Samenvatting Algoritmen en Datastructuren

Lesgever: Prof. dr. Veerle Fack
Auteur(s): Manu De Buck
Laatste bewerking:
Bibliografie: Algoritmen en Datastructuren, Veerle Fack, acco

April 8, 2018

Chapter 1

Inleiding

1.1 Algoritmen

Functie: Koppeling tussen inputs (**het domein**) en outputs (**het bereik**).
Parameters: Waarden waaruit de input bestaat.
→Zelfde input genereert **altijd** zelfde output.
Algoritmisch probleem: Eindige of oneindige verzameling toegelaten inputwaarden, samen met een specificatie van de gewenste output als functie van de input, noemt met een algoritmisch probleem. Als er ten minste n oplossingsmethode bestaat die voor elke legale input de gewenste output voorbrengt.
Algoritme: is een methode die wordt gevolgd om een algoritmisch probleem op te lossen.

1. Algoritme moet correct zijn
2. Bestaat uit concreet aantal stappen. Elke stap uitvoerbaar in eindige hoeveelheid tijd.
3. Geen dubbelzinnigheid betreffende de stappen.
4. Bestaat uit eindig aantal stappen.
5. Het algoritme moet eindigen.

1.2 Ontwerp en specificatie van algoritmen

Pseudocode: Mengeling van natuurlijke taal en constructies uit een programmeertaal.
→Input, beschrijving output en reeks pseudocodeopdrachten die algoritme weergeven.

1.3 Correctheid van algoritmen

Formele wiskundige bewijstechnieken om de correctheid van een algoritme aan te tonen. Indien bewijs niet gevonden wordt: beroep doen op **het uitvoeren van testen**.

1.3.1 Bewijzen door contradictie

Geven van een tegenvoorbeeld voor de bewering, maar volstaat niet om te bewijzen dat bewering waar is.
Wel correct: **bewijzen door contradictie:** veronderstellen dat bewering niet waar is, daaruit tegenstrijdigheid afleiden.

1.3.2 Bewijzen door inductie

Bewijzen dat een reeks beweringen X_1, X_2, ..., X_n waar is, door eerst te bewijzen dat X_1 waar is en vervolgens te veronderstellen dat X_i waar is (inductiehypothese) en te bewijzen dat X_{i+1} waar is (inductiestap).
Eenvoudige vorm inductie vs. **sterke wiskundige inductie**
Uit het ene volgt het volgende vs. We nemen aan dat alles voorafgaand aan X_j voor $j = 1$, ..., i waar is.

1.3.3 Bewijzen met lusvarianten

Lusvariante: is eenbewering over variabelen die waar is vooraleer de lus uitgevoerd word en die ook waar is na elke iteratiestap in de lus.

1.4 Efficiëntie van algoritmen

1.4.1 Analyse van algoritmen

Belang voor het nagaan van complexiteit van algoritme in ruimte en tijd. Een **schatting** maken van de **benodigde geheugenruimte** en de **uitvoeringstijd**.
Een algortime is **efficiënt** als het het gestelde probleem oplost binnen de vooropgestelde beperkingen qua resources. De **kost** van een oplossing is de hoeveelheid resources die de oplossing verbruikt.
complexiteitsanalyse van de algoritmen laat toe ze **onderling te vergelijken** en afhankelijk daarvan de efficiëntste te selecteren.
Twee doelstellingen bij oplossen probleem:

1. Algoritme ontwerpen dat eenvoudig te begrijpen, coderen en debuggen is.

2. Algoritme ontwerpen dat de beschikbare resources efficiënt gebruikt.

Zie hoofdstuk 2 voor **asymptotische analyse**.

1.4.2 Snelle schattingen

Maken van een snelle schatting:

1. Bepaal de belangrijke factoren die het probleem beïnvloeden.
2. Stel een vergelijking op die de parameters van het probleem met elkaar verbindt.
3. Selecteer waarden voor de parameters, en gebruik de bekomen vergelijking om een geschatte oplossing te bekomen.

1.5 Datastructuren

Datastructuur: een voorstelling van gegevens en de bijbehorende bewerkingen op die gegevens.
Selecteren datastructuur:

1. Analyseer het probleem om te bepalen welke vereisten qua resources elke oplossing moet voldoen.
2. Bepaal de basisbewerkingen die moeten worden ondersteund, en de vereisten waaraan ze moeten voldoen. Voorbeelden van basisbewerkingen zijn het toevoegen van een element aan de datastructuur, het verwijderen van een element uit de datastructuur en het opzoeken van een gegeven element.
3. Selecteer de datastructuur die het best aan deze vereisten voldoet.

Vereisten op bepaalde sleutelbewerkingen:

1. Worden alle gegevens aan de datastructuur toegevoegd vooraleer de andere bewerkingen (zoals opzoekingen) gebeuren of zijn toevoegingen afgewisseld met andere bewerkingen?
2. Kunnen elementen verwijderd worden?
3. Worden de elementen verwerkt in een specifieke volgorde of is willekeurige toegang mogelijk?

Abstract datatype: Een abstracte specificatie van een datastructuur die de formele beschrijving van een data-object evenals een beschrijving van de bewerkingen die op de structuur kunnen worden uitgevoerd. **(ADT)**.
Door abstractie te maken van het type van de componenten en meer in het bijzonder dit type als een parameter te behandelen, specificeert men een **generische datastructuur**.

Chapter 2

Analyse van algoritmen

2.1 Complexiteit van algoritmen

2.1.1 Inleiding

Asymptotische analyse van het algoritme: meet de efficiëntie van een algoritme, of zijn implementatie als een computerprogramma wanneer zijn inputgrootte groot wordt. Het is een schattingstechniek, die niets zegt over de relatieve verdiensten van twee programma's waarbij het ene net iets sneller is dan het andere.
Analyse bestaat typisch uit:

1. Inschatten nodige **uitvoeringstijd**
2. Inschatten benodigde **geheugenruimte** voor een **datatsructuur**

2.1.2 Theoretisch model

Inputgrootte: het aantal inputgegevens dat verwerkt wordt.
De inputgegevens vormen vaak maat voor de omvang of **complexiteit** van een probleem.
Uitvoeringstijd van een algoritme is een functie van de grootte van de input.
Voor **theoretische analyse:** uitvoeringstijd uitdrukken als aantal basisbewerkingen uitgevoerd bij de oplossing van het probleem.
Basisbewerking: zijn uitvoeringstijd is niet afhankelijk van specifieke waarden van operanden.

Notatie: voor een inputgrootte n noteren we de uitvoeringstijd T van het algoritme als een functie van n dus als *T(n)*. Hierbij veronderstellen we dat *T(n)* een niet-negatieve waarde heeft.

2.1.3 Functies voor de uitvoeringstijd

Orde van toename van een functie, is de mate waarin de functie stijgt voor toenemende waarden van n.
Voorbeelden:

1. Lineair (n)
2. $n\log n$
3. Kwadratisch (n^2)
4. Kubisch (n^3)
5. Exponentieel (a^n, $a \in \mathbb{R}$)

2.1.4 Asymptotische analyse

Orde van toename, verwaarloost de constanten en lagere-ordetermen. Dit vereenvoudigt de analyse.

Notatie De **Θ-notatie** stelt de orde van toename als functie voor.
Bijvoorbeeld, $T(n) = 5n + 3$, dan is de uitvoerinstijd $T(n) = \Theta(n)$.
Dit laat toe de relatieve performantie van algoritmen te vergelijken.

Dergelijke analyse noemen we **asymptotische analyse van een algoritme**.

2.1.5 Gemiddelde, beste en slechtste uitvoeringstijd

Voor een gegeven algoritme kunnen we een onderscheid maken tussen $T_g(n)$, $T_s(n)$, $T_b(n)$ - respectievelijk de **gemiddelde uitvoeringstijd**, de **slechtst mogelijke uitvoeringstijd** en de **best mogelijke uitvoeringstijd** van het algoritme als functie van de probleemgrootte n. Vanzelfsprekend geldt er: $T_b(n) \leq T_g(n) \leq T_s(n)$.

2.2 Asymptotische notaties

2.2.1 Definities

Zijn gedefinieerd om asymptotische gedrag van twee gegeven functies *f en g* gedefinieerd op $\mathbb{N}$, waarbij verondersteld wordt dat *f en g asymptotische niet-negatieve functies zijn.*

Bovengranzen en O-notatie

Bovengrens geeft aan wat hoogste orde van toename is dat algoritme kan hebben.
Merk op: niet hetzelfde als slechtst mogelijke uitvoeringstijd voor een gegeven input van grootte n.
"Heeft een bovengrens voor zijn orde van toename van *f(n)*" : de O-notatie.

Definitie 2.2.1. *$f(n) = O(g(n))$ indien er constanten $c \in \mathbb{R}^+_{>0}$ en $n_0 \in \mathbb{N}$ bestaan, zodanig dat $0 \leq f(n) \leq cg(n)$ voor alle $n \geq n_0$*

Men zegt: **f(n) wordt asymptotisch naar boven toe begrensd door g(n)**.

Ondergrenzen en Ω-notatie

Ondergrens van een algoritme wordt genoteerd door de Ω-notatie.

Definitie 2.2.2. *$f(n) = \Omega(g(n))$ indien er constanten $c \in \mathbb{R}^+_{>0}$ en $n_0 \in \mathbb{N}$ bestaan, zodanig dat $f(n) \geq cg(n) \geq 0$ voor alle $n \geq n_0$*

Men zegt dat **asymptotisch naar onder toe begrensd wordt door g**.

Merk op: f(n) = O(g(n)) a.s.a $g(n) = \Omega(f(n))$

Θ-notatie

Als bovengrens en ondergrens gelijk zijn op een constante factor na: uitdrukken door Θ-notatie.

Definitie 2.2.3. *$f(n) = \Theta(g(n))$ a.s.a $f(n) = O(g(n))$ en $f(n) = \Omega(g(n))$, m.a.w. indien er constanten c_1, $c_2 \in \mathbb{R}^+_{>0}$ en $n_0 \in \mathbb{N}$ bestaan, zodanig dat $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ voor alle $n \geq n_0$.*

Men zegt dat f en g op een positieve constante na, **hetzelfde gedrag op oneindig** vertonen. Merk op dat de volgende symmetrie-eigenschap nu geldig is: $f(n) = \Theta(g(n))$ a.s.a. $g(n) = \Theta(f(n))$

o- en ω-notatie

voor het specificeren van strikte boven en ondergrenzen op de orde van toename.

