Algoritmen en programmeren

Nota's voor de leerkrachten

Kris Coolsaet

2024

1 Algoritmen en programmeren

Dit document bevat de begeleidende tekst bij enkele nascholingen van Universiteit Gent rond de specifieke minimumdoelen *Algoritmen en programmeren*. Die nascholingen werden een eerste keer gegeven in 2023-2024.

1.1 De minimumdoelen

De specifieke minimumdoelen '*07.01 Algoritmen en programmeren*' in het vakgebied Informaticawetenschappen zijn in hun officiële vorm heel summier geformuleerd (zie inzet) en daardoor sterk voor interpretatie vatbaar. Aangezien wij (de auteur en zijn medewerkers) hebben meegeholpen aan het vastleggen van die minimumdoelen, aan de eerder verworpen, maar meer gedetailleerde specifieke eindtermen, en aan de eruit voortvloeiende leerplannen, zijn we goed geplaatst om wat meer uitleg te geven bij de achtergrond en de bedoeling van deze minimumdoelen. Tegelijk kaderen we ze hier in de context van de nascholingen.

Minimumdoelen

07.01.01 De leerlingen programmeren zelf ontworpen oplossingen voor concrete problemen.
Onderliggende (kennis)elementen

- Algoritmen en datastructuren
- Algoritmische technieken
- Numerieke methodes
- Gebruik van softwarebibliotheken
- Gestructureerde programmeertaal
- Invoer van en uitvoer naar externe gegevensbronnen

1.2 Gestructureerde programmeertaal

Elke leerling wordt geacht de beginselen van een tekstuele programmeertaal onder de knie te hebben. De toevoeging *gestructureerd* aan de beschrijving duidt aan dat er geen *objectgerichte* programmeertaal hoeft gebruikt te worden, en meer specifiek, dat de begrippen zoals overerving, polymorfisme en interfaces niet moeten behandeld worden. Dit betekent echter niet dat we woorden zoals *object*, *klasse* en *methode* uit de weg moeten gaan. Deze begrippen zullen onvermijdelijk opduiken wanneer we gebruik maken van externe bibliotheken (zie §1.4) en ook de puntnotatie voor het oproepen van een methode van een object wordt vaak gebruikt. We noemen dit object*gebaseerd* programmeren.
Voor de nascholingen kozen we voor de programmeertaal Python, een programmeertaal die in het middelbaar onderwijs, maar ook in wetenschappelijke richtingen van het hoger onderwijs, zeer populair is. Als eerste kennismaking met programmeren, is dit zeker een geschikte taal.
Om het onderwerp algoritmen aan te leren is Python echter niet 100% geschikt, en we zullen ons daarom af en toe eens in didactische bochten moeten wringen om enkele tekortkomingen te compenseren[1]:

- Python is een geïnterpreteerde programmeertaal[2] en daardoor een stuk trager dan een gecompileerde taal zoals C. Op zich is dit niet zo erg – absolute snelheid is voor ons niet belangrijk en hoe dan ook zijn de huidige computers meer dan snel genoeg voor wat we er hier mee willen doen. Het probleem is echter dat de kern van Python en zijn belangrijkste bibliotheken wel in een gecompileerde taal zijn geschreven. Daardoor is een inefficiënt algoritme dat in een Python-bibliotheek is voorgeprogrammeerd vaak toch nog een stuk sneller dan een efficiënt algoritme dat je zelf hebt geschreven. Dit maakt het moeilijker om het belang van efficiëntie bij algoritmen te illustreren.

[1] En daardoor zijn de voorbeelden die we gebruiken helaas ook niet altijd direct te vertalen naar een andere programmeertaal zoals C# of Java.
[2] Er bestaan ook gecompileerde versies van Python, maar die worden in het onderwijs nog weinig gebruikt.

- Python kent enkel *lijsten* en geen *arrays*. De beperking dat arrays een vaste lengte hebben, biedt een natuurlijke motivatie voor een aantal technieken die arrays '*in place*' aanpassen, een motivatie die wegvalt wanneer je enkel met lijsten werkt. Daarbij komt nog dat elementen toevoegen en verwijderen aan een lijst, wat men in de algoritmische praktijk vaak vermijdt omdat het veel kost, in Python opnieuw relatief snel gaat, omdat die bewerkingen deel uitmaken van de kern van Python (zie hierboven).

Alternatieven zoals C, C# en Java hebben dan weer het nadeel dat ze een hogere instap vereisen. C# en Java zijn bovendien objectgerichte talen, en voldoende aandacht besteden aan de objectgerichte aspecten van de taal, vraagt extra tijd, tijd die je misschien liever besteedt aan andere elementen uit de minimumdoelen.

1.3 Invoer en uitvoer naar externe gegevensbronnen

De eerste programma's die je leert schrijven, zijn kort, vragen weinig invoer, geven weinig uitvoer en doen niets dat echt spectaculair kan genoemd worden. Naarmate de weken vorderen, en je bijvoorbeeld met lijsten begint te werken, verdwijnen de 'speelgoedvoorbeeldjes' en begin je door te hebben dat een goed geschreven programma wel eens zijn nut zou kunnen hebben. Helaas moet je telkens wanneer je het programma opstart alle gegevens die je nodig hebt opnieuw invoeren. En als het programma is afgelopen, ben je terug alles kwijt. Op de duur werkt dit demotiverend.

Daarom is het belangrijk heel snel te tonen hoe gemakkelijk het is om in een programma met *persistente* gegevens te werken – dat gegevens kunnen worden opgeslagen in een bestand om dan later opnieuw door hetzelfde (of door een ander) programma te worden verwerkt. Het breidt ook het toepassingsgebied van je programma's uit, in plaats van bijvoorbeeld 5 woordjes in te tikken waarmee je dan een bewerking doet, kan je nu een bestand met daarin alle Nederlandse woorden inlezen om er iets nuttigs mee te doen.

Je kan ook gebruik maken van bestanden die door andere software zijn geproduceerd (in andere vakken), zoals werkbladen met cijfergegevens, of gegevens die je hebt gedownload van het Internet. Omgekeerd kan jouw programma nu ook gegevens opslaan in een vorm die je dan met andere software verder verwerkt.

1.4 Gebruik van softwarebibliotheken

De tijd dat je een volledig programma helemaal op je eentje schreef, is lang voorbij. Software wordt in grote teams ontwikkeld waarbij elke programmeur slechts een klein deeltje van het geheel programmeert. *Programmeren doe je nooit alleen!* Eén aspect hiervan is het kunnen gebruiken van functies en procedures (in de praktijk meestal klassen en methodes) die door een andere programmeur geschreven zijn, zonder dat je de broncode ervan hebt en waarbij je dus enkel kunt voortgaan op de beschikbare documentatie.

Voor deze minimumdoelen wordt het samenwerkingsaspect bij het schrijven van software beperkt tot het gebruik van software*bibliotheken*.

In deze tekst maken we gebruik van enkele populaire externe bibliotheken waarover je uitgebreide documentatie en heel wat voorbeelden op het Internet terugvindt. Wij kozen daarvoor in de eerste plaats voor toepassingen die motiverend werken: een PIL-bibliotheek waarmee je eenvoudige afbeeldingen kan maken, en `matplotlib` voor het tekenen van grafieken. Daarnaast komen ook nog enkele standaardmodules uit Python aan bod (t.t.z., *interne* bibliotheken) die bij elke Python-installatie aanwezig zijn en bijna als onderdeel van de programmeertaal kunnen worden beschouwd: `math`, `random`, `time`, …

1.5 Algoritmen en datastructuren

Bij je eerste programma's kan je over het algemeen je ontwerp baseren op hoe je het gestelde probleem zelf, als mens, stap voor stap oplost. De moeilijkheid bestaat er dan voornamelijk in om deze stappen op een correcte manier naar een programma te vertalen.

Voor minder eenvoudige programma's die heel veel gegevens moeten verwerken, en in beperkte tijd, volstaat dit niet en wordt het nuttig de bijzondere technieken te kennen die hiervoor over de jaren heen ont-

wikkeld zijn. Dit zijn de *algoritmen* waarvan sprake in deze tekst.
We gebruiken hier dus een veel specifiekere invulling dan de meer algemene definitie die bijvoorbeeld in de (didactische) context van *computationeel denken* gehanteerd wordt: 'een algoritme is een stapsgewijze reeks instructies of regels die worden gevolgd om een specifieke taak of probleem op te lossen'. In die betekenis geldt elk programma of elke functie of procedure in een programma als algoritme, en dat is niet wat met dit minimumdoel wordt bedoeld.
Misschien dekt de term *standaard*algoritme beter de lading: een algoritme dat in de wetenschappelijke literatuur een zekere status heeft verworven, dat de tand des tijds heeft doorstaan en dat algemeen gekend is als de geschikte manier om een bepaald probleem op te lossen. Voorbeelden zijn: het algoritme van Euclides om de grootste gemene deler te vinden van twee getallen, het '*merge sort*'-algoritme om gegevens te sorteren (zie hoofdstuk ?), en de technieken die een routeplanner gebruikt om de snelste route te vinden naar jouw bestemming.
Een *datastructuur*, ook wel gegevensstructuur genoemd, zouden we wat simplistisch kunnen definiëren als een bepaalde manier om gegevens in een programma op te slaan – in Python is een *lijst* hiervan een typisch voorbeeld. In de praktijk is het echter ook heel belangrijk te omschrijven op welke manier je die gegevens verwacht te gebruiken of te verwerken. Een *geordende lijst* – een lijst met getallen waarbij we erop toezien dat die steeds van klein naar groot geordend blijven, omdat dit het zoeken veel efficiënter maakt (zie hoofdstuk 3), is een andere gegevensstructuur dan een *frequentietabel* (zie hoofdstuk 7) – een lijst die bijhoudt hoeveel keer bepaalde waarden in een reeks gegevens voorkomen, ook al worden die beiden in Python door een lijst van getallen voorgesteld.
Met andere woorden, een bepaalde datastructuur is steeds onlosmakelijk verbonden met de algoritmen die nodig zijn om die doeltreffend te gebruiken (wat is de beste manier om een element toe te voegen aan een geordende lijst zodat ze op volgorde blijft?). Vandaar dat informaticawetenschappers 'algoritmen en datastructuren' steeds als een ondeelbare term zien en er aan de universiteiten geen afzonderlijke vakken 'algoritmen' en 'datastructuren' bestaan.
In dit document benaderen we de datastructuren (en hun algoritmen) op twee verschillende manieren.

- Als *abstracte* datastructuren - 'zwarte dozen' die een handig hulpmiddel blijken in bepaalde situaties, maar die we kunnen gebruiken zonder dat we hoeven te weten hoe die er 'van binnen' uitzien. Zo is een lijst in Python een abstracte datastructuur. We weten bijvoorbeeld niet wat er achter de schermen gebeurt wanneer we een element in een lijst tussenvoegen[3].
- Als *concrete* datastructuren, waar we in detail ingaan op hoe die datastructuren worden geprogrammeerd. Door een dieper inzicht in de werking ervan kan je ook beter inschatten in welke context deze structuren efficiënt kunnen ingezet worden. Het helpt je ook bij het ontwerp van eigen algoritmen voor problemen die verwant zijn aan de klassieke vragen maar er lichtjes van afwijken.

1.6 Algoritmische technieken

De algoritmen en datastructuren uit de vorige paragraaf dienen om *standaard*problemen op te lossen (zoeken in een grote groep gegevens, sorteren ervan, enz.) In de praktijk is het echter vaak nodig om zelf een algoritme te ontwerpen in een specifieke context, zoals bijvoorbeeld bij het zoeken naar een (bijna) optimale oplossing van een probleem – een korte route, een efficiënte planning, een doeltreffende verdeling van middelen, enz.
Er bestaan een aantal algemene strategieën, of *algoritmische technieken*, die je hierbij helpen. Zo kan je bij een optimalisatieopdracht alle mogelijkheden één voor één nagaan (*exhaustief zoeken*) of tevreden zijn met een suboptimale maar snelle oplossing waar je bij het zoeken nooit op je stappen terugkeert (*gretig algoritme*). Je kan je probleem opsplitsen in gelijkaardige problemen die je eerst oplost en dan later samenvoegt (*verdeel en heers*) – vaak met behulp van *recursie*, en dan kan je dit in sommige gevallen een stuk verbeteren door veel gebruikte tussenresultaten op te slaan in plaats van telkens opnieuw te berekenen (*memoization* en *dynamisch programmeren*).

[3] En helaas kan deze onwetendheid je programma minder efficiënt maken dan het zou kunnen zijn. Zie uitbreidingsoefening 1.1*.

In deze tekst beperken we ons voornamelijk tot het beschrijven van de meest voorkomende algoritmische technieken aan de hand van voorbeelden, en dan vooral in de latere hoofdstukken. Op enkele uitzonderingen na, valt het zelf kunnen bedenken van algoritmen gebaseerd op deze technieken, buiten het bestek van deze nota's.

1.7 Numerieke methodes

De eerste computers waren niet veel meer dan veredelde rekenmachines en hadden voornamelijk tot doel berekeningen die anders met de hand moesten worden gemaakt, sneller machinaal te kunnen uitvoeren. Grote aantallen berekeningen maken (*number crunching*) is tegenwoordig misschien niet meer de hoofdtaak van de meeste computers – hoewel bij computerspellen en kunstmatige intelligentie toch nog heel veel achter de schermen wordt gerekend – toch blijft het belangrijk te weten hoe in de praktijk numerieke gegevens worden verwerkt.
Hoewel we in deze nascholingen enkele numerieke algoritmen bespreken – reeds vanaf de eerste les – is hun aantal eerder beperkt. Bij implementatie van een numeriek algoritme is de voornaamste moeilijkheid immers vaak om een (ingewikkelde) wiskundige formule te vertalen naar programmacode maar is de structuur van het eindresultaat uiteindelijk vrij eenvoudig – een for-lus die alle elementen van een lijst overloopt, of een while-lus die blijft doorgaan tot een bepaalde waarde klein genoeg wordt. Vanuit programmeer- en algoritmisch oogpunt is dit minder interessant.
Tot slot geven we ook nog mee dat wij het gebruik van de cosinus- of logaritmefunctie uit de `math`-module van Python niet zien als een toepassing van een numerieke methode, net zoals het gebruiken van een lijst nog geen algoritme is.

1.8 Pre-algoritmen

Voor een beginnende programmeur is de stap van het begrijpen van een algoritme naar het implementeren ervan soms groot, omdat algoritmen vaak programmeertechnieken gebruiken die in een basiscursus weinig aan bod komen of niet voldoende worden ingeoefend. Deze technieken plaatsen we hier onder de noemer *pre-algoritmen*. Enkele voorbeelden:

- Een for-lus die alle getallen van 1 tot 100 overloopt, is één ding. Iets anders wordt het als je de ene for-lus binnen de andere plaatst en als de grenzen van de binnenste lus dan bovendien nog afhangen van wat er in de buitenste lus gebeurt.
- In een basiscursus overloop je de elementen van een lijst meestal van voor naar achter, of een enkele uitzonderlijke keer van achter naar voor. Bij veel algoritmen worden lijstelementen in een andere volgorde benaderd. Zo kan er bijvoorbeeld afwisselend vooraan en achteraan in de lijst gekeken worden, waarbij er dan twee indices nodig zijn die opschuiven in de lijst. Dit vergt wat wennen.
- Zogenaamde *dictionaries* vormen een belangrijk onderdeel van de programmeertaal Python dat in een basiscursus meestal niet aan bod komt. Ook dit catalogeren we onder pre-algoritmen.

Enkele van de hoofdstukken in deze tekst zullen weinig nieuwe 'algoritmische' leerstof aanbrengen maar dienen precies om dit soort technieken beter in te oefenen.

1.9 Opzet van deze tekst

Voor de leerinhoud die we in deze tekst behandelen verwachten we dat de lezer beschikt over een zekere basiskennis van de programmeertaal Python, bijvoorbeeld zoals die in Module 1 is gegeven[4], wat niet wegneemt dat we ook hier nog enkele bijkomende aspecten van de programmeertaal zullen behandelen.
De tekst is onderverdeeld in verschillende hoofdstukken. Het staat je als lesgever vrij om daar een eigen keuze uit te maken – niet alle hoofdstukken hoeven zelfs volledig te worden behandeld. De lesgever die een 14-tal lesuren ter beschikking heeft – naast het 7-tal lesuren dat nodig is om een basiskennis programmeren bij te brengen – zal wellicht (slechts) een tiental hoofdstukken kunnen behandelen[5].

---

[4]De lesnota's van module 1 staan ter beschikking op dezelfde website als deze tekst.
[5]In een eerdere versie hadden we bij elk hoofdstuk aangegeven hoeveel tijd je kon verwachten eraan te moeten be-

Latere hoofdstukken maken soms gebruik van concepten die in eerdere hoofdstukken zijn aangebracht. We stellen daarom voor dat je toch zeker in het begin min of meer de voorgestelde volgorde aanhoudt. In het begin van elk hoofdstuk geven we aan welke de competenties zijn die in dat hoofdstuk worden behandeld. Dit kan als een leidraad dienen. Er is bovendien een afzonderlijk document met een aantal uitbreidingsoefeningen bij de verschillende hoofdstukken. Je kan die gebruiken als differentiatie of als aanvulling[6].
De oefeningen zijn genummerd per hoofdstuk (2.1, 2.2, …) – de uitbreidingsoefeningen krijgen een ster in hun nummering (2.1*,2.2*,…).
Bij opgaven en oefeningen, of bij vragen die we in deze tekst formuleren, plaatsen we vaak enkele aandachtspunten in deze kleur en dit lettertype. Dit is informatie die je in de klas kan gebruiken bij het bespreken van de oefeningen en de oplossingen van de leerlingen.
Deze tekst staat niet alleen. Voor veel oefeningen en opgaven vertrekken we van bestaande broncode of heb je bijkomende hulpbestanden nodig. Deze bestanden (en trouwens ook deze tekst) vind je op de website https://inigem.ugent.be/progalg.html. Daar staan ook de oplossingen van de meeste oefeningen. Als we dus een (Python-)bestand vermelden (in dit lettertype: *4_lotr.py)* dan doelen we op een bestand dat je vanaf die website kunt downloaden. De bestanden zijn gegroepeerd in mapjes per hoofdstuknummer (*01*, *02*, …) met een toegevoegde 'u' voor de uitbreidingsoefeningen (*01u*, *02u*, …). Je kan ook een ZIP-archief downloaden met daarin alle bestanden.
Dit document is bedoeld voor de leerkracht. Nota's die specifiek naar de leerlingen gericht zijn, komen wellicht ter beschikking vanaf juli 2024.

1.10 Software

Voor de nascholingen kozen we Thonny (https://thonny.org/) als programmeeromgeving. Naast de standaardinstallatie heb je ook de volgende 'pakketten' (externe Python-bibliotheken) nodig:

```
matplotlib, pillow
```

Je kan die installeren via het menu *Hulpmiddelen → Pakketten beheren*.
Je mag gerust een andere programmeeromgeving gebruiken, er is niets in deze tekst dat expliciet naar Thonny verwijst. We raden je echter aan om iets te kiezen dat niet al te gesofisticeerd is – dus liever geen *PyCharm*, *Eclipse* of *Visual Studio*. Beginnende programmeurs behoeven een lage instapdrempel en hebben er baat bij zelf te moeten nadenken over elke stap en niet meteen geholpen te worden door de AI van de programmeeromgeving. De zeer gebruiksvriendelijke debugger van Thonny is wel een pluspunt.

1.11 Licentie

De nota's en bijbehorende bestanden vallen onder de *Creative Commons*-licentie 'Naamsvermelding Niet-Commercieel GelijkDelen 4.0 Internationaal' (CC BY-NC-SA 4.0). Je mag dit werk 'remixen', veranderen en voor niet-commerciële doeleinden hergebruiken, zolang de oorspronkelijke auteurs vernoemd worden en zolang je uw creaties onder een identieke licentie aanbiedt.

Juridische informatie vind je op https://creativecommons.org/licenses/by-nc-sa/4.0/deed.nl.

steden. Uiteindelijk hebben we, na terugkoppeling door deelnemers aan de nascholingen, deze informatie weggelaten omdat ze te veel afhangt van het beoogde leerlingenpubliek en van de andere minimumdoelen informaticawetenschappen die ze ook nog dienen te behalen.
[6]Achteraf bekeken is het onderscheid tussen 'gewone' oefeningen en uitbreidingsoefeningen eerder artificieel. Er zijn heel wat plaatsen in deze tekst waar een uitbreidingsoefening evengoed dienst doet als de oefening die er nu staat.

