PROGRAMMEREN 2

TOPIC 13: COMPLEXITEIT VAN ALGORITMES

Download PDF-versie

UCLL HOGESCHOOL

OVERZICHT

- Probleemstelling + domeinen
- Regels rond complexiteit bepalen
- Complexiteit visueel voorgesteld
- Complexiteit van enkele algoritmes

PROBLEEMSTELLING

PROBLEEMSTELLING

Anders dan wiskundigen zijn informatici geïnteresseerd in het **vergelijken** van manieren om iets op te lossen.

Welke concepten van computationeel denken zijn hier aan gelinkt?

Kan je voorbeelden geven uit Programmeren 1/Programmeren 2/elders waar er meerdere oplossingen waren voor hetzelfde probleem?

PROBLEEMSTELLING

Anders dan wiskundigen zijn informatici geïnteresseerd in het **vergelijken** van manieren om iets op te lossen.

Welke concepten van computationeel denken zijn hier aan gelinkt?

Algoritmisch denken en **Evaluatie**.

Kan je voorbeelden geven uit Programmeren 1/Programmeren 2/elders waar er meerdere oplossingen waren voor hetzelfde probleem?

PROBLEEMSTELLING

Anders dan wiskundigen zijn informatici geïnteresseerd in het **vergelijken** van manieren om iets op te lossen.

Welke concepten van computationeel denken zijn hier aan gelinkt?

Algoritmisch denken en **Evaluatie**.

Kan je voorbeelden geven uit Programmeren 1/Programmeren 2/elders waar er meerdere oplossingen waren voor hetzelfde probleem?

- Sorteeralgoritmes → **focus van deze les**
- Een programma proceduraal of objectgericht schrijven
- Datastructuren met gelijkaardige kenmerken (lijsten vs tuples, dictionaries vs objecten)

PROBLEEMSTELLING

Anders dan wiskundigen zijn informatici geïnteresseerd in het **vergelijken** van manieren om iets op te lossen.

Welke concepten van computationeel denken zijn hier aan gelinkt?

Algoritmisch denken en **Evaluatie**.

Kan je voorbeelden geven uit Programmeren 1/Programmeren 2/elders waar er meerdere oplossingen waren voor hetzelfde probleem?

- Sorteeralgoritmes → **focus van deze les**
- Een programma proceduraal of objectgericht schrijven
- Datastructuren met gelijkaardige kenmerken (lijsten vs tuples, dictionaries vs objecten)

In deze les leren we de efficiëntie van algoritmes te evalueren. We focussen specifiek op sorteeralgoritmes, maar de principes zijn algemeen toepasbaar.

DOMEINEN

Het ontwerpen van algoritmes valt onder het domein **algoritmiek**. Delen ervan zijn al aan bod gekomen in eerdere lessen. Er zijn nog veel meer algoritmische technieken voor fictieve en reële problemen, maar deze komen niet verder aan bod in deze les.

Het analyseren van de efficiëntie van algoritmes valt onder **complexiteitsanalyse**. Zie ook de les rond fundamenten van informatica.

Complexiteitsanalyse steunt op formele wiskundige regels; vooral limieten zijn belangrijk. In deze les proberen we op een meer intuïtieve manier er vat op te krijgen. We beperken ons tot tijdsanalyse (hoe lang de uitvoering van een algoritme duurt). Er bestaat ook ruimteanalyse (hoeveel geheugen een algoritme nodig heeft tijdens uitvoering), maar dit behandelen we niet.

ALGEMENE REGELS ROND COMPLEXITEITSANALYSE

ALGEMENE REGELS (1)

In complexiteitsleer zoekt men uit hoe *snel* een algoritme werkt. Dit wordt niet uitgedrukt in seconden, maar in een **grootteorde** van hoeveelheid werk. We zijn geïnteresseerd in het beantwoorden van de volgende vraag:

Stel dat de invoer drastisch toeneemt, hoeveel meer tijd zal het algoritme nodig hebben om het resultaat (de uitvoer) te vinden?

Bij de analyse van de complexiteit van een algoritme, wordt bijna altijd uitgegaan van het **worst-case** scenario. Men gaat ook uit van een **(bijna) oneindig grote invoer**. Dit leidt tot een aantal regels rond verwaarlozing van bepaalde operaties.