Definitie 2.2.4. *$f(n) = o(g(n))$ a.s.a. $f(n) = O(g(n))$ en $f(n) \neq \Theta(g(n))$.*

Definitie 2.2.5. *$f(n) = \omega(g(n))$ a.s.a. $f(n) = \Omega(g(n))$ en $f(n) \neq \Theta(g(n))$.*

2.2.2 Werken met asymptotische notaties

De limietregel

Vergelijken door $\lim_{n \to \infty}$ f(n) / g(n) (evt. regel van de l'Hôpital.

1. De limiet is 0: $f(n) = 0(g(n))$ en $f(n) \neq \Theta(g(n))$, of dus $f(n) = o(g(n))$.
2. De limiet is een constante c $\neq 0$: $f(n) = \Theta(g(n))$.
3. De limiet is $+\infty$: $f(n) = \Omega(g(n))$ en $f(n) \neq \Theta(g(n))$, of dus $f(n) = \omega(g(n))$.
4. De limiet bestaat niet: dan moet een eventueel ordeverband tussen f(n) en g(n) op een andere manier worden bepaald.

Vereenvoudigingsregels

1. Als $f(n) = \Theta(g(n))$ en $g(n) = \Theta(h(n))$, dan $f(n) = \Theta(h(n))$.
   Als een functie g(n) een maat geeft voor de orde van toename van de kostfunctie f(n), dan geeft elke functie h(n) die een maat geeft voor de orde van toename van g(n) ook een maat voor de orde van toename van f(n).
2. Als $f_1(n) = \Theta(g_1(n))$ en $f_2(n) = \Theta(g_2(n))$, dan $f_1(n) + f_2(n) = \Theta(max(g_1(n), g_2(n)))$.
   Van twee gedeelten van een algoritme die na elkaar worden uitgevoerd moeten we enkel het duurste gedeelte beschouwen.
3. Als $f_1(n) = \Theta(g_1(n))$ en $f_2(n) = \Theta(g_2(n))$, dan $f_1(n) f_2(n) = \Theta(g_1(n) g_2(n))$.
   Als een bepaalde actie een aantal keren herhaald wordt en elke herhaling dezelfde kost heeft, dan is de totale kost gegeven door de kost van de actie vermenigvuldigd met het aantal herhalingen. Deze regel is nuttig bij het analyseren van eenvoudige lussen in algoritmen.

   Analoge regels zijn geldig voor O- en Ω-notatie

2.2.3 Asymptotisch gedrag van standaarfuncties

Stelling 2.2.6

1. *Als T(n) een veelterm in n van graad d is, dan is T(n) = $\Theta(n^d)$.*
2. *$n^b = o(a^n)$, voor alle reële constanten a en b, met $a > 1$.*
3. *$(log_c n)^b = o(n^a)$, voor alle reële constanten a, b, c met a, $c > 0$.*

Verwisselen tussen basissen van logaritmen gaat met behulp van volgende formule: $log_a x = \frac{log_c x}{log_c a}$, met c > 0.

Stelling 2.2.7 *De Fibonacci-getallen, gedefinieerd als $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-1}$ voor $n > 1$, vormen een exponentieel stijgende functie.*

Bewijs. De volgende eigenschap geldt voor de Fibonacci-getallen $F_i = (\phi^i - \phi_-^i)/\sqrt{5}$, met $\phi = (1 + \sqrt{5})/2 = 1.61...$ en $\phi_- = (1 - \sqrt{5})/2 = -0.61....$ Aangezien $|\phi_-| < 1$, geldt dat $|\phi_-^i|/\sqrt{5} < 1/2$, zodat geldt dat $F_i = round(\phi^i/\sqrt{5})$.

Stelling 2.2.8 *De faculteitsfunctie is snelstijgend, met volgende eigenschappen:*

1. $n! = o(n^n)$,
2. $n! = \omega(2^n)$,
3. $log_2(n!) = \Theta(nlogn)$.

Bewijs. We bewijzen enkel eigenschap 3. Daartoe bewijzen we eerst een bovengrens:

$$\begin{aligned} log_2(n!) &= log_2 n + log_2(n-1) + ... + log_2 1 \\ &=\leq log_2 n + log_2 n + ... + log_2 n \\ &= nlog_2 n \end{aligned} \tag{2.1}$$

Hieruit volgt dat $log_2(n!) = O(nlogn)$. Vervolgens bewijzen we een ondergrens:

$$\begin{aligned} log_2(n!) &= logn + log_2(n-1) + ... + log_2 1 \\ &\geq log_2 n + log_2(n-1) + ... + log_2(\lceil n/2 \rceil) \\ &\geq log_2(\lceil n/2 \rceil) + log_2(\lceil n/2 \rceil) + ... + log_2(\lceil n/2 \rceil) \\ &= \lceil (n+1)/2 \rceil * log_2(\lceil n/2 \rceil) \\ &\geq (n/2) log_2(n/2) \\ &= (n/2) log_2(n - n/2) \\ &\geq (nlog_2 n)/4, n \geq 4 \end{aligned} \tag{2.2}$$

Hieruit volgt dat $log_2(n!) = \Omega(nlogn)$. Dus hebben we bewezen dat $log_2(n!) = \Theta(nlogn)$

2.3 Bepalen van tijds- en geheugencomplexiteit

2.3.1 Het tijd/ruimte-tradeoff-principe

Dit principe zegt ons dat in vele gevallen een reductie in uitvoeringstijd allen kan worden bekomen als men bereid is om geheugenruimte op te offeren, en vice versa.

2.4 Praktische beschouwingen

2.5 Geamortiseerde complexiteitsanalyse

Beschouwt de kost van een ganse sequentie van m bewerkingen , en kent aan iedere individuele bewerking een gedeelte van de totale kost toe. Men noemt dit de **geamortiseerde kost** van de bewerking. Indien we voor een bewerking kunnen aantonen dat het slechtste geval niet herhaaldelijk kan voorkomen, kunnen we een betere begrenzing voor de totale tijd bekomen en kunnen we de bewerkingen beschouwen alsof ze uitgemiddelde begrenzing van deze totale begrenzing heeft. We noemen dit een **geamortiseerde tijdsbegrenzing**.

2.6 Handelbare en onhandelbare problemen

Handelbaar Een computationeel probleem wordt handelbaar genoemd als er een polynomiaal[1] algoritme bestaat om het probleem op te lossen; de betekenis hiervan is dat er dan een efficiënt algoritme voor het probleem bestaat.

Onhandelbaar Een computationeel probleem wordt onhandelbaar genoemd als kan worden bewezen dat er geen polynomiaal algoritme is om het probleem op te lossen.

NP-complete De meeste van deze onhandelbare problemen behoren tot de klasse van NP-complete problemen. (De klasse NPC). Hiervan is wel reeds bewezen dat, als er voor n van de problemen uit de klasse NPC een polynomiaal algoritme bestaat, er dan ook voor alle andere problemen uit de klasse NPC een polynomiaal algoritme bestaat.

Beslissingsprobleem Een beslissingsprobleem is een probleem dat enkel een antwoord "ja" of "nee" vereist, afhankelijk van het feit of de input een bepaalde eigenschap heeft. Zo'n probleem behoort tot de **klasse** P als er een polynomiaal algoritme bestaat om het probleem op te lossen. Anders behoort het tot de **klasse NP**[2] als er een manier is om de correctheid van een "ja"-antwoord in polynomiale tijd te verifiëren.

Polynomiaal herleidbaar Een beslissingsprobleem R is polynomiaal herleidbaar tot Q als er een transformatie in polynomiale tijd bestaat van elke instantie I_R van probleem R naar een instantie I_Q van probleem Q, zodanig dat de instanties I_R en I_Q hetzelfde antwoord ("ja" of "nee") hebben.

NP-moeilijk Een beslissingsprobleem is NP-moeilijk als elk probleem in de klasse NP polynomiaal herleidbaar is tot R.

[1] Als de benodigde tijd, als functie van n, begrensd wordt door een polynoom

[2] Niet-deterministisch polynomiaal.

NP-compleet Een NP-moeilijk beslissingsprobleem R is NP-compleet als R tot de klasse NP behoort. De klasse van NP-complete problemen wordt ook de klasse NPC genoemd.

Chapter 3

Algoritmen en abstracte datatypes in de Java API

Chapter 4

Gebruik van stapels en (prioriteitswachtlijnen)

4.1 Stapels en compilers

Stapel Een stapel is een collectie-datatsructuur waarbij geldt dat het element dat het laatst werd toegevoegd, het eerst weer wordt opgehaald. Dit principe wordt ook wel LIFO (Last In First Out) genoemd.

4.2 Simulatie en prioriteitswachtlijnen

Prioriteitswachtlijnen Op bepaalde punten moet de volgende gebeurtenis in een collectie van gebeurtenissen worden bepaald, of een gebeurtenis op de juiste manier aan de collectie worden toegevoegd zodat deze op het juiste moment als volgende gebeurtenis aanzien kan worden. Hiervoor gebruikten we een prioriteitswachtlijn.

Discrete tijdsgestuurde simulatie Dit is een simulatie waarbij bij het begin van de simulatie de simulatieklok op nul gezet wordt, en vervolgens wordt de klik telkens één tik vooruit gezet en gecontroleerd of een gebeurtenis optreedt of niet.

Gebeurtenisgestuurde simulatie Dit is een simulatie waarbij de simulatieklok telkens vooruit geplaatst wordt naar de volgende gebeurtenis. Dit is conceptueel makkelijker te verwezenlijken.

Chapter 5

Recursie

5.1 Ontwerp van recursieve algoritmen

Recursief Een algoritme is recursief als het zichzelf oproept om een gedeelte van het werk uit te voeren. Opdat dit succesvol zou zijn, moet de oproep naar zichzelf een kleiner probleem dan het oorspronkelijke probleem betreffen.

Complexiteitsanalyse Dit verloopt vaak moeilijker, aangezien deze dikwijls beschreven worden door een **recurrente betrekking** die moet worden opgelost.

5.2 Analyse van recursieve algoritmen

Recurrente betrekking De uitvoeringstijd wordt doorgaans beschreven door een recurrente betrekking die moet worden opgelost. We beperken ons tot het bepalen van asymptotische Θ- of O-grenzen voor de oplossing.

5.2.1 Iteratiemethode

De betrekking wordt iteratief volledig uitgewerkt tot een sommatie waarin enkel n en de initile waarden optreden.

5.2.2 Substitutiemethode

Oplossing voor recurrente betrekking wordt vooropgesteld, en vervolgens wordt m.b.v. wiskundige inductie bewezen dat deze oplossing werkt.

5.2.3 Recursiebomen

Recursieboom dit is een handig hulpmiddel om te visualiseren wat er precies gebeurt bij het uitwerken van een iteratie voor een recurrente betrekking.

5.2.4 Master-methode

Deze methode geeft een recept voor het oplossen van recurrente betrekkingen van ed vorm $T(n) = aT(n/b) + f(n)$.

