Samenvatting algoritmen

# # Deel 1: inleiding

## ## Hoofdstuk 1: Onderwerp en doel

**### Wat zijn algoritmen?**

Algoritmen zijn methodes om problemen op te lossen die geschikt zijn voor implementatie op een computer. De gegevens die ze daarbij gebruiken kunnen op verschillende manieren georganiseerd worden.

* Structuur is minstens even belangrijk als het algoritme zelf

Een algoritme moet efficiënt zijn. Zowel voor:

- gebruikte geheugenruimte

- uitvoeringstijd

Snelle computers zijn niet voldoende, ook goede algoritmen zijn belangrijk. hoe sneller computers, hoe belangrijker efficiëntere algoritmen want een snelle computer kan een ingewikkelder probleem oplossen.

**### Bedoeling vak**

* Overzicht geven van beschikbare algoritmen en gegevensstructuren en hun eigenschappen, om ze daarna juist te kunnen gebruiken
* Inzicht bieden in de implementatie van efficiënte algoritmen en gegevensstructuren. Zo kan het algoritme aangepast worden om specifieke problemen op te lossen
* Tonen dat het gedrag van een algoritme kan voorspeld worden zonder het te implementeren en te leren hoe men dat doet.
* Oplossingenstechnieken bijbrengen. Veel algoritmen zijn concrete toepassingen van algemene algoritmische methodes, die voor gelijkaardige en nieuwe problemen aangewend kunnen worden.

In dit vak leren we algoritmen met niet numerieke gegevens.

**### Algoritme vs programma**

Een algoritme is geen programma en staat ook los van de computer waar het eventueel op uitgevoerd zal worden. Een programma is een implementatie in een bepaalde programmeertaal die en of meerdere algoritmen combineert.

## ## Hoofdstuk 2: Efficiëntie van algoritmen

## ## Hoofdstuk 2.1: Analyse van algoritmen

- Uitvoeringstijd van algoritmen staat centraal

- Voor veel problemen bestaan meerdere oplossingen

🡺 Daarom is het nuttig om op voorhand de uitvoeringstijd van algoritmen te kunnen voorspellen, zonder ze te moeten implementeren en uit te voeren. Dit kunnen we bereiken door een **analyse** uit te voeren.

Een exacte tijdsduur voorspellen is echter een illusie. Dit hangt af van verscheidene factoren:

* Programmeertaal
* Bekwaamheid van de programmeur
* Gebruikte compiler
* Processorarchitectuur van de computer

De exacte tijdsduur is dus ook afhankelijk van de gebruikte computer. Daarom willen we een andere methode in het leven roepen: We gaan **het aantal primitieve operaties bepalen** die elke implementatie zou moeten uitvoeren op een typische sequëntiële processor (het **zogenaamde RAM-model, later meer**).

Allereerst hangt de uitvoeringstijd af van het aantal verwerkte gegevens:

**### We onderscheiden 3 gevallen:**

* Het beste geval (best-case running time)
  + Bv: ordeningsalgoritmen zijn het snelst wanneer alles al in een juiste volgorde staat
* Het slechtste geval (worst-case running time)
  + Bv: ordeningsalgoritmen zijn het traagst wanneer de te sorteren gegevens in omgekeerde volgorde staan
* Het gemiddelde geval (average case running time)
  + Meestal het moeilijkste te bepalen
  + Probleem met de definitie van ‘gemiddeld’. Dit hangt af van persoon tot persoon
  + Om analyse mogelijk te maken meestal worden de beginveronderstelling **niet zeer realistisch gekozen**.
  + vb: bij rangschikken kan een beginvoorwaarde zijn dat de permutatie van de invoergegevens even waarschijnlijk is. Dit is in de praktijk zelden het geval.

