

PROGRAMMEREN 2

TOPIC 7: RECURSIE

OVERZICHT

- Recursie in het algemeen
- Recursie in informatica
- Opdrachten, deel 1
- Functies met meerdere recursieve oproepen
- Voor- en naddeel
- Haskell
- Opdrachten, deel 2

INLEIDING

WAT IS RECURSIE?

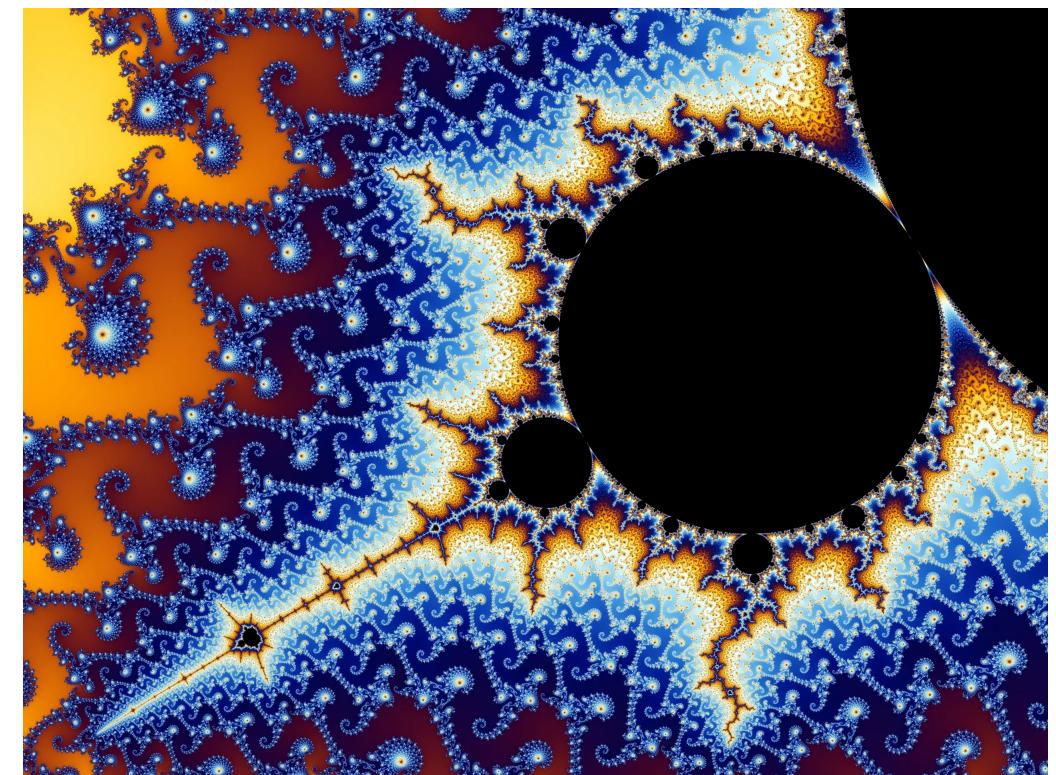
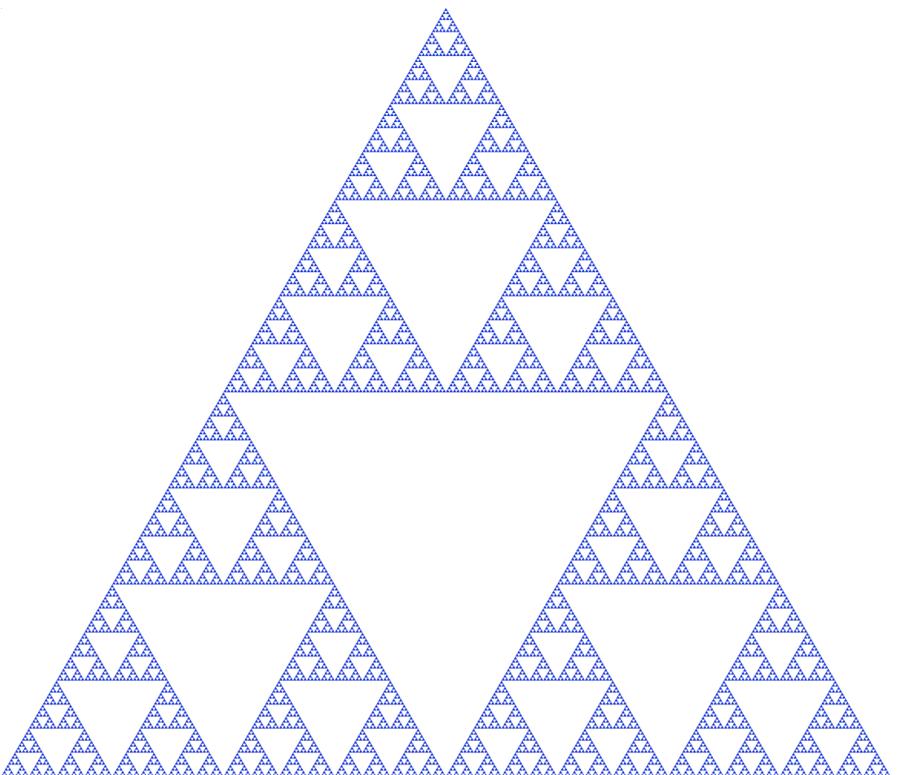
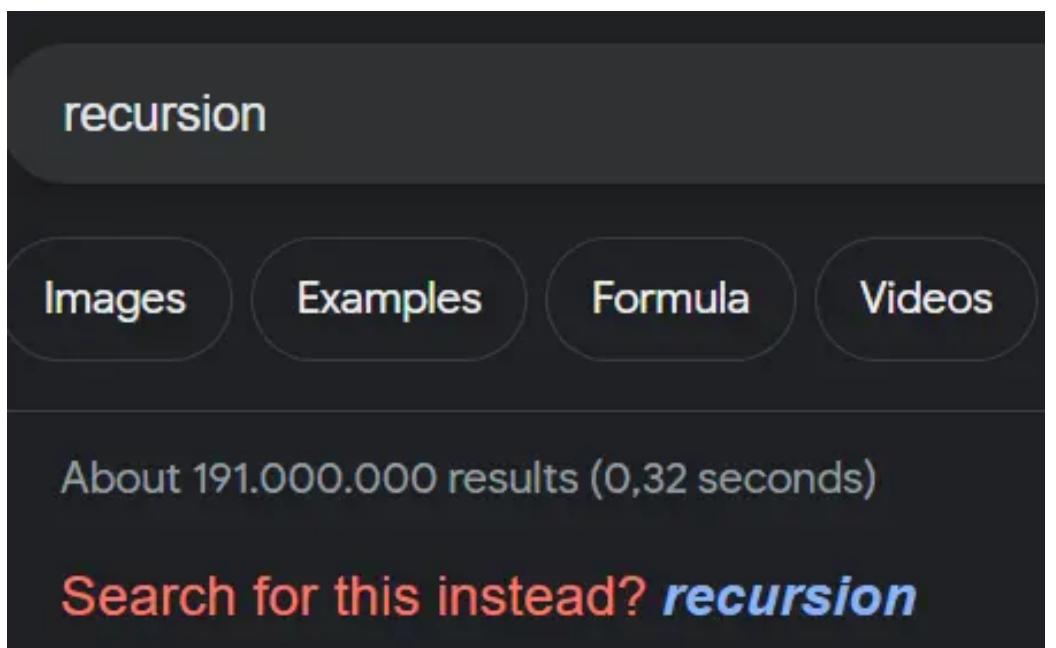
"In order to understand recursion, one must first understand recursion."

