

# Les 3: Procedures en Processen

---

Sessie 1

# Les 3: Procedures en Processen

---

Procedures zijn patronen of voorschriften die beschrijven hoe een 'rekenproces' moet verlopen. In deze les worden de processen bestudeerd die door simpele procedures beschreven zijn. De manier waarop deze processen tijd en ruimte gaan consumeren krijgt aandacht.

# Terminologie

optredend of gedefinieerd  
als onderdeel van zichzelf  
(direct of indirect)

herhalend (herhaling  
van onderdelen van  
een proces zij het met  
instelbare veranderlijke  
waarden)

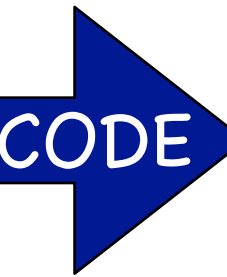
recursief proces  
iteratief proces  
staartrecursie  
lineair proces  
boomrecursie  
orde van groei

recursive process  
iterative process  
tail recursion  
linear process  
tree recursion  
order of growth

# Soms vertaalt een definitie rechtstreeks naar Scheme code

Inductieve definitie van faculteit

$$\begin{cases} 0! = 1 \\ n! = n * (n-1)! \text{ als } n > 0 \end{cases}$$



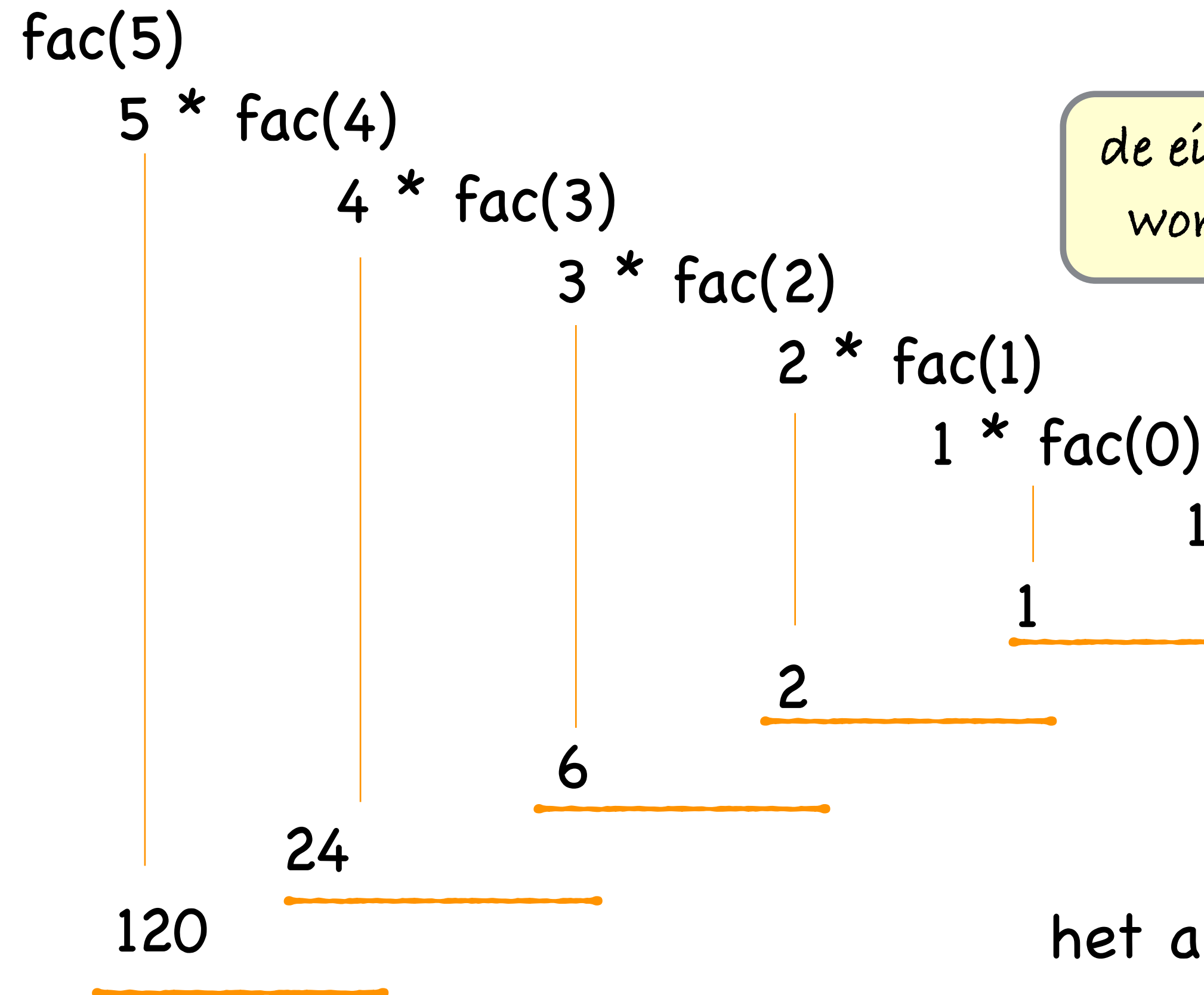
```
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))
```

