Les 1 30/09

PRINT

INPUT - OUTPUT

```
import math

a=5
b=1
c=-2

discriminant=b**2-4*a*c

x1=(-b+math.sqrt(discriminant))/(2*a)
x2=(-b-math.sqrt(discriminant))/(2*a)

print("eerstenulpuntis:",x1)
print("tweedenulpuntis:",x2)
```

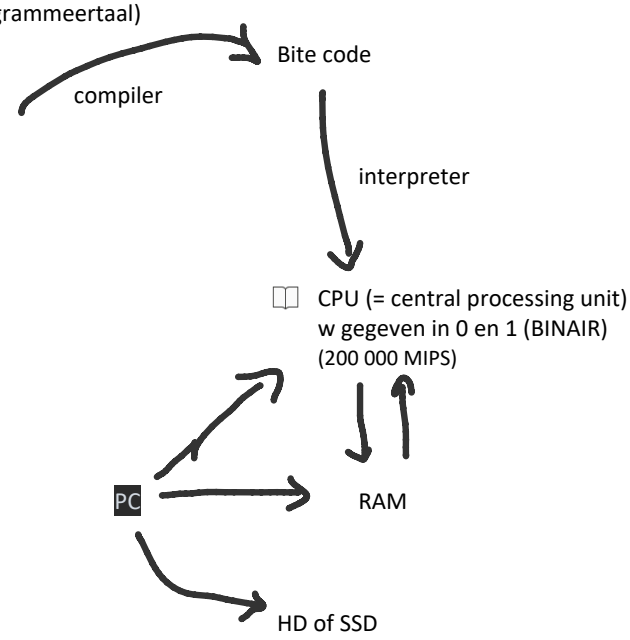
Veranderen met een input

b*b = b**2

HPL (= hoge programmeertaal)

Vb. Python

```
print ("hello")
...
...
...
...
```



a = 5: voor de waarde van a w er een plaats vrijgehouden in het geheugen, deze plaats bevat nu de waarde 5. a kan nu gebruikt w als variabele in een berekening
= toekenningsopdracht

Gehele deling = de deling zonder de rest (steeds een natuurlijk getal)

Gehele deling a en b: a//b

Rest gehele deling a en b: a%b

Overige commando's

```
Input()
Int()
Float()
String()
```

Integer = gehele getallen
Float = rationale getallen
String = tekst

★ Veel gemaakte programmafouten

Syntaxfouten = fouten spelling van python
Uitvoeringsfouten = fouten tijdens draaien programma
Logische fout = fout in opbouw programma

Les 2 07/10

07 October 2021 13:59

IF - ELSE WHILE

★ Relationele operatoren in Python

Table 1 Relational Operators

Python	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

"=" kent een waarde toe
"==" kijkt een waarde na
Gebruikt in if of while constructies

Vergelijking van strings: volgorde en hiërarchie!
All UPPERCASE letters come before lowercase
'space' comes before all other printable characters
Digits (0-9) come before all letters
See Appendix A for the Basic Latin (ASCII) Subset of Unicode

Aparte commando's string en number. Zie HB!

Datatypes:
String
Float
Boolean
Twee waarden: true, false
Boolean variabelen: failed = true/false
Operatoren: and, or, not (denk aan waardetabellen!)

★ Boolean variabels

AND

```
if temp > 0 and temp < 100 :
   print("Liquid")
```

OR

```
if temp <= 0 or temp >= 100 :
   print("Not liquid")
```

NOT

```
if not attending or grade < 60 :
   print("Drop?")
```

```
if attending and not(grade < 60) :
   print("Stay")
```

--> betere versie

```
if attending and grade >= 60 :
   print("Stay")
```

Geneste commando's
Vb.: programma discriminant (uitgebreid, les 2)
Vb.: schaal van richter (let op de verschillende interpretaties!)

```
if richter >= 8.0 :   # Handle the 'special case' first
   print("Most structures fall")
else:
   if richter >= 7.0 :
      print("Many buildings destroyed")
   else:
      if richter >= 6.0 :
         print("Many buildings damaged, some collapse")
      else:
         if richter >= 4.5 :
            print("Damage to poorly constructed buildings")
         else :    # so that the 'general case' can be
  handled last
            print("No destruction of buildings")
```

=

```
if richter >= 8.0 :   # Handle the 'special case' first
   print("Most structures fall")
elif richter >= 7.0 :
   print("Many buildings destroyed")
elif richter >= 6.0 :
   print("Many buildings damaged, some collapse")
elif richter >= 4.5 :
   print("Damage to poorly constructed buildings")
else :    # so that the 'general case' can be handled last
   print("No destruction of buildings")
```