— Wikipedia

Men spreekt over recursie als iets **kopieën van zichzelf** bevat.

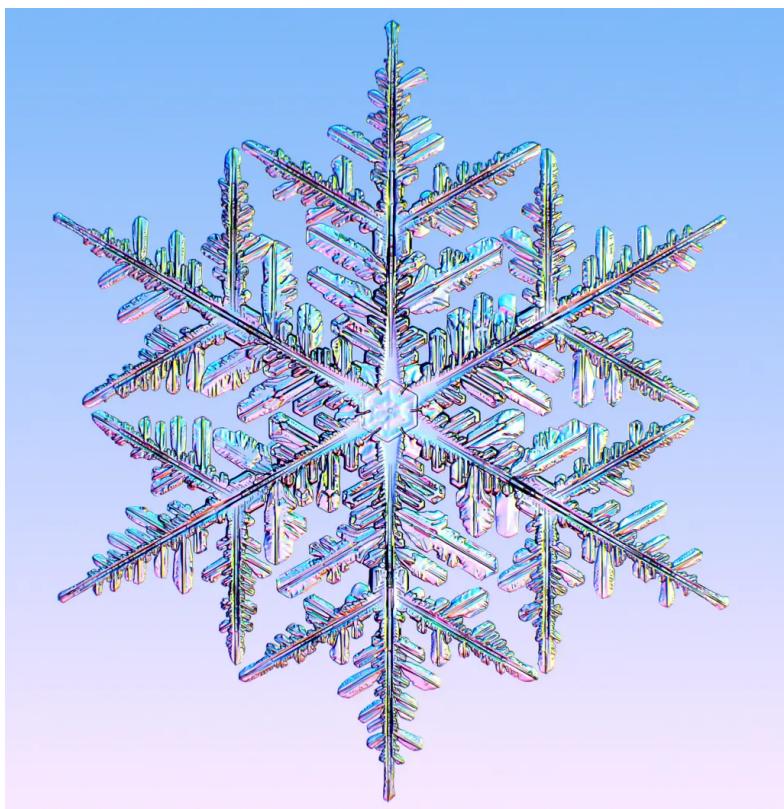
VOORBEELDEN

- Oneindige spiegel
