PROGRAMMEREN 2

TOPIC 11: FUNDAMENTEN VAN INFORMATICA

Download PDF-versie

UCLL HOGESCHOOL

OVERZICHT

- Inleiding
- Onberekenbare functies
- Turing machines
- Universele programmeertalen
- Halting problem
- Compexiteitstheorie
- Andere domeinen

INLEIDING

INLEIDING

- We werpen in deze les een blik op de wiskundige grondslagen van de informatica
- Wiskundig model voor berekenen met een computer: Turing machine
- Berekenbaarheid en onberekenbaarheid
- Complexiteit van algoritmen en problemen
- Andere relevante domeinen in wiskunde en in "theoretische informatica"

Je kan een heel vak vullen met leerstof hierrond. In dit topic proberen we ons te beperken tot de belangrijkste zaken.

De leerstof in dit topic zal minder snel aan bod komen in een 'gewone' les in het lager middelbaar. Als toekomstige leraar informatica wordt echter verwacht dat je achtergrondkennis bezit over je vak en de grondslagen ervan begrijpt.

(ON)BEREKENBARE FUNCTIES

PROBLEMEN OPLOSSEN

Een fundamenteel doel van informatica is het **oplossen van problemen**. Een probleem is een heel ruim begrip in deze context. Een oplossing voor een probleem is een algoritme.

Sommige problemen zijn zo lastig zijn dat zelfs de meest krachtige computers ze niet binnen een “redelijke” tijd kunnen oplossen. En er zijn zelfs problemen waarvoor we kunnen bewijzen dat er **geen algoritme kan bestaan** om ze op te lossen!

FUNCTIES

In de theoretische informatica is een functie een **verband** tussen bepaalde invoer- en uitvoerwaarden.

Merk op dat dit veel dichter aansluit bij de definitie van een functie in wiskunde. Een functie in de moderne informatica is een herbruikbaar stuk code dat iets doet en/of iets teruggeeft.

Het **berekenen van een functie** houdt in dat we voor een gegeven invoer de uitvoer bepalen.

Sommige functies zijn zo complex dat functiewaarden niet algoritmisch kunnen berekend worden: we noemen ze **onberekenbaar (noncomputable)**; andere daarentegen heten **berekenbaar (computable)**.

Geen enkele machine die algoritmes uitvoert (computer) zal ooit een onberekenbare functie kunnen uitrekenen.

DENKBEELDIGE COMPUTERS

TURING MACHINES

TURING MACHINES: CONTEXT

Beeld je in:

- Je bent een wiskundige in het begin van de 20e eeuw
- Je zoekt een manier om te bepalen of functies (on)berekenbaar zijn
- De samenleving is sterk geïndustrialiseerd; overal zijn er fysieke machines die (fysieke) invoer transformeren naar (fysieke) uitvoer
- Kan je het idee van machines ook gebruiken in de context van berekenbaarheid?

Een beetje terminologie:

- Een **alfabet** is een set van symbolen waarmee invoer en uitvoer opgebouwd kan worden. In binair rekenen bestaat het alfabet uit twee symbolen (0 en 1). Een alfabet kan ook uit meerdere cijfers, letters, speciale tekens, ... bestaan.

TURING MACHINES

Een Turing machine T is een **denkbeeldige** machine:

- Bedacht door Alan Turing in 1936
- Bestaat (in gedachten) uit:
  - een **oneindig** lange band, verdeeld in vakjes
  - elk vakje bevat precies 1 symbool uit het **alfabet** van T
  - een lees/schrijfkop kan het symbool in het onderliggende vakje **lezen** of er één **schrijven**
- Bevindt zich steeds in 1 van een (eindig) aantal mogelijke **toestanden**
- Start haar berekening in haar starttoestand en stopt wanneer ze de stoptoestand bereikt heeft

Een Turing machine kan invoer transformeren naar uitvoer, net zoals echte machines. Maar in tegenstelling tot echte machines, transformeert het (denkbeeldige) invoer naar (denkbeeldige) uitvoer.

