Algoritmen en programmeren

Uitbreidingsoefeningen

Kris Coolsaet
2024

Uitbreidingsoefeningen

De oefeningen zijn geordend per hoofdstuk

1 Algoritmen en programmeren

Oefening 1.1* Ingebouwd vs. zelf geprogrammeerd.

(Deze oefening stel je best uit tot na hoofdstuk 3 – er moeten tijdsmetingen worden gedaan)

Hieronder vind je twee verschillende definities van een functie `roteer` die alle elementen in een lijst één plaats naar rechts verschuift en het laatste element helemaal vooraan plaatst. De eerste definitie gebruikt een for-lus om elk element van de lijst één voor één te verplaatsen, de tweede definitie gebruikt ingebouwde slice-bewerkingen van Python.

```
# met for-lus
def roteer_1 (lijst):
    laatste = lijst[-1]
    for index in range(1,len(lijst)):
        lijst[-index] = lijst[-index-1]
    lijst[0] = laatste

# met ingebouwde slice-bewerkingen
def roteer_2 (lijst):
    laatste = lijst[-1]
    lijst[1:] = lijst[:-1]
    lijst[0] = laatste
```

Van welke van deze twee verwacht je dat ze het snelst wordt uitgevoerd? Of verwacht je weinig verschil in uitvoeringstijd? Test dit uit met een kort Python-programma, voor een lijst van een duizendtal elementen. (Vertrek van *1_roteren.py*.)

De tweede versie blijkt bijna 10 keer sneller te zijn dan de eerste. Dit bevestigt wat we in §1.2 reeds hadden aangehaald: ingebouwde functies in Python zijn een stuk sneller dan functies die je zelf schrijft.

Oefening 1.2* Vooraan of achteraan toevoegen aan een lijst.

(Deze oefening stel je best uit tot na hoofdstuk 7 – er wordt immers een grafiek getekend)

Hieronder zie je twee versies van een functie die een lijst omkeert – een nieuwe lijst aanmaakt met daarin dezelfde elementen als de oorspronkelijke, maar in omgekeerde volgorde. De eerste functie maakt de nieuwe lijst aan door achteraan elementen toe te voegen, de tweede door ze vooraan toe te voegen.

```
# invoegen achteraan de nieuwe lijst
def omkeren_1 (lijst):
    resultaat = []
    for getal in lijst:
        resultaat.append(getal)
    return resultaat

# invoegen vooraan de nieuwe lijst
def omkeren_2 (lijst):
    resultaat = []
    for index in range (1,len(lijst)+1):
        resultaat.insert(0,lijst[-index])
    return resultaat
```

Van welke van deze twee verwacht je dat ze het snelst wordt uitgevoerd? Of verwacht je weinig verschil in uitvoeringstijd? Test dit uit met een kort Python-programma, telkens voor een lijst van 10000, 20000, …, 90000 elementen. Maak een grafiek waarop je een (eventueel) verschil in snelheid duidelijk kan zien. (Vertrek van 2_*invoegen_toevoegen.py*.)

De eerste versie is duidelijk een stuk sneller dan de tweede, en het verschil wordt groter naarmate de lijst groter wordt. Python voorziet in het computergeheugen voor elke lijst *achteraan* alvast wat reserveplaats voor het geval er een element moet worden toegevoegd. (Is alle plaats opgebruikt, dan wordt de lijst naar een nieuwe geheugenplaats gekopieerd die meteen nog wat meer reserve voorziet.) Vooraan is er echter geen reserve, en dus moeten de andere elementen één voor één worden opgeschoven zodra we vooraan een nieuw element toevoegen – zoals in oefening 1.1* hierboven.

Oefening 1.3* De variantie.

In de statistiek wordt de *variantie* van een reeks waarden $x_1, x_2, \ldots, x_N$ gegeven door de volgende twee (equivalente) formules

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{n}(x_i-\mu)^2 = \left(\frac{1}{N}\sum_{i=1}^{n}x_i^2\right)-\mu^2$$

waarbij N het aantal waarden voorstelt en μ hun gemiddelde. De variantie is een maat voor de spreiding van de gegevens.

Schrijf twee functies `variantie_1` en `variantie_2` die de variantie berekenen van een lijst `x` van kommagetallen en controleer op een aantal random voorbeelden dat beide formules wel degelijk dezelfde waarde hebben.

Dit is een illustratie van wat we in §1.7 hebben vermeld als 'vertalen van een formule naar een programma'. Het eindresultaat is een vrij eenvoudige for-lus die vanuit programmeeroogpunt weinig interessants te bieden heeft.

In de (numerieke) informaticawetenschappen geeft men de voorkeur aan de tweede formule boven de eerste. Kan je bedenken waarom? Wat als je de gegevens uit een (zeer groot) bestand moet halen, dit bestand maar één keer kan overlopen en het niet eerst kan opslaan in een lijst?