Stelling 5.2.1, master-stelling *Zij $a \geq 1$ en $b > 1$ constanten, zij f(n) een asymptotische positieve functie, en zij $T(n)$ gedefinieerd door de recurrente betrekking $T(n) = aT(n/b) + f(n)$.*

1. Als f(n) = $O(n^{log_b a - \varepsilon})$ voor zekere constante $\varepsilon > 0$, dan is $T(n) = \Theta(n^{log_b a})$.
2. Als f(n) = $\Theta(n^{log_b a})$, dan is $T(n) = \Theta(n^{log_b a} logn)$.
3. Als f(n) = $\Omega(n^{log_b a + \varepsilon})$ voor zekere constante $\varepsilon > 0$, en als $af(n/b) \leq cf(n)$ voor zekere constante $c < 1$ en voldoende grote n (d.i. regulariteitsvoowaarde), dan is $T(n) = \Theta(f(n))$

5.3 Wanneer recursie af te raden is

5.3.1 Staartrecursie

Staartrecursie Over het algemeen geldt dat recursie wordt afgeraden (wegens overhead) als het kan vervangen worden door een simpele lus. We noemen dit staartrecursie (De recursieve versie).

Chapter 6

Verdeel-en-heers-algoritmen

6.1 Ontwerpstrategieën voor algoritmen

Brute kracht (brute force)

De brute-kracht-methode lost een probleem op door alle mogelijkheden uit te proberen. Zeer inefficiënt.

Tijd/ruimte-tradeoffs

Gebruiken van extra geheugenruimte om uitvoeringstijd te reduceren. Een voorbeeld hiervan is de frequentietabel.

Verdeel-en-heers (Divide-and-conquer)

Algoritmen volgens deze strategie bestaan uit twee gedeelten:

1. **Verdeel-fase**, oorspronkelijke probleem wordt opgesplitst in kleinere deelproblemen die recursief worden opgelost.
2. **Heers-fase**, waarbij oplossing voor het oorspronkelijke probleem wordt geconstrueerd uit de oplossingen van de deelproblemen.

De bekomen deelproblemen moeten disjunct zijn opdat deze methode efficiënt zouden zijn.

Verminder-en-heers (Decrease-and-conquer)

Variant op verdeel-en-heers, genaamd verminder-en-heers, hierbij wordt de oplossing van een probleem bepaald door slechts één gelijkaardig deelprobleem. Dikwijls hebben we hierbij te maken met een geval van staartrecursie. Er zijn hierop drie mogelijke variaties:

1. **Verminder met een constante**. De probleemgrootte wordt in elke stap verminderd met constante waarde.

2. **Verminder met een constante factor**. Probleemgrootte wordt in elke stap gedeeld door een constante factor, typisch 2.

3. **Verminder met een variabele grootte**. Probleemgrootte in elke stap verkleinen met of gedeeld door een waarde die kan variëren in de verschillende stappen.

Transformeer-en-heers (Transform-and-conquer)

Transformeren van oorspronkelijk probleem naar een ander probleem. Drie variaties van de techniek:

1. **Vereenvoudiging van het probleem**. Het probleem wordt getransformeerd naar een eenvoudiger of meer geschikte variant van hetzelfde probleem. Bijvoorbeeld: input probleem vooraf sorteren.

2. **Verandering van voorstelling**. Veranderen naar een andere representatie van het probleem.

3. **Reductie van het probleem**. Het probleem wordt getransformeerd naar een ander probleem waarvoor reeds een algoritme gekend is. Bijvoorbeeld: de klassieke problemen uit de grafentheorie.

6.2 Zoeken in rijen

In een gesorteerde rij kan sneller gezocht worden met behulp van de **binaire zoekmethode.**

6.2.1 Het sequentiële zoekalgoritme

Eenvoudige oplossing overloopt een voor een de objecten en vergelijkt ze met het gezochte element. Dit wordt herhaald totdat het element gevonden is, of totdat de gehele rij overlopen is.

Sentinel of 'schildwacht' Dit voorkomt de nood aan twee te controleren condities bij een lus over een rij. Het gezochte element wordt achteraan de lijst toegevoegd zodat het element zeker gevonden wordt.

Bij een gesorteerde rij kunnen hieraan enkele verbeteringen worden aangebracht. Het kan bijvoorbeeld worden stopgezet wanneer een element bereikt is dat groter is dan het te zoeken element.

Bij het sequentiële zoeken is de slechtste uitvoeringstijd $\Theta(n)$

6.2.2 De binaire zoekmethode

We kunnen een willekeurig element nemen uit een gesorteerde rij. Dit element vergelijken we met het gezochte element. Afhankelijk hiervan is het element gevonden, of moeten we links/rechts van het willekeurige element gaan zoeken vanwege het gesorteerd zijn van de rij. Als willekeurig element neemt men meestal het element dat in het midden van de rij staat, namelijk het element $k = \lfloor (i+j)/2 \rfloor$.

Stelling 6.2.1 *De binaire zoekmethode heeft uitvoeringstijd* $T(n) = \Theta(logn) in het slechtste geval$

Bewijs. De recurrente betrekking voor de uitvoeringstijd wordt gegeven door $T(n) = T(n/2) + \Theta(1)$. Steunend op de masterstelling, is de oplossing hier van $T(n) = \Theta(logn)$. De binaire zoekmethode is dus een logaritmisch algoritme.

6.3 Het probleem van de maximale deelrijsom

Eigenschap 6.3.1. *Voor willekeurige* $i \geq 0$*, als* $a_i, ..., a_j$ *de eerste deelrij is waarvoor de som negatief wordt, dan is voor elke* $i \leq p \leq j$ *en elke* $q \geq p$*, de deelrij* $a_p, ..., a_q$ *ofwel geen maximale deelrij, ofwel een reeds geziene maximale deelrij.*

Bewijs. Voor p = i volgt het gestelde onmiddelijk uit bovenstaande observatie. Voor $p > i$ is de beschouwde deelrij ofwel van de vorm $a_i, ..., a_p, ..., a_j, ..., a_q$ ofwel van de vorm $a_i, ..., a_p, ..., a_q, ..., a_j$. Aangezien j de eerste index is waarvoor de som negatief wordt, is de som van $a_i, ..., a_{p-1}$ positief en is dus de som van $a_p, ..., a_q$ kleiner dan of gelijk aan de som van $a_i, ..., a_q$. In het eerste geval, als $j < q$, weten we reeds dat de deelrij $a_i, ..., a_q$ geen maximale deelrij is. Anders is $a_i, ..., a_q$ een reeds geziene deelrij met een grotere som.

On-line algoritmen Als de rij slechts eenmalig wordt doorlopen, het volstaat element per element te lezen, zonder de gegevens in het centrale geheugen in te lezen. Het heeft ook op elk moment de maximale deelrijsom van het reeds gelezen gedeelte van de inputrij.

6.4 Het probleem van het dichtste puntenpaar

Chapter 7

Sorteeralgoritmen

7.1 Kwadratische sorteeralgoritmen

7.1.1 Sorteren door omwisseling BubbleSort

BubbleSort Naast elkaar staande elementen worden vergeleken en verwisseld als ze niet in de goede volgorde staan. Het grootste element van de twee wordt naar achteren verschoven. Dit proces wordt herhaaldelijk uitgevoerd, tot n - 1 (telkens - 1, aangezien het laatste element van elke volledige iteratie op zijn juiste plaats staat). Dit wordt herhaald tot het voorlaatste element, want na n-1 fasen is de rij uiteindelijk gesorteerd.

Stelling 7.1.1 *BubbleSort heeft tijdscomplexiteit* $T(n) = \Theta(n^2)$

Bewijs De probleemgrootte n is de dimensie van de rij. De essentiële bewerkingen zijn vergelijkingen tussen rij-elementen en verwisselingen van rij-elementen. Het aantal vergelijkingen $C(n)$ in het BubbleSort is hetzelfde voor alle mogelijke rijen van lengte n, namenlijk: $C(n) = n(n-1)/2$. Het aantal verwisselingen $S(n)$ is afhankelijk van de inputrij, en kan variëren van $S_b(n) = 0$ in het beste geval tot $S_s(n) = n(n-1)/2$ in het slechtste geval. De totale complexiteit is dus $T(n) = \Theta(n^2)$

7.1.2 Sorteren door selectie SelectionSort

SelectionSort Is een rechtlijnig algoritme. Het grootste element inde rij wordt bepaald en achteraan geplaatst. Vervolgens wordt het tweede-grootste element bepaald en op de voorlaatste plaats gezet. Dit wordt herhaald op steeds kortere deelrijen, totdat de deelrij uiteindelijk maar ëën element meer bevat.

Stelling 7.1.2 *SelectionSort heeft tijdscomplexiteit* $T(n) = \Theta(n^2)$

Bewijs Het aantal vergelijkingen is $C(n) = n(n-1)/2$, want in elke stap van de dubbele for-lus gebeurt een vergelijking. Het aantal verwisselingen is hoogstens $S_s(n) = n-1$, hetgeen onmiddelijk duidelijk is uit de implementatie: de verwisseloperatie staat in de buitenste lus en kus hoogstens n-1 keer worden uitgevoerd. Het kan gebeuren dat er geen enkele verwisseling nodig is, nl. als de gegeven rij al gesorteerd is, m.a.w. $S_b(n) = 0$. De totale tijdscomplexiteit is dus $T(n) = \Theta(n^2)$

7.1.3 Sorteren door tussenvoegen InsertionSort

InsertionSort In de begintoestand is het eerste element op zichzelf beschouwd, gesorteerd. In de eindtoestand zijn alle elementen, als groep beschouwd, gesorteerd. De basisbewerking van het algoritme is het rangschikken van de elementen op de posities 1 t.e.m. i, waarbij i een waarde tussen 2 en n heeft. Daarbij wordt verondersteld dat de elementen op posities 1 t.e.m. i-1 reeds gesorteerd zijn en wordt het element op positie i op de juiste plaats tussengevoegd. Het algoritme bestaat uit een aantal fasen waarbij i achtereenvolgens waarden van 2 t.e.m. n aanneemt.

Stelling 7.1.3 *InsertionSort heeft een slechtste-geval-uitvoeringstijd $T_s(n) = \Theta(n^2)$ en een beste-geval-uitvoeringstijd van $T_b(n) = \Theta(n)$.*

Bewijs In het slechtste geval is het aantal stappen uitgevoerd door de dubbele for-lus gegeven door $n(n-1)/2$. Elke stap komt overeen met een vergelijking en een verwisseling dus de uitvoeringstijd in het slechtste geval is $\Theta(n^2)$. Als echter de rij bij het begin van het algoritme reeds gesorteerd is, dan is de uitvoeringstijd $\Theta(n)$, omdat de test bij het begin van de binnenste for-lus altijd faalt en de lus dus niet uitgevoerd wordt. Dit is ook het best mogelijke geval, want de buitenste lus heeft steeds n-1 stappen.