Een Turing machine kan iets transformeren in een aantal stappen. In elke stap:

- wordt het symbool in het onderliggend vakje gelezen
- wordt daar een symbool geschreven
- kan de lees/schrijfkop 1 vakje naar links of naar rechts bewegen
- wordt overgegaan naar een welbepaalde andere toestand

Een Turing machine wordt aangestuurd door een **programma**. Een programma houdt rekening met de huidige toestand en het gelezen symbool in stap 1. Vergelijk het met een arbeider die een machine bestuurt op basis van hun zintuigen en een handleiding.

VOORBEELD

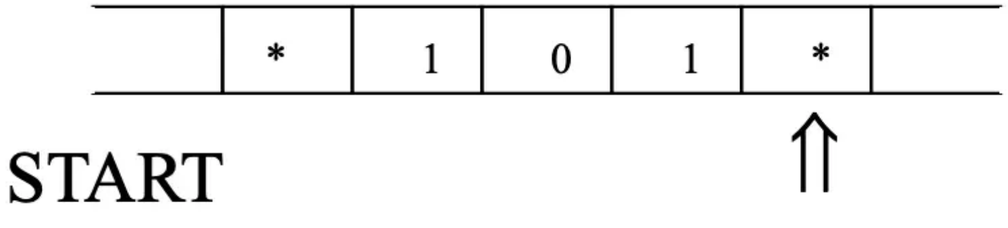
Beeld je in dat je een machine wilt maken die bij een binair getal +1 optelt.

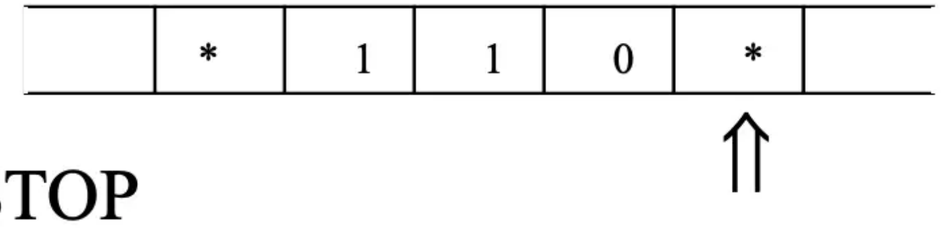
- Alfabet: 0, 1, * (speciaal karakter voor start/stop)
- Invoer: binair getal
- Uitvoer: binair getal
- Handleiding:

huidige toestand	inhoud voorliggend vak	te schrijven symbool	beweegrichting lees/schrijfkop	toestand na uitvoering
START	*	*	Links	TEL OP
TEL OP	0	1	Rechts	KEER TERUG
TEL OP	1	0	Links	DRAAG OVER
TEL OP	*	*	Rechts	STOP
DRAAG OVER	0	1	Rechts	KEER TERUG
DRAAG OVER	1	0	Links	DRAAG OVER
DRAAG OVER	*	1	Links	OVERLOOP
OVERLOOP	*	*	Rechts	KEER TERUG
KEER TERUG	0	0	Rechts	KEER TERUG
KEER TERUG	1	1	Rechts	KEER TERUG
KEER TERUG	*	*	geen beweging	STOP

VOORBEELD

Laat je deze machine draaien op de situatie links, dan zal de band op het einde er uit zien zoals rechts getoond:





Opdracht: gebruik de handleiding op vorige slide om, vertrekkende van de situatie links, de stappen van het programma één voor één uit te voeren. Controleer dat je inderdaad dezelfde uitkomst krijgt.

TURING MACHINES EN FUNCTIES

Een Turing machine transformeert invoer naar uitvoer. Men zegt dat de machine waarden **berekent**.