- Recursie op Google
- Driehoek van Sierpinski
- Fractalen

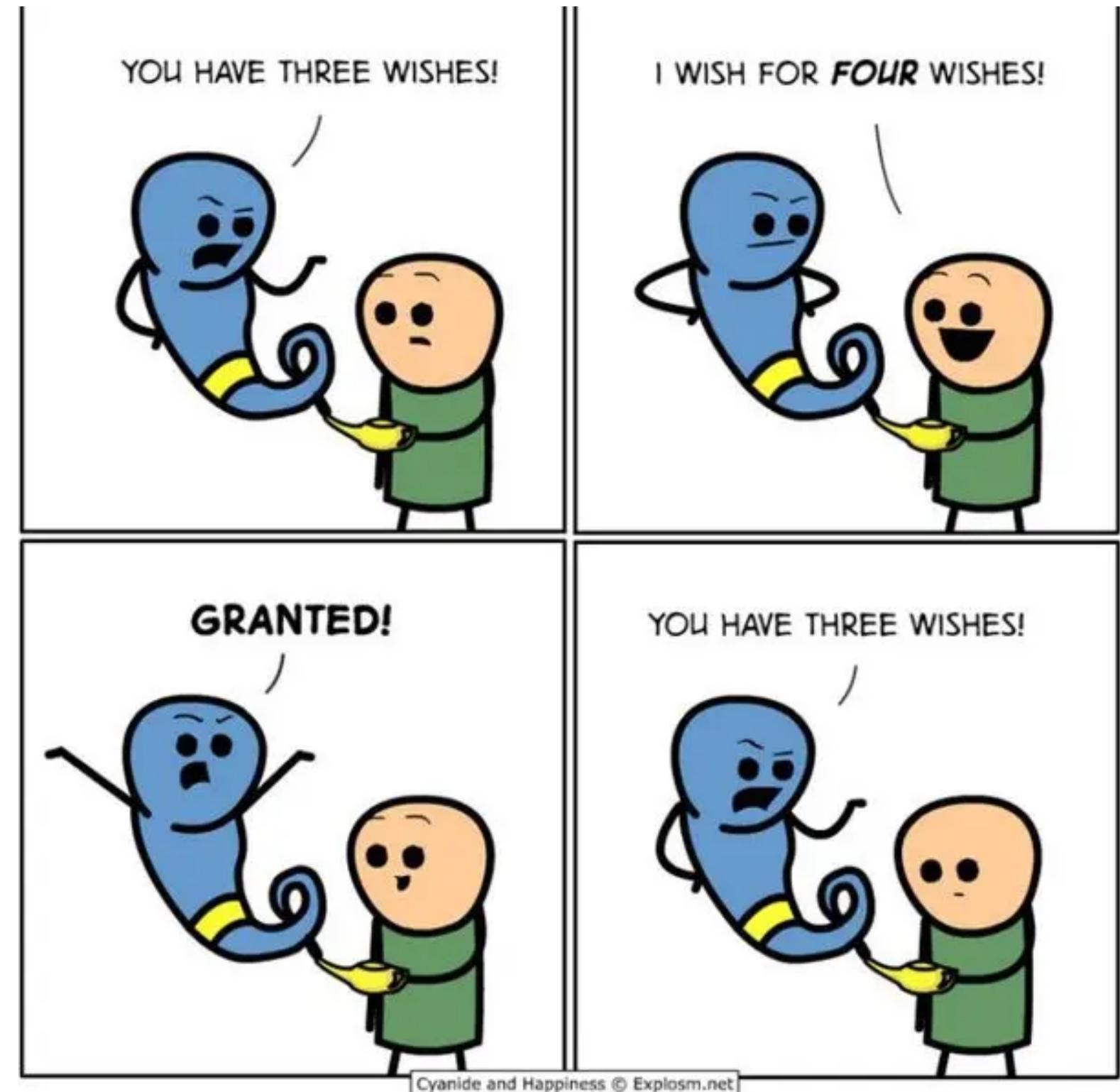
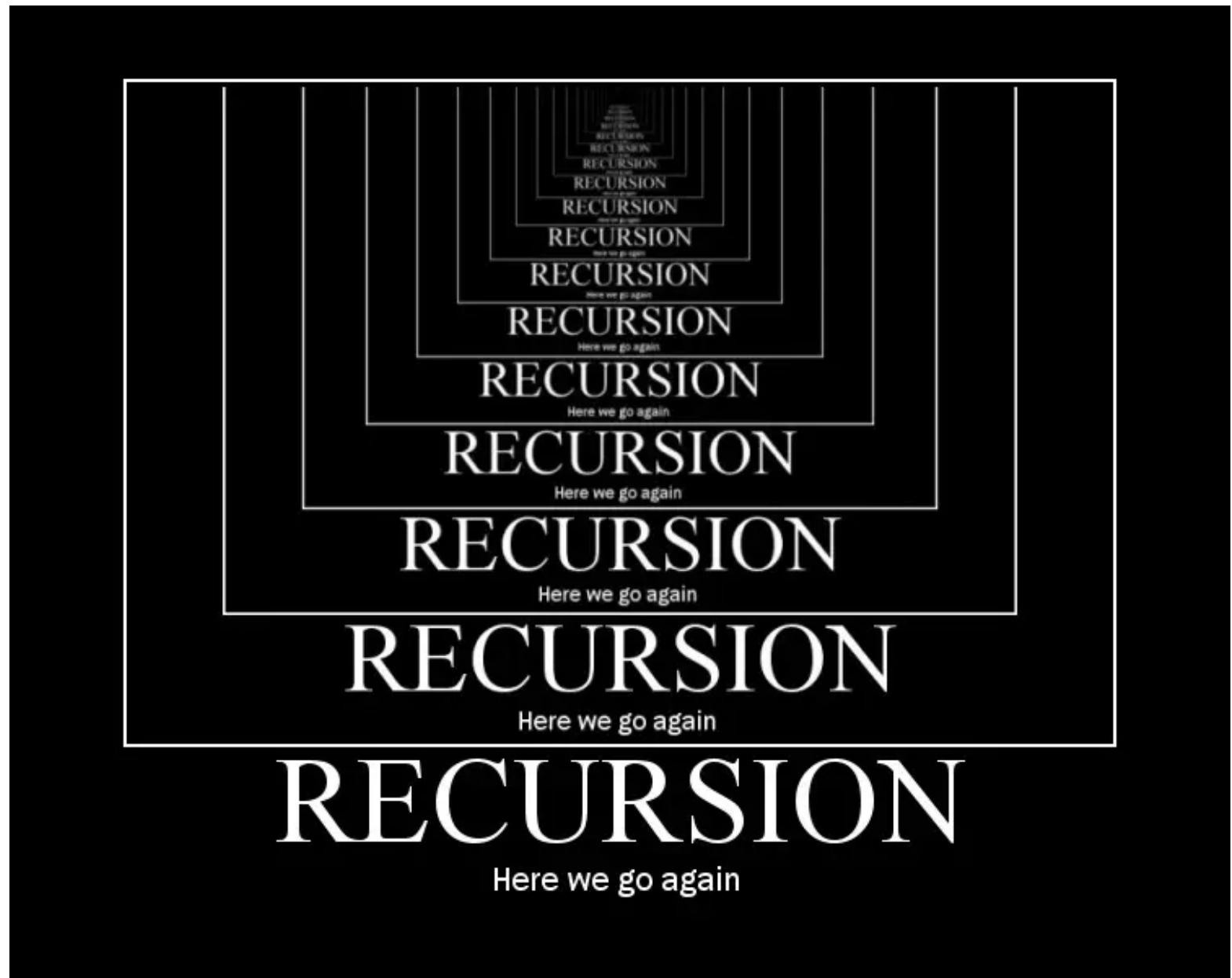