**### Las Vegas-algoritme** : Een algoritme waarbij de kans dat het mis gaat klein genoeg is zodat men het risico neemt.

**### De berekening zelf**

* Het resultaat gaat dus altijd afhangen van
* We zijn niet geïnteresseerd in de exacte tijd
* Daarom kiezen we **constanten** voor primitieve operaties, zodat de ‘exacte’ waarde van de factoren niet moeten berekenen.
* De constanten worden voorgesteld met een **symbolische naam**

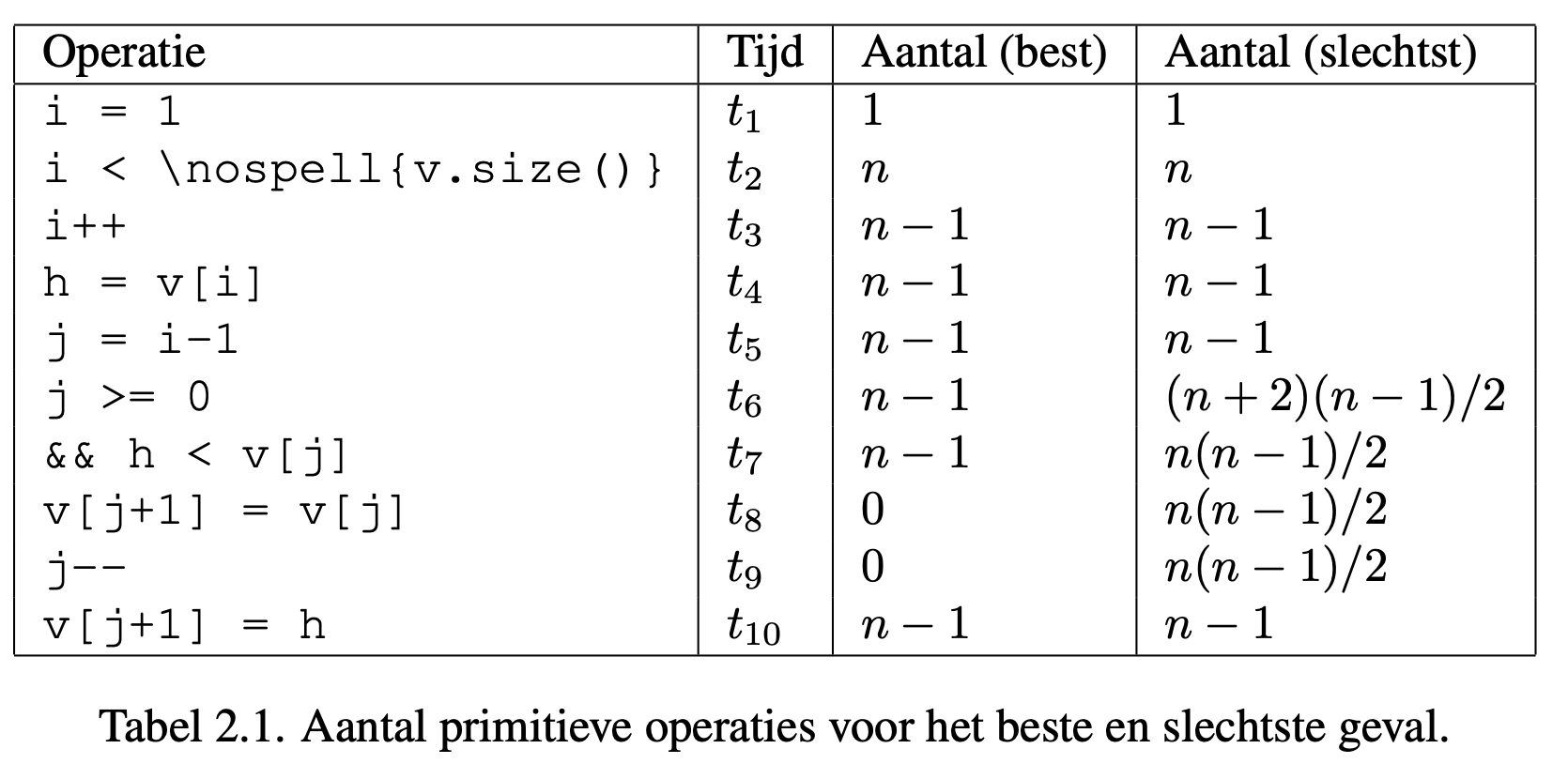
*Voorbeeld*

[*https://upload.wikimedia.org/wikipedia/commons/9/9c/Insertion-sort-example.gif*](https://upload.wikimedia.org/wikipedia/commons/9/9c/Insertion-sort-example.gif)