De tweede formule heeft het voordeel dat je ze kan berekenen door alle waarden slechts één keer te overlopen. Je moet dan wel tegelijkertijd de som van de kwadraten en som van de elementen zelf berekenen – en als je vanaf een bestand inleest wellicht ook nog tellen hoeveel waarden er zijn. Bij de eerste formule moet je eerst het gemiddelde bepalen en dan een tweede keer door alle waarden lopen om de som van hun kwadraten te berekenen.

Oefening 1.4* Cosinus.

$$\cos x = 1-\frac{x^2}{1\cdot 2}+\frac{x^4}{1\cdot 2\cdot 3\cdot 4}-\frac{x^6}{1\cdot 2\cdots 6}+\frac{x^8}{1\cdot 2\cdots 8}-\cdots$$

Je kan de cosinus van een hoek x berekenen door voldoende termen uit de volgende reeks op te tellen. Schrijf een functie `cosinus(x)` die de cosinus van x op deze manier berekent tot op 18 cijfers na de komma – m.a.w. stop wanneer de laatste term die je bij de som hebt opgeteld kleiner wordt dan 10^{-18} (in Python wordt dit getal genoteerd als 1E-18, m.a.w., in *wetenschappelijke notatie*.)

Bereken op die manier de cosinus van π/10 en druk die af naast de waarde die door de ingebouwde cosinusfunctie (`math.cos`) door Python zelf wordt teruggegeven. Doe dit ook met 61π/10, die exact dezelfde waarde zou moeten opleveren. Wat merk je?

Voor π/10 zouden de beide waarden van cosinus dezelfde moeten zijn, voor 61π/10 zie je echter een verschil bij de functie die je zelf hebt geprogrammeerd. Dit komt door afrondingsfouten en het feit dat je in het tweede geval veel meer termen nodig hebt dan in het eerste om tot dezelfde precisie te komen. Het ingebouwde algoritme houdt daar rekening mee door eerst met behulp van goniometrische formules alle cosinussen en sinussen te reduceren tot hoeken tussen 0 en π/4.

2 Zoeken

Oefening 2.1* Alle nulwaarden.
Schrijf een programma dat alle drie de nulwaarden bepaalt van de functie $f(x) = x^3 - 8x - 3$, telkens tot op 12 cijfers na de komma. Hierbij is het belangrijk dat je niet gewoon dezelfde code drie keer knipt en plakt, maar dat je één functie schrijft die je dan drie keer oproept. Denk goed na over welke parameters je die functie zult geven.
Het bepalen van de grootste nulwaarde (de meest rechtse) verloopt bijna op dezelfde manier als bij de kleinste. Voor de middelste moet je echter bijkomende moeite doen (waarom?). Schrijf daarom eerst een tweede functie waarmee je het middelste geval kunt oplossen. Pas als die werkt, bedenk je hoe van die twee functies er één kunt maken door een bijkomende parameter toe te voegen.
Het verschil tussen het interval dat je gebruikt om de eerste nulwaarde te bepalen en dat voor de tweede is het teken van de functiewaarde aan beide kanten van het interval: bij het eerste interval heeft het linker eindpunt een negatieve functiewaarde en het rechter eindpunt een positieve, terwijl dit bij het tweede interval omgekeerd is. Voeg daarom een bijkomende parameter toe die aangeeft in welke van beide situaties je u bevindt (stijgende of dalende functie).

Oefening 2.2* Nulwaarden van een veelterm van de derde graad.
Schrijf een programma dat een nulwaarde bepaalt van een functie van de vorm $f(x) = x^3 - px - q$, waarbij de gebruiker de kommagetallen p en q ingeeft.
Deze opgave lijkt op de oefening uit de les (en op oefening 2.1* hierboven), maar dat p en q niet vooraf bekend zijn, maakt het een beetje moeilijker – we kunnen niet zomaar op voorhand een eerste interval vastleggen waarmee we de zoektocht beginnen. Bij elke keuze die we maken, kan er wel een p en een q zijn waarvoor de eindpunten van het interval beide een positieve of beide een negatieve functiewaarde hebben, en dan werkt het algoritme niet.
Gelukkig schiet de wiskunde ons hier te hulp: omdat f een veelterm is van een oneven graad en de coëfficiënt van de hoogste graad positief is, zijn we zeker dat er steeds een x kan gevonden worden waarvoor $f(x) > 0$, als we x maar groot genoeg kiezen, en ook steeds een x' waarvoor $f(x') < 0$, als we x' maar klein genoeg kiezen. Om x te vinden, kan je als volgt te werk gaan: kijk eerst of x=1 voldoet, en probeer anders x=2, 4, 8, 16, … (telkens verdubbelen). En voor x' kan je hetzelfde doen met opeenvolgende waarden x'=-1, -2, -4, …
Het zoeken naar x en x' gebeurt met een eenvoudige while-lus. Eigenlijk hoef je maar één van die twee lussen uit te voeren, want als de eerste x niet voldoet, kan je die als x' gebruiken. Merk op dat x' geen correcte naam is voor een variabele in Python.