Turing machines en functies hebben wel erg veel gelijkenissen. De hypothese van Church-Turing formaliseert dit:

> *"Een functie is berekenbaar als de waarden berekend kunnen worden op basis van een mechanisch proces (zoals Turing machines)."*
>
> — Church and Turing

Met andere woorden: de mogelijkheden van een Turing machine (lezen, schrijven, bewegen, ...) zijn voldoende om (bijna) alle gangbare algoritmes uit te voeren. Meer heb je in principe niet nodig! **Turing machines zijn daarom een maatstaf voor expressiekracht.**

TER INFO

Je moet in dit vak niet zelf programma's kunnen schrijven voor Turing machines. Je moet ze enkel kunnen lezen en begrijpen.

De hypothese van Church-Turing is slechts een hypothese. Ze kan niet bewezen worden. Toch worden de hypothese en Turing machines algemeen aanvaard als fundamentele concepten in de informatica.

Er zit een hele geschiedenis achter de hypothese van Church-Turing, maar deze hoef je niet te kennen voor dit vak. Als je er meer over wilt weten, bekijk dan bv. The Church-Turing Thesis Explained.

TURING MACHINES TOEGEPAST

UNIVERSELE PROGRAMMEERTALEN

TURING COMPLEETHEID

Turing machines zijn de norm voor expressiekracht. Je kan er bijna alle gangbare algoritmes mee uitvoeren en dus problemen mee oplossen.

Hoe verhoudt dit zich tot:

- De binaire taal van moderne computers? (Ook wel machinetaal genoemd)
- Een programmeertaal zoals Python, Java, COBOL, C++, ...?
- Een visuele taal zoals Scratch?
- HTML en Markdown, twee talen om documenten mee te schrijven?
- Software zoals Microsoft Office, videogames, ...
- ...

Men noemt een systeem (een computer, een taal, software, ...) **Turing compleet** als het dezelfde expressiekracht heeft als een Turing machine. Specifiek voor programmeertalen zegt men dat ze **universeel** zijn als ze Turing compleet zijn.

Het doel in deze sectie is om na te gaan voor een aantal systemen of ze (niet) Turing compleet zijn.

TURING COMPLEETHEID BEWIJZEN

Van vrijwel elke programmeertaal wordt verwacht dat ze Turing compleet is. Enkel dan kunnen we ze gebruiken voor eender welk algoritme te schrijven. Maar hoe bewijs je dit voor een taal?

Voorbeeld: de nieuwe programmeertaal Bob:

- variabelen met als waarden bitpatronen (dus slechts één type toegelaten, geen string/boolean/...)
- 3 toekenningsopdrachten:
  - clear varnaam; (allemaal 0-en in var)
  - incr varnaam; (binaire waarde van var met 1 verhogen)
  - decr varnaam; (binaire waarde van var met 1 verlagen, 0 blijft 0)
- 1 herhalingsopdracht: `while varnaam not 0 do ... end;`

TURING COMPLEETHEID BEWIJZEN

Kan je met deze regels bijvoorbeeld een programma maken om een waarde van X naar Y te kopiëren?

TURING COMPLEETHEID BEWIJZEN

Kan je met deze regels bijvoorbeeld een programma maken om een waarde van X naar Y te kopiëren?

JA!

```
clear Y;
clear Hulp;
while X not 0 do
  incr Hulp;
  decr X;
end;
while Hulp not 0 do
  incr X;
  incr Y;
  decr Hulp;
end;
```

TURING COMPLEETHEID BEWIJZEN

Dit is nog geen 100% garantie, maar dit is een klassieke eerste test voor de compleetheid van talen na te gaan.

Een volledig correct bewijs houdt in dat je in je systeem (in dit geval een programmeertaal) een programma schrijft dat Turing machines simuleert. Als je taal eender welke Turing machine kan simuleren, dan kan het ook elk probleem oplossen dat een Turing machine kan oplossen.