!=

```
if richter >= 8.0 :
   print("Most structures fall")
if richter >= 7.0 :
   print("Many buildings destroyed")
if richter >= 6.0 :
   print("Many buildings damaged, some collapse")
if richter >= 4.5 :
   print("Damage to poorly constructed buildings")
```

Vb. Count-controlled loops: Som 1 tot n berekenen = (n*(n+1))/2

Count-controlled loops

= while-loop onder invloed van een telopdracht

```
counter = 1            # Initialize the counter
while counter <= 10 :     # Check the counter
   print(counter)
   counter = counter + 1  # Update the loop variable
print(”----”)
```

Event-controlled loops

= while-loop onder invloed van een voorwaarde

```
##
#  This program computes the time required to double an investment.
#

# Create constant variables.
RATE = 5.0
INITIAL_BALANCE = 10000.0
TARGET = 2 * INITIAL_BALANCE

# Initialize variables used with the loop.
balance = INITIAL_BALANCE
year = 0

# Count the years required for the investment to double.
while balance < TARGET :
   year = year + 1
   interest = balance * RATE / 100
   balance = balance + interest

# Print the results.
print("The investment doubled after", year, "years.")
```

Veel gemaakte programmafouten

Foute voorwaarde, vb.: >= ipv >
Oneindige loops
Loop die maar 1x mee gaat

Sentinel values

Gebruikt in een while-loop die een reeks getallen leest en controleert. Er is geen duidelijkheid over wanneer de while-loop moet stoppen, dus voordat deze start wordt er een waarde ingesteld zodat de loop deze leest als 'einde loop'.

Opm.: in het algemeen wordt de waarde -1 hiervoor gebruikt

While getal != -1:	Start je commando op deze manier om de sentinel variabe;le te installeren

Vb.: berekening gemiddeld salaris

```
# initialisation
total = 0.0
count = 0
# Initial read -  read the first value
salary = float(input("Enter a salary or -1 to
```

0	vraag getal n
1	Som = 0
2	Voor elke " " >= 0 and " " <= 0 som <-> som +1
3	Schrijf som uit

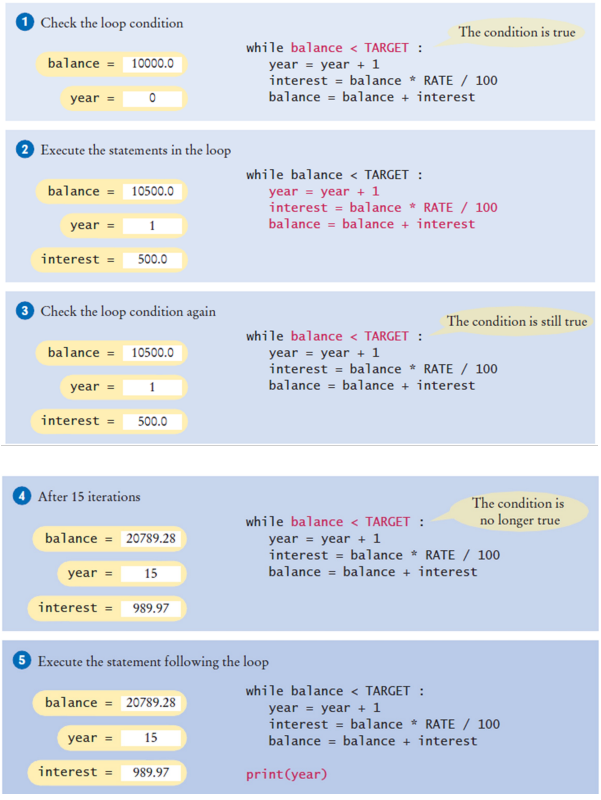
```
n=int(input("geefwardevoorn"))

#startwaarde
som=0
teller=0

while(teller<n):
teller=teller+1
som=som+teller

print("desomis",som)
```

Uitvoering loop voorbeeldprogramma event-controlled loop



```
finish: "))
# We will keep on processing values until a
negative value is input
while salary >= 0.0 :
   total = total + salary
   count = count + 1
   # Next read
   salary = float(input("Enter a salary or -1 to
finish: "))
# Compute an print average salary
if count > 0:
    average = total / count
    print("Average salary is", average)
else:
    print("No data was provided...")
```

Les 3 14/10

14 October 2021 14:08

FOR LISTS TUPLES

For-loop

Lijkt op While-loop
Verkort: geen teller
(als teller w gevraagd in opgave,
gebruik dan wel while!)
Maakt gebruik van CONTAINER

Container = lijst of verz van aantal tekens

Opm.: niet per def een range, wel vaak voorkomend
Vb.: range(x) = 0, 1, 2, …, x-1
For i in range(x) --> print(i)
Vb.: range(2, 10) = 2, 3, …, 9
Vb.: range(2, 10, 3) = 2, 5, 8
Stappen van 3
Opm.: range bevat in zekere zin teller van de while-loop!