NOG MEER VOORBEELDEN

- Treed vaak op in de natuur als uitkomst van biologische processen
- Matroesjka



NOG MEER VOORBEELDEN



RECURSIE IN INFORMATICA

RECURSIE IN INFORMATICA

Recurсie is een programmeertechniek waarbij een functie zichzelf direct of indirect oproept. Men noemt dit een *recursieve* functie.

Er is veel overlap tussen recursie en herhalingen. Veel programma's met herhalingen kan je herschrijven met recursie. Recursie heeft bepaalde voor- en nadelen ten opzichte van 'gewone' herhalingen. Deze komen aan bod verder in de les.

Python

```
def recursieve_functie():
    recursieve_functie()

def rec_a():
    rec_b()

def rec_b():
    rec_a()
```

VOORBEELD: FACULTEIT

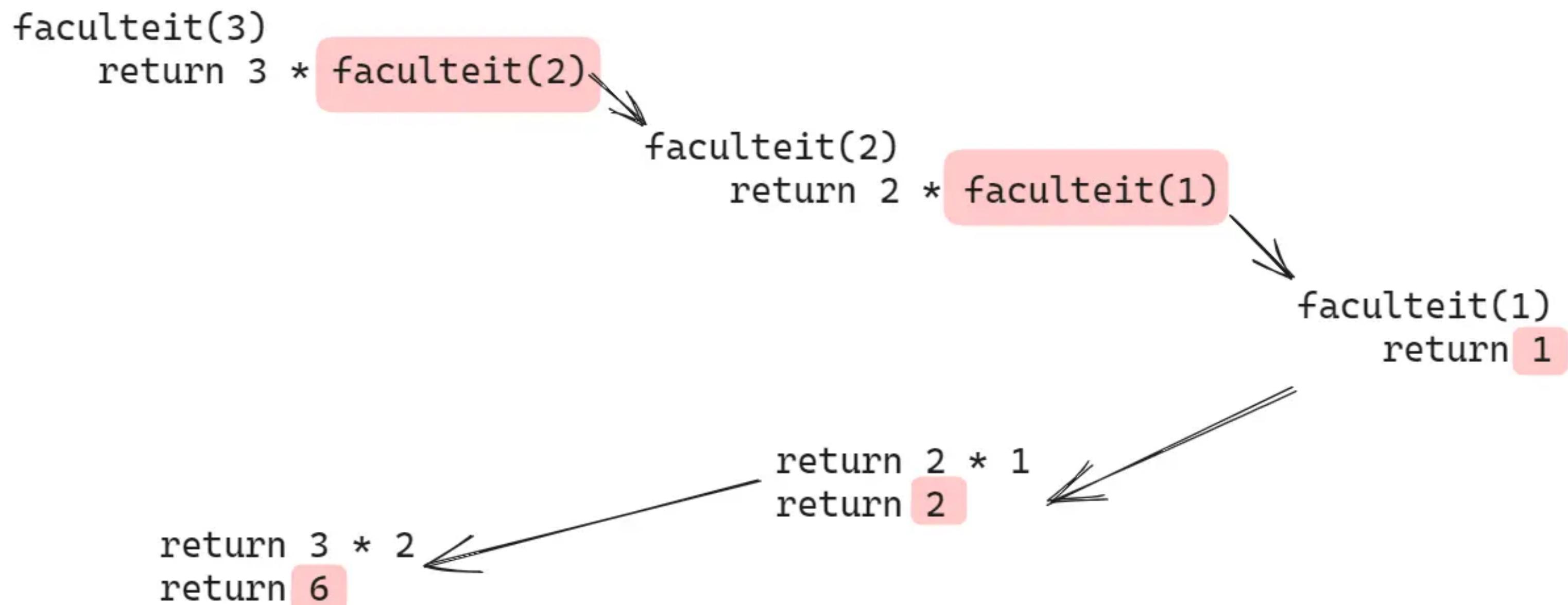
Python

```
def faculteit(x):
    """
    Implementeert de wiskundige bewerking ! op een recursieve manier.
    Bijvoorbeeld: 3! = 3 * 2 * 1
    """
    if x <= 1:
        return 1
    return x * faculteit(x-1) # de functie roept zichzelf op
```

```
# Ter vergelijking: de iteratieve versie
def faculteit_iter(x):
    """
    Implementeert de wiskundige bewerking ! op een iteratieve manier.
    """
    result = 1
    for i in range(1, x + 1):
        result = result * i
    return result
```

WAT GEBEURT ER TIJDENS DE UITVOERING?