de functie roept  
zichzelf 1 keer op

> (fac 5)  
120

# Deze definitie levert een lineair recursief proces

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



de eindconditie wordt bereikt

voor dit speciale geval is het resultaat gekend

het eindresultaat wordt opgebouwd tijdens het terugkeren

het aantal oproepen dat nodig is groeit **lineair** t.o.v. de input

# Lineair recursief proces voor faculteit: een trace

```
> (#%require racket/trace)
```

```
> (trace fac)
```

```
> (fac 5)
```

```
> (fac 5)
```

```
> (fac 4)
```

```
> > (fac 3)
```

```
> > (fac 2)
```

```
> > > (fac 1)
```

```
> > > (fac 0)
```

```
< < < 1
```

```
< < < 1
```

```
< < 2
```

```
< < 6
```

```
< 24
```

```
< 120
```

```
120
```

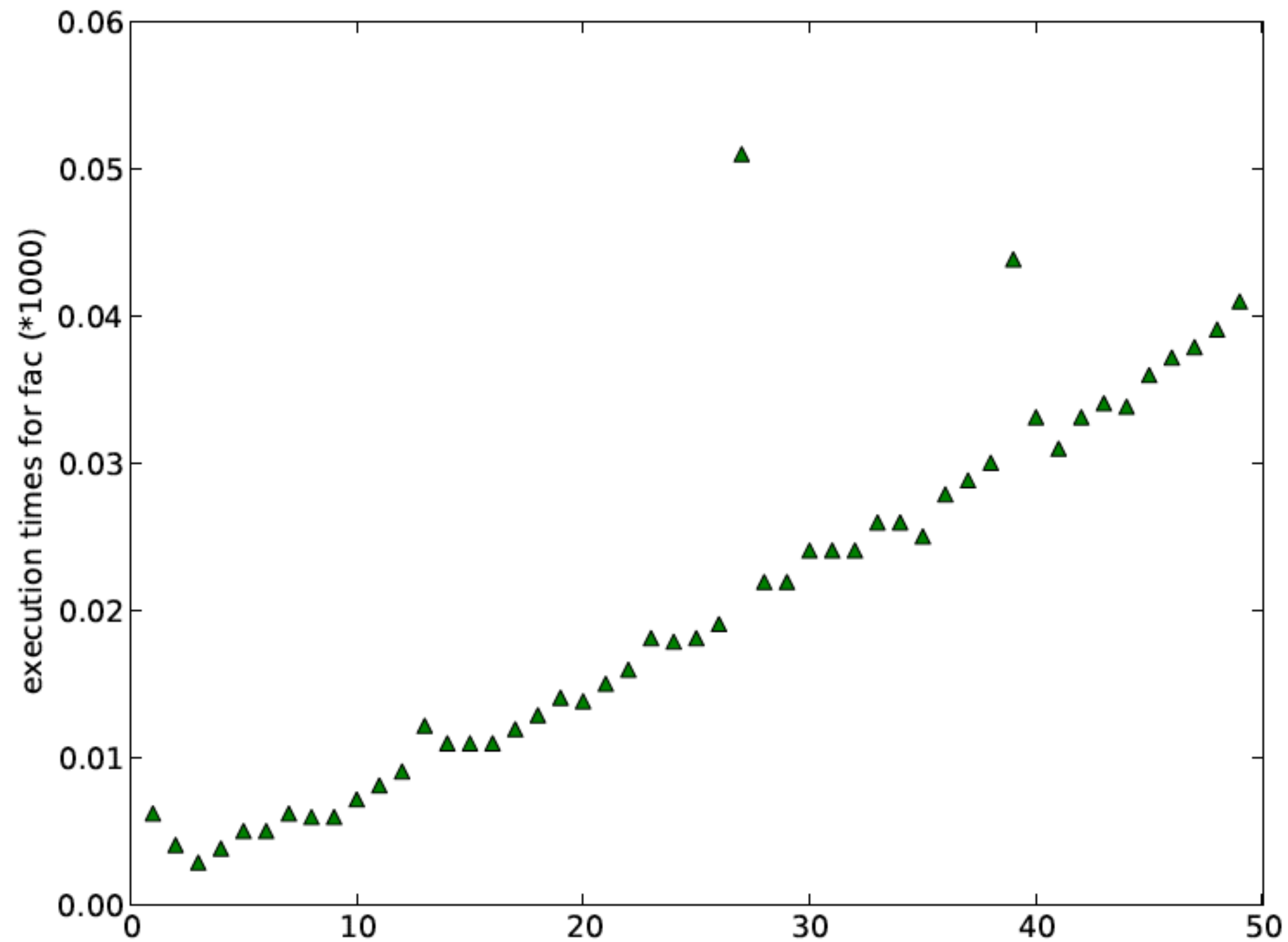
bij elke oproep wordt een  
lijn uitgeprint