Ben ik mee: while veranderen naar for-lus

Voorbeeld:

```
i = 1
while i < 11 :
   print(i)
   i = i + 1
```

while version

```
for i in range(1, 11) :
   print(i)
```

for version

Uitbreiding "print()"
The end="" suppresses the new line,
so the numbers are all printed on the same line

Geziene variabelen: Strings, Integers, Floating points, Booleans
Nieuwe variabelen: lists, tupels, sets, dictionaries, …

Lijst = groep geordende elementen
Values = [x1, x2, …, xn]
First value = values[0] = x1
Opm.: values[n] = ERROR
Length = len(values) = n
Empty list = []
values[n-1] = values[-1] = xn
Opm.: values[-(n+1)] = ERROR

Waarde en index

Opm.: "values" is ook de lijstnaam!

Traversing lists:

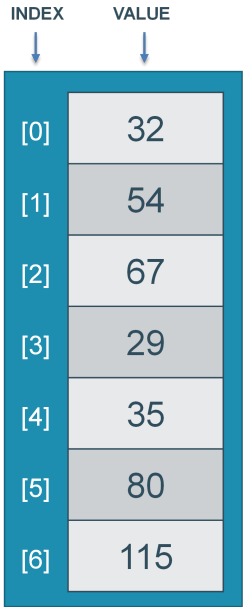
```
For i in values:
    Print(i)

For i in range(n):
    Print(i, values[i])

For i in range(len(values)):
    Print(i, values [i])
```

Voorbeeld:

Values = [32, 54, 67, 29, 35, 80, 115]
First value = values[0] = 32
Values[7] = ERROR
Length = len(values) = 7
Empty list = []
Values[6] = values[-1] = 115
Values[-8] = ERROR



Output = lijst

Manipulaties met lijsten:

Elementen toevoegen:
- Append: achteraan bijvoegen
  - Listname.append(x)
- Insert: vooraan bijvoegen
  - Listname.insert(plaats, x)

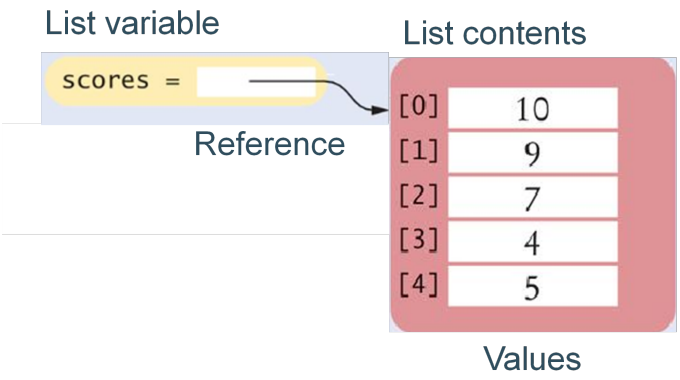
Elementen verwijderen:
- Remove: verwijderd waarde
  - Listname.remove(x)
- Pop: verwijderd laatste waarde
  - Listname.pop()
- '' : waarde bij index
  - Listname.pop(index)
- Clear: maakt lijst leeg
  - Listname.clear()

Info over de lijst:
- Index: index van waarde (eerstvoorkomende als deze herhaald w)
  - Listname.index(x)
- Count: hoe vaak een waarde herhaald w
  - Listname.count(x)
- In: gaat na of waarde voorkomt in lijst (respons true/false)
  - x in list
- Sum: som alle elementen
  - Sum(list)
- Max: grootste element uit lijst
  - Max(list)
- Min: kleinste element uit lijst
  - Min(list)

Transformaties met lijst:
- Reverse: omkeren van lijst
  - Listname.reverse()
- Sort: sorteert de lijst (standaardwaarden of volgens eigen critera)
  - Listname.sort()
- Copy: lijst kopiëren
  - Listname.copy()

Tables
?!