De complexiteit van een algoritme wordt uitgredukt door de grote-O notatie (Engels: *Big-O notation*), geschreven als $O(...)$. Het geeft de *grootteorde* weer, niet de exacte tijdsduur of het exact aantal operaties.

ALGEMENE REGELS (2)

Wiskundige operaties (`+`, `-`, `*`, `/`, `%`) vragen een constante hoeveelheid tijd, ongeacht hoe groot/klein de getallen zijn. Constante tijd wordt uitgedrukt door $O(1)$.

Simpele acties die een effect hebben op de computer, worden meestal ook als constant beschouwd. **Dit is niet veralgemeenbaar**, maar in de context van deze les mag hiervan uitegaan worden. Voorbeelden:

- Een variabele een waarde toekennen
- `input()` en `print()`
- `len()`
- `return`

Meerdere constante operaties na elkaar resulteert nog steeds in $O(1)$. De impact van meerdere constante operaties wordt gezien als verwaarloosbaar tegenover ander werk.

VOORBEELD

Onderstaande functie accepteert een cijfer `x` en voert een aantal bewerkingen uit.

```python
def kwadrateer_en_deel_door_drie(x):
    y = x * x   # O(1)
    z = y / 3   # O(1)
    return z    # O(1)
```

Wat is de complexiteit van de functie in zijn geheel?

VOORBEELD

Onderstaande functie accepteert een cijfer `x` en voert een aantal bewerkingen uit.

```python
def kwadrateer_en_deel_door_drie(x):
    y = x * x   # O(1)
    z = y / 3   # O(1)
    return z    # O(1)
```

Wat is de complexiteit van de functie in zijn geheel?

Elk van de stappen heeft complexiteit $O(1)$. De complexiteit van de functie in zijn geheel is dus $O(1)$.

Dit betekent dat de functie `kwadrateer_en_deel_door_drie()` altijd in constante tijd zal werken, ongeacht de grootte van `x`.

$O(1)$ is het best mogelijke resultaat dat een algoritme/functie kan hebben. In de praktijk is dit zelden haalbaar.

ALGEMENE REGELS (3)

De voornaamste bron van werk komt door het gebruik van **herhalingen die afhangen van de grootte van invoer**.

Stel dat je als invoer een lijst van `n` elementen hebt. Als je één of meerdere constante operatie n keer uitvoert omwille van een herhaling, dan heeft het geheel complexiteit $O(n)$.

Bij `for`-lussen is het meestal makkelijk te voorspellen hoeveel keer ze uitgevoerd zal worden. Bij `while`-lussen moet je soms meer nadenken.

Tegenover $O(n)$ wordt $O(1)$ verwaarloosd.

VOORBEELD

De onderstaande functie berekent de som van alle cijfers in `lijst`:

Python

```
def bereken_som(lijst):
  som = len(lijst)          # O(1)
  for element in lijst:     # O(n)
    som += element            # O(1)
  return som                # O(1)
```

Wat is de complexiteit van de functie in zijn geheel?

VOORBEELD

De onderstaande functie berekent de som van alle cijfers in `lijst`:

```python
def bereken_som(lijst):
    som = len(lijst)          # O(1)
    for element in lijst:     # O(n)
        som += element            # O(1)
    return som                # O(1)
```

Wat is de complexiteit van de functie in zijn geheel?

`som += element` wordt n keer uitgevoerd, waarbij `n` de lengte van de lijst is. De herhaling is dus $O(n)$. De andere acties zijn $O(1)$, maar deze zijn verwaarloosbaar tegenover de herhaling.

ALGEMENE REGELS (4)

Bij een keuze of een splitsing in een algoritme, neem je altijd de **optie die op dat moment het minst gunstig is**. Dit kan dus variëren doorheen de uitvoering van het algoritme.

Dit geldt voor `if-elif-else`, maar ook `switch-case` en andere 'exotische' vormen van selecties.

De complexiteit van een functie `f` heeft impact op elke functie die `f` oproept. De complexiteit wordt overgedragen van 'lagere' functies naar 'hogere' functies.