bij het terugkeren uit een  
oproep wordt een lijn uitgeprint

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

na terugkeer wordt er nog  
een bewerking uitgevoerd op  
het resultaat (*constructieve  
recursie*)

# Lineair recursief proces voor faculteit: een plot



# Een lineair iteratief proces voor faculteit

fac-bis(5, 1)

~~5~~ \* fac-bis(4, 5)

~~4~~ \* fac-bis(3, 20)

~~3~~ \* fac-bis(2, 60)

~~2~~ \* fac-bis(1, 120)

~~1~~ \* fac-bis(0, 120)  
120

bij elke oproep wordt  
een tussenresultaat  
opgebouwd

de eindconditie  
wordt bereikt

het eindresultaat  
staat klaar

na terugkeer wordt  
er 'NIETS' meer  
gedaan met het  
resultaat  
(= staartrecursie)

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

> (fac 5)  
120

CODE



# Lineair iteratief proces voor faculteit – 2 versies

```
(define (fac n)
  (define (fac-iter counter result)
    (if (> counter n)
        result
        (fac-iter (+ counter 1) (* counter result))))
  (fac-iter 1 1))
```

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

fac-iter(1,1)

fac-iter(2,1)

fac-iter(3,2)

fac-iter(4,6)

fac-iter(5,24)

fac-iter(6,120)

120

idem maar  
telt van 1  
naar n

fac-iter(5,1)

fac-iter(4,5)

fac-iter(3,20)

fac-iter(2,60)

fac-iter(1,120)

fac-iter(0,120)

120

als de eindconditie  
wordt bereikt staat  
het resultaat klaar

# Lineair iteratief proces voor faculteit – traces

```
(define (fac n)
  (define (fac-iter counter result)
    (if (> counter n)
        result
        (fac-iter (+ counter 1) (* counter result))))
  (trace fac-iter)
  (fac-iter 1 1))
```

```
> (fac 5)
>(fac-iter 1 1)
>(fac-iter 2 1)
>(fac-iter 3 2)
>(fac-iter 4 6)
>(fac-iter 5 24)
>(fac-iter 6 120)
<120
120
```

optellen van 1  
tot  $n+1$

```
(define (fac n)
  (define (fac-iter counter result)
    (if (= counter 0)
        result
        (fac-iter (- counter 1) (* counter result))))
  (trace fac-iter)
  (fac-iter n 1))
```

```
> (fac 5)
>(fac-iter 5 1)
>(fac-iter 4 5)
>(fac-iter 3 20)
>(fac-iter 2 60)
>(fac-iter 1 120)
>(fac-iter 0 120)
<120
120
```

terugtellen van  
 $n$  naar 0

de **toestand** van het proces wordt in 2 variabelen gevat !