Reference semantics



Opm.: scores = values

```
scores = [10, 9, 7, 4, 5]
values = scores     # Copying list reference
```

```
...
scores[3] = 10
print(values[3])   # Prints 10
```

= aliases

Slicing = lijsten 'snijden' om sublists te creëren

Sublist t [i : j] deel van lijst t bevat i, i+1, …, j-1
- Opm.: de omkering t [-i : -j] is ook mogelijk!

Standaardwaarde voor sublists is index 0 als ondergrens en lengte van de lijst, len(t) als bovengrens
--> geen onder- of bovengrens ingeven snijd kop of staart af
- Te grote of kleine waarde? W gecorrigeerd naar standaardwaarden

Populating = vullen van lijsten

- Vb.: lijst van kwadraten natuurlijke getallen 1-9

Manier 1:
```
squares = []
for x in range(10):
squares.append(x ** 2)
# squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Manier 2:
```
squares = [x ** 2 for x in range(10)]
# squares = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Tuple = geordende lijst, onveranderlijk en enkel-lezen

Omzetting lijst --> tuple
```
L = [1, 2 3]
T = tuple(L)  #(1, 2, 3)
```
Omzetting tuple --> lijst
```
T = (1, 2, 3)
L = list(T)    #[1, 2, 3]
```

Les 4 21/10

21 October 2021 14:07

FUNCTIES

Functies geven structuur aan programma's, bakenen delen af in programma
(leesbaarheid en snellere berekeningen)
Je geeft een naam aan een opvolging van instructies en inputs

Functie heeft steeds een of meerdere inputs/parameters en geeft een resultaat of output terug

Een functie kan ook niets terug geven in Python

Functie definiëren:

Def variabele (parameter 1, parameter 2, …)
+ statements

Vb.:

```
def cube_volume(side_length):
    volume = side_length ** 3
    return volume
```

Gebruik de variabele om de uitkomst van de functie terug te printen

De main-functie
= hoofdfunctie van het programma
Afspraak dat deze functie in elk programma voorkomt, de te printen variabele is ook aan deze functie gekoppeld

Vb.:

```
def main():
    result = cube_volume(2)
    print("A cube with side length 2 has volume", result)

def cube_volume(side_length):
    volume = side_length ** 3
    return volume

main()
```

Returnopdracht
--> gebruik vanaf zekerheid uitkomst

Nut?
- Eindigt functie
- Koppelt eindwaarde terug aan functie

Meerdere returnopdrachten
= mogelijk
1 opdracht per geopende functie

Vb.:

```
def cube_volume(side_length):
    if (side_length < 0):
        return 0
    return side_length ** 3
```

Afslanken van aantal opdrachten door variabele aan return te koppelen

Vb.:

```
def cube_volume(side_length) :
    if side_length >= 0:
        volume = side_length ** 3
    else:
        volume = 0
    return volume
```

Opm.: koppel aan elke 'tak' een return!

Variabelen in functies

Lokale variabele: bestaat enkel binnen de functie
Parametrische variabele: de waarde van het argument in de functie
Vb. dia 24 (met animatie)
Globale variabelen

Opm.: definieer de variabelen duidelijk genoeg, duidelijkheid bij print
gebruik juiste waarden voor de variabelen

Iteratieve functies?

Globale variabelen?

Testen van een functie?
Manueel: run via print
Automatisch: ASSERT commando
= assert + booleaanse expressie
True: niks gebeurt
False: AssertionError w gegenereerd

Assert
https://www.w3schools.com/python/ref_

Stepwise refinement
= afbreken vraagstuk in deelproblemen adhv functies
Nut? oplosbaarheid, leesbaarheid, stapsgewijs testen (en vinden van fouten)

Vb.
Ask the user for a positive integer
and perform input validation using a while-loop, print out all prime numbers that are smaller or equal to the input number.

Assert
https://www.w3schools.com/python/ref_keyword_assert.asp

Ask the user for a positive integer
and perform input validation using a while-loop, print out all prime numbers that are smaller or equal to the input number.

```
def can_be_divided_by (n, d):
  return n % d == 0

def is_prime_number (n)
  ...
def ask_number_with_input_validation ():
  ...
def main ():
  ...
main()
```

Recursieve functies
= functie die zichzelf terug oproept
Stop inbouwen voor functie (w speciaal geval van uitkomst vorige uitvoering functie)