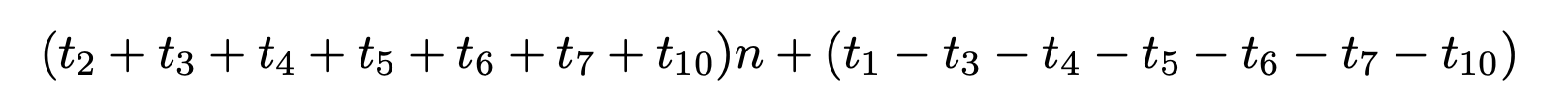
1. **void** insertion\_sort (vector<T> & v) {
2. // Stijgend rangschikken
3. **for**(**int** i = 1; i < v.size(); i++){
4. // De eerste i getallen staan reeds in volgorde
5. T h = move(v[i]);
6. **int** j = i - 1;
8. // Hier gebeurt het sorteren zelf
9. **while**(j >= 0 && h < v[j]){
10. v[j + 1] = move(v[j]);
11. j--;
12. }
13. v[j+1] = move(h);
14. }
15. }

* Elke toewijzing en elke afzonderlijke test vereisen een constante tijd die we als aanduiden.
* De grootte van de vector, of dus het aantal getallen is gelijk aan
* De opdrachten in de for-lus worden keer uitgevoerd.
* Het aantal keer dat de while uitgevoerd wordt hangt af van de plaats en de volgorde van het in te voegen getal en de reeds gerangschikte deelvector.

**### Operaties**



**### Beste geval: Alles is al gesorteerd**



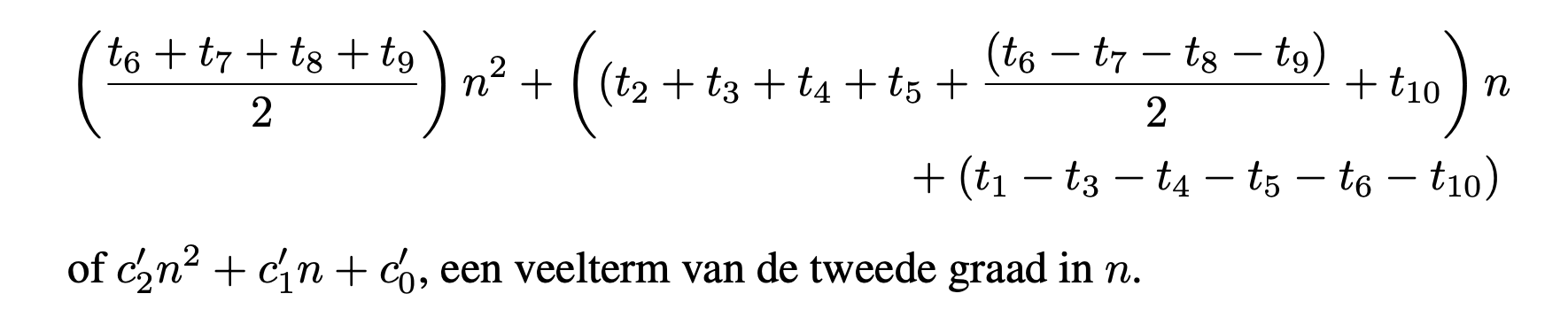
De factor n, staat dus voor de forlus die alle operaties

* De tweede factor van de som zorgt dus voor de correctie naar n-1 van de bepaalde tijden.

**🡺 **

**### Slechtste geval: De vector is omgekeerd gesorteerd**

* Elke while lus moet dan i keer herhaald worden (een waarde moet altijd helemaal naar het begin verschoven worden, dus i keer)
* Bijgevolg zal de veelterm kwadratisch zijn



* Als je kijkt in de tabel voor zie je de term (n+2)(n-1)/2. Dit is een vaak voorkomende formule voor de sommatie van 2 lussen die niet volledig doorgevoerd worden. Het resultaat is dus gewoon kwadratisch