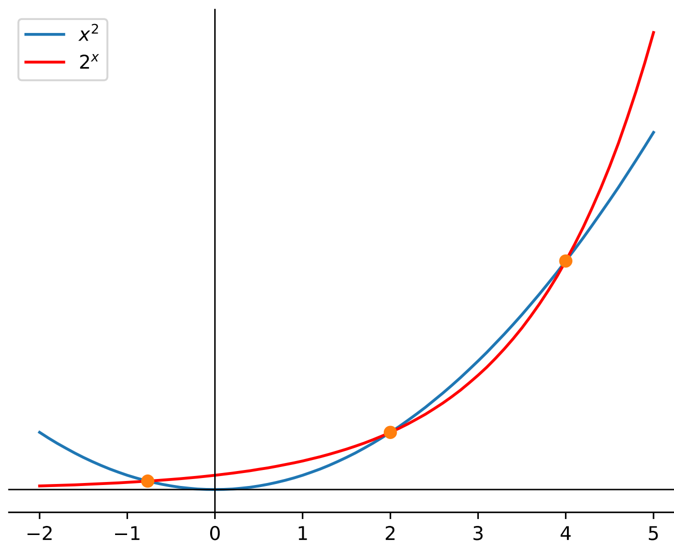
2 Zoeken

Overzicht

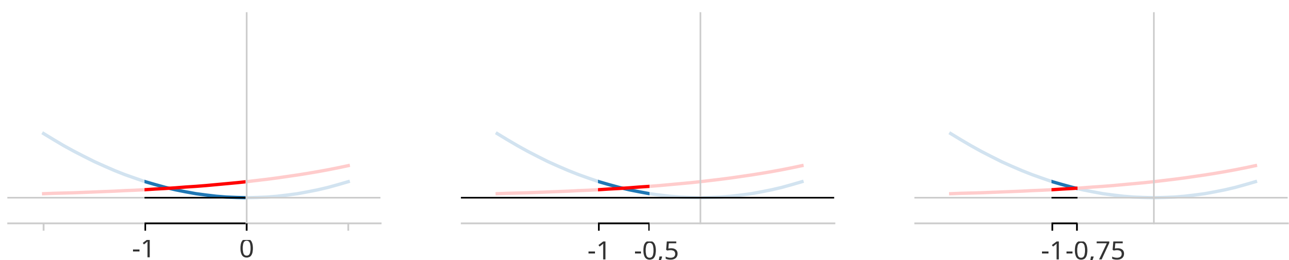
Algoritmen en datastructuren: binair zoeken
Numerieke methodes: bisectiemethode voor het vinden van nulwaarden

2.1 Bisectiemethode

Welk getal x heeft de eigenschap dat 2^x gelijk is aan x^2? Je ziet al snel dat x=2 en x=4 hieraan voldoen, maar is er nog een andere oplossing? Ook als x niet geheel hoeft te zijn en niet positief? Als we de grafieken van de functies $f(x)=2^x$ en $g(x)=x^2$ tekenen in hetzelfde assenstelsel, dan zien we dat er nog een oplossing is, met x iets kleiner dan -0,75. We willen graag deze x bepalen tot 12 cijfers na de komma.



Er is een eenvoudige numerieke methode om dergelijke problemen op te lossen en die werkt als volgt: we bepalen telkens kleinere intervallen waarin het gezochte snijpunt zich bevindt, elk interval half zo groot als het vorige, en we blijven dit doen tot het interval klein genoeg is om de gevraagde precisie te bekomen.



In dit voorbeeld zijn de opeenvolgende intervallen de volgende

[-1, 0], [-1, -0,5], [-1, -0,75], [-0,875, -0,75], [-0,8125, -0,75], …

Elk interval is half zo groot als het vorige en heeft er één eindpunt mee gemeen, soms het linker eindpunt en soms het rechter. Hoe weten we welk eindpunt we moeten kiezen? Anders gezegd, welke helft moeten we telkens kiezen wanneer we het interval halveren?

We moeten telkens de helft kiezen die het snijpunt bevat, maar dat snijpunt kennen we helaas niet. Wel weten we dat links van het snijpunt de rode grafiek *onder* de blauwe ligt en rechts van het snijpunt de rode grafiek *boven* de blauwe. We kiezen dus telkens de helft op zo'n manier dat deze eigenschap blijft gelden. (In de 'verkeerde' helft ligt de blauwe kromme volledig boven de rode, of omgekeerd.) Om te weten welke kromme bovenaan ligt en welke onderaan, gebruiken we de functiewaarden 2^x en x^2 in beide eindpunten, m.a.w., wanneer x de waarde heeft van het linker eindpunt en wanneer x de waarde heeft van het rechter eindpunt. Als je er even bij stilstaat, zie je snel dat je zelfs enkel de functiewaarden hoeft te berekenen wanneer x de waarde heeft die overeenkomt met het *midden* van het interval.

Oefening 2.1 Schrijf een programma dat de gezochte X-waarde afdrukt tot op 12 cijfers na de komma.

Enkele tips:

- Gebruik variabelen `links`, `rechts` en `midden` waarin je de linker en rechter X-waarde van het interval bijhoudt, en de X-waarde van het midden van het interval
- Welke lus gebruik je hier best? Een for-lus of een while-lus?
- Wanneer stopt de lus? Hoe geef je dit aan in het programma?
- Wat is de voorwaarde waarmee je bepaalt of je met de linker helft of de rechter helft van het interval verder doet?
- Wat zijn de linker en rechter X-waarde van het nieuwe interval? Uit welke variabelen kan je die halen, en in welke variabelen moet je die stoppen?

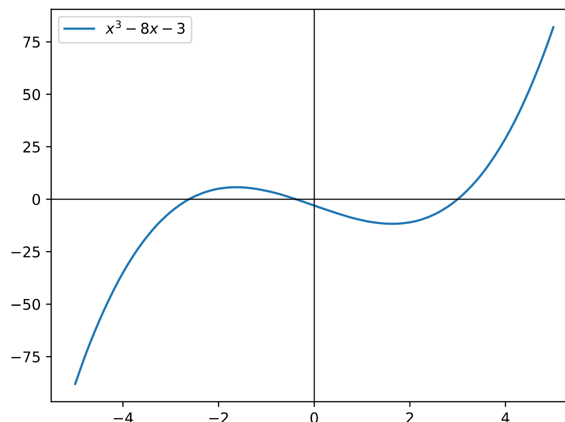
Wetenschappelijke notatie

Voor hele grote of hele kleine kommagetallen mag je in Python *wetenschappelijke notatie* gebruiken. Het getal $1{,}2\times10^{23}$ noteer je dan als `1.2E23` en in de plaats van 0,000001 schrijf je `1.0E-6` of `1E-6`.

De X-waarde die we zoeken is *x*=-0,766664695962163.

Op (bijna) dezelfde manier kan je de *nulwaarde* zoeken van een functie in een bepaald interval.

Oefening 2.2 Hieronder zie je de grafiek van de functie $f(x) = x^3 - 8x - 3$.



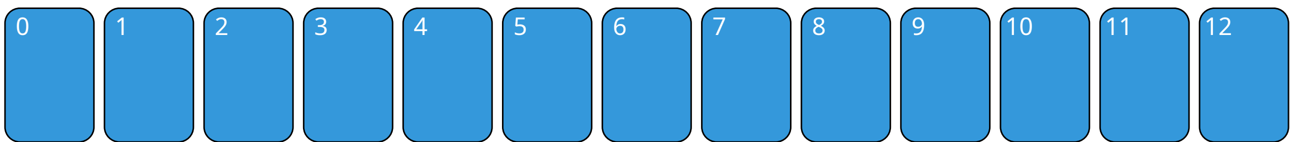
De grafiek snijdt de X-as op drie plaatsen, dus zijn er drie nulwaarden. Schrijf een programma dat de kleinste (de meest linkse) nulwaarde zoekt[7]. Dit vraagt niet veel meer dan enkele kleine aanpassingen aan de oplossing van oefening 2.1.

De X-waarde die we zoeken is x=-2.618033988749 = -(3+√5)/2

2.2 Binair zoeken

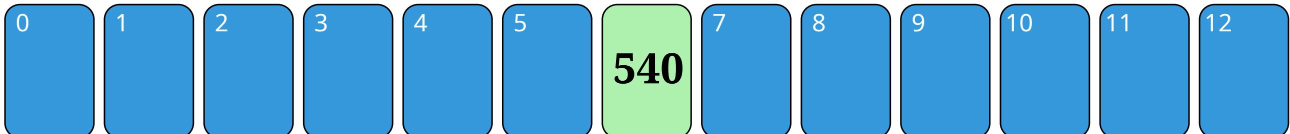
Open het bestand *bs.html* in een browser. De pagina toont een aantal kaarten met op elke kaart een positief geheel getal. De kaarten liggen met de rug maar boven. Klikken op een kaart draait ze om.

[7]Het zoeken van *alle* nulwaarden is uitbreidingsoefening 2.1. De grootste nulwaarde kan je trouwens gemakkelijk vinden *zonder* programma, en dan met een klein beetje wiskundig denken, ook de andere twee.

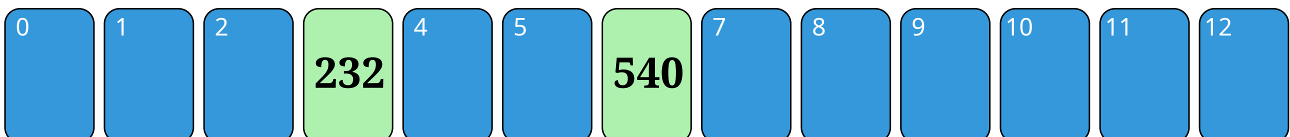


Je krijgt een nummer en moet de kaart vinden waarop dat nummer staat. De enige aanwijzing is dat de kaarten van klein naar groot zijn gerangschikt. Hoe snel kan je de juiste kaart vinden?
Je kan de kaart heel snel vinden als je de volgende strategie gebruikt.

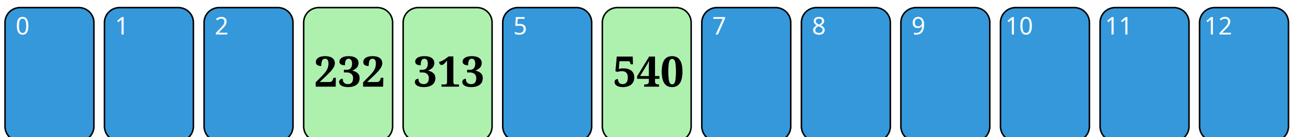
Draai de middelste kaart om. (In dit voorbeeld zoeken we naar de kaart waarop 503 staat[8].)



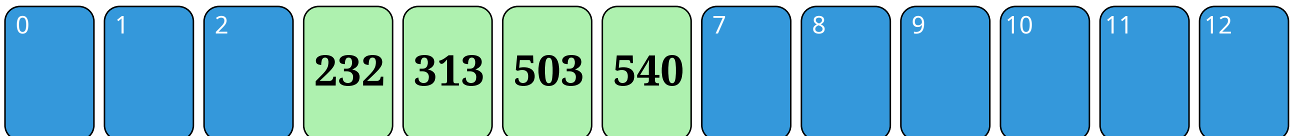
Omdat de kaarten van klein naar groot gerangschikt zijn – en dat is hier heel belangrijk! – weten we dat de kaart met 503 links van het midden ligt. We hoeven dus enkel nog naar de kaarten te kijken met nummer 0 tot 5. We draaien nu kaart 3 om, want die ligt (ongeveer) in het midden van het gebied [0-5].



Het getal 503 is groter dan 232, dus ligt de gezochte kaart nu in het ‘interval’ [4-5]. We kiezen 4.



Helaas was dit niet de juiste keuze. Onze kaart is dus nummer 5.



Na 4 keer klikken, hebben we het getal gevonden.
Deze strategie heet *binair zoeken*. Komt dit je bekend voor? Dat is niet te verwonderen. Wat we hier met de kaarten doen lijkt heel goed op de bisectiemethode uit §2.1. Opnieuw hebben we een ‘interval’ waarbinnen we zoeken en dat we bij elke stap halveren. Dit zijn de opeenvolgende intervallen voor onze laatste zoektocht:

[0,12], [0,5], [4,5], [5,5]

Doordat we telkens de intervalgrootte (minstens) halveren – de opeenvolgende groottes zijn 13, 6, 2 en 1 – gaat het zoeken zeer snel. We zouden eender welke van de 13 kaarten in minstens 4 stappen hebben kunnen vinden. In hoeveel stappen zouden we elke kaart uit 27 kaarten kunnen vinden? Uit honderd kaarten? Uit duizend?

Na 1 keer halveren is de stapel van 27 kaarten teruggebracht naar één van 13 kaarten, het vorige geval. We hebben dus hoogstens 1 stap meer nodig, t.t.z., 4+1=5 stappen. Honderd kaarten worden er 50 na één stap en 25 na twee stappen, dus 5+2=7 stappen is voldoende. Bij duizend kaarten volstaan 10 stappen, en in het algemeen voor N kaarten $\log_2 N$ stappen.

Oefening 2.3 Het programma hieronder (3_*binair_zoeken.py)* zoekt in een gegeven lijst een getal dat door de gebruiker wordt ingetikt. De functie `index_in_lijst(getal)` geeft de index terug waar `getal` in die lijst te vinden is, of -1 als het getal niet in de lijst zit. (Er ontbreken nog wel enkele stukken…)

[8]De webpagina toont telkens andere getallen en vraagt je ook altijd om een ander getal te vinden.

```python
# Lijst van ± 200 getallen tussen 0 en 1000.
# De getallen staan gerangschikt van klein naar groot
lijst = […]

def index_in_lijst (getal):
    links  = ???
    rechts = ???
    while links < rechts:
        midden = ???
        if getal == lijst[midden]:
            return midden
        elif getal < lijst[midden]:
            rechts = ???
        else:
            links  = ???
    return -1

# programma
gezocht_getal = int(input("Welk getal zoek je? "))
index = index_in_lijst(gezocht_getal)
if index < 0:
    print (f"{gezocht_getal} zit niet in de lijst")
else:
    print (f"{lijst[index]} heeft index {index} in de lijst")
```

De structuur van de functie `index_in_lijst` lijkt heel goed op wat we in §2.1 hebben geprogrammeerd. Toch zijn er enkele belangrijke verschillen:

- De eindpunten van de intervallen zijn geen kommagetallen maar gehele getallen. De lengte van zo'n interval is daarom niet altijd precies deelbaar door 2. Hou daar rekening mee.
- Het blijkt in de praktijk eenvoudiger om het interval te gebruiken dat `links` *wel* bevat, maar `rechts` (net) *niet* – alsof je de slice `lijst[links:rechts]` bedoelt.
- Merk op dat de if-opdracht[9] hier drie mogelijkheden onderzoekt in plaats van twee. We springen uit de while-lus zodra we het getal gevonden hebben[10]. Dat betekent ook dat je het midden zelf niet in het volgende interval hoeft op te nemen.

Vul de vraagtekens (`???`) aan in het programma. Test het op verschillende getallen waarvan je weet dat ze in de lijst zitten, maar ook op verschillende getallen waarvan je weet dat ze er niet in zitten, ook negatieve getallen en getallen groter dan 1000!

Je hebt een *gehele* deling door 2 (met `//`) nodig om het midden te bepalen. Beide intervallen zijn dan niet altijd precies even groot, maar dat geeft niet. Bij het rechter interval neem je `links=midden+1`, anders loop je kans dat het programma in een oneindige lus terecht komt wanneer het getal *niet* in de lijst zit. Merk ook op dat de laatste lijn van het programma `lijst[index]` afdrukt en niet `gezocht_getal`, als bijkomende controle.

Binair zoeken

Binair zoeken is een techniek (algoritme) waarmee je *snel* een element in een lijst[11] terugvindt aan de hand van zijn *sleutel*, op voorwaarde dat de lijst volgens die sleutel is geordend. Dit algoritme is zo performant omdat het bij elke stap de zoekruimte halveert.

---

[9]Soms hoor je iemand over een if-*lus* spreken. Dat klopt niet: 'lus' impliceert herhaling en bij een if-opdracht wordt er niets herhaald.

[10]Uit een lus te springen raden we doorgaans af. Vaak kan je het programma herschikken zodat dit kan vermeden worden – zie hoofdstuk ?. In dit specifieke geval wordt de code daardoor misschien te onleesbaar en daarom laten we het hier oogluikend toe.

[11]Dit werkt enkel wanneer we snel het *n*-de element van een lijst kunnen ophalen, onafhankelijk van de waarde van *n*. Er bestaan lijstimplementaties (zoals de zogenaamde *gelinkte lijst*) die die eigenschap niet hebben.
Binair zoeken werkt trouwens even goed op arrays.

3 Postnummers

Overzicht

Algoritmen en datastructuren: lineair zoeken, geordende lijst, complexiteit meten
Externe gegevensbronnen: invoer uit een tekstbestand
Softwarebibliotheken: module `time` voor tijdsmetingen

3.1 Gesorteerd invoegen

Tijdens een vakantiejob in één of ander stoffig hoekje van een ministerie moet je vaak de naam van een gemeente opzoeken aan de hand van zijn postnummer. Om je te helpen krijg je een papieren lijst waarop alle gemeenten staan opgelijst met hun postnummer. Welke lijst verkies je?

- Een lijst alfabetisch gerangschikt volgens de naam van de gemeente, of
- een lijst numeriek gerangschikt volgens postnummer, of
- een lijst gerangschikt volgens familienaam van de burgemeester van de gemeente.

Je keuze is snel gemaakt: als de lijst op de juiste manier gerangschikt is, gaat het veel sneller om er iets in op te zoeken. Niet alleen voor een mens, maar ook voor een computer – die dan wellicht het binair zoeken uit §2.2 zal gebruiken. De inhoud van een lijst geordend houden levert dus een belangrijk voordeel op als je snel en efficiënt wil werken!

Geordende lijst

Een geordende lijst is een eenvoudige datastructuur die men gebruikt om snel gegevens terug te vinden – een stuk sneller dan in een lijst waar de gegevens zomaar door elkaar staan. Het nadeel van een geordende lijst is dat het wat meer tijd vraagt om een nieuw element aan die lijst toe te voegen. Je kan het element niet zomaar achteraan plaatsen omdat de lijst dan niet langer correct geordend is, en je dan het voordeel van snel zoeken terug verliest.

Een lijst geordend houden terwijl je ze invult, is niet moeilijk: je zoekt telkens de plaats in de lijst waar het element thuishoort en voegt het element er dan op die plaats tussen.
Je kan die plaats best 'binair' zoeken – want dat gaat snel. Je kan echter niet helemaal dezelfde strategie volgen als in §2.2 – de methode `index_in_lijst` zou dan altijd -1 teruggeven. We doen een kleine aanpassing:

```
def index_in_lijst (getal):
    links = 0
    rechts = len(lijst)
    while links < rechts:
        midden = (links + rechts) // 2
        if element <= lijst[midden]:
            rechts = midden
        else:
            links = midden + 1
    return links
```

(Er is één geval uit de if-opdracht verdwenen en de laatste regel is ook aangepast.)
Het is niet vanzelfsprekend dat deze functie precies doet wat we ervan verwachten. Probeer dus 'met de hand' uit wat er gebeurt wanneer we `index_in_lijst` oproepen met `getal` gelijk aan achtereenvolgens 0,1,2,3,…,12 voor de lijst [1,3,5,7,9,11] – en controleer dat dit in elk van die gevallen het correcte resultaat is. Hieronder tonen we dat dit alvast klopt voor 9 en 4.

Oefening 3.1 (Vertrek vanaf *1_postnummers.py.)* Het tekstbestand *postnummers-1.txt* bevat een lijst van alle Belgische gemeenten met hun postnummers:

```
3798 's Gravenvoeren
2970 's Gravenwezel
3700 's Herenelderen
9420 Aaigem
8511 Aalbeke
…
(2920 lijnen)
```

Helaas zijn ze geordend op naam van de gemeente en willen we liever een lijst die gerangschikt is op postnummer.
Lees het tekstbestand in en stop de gemeenten in een lijst die je geordend houdt op postnummer. Zoek daarna enkele gemeenten op om te zien of de lijst wel in de juiste volgorde staat. Druk eventueel de uiteindelijke lijst af om helemaal zeker te zijn.
De leerlingen kunnen ook een eigen gemeente opzoeken. Dit zal echter niet altijd perfect lukken wanneer er meerdere gemeenten zijn met hetzelfde postnummer, omdat dan de alfabetisch eerste wordt getoond. Zolang het juiste postnummer wordt gevonden, is hun oplossing wellicht correct.
Als je op het einde alle gemeenten opnieuw laat afdrukken, komt er tussen elke gemeente een extra lege lijn. Dit komt omdat elke string die de lijn voorstelt op het einde een *new line*-teken bevat. Zie inzet.

Tekstbestand inlezen

Om alle lijnen uit een tekstbestand één voor één in te lezen en te verwerken, schrijf je

```
with open("mijnbestand.txt", "r") as bestand:
    for lijn in bestand:
        # doe iets met lijn
        …
```

(waarbij je natuurlijk '*mijnbestand.txt*' vervangt door de echte naam van het bestand).
De variabele `lijn` bevat achtereenvolgens de verschillende lijnen van het bestand, als een string. Die string bevat achteraan een extra *new line*-teken, maar voor oefening 3.1 is dit niet van belang.
De `"r"` bij open geeft aan dat we het bestand willen *lezen* (r = read).