# Lineaire Recursie $\leftrightarrow$ Iteratie

## Lineair Recursief Process

- ✓ aantal stappen groeit lineair met de 'grootte' van de input
- ✓ geheugengebruik groeit ook lineair met de 'grootte' van de input
- ✓ wordt gegenereerd door een functiedefinitie die constructieve recursie gebruikt

kan vrij mechanisch omgezet worden in een

## Lineair Iteratief Process

- ✓ aantal stappen groeit lineair met de 'grootte' van de input
- ✓ geheugengebruik is constant
- ✓ wordt gegenereerd door een functiedefinitie die staartrecursie gebruikt OF door het gebruik van een iteratieve controlestructuur

“Een **iteratief proces** is een proces waarvan de toestand op elk moment kan gevat worden in een op voorhand gekend eindig aantal variabelen”

# Deze definitie vertaalt ook rechtstreeks naar Scheme code

## Alternatieve definitie van faculteit

$n! = n * (n-1) * (n-2) * \dots * 2 * 1$

or

$n! = 1 * 2 * 3 * \dots * (n-2) * (n-1) * n$

```
(define (fac n)
  (do ((counter 1 (+ counter 1))
      (result 1 (* result counter)))
      ((> counter n) result)))
```

```
> (fac 5)
120
```

```
(define (fac n)
  (do ((counter n (- counter 1))
      (result 1 (* result counter)))
      ((= counter 0) result)))
```

```
> (fac 5)
120
```

# Do special form

```
(define (fac n)
  (do ((counter 1 (+ counter 1))
      (result 1 (* result counter)))
      ((> counter n) result)))
```

```
(do ( (<name-1> <init-1> <next-1>)
      (<name-2> <init-2> <next-2>)
      . . .
      (<name-n> <init-n> <next-n>) )
  (<end-test> <result-value>)
  <body-expr-1>
  . . .
  <body-expr-n>)
```

# Do is een iteratieve controle structuur

```
(define (fac n)
  (do ((counter n (- counter 1))
      (result 1 (* result counter)))
      ((= counter 0) result)
      (display counter)
      (display " ")
      (display result)
      (newline)))
```

```
> (fac 5)
5 1
4 5
3 20
2 60
1 120
120
```

de **toestand** van het  
proces wordt in 2  
variabelen gevat

# Vergelijk met vorige oplossing!

```
(define (fac n)
  (do ((counter n (- counter 1))
      (result 1 (* result counter)))
      ((= counter 0) result)
      (display counter)
      (display " ")
      (display result)
      (newline))))
```

```
> (fac 5)
5 1
4 5
3 20
2 60
1 120
120
```

```
(define (fac n)
  (define (fac-iter counter result)
    (if (= counter 0)
        result
        (fac-iter (- counter 1) (* counter result))))
  (trace fac-iter)
  (fac-iter n 1))
```

```
> (fac 5)
>(fac-iter 5 1)
>(fac-iter 4 5)
>(fac-iter 3 20)
>(fac-iter 2 60)
>(fac-iter 1 120)
>(fac-iter 0 120)
<120
120
```

# Les 3: Procedures en Processen


---

Sessie 2



# Ook de definitie van Fibonacci vertaalt makkelijk naar Scheme

## Recursive definition of Fibonacci

  $\text{fib}(0) = 0$   
 $\text{fib}(1) = 1$   
 $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \quad n > 1$

```
(define (fib n)
  (cond
    ((= n 0) 0)
    ((= n 1) 1)
    (else (+ (fib (- n 1)) (fib (- n 2))))))
```