Stel dat we `faculteit(3)` proberen uit te voeren. Wat doet Python achter de schermen?



WAT GEBEURT ER TIJDENS DE UITVOERING? (2)

Telkens wanneer we een functie oproepen, zal Python eerst **de huidige toestand (variabelen, parameters, huidige plaats in de code) opslaan** in het geheugen van de computer. Er wordt als het ware een foto genomen van de huidige scène. Men noemt dit stuk geheugen de **call stack**.

Waarom wordt dit gedaan? Waarom is die informatie belangrijk?

WAT GEBEURT ER TIJDENS DE UITVOERING? (2)

Telkens wanneer we een functie oproepen, zal Python eerst **de huidige toestand (variabelen, parameters, huidige plaats in de code) opslaan** in het geheugen van de computer. Er wordt als het ware een foto genomen van de huidige scène. Men noemt dit stuk geheugen de **call stack**.

Waarom wordt dit gedaan? Waarom is die informatie belangrijk?

Omdat daarmee de computer de toestand kan herstellen als de functie voltooid is. Op basis van de foto wordt de vorige scène weer opgebouwd.

WAT GEBEURT ER TIJDENS DE UITVOERING? (2)

Telkens wanneer we een functie oproepen, zal Python eerst **de huidige toestand (variabelen, parameters, huidige plaats in de code) opslaan** in het geheugen van de computer. Er wordt als het ware een foto genomen van de huidige scène. Men noemt dit stuk geheugen de **call stack**.

Waarom wordt dit gedaan? Waarom is die informatie belangrijk?

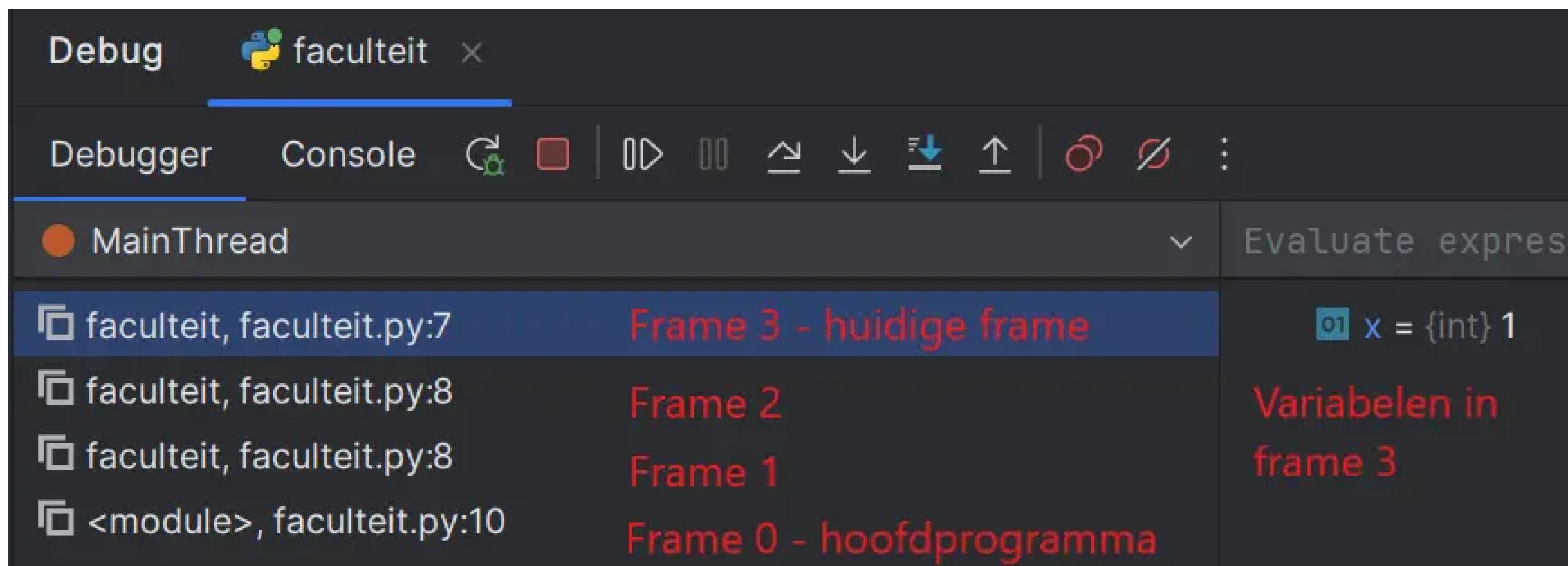
Omdat daarmee de computer de toestand kan herstellen als de functie voltooid is. Op basis van de foto wordt de vorige scène weer opgebouwd.

Dit geldt voor **alle** functies (recursief en niet-recursief). Het geldt in veel - maar niet alle - programmeertalen.