**### Gemiddelde geval**

* Elke permutatie van de te rangschikken getallen is even waarschijnlijk. Een tussen te voegen getalen kan dus met dezelfde waarschijnlijkheid op elke mogelijk positie terechtkomen. Gemiddeld zal dus altijd de helft van de tabel doorsparteld moeten worden. Dan krijgen we iets in de aard van:

**### Conclusie**

Met de constanten zijn de functies heel wat eenvoudiger, toch is het nodig om een stap verder te gaan. We zijn namelijk geïnteresseerd in de waarde van de **toename van de tijd** als **n** stijgt. Deze waarde wordt bepaald door de **grootste term**. Toegepast op ons voorbeeld:

* Beste tijd:
* Slechtste tijd:
* Gemiddelde tijd:

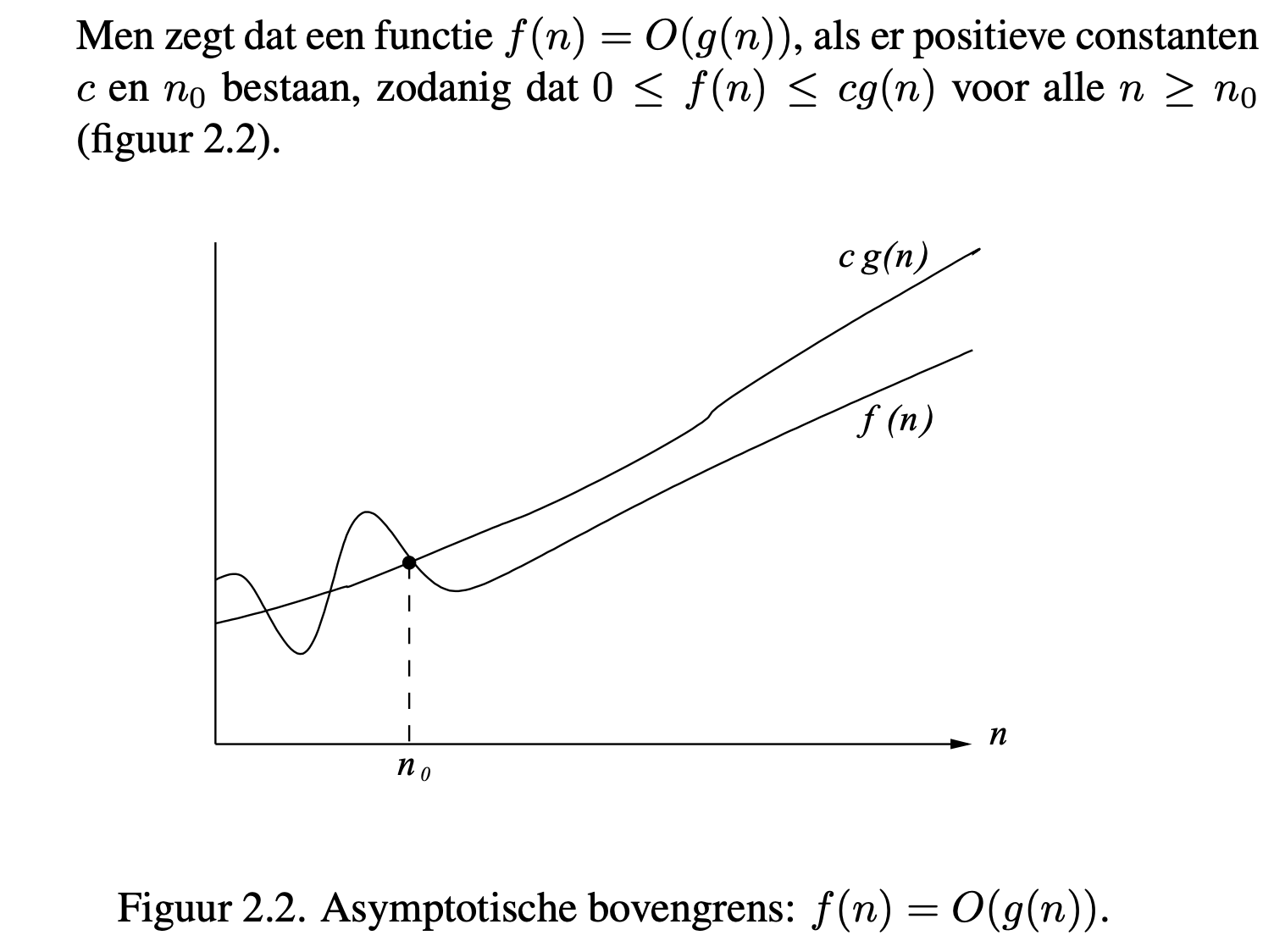
Wanneer men twee algoritmen heeft voor hetzelfde probleem, wordt het algoritme waarvan de uitvoeringstijd voor het slechtst mogelijk geval het minst snel stijgt voor de **slechtste tijd** gezien als het betere algoritme.