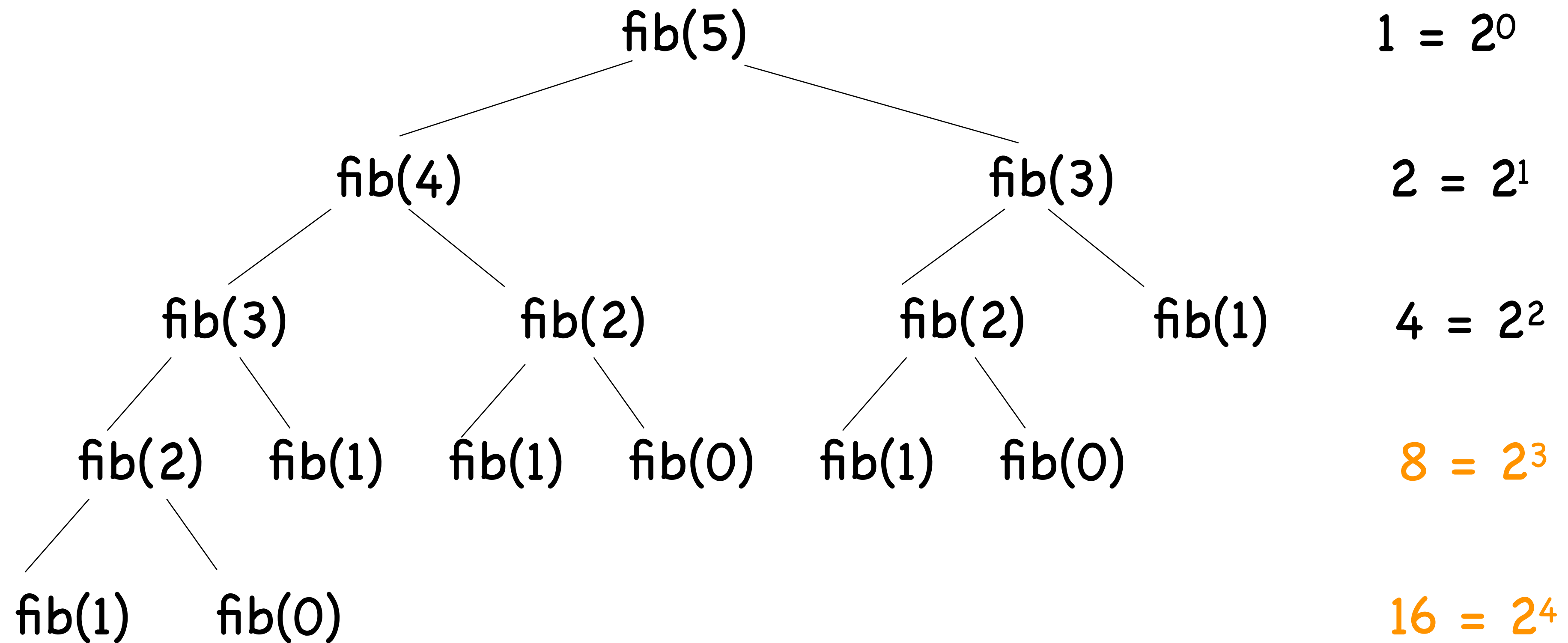
EN de resultaten  
worden gecombineerd

de functie roept  
zichzelf 2 keer op

```
> (fib 5)
5
> (fib 7)
13
```

CODE

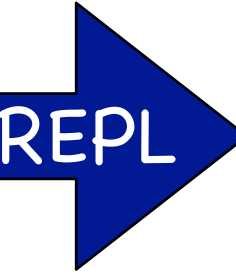
# Deze definitie levert een boomrecursief proces



in elk blad is een  
eindconditie bereikt

het aantal oproepen dat nodig is  
groeit **exponentieel** t.o.v. de input

# Boom recursief proces voor Fibonacci getallen: een trace



```
> (fib 5)
>(fib 5)
> (fib 4)
> >(fib 3)
> > (fib 2)
> > >(fib 1)
< < <1
> > >(fib 0)
< < <0
< < 1
> > (fib 1)
< < 1
< <2
> >(fib 2)
> > (fib 1)
< < 1
> > (fib 0)
< < 0
< <1
< 3

> (fib 3)
> >(fib 2)
> > (fib 1)
< < 1
> > (fib 0)
< < 0
< <1
> >(fib 1)
< <1
< 2
<5
5
>
```

An orange arrow originates from the bottom of the left trace, specifically from the '< 3' line, and points diagonally upwards and to the right, ending at the '> (fib 3)' line of the right trace.