Enkele voorbeelden van implementaties van Turing machines:

- Python
- Haskell
- Bob: kan je zelf proberen (niet eenvoudig)

MACHINETAAL

Machinetalen stellen instructies en data voor als 0'en en 1'en. Een lijn van 32 symbolen (0 en 1) komt bv. overeen met één instructie om 1 + 1 te doen. Ze delen een aantal concepten met hogere programmeertalen, maar hebben ook hun eigen speciale regeltjes.

Omdat machinetalen zo moeilijk te lezen zijn, bestaan er ook **assemblytalen**. Ze zijn 'iets leesbaarder' dan machinetalen, maar zijn nog steeds heel low level.

Machinetalen zijn Turing compleet, aangezien alle hogere programmeertalen hier op moeten steunen. Assemblytalen zijn een bijna letterlijke vertaling van machinetalen, dus zijn ook Turing compleet.

In dit vak zien we geen machinetalen of assemblytalen in detail. Toch geïnteresseerd? Bekijk bv. Assembly Language in 100 Seconds.

PROGRAMMEERTALEN

Voorbeelden uit vorige lessen/vakken:

- Python: imperatief + objectgericht
- Java: enkel objectgericht
- Rust: objectgericht, maar heeft geen overerving
- Haskell: declaratief, heeft geen lussen of aanpasbare variabelen
- Scratch: blokgebaseerd, deels objectgericht, kan niet met files werken aangezien het in een browser draait

Al deze talen zijn Turing compleet. Wat kan je hieruit besluiten over programmeerconcepten?

PROGRAMMEERTALEN

Voorbeelden uit vorige lessen/vakken:

- Python: imperatief + objectgericht
- Java: enkel objectgericht
- Rust: objectgericht, maar heeft geen overerving
- Haskell: declaratief, heeft geen lussen of aanpasbare variabelen
- Scratch: blokgebaseerd, deels objectgericht, kan niet met files werken aangezien het in een browser draait

Al deze talen zijn Turing compleet. Wat kan je hieruit besluiten over programmeerconcepten?

Veel programmeerconcepten zijn niet noodzakelijk om Turing compleetheid te bereiken in een taal. Ze bestaan omwille van andere redenen: gebruiksgemak, uitbreidbaarheid, leesbaarheid, ...

HTML EN MARKDOWN

Open de developer tools in je browser (F12 in Chrome). HTML is de `<>...</>`-structuur die de inhoud van een website beschrijft.

HTML is niet Turing compleet. Het bezit niet de vereiste kwaliteiten om dezelfde expressiekracht als een Turing machine te behalen. Er zijn geen variabelen, functies, if-else, ... Dit was ook nooit een doel voor de ontwikkelaars van HTML. Het is gewoon een taal om inhoud op een gestructureerde manier te beschrijven.

Hetzelfde geldt voor Markdown, een andere taal voor het gestructureerd beschrijven van inhoud.

SOFTWARE

Software, geschreven in een programmeertaal, kan zelf ook Turing compleet zijn.
Een paar voorbeelden:

- Microsoft Excel is Turing compleet sinds 2021
- Microsoft Word is (naar mijn weten) niet Turing compleet
- En Microsoft PowerPoint?
- De *Sid Meier's Civilization* reeks van videospellen:
  - Minstens 3 ervan zijn Turing compleet (Link naar paper)
- Het videospel *Factorio* is Turing compleet: je kan een computer bouwen binnenin het videospel

SOFTWARE

Software, geschreven in een programmeertaal, kan zelf ook Turing compleet zijn.
Een paar voorbeelden:

- Microsoft Excel is Turing compleet sinds 2021
- Microsoft Word is (naar mijn weten) niet Turing compleet
- En Microsoft PowerPoint?
  - Blijkbaar wel
  - Een humoristische video die kort uitlegt hoe het kan
  - Link naar de paper
- De *Sid Meier's Civilization* reeks van videospellen:
  - Minstens 3 ervan zijn Turing compleet (Link naar paper)
- Het videospel *Factorio* is Turing compleet: je kan een computer bouwen binnenin het videospel

BEPERKING

Echte systemen kunnen nooit volledig Turing compleet zijn. Turing machines hebben namelijk een bepaald kenmerk dat we nooit kunnen simuleren. Welk kenmerk?