Gemiddelde uitvoeringstijd

Inversie Een inversie van een rij $(a_1, ..., a_n)$ is elk paar (i, j) waarvoor geldt dat $i < j$ maar $a_i > a_j$. *Het verwisselen van twee adjacente elementen die niet in de goede volgorde staan, vermindert het aantal inversies met precies ëën. Een gesorteerde rij heeft geen inversies.*

Gemiddelde uitvoeringstijd Om deze te berekenen gaan we uit van volgende veronderstellingen:

1. De rij bevat geen dubbels.
2. De rij beschouwen we als een permutatie van de eerste n natuurlijke getallen. We veronderstellen dat alle permutaties even waarschijnlijk zijn.

Stelling 7.1.4 *Het gemiddelde aantal inversies in een rij van n verschillende getallen is gegeven door $n(n-1)/4$*

Bewijs Voor een rij A noemen we A_r de rij in omgekeerde volgorde. Beschouw twee willekeurige getallen (x, y) in de rij waarvoor $y > x$. Dit paar correspondeert met een inversie in ofwel A ofwel A_r. Het totale aantal dergelijke paren voor een rij A (en zijn omgekeerde A_r) is gegeven door $n(n-1)/2$. Het aantal inversies in een gemiddelde rij is dus de helft hiervan, of $n(n-1)/4$.

Stelling 7.1.5 *InsertionSort heeft gemiddelde uitvoeringstijd* $T_g g(n) = \Theta(n^2)$.

Bewijs Merk op dat het aantal inversies in de te sorteren rij precies gelijk is aan het aantal keer dat in het algoritme de opdracht voor het verwisselen van a_j en $a_j - 1$ uitgevoerd wordt.

Dus, als er k inversies zijn bij de start van het algoritme, dan moeten er k (impliciete) verwisselingen gebeuren. Aangezien er verder $\Theta(n)$ ander werk nodig is in het algoritme, is de uitvoeringstijd van het sorteren door tussenvoegen gegeven door $\Theta(k + n)$, met k het aantal inversies in de oorspronkelijke rij.

Zoals in voorgaande stelling bewezen is het gemiddelde aantal inversies $\Theta(n^2)$, waaruit onmiddelijk volgt dat sorteren door tussenvoegen kwadratisch is in het gemiddelde geval.

Een ondergrens voor de uitvoeringstijd

Bottleneck Bij deze algoritmen is het feit dat dit algoritme slechts aangrenzende elementen met elkaar vergelijkt en/of verwisselt de bottleneck.

Stelling 7.1.6 *Om het even welk algoritme dat sorteert door het vergelijken en verwisselen van aangrenzende elementen, vereist gemiddeld* $\Omega(n^2)$*uitvoeringstijd.*

Bewijs Het gemiddelde aantal inversies bij het begin van het algoritme is $n(n-1)/4$. Elke verwisseling vermindert het aantal inversies met precies ëën, zodat dus $\Omega(n^2)$ verwisselingen vereist zijn.

7.2 MergeSort

MergeSort gebruikt recursie om te komen tot een efficiënt sorteeralgoritme. De te sorteren rij wordt opgesplitst in twee deelrijen die half zo lang zijn als de oorspronkelijke rij. Meer plrecies, als n de lengte van de oorspronkelijke rij voorstelt, dan hebben de deelrijen resp. lengte $\lfloor n/2 \rfloor$ en $\lceil n/2 \rceil$. Deze twee deelrijen worden recursief gesorteerd en vervolgens samengevoegd tot ëën enkele gesorteerde rij. Het samenvoegen van twee gesorteerde rijen wordt ook **merge** genoemd, vandaar de naam.

Stelling 7.2.1 *MergeSort heeft tijdscomplexiteit* $T(n) = \Theta(nlogn)$.

Bewijs MergeSort is een verdeel-en-heers algoritme, waarvan de uitvoeringstijd bepaald wordt door de recurrente betrekking $T(n) = 2T(n/2) + \Theta(n)$. De oplossing hiervan is $T(n) = \Theta(nlogn)$. Dit is zowel slechtste- als beste- en gemiddelde-geval-uitvoeringstijd, omdat het mergen altijd lineaire tijd kost.

7.3 QuickSort

QuickSort Doorgaans is dit de beste keuze voor een intern sorteeralgoritme. Dit is een zeer snel sorteeralgoritme. De gemiddelde uitvoeringstijd is $\Theta(nlogn)$ en de slechtst mogelijke uitvoeringstijd is $\Theta(n^2)$, maar de kans dat dit slechtste geval zich voordoet kan zeer klein worden gemaakt.

7.3.1 Het QuickSort-algoritme

Partitioneren Het partitioneert de te sorteren rij. Een willekeurig element s uit de rij wordt gekozen; dit element wordt de spil genoemd. Vervolgens wordt de rij opgesplitst in twee disjuncte deelrijen, nl. een deelrij L met elementen kleiner dan de spil en een deelrij R met elementen groter dan de spil. Deze twee deelrijen worden dan recursief gesorteerd, en aaneengeschakeld tot de uiteindelijke gesorteerde rij.

Stelling 7.3.1 *QuickSort sorteert de gegeven rij op correcte wijze.*

Bewijs Daartoe steunen we op de volgende vaststellingen. Het principe van recursie garandeert dat de groep L van de kleine elementen en de groep R van de grote elementen na de recursieve oproepen gesorteerd zijn. De kenmerkende eigenschap van de partitionering garandeert dat het grootste element van L niet groter is dan de spil en dat het kleinste element van R niet kleiner is dan de spil. hieruit kunnen we besluiten dat de uiteindelijk gevormde rij correct gesorteerd is.

7.3.2 Complexiteit van QuickSort

Best mogelijke partitionering Het best mogelijke geval voor QuickSort treedt op wanneer de spil de rij elementen opsplitst in twee deelrijen van gelijke grootte, en dat bij elke stap in de recursie. In dat geval: twee recursieve oproepen van halve probleemgrootte met lineaire overhead, hetgeen analoog is aan de situatie bij MergeSort, met als recurrente betrekking $T(n) = 2T(n/2) + \Theta(n)$. De best mogelijke uitvoeringstijd van QuickSort is dus: $T_b(n) = \theta(nlogn)$.

Slechtst mogelijke partitionering Deelproblemen van ongelijke grootte zijn ongunstig. Veronderstel: spil in elke stap het kleinste element van de deelrij, dan is de groep L van kleine elementen leeg terwijl de groep R van grote elementen de ganse deelrij behalve de spil bevat. De recurrente betrekking die de uitvoeringstijd in dit geval beschrijft, is $T(n) =$

$T(n-1) + \Theta(n)$. Gebruik maken van de iteratiemethode kunnen we hieruit afleiden dat de slechtst mogelijke uitvoeringstijd van QuickSort kwadratisch is:
$T_s(n) = \theta(n^2)$.

Gebalanceerde partitionering en het gemiddelde geval In het gemiddelde geval is de uitvoeringstijd $\Theta(nlogn)$.

7.3.3 Keuze van de spil

Aandachtspunt Het vermijden van de slechtste uitvoeringstijd $\Theta(n^2)$. Eerste of laatste element uit rij als spil wordt afgeraden, aangezien bij gesorteerde of omgekeerd gesorteerde rijen dit voor problemen zorgt. Het middelste element is een betere keuze, aangezien bij een gesorteerde rij, hier twee perfecte partities gevormd kunnen worden. Desondanks kan een kwadratische uitvoeringstijd nog steeds voorkomen, maar de kans om die te bekomen is zeer klein.

Mediaan-van-drie-partitionering Deze methode probeert een beter dan gemiddeld goede spil te kiezen met behulp van de mediaan[1]. De beste keuze van de spil zou dus uiteraard de mediaan zijn, maar het berekenen van deze mediaan kost ook tijd en dit zou het algoritme te veel vertragen. We kunnen dit oplossen door de mediaan van een deelgroep te bepalen. In de praktijk gebruikt men doorgaans volgende drie elementen: het eerste element, het laatste element en het middelste element uit de rij.

7.3.4 Het partitioneren van de rij

Basispartitionering Deze bestaat uit drie stappen. In de eerste stap wordt het spilelement achteraan geplaatst door het te verwisselen met het laatste element. (We veronderstellen voorlopig dat alle elementen verschillend zijn.) In de tweede stap worden alle elementen kleiner dan de spil naar het linkergedeelte van de rij gebracht, terwijl alle elementen groter dan de spil naar het rechtergedeelte worden gebracht. Dit gebeurt door van links naar rechts naar een element groter dan de spil te zoeken, via een huidige positie links die start bij het begin. Analoog zoeken we van rechts naar links naar een element kleiner dan de spil, via een huidige positie rechts. Deze twee elementen worden van plaats verwisseld. Dit wordt herhaalt tot wanneer de twee huidige posities links en rechts mekaar passeren in de rij. De derde stap bestaat uit het verwisselen van de spil met het eerste element groter dan de spil.

Bij sleutels gelijk aan de spil? Beide links en rechts moeten stoppen wanneer een sleutel gelijk aan de spil ontmoet wordt.

[1]De mediaan van een groep van n getallen is het $\lceil n/2 \rceil$-de kleinste getal.

Mediaal-van-drie-partitionering Het is het eenvoudigste manier om het eerste het middelste en laatste element te sorteren. Merk op: eerste element is $\leq$ spil en laatste element is $\geq$ spil, waardoor we de spil naar het voorlaatste element kunnen verplaatsen en links kan starten bij het tweede element en rechts bij het derde laatste element van de deelrij.

7.4 Lineaire sorteeralgoritmen

7.4.1 CountingSort

CountingSort Dit algoritme kan worden gebruikt voor het sorteren van een rij $(a_1, ..., a_n)$, waarbij de sleutels a_i gehele getallen zijn uit het interval $[0, k]$, voor een niet te grote waarde van k. Om de rij te sorteren telt het algoritme hoeveel keer elke waarde voorkomt; uit deze informatie kan de positie van elke sleutel in de gesorteerde rij worden berekend en kan de rij worden gesorteerd. Dit is een stabiel[2] sorteeralgoritme.

Stelling 7.4.1 *De uitvoeringstijd van CountingSort is* $T(n) = \Theta(n)$ *als* $k = O(n)$.

Bewijs Elke lus in het algoritme is ofwel $\Theta(n)$ ofwel $\Theta(k)$. De uitvoeringstijd van het algoritme is dus $T(n) = \Theta(n + k)$. Wanneer $k \leq n$, of meer algemeen, wanneer k een functie van n is waarvoor $k = O(n)$, dan wordt k verwaarloosbaar tegenover n en krijgen we dus een lineaire uitvoeringstijd $T(n) = \Theta(n)$ voor het algoritme.