CALL STACK INSPECTEREN

Met een debugger (zoals die van Pycharm) kan je de call stack bekijken. Elke functieoproep genereert een **frame** (foto) die op de call stack wordt geplaatst.

1. Zet een breakpoint (rode bol) naast de lijnen code waar je wilt dat het programma pauzeert
2. Voer het programma uit in debug mode (keverknop)
3. Wanneer het programma pauzeert op een punt, kan je de huidige frame en alle 'bovenliggende' frames bekijken



OPDRACHTEN, DEEL 1

HOE SCHRIJF JE EEN CORRECT RECURSIEF ALGORITME?

Een recursief algoritme heeft altijd minstens volgende kenmerken:

- Een of meerdere **basisgevallen** (*base cases*): dit zijn (vaak triviale) stopcondities. In `faculteit(x)` is $0! = 1$ het basisgeval
- Een of meerdere **recursieve** gevallen: hierin wordt de functie opnieuw opgeroepen met aangepaste parameters

Voor de rest kan je in een recursieve functie alles doen wat ook in gewone functies kan. Zelfs lussen zijn toegelaten - al zal dat niet vaak nodig zijn in de praktijk.

OPDRACHT: FIBONACCI

Doel: Implementeer `fibonacci(x)`, een functie die het `x`e cijfer van de rij van Fibonacci teruggeeft.

1. Geen idee meer wat de rij van Fibonacci is? Zoek zelf online naar een kort antwoord
2. Schrijf eerst 4 testen voor de functie. Gebruik `assert` hiervoor
3. Implementeer de functie. Gebruik geen lussen, enkel recursie
4. Controleer of de functie voldoet aan je testen
5. Gebruik debug mode om te kijken hoe de call stack evolueert naarmate je functie meer wordt opgeroepen. Tip: zet alle testen uit (maak er comments van), behalve één naar keuze.

Extra: implementeer `fibonacci_iter(x)`. Deze functie geeft hetzelfde terug als `fibonacci(x)`, maar werkt iteratief. Hergebruik je testen om te controleren of de implementatie correct is.

OPDRACHT: LENGTE VAN EEN LIJST BEPALEN

Doel: Implementeer `my_len(l)`, een functie die de lengte van de lijst `l` teruggeeft. Ze doet dus hetzelfde als de ingebouwde Python-functie `len()`.

1. Voor deze opdracht kan list slicing van pas komen. Zoek op wat list slicing in Python inhoudt. Zie bv. [python-slice-usage](#)
2. Schrijf eerst 3 testen voor de functie, inclusief 1 voor de base case. Gebruik `assert` hiervoor
3. Implementeer de functie. Gebruik geen lussen, enkel recursie. Het is verboden om ingebouwde functie zoals `len()` en `count()` te gebruiken
4. Controleer of de functie voldoet aan je testen

Extra: implementeer een iteratieve versie van de functie.

GEVORDERDE RECURSIE

FUNCTIES MET MEER DAN ÉÉN RECURSIEVE OPROEP NA ELKAAR

RECURSIEF SORTEREN

Een recursief algoritme om data te sorteren:

1. Splits de verzameling in twee delen, **klein** en **groot**, waarbij je ervoor zorgt dat alle elementen in **klein** kleiner zijn dan alle elementen in **groot**
2. Herhaal (1) op **klein**
3. Herhaal (1) op **groot**
4. Zet **groot** achter **klein**

Een meer uitgebreide versie van dit algoritme staat gekend als **Quicksort**.

Dit algoritme bevat twee recursieve oproepen na elkaar. Dit is al veel moeilijker om te vertalen naar een iteratieve versie (een versie met **for** of **while**).