BEPERKING

Echte systemen kunnen nooit volledig Turing compleet zijn. Turing machines hebben namelijk een bepaald kenmerk dat we nooit kunnen simuleren. Welk kenmerk?

Oplossing: Turing machines hebben **oneindig veel geheugen**, dit kunnen we nooit simuleren. Daarom zijn echte systemen nooit volledig Turing compleet. In de praktijk is dit zelden een probleem, daarom wordt het soms genegeerd bij de definitie van Turing compleetheid.

EEN ONBEREKENBARE FUNCTIE

HALTING PROBLEM

HALTING PROBLEM

Het stopprobleem (halting problem) gaat als volgt: gegeven een programma, bestaat er een functie die bepaalt of het programma ooit zal stoppen of niet?

Als dit kan, dan bestaat er een Turing machine voor die het probleem oplost.

- Invoer: een programma, voorgesteld als een reeks bits (0'en en 1'en)
- Uitvoer: 1 als het programma stopt, 0 als het nooit stopt

Je kan eender welk programma eenvoudig transformeren naar bits:

- Transformeer elk symbool naar zijn decimale voorstelling in ASCII of Unicode
- Transformeer de decimale voorstelling naar een binaire voorstelling

HALTING PROBLEM

Het halting problem is onberekenbaar: er bestaat geen functie voor, en dus ook geen Turing machine om het uit te voeren

Bewijs: zie bv. Halting problem. Het bewijs is geen leerstof voor dit vak.

Meer algemeen is er een hele groep van **onoplosbare problemen**. Voor deze problemen kunnen we geen Turing machines bouwen. Dit toont aan dat er grenzen zijn aan wat we kunnen doen met Turing machines - en afgeleiden ervan zoals computers en programmeertalen.

COMPLEXITEIT VAN PROBLEMEN

COMPLEXITEIT

We weten al het volgende:

1. Sommige problemen zijn oplosbaar: er bestaat een algoritme/Turing machine voor
2. Sommige problemen zijn onoplosbaar: de bijhorende functie is onberekenbaar, er bestaat geen Turing machine voor

Categorie 2 is theoretisch interessant, maar weinig helpvol in de praktijk.
Categorie 1 kunnen we wel nog verder opsplitsen op basis van complexiteit van het probleem. Dit wordt onderzocht in complexiteitstheorie.

COMPLEXITEITSTHEORIE

- De complexiteit van een probleem wordt gemeten door de complexiteit van zijn (meest efficiënte) oplossing(en), uitgedrukt als een algoritme
- De (tijds)complexiteit van een algoritme wordt bepaald door het aantal stappen dat een computer nodig heeft om het uit te voeren (en dus de benodigde uitvoeringstijd). Dit is niet noodzakelijk hetzelfde als de “ingewikkeldheid” van de programmacode voor mensen
- Als het meest efficiënte (bekende) algoritme voor een probleem een tijdscomplexiteit heeft van $\Theta(f(n))$, dan zit het probleem in klasse $O(f(n))$
  - bv: sorteren zit in $O(n*log2(n))$

CATEGORIEËN VAN COMPLEXITEIT

Problemen kunnen in verschillende categorieën ingedeeld worden op basis van hun complexiteit. We beperken ons tot twee belangrijke:

- De polynomiale groep P bevat alle problemen waarvan $f(n)$ in $O(f(n))$ een **veelterm** is
- Alle niet-polynomiale problemen vallen buiten P. De $f(n)$ is geen veelterm. Voorbeelden zijn:
  - $O(2^n)$
  - $O(n!)$
  - ...