3 Postnummers

Oefening 3.1* Ordenen op naam van de gemeente.
In oefening 3.1 heb je een bestand van gemeenten dat geordend is op naam omgezet naar een lijst die geordend is op postnummer. In deze oefening doe je het omgekeerde: je vertrekt van het bestand *postnummers-2.txt* dat op postnummer is geordend

```
1000 Brussel
1000 Bruxelles
1005 Ass. Réun. Com. Communau. Commune
1005 Brusselse Hoofdstedelijke Raad
1005 Conseil Region Bruxelles-Capitale
…
(2920 lijnen)
```

en je zet dit om naar een lijst die gerangschikt is op naam. Vertrek van je oplossing voor oefening 3.1. Er is maar een kleine aanpassing nodig. Zoek op het einde ter controle een aantal gemeenten op aan de hand van hun naam.
Dit keer moet je geen volledige lijnen met elkaar vergelijken, maar enkel hun *slices* vanaf positie 5. Bij parameter `element` heb je de keuze om de volledige lijn te gebruiken of enkel de *slice*. Deze laatste optie maakt het eenvoudiger om achteraf een gemeente op te zoeken op naam.

4 Poker

Oefening 4.1* Bereken de kans op een *flush* in een Pokerhand , d.w.z., vijf kaarten van dezelfde kleur. Gebruik hierbij de andere methode om 5 willekeurige kaarten te trekken dan waar je in oefeningen 4.3/4.4 voor had gekozen.

De kans is ±0,20%. Vermijd om uit een lus te springen bij het nakijken of alle vijf kaarten dezelfde kleur hebben. Gebruik liever een *while met dubbele conditie* – zie hoofdstuk ?.

Oefening 4.2* Kans of een *full house*.

Dit lijkt een eenvoudige variant op de oefeningen uit het hoofdstuk, maar nakijken of een hand een full house is, leidt zeer gemakkelijk tot lange en onleesbare code. De netste oplossing is om een frequentietabel te gebruiken maar die wordt pas geïntroduceerd in hoofdstuk 7. Anderzijds kan dit dus een goede oefening zijn om achteraf het nut van een frequentietabel in te zien. Zie ook oefening 4.4*.

Een pokerhand heet 'full house' wanneer ze bestaat uit drie kaarten met dezelfde waarde, gecombineerd met twee andere kaarten van dezelfde (andere) waarde – een combinatie van een paar en een trio. Gebruik opnieuw de Monte Carlo-methode om deze kans te bepalen.

De kans is ±0,14%.

Oefening 4.3* *Echte* kans op een four of a kind

Deze oefening is een illustratie van de algoritmische techniek *exhaustief zoeken*. Ze gebruikt ook vernestelde for-lussen met grenzen die afhangen van de tellers (pre-algoritme).

Om de (wiskundig) correcte kans te bepalen op een kwartet moet je niet zomaar willekeurige sets van 5 kaarten bekijken, zelfs als doe je dit een miljoen keer, maar moet je *alle* mogelijke sets van 5 kaarten overlopen. Omdat we de volgorde waarin we de kaarten trekken niet belangrijk is (harten aas, heer, vrouw, boer en 10 is dezelfde 'hand' als harten tien, vrouw, aas, heer en boer) is dit nog net haalbaar voor een moderne computer – er zijn ongeveer 2,6 miljoen mogelijkheden.

Hoe zorg je er voor dat je precies alle mogelijkheden overloopt? Opnieuw stel je de kaarten voor met getallen in het interval [0,51]. Je wil dus alle vijftallen $[k_1, k_2, k_3, k_4, k_5]$ overlopen met $0 \le k_1, k_2, k_3, k_4, k_5 \le 51$. Omdat de vijf kaarten verschillend moeten zijn, en omdat de volgorde van kaarten binnen een hand niet belangrijk is, kunnen we die volgorde zelf kiezen zodat $0 \le k_1 < k_2 < k_3 < k_4 < k_5 \le 51$ geldt.

Bekijk eerst het meer eenvoudige geval met slechts twee kaartnummers. Hoe zou je alle tweetallen $[k_1, k_2]$ overlopen waarvoor $0 \le k_1 < k_2 \le 51$, m.a.w., de tweetallen

[0,1], [0,2], [1,2], [0,3], [1,3], [2,3], [0,4], [1,4], [2,4], [3,4], … , [0,51], [1,51], [2,51], …, [50,51]

Dit doe je met twee vernestelde for-lussen waarbij de grens van de tweede afhangt van de teller van de eerste:

```
for k2 in range (52):
    for k1 in range(k2):
         # doe iets met k1 en k2
```

Alle vijftallen overlopen doe je op dezelfde manier, maar dan met 5 vernestelde for-lussen in plaats van slechts 2!