Vb.: Fibonacci

Iteratief:

```
def fibonacci (number) :
  if number == 0:
    return 0
  elif number == 1:
    return 1
  else:
    fib_min_2 = 1
    fib_min_1 = 1
    fib = fib_min_1 + fib_min_2
    for i in range(number-2):
      fib = fib_min_1 + fib_min_2
      fib_min_2, fib_min_1 = fib_min_1, fib
    return fib
```

Recursief:

```
def fibonacci(n):
  if number == 0:
    return 0
  elif number == 1:
    return 1
  else:
    fib = fibonacci (n - 1)
        + fibonacci (n - 2)
    return fib
```

Trager in verwerking, maar eleganter om te lezen en te begrijpen

Les 5 28/10

28 October 2021 23:11

CORRECTHEIDSBEWIJZEN
= het nagaan of een stuk code al dan niet correct is (zonder run)

Specificatie
= beschrijving **wat** een stuk code moet doen
'een programma is correct als het voldoet aan zijn specificatie'

Pre-conditie: conditie waaronder het programma moet werken (bv met integers)

Post-conditie: conditie waar het programma op het einde aan moet voldoen (m.a.w. het resultaat)

Vb.:
"Schrijf een programma dat voor een natuurlijk getal n,
de faculteit van n berekent."

Pre-conditie:	n is een geheel getal, groter dan 0
Post-conditie:	de variabele fac bevat de faculteit van n, dus n !

Implementatie
= beschrijving **hoe** een stuk code moet evalueren

Partiële vs. Totale correctheid

{P}

while E:
p

{Q}

De regel voor partiële correctheid kan gebruikt worden om te verifiëren dat,
als de lust eindigt,
de uitvoering eindigt waarbij aan eindtoestand {Q} is voldaan.
--> lusINvariant

Om de totale correctheid van een lus te verifiëren moeten we aantonen
dat de lus eindigt.
--> lusVARIANT
Opm.: w opgebouwd uit een partieel bewijs

```
assert 0 <= n  and  i == 0  and  som == 0
assert som == sum (range (i + 1))

while i != n:
    oude_variant = n - i
    assert (som == sum (range (i+1)) ) and i != n          and n-i == oude_variant
                                                           #OK
    assert som + (i+1) == sum (range (i+2))       and 0 <= n-(i+1) < oude_variant
    i = i + 1
    assert som + i == sum (range (i+1))                and 0 <= n-i < oude_variant
    som = som + i
    assert som == sum (range (i + 1))                  and 0 <= n-i < oude_variant

assert ( som == sum (range (i + 1)) ) and i == n
assert som == sum ( range(n+1) )    # som == som van 0 t.e.m. n
```

Les 6 04/11

SETS DICTIONAIRIES IIST

06 November 2021 23:56

List = [1, 2, 3, 4]
Set = {1, 2, 3, 4}
Dict = {1: Miet, 2: Staf, 3: Sooi, 4: Melanie}

Set = verzameling
'Container' of datastructuur in geheugen die datatypes (float, int, string) bevat

Elementen?
- Ongeordend
- Uniek (geen herhaling van elementen)

Aanmaken:
Lege set: my_set = set()
Gevulde set: my_set = {e1, e2, …}

Operaties met sets
- Additie
  .add(e)
  .update([e1, e2, …])
- Eliminatie
  .remove(e)
    ERROR als element e niet in lijst
  .discard(e)
    Geen reactie als e niet in lijst
  .clear()
- Iteratie
  For element in list:
    Print(element)
- Sorteren
  Sorted(set)
- Samenvoegen
  My_set = one_set.union(other_set)

List vs Set

List: volgorde en # voorkomens relevant
Set: volgorde en # voorkomens niet relevant, werkt sneller dan list

Operaties met sets

Table 1 Common Set Operations

Operation	Description

`s = set()`
`s = set(seq)`
`s = {e1, e2, ..., en}`
	Creates a new set that is either empty, a duplicate copy of sequence *seq*, or that contains the initial elements provided.

`len(s)`	Returns the number of elements in set *s*.
element `in s` *element* `not in s`	Determines if *element* is in the set.
`s.add(element)`	Adds a new element to the set. If the element is already in the set, no action is taken.
`s.discard(element)` `s.remove(element)`	Removes an element from the set. If the element is not a member of the set, `discard` has no effect, but `remove` will raise an exception.
`s.clear()`	Removes all elements from a set.
`s.issubset(t)`	Returns a Boolean indicating whether set *s* is a subset of set *t*.
`s == t` `s != t`	Returns a Boolean indicating whether set *s* is equal to set *t*.
`s.union(t)`	Returns a new set that contains all elements in set *s* and set *t*.
`s.intersection(t)`	Returns a new set that contains elements that are in *both* set *s* and set *t*.
`s.difference(t)`	Returns a new set that contains elements in *s* that are not in set *t*.