## ## 2.2 : Asymptotische benadering van functies

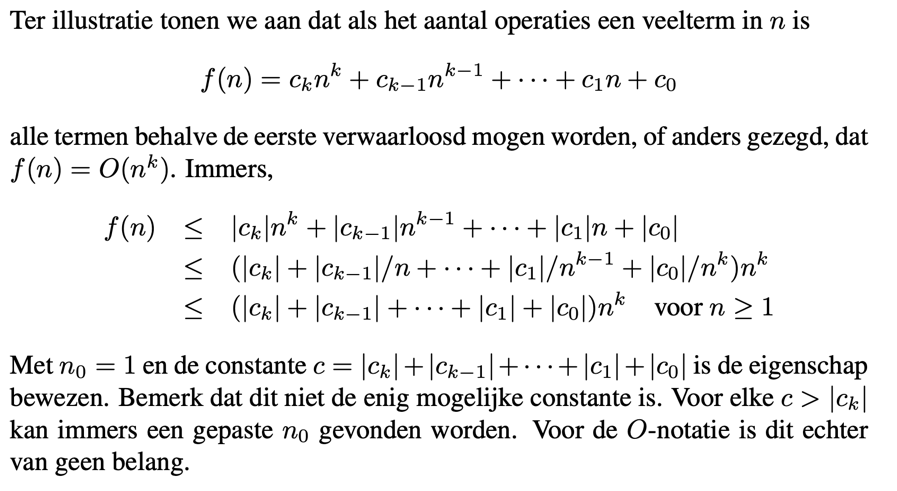
De vereenvoudigde functies bij de conclusie uit het vorig deeltje is een voorbeeld van een **asymptotische benadering**. Voor bepaalde algoritmen is het moeilijk om zoals in vorig deel te werk te gaan, namelijk eerst een uitvoeringstijd opstellen en die vervolgens vereenvoudigen. In dat geval proberen we een functie te vinden die de uitvoeringstijd zo goed mogelijk **begrenst.** Ook hier zijn er weer enkele notaties

**### -notatie**

Een functie groeit niet sneller dan . Deze laatste is dus een bovengrens.