Zoals vaak bij exhaustieve algoritmen, kan je het algoritme versnellen door wat dieper na te denken. In dit geval kan je bedenken dat het aantal 5-tallen met een kwartet erin dat een klaverenaas bevat exact hetzelfde moet zijn als het aantal 5-tallen met een kwartet erin dat een schoppenvijf bevat, of eender welke andere kaart. Je kan met andere woorden bij het tellen één kaart vast kiezen, bijvoorbeeld k_5=51. Je krijgt dan 4 vernestelde lussen in plaats van 5 wat het algoritme ongeveer 10 keer sneller maakt.

Oefening 4.4* Is random wel willekeurig genoeg?

Ook hier gebruik je best een frequentietabel, die echter pas in hoofdstuk 7 wordt behandeld – zie ook oefening 4.2*.

Simuleer het 6000 keer gooien van een dobbelsteen en druk af hoeveel enen je hebt gegooid, hoeveel tweeën, hoeveel drieën, enz. Elk aantal zou ongeveer 1000 moeten zijn – als het 6 keer exact 1000 is, dan is er iets verdachts aan de hand. Van een echte dobbelsteen zou je dit ook niet verwachten.

Oefening 4.5* Dobbelen

Jouw vriend stelt het volgende spelletje voor: je gooit met twee dobbelstenen, als het totaal aantal ogen 5,

6, 7 of 8 is, wint jouw vriend, anders win jij. Wie maakt er meest kans om te winnen? Gebruik de Monte Carlomethode om deze kans in te schatten.
De kans dat je vriend wint is precies 5/9 = 55.555%.

Oefening 4.6* Verzwaarde dobbelstenen
De dobbelstenen die je in het spelletje van oefening 4.5* gebruikt, zijn niet helemaal eerlijk: ze zijn verzwaard zodat de kans om een aas (1) te gooien dubbel zo groot is dan voor de andere getallen. Wat is nu de kans dat jouw vriend wint? Bedenk eerst hoe je met willekeurige getallen ervoor kunt zorgen dat de kans op een 1 dubbel zo groot is als bij de andere getallen.
Vraag een willekeurig getal op in het interval [0,6]. Als het resultaat 0 is, wijzig het dan in 1. De kans dat je vriend wint is nu precies 26/49 = 53.06122%. Hij maakt dus nog steeds de meeste kans.

5 Chatbot

Oefening 5.1* Een dictionary inlezen vanuit een bestand.
Je hebt misschien nog niet geleerd hoe je het woordenboek uit oefening 5.1 kan opslaan in een bestand, maar het omgekeerde kan je wel: het woordenboek 'inlezen' vanuit een bestand. Het bestand *nl_fr.txt* heeft de volgende inhoud

```
huis maison
auto voiture
tafel table
boom arbre
…
```

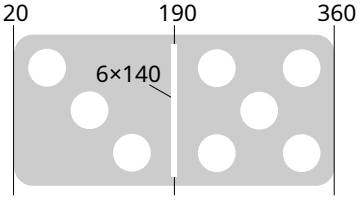
Pas je oplossing van oefening 5.1 zo aan dat het beginwoordenboek wordt ingelezen vanuit dit bestand. Elke lijn bestaat uit een Nederlands woord, een spatie en de Franse vertaling van dat woord.
Elke lijn opsplitsen in twee afzonderlijke woorden gaat het gemakkelijkst met `lijn.split()`. Dan is er ook geen probleem met het bijkomende *new line*-teken aan het einde van een lijn. De functie `split` zonder parameters splitst namelijk op witruimte, niet enkel op spaties.

Oefening 5.2* Opsplitsen in echte woorden.
In oefening 5.2 heb je de zinnen opgesplitst in woorden met behulp van `split`. Dit is niet helemaal perfect, want in een zin zoals 'mijn computer is traag, denk ik', wordt het woord 'traag' niet herkent, omdat `split` er 'traag,' van maakt, met een komma. Helaas heeft Python geen ingebouwde functie die een zin opsplitst in woorden en daarbij ook leestekens negeert. Je zal die dus zelf moeten schrijven.
Schrijf dus een functie `splits(zin)` die een lijst teruggeeft van alle woorden in een zin. Een woord kan je herkennen als een opeenvolging van alfabetische tekens (a, b, c, …, maar ook A, ë, ç,…) zonder dat er andere tekens tussenkomen. Met `ch.isalpha()` kan je nagaan of een letterteken `ch` alfabetisch is. Test je programma uit op een aantal zinnen, bijvoorbeeld op

```
" U ziet   hier, met één   beetje 'geluk', enkele woorden !\n"
```

Als alles werkt, kan je de functie in je chatbot inlassen voor een nog beter resultaat!
De gemakkelijkste oplossing is wellicht om een nieuwe string te maken die dezelfde letters bevat als de oorspronkelijke zin, in dezelfde volgorde maar dan enkel wanneer dit een spatie of een alfabetisch letterteken is – en dat op dit resultaat uiteindelijk split toe te passen. Een stuk moeilijker, maar ook wel iets efficiënter, is het om de zin letter per letter te overlopen, bij te houden of je in een woord zit of niet, en op die manier opeenvolgende alfabetische letters te verzamelen tot opeenvolgende woorden.