VOORBEELD

Onderstaande functie accepteert m lijsten, elk van lengte n:

Python

```
def bereken_totale_som_van_lijsten_met_oneven_start(lijsten):
  totale_som = 0                          # O(1)
  for lijst in lijsten:                   # O(m)
    if lijst[0] % 2 == 1:                 # O(1)
      totale_som += bereken_som(lijst)    # O(n)
    else:
      print("Lijst overgeslagen")         # O(1)
  return totale_som
```

Wat is de complexiteit van de functie in zijn geheel? Let op het verschil tussen m en n.

VOORBEELD

Onderstaande functie accepteert m lijsten, elk van lengte n:

```python
def bereken_totale_som_van_lijsten_met_oneven_start(lijsten):
  totale_som = 0                          # O(1)
  for lijst in lijsten:                   # O(m)
    if lijst[0] % 2 == 1:                 # O(1)
      totale_som += bereken_som(lijst)    # O(n)
    else:
      print("Lijst overgeslagen")         # O(1)
  return totale_som
```

Python

Wat is de complexiteit van de functie in zijn geheel? Let op het verschil tussen m en n.

- We nemen steeds de slechts mogelijke optie in de `if-else`. In dit geval is de `if` altijd slechter, aangezien `bereken_som()` complexiteit $O(n)$ heeft (zie vorige slides)
- De `if-else` wordt m keer uitgevoerd
- het geheel van `for`-lus met `if-else`-selectie is dus $O(m * n)$
- De andere operaties zijn $O(1)$ en mogen dus verwaarloosd worden

ALGEMENE REGELS (5)

Behalve functies, kan je ook **lussen en selecties met elkaar (en met functies) combineren ('nesten')**. De complexiteiten worden dan vermenigvuldigd zoals in het vorige voorbeeld met $O(m * n)$.

Python

```python
def print_speciale_reeks(n):
  for i in range(n):          # O(n)
    for j in range(n / 2):    # O(n)
      print(i + j)            # O(1)
```

Wat is de complexiteit van deze functie?

ALGEMENE REGELS (5)

Behalve functies, kan je ook **lussen en selecties met elkaar (en met functies) combineren ('nesten')**. De complexiteiten worden dan vermenigvuldigd zoals in het vorige voorbeeld met $O(m * n)$.

Python

```
def print_speciale_reeks(n):
  for i in range(n):          # O(n)
    for j in range(n / 2):    # O(n)
      print(i + j)            # O(1)
```

Wat is de complexiteit van deze functie?

- `print(i + j)` is een constante operatie
- `for j in range(n / 2)` is $O(n)$. Er staat wel `n / 2`, maar de constante factor $1/2$ weegt niet op tegen een zeer grote n
- `for i in range(n)` voert de lus erbinnen nog eens n keer uit. Het geheel is dus $O(n * n)$ of $O(n^2)$

ALGEMENE REGELS (6)