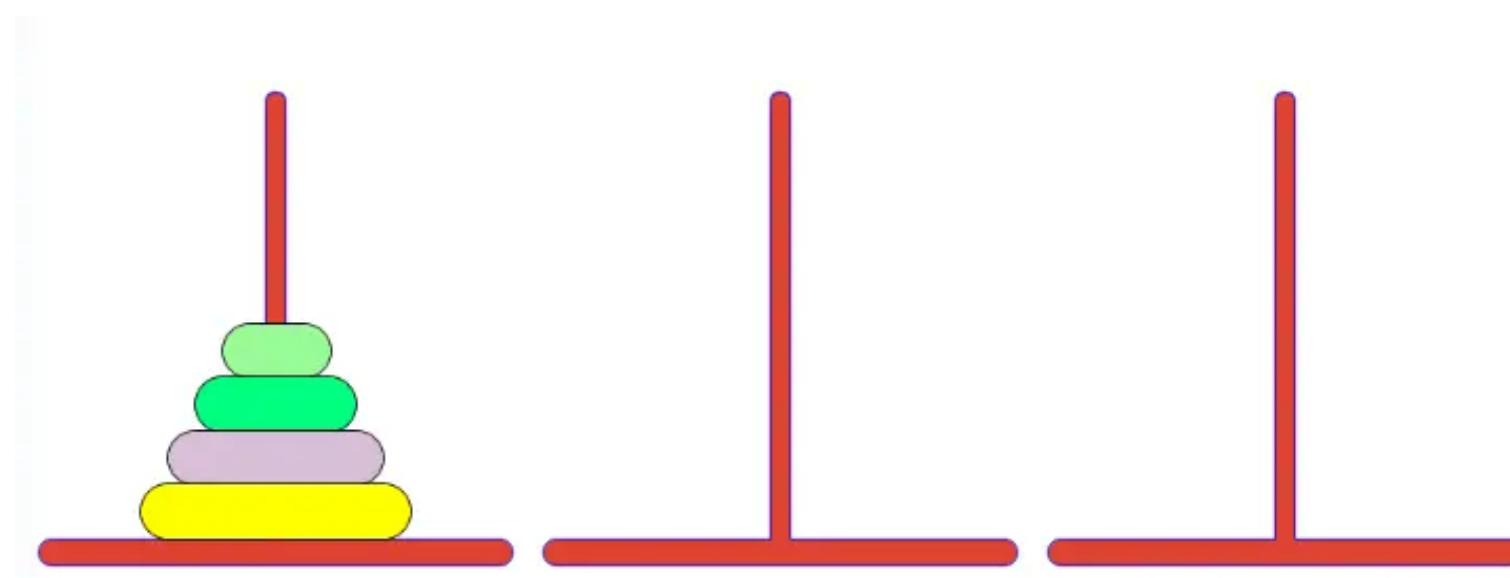
TORENS VAN HANOI

Deze oude puzzel werkt als volgt:

- Er zijn 3 pinnen: bron, hulp, doel
- Er zijn N schijven (minstens 3), gestapeld volgens grootte
- Doel: verplaats de schijven 1 voor 1 van bron naar doel zodat ze ook op doel gestapeld zijn volgens grootte
- Het is niet toegelaten om een grotere schijf op een kleinere te leggen

Kan jij de puzzel oplossen in het minimaal aantal beurten? Welk recursief patroon keert steeds terug?

Online simulatie



TORENS VAN HANOI: ALGORITME IN MENSENTAAL

Om **N** schijven van **bron** naar **doel** te verplaatsen:

1. Verplaats **N-1** schijven van **bron** naar **hulp**
2. Verplaats schijf **N** van **bron** naar **doel**
3. Verplaats **N-1** schijven van **hulp** naar **doel**

Probeer dit algoritme in Python te programmeren:

1. Maak een functie **hanoi(N, bron, doel, hulp)**
2. Bepaal de base case
3. Bepaal de recursieve cases
4. Print telkens de actie die moet gebeuren. Bijvoorbeeld:
 1. Verplaats schijf 1 van BRON naar HULP
 2. Verplaats schijf 2 van BRON naar DOEL
 3. ...

TORENS VAN HANOI: OPLOSSING

Merk op dat er twee recursieve oproepen zijn, met verschillende parameters.
Tussendoor gebeurt er nog iets anders (een `print()`).

TORENS VAN HANOI: OPLOSSING

Python

```
def hanoi(n, bron, doel, hulp):
    if n == 1:
        print("Verplaats schijf 1 van {} naar {}".format(bron, doel))
    else:
        hanoi(n - 1, bron, hulp, doel)
        print("Verplaats schijf {} van {} naar {}".format(n, bron, doel))
        hanoi(n - 1, hulp, doel, bron)
```

Merk op dat er twee recursieve oproepen zijn, met verschillende parameters.
Tussendoor gebeurt er nog iets anders (een `print()`).

VOOR- EN NADELEN VAN RECURSIE

RISICO'S EN BEDENKINGEN

Op basis van voorgaande voorbeelden en opdrachten, zou je denken dat recursie heel vaak gebruikt wordt. In de praktijk is dit echter niet het geval. Waarom? Wat zijn de voor- en nadelen tegenover herhalingen?

Om recursie dieper te begrijpen, is het noodzakelijk om meer aandacht te besteden aan het geheugen van computers.

ELEGANTIE EN LEESBAARHEID

Recursieve code is vaak 'eleganter' en eenvoudiger te lezen. Ze is ook eenvoudiger te programmeren, indien je de oplossing in mensentaal of een formele taal recursief kan beschrijven (bv. definitie van faculteit in de lessen wiskunde).

Recursieve code schrijven vraagt wat oefening, zeker als je gewend bent om iteratieve code te schrijven (lussen).

GEHEUGENGEBRUIK

faculteit(n) neemt op een bepaald punt tijdens uitvoering het volgende geheugen in:

- **n** frames
- elk frame heeft 1 variabele **x**
- elk frame bevat ook 'onzichtbare' contextdata (framenummer, parent frame, ...)

Ter vergelijking: **faculteit_iter(n)** vereist 1 frame in totaal met daarin 2 variabelen. Dit is onafhankelijk van de grootte van **n**.

→ **Gigantisch verschil!**

Wat als we 1 000, 10 000, ... frames nodig hebben?

ONEINDIGE LUSSEN?

Als je de base case in een recursieve functie verwijdert - of vergeet te programmeren - krijg je vanzelf een oneindige lus. Of toch niet?

```
RecursionError: maximum recursion depth exceeded in comparison
```

Python

Python - en ook andere talen - proberen hun geheugengebruik te begrenzen. Recursie leidt makkelijk tot een hoog geheugengebruik, door het aantal frames dat een recursieve functie kan produceren.

ONEINDIGE LUSSEN?

Als je de base case in een recursieve functie verwijdert - of vergeet te programmeren - krijg je vanzelf een oneindige lus. Of toch niet?

Python

```
RecursionError: maximum recursion depth exceeded in comparison
```

Python - en ook andere talen - proberen hun geheugengebruik te begrenzen. Recursie leidt makkelijk tot een hoog geheugengebruik, door het aantal frames dat een recursieve functie kan produceren.