3.2 Lineair zoeken

De gewone manier van zoeken in een lijst door ze van voor naar achter te doorlopen, noemen we *lineair zoeken*. We gaan in deze paragraaf wat dieper in op het verschil in snelheid tussen lineair en binair zoeken.
Opgave Stel je telkens de volgende drie vragen:

1. Hoeveel elementen in de lijst moet je bekijken in het *beste* geval, m.a.w., als je geluk hebt en je juist het ding zoekt dat het minste werk vraagt?
2. Hoeveel elementen in de lijst moet je bekijken in het *slechtste* geval?
3. Hoeveel elementen gemiddeld – bijvoorbeeld als je 100 keer naar elementen zoekt die her en der in de lijst te vinden zijn.

Doe dit voor de volgende gevallen, telkens met lineair zoeken in een lijst van 1000 elementen

1. Als de lijst niet geordend is en het element dat je zoekt, zit in de lijst.
   Beste: 1, slechtste: 1000, gemiddelde: ± 500
2. Als de lijst niet geordend is en het element dat je zoekt, zit *niet* in de lijst. M.a.w., hoeveel elementen moet je bekijken vooraleer je doorhebt dat het niet in de lijst zit?
   Beste: 1000, slechtste: 1000, gemiddelde: 1000

3. Als de lijst geordend is en het element zit in de lijst.
   Zelfde als bij 1.
4. Als de lijst geordend is en het element zit niet in de lijst.
   Zelfde als bij 1. en 3.

En wat is het antwoord (ongeveer) wanneer je binair zoeken gebruikt op een geordende lijst van 1000 elementen?

Als je de laatste versie gebruikt (§3.1 – oefening 3.1) moet je in elk van de gevallen 10 elementen bekijken. Bij de voorlaatste versie (§2.2 – oefening 2.3) kan het in het beste geval minder zijn: 1, als wat je zoekt precies in het midden van de lijst staat, en telkens 10 wanneer het niet in de lijst zit. Het gemiddelde is ± 9 als het element erin zit, maar dat is niet zo gemakkelijk te berekenen.

Hier was het nog vrij eenvoudig om in te zien dat de binaire zoekmethode een stuk sneller is dan lineair zoeken. In de praktijk is het niet altijd zo eenvoudig om te voorspellen welke van twee methodes de snelste is. Er zit dan niets anders op dan het uit te proberen en de tijd te meten die beide nodig hebben om dezelfde taak uit te voeren.

Tijd meten

Met de functie `time()` uit de module `time` meet je de tijd die verloopt tussen twee punten in je programma. De functie geeft een *tijdstempel* terug – het aantal seconden dat verlopen is sinds 1 januari 1970, middernacht.
Meestal gebruik je `time()` twee keer: één keer vóór een programmafragment en één keer erna. Het verschil tussen de twee tijdstempels geeft dan de tijdsduur van dat fragment.

Oefening 3.2 (Vertrek van je oplossing van oefening 3.1 en van *2_postnummers.py*)
De functie `index_van` gebruikt *lineair* zoeken om een element in een lijst te vinden:

```
def index_van (element,lijst):
    index = 0
    while index < len(lijst) and lijst[index] < element:
        index += 1
    return index
```

Meet de tijd die het duurt om de geordende lijst met postnummers te maken met lineair zoeken en met binair zoeken. Merk je een duidelijk verschil?
Binair zoeken blijkt hier 10 à 40 keer sneller dan lineair zoeken – afhankelijk van de computer.

Modules

Om de functies uit een Python-module (= bibliotheek) te gebruiken, moet je die importeren. Je kan dit op verschillende manieren doen. De opdracht

```
import math
```

zorgt er bijvoorbeeld voor dat je vanaf nu `math.cos(hoek)` kunt schrijven om de cosinus van een hoek te bepalen. Wil je niet telkens het voorvoegsel '`math.`' herhalen, dan kan je deze opdracht gebruiken:

```
from math import cos, sin
```

Dit laat je toe om cosinus en sinus van een hoek voortaan af te korten tot `cos(hoek)` en `sin(hoek)`.

Postscriptum (voor de leerkracht)
Bij een gecompileerde programmeertaal (zie opmerking bij §1.2) is de snelste manier om een lijst geordend te houden niet wat we hier in oefening 3.1 hebben uitgewerkt. In de plaats zoekt men lineair van achter naar voor en verplaatst men ondertussen de elementen die men tegenkomt telkens één positie naar achteren.

```
def index_van (element,lijst):
    index = len(lijst)
    lijst.append("") # wordt hoe dan ook overschreven
    while index > 0 and lijst[index-1] > element:
        lijst[index] = lijst[index-1]
        index -= 1
    lijst[index] = element
```

Dit vormt ook de basis van het algoritme *sorteren door invoegen* dat we behandelen in §9.2.

4 Poker

Overzicht

Pre-algoritmen: meervoudige selectie vermijden, while met dubbele conditie
Numerieke methoden: simulatie, Monte Carlomethode
Softwarebibliotheken: module `random` voor willekeurige getallen

In dit hoofdstuk gebruiken we de computer om in te schatten hoe groot de kans is om bij Poker een *kwartet* (Engels: *four of a kind)* in de hand te krijgen – vier kaarten met dezelfde *waarde*, één voor elke *kleur*, zoals de vier heren in de foto hiernaast.
We doen dit met een *simulatie*: we laten de computer nadoen dat we telkens opnieuw 5 willekeurige kaarten trekken uit een (nieuw) pak van 52 kaarten en tellen hoeveel keer we daarbij vier dezelfde kaarten vinden. We doen dit een miljoen keer, zodat we een goede schatting krijgen van de werkelijke kans.
We werken het programma stap voor stap uit.

4.1 Speelkaarten

Je kan speelkaarten in een computerprogramma op veel verschillende manieren voorstellen. Hier zullen we getallen gebruiken, genummerd zoals hiernaast.
Straks zullen we het nodig hebben te weten of twee kaarten dezelfde waarde hebben. Hoe kan je aan de nummers van twee kaarten zien of ze dezelfde waarde hebben? Of ze dezelfde kleur hebben?
Je deelt het nummer door 13. Geeft dit dezelfde rest (%13) dan hebben de kaarten dezelfde waarde. Geeft dit hetzelfde quotiënt (//13) dan hebben ze dezelfde kleur. Je vindt enkele voorbeelden hiernaast in de tabel.

0	Hartenaas
1	Hartentwee
2	Hartendrie
…	
12	Hartenheer
13	Ruitenaas
14	Ruitentwee
15	Ruitendrie
…	
25	Ruitenheer
26	Klaverenaas
27	Klaverentwee
28	Klaverendrie
…	
39	Schoppenaas
40	Schoppentwee
41	Schoppendrie
…	
48	Schoppentien
49	Schoppenboer
50	Schoppenvrouw
51	Schoppenheer

Bij het afdrukken van een kaart willen we zijn naam afdrukken en niet zijn nummer. Je kan hiervoor bijvoorbeeld deze functie gebruiken:

```
def kaart_naam (nummer):
    if nummer == 0:
        return "hartenaas"
    elif nummer == 1:
        return "hartentwee"
    # ... 52 gevallen
    else:
        return "schoppenheer"
```

Je voelt zelf wel aan dat deze if-opdracht met 52 verschillende gevallen wellicht niet de aangewezen manier is om dit te programmeren. Kan je een betere manier bedenken?
Dit kan op twee manieren verbeterd worden.

- Je kan de kleur (4 mogelijkheden) en waarde (13 mogelijkheden) van de kaart afzonderlijk omzetten naar een string, en dan samenvoegen.
- Je kan de naam 'opzoeken' in een tabel: het nummer van de kaart is dan de index in een lijst die je één keer op voorhand invult.

De beste implementatie combineert beide suggesties: je gebruikt een lijst met de vier kleurennamen en een lijst met de namen voor de 13 waarden.

Oefening 4.1 Schrijf een programma dat 5 willekeurige kaarten (kaartnummers) kiest en hun namen afdrukt. Dit is een mogelijke uitvoer. (Je kan wat tikwerk vermijden door te starten vanaf *1_vijf_kaarten.py*)

```
schoppentien
klaverenboer
ruitenaas
klaverenboer
hartendrie
```

Je hoeft er niet voor te zorgen dat alle kaarten verschillend zijn.

Willekeurige getallen

De Python-module `random` kan biedt een aantal functies om willekeurige getallen te genereren. Wij zullen hier vooral de functie `random.randint(a,b)` gebruiken waarmee je een willekeurig geheel getal opvraagt in het interval [a,b] – a en b meegerekend.
Omdat een computer enkel vaste 'recepten' kan volgen, zijn die getallen niet echt willekeurig – men spreekt ook van *pseudo*-willekeurige getallen – maar toch doet hij een zeer geslaagde poging om ze er zo te doen uitzien: als je 6000 keer met een 'pseudo-dobbelsteen' werpt[12] (= willekeurige getallen opvraagt uit [1,6]) dan zullen er ongeveer 1000 enen gegooid worden, 1000 tweeën, 1000 drieën, enz.

4.2 Four of a kind?

Oefening 4.2 Schrijf een Python functie `is_kwartet(lijst)` die kijkt of er in een lijst van vijf kaart(nummers) er (minstens) vier zijn die zelfde waarde hebben – en dan overeenkomstig `True` of `False` teruggeeft. Test dit uit op een aantal voorbeelden:

```
[0,12,13,39,26] → True   [1,41,3,17,8] → False   [12,50,37,11,24] → True
```

Hoe zou je kijken of alle 5 kaartnummers dezelfde waarde hebben? Gebruik dit als inspiratie.

Let erop dat je niet de kaartnummers zelf vergelijkt, maar wel hun rest bij deling door 13. Er zijn verschillende implementaties mogelijk. In de voorbeeldoplossing kijken we hoeveel keer het eerste element van de lijst in de lijst voorkomt, en hoeveel keer het tweede. Er is een kwartet als minstens één van beide getallen 4 is. (We doen hier niet de moeite om vroeger te stoppen wanneer we na 4 kaarten al een kwartet hebben gevonden – dat levert weinig op.)

4.3 Vijf verschillende kaarten

Oefening 4.3 Schrijf een functie `neem_5_kaarten()` die een lijst teruggeeft met daarin 5 willekeurige kaartnummers. Zorg er dit keer voor dat de kaarten allemaal verschillend zijn! Pas je oplossing van oefening 4.1 hiermee aan.
Er zijn een aantal manieren waarop je dit kan aanpakken (kies er één van)

1. Vraag telkens een willekeurig getal (= kaartnummer) op. Stop dit enkel in de lijst van kaarten die je aan het opbouwen bent als het er nog niet in zit. Blijf dit herhalen totdat de lijst 5 lang is.
2. Maak een lijst aan met daarin alle kaarten (= alle getallen van 0 t.e.m. 51 – zie inzet). Deze lijst stelt het *pak* kaarten voor. Je trekt een willekeurige kaart uit het pak en *verwijdert het uit het pak*. Doe dit 5 keer.

Tip

Je kan een *range* omzetten naar een lijst met behulp van `list(…)`. Dus: `list(range(1,100,2))` geeft je een lijst van alle oneven getallen van 1 t.e.m. 99.

De voorbeeldoplossingen bevatten misschien een aantal nieuwigheden: de '`not in`'-bewerking als tegengestelde van '`in`', de functie 'pop' voor lijsten, '`del`' om iets uit een lijst te verwijderen, '`_`' als naam voor de lusvariabele van een for-lus als je die voor de rest niet gebruikt, en '`list`' uit de inzet hierboven.
Er is een derde mogelijkheid die door de leerlingen misschien voorgesteld wordt: gebruik `random.shuffle` om het pak met kaarten te schudden, en neem dan de vijf eerste. De manier waarop de `shuffle` door Python intern wordt uitgevoerd, lijkt goed op optie 2 hierboven, maar dan voor 52 kaarten in plaats van 5. M.a.w., deze derde optie is minder efficiënt, en zelfs het feit dat `shuffle` een interne procedure is, compenseert dat niet (zie ook §1.2).

[12] Zie uitbreidingsoefening 4.4*.

4.4 Alles samen

Oefening 4.4 (Voegt alles samen). We zouden bijna vergeten wat het doel was van dit hoofdstuk. Maak een schatting van de kans op *four of a kind* in een Pokerhand met 5 kaarten, zoals we in de inleiding van het hoofdstuk hebben geschetst.
De kans is ±0,024%

Monte Carlomethode

Wanneer een probleem te lastig is om direct wiskundig aan te pakken of wanneer het in de praktijk te moeilijk is om alle mogelijke vormen ervan uit te proberen[13], kan men de oplossing vaak benaderen door het probleem onder heel wat verschillende (willekeurige) begincondities te simuleren. Vaak maakt men hier gebruik van in de kansrekening en de statistiek.
Men noemt dit de *Monte Carlo*methode naar het Monte Carlo Casino in Monaco met zijn vele kansspelen.

Tot slot We konden de kaarten in het programma ook voorstellen als strings (`"hartenaas"`…`"schoppenheer"`). Waarom was het voordelig om hier gehele getallen 0…51 te gebruiken? Was er ook een nadeel?
Het belangrijkste voordeel is wellicht dat het vrij gemakkelijk (en efficiënt) is om te bepalen of twee kaarten dezelfde waarde hebben, of dezelfde kleur, wanneer je ze door getallen voorstelt – dat is ook waarom we bij 0 beginnen tellen en niet bij 1. Een willekeurig getal kiezen uit een bepaald bereik lijkt op het eerste zicht eenvoudiger dan een willekeurige string kiezen uit een lijst, maar dat is niet echt zo: er bestaat een functie `random.choice(lijst)` die dit laatste doet. En als we de naam van een kaart willen afdrukken, kan dit directer wanneer de kaart reeds als een string is opgeslagen.

[13]Ons ‘probleem’ kan echter wel wiskundig aangepakt worden (de kans is exact 1/4165) en het is ook haalbaar om alle mogelijkheden uit te proberen – zie uitbreidingsoefening 4.3*.

5 Chatbot

Overzicht

Pre-algoritmen: Python dictionary
Algoritmen en datastructuren: associatieve array (*map*)
Softwarebibliotheken: module `random` voor willekeurige getallen

5.1 Dictionaries

Hieronder zie je drie tabellen. Elke tabel heeft twee kolommen. De eerste kolom bevat telkens een *sleutel*, de tweede een *waarde* die bij die sleutel past – het e-mailadres (waarde) van een vriend (sleutel), de top-score (waarde) van een gamer (sleutel) en de lijst van beoordelingscijfers (waarde) die je hebt binnengekregen voor een film (sleutel).

sleutel	waarde
john	johnandjane@email.com
noah	noah123@hotmail.com
olivia	olivia.vdg@gmail.com
mila	mila@cardassia.org
louis	louisXIV@soleil.gov.fr
jane	johnandjane@email.com

E-mailadressen

sleutel	waarde
Striker	1 200 350
Phoenix	1 500 610
B0rg	1 200 350
Stealth	1 000 730
Knite	1 400 310
Thund3r	2 100 120

Topscore

sleutel	waarde
Divergent	[4.3, 2.5, 3.1]
Inception	[4.0, 4.5, 5.0, 4.2]
Jumanji	[3.5, 4.5, 3.8, 4.9, 2.7]
Twilight	[3.0]
Blade	[4.0, 3.2, 4.8]
Eight Grade	[ ]

Filmbeoordelingen

In Python kan je zo'n tabel voorstellen als een zogenaamde *dictionary*. Merk op:

- Sleutels in een dictionary zijn doorgaans strings, maar
- waarden kunnen allerlei types hebben – strings, getallen, lijsten, enz.
- De sleutels zijn allemaal *verschillend* – bij elke sleutel hoort precies één waarde, maar
- de waarden hoeven niet verschillend te zijn. Twee verschillende sleutels kunnen dezelfde waarde hebben – John en Jane hebben bijvoorbeeld hetzelfde e-mailadres.

Een dictionary dient om gemakkelijk en snel de waarde op te zoeken die bij een bepaalde sleutel hoort. Het omgekeerde – sleutels zoeken die bij een gegeven waarde horen – is niet eenvoudig en bovendien traag.
Een dictionary maak je aan met een accolade- en dubbelepuntnotatie (zie ook *0_dictionaries.py*).

```
emails = {
    "john"  : "johnandjane@email.com",
    "noah"  : "noah123@hotmail.com",
    "olivia": "olivia.vdg@gmail.com",
    "mila"  : "mila@cardassia.org",
    "louis" : "louisXIV@soleil.gov.fr",
    "jane"  : "johnandjane@email.com"
}
```

Je vraagt waarden op met vierkante haken (met ertussen de sleutel).

```
sleutel = "john"
waarde = emails[sleutel]
print (f"John's e-mailadres: {waarde}")
# meer direct:
waarde = emails["john"]
print (f"John's e-mailadres: {waarde}")
```

Je gebruikt ook vierkante haken om de waarde aan te passen die bij een sleutel hoort.

```
emails["noah"] = "noah321@hotmail.com"
topscores["Phoenix"] += 90_800
beoordelingen["Jumanji"].append(4.5)
```

Toevoegen van een sleutel die nog niet bestaat, met zijn waarde, doe je op dezelfde manier.

```
topscores["Giant"] = 900_230
```

Een dictionary mag leeg zijn. Je kan er achteraf nog altijd nieuwe sleutel/waardeparen aan toevoegen.

```
nl_fr = {}
nl_fr["boek"]  = "livre"
nl_fr["tafel"] = "table"
nl_fr["vuur"]  = "feu"
print (nl_fr) # drukt de ganse dictionary af
```

Een waarde opvragen voor een sleutel die in de dictionary niet bestaat, geeft jammer genoeg een foutmelding en sluit het programma af. Je kan op voorhand controleren of een sleutel aanwezig is met ‘`in`’.

```
zoekterm = "Jumangi" # tikfoutje :-(
if zoekterm in beoordelingen:
    print (beoordelingen[zoekterm])
else:
    print ("Beoordeling is niet gekend")
```

Associatieve array

Een associatieve array, ook kortweg *map* genoemd (Engelse uitspraak), is een datastructuur die sleutels met waarden verbindt op een manier die toelaat om heel snel aan de hand van een sleutel de betreffende waarde op te vragen. Python dictionaries zijn associatieve arrays.
Intern worden deze datastructuren vaak geïmplementeerd als hash-tabellen (zie hoofdstuk ?) maar ook andere zoekstructuren worden gebruikt.

Oefening 5.1. Het Engelse woord *dictionary* betekent letterlijk ‘woordenboek’. Als eerste toepassing leggen we daarom een woordenboek aan dat Nederlandse woorden (de sleutels) kan vertalen naar het Frans (de waarden).
Begin met een dictionary met daarin alvast een drietal vertaalde woorden. Laat de gebruiker een Nederlands woord intikken en druk dan de vertaling af. Zit het woord niet in het woordenboek, vraag dan de correcte Franse vertaling en stop die in het woordenboek. Stop het programma zodra de gebruiker ‘bye’ intikt[14].

```
Geef een Nederlands woord: vuur
Het woord 'vuur' is 'feu' in het Frans
Geef een Nederlands woord: tafel
Het woord 'tafel' is 'table' in het Frans
Geef een Nederlands woord: bed
Ik ken de vertaling niet van 'bed'
Geef de Franse vertaling: lit
Geef een Nederlands woord: bed
Het woord 'bed' is 'lit' in het Frans
Geef een Nederlands woord: bye
```

[14]Helaas weet je nog niet hoe je het woordenboek dat je zorgvuldig hebt opgebouwd in een bestand kunt opslaan zodat je niet ‘van nul’ hoeft te beginnen wanneer je het programma de volgende keer opstart. Dit leer je in hoofdstuk 7.

5.2 Chatbot

De idee voor deze chatbot komt bijna letterlijk uit het boek *Programmeren in Java met BlueJ, een 'objecten eerst'-benadering van David J. Barnes en Michael Kölling (Pearson 2017),* maar dan aangepast aan de programmeertaal Python, en vertaald naar het Nederlands.

Oefening 5.2. (Vertrek van *2_chatbot.py*.) Maak een chatbot die antwoorden geeft zoals deze.

```
Welkom bij SuperChat, een antwoord op al je vragen.
(als ze tenminste iets met onze software te maken hebben)

Stel je vraag/vragen. Eindig met stop.
?> Als ik op de herstartknop druk, gebeurt er niets.
Kun je daar wat dieper op ingaan?
?> Soms herstart hij wel, maar dan traag en haperend.
Ik denk dat dit te maken heeft met je hardware. Het
upgraden van je processor zou alle prestatieproblemen moeten
oplossen. Heb je een probleem met je software?
?> Dit is toch geen probleem met de software?
Geen enkele andere klant heeft hier ooit over geklaagd. Wat is uw
systeemconfiguratie?
?> Ik gebruik Windows 11.
Dit is een bekende bug die te maken heeft met het
Windows besturingssysteem. Meld dit alstublieft bij Microsoft.
Wij kunnen hier niets aan doen.
?> Stop.

Tot ziens. Het deed ons plezier u goed te kunnen helpen
```

Als je dit live demonstreert, hou er dan rekening mee dat de antwoorden niet noodzakelijk dezelfde zullen zijn als in dit voorbeeld, omdat er in het programma willekeurige getallen worden gebruikt.