6 Dobbelstenen

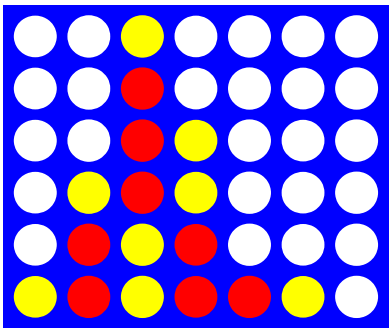


Oefening 6.1* Dominosteen
Een dominosteen heeft veel weg van 2 dobbelstenen die naast elkaar getekend zijn (maar een heel klein beetje overlappen). Schrijf een procedure `teken_domino(aantal_links, aantal_rechts)` die een dominosteen tekent met het opgegeven aantal ogen. Je kan knippen en plakken vermijden door eerst een procedure `teken_ogen` te schrijven die

enkel de ogen van (één kant van) een domino of van een dobbelsteen tekent. Baseer je op je oplossing van oefening 6.2 Vergeet de verticale streep niet in het midden.

Oefening 6.2* Vier op een rij

Teken een 'vier op een rij'-bord zoals in de prent hiernaast. Welke schijven er rood zijn en welke geel wordt aangegeven door een lijst van strings



```
schijven = [
    "1", "221", "112221", "2211", "2", "1", "" , ""
]
```

Let op, de inhoud van deze lijst komt overeen met de opeenvolgende kolommen, en niet met de rijen, telkens van onder naar boven. Speler 1 speelt met geel, speler 2 met rood. Je programma moet werken voor elke lijst van deze vorm. Je mag aannemen dat er altijd 7 strings in de lijst staan en dat elke string niet meer dan zes tekens bevat. De cirkels hebben een straal van 20 pixels en staan 10 pixels van elkaar en van de rand. (Vertrek van 2_*vier_op_een_rij.py*)

We stellen voor om kolom per kolom te werken en dan voor elke kolom eerst de gekleurde schijven te tekenen en dan pas de overblijvende cellen met witte cirkels te vullen voor de gaten. Je kan ook eerst een bord tekenen met 42 gaten, maar dan doe je overbodig werk (al programmeert dat wel wat gemakkelijker). Het kan wat uitzoeken vergen voor je de coördinaten en de afmetingen helemaal correct hebt – ze eerst uitrekenen met een schets op papier kan helpen. Merk op dat de rijnummers van onder naar boven lopen, terwijl de Y-as nog steeds van boven naar onder is gericht.

7 De wet van Benford

Oefening 7.1* Bevolkingsaangroei

Bij Statbel, het Belgische statistiekbureau, haal je allerhande cijfergegevens op over België. Wij hebben voor jou het bestand *bevolking.csv* gedownload met daarin het aantal inwoners per leeftijdscategorie voor de jaren 2014-2023. Teken met `matplotlib` een grafiek van deze gegevens die mooi aantoont dat de totale Belgische bevolking elk jaar stijgt. (Herken je de COVID-pandemie in jouw grafiek?)

Tip: gebruik de eerste lijn van het bestand om te weten hoeveel cijferkolommen er zijn. Dan werkt je programma ook nog wanneer er later kolommen voor 2024 en 2025 worden toegevoegd.

Je kan de URL https://bestat.statbel.fgov.be/bestat/crosstable.xhtml?view=1b9e219b-0387-4a70-880a-dc5eccaa244c gebruiken om de gegevens op te halen. Je moet op die pagina dan wel nog instellen welke rijen en kolommen je wil voor je de gegevens downloadt. De bedoeling van de oefening is een lijst bij te houden met de aantallen per jaar en die telkens aan te passen bij elke lijn die je inleest. Lees niet eerst het volledige bestand in om dan pas later de 'kolomwaarden' op te tellen.

Oefening 7.2* Vergrijzing van de bevolking.

Gebruik het bestand *bevolking.csv* uit oefening 7.1* om aan te tonen dat de Belgische bevolking vergrijst. Maak daarvoor een grafiek (met `matplotlib`) die toont hoe het percentage van Belgen ouder dan 60 jaar met de jaren stijgt. (Herken je de COVID-pandemie?) **Belangrijk:** je mag de lijnen van het bestand maar één keer overlopen.

