1 Algoritmen en programmeren

Dit document bevat de nota's bij modules 2 en 3 van de nascholingen rond de specifieke minimumdoelen *Algoritmen en programmeren*. Die nascholingen worden een eerste keer gegeven in 2023-2024 aan Universiteit Gent.

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.
- Python kent enkel *lijsten* en geen *arrays*. De beperking dat arrays een vaste lengte hebben, biedt

[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.

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 ontwikkeld 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 ?), 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 ?), is een andere gegevensstructuur dan een *frequentietabel* (zie ?) - 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*).
In deze tekst beperken we ons voornamelijk tot het beschrijven van de meest voorkomende algoritmische

[3] En helaas kan deze onwetendheid je programma minder efficiënt maken dan het zou kunnen zijn. Zie uitbreidingsoefening 101.

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, nu eens bij de ene index, dan eens bij een andere, en dat 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 de nascholingen en van deze tekst

De leerinhoud die we in de nascholingen behandelen, komt grosso modo overeen met met het volgende aantal lesuren dat je op school nodig hebt om de leerstof te behandelen:

Module 1	**Module 2**	**Module 3**
7 lesuren	14 lesuren	7 lesuren

Modules 1+2 zijn bedoeld voor scholen waar voor informaticawetenschappen slechts 1 lesuur per week wordt voorzien, gedurende 1 jaar. Module 3 kan gebruikt worden in scholen die meer lesuren aan informaticawetenschappen hebben toebedeeld, of waar al in de 2e graad programmeren (in Python) wordt aangeleerd. In deze nota's behandelen we modules 2 en 3 – en verwachten we dat de lezer beschikt over een ze-

kere basiskennis van de programmeertaal Python, bijvoorbeeld zoals die in Module 1 is gegeven.
De leerinhoud is onderverdeeld in verschillende hoofdstukken die elk corresponderen met één of twee lesuren. Er is bovendien een afzonderlijk document met een aantal uitbreidingsoefeningen bij de verschillende hoofdstukken. Je kan die gebruiken als differentiatie of wanneer je net iets meer dan de 14 (of 21) lesuren ter beschikking hebt dan we hier hebben begroot.
Bij opgaven en oefeningen, of bij vragen die we in deze tekst formuleren, plaatsen we vaak enkele aandachtspunten in dit kleur en lettertype. Dit is informatie die je in de klas kan gebruiken bij het bespreken van de antwoorden en de oplossingen van de leerlingen.
Latere hoofdstukken maken vaak gebruik van concepten die in eerdere hoofdstukken zijn aangebracht. Het is dus best om in de lessen dezelfde volgorde aan te houden, zeker in het begin. Hoewel de latere hoofdstukken eigenlijk bij module 3 horen, is het wel mogelijk om daaruit een onderwerp te pikken dat je dan toch in de eerste 14 lesuren opneemt. 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. Het totale aantal lesuren waarvoor we inhoud hebben voorzien, iets uiteindelijk iets groter geworden dan 21. Het staat je vrij om in je eigen lessen (delen van) hoofdstukken weg te laten. We duiden hier en daar aan wat we essentieel vinden en wat gemakkelijker kan weggelaten worden.
De oefeningen hebben een ongebruikelijke nummering, het hoofdstuknummer gevolgd door 2 cijfers. Er zitten ook ‘gaten’ in de opeenvolging van nummers. Zo heeft hoofdstuk 2 bijvoorbeeld oefeningen 201, 202, 205, 208, 209 en uitbreidingsoefeningen 251, 252 en 256. Dit maakt het gemakkelijker om achteraf oefeningen toe te voegen zonder de bijbehorende bestandsnamen telkens te moeten wijzigen.
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 van de nascholingen (voorlopig nog verborgen). Daar staan ook de oplossingen van de meeste oefeningen. Als we dus schrijven ‘vertrek van bestand *412_lotr.py*’, dan doelen we op een bestand dat je vanaf die website kunt downloaden.
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, numpy
```

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. Een gebruiksvriendelijke debugger is wel een pluspunt.

1.11 Licentie