Zoals heel wat chatbots op het Internet, gebruikt ook deze een aantal trucjes om er slimmer uit te zien dan hij werkelijk is:

- Als hij een bepaald trefwoord in de vraag herkent, zoals 'windows' of 'traag' dan geeft hij daarop een standaardantwoord dat hij opzoekt in zijn database.
- Bevat de vraag geen enkel bekend trefwoord, dan kiest hij een willekeurig antwoord uit een lijst.

We geven je de 'database' cadeau, in de vorm van een dictionary (`antwoorden_dict`). Daarnaast krijg je ook nog een lijst (`antwoorden_lijst`) met standaardantwoorden waaruit je er willekeurig één kan kiezen. Een deel van het programma is ook al ingevuld. Nog enkele tips:

> ***Tips***
>
> - De module `random` heeft een functie waarmee je een willekeurig element uit een lijst kan kiezen. Zoek op hoe die heet en hoe je die gebruikt.
> - Gebruik `string.split()` om een string in afzonderlijke woorden te splitsen. Het resultaat is een lijst.

- Wanneer je in de vraag van de gebruiker een trefwoord (sleutel) vindt, geef dan de bijhorende waarde uit `antwoorden_dict` als antwoord terug.
- Vind je geen trefwoord, kies dan een willekeurig antwoord uit `antwoorden_lijst`.
- Bevat de vraag meer dan één trefwoord, dan reageer je enkel op het eerste.
- Zet de vraag om naar kleine letters, zodat hij ook 'Windows' als een trefwoord herkent.
- Schrijf een afzonderlijke methode `zoek_antwoord(vraag)` die het antwoord op een vraag teruggeeft en roep die op in de hoofdlus van het programma.
- Stop wanneer de invoer *begint* met 'stop'. Anders zou de chatbot in het voorbeeld hierboven niet gestopt zijn.

We suggereren hier om een afzonderlijke methode te schrijven, niet alleen omdat dit het programma leesbaar houdt, maar ook omdat het bij de meest eenvoudige oplossing nuttig is om uit een lus te springen met `return`.

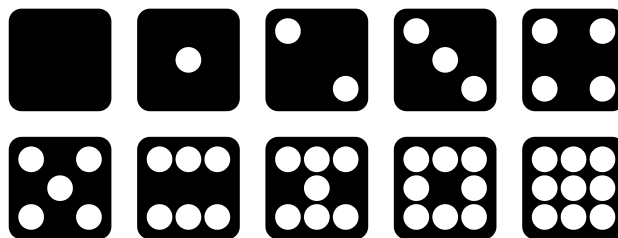
Postscriptum (voor de leerkracht)

We gebruiken met opzet enkel dictionaries met *strings* als sleutels. Leerlingen hebben de neiging om overal dictionaries in te zien, ook waar lijsten beter passen. Wanneer de sleutels kleine opeenvolgende getallen zijn, gebruik je die 'sleutels' liefst als index in een lijst. (Strikt genomen is een lijst immers ook een associatieve array.) Zelfs bij indices in een groter bereik – zoals bij postnummers bijvoorbeeld – is een lijst vaak sneller dan een dictionary.

6 Dobbelstenen

Overzicht

Pre-algoritmen: meervoudige selectie vermijden, vernestelde for-lussen met extra teller
Algoritmen en datastructuren: ontwerp
Softwarebibliotheken: `PIL` voor het tekenen van afbeeldingen, documentatie opzoeken



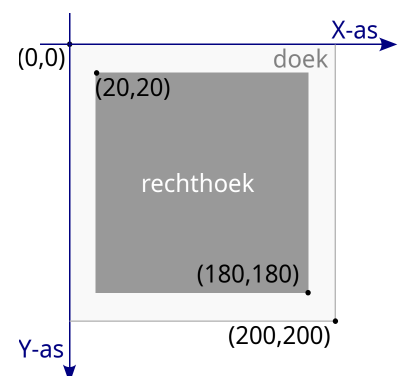
6.1 Tekenen met PIL/Pillow

De PIL-bibliotheek (*Python Imaging Library*) is een populaire Python-bibliotheek in Python waarmee je afbeeldingen kan bewerken. Wij gebruiken ze hier vooral om tekeningen te maken die bestaan uit combinaties van geometrische figuren (rechthoeken, cirkels, lijnen, …)[15] in allerhande kleuren. Het resultaat tonen we dan op het scherm of slaan we op in een afbeeldingsbestand.
De oorspronkelijke PIL-bibliotheek is niet langer up to date en werd vervangen door het Pillow-pakket. Documentatie vind je dan ook eerder onder de internet-zoekterm 'python pillow documentation'.
Als eerste voorbeeld (zie *1_rechthoek.py*) tekenen we een zwart vierkant van 160×160 pixels gecentreerd op een witte achtergrond met afmeting 200×200. Eerst moet je de bibliotheek importeren en alles klaarzetten

```
from PIL import Image, ImageDraw

prent = Image.new("RGB", [200, 200], color="white")
doek  = ImageDraw.Draw(prent)
```

De variabele `prent` stelt de ganse afbeelding voor. Je geeft de afmetingen (in pixels) en de achtergrondkleur op wanneer je de prent aanmaakt. De variabele `doek` gebruik je om te tekenen - je tekent dus op het doek en niet rechtstreeks op de prent.
Een rechthoek op het *doek* tekenen kan met één enkele opdracht

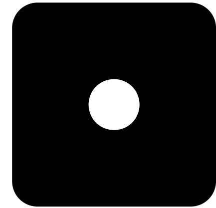
```
doek.rectangle([20, 20, 180, 180], fill="black")
```



De vier getallen die je opgeeft, zijn achtereenvolgens de X- en de Y-coördinaten van de linkerbovenhoek en de rechterbenedenhoek van de rechthoek. Bij computertekeningen ligt het nulpunt van het assenstelsel helemaal links bovenaan de prent (en niet in het midden zoals je misschien verwacht) en wijst de Y-as van boven naar onder (dus het omgekeerde van wat je in de wiskunde gewoon bent) – zie de figuur hiernaast.
Tot slot moet je de *prent* nog op het scherm tonen

```
prent.show()
```

[15] Grafieken maken we echter met een andere bibliotheek, nl. `matplotlib`. Zie hoofdstuk ?.

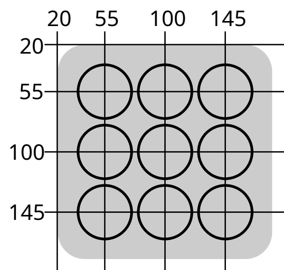


Oefening 6.1 Pas de tekening aan zodat ze een dobbelsteen toont met één oog. Het oog is een witte cirkel met straal 20 pixels precies in het midden van de rechthoek. De hoeken van de rechthoek zijn afgerond met cirkelbogen met opnieuw een straal van 20 pixels. Sla de prent op in een bestand met de naam *dobbelsteen-1.png*. Zoek een aantal zaken zelf op (op het Internet):

- Hoe teken je een *afgeronde* rechthoek?
- Hoe teken je een cirkel? (Dat is iets moeilijker terug te vinden.)
- Hoe bewaar ik mijn prent in een bestand in plaats van hem te tonen op het scherm?

Er zijn een aantal manieren om een cirkel te teken met PIL. Je kan een *ellips* tekenen met `doek.ellipse`, of een *afgeronde rechthoek* van 40×40 met afrondingen van straal 20. In beide gevallen geef je geen middelpunt of straal op, maar de coördinaten van de omhullende rechthoek. (Er is ook een methode `doek.arc` die je kan gebruiken, maar daarmee maak je het nodeloos ingewikkeld.) Merk op dat ChatGPT en Bing Copilot direct het correcte antwoord geven op de vraag 'hoe teken ik een cirkel met PIL'. Zij kiezen voor een ellips.

6.2 Alle dobbelstenen



Oefening 6.2 Schrijf een procedure `teken_dobbelsteen(aantal_ogen)` die een afbeeldingsbestand maakt van een dobbelsteen met een gegeven aantal ogen – van 0 tot en met 9 ogen zoals in de prent bovenaan dit hoofdstuk. De figuur hiernaast geeft je de nodige coördinaten. Gebruik bestandsnamen zoals *dobbelsteen-0.png*, …, *dobbelsteen-9.png,* waar het aantal ogen in de naam staat. **Belangrijk!** Denk voor je dit programmeert eerst goed na hoe je dit kunt doen zonder code tien keer te knippen en te plakken en zonder een if-opdracht met 10 verschillende gevallen.

Dit kan klassikaal besproken worden. De meervoudige selectie kan je, zoals in §4.1, vermijden door gegevens in een lijst op te slaan waarvoor het aantal ogen als index fungeert. De vraag is nu hoe we die lijst precies invullen.

Op index *n* moet op één of andere manier een beschrijving staan van hoe een dobbelsteen met *n* ogen eruit ziet, een (korte) beschrijving die we in de rest van het programma gemakkelijk kunnen gebruiken.

- Een mogelijkheid is om van elk van de negen cirkels achtereenvolgens aan te geven of ze 'aan-' of 'uitgeschakeld' zijn. Dit kan met een lijst van Booleaanse waarden, maar dan krijgen we uiteindelijk een lijst van lijsten waar de leerlingen nog niet mee vertrouwd zijn. Als alternatief kan je voor elk aantal ogen een string van 9 lettertekens bijhouden, waarbij elke letter 'aan' of 'uit' betekent. Bijvoorbeeld `"100010001"` voor drie ogen, of `"+---+---+"` of `"0...0...0"`.
- Een andere optie is om in de string de posities bij te houden waarvoor er cirkels moeten getekend worden, dus `"048"` voor drie ogen, `"012678"` voor zes ogen en `""` voor geen enkel oog.

Je kan ook coördinaten opslaan in de lijst, maar de andere suggesties zijn compacter en vergen minder werk.

De negen plaatsen waar er ogen kunnen staan, kan je aangeven met een rij- en kolomnummer, maar ook met een (volg)nummer. X- en Y-coördinaten van de cirkels bereken je uit de rij- en kolomnummers. Het nummer heb je nodig als index in een lijst. De tabel toont je het verband tussen de drie getallen. We tellen (vanzelfsprekend?) vanaf nul.

	Kolom 0	Kolom 1	Kolom 2
Rij 0	Nr 0	Nr 1	Nr 2
Rij 1	Nr 3	Nr 4	Nr 5
Rij 2	Nr 6	Nr 7	Nr 8

In het programma moet je enerzijds door alle rijen en kolommen lopen, maar tegelijkertijd ook door alle volgnummers. Er zijn drie manieren waarop je dit kan doen (zie hieronder). Welke je kiest, hangt vooral af van je persoonlijke smaak.

X-coördinaten hangen af van kolomnummers, Y-coördinaten van rijnummers. Dat wordt gemakkelijk door elkaar gehaald omdat de natuurlijke volgorde rij-kolom is, en niet kolom-rij.

<table>
<tr>
<td>Bereken het volgnummer uit het rij- en kolomnummer
<pre>for rij in range(3):
    for kolom in range(3):
        nr = ???
        ...</pre></td>
<td>Bereken de rij- en kolomnummers uit het volgnummer
<pre>for nr in range(9):
    rij   = ???
    kolom = ???
    ...</pre></td>
<td>Verhoog het nummer zelf terwijl je door rijen en kolommen loopt
<pre>nr = 0
for rij in range(3):
    for kolom in range(3):
        ...
        nr += 1</pre></td>
</tr>
</table>

Vul zelf nog de vraagtekens in.

Er geldt nr = 3 × rij + kolom. De kolom is de *rest* van de deling nr ÷ 3, de rij is het *quotiënt.* Gebruik een gehele deling.

7 De wet van Benford

Overzicht

Pre-algoritmen: meervoudige selectie vermijden
Algoritmen en datastructuren: frequentietabel
Externe gegevensbronnen: invoer van en uitvoer naar een CSV-bestand
Softwarebibliotheken: `matplotlib` voor het tekenen van histogrammen

Achtergrond

In 1938 publiceerde Frank Benford een artikel in een wetenschappelijk tijdschrift waarin hij het verschijnsel beschrijft dat in veel gegevensverzamelingen uit de praktijk (maar niet allemaal) de meeste van die getallen met een 1 beginnen. Minder getallen beginnen met een 2 en de minste met een 9. Dit wijst erop dat de kans om begincijfer te zijn niet voor alle cijfers van 1 tot en met 9 hetzelfde is. Benford toonde aan dat de kans dat in een reeks getallen een getal met een 1 begint, ongeveer 30% is. De kans dat een getal met een 9 begint, is daarentegen slechts 5%.

Bron: Wikipedia – https://nl.wikipedia.org/wiki/Wet_van_Benford

In dit hoofdstuk zoeken we uit of dit klopt voor twee publieke datasets: de *inwonersaantallen* van de Belgische gemeenten en de *huisnummers* van de Vlaamse middelbare scholen.

7.1 Rekenbladgegevens verwerken

	A	B	C	D	E
1	Globaal bevolkingscijfers per gemeente				
2	Situatie op 1/11/2023				
3	Niscode	Gemeente	mannen	vrouwen	totaal
4	11001	AARTSELAAR	7.394	7.485	14.879
5	11002	ANTWERPEN	274.283	271.578	545.861
6	11004	BOECHOUT	6.752	7.095	13.847
7	11005	BOOM	9.825	9.783	19.608
8	11007	BORSBEEK	5.550	5.815	11.365
9	11008	BRASSCHAAT	18.426	20.116	38.542
10	11009	BRECHT	15.231	15.320	30.551
11	11013	EDEGEM	11.117	11.890	23.007
12	11016	ESSEN	9.779	9.846	19.625
13	11018	HEMIKSEM	6.153	6.227	12.380
14	11021	HOVE	4.028	4.379	8.407
15	11022	KALMTHOUT	9.600	10.065	19.665
16	11023	KAPELLEN	13.969	14.286	28.255
17	11024	KONTICH	10.621	11.094	21.715
18	11025	LINT	4.355	4.378	8.733
19	11029	MORTSEL	12.925	13.707	26.632
20	11030	NIEL	5.422	5.508	10.930

De gegevens over de aantallen inwoners van de Belgische gemeenten zijn publiek beschikbaar en kan je downloaden van de website van de Federale Overheidsdienst Binnenlandse Zaken, onder de noemer Statistieken van bevolking. Je kan de cijfers downloaden als een Excel-rekenblad (*stat-1-1_n.xlsx*).
In Python rechtstreeks met een rekenblad werken is niet zo eenvoudig[16]. Daarom zetten we het rekenbladbestand in een eerste stap om naar een tekstbestand dat we lijn per lijn kunnen verwerken zoals in hoofdstuk 2. Open het bestand met je rekenbladprogramma, doorloop het eens snel boven naar onder om de algemene structuur ervan in te prenten, en bewaar het dan opnieuw *als een CSV-bestand* (met naam *stat-1-1_n.csv*). Het resultaat is een tekstbestand daar er ongeveer zo uitziet

```
Globaal bevolkingscijfers per gemeente;;;;
Situatie op 1/11/2023;;;;
Niscode;Gemeente;mannen;vrouwen;totaal
11001;AARTSELAAR;7.394;7.485;14.879
11002;ANTWERPEN;274.283;271.578;545.861
11004;BOECHOUT;6.752;7.095;13.847
…
(585 lijnen)
```

Het kan er bij jou ook lichtjes anders uitzien, afhankelijk van de instellingen die je hebt gemaakt bij het omzetten. Misschien staan er bij jou komma's in plaats van puntkomma's, of staan de aantallen tussen aanhalingstekens. Dit vormt niet echt een probleem, maar je zal er wel rekening mee moeten houden voor wat volgt. (Dat er geen accenten in de gemeentenamen voorkomen is trouwens ook mooi meegenomen, dan hoef je je niet te bekommeren om de internationale codering.)

---

[16]Er bestaan externe Python-modules, zoals `pandas` en `openpyxl` om rechtstreeks met rekenbladen te werken. Voor onze voorbeelden vonden we het niet nuttig om nog maar eens een bijkomende softwarebibliotheek te introduceren

Oefening 7.1. Verwerk dit bestand lijn per lijn en druk na afloop af in hoeveel procent van de gevallen de laatste kolom begint met een 1, hoeveel keer met een 2, …, in de volgende vorm:

```
1;36.706689536878216
2;17.32418524871355
…
9;4.802744425385934
```

(Waarom denk je dat we precies die vorm gekozen hebben?)
Om dit later als CSV-bestand op te slaan en als rekenblad te gebruiken – om er een grafiek van te maken, bijvoorbeeld.
Vooraleer je aan het programmeren slaat, moet je goed bedenken hoe je dit zult aanpakken. Wat doe je bijvoorbeeld met het eerste cijfer van het inwonersaantal nadat je dit hebt bepaald? Misschien iets zoals het volgende?

```
if eerste_cijfer == 1:
    aantal1 += 1
elif eerste_cijfer == 2:
    aantal2 += 1
…
else:
    aantal9 += 1
```

Ondertussen heb je al door (zie ook §4.1 en §6.2) dat je zo'n lange if/elif/else-opdracht – wat we een *meervoudige selectie* noemen – vaak kan vervangen door iets wat eleganter oogt, en meestal ook efficiënter is. Ook hier is dit het geval. Dat we 9 verschillende variabelen hebben met zeer gelijkende namen en functionaliteit, is ook een indicatie dat het anders kan. Wat als we `aantal[1]`…`aantal[9]` zouden schrijven in plaats van `aantal1`…`aantal9`?

```
if eerste_cijfer == 1:
    aantal[1] += 1
elif eerste_cijfer == 2:
    aantal[2] += 1
…
else:
    aantal[9] += 1
```

De variabele `aantal` wordt nu een lijst waarvan we de elementen gebruiken met index 1 t.e.m. 9 (aantal[0] gebruiken we niet, maar een lijst begint nu eenmaal op index 0, dat kunnen we moeilijk vermijden).
Je kan ook de aantallen voor eerste cijfer 1 opslaan op positie 0 in de lijst, die voor eerste cijfer 2 op positie 1, enz., maar dat maakt het programma minder leesbaar. Over dat ene extra lijstelement maken we ons geen zorgen.
Waar je anders gewoon bent om lijsten element per element aan te vullen, gebruik je de lijst `aantal` op een lichtjes andere manier: in het begin van het programma zorg je ervoor dat de lijst voldoende groot is en meteen opgevuld met nullen.
Dus: `aantal = [0] * 10`.
Een lijst die we op deze manier gebruiken om aantallen bij te houden, heet een *frequentietabel* – zie inzet op de volgende bladzijde.

CSV-formaat

CSV staat voor het Engelse *Comma-Separated Values* (waarden gescheiden door komma's) en is een veelgebruikt bestandsformaat voor het opslaan van eenvoudige tabelgegevens. Het is een tekstformaat waarin elke lijn overeenkomt met één rij van de tabel, en waar binnen een lijn de opeenvolgende waarden uit de verschillende kolommen worden opgesomd gescheiden door komma's – of tegenwoordig ook vaak door puntkomma's. Soms worden waarden tussen dubbele aanhalingstekens geplaatst om verwarring te vermijden (bijvoorbeeld als de waarde zelf een komma bevat).

Omdat we de aantallen in een frequentietabel opslaan, kan de meervoudige selectie herleid worden tot één enkele lijn!

```
aantal[eerste_cijfer] += 1
```

7.2 De inwonersaantallen

Vooraleer je het programma schrijft, moet je nog stilstaan bij enkele aandachtspunten:

1. Hoe haal je het eerste cijfer van het inwonersaantal (een geheel getal) uit de lijn (een string) die je uit het bestand haalt? Het inwonersaantal dat we zoeken staat in de laatste (paarse) kolom van het rekenblad.
2. Een aantal rijen in het rekenblad (= lijnen in het CSV-bestand) mag je *niet* gebruiken. Welke?
3. Op het einde moet je de opgeslagen aantallen (= elementen uit de frequentietabel) nog omzetten naar percentages. Hoe doe je dit?

We helpen je verder met punt 2.

Voor punt 1 bepaal je de plaats van de laatste puntkomma (of komma) in de lijn met `rfind`. Het cijfer dat je zoekt staat dan één positie verder in de lijn. (Het is niet nodig om eerst het volledige inwonersaantal te extraheren.) Gebruik `int(…)` om dit cijfer om te zetten naar een index in de frequentietabel. Het alternatief om de lijn met behulp van `split` in 'kolommen' te splitsen doet onnodig werk en kan tot fouten leiden als het CSV-bestand komma's als scheidingstekens bevat – er staan immers in de 2e kolom komma's bij bepaalde gemeenten. Voor punt 3 bereken je met `sum(…)` de som van alle frequenties. Op index 0 staat een nul die dit resultaat niet beïnvloedt.