Foreach-lus bij dictionaries

Python kent een foreach-lus waarmee je tegelijk alle sleutel/waardeparen van een dictionary overloopt.

```
for sleutel,waarde in dictionary.items():
    # doe iets met de sleutel en de waarde
```

(Let op de '`.items()`' die je op de dictionary moet toepassen.) Een gewone foreach-lus overloopt enkel

de sleutels.

```
for sleutel in dictionary:
    # doe iets met de sleutel
```

Oefening 7.3* Hamburgerfeest
In een Vlaamse provinciestad nodigt de burgemeester telkens op het eerste weekend van het nieuwe jaar iedereen van de stad uit op een groot gratis hamburgerfeest. Om alles in goede banen te leiden, wordt dit jaar aan de bewoners gevraagd om via een website in te schrijven en op te geven welke hamburger(s) ze verkiezen – rund, kip, vegetarisch, vegan, koosjer of halal. De keuzes worden door de webserver telkens achteraan toegevoegd aan het bestand *hamburgers.txt:*

```
rund
rund
halal
kip
vegan
rund
halal
vegetarisch
…
(± 5000 lijnen)
```

Schrijf een programma dat afdrukt hoeveel van elke optie er door de inwoners werd gekozen.

```
rund: 3140
kip: 1214
vegetarisch: 560
vegan: 224
koosjer: 298
halal: 415
```

(De uitvoerlijnen hoeven niet noodzakelijk in deze volgorde te staan.)
Je mag hierbij het bestand slechts één keer doorlopen. **Tip:** gebruik een dictionary als frequentietabel.