Problemen in P kan je 'efficiënt' oplossen met een computer. Andere problemen zijn complexer. Een speciale subcategorie is NP. Dit staat voor *niet-deterministische polynome tijd*.

NP: BIJNA P?

Problemen in NP hebben de eigenschap dat ze een 'toevallige' stap bevatten. Meer specifiek:

- Om een **nieuwe** oplossing te vinden, moet je alle opties nagaan. Deze opties vragen niet-polynome tijd
- Om een **bestaande** oplossing te **verifiëren**, moet je andere stappen doen die slechts polynome tijd vragen
- Als je 'toevallig' een oplossing hebt, worden de problemen dus P in plaats van NP!

NP: BIJNA P?

Problemen in NP hebben de eigenschap dat ze een 'toevallige' stap bevatten. Meer specifiek:

- Om een **nieuwe** oplossing te vinden, moet je alle opties nagaan. Deze opties vragen niet-polynome tijd
- Om een **bestaande** oplossing te **verifiëren**, moet je andere stappen doen die slechts polynome tijd vragen
- Als je 'toevallig' een oplossing hebt, worden de problemen dus P in plaats van NP!

Een concreet voorbeeld: een oplossing vinden voor sudoku's

- Alle algoritmes om oplossingen te **genereren** vragen niet-polynome tijd. Dit is wat het een uitdagende puzzel maakt voor mensen
- Een oplossing **verifiëren** kan gebeuren in polynome tijd. Ook mensen kunnen dit relatief snel doen
- *Sudoku's oplossen* is dus een probleem dat behoort tot NP

$P = NP$?

Het P versus NP problem is een bekend, onopgelost vraagstuk in informatica. Er is nog geen bewijs voor of tegen.

De cruciale vraag is: **is elk probleem waar je de oplossing in polynome tijd kan verifëren, ook oplosbaar in polynome tijd?**

Er is een hele categorie van problemen die *NP-compleet* zijn. Als we ooit kunnen aan tonen dat één van deze problemen oplosbaar is in P, dan geldt dit ook voor de rest in die categorie. Daarmee zou bewezen kunnen worden dat $P = NP$.

Aan het bewijs hangt een prijs vast van $1.000.000 en eeuwige roem (in de community van informatici).

TER INFO

Om de complexiteit van een algoritme te bepalen, gebruik je limieten uit de wiskunde. Dit wordt hier niet verder behandeld.

Moest ooit bewezen worden dat $P = NP$, dan kan dit serieuze gevolgen hebben voor alle veiligheidssystemen wereldwijd. Alle moderne encryptie zou dan in één klap nutteloos worden, want ze steunen op de assumptie dat $P != NP$.

OM AF TE SLUITEN

ANDERE DOMEINEN

We hebben in dit topic maar een paar onderwerpen behandeld. Er zijn er nog veel meer, zoals:

- Grafentheorie
  - Algoritmes voorstellen
  - Bomen als datastructuren
  - (Tegenwoordig ook deel van wiskunde in het middelbaar)
- Automatentheorie en formele talen
  - Fundamenten van moderne programmeertalen
- Algebra voor cryptografie
- Logica
  - Correctheidsbewijzen van programma's
  - Logisch programmeren
- Lambda calculus
  - Fundament voor functioneel programmeren

BOEKEN

Over elk domein bestaan verschillende boeken. De meesten zijn erg wiskundig.

Aanrader van de vorige docent:

- Douglas R. Hofstadter, Ronald Jonkers (vert.), Gödel, Escher, Bach: een eeuwige gouden band, Contact, 1985 (tegenwoordig: Atlas Contact, ??de druk juli 2021)
- of het Engelstalige origineel: Douglas R. Hofstadter, Gödel, Escher, Bach: an Eternal Golden Braid, Basic Books, 1979 (nieuwe editie bij de 20ste “verjaardag” in 1999)
- Minder wiskundig, meer bedoeld als toegankelijke introductie tot de domeinen
- Bevat ook links met artificiële intelligentie