In §3.1 (inzet) tonen we hoe je met een for-lus alle lijnen van een bestand één voor één kan inlezen en verwerken. Python voorziet echter ook een functie `readline()` waarmee je één enkele lijn van het bestand inleest.

```
with open ("mijnbestand.txt", "r") as bestand:
    ...
    lijn = bestand.readline()
    ...
```

Telkens wanneer je `readline` oproept, krijg je de volgende lijn van het bestand. Dit kan nuttig zijn wanneer je een vast aantal lijnen uit een bestand wil verwerken zonder meteen het ganse bestand in te lezen. Bovendien laat Python je toe om, nadat je op deze manier een aantal lijnen hebt ingelezen en verwerkt, toch nog de vertrouwde for-lus op te roepen. Deze lus leest dan verder vanaf de plaats in het bestand waar je op dat moment gekomen bent en begint niet opnieuw van het begin! Dit kan je handig gebruiken om de drie hoofdinglijnen uit het werkblad over te slaan.

Er staat echter nog een lijn in het bestand die we best overslaan – namelijk de laatste lijn die de totalen bevat. Hoe krijg je dit voor elkaar – en nee, er is geen ingebouwde Python-functie die je daarbij kan helpen. Hoe los je dit op?

Je kan de lijn niet overslaan, maar je kan in de plaats de laatste waarde van `eerste_cijfer` terug uit de frequentietabel halen nadat het bestand volledig is verwerkt. Het alternatief om eerst alle lijnen uit het bestand in een afzonderlijke lijst opslaan en dan pas die lijnen te verwerken die je nodig hebt, is *overkill*.

Frequentietabel

In een frequentietabel slaat men op *hoeveel keer* een bepaalde waarde voorkomt in een set gegevens, op een manier die het gemakkelijk maakt, en snel, om die aantallen te tellen. Als de waarden kleine, min of meer opeenvolgende, gehele getallen zijn, gebruik je een *lijst* als frequentietabel[17]: de waarde wordt dan een index in de lijst, en de lijstelementen bevatten dan het aantal keer dat de betreffende waarde voorkomt – de *frequentie* van die waarde. In andere gevallen kan je een dictionary gebruiken met de waarden als sleutels en de frequenties als waarden.

[17] In een programmeertaal met arrays, gebruik je een array eerder dan een lijst – zie opmerkingen bij §1.2.

7.3 Staafdiagrammen

Oefening 7.2 Maak een staafdiagram van de percentages uit oefening 7.1.
Er zijn twee manieren waarop je dit kunt doen, een luie en een interessante. De *luie* manier werkt zo:

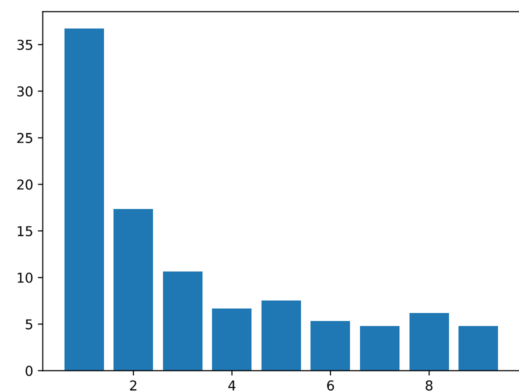
- Druk de uitvoer van het programma niet op het scherm af, maar schrijven het uit naar een tekstbestand – een CSV-bestand
- Open achteraf dit bestand met je rekenbladsoftware en gebruik deze software om er een staafdiagram van te maken[18].

Uitschrijven van gegevens naar een tekstbestand is helemaal niet moeilijk. We tonen je hoe je dit doet:

```
with open ("frequenties.csv", "w") as bestand:
    ...
    print (f"{cijfer};{percentage}",file=bestand)
    ...
```

De with-opdracht is bijna dezelfde als wanneer je van een tekstbestand leest, alleen staat er nu `"w"` als tweede parameter van `open` in plaats van `"r"`. Om iets naar het bestand te schrijven, gebruik je dezelfde print die je al gewoon bent, maar je voegt een bijkomend argument toe van de vorm '`file=bestand`' waar `bestand` de bestandsvariabele is die je in de with-opdracht hebt geïntroduceerd.

De *interessante* manier is om Python zelf het staafdiagram te laten tekenen. Dit doen we met behulp van de softwarebibliotheek `matplotlib`.



Deze bibliotheek is beter geschikt voor het maken van grafieken en staafdiagrammen dan `PIL` uit hoofdstuk 6. `PIL` is eerder bedoeld voor eenvoudige geometrische afbeeldingen – ook al kan ook dat met `matplotlib`. Doordat `matplotlib` zoveel mogelijkheden biedt, is ze helaas niet eenvoudig te gebruiken en ook de documentatie is voor een beginner vrij ontoegankelijk. Daarom beperken we ons in deze tekst tot minimale basistoepassingen.

Om een staafdiagram of een grafiek te tekenen, heb je een lijst van X-waarden een lijst van Y-waarden nodig. Bij een grafiek zijn dit de coördinaten van de punten van de grafiek, bij een staafdiagram de coördinaten van de bovenkanten van de staven. In ons voorbeeld zijn de X-waarden de opeenvolgende getallen [1,2,…,9] en de Y-waarden de overeenkomstige frequentiepercentages [36.7066…, 17.3241…, …, 4.8027…]. Voor de lijst van X-waarden bestaat er een eenvoudig trucje:

```
x_waarden = list(range(1,10))
```

de lijst met Y-waarden moet je zelf opbouwen aan de hand van de frequentietabel.
Het staafdiagram maak je tenslotte op de volgende manier:

```
from matplotlib import pyplot
pyplot.bar(x_waarden, y_waarden)
pyplot.show()
```

(Probeer ook eens `plot` in plaats van `bar`.)

[18]Als de leerlingen nog niet weten hoe ze grafieken moeten maken in een rekenblad, dan wordt de luie manier plots wat meer uitdagend…

7.4 De huisnummers

Oefening 7.3 Controleer de wet van Benford – en maak het bijbehorende staafdiagram – voor de huisnummers van de Vlaamse scholen. Je vindt de nodige gegevens in het bestand *Scholen-secundair-onderwijs.csv* dat we hebben gedownload bij Onderwijs Vlaanderen. Het bestand is reeds in CSV-formaat. Kijk goed na waar de huisnummers terug te vinden zijn en denk na hoe je het eerste cijfer ervan zult ophalen in je programma.

Het huisnummer staat in de 8^e^ kolom en staat tussen aanhalingstekens. Er is een school waarvan het huisnummer niet met een cijfer begint, dus daarmee moet je rekening houden. De eerste lijn van het bestand moet je overslaan. In tegenstelling tot bij vorige oefening is `split` hier wel de aangewezen manier om het huisnummer te isoleren, omwille van de duidelijkheid, alhoewel het zeker geen slechte oefening is om een functie te schrijven die de positie teruggeeft van de *n*^e^ puntkomma in een string.

Postscriptum (voor de leerkracht)

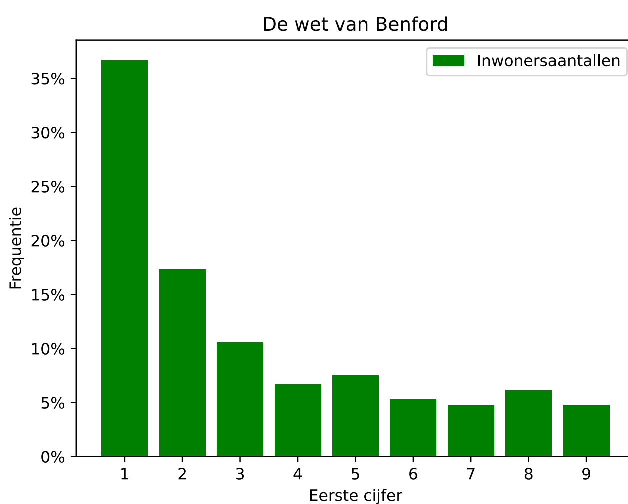
De `matplotlib`-bibliotheek biedt heel wat mogelijkheden om grafieken verder te 'verfraaien'. Vanuit het oogpunt van *Algoritmen en programmeren* vonden we dit didactisch minder interessant, maar voor wie toch met zijn leerlingen daarop wat dieper wil ingaan, drukken we de code af waarmee je de diagrammen verkrijgt zoals onderaan de pagina.

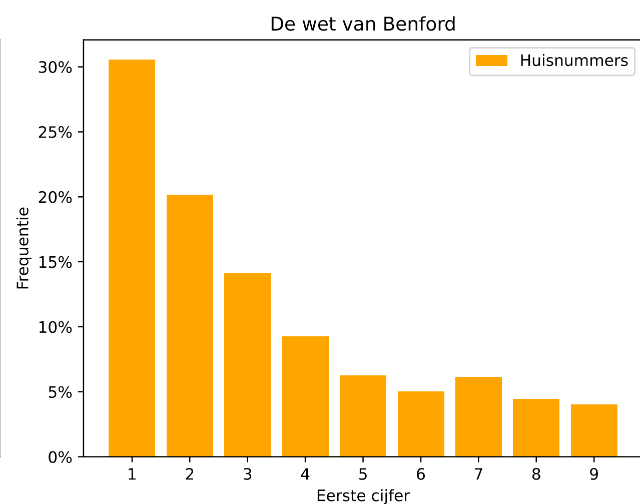
Onderschriften en legende:

```
pyplot.bar(x_waarden,y_waarden,label="Huisnummers",color="orange") # label + kleur
pyplot.title("De wet van Benford")
pyplot.xlabel("Eerste cijfer")
pyplot.ylabel("Frequentie")
pyplot.legend()
```

Om formattering en aantal markeringen van de assen aan te passen, heb je een bijkomende import nodig:

```
from matplotlib import ticker

assen = pyplot.gca()
assen.yaxis.set_major_formatter(ticker.PercentFormatter(decimals=0))
assen.xaxis.set_major_locator(ticker.MultipleLocator(1))
```





8 Sudoku

Overzicht

Pre-algoritmen: vernestelde for-lussen
Algoritmen en datastructuren: frequentietabel, 2-dimensionale tabel

Achtergrond

Een *Latijns vierkant* van orde *n* is een vierkant van *n* rijen en *n* kolommen, gevuld met getallen 1…*n* zodat elk getal precies één keer voorkomt in elke rij en in elke kolom.
Een *Sudoku* is een Latijns vierkant van orde 9 met de bijkomende eis dat ook elk cijfer precies 1 keer voorkomt in de negen 3 × 3 blokken waarin het rooster wordt onderverdeeld (zie figuur).
In een Sudoku-*puzzel* moet je een gedeeltelijk ingevulde Sudoku aanvullen zodanig dat aan al deze regels is voldaan.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

8.1 Sudoku's in Python

De programmeertaal Python bezit geen kant en klare datastructuur waarmee je een tabel met rijen en kolommen, zoals een Sudoku, kunt voorstellen. Voor een dergelijke *2-dimensionale tabel* gebruik je in Python daarom meestal een lijst van lijsten. De Sudoku hierboven sla je op in een variabele op de volgende manier:

```
sudoku = [
    [5, 3, 4, 6, 7, 8, 9, 1, 2],
    [6, 7, 2, 1, 9, 5, 3, 4, 8],
    [1, 9, 8, 3, 4, 2, 5, 6, 7],
    [8, 5, 9, 7, 6, 1, 4, 2, 3],
    [4, 2, 6, 8, 5, 3, 7, 9, 1],
    [7, 1, 3, 9, 2, 4, 8, 5, 6],
    [9, 6, 1, 5, 3, 7, 2, 8, 4],
    [2, 8, 7, 4, 1, 9, 6, 3, 5],
    [3, 4, 5, 2, 8, 6, 1, 7, 9]
]
```

Met zo'n 2-dimensionale tabel werken is in wezen niet zo moeilijk zolang je onthoudt dat het eigenlijk een lijst van lijsten is – en met lijsten werken ben je al gewoon:

- Rijen en kolommen tel je vanaf 0. Het zijn immers indices in lijsten.
- De uitdrukking `sudoku[8]` stelt de laatste rij voor van de tabel `sudoku`. Het zevende element van die rij, dus op index 6, noteren we als `sudoku[8][6]`. De waarde van `sudoku[8][6]`is 1, want er staat een 1 op rij 8 en kolom 6.
- Anders gezegd, het element op rij `r` en kolom k noteer je als `sudoku[r][k]`.

Een 2-dimensionale tabel is een lijst van lijsten. Je kan dus foreach-lussen gebruiken om alle elementen in die tabel te overlopen

```
for rij in sudoku:
    for element in rij:
        # doe iets met element
        ...
```

Heb je ook de rij- en kolomnummers nodig, dan kan je ook gewone for-lussen gebruiken

```
for rijnummer in range(9):
    for kolomnummer in range(9):
        element = sudoku_1[rijnummer][kolomnummer]
        # doe iets met element
        ...
```

Combinaties van beide zijn ook mogelijk.

Oefening 8.1 Schrijf een procedure `print_sudoku(sudoku)` die een gegeven Sudoku afdrukt zoals hieronder. Let erop dat je tussen de cijfers en de randen telkens precies één spatie afdrukt.

```
---------------------
| 5 3 4 6 7 8 9 1 2 |
| 6 7 2 1 9 5 3 4 8 |
| 1 9 8 3 4 2 5 6 7 |
| 8 5 9 7 6 1 4 2 3 |
| 4 2 6 8 5 3 7 9 1 |
| 7 1 3 9 2 4 8 5 6 |
| 9 6 1 5 3 7 2 8 4 |
| 2 8 7 4 1 9 6 3 5 |
| 3 4 5 2 8 6 1 7 9 |
---------------------
```

Tip

Gebruik `print("…", end="")` om iets af te drukken zonder dat Python daarna een nieuwe lijn neemt.

In *1_print_sudoku.py* vind je voorbeelden waarmee je je oplossing kan uittesten.

Er zijn twee mogelijke benaderingen: ofwel gebruik je twee (vernestelde) foreach-lussen, ofwel gebruik je twee gewone for-lussen, voor rij- en kolomnummers. De eerste oplossing is eleganter – want je hebt de rij- en kolomnummers niet nodig – de tweede is en betere voorbereiding voor oefening Fout: Bron van verwijzing niet gevonden.

8.2 Sudoku's controleren

Oefening 8.2 2_*controleer_sudoku.py* bevat (de aanzet tot) een programma waarmee je kan controleren of een 9 × 9 tabel van cijfers een Sudoku is. Vul de verschillende functies aan zoals aangegeven en test het programma uit op de voorbeelden die erin staan. Doe dit in afzonderlijke stappen, en test elke stap voor je aan de volgende begint.

1. Schrijf een functie `is_correct_lijst(lijst)` die kijkt of een *lijst* van getallen elk cijfer van 1 t.e.m. 9 hoogstens één keer bevat (en al naargelang `True` of `False` teruggeeft). Gebruik hierbij een frequentietabel zoals in hoofdstuk 7.
   Je kan ook controleren of de lijst elke cijfer *precies* één keer bevat. Voor deze oefening is dit voldoende, maar dit kan handig zijn als we ook lege vakjes in het rooster toelaten – zie uitbreidingsoefening 8.3*.
2. Gebruik `is_correct_lijst` in de functie `is_correct_rijen` om te kijken of alle rijen van de Sudoku voldoen aan de regels. Tip: een 2-dimensionale tabel is een lijst van lijsten.
   M.a.w., je kan met een foreach-lus alle rijen van de Sudoku overlopen, en ze telkens als argument meegeven aan `is_correct_lijst`. Je kan uit de lus springen zodra je een rij vindt die niet voldoet, of in de plaats een while-lus gebruiken met dubbele conditie (hoofdstuk ??).
3. Implementeer de functie `is_correct_kolommen` die kijkt of de kolommen van de Sudoku voldoen aan de regels. Helaas kan je hier niets hergebruiken uit 1. en 2., behalve de achterliggende ideeën. Gebruik opnieuw een frequentietabel.
   Hier is een while-lus met dubbele conditie minder aangewezen – je moet namelijk een *dubbele* lus vroegtijdig afbreken.
4. Controleren of alle 3×3-blokjes aan de Sudoku-regels voldoen, doe je opnieuw met een frequentietabel. Schrijf eerst een afzonderlijk functie die één blokje controleert, zodat je die later 9 keer kan oproepen. Denk eerst goed na welke parameters die functie moet hebben!
   Behalve de Sudoku zelf, nog twee parameters die aangeven welk blokje je bedoelt. Dit kunnen twee getallen zijn uit het interval [0,2] of anderzijds kan je ook de rij- en kolomnummers meegeven van de linkerbovenhoek van het blokje, dus telkens 0, 3 of 6. We hebben hier met opzet een hulpfunctie geïntroduceerd om de leesbaarheid te verhogen. Zonder deze functie heb je een 4-voudig vernestelde for-lus nodig .

8.3 Sudoku's uit Sudoku's

Er bestaan heel wat transformaties[19] waarmee je van één Sudoku een andere kunt maken die nog steeds aan de Sudoku-regels voldoet. Zo kan je de Sudoku bijvoorbeeld horizontaal of verticaal of diagonaal spiegelen, of 90º draaien. Je kan ook de inhoud van de Sudoku hernummeren, door overal 1 door 9 te vervangen, 2 door 8, 3 door 7, 4 door 6, …, 8 door 2 en 9 door 1.

Origineel

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Verticaal gespiegeld

3	4	5	2	8	6	1	7	9
2	8	7	4	1	9	6	3	5
9	6	1	5	3	7	2	8	4
7	1	3	9	2	4	8	5	6
4	2	6	8	5	3	7	9	1
8	5	9	7	6	1	4	2	3
1	9	8	3	4	2	5	6	7
6	7	2	1	9	5	3	4	8
5	3	4	6	7	8	9	1	2

Horizontaal gespiegeld

2	1	9	8	7	6	4	3	5
8	4	3	5	9	1	2	7	6
7	6	5	2	4	3	8	9	1
3	2	4	1	6	7	9	5	8
1	9	7	3	5	8	6	2	4
6	5	8	4	2	9	3	1	7
4	8	2	7	3	5	1	6	9
5	3	6	9	1	4	7	8	2
9	7	1	6	8	2	5	4	3

Oefening 8.3 Schrijf 2 functies `spiegel_horizontaal(…)`en `spiegel_verticaal(…)` die een Sudoku als parameter nemen en een *nieuwe* Sudoku retourneren die gespiegeld is zoals hierboven. Druk ter controle het origineel en de 2 getransformeerde Sudoku's af met de procedure die je hebt geschreven voor oefening 8.1.

- Let erop dat je de inhoud van de oorspronkelijke Sudoku niet verandert.
- Hier en daar kan je nuttig gebruik maken van het feit dat een 2-dimensionale tabel een lijst is.
  In het bijzonder is verticaal spiegelen niets anders dan die lijst omkeren. Dit kan met één korte return-opdracht. Daarnaast kan je ook af en toe een foreach-lus gebruiken.
- Er bestaat in Python een heel compacte manier om het 'omgekeerde' van een lijst te produceren (= dezelfde elementen maar in omgekeerde volgorde). Ken je die nog?
  Met behulp van *splicing*: `lijst[::-1]`.

[19] 1.218.998.108.160 om precies te zijn – zie Mathematics of Sudoku op Wikipedia

9 In-place

Overzicht

Pre-algoritmen: *in-place* bewerken van lijsten
Algoritmen en datastructuren: sorteren door invoegen, selectiesortering
Softwarebibliotheken: willekeurige getallen

Achtergrond

Bij de implementatie van heel wat algoritmen en datastructuren doe je frequent kleine aanpassingen aan lijsten waarbij de grootte van de lijst niet verandert[20], bijvoorbeeld de waarde van een element wijzigen, elementen met elkaar van plaats verwisselen, enz. Omdat je de lijst zelf bewerkt en ze niet telkens door een nieuwe lijst vervangt, noemt men dit *in-place*-bewerkingen en *in-place*-algoritmen.
Deskundig toepassen van deze technieken maakt algoritmen sneller en vergt minder computergeheugen, wat bijvoorbeeld belangrijk is bij kleinere 'computers' zoals micro:bit of Arduino.

9.1 In-place tussenvoegen

In onderstaande programmafragmenten willen we de elementen van een lijst één plaats naar achter opschuiven en het laatste element terug vooraan plaatsen – zoals '`lijst=[lijst[-1]]+lijst[:-1]`', maar dan in-place.

```
laatste = lijst[-1]
for index in range(1, len(lijst)):
    lijst[index] = lijst[index-1]
lijst[0] = laatste
```

```
laatste = lijst[-1]
for index in range(len(lijst)-1, 0, -1):
    lijst[index] = lijst[index-1]
lijst[0] = laatste
```

Slechts één van die fragmenten werkt correct. Zie je welke van de twee (zonder ze uit te proberen?) En wat doet dan het andere fragment?
Enkel het rechterfragment is correct. Het resultaat van het linkerfragment is een lijst waarvan alle elementen dezelfde zijn, behalve het eerste.