7.4.2 RadixSort

RadixSort Dit is een algoritme dat kan worden gebruikt voor het sorteren van een rij $(a_1, ..., a_n)$ van positieve gehele getallen, die een beperkt aantal cijfers hebben.Om deze rij te sorteren wordt eerst een stabiel sorteeralgoritme gebruikt om de getallen te sorteren op hun laatste cijfer, vervolgens op hun voorlaatste cijfer, enzovoort, tot uiteindelijk gesorteerd wordt op het eerste cijfer.

Stelling 7.4.2 *RadixSort sorteert de gegeven rij op correcte manier.*

Bewijs Zij m het aantal cijfers in de getallen. Zij x een sleutel uit de te sorteren rij. We bewijzen dat, wanneer het algoritme stopt, elke sleutel met waarde groter dan x in de rij na x komt, waaruit volgt dat de rij gesorteerd is.
Beschouw de decimale voorstelling van $x = x_{m-1}10^{m-1} + ... + 10x_1 + x_0$ en van een element $y = y_{m-1}10^{m-1} + ... + 10y_1 + y_0$ uit de rij, en onderstel dat $y > x$. Zij l de grootste index waarvoor $x_l \neq y_l$. Aangezien $y > x$ is dus $y_l > x_l$. Wanneer CountingSort sorteert met

[2]Een sorteeralgoritme wordt stabiel genoemd als elementen die dezelfde sleutel hebben (de sleutel is dat kenmerk van een element dat wordt vergeleken met de sleutel van een ander element om de volgorde te bepalen) niet bij het sorteren ten opzichte van elkaar van volgorde veranderen. *Bron: Wikipedia*

het cijfer corresponderend met 10^l als sleutel, wordt y dus achter x geplaatst. Bij volgende uitvoeringen van CountingSort is telkens $x_i = y_i$ $(i > l)$ en aangezien CountingSort stabiel is, blijft y in de rij achter x geplaatst.

Stelling 7.4.3 *De uitvoeringstijd van RadixSort is $T(n) = \Theta(n)$, wanneer het aantal cijfers m begrensd is door een constante.*

Bewijs De uitvoeringstijd van het sorteren in elke stap van de for-lus is $\Theta(n+9)$, dus $\Theta(n)$. Aangezien de for-lus m stappen heeft, is de totale complexiteit dus $T(n) = \Theta(mn)$. Wanneer we veronderstellen dat $m = \Theta(1)$, dan is de uitvoeringstijd van RadixSort $T(n) = \Theta(n)$.

Chapter 8

Grafen

8.1 Grafen

8.1.1 Wat is een graaf

Grafen Grafen bestaan bestaan uit **toppen** en **bogen** en een **incidentierelatie** hiertussen. De toppen en bogen kunnen bijkomende attributen hebben zoals kleur of gewicht.

Grafen, toppen en bogen Een graaf G = (V(G), E(G)) bestaat uit een eindige verzameling V(G) van objecten, toppen genoemd, en een verzameling E(G) van paren van elementen uit V(G), bogen genoemd. V(G) noemen we de **toppenverzameling** van G genoemd en E(G) de **bogenverzameling**. Elke boog is geassocieerd met een verzameling van twee toppen, **eindpunten** genaamd. Een boog **verbindt** zijn eindpunten. Het aantal toppen in G noemen we de **orde** van de graaf en het aantal bogen noemen we de **grootte** ≈ een (n, m)-graaf.

Stelling 8.1.1 *Wanneer G een (n,m)-graaf is, dan geldt dat* $m \leq n(n-1)/2$.

Bewijs Dit volgt onmiddelijk uit het feit dat er $n(n-1)/2$ mogelijke paren van elementen van V(G) zijn.

Adjacent Adjacente toppen zijn twee toppen, verbonden door een boog. We noteren: u ∼ v. Adjacente bogen zijn twee bogen die een eindpunt gemeen hebben. Als een top v een eindpunt is van een boog e, dan noemen we v **incident** met e en vice versa.

Dichte graaf Als de meeste bogen aanwezig zijn, dan is $|E| = \Theta(|V|^2)$, dan spreken we van een dichte graaf. In de meeste gevallen is de graaf eerder **ijl**, dan is $|E| = \Theta(|V|)$ of slechts iets meer.

Simpele grafen, multigrafen en pseudografen Multigraaf: toppen kunnen door meer dan ëën boog verbonden worden. We noemen deze twee of meer bogen, **parallelle bogen**. Een collectie van parallelle bogen noemen we een **multiboog**. **Zelflus**: top die met zichzelf is verbonden. Indien een boog geen zelflus is, dan wordt die een **eigenlijke boog** genoemd. Als we zelflussen en multibogen toelaten bekomen we een **pseudograaf**. Een graaf zonder deze bogen noemen we een **simpele graaf**.

Gerichte grafen en gewogen grafen Een **gerichte boog of pijl** is een boog waarvan een van de eindpunten als **kop** wordt aangeduid, en de andere als **staart**. De pijl is gericht van zijn kop naar zijn staart. Multipijl: verzameling van twee of meer pijlen die dezelfde kop en dezelfde staart hebben. Een **gerichte graaf** is een graaf waarvan alle bogen gericht zijn. Een **gedeeltelijk gerichte graaf** bevat gerichte en ongerichte bogen. Een **onderliggende graaf** van een (gedeeltelijk) gerichte graaf is de graaf die we bekomen door de richtingen op de bogen te verwijderen.
Anders kan het ook nuttig zijn om een attribuut aan de bogen van een graaf te hechten, *Bijvoorbeeld de afstand tussen twee steden.* Een **gewogen graaf** is een graaf waarbij elke boog een getal toegekend is. We noemen dat getal **het gewicht**.

8.1.2 De graad van een top

Nabuurschap Het nabuurschap van een top v van een graaf G wordt gedefinieerd als $N_G(v) = \{u \in V(G) | vu \in E(G)\}$. De **graad** $deg_G(v) = |N_g(v)|$, het aantal toppen adjacent met v, of aantal bogen incident met v. Een top met graad 0 is een **geïsoleerde top**. Een top met graad 1 is een **eindtop**. Een **even top** is een top met een even graad en idem voor een **oneven top**. De **gradenrij** van een graaf G, is de rij gevormd door de graden van de toppen in stijgende volgorde te rangschikken. De kleinste graad is de **minimumgraad** en doorgaans genoteerd als $\delta(G)$. De grootste waarde is de **maximumgraad** van G en genoteerd als $\Delta(G)$.

Stelling 8.1.2 *Een graaf G van orde $n > 1$ heeft minstens één paar toppen waarvan de graden gelijk zijn.*

Bewijs Het is gemakkelijk in te zien dat voor een graaf G van orde n en een top v van G geldt dat $0 \leq deg(v) \leq n - 1$. Er zijn dus n mogelijke waarden voor de graad van een top, nl. 0, ..., n-1. Er kan echter niet zowel een top van graad 0 als een top van graad n-1 zijn, omdat de aanwezigheid van een top van graad 0 impliceert dat elk van de n-1 andere toppen met hoogstens n-2 toppen adjacent kan zijn. De n toppen van G kunnen dus hoogstens n-1 mogelijke waarden voor hun graad realiseren. Gebruik makend van het *pigeonhole principle* volgt hieruit dat minstens twee van de n toppen dezelfde graad hebben.

Stelling 8.1.3 (Euler) *Zij G een graaf met orde n en een grootte m, en zij $V(G) = \{v_1, ..., v_n\}$. Dan geldt dat $\Sigma_{i=1}^{n} deg(v_i) = 2m$.*

Bewijs Wanneer de som van de graden van de toppen berekend wordt, wordt elke boog tweemaal meegerekend, nl. eenmaal voor elk van zijn twee incidente toppen.

Ingraad, Uitgraad De ingraad van een top v in een gerichte graaf G is het aantal pijlen dat naar v gericht is. De uitgraad van v is het aantal pijlen dat vanuit v gericht is.

Stelling 8.1.4 *In een gerichte graaf G zijn de som van de ingraden en de som van de uitgraden allebei gelijk aan het aantal bogen van G.*

Bewijs Elke gerichte boog e draagt één bij aan de ingraad van de staart van e en één aan de uitgraad van de kop van e.

8.1.3 Paden en samenhangendheid

Wandeling Een wandeling van top v_0 naar top v_l is een rij met afwisselend toppen en bogen, van de vorm $W = (v_0, e_1, v_1, e_2, ..., v_{l-1}, e_l, v_l)$, zondanig dat de eindpunten van elke boog e_i uit de rij v_{i-1} en v_i zijn, voor elke $i = 1, ..., l$. In een gerichte graaf is dit een **gerichte wandeling** van v_0 naar v_l, en een rij met afwisselend toppen en pijlen. Een (gerichte) wandeling van een top x naar een top y wordt ook een (gerichte) x-y-wandeling genoemd. Als $x = y$, dan noemen we dit een **gesloten**, anders een **open** wandeling.

Spoor Een spoor is een wandeling waarin geen herhalende bogen voorkomen. Een gesloten spoor noemen we een **circuit**.

Pad Een pad is een wandeling waarin geen herhalende toppen voorkomen, behalve evt. begin en eindpunt. Een gesloten pad noemen we een **cykel**.

Gericht We hebben eveneens een gericht spoor en een gericht pad.

Lengte/Taille Lengte van een wandeling, spoor of pad is het aantal bogen hierin. De taille van een graaf G is de lengte van de kortste cykel in G.

Euleriaans spoor Dit is een open spoor dat elke boog van G bevat, een **euleriaans circuit** is een gesloten spoor dat elke boog van G bevat. Een **euleriaanse graaf** is een graaf die een euleriaans circuit bevat.

Hamiltoniaans pad Is een pad dat elke top van G aandoet. Een **hamiltoniaanse cykel** is een cykel die elke top van G aandoet. Een **hamiltoniaanse graaf** is een graaf die een hamiltoniaanse cykel bevat.

Een top v van een graaf G is **bereikbaar vanuit** een top u van G als er een wandeling van u naar v is. Een graaf G is **Ssamenhangend** als er voor elk paar toppen u en v een wandeling van u naar v is, m.a.w. als elke top bereikbaar is vanuit elke andere top.

Twee toppen u en v in een gerichte graaf D zijn **wederzijds bereikbaar** als D een gerichte u-v-wandeling een een gerichte v-u-wandeling bevat. Een gerichte graaf is **sterk samengangend**als elke twee toppen ervan wederzijds bereikbaar zijn. Een gerichte graaf is **zwak samenhangend** als de onderliggende graaf samenhangend is.