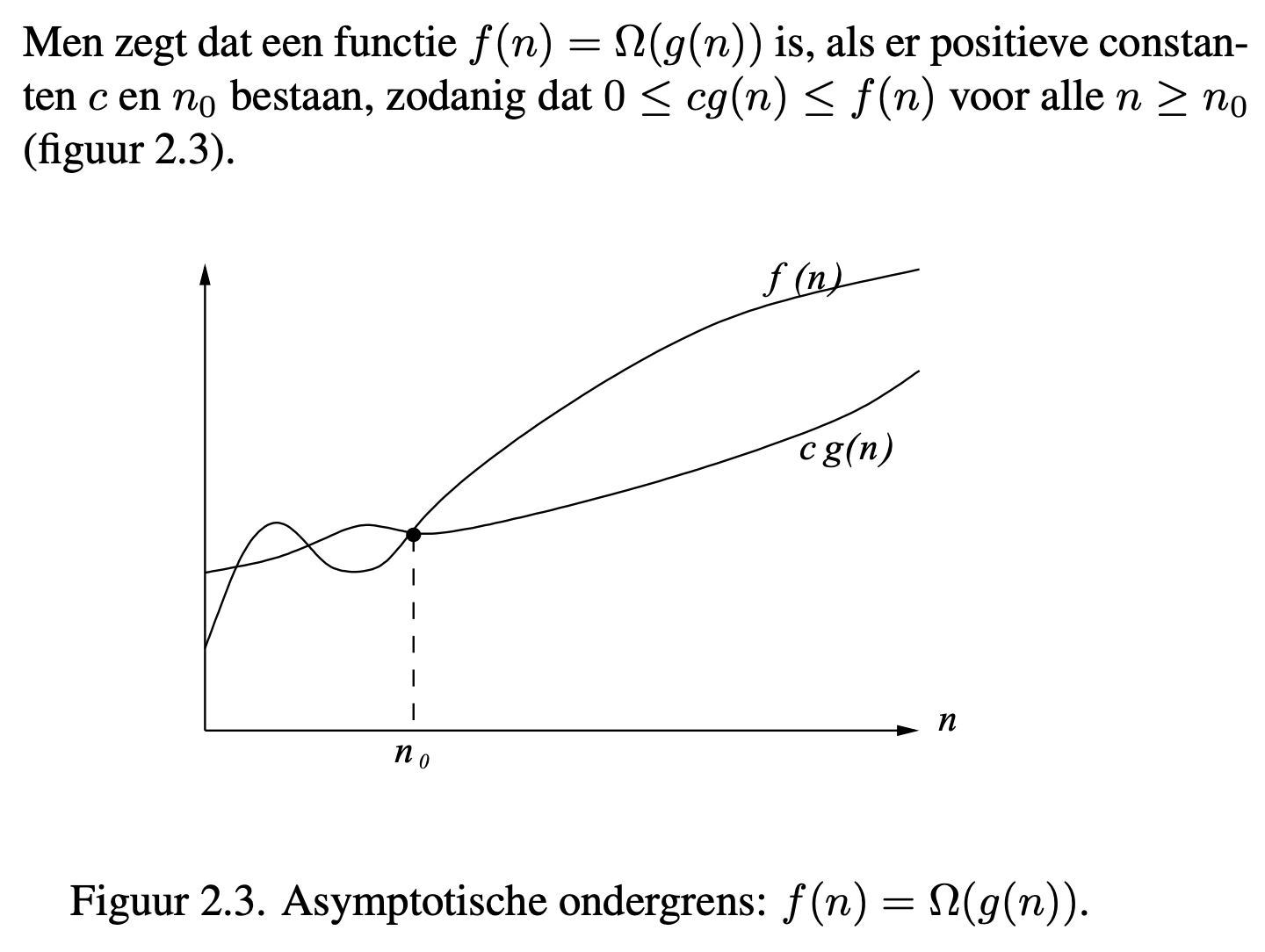
! De constante bij is niet nader gespecifieerd, ze moet gewoon bestaan !



Dit is het bewijs dat we enkel rekening moeten houden met de eerste term.