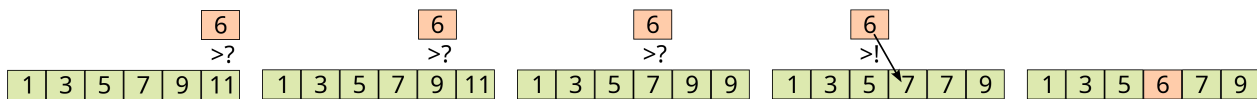
Oefening 9.1 Schrijf een programma dat alle elementen van een lijst één plaats naar voor opschuift en het eerste element achteraan plaatst – zoals '`lijst=lijst[1:]+[lijst[0]]`', maar dan in-place. Test dit uit op lijsten van verschillende lengte. Denk op voorhand goed na welke for-lus je zal gebruiken.
Dit keer moet je van voor naar achter lopen door de lijst.

Oefening 9.2 Schrijf een procedure `invoegen(getal,lijst)` die een geheel getal op de juiste plaats invoegt in lijst die reeds *geordend* is. Doe dit in-place, zonder slices te nemen of nieuwe lijsten aan te maken zoals in §3.1. We schetsen het algoritme hieronder.

- Doorloop de lijst van achter naar voor
- Is het huidige element in de lijst groter dan het getal dat je wil invoegen, verplaatst dan het lijstelement één plaats naar achter en vervolg je zoektocht één plaats verder naar voor.
- Is het getal groter dan (of gelijk aan) het huidige lijstelement, dan heb je de juiste plaats gevonden. Kopieer het getal naar die plaats in de lijst.

Belangrijk! We laten de lengte van de lijst onveranderd. Het laatste element van de lijst zal dus wellicht verloren gaan bij het invoegen.
Hieronder schetsen wat er gebeurt wanneer je het getal 6 invoegt in de lijst [1,3,5,7,9,11].

[20] In een andere programmeertaal dan Python werkt men met *arrays* in de plaats van lijsten – zie §1.2.



Om dit helemaal tot een goed einde te brengen, moet je rekening houden met twee belangrijke details:

- Wat als het getal dat je wil invoegen groter is dan het laatste element in de lijst?
  Dan *mag* je niets te doen. Anders probeer je misschien het getal te plaatsen op een positie buiten de lijst.
- Wat als het getal dat je wil invoegen kleiner is dan het eerste element in de lijst?
  Invoegen gebeurt op dezelfde manier als in de andere gevallen, je moet enkel zorgen dat de lus op tijd stopt zodat je niet per ongeluk index -1 gebruikt . Een while-lus met dubbele conditie is hier aangewezen.
- Ook voor hele kleine lijsten moet dit werken – voor lijsten van 1 element of voor lege lijsten.
  Lege lijsten vergen een bijkomende if-opdracht.

Uiteindelijk krijg je het volgende resultaat – zie ook *2_invoegen.py*.

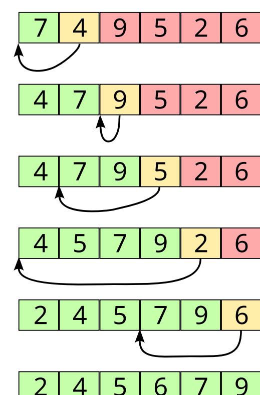
```
def invoegen(getal,lijst):
    if len(lijst) > 0 and getal < lijst[-1]:
        index = len(lijst)-2
        while index >= 0 and lijst[index] > getal:
            lijst[index+1] = lijst[index]
            index -= 1
        lijst[index+1] = getal
```

9.2 Sorteren door invoegen

Voor wie dit sorteeralgoritme te ingewikkeld vindt, voorzien we in §9.3 *selectiesortering* als alternatief.

In de informaticawetenschappen kent men ondertussen heel wat verschillende algoritmes om gegevens te sorteren. (Wikipedia telt er op dit moment een kleine vijftig, de ene al exotischer dan de andere.) *Sorteren door invoegen* (Engels: *insertion sort*) is daarvan zeker niet het snelste, maar is relatief eenvoudig om te begrijpen, werkt toch nog behoorlijk op kleine sets en lijkt goed op hoe mensen zelf sorteren, bijvoorbeeld wanneer ze bij het kaartspelen kaarten in hun hand op volgorde plaatsen.

In grote lijnen komt het algoritme neer op wat je in §3.1 deed om een lijst gesorteerd te houden terwijl je er nieuwe elementen aan toevoegt. Alleen gebeurt het nu in-place en zijn de elementen die je wilt invoegen al in de lijst aanwezig.



- Tijdens het algoritme is er steeds vooraan een eerste deel van de lijst waarvan de elementen reeds in volgorde staan. Achteraan is dit nog niet zo.
- In elke stap voeg je het eerste element van het achterste deel tussen in het voorste gedeelte, op een manier die bijna identiek is als bij de procedure `invoegen` hierboven. (Het opvallendste verschil is dat we de eerste if-opdracht kunnen weglaten. Waarom?)
- Zo wordt bij elke stap het gesorteerde deel één element groter, totdat uiteindelijk de ganse lijst is gesorteerd.

De afbeelding hiernaast toont de verschillende stappen die doorlopen worden bij het sorteren van de lijst [7,4,9,5,2,6]. Het groene gedeelte is gesorteerd, het gele element wordt er in een volgende stap tussengevoegd.

Het uiteindelijke algoritme ziet er zo uit – zie ook *3_isort.py*.

```
def sorteer(lijst):
    for lengte in range(1,len(lijst)): # lengte van het groene deel
        getal = lijst[lengte]          # gele getal
        # bijna letterlijke kopie van invoegen(…)
        index = lengte - 1
        while index >= 0 and lijst[index] > getal:
            lijst[index+1] = lijst[index]
            index -= 1
        lijst[index+1] = getal
```

Oefening 9.3 Schrijf een programma waarmee je test of de methode `sorteer` hierboven wel degelijk werkt. Doe dit door 1000 lijsten aan te maken met daarin telkens 20 willekeurige gehele getallen in het interval [0,30] (de kans is dus groot dat de lijst bepaalde getallen meerdere keren bevat). Sorteer elk van die lijsten en kijk of ze inderdaad gesorteerd zijn – zo nee print je de ongesorteerde lijst af. Schrijf hiervoor een functie `is_gesorteerd (lijst)` die van een lijst controleert of alle element in stijgende volgorde staan. Om er helemaal zeker van te zijn dat je test werkt, introduceer je eens een fout in `sorteer`. Vervang bijvoorbeeld `lengte-1` op lijn 5 door `lengte-2`. Dat zou foutmeldingen moeten opleveren.

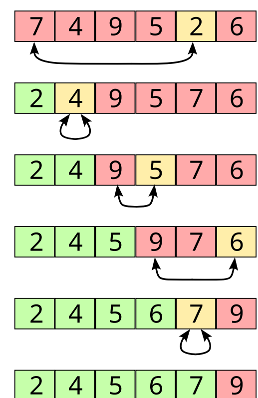
9.3 Selectiesortering

Dit is een alternatief voor §9.2 (Sorteren door invoegen). Het is niet nodig om in de les beide sorteeralgoritmes te behandelen. Hoewel ze vanuit programmeeroogpunt wel interessant zijn, zijn het in wezen inefficiënte algoritmes. Als je toch een bijkomend sorteeralgoritme introduceert, kies dan liever voor *merge sort* uit §??.

Om een lijst in-place te sorteren met behulp van selectiesortering, ga je als volgt te werk:

- Je verdeelt 'in gedachten' de lijst in twee delen – een 'groen' gedeelte dat reeds geordend is en een 'rood' gedeelte dat nog gesorteerd moet worden (zie figuur). 'In gedachten' betekent dat we bijvoorbeeld de lengte van het groene gedeelte bijhouden in een variabele.
- Bepaal wat het kleinste element is in het rode deel en (vooral) wat zijn positie is. (Dit element hebben we geel gekleurd in de afbeelding.)
- Verwissel dit kleinste element met het eerste element van het rode gedeelte. Deze positie wordt nu groen.
- Herhaal dit tot de volledige lijst gesorteerd is.

Merk op dat je moet beginnen met een groen gedeelte van lengte 0 maar dat je kan stoppen zodra het rode gedeelte slechts één element bevat.



Het uiteindelijk algoritme ziet er zo uit – zie ook 4_*ssort.py*.

```
def sorteer(lijst):
    for lengte in range(len(lijst)-1): # lengte van het groene deel
        # bepaal de positie van het minimum
        min_pos = lengte
        for pos in range (lengte+1,len(lijst)):
            if lijst[pos] < lijst[min_pos]:
                min_pos = pos
        # verwissel
        if min_pos != lengte:
            tijdelijk = lijst[lengte]
            lijst[lengte] = lijst[min_pos]
            lijst[min_pos] = tijdelijk
```

Oefening 9.4 Maak oefening 9.3 maar nu voor bovenstaande versie van sorteer. Om er helemaal zeker van te zijn dat je test werkt, introduceer je eens een fout in `sorteer`. Vervang bijvoorbeeld `lengte+1` op lijn 5 door `lengte+2`. Dat zou foutmeldingen moeten opleveren.

10 Verkeer

Overzicht

Algoritmen en datastructuren: wachtrij (queue)
Numerieke methoden: simulatie
Softwarebibliotheken: willekeurige getallen, grafieken maken

10.1 Simulatie

In dit hoofdstuk simuleren we het aan en afrijden van auto's aan een kruispunt met verkeerslichten. We kijken hoe lang de auto's gemiddeld moeten wachten en zetten dit uit in een grafiek.

- Elke minuut rijden er 18 auto's weg van het kruispunt
- Elke minuut komen er tussen 0 en 40 auto's toe op het kruispunt (een willekeurig aantal)
- De wachttijd van één auto die wegrijdt is het verschil tussen de minuut waarop hij aan het kruispunt aankomt en de minuut waarop hij wegrijdt. (Voor de eenvoud rekenen we in minuten en niet in seconden.)
- We wensen elke minuut de gemiddelde wachttijd te kennen van alle auto's die konden wegrijden. We zetten die uit in een grafiek. We volgen een tijdsspanne van 60 minuten.

2006 © M.M.Minderhoud

Hoewel we het in onze beschrijving hebben over auto's en kruispunten, is de enige informatie die je voor deze simulatie nodig hebt, het tijdstip waarop een auto aankomt. We stellen dit voor als een geheel getal – het aantal minuten sinds het begin van de simulatie. We moeten ook bijhouden welke auto's er nog staan te wachten om van het kruispunt weg te rijden. Daarvoor gebruiken we een *wachtrij* (zie inzet): de auto's die eerst aanrijden zijn ook de auto's die eerst wegrijden.

Wachtrij (queue)

Een *wachtrij* (Engels: *queue*) is een datastructuur waaraan je gemakkelijk elementen kunt toevoegen om ze er achteraf opnieuw *in dezelfde volgorde* uit op te halen. Dit volgt het FIFO-principe – '*first in, first out*': het verwerken van de elementen van een wachtrij gebeurt steeds in dezelfde volgorde waarop ze zijn aangeleverd, zoals bij een rij mensen die beleefd staan te wachten op een bus, of een rij auto's in de file.
In Python kan je een *lijst* gebruiken als wachtrij. Je voegt dan elementen achteraan toe met append en neemt ze vooraan terug weg met `pop(0)`. (Er bestaan ook meer efficiënte implementaties.)

Oefening 10.1 Programmeer deze simulatie. Denk goed na en bespreek wat er precies in die wachtrij moet worden opgeslagen. Je kan dit afleiden uit de beschrijving hierboven.

Voor elke nieuwe auto moet je het tijdstip aan de wachtrij toevoegen waarop die auto wordt toegevoegd. Wanneer je de auto dan later uit de wachtrij haalt, vergelijk je dat opgeslagen tijdstip met het 'huidige' tijdstip om te zien hoe lang de auto in de wachtrij heeft gestaan. Let erop dat je niet meer auto's uit de wachtrij probeert weg te halen dan erin staan! En als de wachtrij leeg is, moet je opletten bij het bepalen van de gemiddelde wachttijd – deel niet door 0!

Python

De uitdrukking `lijst.pop(index)` geeft het element terug op plaats `index` in `lijst` en *verwijdert* dat element dan ook meteen uit die lijst. Deze functie wordt vaak gebruikt bij wachtrijen (en stapels – hoofdstuk 11).

Wat zie je op de grafiek? Blijft de wachttijd de ganse tijd ongeveer hetzelfde? Of stijgt of daalt hij? Had je dit verwacht? Wat is het effect van het aanpassen van de 'parameters' – het aantal auto's dat per minuut kan wegrijden of het maximum aantal auto's dat per minuut kan komen aanrijden.

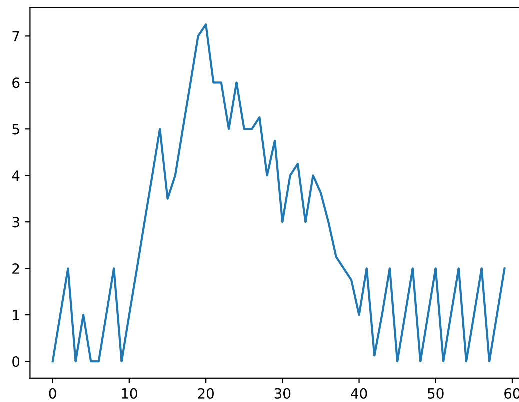
Met de oorspronkelijke parameters is het verkeer een ramp, wat ook niet te verwonderen is: gemiddeld komen er telkens 20 auto's bij en gaan er slechts 18 weg. Maar ook als er 20 auto's kunnen wegrijden, blijft de wachttijd nog licht-

jes stijgen. Dat komt omdat er geen 20 auto's kunnen wegrijden als de wachtrij minder dan 20 auto's bevat. Probeer ook eens wat er gebeurt als er telkens 40 auto's mogen wegrijden – dit is een goede test van het programma.

Oefening 10.2 Voeg verkeerslichten toe aan de simulatie: er kunnen tussen 40 à 70 nieuwe auto's het kruispunt oprijden maar enkel bij groen licht – op tijdstippen 0, 3, 6, …. Schrijf en gebruik hiervoor een functie `nieuwe_auto_s(t)` die teruggeeft hoeveel auto's er bijkomen op tijdstip `t` (= aantal minuten sinds het begin van de simulatie).

Oefening 10.3 Net voorbij het kruispunt gebeurt er een ongeval op tijdstip 10. Totdat het ongeval opgeruimd is (op tijdstip 20) kunnen daardoor slechts 6 auto's per minuut het kruispunt verlaten. Pas je oplossing van de vorige oefening aan om dit ongeval te simuleren. Schrijf en gebruik hierbij een functie `auto_s_weg(t)` die teruggeeft hoeveel auto's er op tijdstip `t` het kruispunt verlaten. Herken je het ongeval in de grafiek?

Het verkeer na het ongeval is opnieuw een ramp: na het ongeval heeft elke auto ± 10 minuten vertraging – en zelfs als je de simulatie een paar uur verder laat lopen, verbetert dit niet. Vervang je de 18 daarentegen door 24, en dan zie je dat de file zich wel terug oplost.



11 Woordladder

Overzicht

Algoritmen en datastructuren: stapel (stack), binair zoeken
Externe gegevensbronnen: tekstbestand inlezen als lijst
Softwarebibliotheken: binair zoeken met `bisect`

Een *woordladder* is een opeenvolging van woorden waarbij elk woord van het vorige in precies één letter verschilt. Een eenvoudig woordspelletje geeft je twee woorden op en vraagt je de kortst mogelijke woordladder te vinden (van bestaande Nederlandse woorden) die deze twee woorden met elkaar verbindt.

H	O	R	E	N
H	O	P	E	N
K	O	P	E	N
K	A	P	E	N
K	A	P	E	L

11.1 Woordladder controleren

Oefening 11.1 Schrijf een programma waarmee je een woordladder kunt ingeven en dat controleert of die aan alle regels voldoet. Vraag de twee woorden op bij de start van het programma.

```
Met welk woord wil je starten? horen
Met welk woord wil je eindigen? kapel

Je start met 'horen' en je moet eindigen met 'kapel'
> hopen
> hoppen
Het woord moet uit precies 5 letters bestaan
? hopen
Je moet minstens één letter veranderen
? hapen
'hapen' is geen Nederlands woord
? kopen
> kapel
'kapel' kan niet de volgende stap zijn na 'kopen'
? kapen
> kaper
> kapel
Dit is de woordenreeks die je hebt gebruikt:
horen
hopen
kopen
kapen
kaper
kapel
```

In de plaats van het start- en het eindwoord aan de gebruiker op te vragen, kan je ook twee willekeurige woorden kiezen uit de lijst van vijfletterwoorden die we in het programma gebruiken. Helaas is de kans op een 'speelbaar' paar woorden heel klein.

Zoals je uit de uitvoer kan zien, moet je vier verschillende regels controleren.

- Het woord dat je intikt moet uit precies 5 letters bestaan
- Het volgende woord mag niet in meer dan één letter verschillen van het vorige
- Het volgende woord moet in minstens één letter verschillen van het vorige
- Het woord moet een Nederlands woord zijn van 5 letters

Zolang aan één van die regels niet voldaan is, geef je een gepaste foutmelding en vraag je een nieuwe invoer – en je gebruikt daarbij een andere prompt (? in de plaats van >)

We plaatsen dit gedeelte van het programma in twee functies die we hieronder afdrukken. (Zie ook *1_woordladder.py*.)
De functie `zijn_regels_voldaan` controleert of aan alle regels is voldaan en geeft dan `True` terug. Zijn ze niet voldaan, dan geeft de functie `False` terug en drukt ze tegelijk een foutboodschap af.

```
def zijn_regels_voldaan (vorig_woord, nieuw_woord):
    voldaan = False
    if len(nieuw_woord) != 5:
        print ("Het woord moet uit precies 5 letters bestaan")
    else:
        verschillen = tel_verschillen(vorig_woord, nieuw_woord)
        if verschillen == 0:
            print("Je moet minstens één letter veranderen")
        elif verschillen > 1:
            print(f"'{nieuw_woord}' kan niet de volgende … na '{vorig_woord}'")
        elif not is_Nederlands(nieuw_woord):
            print(f"'{nieuw_woord}' is geen Nederlands woord")
        else:
            voldaan = True
    return voldaan
```

De functie `volgend_woord` gebruikt dan `zijn_regels_voldaan` om een geldige invoer te (blijven) vragen aan de gebruiker. Het eerste geldige resultaat wordt dan teruggegeven.

```
def volgend_woord (vorig_woord):
    nieuw_woord = input ("> ")
    while not zijn_regels_voldaan(vorig_woord, nieuw_woord):
        nieuw_woord = input ("? ")
    return nieuw_woord
```

En dit is dan het hoofdprogramma

```
start_woord = input("Met welk woord wil je starten? ")
eind_woord = input("Met welk woord wil je eindigen? ")
lijst = []

print()
print (f"Je start met '{start_woord}' en je moet eindigen met '{eind_woord}'")
woord = start_woord
lijst.append (woord)
while woord != eind_woord:
    woord = volgend_woord(woord)
    lijst.append (woord)

print ("Dit is de woordenreeks die je hebt gebruikt:")
for woord in lijst:
    print (woord)
```

Om helemaal volledig te zijn, moet je eigenlijk ook nog controleren of de twee woorden die in het begin van het programma worden ingegeven wel geldige Nederlandse woorden van 5 letters zijn. Zie uitbreidingsoefening 1051.

Er blijven nog twee functies over die je zelf moet programmeren

- De functie `tel_verschillen(woord1, woord2)` die telt op hoeveel plaatsen twee strings van elkaar verschillen en dit aantal teruggeeft. Je mag hierbij aannemen dat beide strings lengte 5 hebben – de functie wordt immers pas opgeroepen nadat hun lengte reeds is gecontroleerd.
  Je kan bovendien eisen dat je het tellen stopt zodra je minstens 2 verschillen hebt gevonden. In dat geval heb je een while-lus nodig met dubbele conditie (hoofdstuk ??) in de plaats van een eenvoudige for-lus.
- De functie `is_Nederlands(woord)` die kijkt of de gegeven string een Nederlands woord is (van vijf letters).

Voor dit laatste kan je het bestand *woorden5letters.txt* ge-

Python

Met `bestand.read().splitlines()` lees je een bestand in één keer in als een lijst van strings: `read()` leest het volledige bestand in één keer in als één lange string, `splitlines()` splitst die string dan op in afzonderlijke lijnen.

bruiken dat alle Nederlandse woorden van 5 letters bevat, in alfabetische volgorde.
Als je daar de tijd voor hebt, kan je *woorden5letters.txt* ook zelf door de leerlingen laten maken. Zie uitbreidingsoefening 11.2*.
Lees dit bestand in als een lijst van strings (zie inzet). Gebruik *binair zoeken* (§2.2) om er snel een gegeven woord in op te zoeken.
Natuurlijk ben je niet de eerste die binair zoeken nodig heeft in een Python-programma. Het mag je dus niet verwonderen dat er een Python-bibliotheek bestaat die je daarbij kan helpen – namelijk de module `bisect`.