# Boom recursief proces voor Fibonacci getallen: een trace

```
> (fib 7)
>(fib 7)
> (fib 6)
> >(fib 5)
> > (fib 4)
> > >(fib 3)
> > > (fib 2)
> > > >(fib 1)
<<<<1
> > > >(fib 0)
<<<<0
<<<<1
<<<<1
> > > (fib 1)
<<<<1
<<<<2

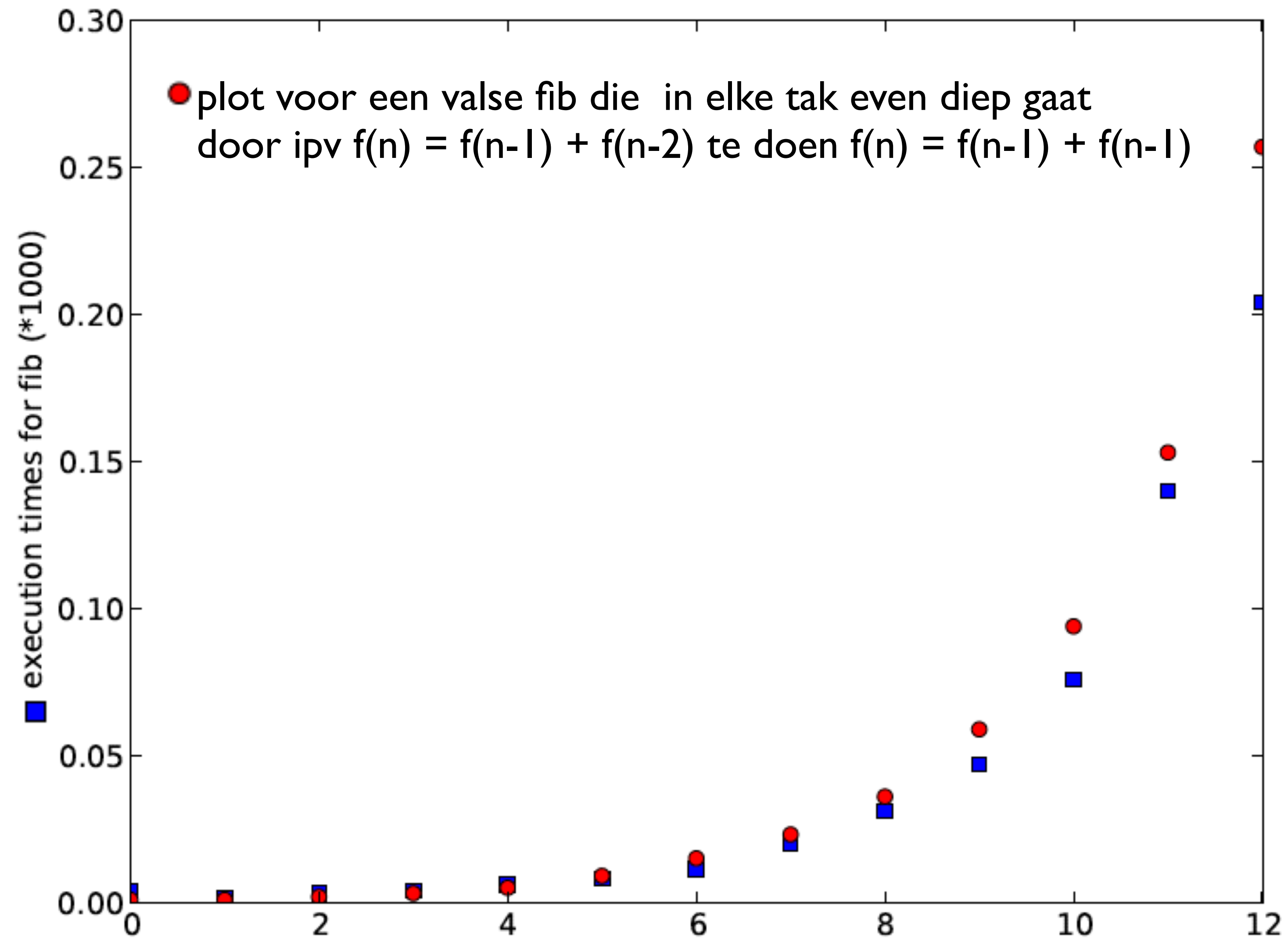
> > >(fib 2)
> > > (fib 1)
<<<<1
> > > (fib 0)
<<<<0
<<<<1
> > >(fib 1)
<<<<1
<<<<2
<<<<5

> > >(fib 4)
> > (fib 3)
> > >(fib 2)
> > > (fib 1)
<<<<1
> > > (fib 0)
<<<<0
<<<<1
<<<<1
> > >(fib 1)
<<<<1
<<<<2
> > (fib 2)
> > >(fib 1)
<<<<1
> > >(fib 0)
<<<<0
<<<<1
<<<<3
<<<<8

> (fib 5)
> >(fib 4)
> > (fib 3)
> > >(fib 2)
> > > (fib 1)
<<<<1
> > > (fib 0)
<<<<0
<<<<1
<<<<0
<<<<1
> > >(fib 1)
<<<<1
<<<<2
> > (fib 2)
> > >(fib 1)
<<<<1
> > >(fib 0)
<<<<0
<<<<1
<<<<1
<<<<3
> >

<<<3
< 8
> (fib 5)
> >(fib 4)
> > (fib 3)
> > >(fib 2)
> > > (fib 1)
<<<<1
> > > (fib 0)
<<<<0
> > >(fib 1)
<<<<1
> > >(fib 0)
<<<<0
> > (fib 2)
> > >(fib 1)
<<<<1
> > >(fib 0)
<<<<0
> > (fib 1)
<<<<1
<<<<2
< 5
<13
13
>
```