Dictionaries = woordenboeken
'container' of datastructuur die unieke sleutels aan een (niet-unieke) waarde koppelt. Deze sleutels worden op hun beurt opgeslagen als een set.

Aanmaken:

Lege dict: my_dict = {}
Gevulde dict: m_dict = {k1: e1, k2: e2, …}

Operaties met dict

- Additie
  dict[k1] = v
  k1 in dict? Aanpassing naar v
  k1 niet in dict? Toevoeging v
- Eliminatie
  del dict[k1]
  dict.clear()
- Iteratie
  dict[k1]
  ERROR als k1 niet bestaat
  dict.get(k1)
  Geen reactie als k1 niet bestaat

Operaties met dict

Table 2 Common Dictionary Operations

Operation	Returns
`d = dict()` `d = dict(c)`	Creates a new empty dictionary or a duplicate copy of dictionary *c*.
`d = {}` `d = {k1: v1, k2: v2, ..., kn: vn}`	Creates a new empty dictionary or a dictionary that contains the initial items provided. Each item consists of a key (*k*) and a value (*v*) separated by a colon.
`len(d)`	Returns the number of items in dictionary *d*.
key `in` *d* *key* `not in` *d*	Determines if the key is in the dictionary.
`d[key] = value`	Adds a new *key*/*value* item to the dictionary if the *key* does not exist. If the key does exist, it modifies the value associated with the key.
`x = d[key]`	Returns the value associated with the given key. The key must exist or an exception is raised.
`d.get(key, default)`	Returns the value associated with the given key, or the default value if the key is not present.
`d.pop(key)`	Removes the key and its associated value from the dictionary that contains the given key or raises an exception if the key is not present.
`d.values()`	Returns a sequence containing all values of the dictionary.

Les 8 25/11

11 December 2021 14:12

COMPLEXITEIT

Best & worst case

- vaste implementatie algoritme
- vaste invoergrootte n

Verschil:

Best case = invoer met minimaal tijdsgebruik, min aantal operaties

Worst case = invoer met maximaal tijdsgebruik, max aantal operaties

Average case = invoer zonder verdere beperkingen en veronderstellingen

≠ gemiddelde van worst en best case!

Voorbeelden:

Insertion sort: best case = gesorteerde lijst; worst case = omgekeerd gesorteerde lijst

Invoergrootte n

Bij getal als invoer

= x (waarde van x)

= 1+⌊log2 x⌋ (aantal cijfers/bits in x)

Bij collectie (set, list, etc.) als invoer

= len(xs) (aantal elementen)

Meten en plotten

--> assert, input, verwerking etc. w vanzelf meegeteld, we zoeken echter puur de 'tijd' van het algoritme

Kostenmodel

Opstellen van tijdsverloop algoritme adhv basisoperaties {op} met bepaalde uitvoeringstijd {t_op}

Vereenvoudiging & gevolg:

Elke basisoperatie zelfde cste tijd & herschaling naar uitvoeringstijd = 1

--> tellen van basisoperaties ipv tijdseenheden

Basisoperaties:

- Bewerkingen op getallen (indien niet te groot)
  + - * / // %
- Vergelijken van getallen
  == < > <= >=
- Lezen en schrijven van een variabele
  x x=…
- Lezen en schrijven van een element in een lijst (geheugenlocatie)
  x[i] x[i]=…

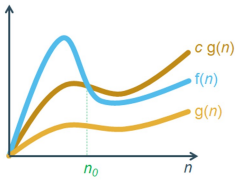
Grote O

= gedrag functie voor grote n (geeft geen exact aantal uitvoeringen!)

Definitie: $f(n) = O(g(n)) \Leftrightarrow \exists\, c, n0: \forall\, n \geq n0: f(n) \leq c\ g(n)$

Intuïtief: eens n groot genoeg is, dan stijgt f(n) minder snel dan g(n)

Praktisch: $f(n) \approx c * g(n)$ voor grote n



Voorbeelden:

3n**2 + 2n	3n**2 > 2n → 2n is verwaarloosbaar en macht groeit sneller dan veelvoud → 3n**2 + 2n = O(n**2)
3 log_3(n) + 7	3log_3(n) + 7 → grondtal (hier 3) maakt niet in grote O → O(log_3(n))
3**n + 2**n + n log(n)	3**n + 2**n + nlog(n) → Log groeit trager dan macht → O(3**n)