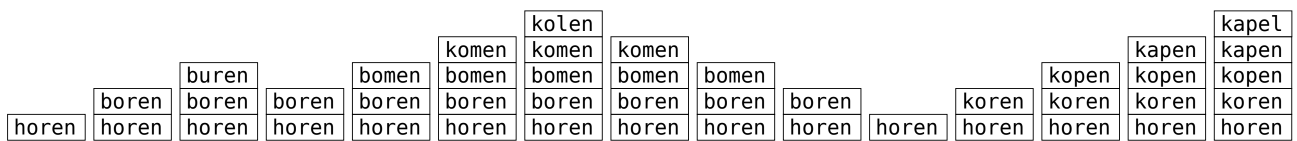
```
from bisect import bisect_left
```

Zoek zelf op, via het Internet, hoe je die functie `bisect_left` gebruikt om te zoeken of een element zich in een (geordende) lijst bevindt, en gebruik dit in jouw implementatie van `is_Nederlands`.

11.2 Op je stappen terugkeren

Oefening 11.2 Om het de gebruiker gemakkelijker te maken, voegen we een ‘opdracht’ toe aan het programma waarmee hij kan de vorige stap in de woordladder kan ongedaan maken: tik je een minteken in als ‘woord’ dan keert het programma één stap terug:

```
…
Je start met 'horen' en je moet eindigen met 'kapel'
> boren
> buren
> -
= boren
> bomen
> komen
> kolen
> -
= komen
> -
= bomen
> -
= boren
> -
= horen
> koren
> kopen
> kapen
> kapel
Dit is de woordenreeks die je hebt gebruikt:
horen
koren
…
kapel
```



De afbeelding schetst wat er in elke stap door het programma wordt bijgehouden. Geef je een nieuw (geldig) woord in, dan wordt dit bovenaan een *stapel* geplaatst (zie inzet op de volgende bladzijde). Geef je echter een minteken in, dan wordt het bovenste element van de stapel verwijderd, en drukt het programma af wat er nu bovenaan de stapel staat. Na afloop wordt de volledige stapel afgedrukt.
Pas de oplossing van oefening 11.1 aan zodat ze deze mintekenopdracht toelaat. Hou rekening met het volgende:

- Hou de ingegeven woorden dus bij in een stapel.

- Het moet mogelijk zijn om meer dan één keer achter elkaar een minteken in te geven
- Als de stapel slechts één element meer bevat, mag je er niets meer afhalen
- De functie `volgend_woord` (of `zijn_regels_voldaan`) moet ook `"-"` als een geldig woord aanvaarden

Stapel (stack)

Een *stapel* (Engels: *stack*) is een datastructuur waar je elementen aan toevoegt met de bedoeling ze er achteraf opnieuw *in omgekeerde volgorde* uit op te halen. Een stapel volgt het LIFO-principe – '*last in, first out*': het verwerken van de elementen gebeurt steeds in de omgekeerde volgorde waarop ze zijn aangeleverd, zoals je ook bij een stapel borden het laatste bord dat je erop plaatst het eerste is dat je er weer afhaalt. Een andere bewerking die men vaak op stapels gebruikt, is de zogenaamde *peek* (Engels): het bovenste element van de stapel bekijken *zonder* het eraf te halen.
Stapels worden onder andere gebruikt om een 'geschiedenis' bij te houden zodat gebruikers op hun stappen kunnen terugkeren – zoals bijvoorbeeld met de *back*-toets in een browser.
In Python kan een *lijst* dienen als stapel. Je voegt dan elementen achteraan toe met `append` en neemt ze terug weg met `pop()`. Voor een *peek* vraag je het element op met index -1.

12 Het Oekraïense vlagprobleem

Overzicht

Pre-algoritmen: *in-place* bewerken van lijsten
Algoritmen en datastructuren: testen, complexiteit
Softwarebibliotheken: willekeurige getallen, tijdsmeting, grafieken maken

Edsger Dijkstra

2002 © Hamilton Richards

Edsger Dijkstra (1930-2002) was een invloedrijke Nederlandse informaticawetenschapper wiens bijdragen een blijvende impact hebben gehad op het vakgebied. Hij was een pionier op het gebied van algoritmen, programmeren en formele verificatie. Dijkstra staat vooral bekend om zijn ontwikkeling van het *algoritme van Dijkstra*, een belangrijk algoritme voor het vinden van de kortste paden in een graaf[21] dat aan de basis ligt van de hedendaagse routeplanners. Hij was ook de grondlegger van het gestructureerd programmeren.

Dijkstra stelde het volgende vraagstuk (het *Nederlandse vlagprobleem*): stel dat je een lijst van elementen hebt, elk ofwel rood, wit of blauw gekleurd (de kleuren van de Nederlandse vlag), wat is dan een efficiënt algoritme om die in-place te herschikken zodat alle rode elementen vooraan staan, alle witte in het midden en alle blauwe achteraan?

Bron: Wikipedia – https://nl.wikipedia.org/wiki/Edsger_Dijkstra | https://en.wikipedia.org/wiki/Dutch_national_flag_problem

Voor ons is een bescheiden versie van het vraagstuk van Dijkstra, met de Oekraïense vlag die slechts twee kleuren heeft, nl. blauw en geel, al meer dan genoeg. En voor de eenvoud werken we ook niet met kleuren maar met getallen die ofwel even (blauw) zijn, of oneven (geel).
Om echt met 'kleuren' te werken heb je objecten nodig die o.a. een kleurattribuut hebben. Dit valt echter buiten het bestek van deze nota's.

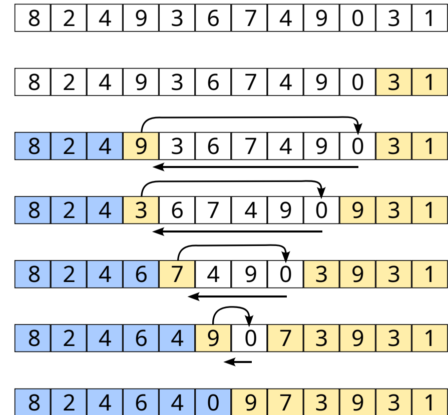
12.1 Herschikken van een lijst van even en oneven getallen

Het doel van deze paragraaf is om een efficiënte functie `herschik(lijst)` te schrijven die de gehele getallen in een lijst herschikt zodat alle even getallen vooraan komen te staan en alle oneven getallen achteraan. (Het is daarbij *niet* nodig dat de oorspronkelijke volgorde binnen elke groep behouden blijft!) In navolging van Dijkstra willen we bovendien dat dit in-place gebeurt!

Oefening 12.1 Zonder deze in-place-restrictie, vind je gemakkelijk een oplossing: doorloop de lijst van vóór naar achter, verwijder elk oneven element dat je tegenkomt en sla dit op in een andere lijst. Voeg die lijst dan op het einde toe aan de oorspronkelijke lijst. Programmeer deze implementatie en test ze.
Dit moet met een while-lus omdat we elementen uit de lijst verwijderen terwijl we ze aan het overlopen zijn. Merk op dat er een andere, misschien nog meer eenvoudige oplossing bestaat: overloop de lijst en maak twee hulplijsten aan, één met de even en één met de oneven elementen. Combineer beide lijsten op het einde.
Oefening 12.2 (Vertrek van *5_even_oneven_in_place.py*.) Om uiteindelijk een in-place-versie te maken, kan je hetzelfde principe toepassen, maar in de plaats van de oneven elementen in een afzonderlijke lijst te verzamelen, plaats je ze achteraan de huidige lijst, nadat je er plaats voor hebt vrijgemaakt door de overige elementen een positie naar voor op te schuiven – zoals in oefening 9.1. Je hebt twee mogelijkheden:

[21] Zie ook hoofdstuk ??.

- Je plaatst het oneven getal helemaal achteraan in de lijst. Dit zorgt echter voor complicaties wanneer je alle even getallen bent gepasseerd. Zie je wat de problemen zijn?
  Je krijgt een oneindige lus die de oneven elementen achteraan continu ‘roteert’.
- Je houdt bij welk gedeelte van de lijst achteraan enkel maar oneven getallen bevat en je plaatst het nieuwe oneven getal daar vlak voor.

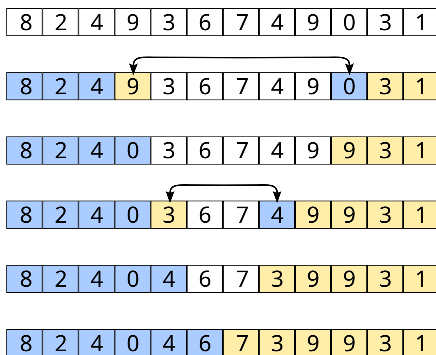


Hiernaast schetsen we het verloop van het algoritme. Het gele gedeelte achteraan bevat enkel oneven getallen, het blauwe vooraan enkel even getallen. We stoppen zodra beide gedeelten elkaar ‘ontmoeten’. Merk op dat we in een eerste stap alvast de oneven getallen achteraan de lijst overlopen om de positie te bepalen waar we het volgende oneven getal zullen invoegen.

Bij het programmeren van deze oplossing heb je twee indices nodig, één die van vóór naar achter in de lijst loopt, en één die van achter naar vóór loopt, en m.a.w. aangeeft waar het gele gebied zich bevindt. Hier is het belangrijk goed vast te leggen wat die twee teller precies voorstelt: de index van het eerste gele element, of de positie waar het volgende oneven getal terecht zal komen? (Beide versies zijn goed.) Ook bij de binnenste lus, waarmee een gedeelte van de lijst één plaats teruggeschoven wordt, moet je goed opletten wat precies de grenzen zijn.

Deze oplossing is in-place, maar het telkens één plaats opschuiven van een gedeelte van de lijst vraagt meer werk dan eigenlijk nodig is.

Oefening 12.3 (Lees de getoonde code en zorg dat je goed begrijpt wat er gebeurt.) Hieronder geven we je de oplossing zoals Dijkstra die voorstelt. Ze is in-place en overloopt elk element van de lijst maar één keer.



- Overloop de lijst van vóór naar achter totdat je een oneven getal vindt,
- … en van achter naar vóór totdat je een even getal vindt.
- Verwissel beide getallen van plaats.
- Herhaal dit tot alle getallen op hun juiste plaats staan.

Dit klinkt eenvoudig, maar het vergt veel aandacht om alle details exact juist te krijgen. Enkele moeilijkheden:

- Dit moet ook werken wanneer alle getallen even zijn of alle getallen oneven.
- Let erop dat je niet per ongeluk het laatste even getal nog verwisseld met het laatste oneven getal (6 en 7 in de laatste stap in de afbeelding hiernaast).

Zie 3_*oekraiense_vlag.py*.

```
def herschik (lijst):
    # zoek index van eerste oneven getal
    index_links = 0
    while index_links < len(lijst) and lijst[index_links] % 2 == 0:
        index_links += 1
    # zoek index van laatste even getal
    index_rechts = len(lijst)-1
    while index_rechts >= 0 and lijst[index_rechts] % 2 != 0:
        index_rechts -= 1
    …
                                                    (vervolg op volgende blz.)
```

```
    # herhaal zolang links rechts niet is gepasseerd
    while index_links < index_rechts:
        # verwissel de getallen
        t = lijst[index_links]
        lijst[index_links] = lijst[index_rechts]
        lijst[index_rechts] = t
        # zoek volgende oneven getal
        index_links += 1
        while lijst[index_links] % 2 == 0:
            index_links += 1
        # zoek volgende even getal
        index_rechts -= 1
        while lijst[index_rechts] % 2 != 0:
            index_rechts -= 1
```

12.2 Tijdmetingen

Misschien geloof je niet zomaar dat Dijkstra's methode sneller is dan die uit oefening 12.2? Dan zit er niets anders op dan beide methodes uit te proberen en te meten hoeveel tijd ze kosten. Hiertoe roep je de verschillende versies van `herschik` op voor lijsten van willekeurige gehele getallen. Hou rekening met het volgende:

- Let op dat je enkel meet hoelang `herschik` duurt, en niet hoelang het duurt om ook een lijst met willekeurige getallen op te stellen
- Om meer nauwkeurige resultaten te krijgen, roep je dezelfde `herschik` niet slechts één maar 20 keer op. Start de klok, roep de procedure 20 keer op en kijk daarna pas hoeveel tijd er verlopen is.
- Elk van die 20 keer moet met een verschillende lijst gebeuren. Je moet de 20 lijsten dus al hebben aangemaakt vóór de tijdsmeting start.

Als je dit allemaal samenneemt, krijg je dit: (zie 4_*tijdsmeting.py*)

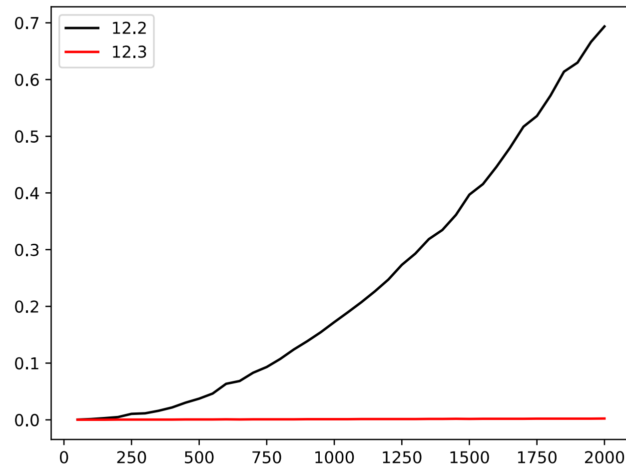
```
from time import time
from random import randint

# geef een lijst terug van een gegeven lengte, gevuld met willekeurig getallen
def maak_lijst(lengte):
    resultaat = []
    for _ in range(lengte):
        resultaat.append(randint(0,100))
    return resultaat

# hoelang duurt het om 20 willekeurige lijsten van de gegeven lengte te herschikken
def tijdsduur (lengte):
    # stop 20 lijsten in een lijst van lijsten
    lijsten = []
    for _ in range(20):
        lijsten.append (maak_lijst(lengte))
    tijd_begin = time ()
    for lijst in lijsten:
        herschik(lijst)
    tijd_einde = time ()
    return tijd_einde - tijd_begin
```

Oefening 12.4 Meet de tijd die je nodig hebt om lijsten van lengte 50, 100, 150, …, 2000 te herschikken en zet dit uit in een `matplotlib`-grafiek (met de lengtes op de X-as). Doe dit voor de herschikmethode uit oefening 12.2 en voor die uit oefening 12.3. Plaats de resultaten samen op één grafiek zodat je ze met elkaar kunt vergelijken.

Het is in Python toegelaten om procedures en functies als parameter mee te geven aan andere procedures of functies. Hier kan je dit eventueel gebruiken om geen twee verschillende versies van `tijdsduur` nodig te hebben[22].

[22]In een objectgerichte programmeertaal zou men hier kiezen voor overerving en/of polymorfisme.

Uit de grafiek lees je onmiddellijk af dat methode 12.2 een stuk trager is dan methode 12.3. Bovendien wordt het verschil tussen beide groter naargelang de lijsten langer worden.
Methode 12.3 (Dijkstra) toont in de grafiek een rechte lijn[23]. Als je de tijden afdrukt, zie je dat deze methode er ongeveer 2 keer langer over doet wanneer de lijst 2 keer zo lang is. Men zegt dat de *complexiteit* van dit algoritme *lineair* is. (De grafiek ziet eruit als een rechte lijn.)
Methode 12.2 doet er echter ongeveer 4 keer zo lang over om een lijst te verwerken die dubbel zoveel elementen bevat. De *complexiteit* van dit algoritme heet daarom *kwadratisch*. (De grafiek ziet eruit als een parabool.)



[23]Ze ziet er horizontaal uit, maar is dat niet. Ze stijgt uiterst langzaam.

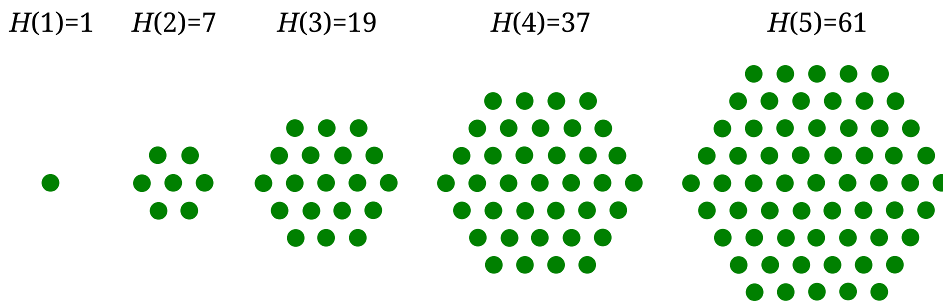
13 Recursie

Overzicht

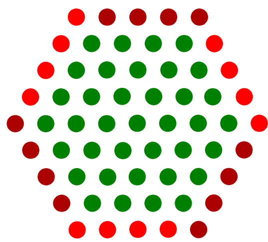
Algoritmische technieken: recursie
Softwarebibliotheken: tekeningen maken met `PLI`

13.1 Recurrente betrekkingen

Het aantal groene stippen in de zeshoeken hieronder noemt men centrale *zeshoeksgetallen*. Het getal $H(n)$ komt dan overeen met een zeshoek van 'zijde' n.



Hoe zou een Python-functie `h(n)` eruit zien die het getal $H(n)$ berekent voor gegeven n?
Zelfs voor wiskundige functies die met een hoofdletter geschreven worden, geven we er de voorkeur aan om een kleine letter te blijven gebruiken in Python. Hoofdletters worden in Python enkel gebruikt in namen van klassen.
De prent hieronder kan hier wellicht helpen als tip. Die toont je hoe je het volgende zeshoeksgetal berekent



uit het vorige. Een zeshoek met zijde 5 krijg je namelijk door 6 keer 4 stippen toe te voegen aan een zeshoek met zijde 4, en dus is $H(5) = H(4) + 6 \times 4$. Meer algemeen, krijg je een zeshoek met zijde n door 6 keer n-1 stippen toe te voegen aan een zeshoek van zijde n-1, en dus geldt

$$H(n) = H(n\text{-}1) + 6 \times (n\text{-}1).$$

Zo'n formule waarmee een functiewaarde voor parameter n wordt berekend uit dezelfde functie toegepast op n-1 (of eventueel ook op n-2, n-3, …) noemt men in de wiskunde een *recurrente betrekking*. Je kan zo'n formule bijna letterlijk overnemen in Python, en dan spreken we van een *recursieve* functie.

```
def h(n):
    return h(n-1) + 6*(n-1)          # nog niet 100% correct…
```

Om dit helemaal te doen werken, moet je nog een kleine aanpassing doen.
Bedenk bijvoorbeeld wat er gebeurt als je met bovenstaande functie de waarde van `h(2)` berekent: in een eerste stap wil Python de waarde van `h(1) + 6*1` uitrekenen, en daarvoor heeft hij `h(1)` nodig. Hij roept de functie `h(…)` dus opnieuw op, maar nu met n=1. Om `h(1)` uit te werken moet Python `h(0) + 6*0` evalueren, en dus `h(…)` nogmaals oproepen, maar nu met n=0. Om `h(0)` te bepalen, heeft Python in een volgende stap `h(-1)` nodig, en daarna `h(-2)`, en `h(-3)`, … en dit stopt nooit.
Je kan dit demonstreren door stap voor stap door het programma te lopen met een debugger.
Je kan deze 'oneindige lus' vermijden door de functiewaarde voor n=1 rechtstreeks terug te geven in de plaats van hiervoor opnieuw h(…) op te roepen. (*0_h_recursief.py*)

```
def h(n):
    if n==1:
        return 1
    else:
        return h(n-1) + 6*(n-1)
```

Zoals je in de inzet kan lezen, heb je bij elke recursie een *basisgeval* nodig waarvoor je geen functieoproep doet. Hier is dit basisgeval n=1.

Recursie

Recursie is een algoritmische techniek waarbij een probleem wordt opgelost door gebruik te maken van (de) oplossing(en) van eenzelfde, maar kleiner, probleem. In de praktijk wordt hierbij vaak een recursieve functie of procedure gebruikt – een functie of procedure die zichzelf oproept. Opdat dit zou werken – opdat het programma zeker zou stoppen – moet je opletten op het volgende:

- Wanneer je een functie oproept binnen zichzelf, moet je voor de 'binnenste' functie andere argumenten gebruiken dan de parameters van de 'buitenste'. Als het om een functie gaat met een geheel getal als parameter, dan is het 'binnenste argument' vaak kleiner dan de 'buitenste parameter'.
- Er moet steeds een *basisgeval* zijn waarbij de functie zichzelf niet oproept. Vaak is dit wanneer de parameter de kleinst mogelijke waarde aanneemt.