8.1.4 Bomen en wouden

Acyclische graaf Dit is een graaf waarin geen cykels optreden. We noemen zo'n graaf een **woud**. Een **boom** is een samenhangende graaf die geen cykels bevat. Een **gerichte acyclische graaf, DAG** is een gerichte graaf waarin geen gerichte cykels optreden.

8.1.5 Deelgrafen

Deelgraaf Is een graaf H van een graaf G als $V(H) \subseteq V(G)$ en $E(H) \subseteq E(G)$. Een **echte deelgraaf** H van een graaf G is een deelgraaf waarvoor de toppenverzameling V(H) een echte deelverzamelig is van V(G). Het **gewicht** w(H) van een deelgraaf H van een gewogen graaf G is de som van de gewichten van de bogen van H.

Zij S een niet-ledige deelverzameling van V(G). De **deelgraaf geïnduceerd door** S, genoteerd als $< S >$, is de maximale deelgraaf van G met toppenverzameling S, m.a.w. $< S >$ bevat precies die bogen van G die toppen van S verbinden. Het is een **top-geïnduceerde deelgraaf, of geïnduceerde deelgraaf**, als H : $< S >$ voor zekere niet-ledige deelverzameling S van de toppen van G.

Een **deelgraaf geïnduceerd door** een niet-ledige deelverzameling X van bogen van G, genoteerd als $< X >$, is de minimale deelgraaf van G met bogenverzameling X, m.a.w. $< X >$ bestaat uit die toppen van G die incident zijn met ten minste één boog van X. een deelgraaf H van een graaf G is een **booggeïnduceerde deelgraaf** als H = $< X >$ voor zekere niet-ledige verzameling X van bogen van G.

Opspannend Een deelgraaf H van een graaf G wordt een **opspannende deelgraaf** van G genoemd als V(H) = V(G). Een **opspannend woud** van een graaf G is een opspannende deelgraaf van G die een woud is. Een **opspannende boom** van een graaf G is een opspannende deelgraaf van G die een boom is.

Stelling 8.1.5 *Een graaf is samenhangend als en slechts als hij een opspannende boom bevat.*

Bewijs Veronderstel dat G een opspannende boom T bevat. Per definitie is T samenhangend, dus is er een pad tussen elk paar toppen van T, en dus ook tussen elk paar toppen van G.

Veronderstel dat G samenhangend is. Als G geen boom is, bevat G minstens één cykel. Wanneer we uit dergelijke cykel een boog verwijderen, blijft de graaf nog steeds samenhangend. Op die manier kunnen we uit elke cykel een boog verwijderen tot we een acyclische graaf bekomen. Deze is nog steeds samenhangend en is dus een opspannende boom van de oorspronkelijke graaf.

Component De component van een graaf G is een maximale samenhangende deelgraaf van G.

8.1.6 Scharnierpunten en bruggen

Schrapping Zij G een graaf. De schrapping van een echte deelverzameling S van toppen uit G is de deelgraaf die bestaat uit de toppen van G die niet tot S behoren, en de bogen van G die niet incident zijn met een top in S. We noteren deze deelgraaf als G - S. Als S bestaat uit één enkele top v, dan noteren we G -v.
De schrapping van een verzameling X van bogen van een graaf G, genoteerd als G-X, is de opspannende deelgraaf van G die bekomen wordt door het verwijderen van de bogen van X uit E(G).

Toevoegen We kunnen ook paren van niet-adjacente toppen van een graaf G toevoegen aan de graaf.

Toppensnede Zij G een samenhangende graaf, een **toppensnede** in G is dan een toppenverzameling S zodanig dat G-S niet samenhangend is. Een **scharnierpunt** is een toppensnede bestaande uit n enkele top.

Bogensnede Een bogensnede in G is een verzameling bogen X, zodanig dat G - X niet samenhangend is, en G - X' wel samenhangend is, voor elke $X' \subset X$. Een bogensnede X partitioneert de toppen van G in twee disjuncte verzamelingen V_1, V_2, zodanig dat elke boog uit X een eindtop in V_1 en een eindtop in V_2 heeft. Een **brug** is een bogensnede bestaande uit één enkele boog.

Stelling 8.1.6 *Een boog e van een samenhangende graaf G is een brug van G als en slechts als e niet op een cykel in G ligt.*

Bewijs Is de boog e een brug, dan moeten zijn toppen noodzakelijkerwijs in verschillende componenten liggen als we e wegnemen uit G. Dit is duidelijk niet het geval als e op een cykel ligt. Omgekeerd, onderstel nu dat de boog e geen brug is. Nemen we e weg, dan bekomen

we een samenhangende graaf. In het bijzonder zijn de eindtoppen v en w van e verbonden met elkaar door een pad dat de boog e niet bevat. Toevoegen van e aan het pad geeft ons een cykel die e bevat.

8.1.7 Speciale families van grafen

Complete graaf Dit is een graaf waarbij elk paar toppen verbonden is door een boog. Een complete graaf met n toppen noteren we als K_n.

Bipartiete graaf Dit is een graaf waarvan de toppenverzameling V kan worden opgesplitst in twee deelverzamelingen V_1 en V_2 zodanig dat elke boog van G een eindpunt in V_1 en een eindpunt in V_2 heeft. Het paar (V_1, V_2) wordt de **bipartitie** van G genoemd, en V_1, V_2 zijn de **bipartitieverzamelingen**

Complete bipartiete graaf Dit is een bipartiete graaf waarbij elke top van de ene bipartitieverzameling verbonden is met elke top van de andere bipartitieverzameling. Zo'n graaf met p toppen en q toppen in respectievelijk V_1, V_2 noteren we als $K_{p,q}$.

Regulier Een graaf G is r-regulier of regulier van graad r, als elke top van G graad r heeft, $r > 0$.

Padgraaf Een padgraaf P_n met n toppen is een samenhangende graaf met $|V(P_n)| = |E(P_n)| + 1$, die getekend kan worden zodanig dat al zijn toppen en bogen op één enkele rechte lijk liggen.

Cykelgraaf Is een samenhangende graaf C_n met n toppen en met $|V(C_n)| = |E(C_n)|$ die zodanig kan worden getekend dat al zijn toppen en bogen op een cirkel liggen.

Hyperkubus Is een d-reguliere graaf Q_d, waarvan de toppenverzameling bestaat uit de bitstrings van lengte d, zodanig dat er een boog is tussen twee toppen als en slechts als ze in slechts één bit verschillen.

8.2 Voorstellen van grafen

8.2.1 Adjacentiematrixvoorstelling

Adjacentiematrix Zij $G = (V, E)$ een ongewogen, ongerichte graaf met toppenverzameling $V = \{v_1, v_2, ..., v_n\}$. De adjacentiematrix $A = [a_{ij}]$ van G is de n x n matrix gedefinieerd door:

$$a_{ij} = \begin{cases} 1, als\ v_i v_j \in E, \\ 0, anders. \end{cases}$$

Gerichte grafen Eerder genoemd kostenmatrix. Hierin wordt het gewicht van de boog v_iv_j weergegeven op positie a_{ij}.

Voor multigrafen en pseudografen $a_{ij} = \begin{cases} \text{het aantal bogen tussen } v_i, v_j, als\ v_i \neq v_j, \\ \text{het aantal zelflussen in } v_i, als\ v_i = v_j. \end{cases}$

Geheugen Deze matrix neemt n^2 geheugen in, wat soms een verspilling is, vandaar volgende voorstelling.

8.2.2 Adjacentielijstvoorstelling

Adjacentielijstvoorstelling Associeert met elke top van G een lijst van toppen die ermee adjacent zijn. *Bijvoorbeeld bij een gerichte graaf: lijst met uitburen of bij gewogen graaf gewicht ook bijhouden.*

Geheugen Voor deze voorstelling zijn er $\Theta(n+m)$ geheugenplaatsen nodig, met n toppen en m bogen van een graaf G.

8.3 Euleriaanse grafen

8.3.1 Stadswandeling in Königsberg

8.3.2 Karakterisatie van euleriaanse grafen

Stelling 8.3.1 *Een samenhangende multigraaf G is euleriaans als en slechts als de graad van elke top even is.*

Bewijs Veronderstel dat G een euleriaanse multigraaf is. Dan bevat G een euleriaans circuit C, dat begint en eindigt in een top v. We tonen aan dat elke top van G even graad heeft. Beschouw eerst een top $u \neq v$. aangezien u noch de eerste noch de laatste top van C is, wordt de top u bij ieder voorkomen in C binnengekomen door een bepaalde boog en verlaten door een andere boog. M.a.w. ieder voorkomen van u in C draagt precies 2 bij tot de graad van u, zodat u even graad heeft. Voor de top v draagt elk van de voorkomens aan het begin en het einde van het circuit C precies 1 bij tot de graad van v, zodat dus ook de top v even graad heeft.

Omgekeerd, veronderstel dat elke top van G even graad heeft. We tonen aan dat G euleriaans is door een euleriaans circuit te construeren. Selecteer een top v van G en start een spoor T in v. We bouwen dit spoor zo ver mogelijk op, totdat we een top w bereiken zodanig dat de enige bogen incident met w reeds tot T behoren. We beweren dat $w = v$. Veronderstel dat $w \neq v$. Iedere keer als w optreedt in T vóór de laatste keer wordt één

boog gebruikt om w binnen te komen en wordt een andere boog gebruikt om w te verlaten. De voorkomens van w in T vóór de laatste keer komen dus overeen met een even aantal bogen incident met w. Wanneer w echter voor de laatste keer in T optreedt, wordt slechts één boog incident met w gebruikt. M.a.w. het spoor T bevat een oneven aantal bogen incident met w. Aangezien w even graad heeft, moet er dus minstens één boog incident met w in T voorkomen. Dus, de bewering dat w = v is correct, en T is dus eigenlijk een circuit. Als T alle bogen van G bevat, dan is T een euleriaans circuit en is G dus en euleriaanse graaf.

Veronderstel nu dat T niet alle bogen van G bevat. Aangezien G samenhangend is, bestaat er een top u in T die incident is met bogen die niet tot T behoren. Beschouw de multigraaf H bekomen door de bogen van T uit G te verwijderen. Aangezien T niet alle bogen van G bevat , is de multigraaf H niet-leeg. Bovendien is elke top van T incident met een even aantal bogen van T, zodat elke top in H ook even graad heeft. Zij H_1 de component van H die de top u bevat. Wanneer we een spoor T' in u beginnen, en dit zo ver mogelijk opbouwen, dan bekomen we net zoals voordien dat T' eindigt in u, zodat T' een circuit is. Wanneer we het circuit T' tussenvoegen in T op een plaats waar u voorkomt, dan bekomen we een circuit T_1 dat begint en eindigt in v, en dat meer bogen dan T bevat.