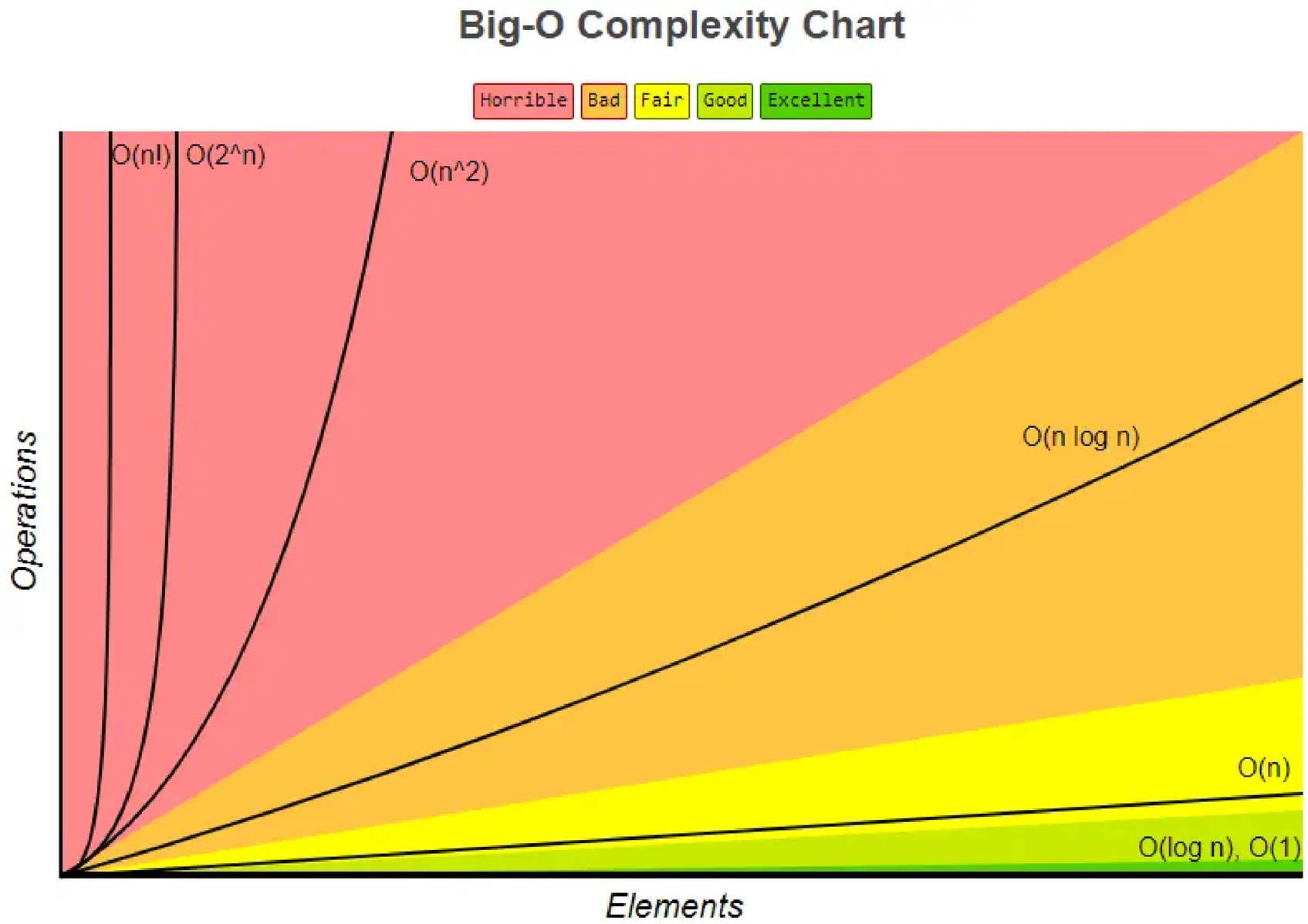
Recursie is iets uitdagender. Als je geluk hebt, kan je de code herschrijven met lussen en van daaruit de voorgaande regels toepassen.

Als herschrijven moeilijk tot onmogelijk is, kan je het volgende proberen:

1. Bepaal de complexiteit van de base cases
2. Bepaal hoeveel recursieve cases worden opgeroepen voor $n = 2, 3, 4, \ldots$ en vermenigvuldig ze met de complexiteit van de base cases
3. Probeer een patroon te vinden en schrijf dit in $O(\ldots)$-notatie

SOORTEN COMPLEXITEIT VISUEEL VOORGESTELD

GRAFIEK



Bron: Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell

UITLEG BIJ GRAFIEK

- Deze grafiek toont verschillende complexiteiten en hun overeenkomstige wiskundige functies. De x-as stelt de grootte van de invoer voor (n)
- `^` betekent 'tot de macht'
- Algoritmes met complexiteit $O(n^2)$ en erger worden als zeer slecht gezien qua complexiteit, en dus te vermijden, tenzij er geen beter algoritme bestaat
- $O(n^3)$, $O(n^4)$, ... bestaan, maar zijn niet getoond. Ze liggen tussen $O(n^2)$ en $O(2^n)$ (zeer slecht dus)
- $O(n!)$ is veruit de slechtst mogelijke complexiteit
- De website geeft ook een lijst van populaire algoritmes uit computerwetenschappen en hun complexiteit

COMPLEXITEIT TOEGEPAST OP ALGORITMES

BUBBLE SORT: VISUEEL (TER OPFRISSING)