Je kan de centrale zeshoeksgetallen ook berekenen *zonder* recursie. Uit de formule hierboven haal je bijvoorbeeld de volgende eigenschap

$$H(n) = 1 + 6 + 12 + 18 + \ldots \text{ (} n \text{ termen)}$$

en die zet je gemakkelijk om naar een for-lus[24] (*0_h_niet_recursief.py*):

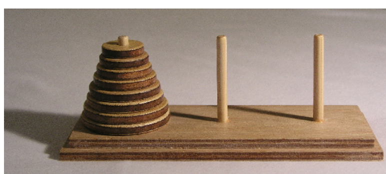
```
def h(n):
    som = 1
    for i in range(1,n):
        som += 6*i
    return som
```

Informaticawetenschappers kiezen graag voor een recursieve oplossing van een probleem wanneer dit korter en eleganter noteert. Vaak is de resulterende Python-code ook duidelijker[25].

Oefening 13.1 Schrijf een *recursieve* functie die berekent hoeveel kanonballen er liggen in een stapel van n ballen hoog. De basis van de stapel is een vierkant.

Oefening 13.2 Schrijf een recursieve functie die $n!$ (n faculteit) berekent. Gebruik de eigenschap $n! = n \times (n-1)!$.

13.2 Torens van Hanoi



2005 © Ævar Arnfjörð Bjarmason

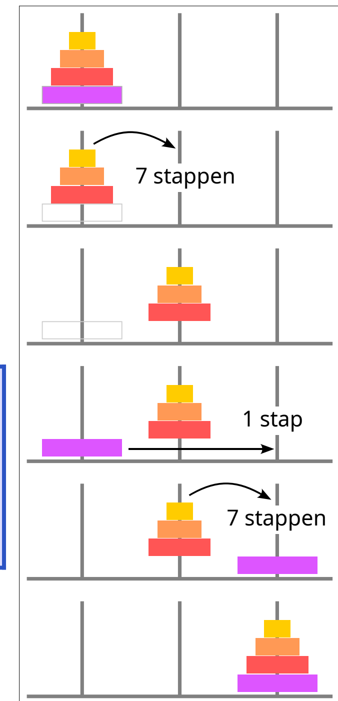
Recursieve functies en procedures hebben ook hun nut voor iets anders dan wiskundeformules. Als voorbeeld bespreken we hier *de Torens van Hanoi,* een bij informaticawetenschappers populaire puzzel die je met recursie kan oplossen. De puzzel is de volgende: op een plank staan drie staven waarover je ronde schijven kan schuiven van verschillende grootte. In het begin staan alle schijven in één kegelvormige stapel op de linkerstaaf. Het is aan jou om alle schijven naar de rechter staaf over te brengen met de volgende restricties: je verplaatst slechts één schijf per stap en je mag nooit een schijf plaatsen bovenop een schijf met een kleinere diameter. Hoe doe je dit (en hoeveel stappen heb je minstens nodig)?

[24]Je kan wiskundig nog een stapje verder gaan en bewijzen dat $H(n)=3n(n-1) + 1$. Maar dit is geen wiskundecursus.

[25]Een kleine waarschuwing is hier wel op zijn plaats: in de voorbeelden uit dit hoofdstuk is de recursieve versie even efficiënt als de niet-recursieve, maar dat is niet altijd het geval. Notoir tegenvoorbeeld is de beroemde Fibonacci-reeks ($F(n) = F(n-1) + F(n-2)$) – zie hoofdstuk ??.

Je kan de puzzel gemakkelijk online uitproberen[26]. Na enig zoeken vind je wel een manier om een toren van drie schijven in zeven stappen naar een andere staaf te verplaatsen – alhoewel je misschien op de middelste staaf terechtkomt in de plaats van helemaal rechts. Vier schijven is echter een stuk moeilijker (en kan in 15 stappen). Hier kan recursie ons helpen (zie figuur rechts):

- Doe even alsof de grootste schijf niet bestaat.
- Verplaats de stapel van drie schijven naar de middelste staaf (in 7 stappen)
- Neem de grootste schijf opnieuw in gedachten en verplaats ze van de linker naar de rechter staaf (1 stap)
- Negeer nogmaals de grootste schijf en verplaats de toren van drie schijven van de middelste naar de meest rechtse staaf (in 7 stappen)
- Je hebt nu de ganse toren van 4 schijven van links naar rechts verplaatst in 15 stappen.



Meer algemeen:

```
Om een toren met n schijven van staaf A naar C te plaatsen met
behulp van staaf B:
    - verplaats een toren met n-1 schijven van staaf A naar B
    - verplaats de laatste schijf van staaf A naar C
    - verplaats een toren met n-1 schijven van staaf B naar C
```

Dit is een *recursieve* procedure, want om een toren van *n* schijven te verplaatsen gebruik je dezelfde procedure (twee keer) maar nu voor een toren van *n*-1 schijven. De belangrijke vraag die we nu nog moeten beantwoorden is: 'Wat is het basisgeval? (En hoe voer je dat uit?)'. Uit het verhaal hierboven verwacht je misschien dat het basisgeval overeenkomt met een toren van 3 schijven, maar dat is niet correct. Welke toren dan wel?

Het voor de hand liggende antwoord is een 'toren' van 1 schijf. En die plaats je dan gewoon van de startschijf A naar doelschijf C (in 1 stap). Dit werkt, zoals we hieronder illustreren, maar je kan als basisgeval ook een 'toren' van 0 schijven gebruiken. En om die te verplaatsen hoef je helemaal niets te doen. Dit maakt de uiteindelijk implementatie nog iets eenvoudiger.

Een eerste stap om de schets hierboven naar een procedure in Python om te zetten, kan er zo uitzien:

```
def hanoi(n):
    if n==1:
        verplaats_schijf()
    else:
        hanoi(n-1)
        verplaats_schijf()
        hanoi(n-1)
```

Maar hier ontbreekt wel één en ander: `verplaats_schijf` moet weten van welke staaf naar welke staaf de schijf moet verplaatst worden, dit zijn namelijk niet altijd de linker en rechter staaf. Ook `hanoi` moet weten van welke staaf naar welke staaf de toren moet verplaatst worden. Voor `hanoi(n)` is dat altijd van uiterst links naar uiterst rechts, maar voor `hanoi(n-1)` is dat al niet meer het geval, en ook niet bij allebei de oproepen hetzelfde. Je moet dus de bewuste schijven telkens als parameter meegeven:

```
def hanoi(n,van,hulp,naar):                # bijv. hanoi(3,van=A,hulp=B,naar=C)
    if n==1:
        verplaats_schijf(van,naar)         # wordt niet uitgevoerd
    else:
        hanoi(n-1,van,naar,hulp)           # doe hanoi(2,van=A,hulp=C,naar=B)
        verplaats_schijf(van,naar)         # doe verplaats_schijf(van=A,naar=C)
        hanoi(n-1,hulp,van,naar)           # doe hanoi(2,van=B,hulp=A,naar=C)
```

[26]Zie bijvoorbeeld https://www.mathsisfun.com/games/towerofhanoi.html.

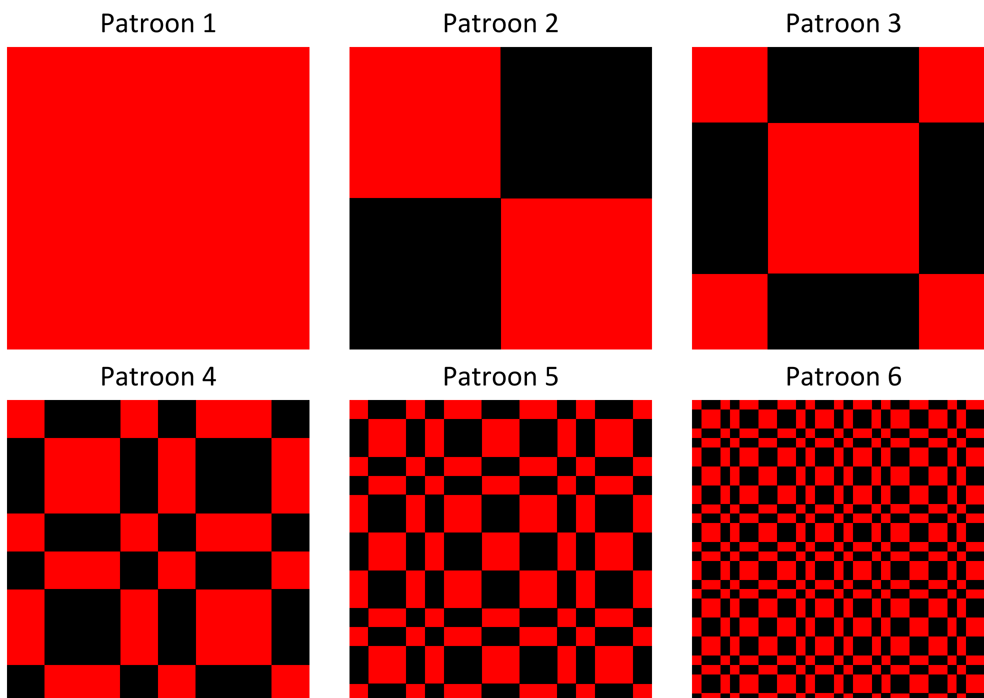
De parameters `van`, `naar` en `hulp` komen overeen met A, B en C in de schets hierboven.
We hebben voor het gemak ook de hulpstaaf als parameter meegegeven, ook al kan die bepaald worden als je de andere twee staven kent, maar dit maakt de code eenvoudiger. De code hierboven is nog niet volledig – zo ontbreekt de procedure verplaats_schijf(…). Zie uitbreidingsoefeningen 13.1*-3*.
Om de code beter te begrijpen, helpt het om stap voor stap na te gaan wat er gebeurt als je `hanoi(3,A,B,C)` oproept:

```
hanoi(3,A,B,C)
    hanoi(2,A,C,B)
        hanoi(1,A,B,C)
            verplaats_schijf(A,C)          A → C
        verplaats_schijf(A,B)              A → B
        hanoi(1,C,A,B)
            verplaats_schijf(C,B)          C → B
    verplaats_schijf(A,C)                  A → C
    hanoi(2,B,A,C)
        hanoi(1,B,C,A)
            verplaats_schijf(B,A)          B → A
        verplaats_schijf(B,C)              C → B
        hanoi(1,A,B,C)
            verplaats_schijf(A,C)          A → C
```

Vergewis je ervan dat de 7 stappen die hier uitgevoerd worden wel degelijk alle regels volgen en een toren met 3 schijven van staaf A naar C verplaatsen!

13.3 Recursief tekenen

Bekijk de tekeningen hieronder. Zie je hoe je de volgende tekening kunt maken uit de vorige?



Patroon *n*+1 bevat 4 kopieën van (kleinere versies van) patroon *n*, twee (linksboven en rechtsonder) in de originele kleuren en twee (linksonder en rechtsboven) waarbij de kleuren rood en zwart zijn omgewisseld.
De oneven patroon kan je ook bekomen door de even patronen 2 keer rechtstreeks en 2 keer gespiegeld te tekenen, maar deze eigenschap kunnen we hier niet gemakkelijk gebruiken.

Oefening 13.3 Teken Patroon 6 in een vierkant van 256 × 256 pixels (met PIL). Omdat het volgende patroon telkens gebruikt maakt van het vorige, ligt het voor de hand om dit met behulp van recursie te programmeren – in grote lijnen op de volgende manier (*3_patroon.py*):

```
def teken_vierkant (???):
    ???

def teken_patroon (n, ???):
    if n == 1:
        teken_vierkant (???)
    else:
        teken_patroon (n-1, ???)
        teken_patroon (n-1, ???)
        teken_patroon (n-1, ???)
        teken_patroon (n-1, ???)

from PIL import Image, ImageDraw

# we tekenen Patroon 6
prent = Image.new("RGB", [300, 300], color="grey")
doek = ImageDraw.Draw(prent)
teken_patroon (6, ???)
prent.show()
```

Maar net zoals bij de torens van Hanoi heb je ook hier meer parameters nodig dan enkel *n* – er zijn er heel wat meer. Kijk eerst naar `teken_vierkant`. Welke informatie heb je nodig om een vierkant te tekenen?

- De lengte van de zijde van het vierkant
- De X- en Y-coördinaten van de linker bovenhoek van het vierkant
- De kleur van het vierkant. Gebruik hier ofwel de string "red" of "black" omdat je die dan rechtstreeks als argument kunt meegeven aan de tekenroutines van PIL.

Diezelfde informatie heb je trouwens ook nodig bij de volledige patronen

- De lengte van de zijde van het volledige patroon.
- De coördinaten van de linker bovenhoek van het patroon.
- De twee kleuren van het patroon – de kleur van de linker bovenhoek en de andere kleur.
  Net zoals de staven bij de torens van Hanoi, kan je de tweede kleur natuurlijk steeds afleiden uit de eerste, maar twee kleuren als parameters meegeven maakt het programma korter en eleganter.

Wat je je nu moet afvragen is het volgende

- Wat gebeurt er met de zijde als we van niveau *n* naar niveau *n*-1 ‘afdalen’?
- En met de coördinaten?
- En met de kleuren?

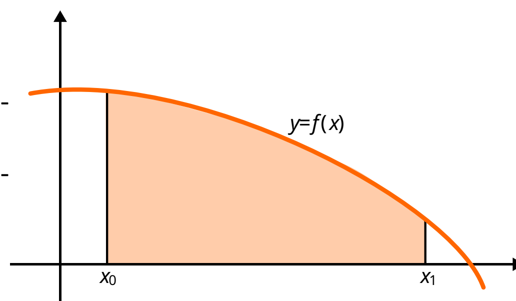
Met deze informatie heb je genoeg om het programma te vervolledigen.

14 Oppervlakte

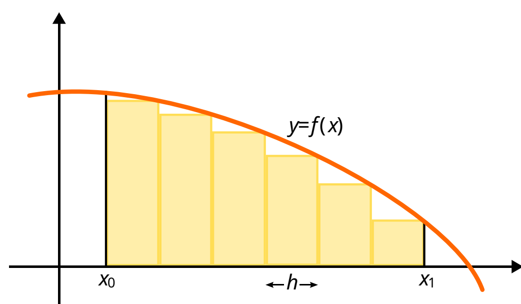
Overzicht

Numerieke algoritmen: oppervlakte onder de grafiek van een functie, trapeziumregel

14.1 Oppervlakte onder de grafiek van een functie

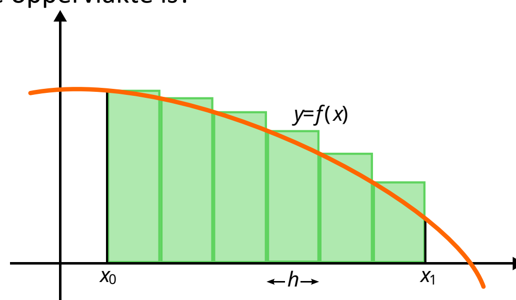


Hoe bepaal je de oppervlakte onder de grafiek van een gegeven functie *f*, in een gegeven interval [x_0,x_1]?
Heeft de functie een eenvoudig voorschrift, dan kan je de oppervlakte vaak exact bepalen met behulp van integraalrekening, maar in de praktijk is dat niet vaak zo of kent men zelfs het voorschrift niet van de functie, maar enkel zijn functiewaarden voor een aantal X-waarden. En dan moet men numerieke benaderingsmethoden gebruiken.



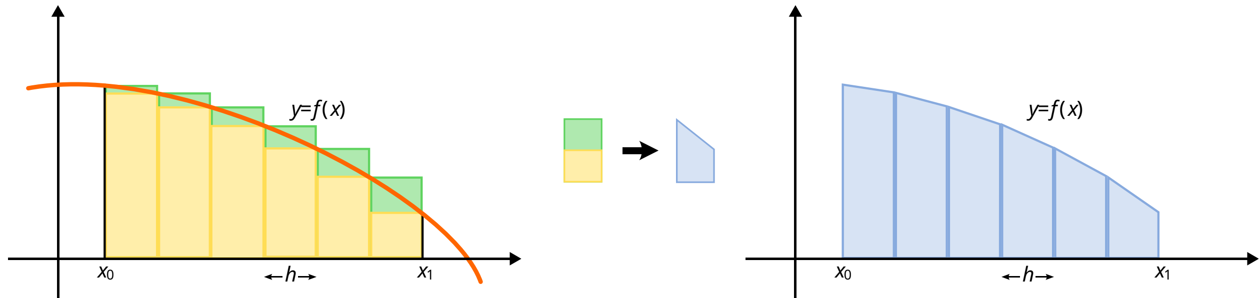
Een eerste mogelijkheid bestaat erin om de gevraagde oppervlakte te benaderen als een som van rechthoeken van gelijke breedte *h*, zoals in de figuur hiernaast. Hoe meer rechthoeken, hoe kleiner *h*, hoe beter de benadering.
De werkelijke oppervlakte is 17/3 = 5.333… Kijk na dat die waarde wel degelijk van onderaan wordt benaderd. Een veel gemaakte denk- of programmeerfout resulteert in een foute keuze voor *x* bij de berekening van de hoogte van de rechthoek waardoor de waarde van bovenaf wordt benaderd (wat eigenlijk oefening 14.2 is).

Oefening 14.1 Bereken op die manier de oppervlakte onder de grafiek van de functie $f(x)=8-x^2$ voor het interval [1.0,2.0]. Schrijf de opeenvolgende waarden uit voor een verdeling in 100, 200, 300, 400, … 2000 rechthoeken van gelijke breedte. Kan je raden wat de werkelijke oppervlakte is?



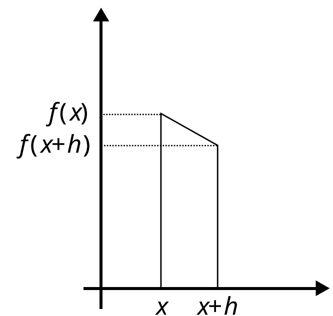
Merk op dat de benaderde waarde op die manier steeds kleiner is dan de werkelijke waarde omdat de rechthoeken steeds volledig onder de grafiek vallen. Je kan echter ook rechthoeken gebruiken die steeds een beetje te groot zijn – zie figuur.
Oefening 14.2 Pas je oplossing van de vorige oefening aan zodat de oppervlakte op deze tweede manier wordt berekend.
Dit vraagt slechts een heel kleine aanpassing – in één enkele lijn.
Oefening 14.3 Pas in jouw beide oplossingen het interval aan tot [-2.0,2.0]. (Als je dit goed geprogrammeerd hebt, komt het erop neer om telkens op één plaats een 1.0 in een 2.0 te veranderen.) Vergelijk de resultaten van beide methoden met elkaar? Wat merk je op? Wat is daarvoor de reden?
Als je blindelings de voorbeeldoplossingen volgt, leveren die bij dit nieuwe interval beide exact dezelfde waarde op. Dit komt doordat de functie symmetrisch is t.o.v. de Y-as en dat het eerste deel stijgend is en het tweede deel dalend. De afbeeldingen op deze bladzijde tonen een dalende functie en suggereren daardoor dat de hoogte van de onderliggende (gele) rechthoek overeenkomt met de functiewaarde van de rechterkant van de rechthoek (*x*+*h* in het interval [*x*,*x*+*h*]). Maar bij een stijgende functie is dat niet correct, daar moet je de functiewaarde nemen van de linkerkant (*x*). Je kan dit oplossen door een gepaste if-opdracht, maar in de praktijk is dit niet belangrijk, omdat men liever een wat meer gesofisticeerde benaderingsmethode gebruikt waarbij dit probleem zich niet voordoet.
Welke methode is nu de beste om te gebruiken? Zoals vaak ligt ‘de waarheid in het midden’: het *gemiddelde* van beide methoden levert een betere benadering op dan beide methoden afzonderlijk. Dit gemiddelde kan ook direct berekend worden zonder eerst beide methoden afzonderlijk uit te voeren – we kunnen stap

voor stap het gemiddelde bepalen van de gele en de groene rechthoek. Dit komt er op neer dat we de oppervlakte onder de grafiek niet benaderen door rechthoeken, maar door trapeziums – zie figuur.



Deze manier om de oppervlakte te berekenen, heet de *trapeziumregel*.
De oppervlakte van één trapezium wordt gegeven door basis × (lange zijde + korte zijde)/2. Hier dus $h \times (f(x)+f(x+h))/2$.
De som van alle trapeziums samen is

$$\frac{h}{2}[f(x_0)+f(x_0+h)]+\frac{h}{2}[f(x_0+h)+f(x_0+2h)]+\cdots+\frac{h}{2}[f(x_1-h)+f(x_1)]$$



Merk op dat elke term van de vorm *f*(...) hier twee keer wordt herhaald, behalve de twee uiterste. We kunnen dit dus herschrijven als

$$h \times \left[\frac{f(x_0)}{2}+f(x_0+h)+f(x_0+2k)+\cdots+f(x_1-h)+\frac{f(x_1)}{2}\right]$$

wat vereenvoudigt tot

$$h \times \left[[f(x_0)+f(x_0+h)+f(x_0+2h)+\cdots+f(x_1)]-\frac{f(x_0)+f(x_1)}{2}\right]$$

Oefening 14.4 Bereken de oppervlakte onder de grafiek van de functie $f(x)=8-x^2$ voor het interval [1.0,2.0] met behulp van deze laatste formule.