Als T_1 alle bogen van G bevat, dan is T_1 een euleriaans circuit en is G een euleriaanse multigraaf. In het andere geval, wanneer T_1 niet alle bogen van T bevat, dan herhalen we bovenstaande procedure totdat we een euleriaans circuit bekomen.

Stelling 8.3.2 *Een samenhangende multigraaf G bevat een euleriaans spoor als en slechts als G precies twee toppen met oneven graad heeft. Bovendien begint het euleriaans spoor dan in een van de toppen met oneven graad en eindigt het in de andere top met oneven graad.*

Chinese postbodeprobleem De **kost** is het totale extra bogen die worden toegevoegd in een graaf om een euleriaans te krijgen.

8.4 Hamiltoniaanse grafen en TSP

8.4.1 Hamiltoniaanse grafen

Onhandelbaar Het bepalen van een hamiltoniaanse cykel in een algemene graaf is een onhandelbaar probleem. Er is dus geen goed algoritme voor gekend.

8.4.2 Handelreizigersprobleem (TSP)

Zoekt een hamiltoniaanse cykel in een graaf waarvoor de totale boogkost minimaal is.

8.4.3 Algoritmen voor TSP

Vereist berekenen van het gewicht van $(n-1)!/2$ hamiltoniaanse cykels van G, als volgt. Een hamiltoniaanse cykel kan worden voorgesteld als een opeenvolging van toppen, startend met top 1, die alle toppen bevat. Een oplossing voor het handelsreizigersprobleem wordt bekomen door alle permutaties van dergelijke opeenvolgingen van toppen te genereren, de lengte van de corresponderende cykels te berekenen en de kortste hiervan te bepalen.

8.5 Gerichte acyclische grafen

8.5.1 Topologisch sorteren van een DAG

Topologische ordening Een topologische ordeningen in een DAG is een ordening van de toppen van een gerichte acyclische graaf zodanig dat, als er een pad is van u naar v, dan v in de ordening ná u komt. Het **topologisch sorteren** is het bepalen van een topologische ordening.

Chapter 9

Bomen

9.1 Bomen

9.1.1 Eigenschappen van bomen

Bomen Zijn de eenvoudigste samenhangende grafen. Is een samenhangende graaf die geen cykels bevat. Hieruit volgt onmiddelijk (Stelling 8.1.6) dat elke boog van een boom een brug is.

Stelling 9.1.1 *Zij G een graaf met n toppen, dan zijn volgende uitspraken equivalent.*

1. *G is een boom.*
2. *G heeft geen cykels en bevat n - 1 bogen.*
3. *G is samenhangend en bevat n - 1 bogen.*
4. *G is samenhangend en elke boog is een brug.*
5. *Elke twee toppen van G zijn door precies één pad verbonden.*
6. *G bevat geen cykels en voor elke nieuwe boog e bevat G + e precies één cykel.*

Bewijs Wanneer $n = 1$ dan zijn alle uitspraken triviaal waar, zodat we veronderstellen dat $n \geq 2$.

$1 \Rightarrow 2$: Dat een boom geen cykel heeft, volgt onmiddellijk uit de definitie van bomen. Dat een boom $n - 1$ bogen heeft, bewijzen we als volgt door sterke inductie op het aantal toppen. K_1 is de enige boom met 1 top en heeft 0 bogen, zodat het gestelde geldt voor $n = 1$. Zij $k \geq 2$ een natuurlijk getal en veronderstel dat het gestelde geldt voor elke boom met minder dan k toppen. Zij T een boom met $n = k$ toppen en m bogen, en zij e een boog van T. We hebben reeds opgemerkt dat e een brug van T is, zodat T - e een woud

met twee componenten is. Noteren we de twee componenten van $T - e$ door T_1 en T_2, met resp. n_1 en n_2 toppen, en m_1 en m_2 bogen. Aangezien $n_i < k$, voor $i = 1, 2$, weten we uit de inductiehypothese dat $m_i = n_i - 1$, voor $i = 1, 2$. Aangezien $n = n_1 + n_2$ en $m = m_1 + m_2 + 1$, geldt dat $m = (n_1 - 1) + (n_2 - 1) + 1 = n_1 + n_2 - 1 = n - 1$. Door inductie volgt het gestelde.

$2 \Rightarrow 3$: Voor elke samenhangende component met n_i toppen van G geldt per definitie dat deze een boom is en dus $n_i - 1$ bogen heeft. Als er zo c componenten zijn, dan hebben we in het totaal $n-c$ bogen. Dus $n-c = n-1$, waaruit $c = 1$ volgt. Aldus is G samenhangend.

$3 \Rightarrow 4$: Is een bepaalde boog geen brug, dan bekomen we door hem te verwijderen een samenhangende graaf met n toppen en $n - 2$ bogen. Door de $n - 2$ bogen allemaal te verwijderen kunnen we in totaal ten hoogste $n - 2 + 1 = n - 1$ samenhangende componenten verkrijgen, strijdig met het feit dat we er n hebben (namelijk, de n toppen). Dus elke boog is een brug.

$4 \Rightarrow 5$: Daar G samenhangend is, zijn twee toppen u en v door ten minste één pad verbonden. Veronderstel nu dat G twee $u - v$-paden bevat, die we noteren door P en Q. Aangezien P en Q verschillende $u - v$-padenzijn, moet er een top x bestaan (eventueel is $x = u$) die zowel tot P als tot Q behoort, zodanig dat de volgende top na x op P verschilt van de top na x op Q. Zij y de eerste top na x op P die ook tot Q behoort (eventueel is $y = v$). Dit levert twee $x - y$-paden die enkel x en y gemeenschappelijk hebben. Deze twee paden produceren een cykel C in G. Geen enkele boog van de cykel C is een brug, strijdig met de gegevens. Dus elke twee toppen u en v worden door een uniek pad verbonden.

$5 \Rightarrow 6$: Indien G een cykel bevatte, dan zouden twee willekeurige toppen van die cykel door twee verschillende paden worden verbonden, een strijdigheid. We voegen nu een boog uv toe en noemen de nieuwe graaf G'. Daar er een u-v-pad bestond in G bekomen we een cykel in G' door de boog uv aan dat pad toe te voegen. Daar G geen cykels bevat, moet elke cykel in G' de boog uv bevatten en een u-v-pad in G (want dit u-v-pad kan de boog uv niet meer bevatten). Maar in G is dit pad uniek. We besluiten dat de cykel uniek is.

$6 \Rightarrow 1$: We moeten aantonen dat G samenhangend is. Zijn er ten minste twee samenhangende componenten G_1 en G_2 (die per definitie allebei bomen zijn), dan voegen we een boog doe die een top in G_1 verbindt met een top in G_2. Deze boog is in de nieuwe graaf duidelijk een brug, dus niet bevat in een cykel, strijdig met de onderstellingen. Er is dus maar één samenhangende component en bijgevolg is G een boom.

Eigenschap 9.1.2 *Wanneer u en v twee niet-adjacente toppen in een boom T zijn, dan bevat $T + uv$ precies één cykel C, die de boog uv moet bevatten. Wanneer een boog e uit C verwijderd wordt, dan wordt opnieuw een boom bekomen.*

9.1.2 Gewortelde bomen

Gerichte boom Dit is een gerichte graaf, waarvan de onderliggende graaf een boom is.

Gewortelde boom Dit is een gerichte boom die een speciale top r bevat, de **wortel** van de boom genoemd, zodanig dat er voor elke top v van de boom een r-v-pad bestaat.

Stelling 9.1.3 *Een gerichte boom T is een gewortelde boom als en slechts als T een top r bevat met ingraad 0, terwijl alle andere toppen v van T ingraad 1 hebben.*

Bewijs Onderstel dat de gerichte boom T geworteld is. De wortel r heeft ingraad 0, want als v een top is die adjacent is met r en vr zou een gerichte boog zijn met kop v, dan is er geen gericht r-v-pad, strijdig met het feit dat r de wortel is. Zij nu v een willekeurige top verschillend van r. Daar er een gericht r-v-pad bestaat, heeft v al ingraad minstens 1. Als nu w een top is adjacent met v en wv is een gerichte boog met kop w, dan beschouwen we een gericht r-w-pad. Daar de onderliggende graaf een boom is, moet ofwel het r-w-pad de top v bevatten (strijdig met de onderstelling dat de boog wv als kop w heeft), ofwel moet het r-v-pad de top w bevatten. Aldus is w uniek en heeft v inderdaad ingraad 1. Onderstel vervolgens dat er een top r is met ingraad 0, en dat elke andere top ingraad 1 heeft. Zij v een willekeurige top. We bewijzen door middel van inductie op de afstand $d(r, v)$ det het unieke r-v-pad in de onderliggende boom gericht is. Dit is duidelijk voor $d(r, v) = 1$, daar de ingraad van r gelijk aan 0 is. Is w de unieke top van het r-v-pad adjacent aan v, dan hebben we een gericht r-w-pad door de inductiehypothese. De boog vw kan onmogelijk v als kop hebben, want dan zou w ingraad minstens 2 hebben. Dus vw is gericht van w naar v en we hebben een gericht r-v-pad.

Gevolg 9.1.4 *Elke gewortelde boom heeft een unieke wortel.*

Niveau Een top x in een gewortelde boom T met wortel r bevindt zich op niveau of **diepte** i als en slechts als het r-x-pad in T lengte i heeft. Het grootste natuurlijke getal h waarvoor er een top op niveau h in een gewortelde boom T is, wordt de **diepte van** T genoemd.

Hoogte De hoogte van een top x in een gewortelde boom T met wortel r is de lengte van het pad van x naar zijn diepste nakomeling die een blad is. De **hoogte van** T is de hoogte van de wortel r.

Kind / Ouder Zij T een gewortelde boom. Wanneer een top v van T adjacent is met u, waarbij u in het niveau onder v ligt, dan wordt u een kind van v genoemd, en v is de ouder van u. Een top w is een **afstammeling** van een top v en v is voorouder van w als het v-w-pad in T onder v ligt. Een top zonder kinderen wordt een **blad** genoemd. De andere toppen, m.a.w. de toppen die wel kinderen hebben, worden de **interne toppen** van de gewortelde boom genoemd.

9.1.3 Binaire bomen

Binaire boom K-aire boom: gewortelde boom waarbij elke top ten hoogste k kinderen heeft. We noemen die boom een complete k-aire boom, als elke top ofwel k kinderen ofwel geen kinderen heeft. Een **binaire boom** is een 2-aire boom waarin één kind als **linkerkind** van zijn ouder beschouwd wordt en het andere als **rechterkind**. Idem: **complete binaire boom**.