8 Sudoku

Oefening 8.1* Pas oefening 8.1 aan zodat je onderstaande resultaat krijgt. Schrijf daarvoor een afzonderlijke hulpprocedure `print_rij(rij)` die één enkele rij 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 |
+-------+-------+-------+
```

Kan je deze oefening programmeren zonder ook maar één if-opdracht te gebruiken?
In tegenstelling tot bij oefening 8.1 heb je hier wel rij- en kolomnummers nodig. Je kan deelbaarheid door 3 gebruiken om op het correcte moment een extra lijn of een extra verticale streep toe te voegen. Als alternatief – en dan zonder if-opdracht – kan je kortere, vernestelde lussen gebruiken: in de plaats van bijvoorbeeld de kolommen in één lus van 9 stappen te doorlopen, gebruik je twee vernestelde lussen, één die de drie blokken overloopt en daarbinnen één die de drie kolommen van dat blok overloopt – en analoog voor de rijen. (Dit geeft dus in principe een viervoudig vernestelde for-lus. We hebben de hulpprocedure ingevoerd om dit te vermijden en het geheel leesbaar te houden, zoals bij oefening 8.2.)

Oefening 8.2* Nog meer Sudoku's uit Sudoku's
In oefening 8.3 heb je 2 transformaties geprogrammeerd die uit een bestaande Sudoku een nieuwe Sudoku maken die nog steeds aan de regels voldoet. Hieronder tonen we nog drie andere transformaties met dezelfde eigenschap. Zo kan je de Sudoku *diagonaal* spiegelen of 90° draaien. Het derde voorbeeld krijg je door de Sudoku te *hernummeren*: door overal 1 door 9 te vervangen, 2 door 8, 3 door 7, 4 door 6, …, 8 door 2 en 9 door 1.

Diagonaal gespiegeld

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

90° gedraaid

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

Hernummerd

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

Breid jouw oplossing van oefening 8.3 uit met drie functies `spiegel_diagonaal(…)`, `roteer_90(…)` en `hernummer(…)` die een *nieuwe* Sudoku retourneren die gespiegeld/geroteerd/hernummerd is zoals hierboven. Druk ter controle het origineel en de 5 getransformeerde Sudoku's af.
Wat is het verband tussen 1 en 9, 2 en 8, 3 en 7, …? (En hoe kan je dit in je programma gebruiken?)
Hun som is telkens 10. Hernummeren zet dus elk element *e* om naar 10-*e*. Je kan hiervoor ook een lijst `[0,9,8,…,1]` gebruiken met daarin de 'vertalingen' van de elementen – maar een meervoudige selectie met 10 gevallen is niet toegelaten! Roteren kan je doen door twee spiegelingen te combineren – eerst diagonaal en dan verticaal. Maar het is ook een goede oefening om dit te programmeren zonder hiervan gebruik te maken.

Oefening 8.3* Sudoku's met lege plekken.
In dit hoofdstuk hadden we het over Sudoku's die volledig zijn ingevuld. Veel van wat we gedaan hebben werkt ook voor Sudoku's waarin hokjes zijn leeg gelaten. Voor een leeg hokje plaatsen we dan een 0 op de corresponderende plaats in de 2-dimensionale tabel waarmee de Sudoku wordt voorgesteld.
Bekijk de broncode van al je oplossingen voor de oefeningen uit dit hoofdstuk. Op welke plaatsen moet je iets veranderen (en wat?) opdat jouw programma's ook zouden werken als er lege plaatsen zijn in de Sudoku's die je gebruikt? (Je hoeft je programma's niet aan te passen.)
Er zijn in essentie slechts twee aanpassingen nodig:

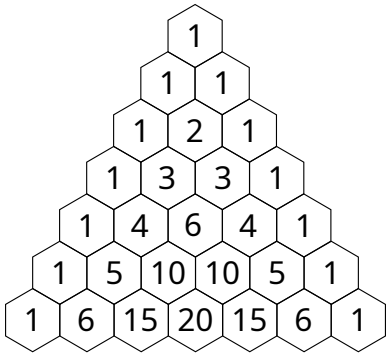
- Bij het afdrukken van de Sudoku's moet je een spatie afdrukken in plaats van een 0. Dat vraagt een bijkomende if-opdracht.
- Bij het controleren van de frequentietabel moet je misschien iets soepeler zijn. Het volstaat bijvoorbeeld niet langer om het aantal 0-en (resp. 1-en) te tellen met `count` en te kijken of dit 1 (resp. 9) is. Schrijf een korte lus die kijkt of de frequentietabel geen getallen bevat die groter zijn dan 1.

Oefening 8.4* Afbeelding van een Sudoku.
Maak een afbeelding van een gegeven Sudoku met behulp van PIL (hoofdstuk 6). Gebruik dikkere lijnen om de onderverdeling in 3 × 3 blokken te verduidelijken. Zoek in de online documentatie van PIL hoe je tekst op een afbeelding plaatst.

9 In-place

Oefening 9.1* Een lijst omkeren in-place.
Schrijf een procedure `reverse(lijst)` die een lijst *in-place* omkeert: alle elementen zijn dezelfde als in de oorspronkelijke lijst, maar hun volgorde is omgekeerd. Belangrijk! Maak geen kopie van de lijst – dus ook niet `lijst[::-1]`.

Oefening 9.2* De driehoek van Pascal.
De driehoek van Pascal, waarvan de eerste 7 rijen hieronder links zijn afgebeeld, is gemakkelijk op te bouwen: elke cel bevat de som van de waarden in de twee cellen erboven, en de randcellen bevatten allemaal enen. (De getallen in de driehoek hebben ook een wiskundige betekenis – het zijn binomiaalcoëfficiënten en aantallen combinaties – maar voor deze oefening is dit niet belangrijk.)



1	0	0	0	0	0	0
1	1	0	0	0	0	0
1	2	1	0	0	0	0
1	3	3	1	0	0	0
1	4	6	4	1	0	0
1	5	10	10	5	1	0
1	6	15	20	15	6	1

Schrijf een procedure `pascal(n)` die de *n*-de rij teruggeeft uit de driehoek van Pascal als een lijst van *n*+1 getallen. Voor *n*=6 geeft je dus [1,6,15,20,15,6,1] terug. Doe dit *in-place*: vertrek van een lijst met *n*+1 nullen en bouw die gradueel op zoals in de rechter afbeelding. Opgelet, de 7 rijen die je daar ziet, stellen allemaal *dezelfde* lijst voor!

Oefening 9.3* Selectiesortering van achter naar voor.
Schrijf een procedure `sorteer` die een gegeven lijst van getallen sorteert met de volgende variant van de selectiesortering:

- Zoek het grootste element in de lijst en verwissel dit met het element achteraan de lijst.
- Zoek nu het grootste element in het deel van de lijst dat alle elementen bevat behalve het laatste. Verwissel dit met het voorlaatste element in de lijst.
- Herhaal dit door telkens het grootste element te zoeken in steeds kortere gedeelten van de lijst en dat dan achteraan dit korte deel te plaatsen.

M.a.w., in plaats van de gesorteerde lijst zoals in §9.3 van voor naar achter op te bouwen, doe je dit van achter naar voor – en telkens door het grootste element te zoeken in plaats van het kleinste.
Test jouw implementatie op dezelfde manier als in oefeningen 9.3 en 9.4.

Oefening 9.4* Tuinkaboutersortering.
Wanneer een tuinkabouter een rij bloempotten van klein naar groot ordent, doet hij dat zo:

- Staan de pot vóór hem en de pot links ervan in de juiste volgorde, dan gaat hij één pot naar rechts,
- anders wisselt hij beide potten om en gaat één pot naar links.

Er zijn twee speciale gevallen:

- Staat hij helemaal links, dan gaat hij gewoon één pot naar rechts.
- Stapt hij voorbij de laatste pot aan de rechterkant, dan is hij klaar.

Hij begint helemaal links.
Schrijf een procedure `sorteer` die een lijst van getallen van klein naar groot ordent volgens deze tuinkaboutermethode. Test jouw implementatie op dezelfde manier als in oefeningen 9.3 en 9.4.

10 Verkeer

(Voorlopig nog) geen uitbreidingsoefeningen.

11 Woordladder

Oefening 11.1* Controle van start- en eindwoorden.
Bij de oefeningen uit hoofdstuk 11 zou je eigenlijk moeten controleren of de start- en eindwoorden wel vijf letters hebben en geldige Nederlandse woorden zijn. Voeg deze controle toe aan je oplossingen. Blijf nieu-

we woorden vragen totdat ze geldig zijn.

Dit kan met een zeer eenvoudige while-lus:

```
start_woord = input("Met welk woord wil je starten? ")
while not is_Nederlands(start_woord):
    print("Dit is geen Nederlands woord van 5 letters!")
    start_woord("Geef een ander woord: ")
```

Merk op dat het niet nodig is om afzonderlijk te testen of het woord wel lengte 5 heeft.

Oefening 11.2* Maak zelf het bestand woorden5letters.txt.
Het bestand *allewoorden.txt* bevat een lijst met alle Nederlandse woorden. Overloop dit bestand lijn per lijn en schrijf alle woorden van precies vijf letters uit naar het bestand *woorden5letters.txt*. Gebruik enkel de 'woorden' die uit kleine letters bestaan – geen hoofdletters, geen aanhalingstekens, geen spaties, …. Let er bij het inlezen op dat je het *new line*-teken op het einde van de lijn telkens verwijdert. Je mag in Python with-opdrachten gerust vernestelen:

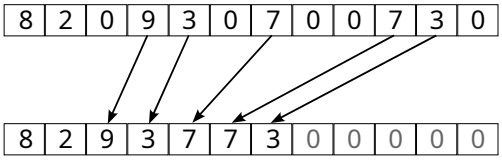
```
with open ("allewoorden.txt", "r") as bestand_in:
    with open ("woorden5letters.txt", "w") as bestand_out:
        ???
```

Je hebt geluk: het oorspronkelijke bestand is reeds alfabetisch gesorteerd, dus het resultaat zal dat ook zijn, en dus bruikbaar voor de oefeningen uit hoofdstuk 11.

12 Het Oekraïense vlagprobleem

Oefening 12.1* Met in-place sneller dan zonder.
Toon aan, met een grafiek, dat Dijkstra's in-place algoritme uit oefening 12.3 ook sneller is dan het algoritme uit oefening 12.1 dat niet in-place is. Let op, het verschil wordt pas merkbaar bij lijsten die een heel stuk langer zijn dan die uit oefening 12.4.

Oefening 12.2* Nullen achteraan.
Schrijf een procedure die alle nullen in een lijst naar achteren verplaatst. Doe dit *in-place* en zorg er tevens voor dat de getallen verschillend van 0 in *dezelfde volgorde* blijven bestaan. (Je kan dus niet zomaar de oplossingen van het Oekraïense vlagprobleem overnemen.) Tip: gebruik twee indices, een *bron* en een *bestemming* en kopieer telkens van bron naar bestemming, zoals in de figuur hieronder, maar dan binnen dezelfde lijst. Hoewel de bron telkens met 1 verhoogd wordt, geldt dat niet steeds voor de bestemming.



Merk op dat de nullen niet hoeven verplaatst of gekopieerd te worden. Je kan gewoon op het einde de rest van de lijst terug met (nieuwe) nullen opvullen. Je zou in plaats van een bron-index ook een foreach-lus kunnen gebruiken, maar dat is af te raden: een lijst die je met een foreach-lus doorloopt, kan je tijdens die lus best niet veranderen.

13 Recursie

Oefening 13.1* Torens van Hanoi
Vervolledig de code uit §13.2 zodat `hanoi(3,"A","B","C")` het volgende afdrukt:

```
Verplaats een schijf van A naar C
Verplaats een schijf van A naar B
Verplaats een schijf van C naar B
Verplaats een schijf van A naar C
Verplaats een schijf van B naar A
```

```
Verplaats een schijf van B naar C
Verplaats een schijf van A naar C
```

Oefening 13.2* Hanoi met meer details
Pas je oplossing aan zodat het volgende wordt afgedrukt:

```
Verplaats schijf 1 van A naar C
Verplaats schijf 2 van A naar B
Verplaats schijf 1 van C naar B
Verplaats schijf 3 van A naar C
Verplaats schijf 1 van B naar A
Verplaats schijf 2 van B naar C
Verplaats schijf 1 van A naar C
```

Oefening 13.3* Hanoi met ander basisgeval
Pas je oplossing aan zodat je in de recursie n=0 als basisgeval gebruikt in plaats van n=1.

Oefening 13.4* Patroon met enkel zwarte vierkanten
In oefening 11.5 worden er heel veel kleine rode en zwarte vierkantjes getekend. Het spaart ongeveer de helft van het werk uit als we één groot rood vierkant zouden tekenen en daarbovenop enkel nog de zwarte vierkantjes. Pas je oplossing zodanig aan dat er enkel nog zwarte vierkantjes getekend worden. Laat daarbij de kleurparameter weg uit `teken_vierkant` en vervang de twee kleurenparameters in `teken_patroon` door één Booleaanse parameter `omgewisseld`.

14 Oppervlakte

Oefening 14.1* Regel van Simpson
Bereken de oppervlakte onder de grafiek van de functie $f(x)=8-x^2$ voor het interval [1.0,2.0] met behulp van de *regel van Simpson*. Je vindt meer uitleg over die methode op het Internet.
Bijvoorbeeld bij https://www.wiskundemagie.be/de-regel-van-simpson/. Omdat het om een kwadratische functie gaat, moet de regel van Simpson hier meteen het exacte antwoord geven, onafhankelijk van in hoeveel delen het interval wordt verdeeld.