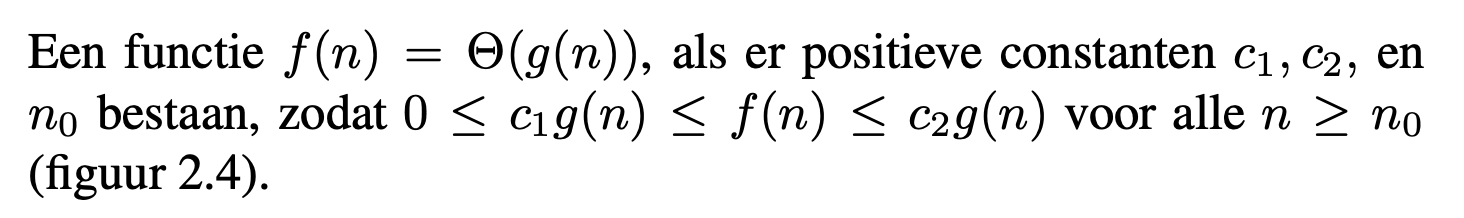
**### -notatie**

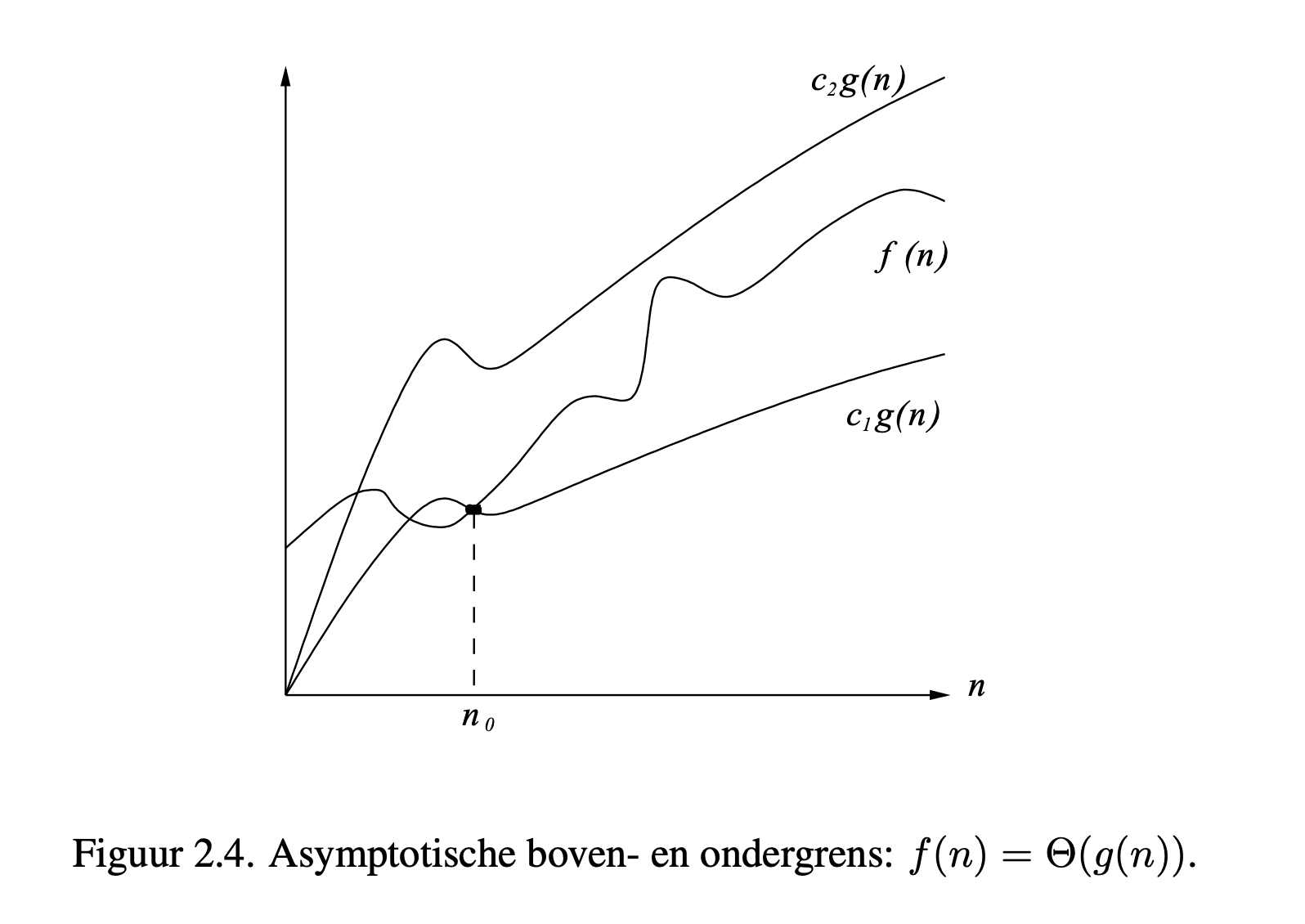
Een functie groeit minstens even snel als . Deze laatste is dan een ondergrens.



**### -notatie**

Een functie heeft zowel als bovengrens als ondergrens .

****

****

* Niet van elk probleem kan de asymptotische efficiëntie van het best mogelijk algoritme bepaald worden
  + Een bekend voorbeeld zijn de **NP-problemen** (Nondeterministic Polynomial). Voor geen enkel van deze problemen is een algoritme bekend dat efficiëntie heeft, voor welke alpha danook.

**### Stelling 1**