abstractie en Proceduretypes

1.2.2 Data abstractie en ADT's

1.2.3 ADTs implementeren in Scheme

1.2.4 Abstractie vs. Genericiteit

1.2.5 Het Dictionary ADT

1.3 Performantie

1.3.1 Het meten van snelheid

1.3.2 Grote O, Ω en Θ

1.3.3 Grote O en Scheme Expressies

1.3.4 Grote O en Scheme Procedures

1.3.5 Het meten van geheugenverbruik