Vaak voorkomen de grenzen

- O(1) constant
  T(n) ≅ c . 1
  T(k . n) ≅ c . 1 ≅ T(n)
- O(log n) logaritmisch
  T(n) ≅ c . log n
  T(n**k) ≅ c . log n**k = k . c . log n ≅ k . T(n)
- O(n) lineair
  T(n) ≅ c . n
  T(k . n) ≅ c . k . n ≅ k . T(n)
- O(n log n) linearitmisch
  T(n) ≅ c . n . log n
  T(nk) ≅ c . nk . log nk = c . n . nk-1 . k . log n ≅ nk-1 . k . T(n)
- O(n2) kwadratisch
  T(n) ≅ c . n**2
  T(k . n) ≅ c . (k . n)**2 ≅ k**2 . T(n)
- O(n3) kubisch
  T(n) ≅ c . N**3
  T(k . n) ≅ c . (k . n)**3 ≅ k**3 . T(n)
- O(2n) exponentieel
  T(n) ≅ c . 2n
  T(n+k) ≅ c . 2n+k = c . 2n . 2k ≅ 2k . T(n)
- O(n!) factorieel
  T(n) ≅ c . n!
  T(n+k) ≅ c . (n+k)! = c . n! . (n+k)!/n! ≅ (n+k)!/n! . T(n)

Conclusie: exp & fac onbruikbaar --> te grote uitkomsten (zelfs voor kleine n)

Basisoperaties en grote O

- Basisoperaties: O(1)
- Operatie op lijst (legte = n):
  lege lijst maken: O(1)
  len(lst): O(1)
  lst[i], lst[i]=… : O(1)
  lst.append(x): O(1)
  lst.insert(i, x):
    best O(1) – i is einde lijst (=append)
    worst O(n) – i is begin lijst
  lst.count(x): O(n)
  x in lst:
    best O(1) – x op eerste plaats
    worst O(n) – x niet in lst
  range(n): O(1)
  list(range(n)): O(n)
  slice lst[i:i+k]: O(k)
- Operatie op dict/set (#elementen = n):
  lege set/dictionary maken: O(1)
  len: O(1)
  x in set/dict: O(1)
  dict[key], dict[key]=add, get, remove: O(1)

Rekenregels grote-O

f = f(n)
g = g(n)

1. O(f) + O(g) = max(O(f), O(g))
   Sequentie van instructies

2. c × O(f) = O(f)
   Herhaling, aantal keer (c) is onafhankelijk van n
   Tijdscomplexiteit onafh hh met c

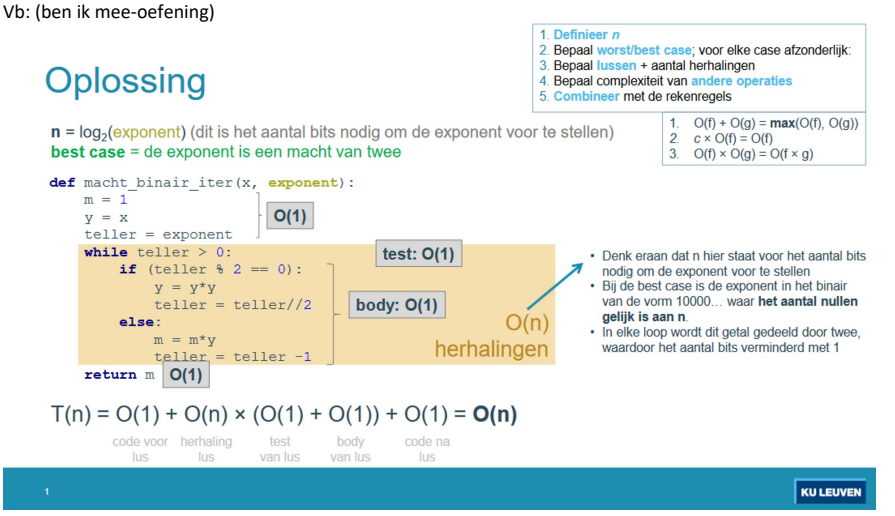
3. O(f) × O(g) = O(f × g)
   Herhaling, aantal keer is afhankelijk van n
   Tijdscomplexiteit afh hh met andere functie