# Boom recursief proces voor Fibonacci getallen: een plot



# Lineair recursief proces voor machtsverheffing

Inductieve definitie voor machtsverheffing

$$\left\{ \begin{array}{l} b^0 = 1 \\ b^n = b * b^{(n-1)} \quad n > 0 \end{array} \right.$$

```
(define (exp b n)
  (if (= n 0)
      1
      (* b (exp b (- n 1)))))
```

```
> (exp 2 5)
32
```

```
> (trace exp)
> (exp 2 5)
>(exp 2 5)
> (exp 2 4)
> >(exp 2 3)
> > (exp 2 2)
> > >(exp 2 1)
> > > (exp 2 0)
< < < 1
< < < 2
< < 4
< < 8
< 16
< 32
32
```

CODE

# Lineair iteratieve processen voor machtsverheffing

```
(define (exp b n)
  (define (exp-iter counter result)
    (if (> counter n)
        result
        (exp-iter (+ counter 1) (* result b))))
  (exp-iter 1 1))
```

```
> (exp 2 5)
32
```

```
(define (exp b n)
  (do ((counter 1 (+ counter 1))
      (result 1 (* result b)))
      ((> counter n) result)))
```

```
> (exp 2 5)
32
```

# Logaritmisch proces voor machtsverheffing

Alternatieve inductieve definitie voor machtsverheffing

$$\left\{ \begin{array}{ll} b^0 = 1 \\ b^n = b * b^{(n-1)} & n \text{ oneven, } n > 0 \\ b^n = (b^{(n/2)})^2 & n \text{ even, } n > 0 \end{array} \right.$$

```
(define (fast-exp b n)
  (cond
    ((= n 0) 1)
    ((even? n) (square (fast-exp b (/ n 2))))
    (else (* b (fast-exp b (- n 1)))))
```

```
> (fast-exp 2 5)
32
```

```
> (fast-exp 2 20)
>(fast-exp 2 20)
> (fast-exp 2 10)
> >(fast-exp 2 5)
> > (fast-exp 2 4)
> > >(fast-exp 2 2)
> > > (fast-exp 2 1)
> > > >(fast-exp 2 0)
< < < <1
< < < 2
< < <4
< < 16
< <32
< 1024
<1048576
1048576
```