Filmpje op YouTube - bubble sort start op 4:01

BUBBLE SORT: PSEUDOCODE

Python

```
def bubble_sort(lijst):
  n = lengte van lijst
  fase = 0
  while fase < n - 1:
    positie = 0
    while positie < n - 1 - fase:
      als lijst[positie] groter is dan lijst[positie + 1]
        dan wissel elementen van plaats
      positie += 1
    fase += 1
  return lijst
```

Bepaal de complexiteit van het hele algoritme. Bepaal eerst de complexiteit van elke lijn apart, en pas de regels van vorige slides toe om ze te combineren tot een eindresultaat.

OPLOSSING

OPLOSSING

Python

```
def bubble_sort(lijst):
  n = lengte van lijst                                    # O(1)
  fase = 0                                                # O(1)
  while fase < n - 1:                                     # O(n)
    positie = 0                                           # O(1)
    while positie < n - 1 - fase:                         # O(n)
      als lijst[positie] groter is dan lijst[positie + 1] # O(1)
        dan wissel elementen van plaats                   # O(1)
      positie += 1                                        # O(1)
    fase += 1                                             # O(1)
  return lijst                                            # O(1)
```

OPLOSSING

Python

```python
def bubble_sort(lijst):
  n = lengte van lijst                                    # O(1)
  fase = 0                                                # O(1)
  while fase < n - 1:                                     # O(n)
    positie = 0                                           # O(1)
    while positie < n - 1 - fase:                         # O(n)
      als lijst[positie] groter is dan lijst[positie + 1] # O(1)
        dan wissel elementen van plaats                   # O(1)
      positie += 1                                        # O(1)
    fase += 1                                             # O(1)
  return lijst                                            # O(1)
```

- Binnenste lus wordt gemiddeld $(n - 1) / 2$ keer uitgevoerd (in het begin meer, naar het einde toe minder). Dit wordt vereenvoudigd tot $O(n)$
- Totale complexiteit is $O(n^2)$ door de geneste lus → vrij slecht
- als de rij in lengte verdubbelt, verviervoudigt (ongeveer) de tijd nodig om ze te sorteren (bv lengte x 10 → +/- tijd x 100)

KAN HET BETER?

Bubble sort is intuïtief te programmeren en te begrijpen, maar is praktisch enkel bruikbaar voor kleine invoer. Programmeurs hebben doorheen de jaren allerlei andere sorteeralgoritmes bedacht die in bepaalde gevallen beter werken. Soms zit er wel een kost aan verbonden.

- Merge sort en Heap sort:
  - **Onthouden meer informatie** zodat dubbel werk vermeden wordt
  - Tijdscomplexiteit: $O(n * log_2 n)$
- Quicksort:
  - Verdeel-en-heers, recursief (zie lessen over recursie + CS Unplugged)
  - Tijdscomplexiteit: gemiddeld $O(n * log_2 n)$, worst case $O(n^2)$
- Python `sort()` gebruikt *timsort*:
  - Geïmplementeerd door Tim Peters in 2002
  - Mix van insertion en merge sort
  - Zit ook in bv. Java (wat gebruikt wordt voor ondermeer Android OS en apps)
- Andere sorteeralgoritmes: zie bv. Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell

COMPLEXITEIT VAN FIBONACCI RECURSIEF

```python
def fib(n):
  if n == 0 or n == 1:
    return n
  else:
    return fib(n - 1) + fib(n - 2)
```

Bepaal de complexiteit van de base cases. Tel het aantal recursieve oproepen voor $n = 2, 3, \ldots$. Kan je een patroon vinden?

FIBONACCI OPLOSSING

De twee base cases `n==0, n==1` zijn telkens $O(1)$.

n	Aantal functieoproepen
0	1
1	1
2	3
3	5
4	9

Als je verder gaat, zal je merken dat het aantal functie-oproepen drastisch toeneemt. De tijdscomplexiteit is in feite $O(2^n)$. Dit is slechter dan eender welk ander algoritme dat we hebben gezien!

Dit verklaart waarom het programma bij $n = 10, 20, 30, \ldots$ al moeite heeft met een resultaat te berekenen.