VERDEELDE MENINGEN

Recursie leidt soms tot 'heftige' discussies. Voorstanders verkiezen de elegante en intuïtieve code, tegenstanders hameren op het zware geheugengebruik.



Kies steeds **bewust** voor recursie. Probeer op voorhand in te schatten of het geheugenverbruik een probleem kan vormen later.



In de meeste opdrachten op school gaan leerlingen niet snel tegen de grenzen van recursie aanbotsen. Het intuïtieve aspect van recursie kan bovendien een voordeel zijn voor beginnende programmeurs.

DE GOUDEN MIDDENWEG?

HASKELL

CONTEXT

Het voornaamste argument tegen recursie is het geheugenverbruik. Nochtans kan je recursieve functies slimmer laten werken. Er zijn allerlei eenvoudige optimalisatietechnieken mogelijk.

De meeste talen 'herkennen' recursieve functies echter niet als speciale functies. Ze behandelen ze zoals andere functies. Optimalisaties zijn daarom niet mogelijk, extreem geheugenverbruik lijkt haast onvermijdelijk.

Kan het anders?

HASKELL

Een **functionele** programmeertaal met **statische** types.



Enkele opvallende kenmerken:

- De taal is *puur*: een functie doet **altijd** hetzelfde bij dezelfde invoer
- Variabelen zijn nooit aanpasbaar. Ze zijn **immutable**, zonder uitzondering
- Types zijn statisch, alle datasoorten zijn op voorhand gekend
- Functies zijn zelf types, net zoals strings, numbers, booleans

HERHALING IN HASKELL

Aangezien variabelen niet kunnen veranderen, ondersteunt Haskell geen **for** of **while**-lus. Waarom?

Toch ondersteunt Haskell herhalingen. Hoe?

HERHALING IN HASKELL

Aangezien variabelen niet kunnen veranderen, ondersteunt Haskell geen **for** of **while**-lus. Waarom?

Beide lussen hebben een variabele nodig die verandert (een booleaanse waarde of een teller-variabele).

Toch ondersteunt Haskell herhalingen. Hoe?

HERHALING IN HASKELL

Aangezien variabelen niet kunnen veranderen, ondersteunt Haskell geen **for** of **while**-lus. Waarom?

Beide lussen hebben een variabele nodig die verandert (een booleaanse waarde of een teller-variabele).

Toch ondersteunt Haskell herhalingen. Hoe?

Via recursie.

HASKELL: VOORBEELD VAN HERHALING

faculteit(x) in Haskell:

```
faculteit :: Integer -> Integer
faculteit 0 = 1
faculteit n = n * faculteit (n - 1)
```

Haskell

1. Definieer de functie **faculteit**. Ze accepteer een **Integer** en geeft een **Integer** terug
2. Schrijf de base case: $0! = 1$
3. Schrijf de recursieve case: $n! = n * (n - 1)!$

Haskell probeert alle varianten van een functie uit van boven naar onder, en kiest steeds de eerste variant die matcht. Bij elke recursieve call herstart hij bovenaan de functiedefinitie.

HASKELL: VOORDELEN

Haskell dwingt af dat programmeurs recursie gebruiken in plaats van lussen. Daardoor kan de taal allerlei trucs en optimalisaties toepassen achter de schermen.

In het algemeen transformeert Haskell elke recursieve call naar een iteratie. Dit gebeurt volledig automatisch tijdens de compilatiestap, net voor je een programma uitvoert. Het is dus zeer geheugenefficiënt, en toch behoudt het de voordelen van recursie.

Haskell is helaas niet zo wijdverspreid. Het is vooral populair onder wiskundigen die willen programmeren, omdat het veel meer wiskundige regels benadert dan andere programmeertalen.

MEER WETEN?

- Online playground (geen installatie nodig)
- Wikibooks
- Learn You a Haskell

OPDRACHTEN, DEEL 2

OPDRACHTEN

GRENZEN VAN RECURSIE

Bepaal vanaf welke invoerwaarde jouw implementatie van `faculteit(x)` de recursiegrens overschrijdt op je computer. Doe hetzelfde voor `fibonacci(x)`.

TOREN VAN HANOI: UITBREIDING

Pas `hanoi(x)` aan zodat het bijhoudt hoeveel stappen er gezet zijn. Laat de functie het aantal stappen teruggeven als een nummer. Je mag hiervoor de functie uitbreiden met variabelen, parameters, ...

VEELVOUDEN VAN 9

Een getal is een veelvoud van 9 als de som van de cijfers een veelvoud van 9 is.
Schrijf recursieve functies om te testen of een gegeven positief geheel getal een
veelvoud van 9 is.

Tip: schrijf een aparte functie om de som van de cijfers in een getal te berekenen.
(Hoe pak je één voor één de cijfers uit een getal?)

Extra: implementeer dit opnieuw, maar nu iteratief (met lussen)

ZOEKEN IN EEN GESORTEERDE LIJST

Implementeer de basisversie van Quicksort uit de slides in Python en test het uit.

Denk na over hoe je het referentie-element telkens selecteert.

Let op: een rij met een even aantal elementen heeft geen “middelste” element.

HASKELL (MOEILIJKER)

Verdiep je in Haskell en hermaak een aantal oefeningen. Suggesties:

- Fibonacci
- Lengte van een lijst
- Toren van Hanoi
- ...