CODE



# Logaritmisch proces voor machtsverheffing: Traces

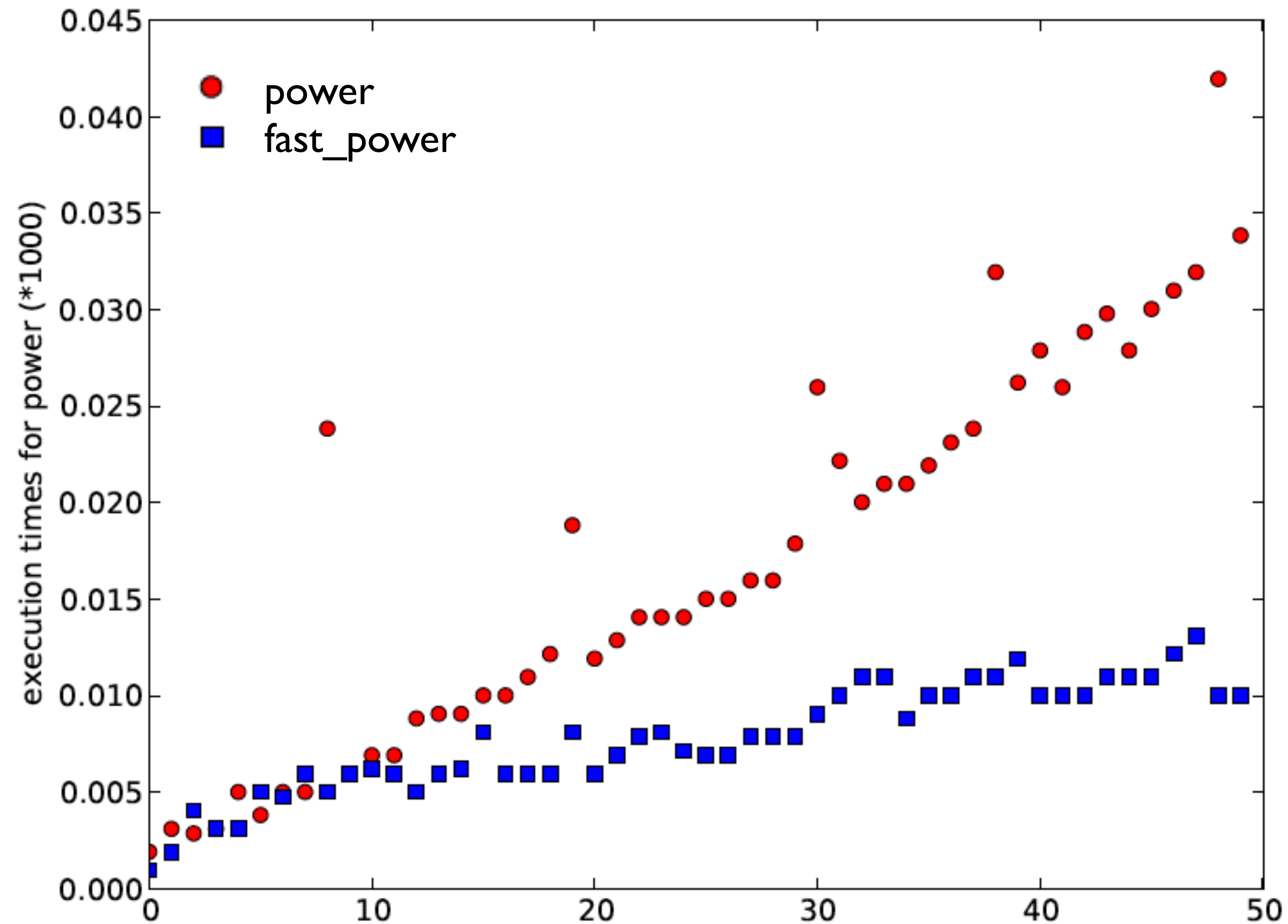
```
> (fast-exp 2 7)
>(fast-exp 2 7)
> (fast-exp 2 6)
> >(fast-exp 2 3)
> > (fast-exp 2 2)
> > >(fast-exp 2 1)
> > > (fast-exp 2 0)
< < < 1
< < <2
< < 4
< <8
< 64
<128
128
```

```
> (fast-exp 2 20)
>(fast-exp 2 20)
> (fast-exp 2 10)
> >(fast-exp 2 5)
> > (fast-exp 2 4)
> > >(fast-exp 2 2)
> > > (fast-exp 2 1)
> > > >(fast-exp 2 0)
< < < <1
< < < 2
< < <4
< < 16
< <32
< 1024
<1048576
1048576
```

```
> (fast-exp 2 70)
>(fast-exp 2 70)
> (fast-exp 2 35)
> >(fast-exp 2 34)
> > (fast-exp 2 17)
> > >(fast-exp 2 16)
> > > (fast-exp 2 8)
> > > >(fast-exp 2 4)
> > > > (fast-exp 2 2)
> > > > >(fast-exp 2 1)
> > > > > (fast-exp 2 0)
< < < < < 1
< < < < <2
< < < < 4
< < < <16
< < < 256
< < <65536
< < 131072
< <17179869184
< 34359738368
<1180591620717411303424
1180591620717411303424
```

het aantal oproepen dat nodig is  
groeit **logaritmisch** met de input

# Plot voor power en fast-power



# Les 4: Hogere orde procedures

---