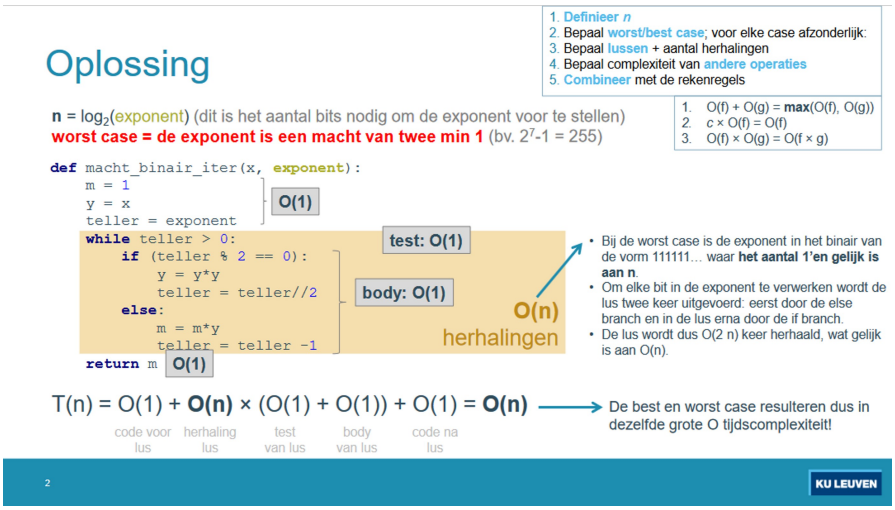
Tijdscomplexiteit iteratieve functies

--> redeneren adhv rekenregels:

1. **Definieer n** in functie van invoer
2. Bepaal **worst/best case**;
   voor elke case afzonderlijk volgende 3 stappen:
3. Bepaal **lussen** waarvan aantal iteraties afhangt
   van n (+ het aantal herhalingen in functie van n)
4. Bepaal complexiteit van **andere operat**ies (inclusief luslichamen)
5. **Combineer** met de rekenregels

Vb: (ben ik mee-oefening)





Tijdscomplexiteit recursieve functies

1: opstellen recursievergelijking

1. **Definieer n** in functie van invoer
2. Complexiteit **basisgeval**
3. Complexiteit **recursieve oproeping**
    - **Hoeveel** recursieve oproepen?
    - Hoe **verandert n**?
4. Complexiteit **alles behalve recursieve oproep**
5. **Samenvoegen** stap 3 & 4

2: geziene vergelijking?

- JA: gebruik gekende oplossing
- NEE: substitutie
    1. Doe 1 substitutie
    2. Herhaal stap 1 tot stap 3
    3. Veralgemeen met parameter k --> patroon
    4. Bepaal min waarde van k, leidt tot basisgeval
    5. k uit stap 4 invullen in stap 3: grote O bepalen

Geziene recursievgl

- $T(n) = T(n\text{-}1) + c$ — vb. Factorial, diameter bekers
    - $T(n) = O(n)$
- $T(n) = T(n/2) + c$ — vb. Binary search
    - $T(n) = O(\log_2 n)$
- $T(n) = 2\,T(n\text{-}1) + c$ — vb. Hanoi
    - $T(n) = O(2^n)$
- $T(n) = 2\,T(n/2) + c\,n$ — vb. Quicksort, Mergesort
    - $T(n) = O(n \log_2 n)$

Controle: wolframalpha

Les 9 02/12

11 December 2021 14:12

BACKTRACKING

Principe

genereert partiële oplossingen,
onderzoekt of die kunnen leiden tot volwaardige oplossingen,
en *komt op haar stappen terug* indien een partiële oplossing ongeschikt blijkt

--> specifiek problemen met randvoorwaarden

Varianten in backtrackingproblemen

1. Bestaat er een oplossing?
2. Als er een oplossing bestaat, geef de oplossing.
3. Geef alle oplossingen.
4. Geef de oplossing die aan voorwaarde voldoet…
   Bvb. kortste, langste, efficiëntste, …

Template backtracking algoritme

2 delen:

1. Examine
   Onderzoeken partiële oplossing:
   - Definitieve oplossing?
   - Geldige oplossing? (maw moeten we het pad verlaten)
   - Deel van volledige oplossing? Extend!
2. Extend
   Uitbreiden partiële oplossing in de hoop volwaardige opl te vinden

Pseudocode als template (variant 3):

Pseudocode van een backtracking-algoritme

```
def solve(partial_solution):
    exam = examine(partial_solution)
    if exam == ACCEPT:
        print(partial_solution)
    elif exam != ABANDON:
        for p in extend(partial_solution):
            solve(p)
```