Lemma 9.1.5. *Een binaire boom bevat ten hoogste 2^i toppen van niveau $i (i \geq 0)$.*

Bewijs Het bewijs verloopt door inductie op i. De wortel is de enige top van niveau $i = 0$, zodat het resultaat waar is voor $i = 0$. De inductiehypothese is dat het resultaat waar is voor het niveai i, m.a.w. het maximum aantal toppen van niveau i is 2^i. Aangezien elke top van niveau i in een binaire boom ten hoogste van graad 2 is, is het maximum aantal toppen van niveau $i + 1$ tweemaal zo groot als het maximum aantal toppen van niveau i, namelijk $2 * 2^i = 2^{i+1}$.

Stelling 9.1.6 *Voor een binaire boom T van hoogte h met n toppen is $n \leq 2^{h+1} - 1$*

Bewijs Wegens Lemma 9.1.5 is het maximum aantal toppen op niveau i in een binaire boom gegeven door 2^i. Het maximum aantal toppen in een binaire boom van hoogte $h \geq 0$ is dus gelijk aan $\Sigma_{i=0}^{h} 2^i = 2^{h+1} - 1$.

Stelling 9.1.7 *Voor een binaire boom T van hoogte h met n toppen is $h \geq \lceil log_2(\frac{n+1}{2}) \rceil$.*

Bewijs Wegens stelling 9.1.6 weten we dat $n \leq 2^{h+1} - 1$, of dus $2^h \geq \frac{n+1}{2}$. Dus $h \geq log_2(\frac{n+1}{2})$, en aangezien h een geheel getal is, $h \geq \lceil log_2(\frac{n+1}{2}) \rceil$.

Stelling 9.1.8 *Voor een binaire boom T van hoogte h met b bladeren is $h \geq log_2 b$.*

Bewijs We gebruiken inductie op h om de equivalente ongelijkheid $b \leq 2^h$ te bewijzen. Voor $h = 0$ bestaat de boom uit één top, die ook een blad is, m.a.w. $b = 1$ en dus $b \leq 2^h$. Veronderstel nu dat de ongelijkheid voldaan is voor elke binaire boom met hoogte kleiner dan h. We beschouwen nu een boom T van hoogte h met b bladeren.
We beschouwen eerst het geval dat de wortel van T slechts 1 kind heeft. Dit is een binaire boom van hoogte $h - 1$ die ook b bladeren heeft. Gebruik makend van de inductiehypothese bekomen we $b \leq 2^{h-1}$. Aangezien $2^{h-1} < 2^h$, is in dit geval dus ook $b \leq 2^h$.
Vervolgens beschouwen we het geval dat de wortel van T twee kinderen heeft. Zij h_l de hoogte van de linkerdeelboom en h_r de hoogte van de rechterdeelboom; er geldt dat $h_l \leq h - 1$ en $h_r \leq h - 1$. Zij b_l het aantal bladeren in de linkerdeelboom en b_r het aantal bladeren in de rechterdeelboom. Uit de inductiehypothese volgt dat $b_l \leq 2^{h_l}$ en $b_r \leq 2^{h_r}$. Hieruit bekomen we dat $b = b_l + b_r \leq 2^{h_l} + 2^{h_r} \leq 2^{h-1} + 2^{h-1} \leq 2^h$, hetgeen het gestelde bewijst.

Stelling 9.1.9 *Voor een ternaire boom T van hoogte h met b bladeren is $h \geq log_3 b$.*

9.2 Bomen en recursie

9.2.1 Recursieve verwerking van gewortelde bomen

Een (niet-ledige) **gewortelde boom** T is een (niet-ledige) eindige verzameling van toppen waarin één top als de **wortel** van de boom wordt aangeduid. De overige toppen zijn gepartitioneerd in $k \geq 0$ disjuncte verzamelingen $T_1, T_2, ..., T_k$ die alle bomen zijn en die **deelbomen** van de wortel worden genoemd.

9.2.2 Doorlopen in preorde, postorde en inorde

Doorlopen van een binaire boom Hieronder verstaat men: systematische manier om alle toppen van de boom te bezoeken.

Preorde Dit betekent dat men eerst de wortel verwerkt, dan zijn linkerdeelboom in preorde en dan zijn rechterdeelboom in preorde doorloopt.

Postorde Dit betekent eerst linkerdeelboom in postorde, dan zijn rechterdeelboom in postorde om ten slotte de wortel zelf te verwerken.

Inorde Dit betekent eerst de linkerdeelboom van wortel doorlopen in inorde, dan de wortel verwerken en dan de rechterdeelboom in inorde verwerken.

In elk van deze gevallen verloopt dit uiteraard recursief

Stelling 9.2.1 *Zij T een boom met n toppen. Dan gebeurt zowel doorlopen in preorde, als doorlopen in postorde en doorlopen in inorde in tijd $\Theta(n)$.*

Bewijs We geven het expliciete bewijs voor doorlopen in preorde; de andere bewijzen zijn volledig analoog. We bewijzen door inductie dat lijn 1 uit het algoritme in figuur (zie pg. 185)(m.a.w. het bezoeken van een top) in totaal n keer uitgevoerd wordt.

Voor $n = 1$ heeft de boom 1 top en wordt de lijn 1 slechts eenmaal uitgevoerd, aangezien er geen recursieve oproepen gebeuren. Veronderstel vervolgens dat $n > 1$ en dat voor elke $m < n$ een boom met m toppen deze lijn m keer uitgevoerd wordt. Zij k het aantal toppen in de linkerdeelboom van de wortel, dan is $n - k - 1$ het aantal toppen in de rechterdeelboom; in beide deelbomen is aantal toppen strikt kleiner dan n. Uit de inductiehypothese volgt dat het totaal aantal keer dat lijn 1 uitgevoerd wordt, gegeven wordt door $1 + k + (n - k - 1) = n$. Hieruit volgt onmiddellijk dat de uitvoeringstijd voor het doorlopen in preorde gegeven wordt door $\Theta(n)$.

9.3 Systematisch doorlopen van een graaf

9.3.1 Breedte-eerst-doorlopen van een graaf

Breedte-eerst-doorlopen Dikwijls afgekort als BFS, bezoekt de toppen van een (evt.gerichte) graaf G op een systematische manier, startend vanuit een bepaalde top r van G, ook de **wortel** genoemd.

Werkwijze De wortel is de eerste actieve top. Tijdens elk stadium in het doorlopen worden alle toppen adjacent vanuit de huidige top onderzocht op toppen die nog niet eerder bezocht zijn. De nieuwe actieve top wordt de minst recent bezochte top die nog niet de rol van huidige top gespeeld heeft. Het proces eindigt wanneer alle toppen als huidige top gediend hebben.
Aangezien de volgende actieve top de *minst recent* bezochte top is: werken met toppen in een **wachtlijn**.

Labelen Dit algoritme werkt m.b.v. het labelen van toppen. Dit kan op een manier naar keuze. *Bijvoorbeeld: alle labels 0, bij elk (nieuw!) bezoek van een top geven we het eerstvolgend beschikbare label. Indien niet alle toppen een positief label hebben, beginnen we het algoritme opnieuw vanaf die top met label 0.*

Breedte-eerst Na dit algoritme, bekomen we een opspannend woud F, we noemen dit woud een **breedte-eerst-woud**. Indien G samenhangend is, dan wordt een **breedte-eerst-boom** bekomen.

Complexiteit Elke boog wordt hoogstens tweemaal bekeken (eenmaal uit elk van zijn eindtoppen), een gesoleerde top wordt hoogstens eenmaal bezocht. De complexiteit is dus $\Theta(n+m)$.

9.3.2 Diepte-eerst-doorlopen van een graaf

Diepte-eerst-doorlopen Dit wordt ook wel DFS genoemd.

Werkwijze Zij $G = (V, E)$ een graaf met $V = \{v_1, v_2, ..., v_n\}$. De huidige top wordt de **actieve top** genoemd. We beginnen met het selecteren van een eerste te bezoeken top, nl. v_1, die het label 1 toegekend krijgt. Vervolgens selecteren we een topo adjacent met 1, deze top krijgt label 2 en deze top wordt de actieve top. De boog die de toppen met label 1 en 2 verbindt, wordt aan een verzameling B toegevoegd.

Algemeen, zij k het label van de huidige actieve top in de zoektocht, en veronderstel dat nite alle toppen in de component van G die k bevat, reeds bezocht zijn. We gaan als volgt verder. Als er onbezochte toppen adjacent met k zijn, selecteer een top adjacent met k die

nog niet bezocht is, en label hem met het volgende beschikbare label; deze top wordt de nieuwe actieve top en de boog tussen k en deze top wordt toegevoegd aan de verzameling B. Wanneer echter alle toppen adjacent met k reeds bezocht zijn, dan beschouwen we k als een **doodlopende top** en we keren terug (genaamd: **backtracking**) naar de top die de actieve to pwas vooraleer we k voor het eerst bezochten en deze top wordt de nieuwe huidige actieve top. Deze stop wordt herhaald totdat elke top in deze component van G bezocht is. Wanneer niet alle toppen van G bezocht werden, dan wordt een niet-bezochte top gekozen als de volgende actieve top, waarna het proces verder gaat.

Label Het label dat wordt toegekend wordt aan een top v wordt de **diepte-eerst-index** van v genoemd en genoteerd door dfi(v).

Diepte-eerst We noemen $< B >$ een opspannend woud, het **diepte-eerst-woud**. Als G samenhangend is dan is $< B >$ een **diepte-eerst-boom**.

Terugboog Elke boog van de graaf G die geen boog is van F, wordt een terugboog genoemd.

Stapel Bij dit algoritme kan een stapel worden gebruikt; een top die we voor de eerste keer tegenkomen wordt op de stapel gepusht - dan start het bezoek van de top. Wanneer een top doodlopend blijkt te zijn, wordt hij van de stapel gehaald - het bezoek is ten einde. Wanneer de stapel leeg is, dan is een volledige component van de graaf doorlopen op een diepte-eerst-manier.

Complexiteit Aangezien elke boog hoogstens tweemaal door het algoritme wordt beschouwd, is de complexiteit voor een graaf G met n toppen en m bogen gegeven door $\Theta(n+m)$, bij een adjacentielijstvoorstelling. Bij het gebruik van een adjacentiematrixvoorstelling, echter, zal de complexiteit $\Theta(n^2)$ zijn.

9.4 Beslissingsbomen

Beslissingsbomen Deze kunnen gebruikt worden voor het voorstellen van een algoritme dat bestaat uit een reeks vergelijkingen tussen elementen.

9.4.1 Ondergrens voor de complexiteit van zoeken

Algoritme gebaseerd op vergelijkingen De enige manier waarop het algoritme informatie verkrijgt over waar de in de rij gezochte sleutel kan optreden, bestaat uit het vergelijken van de sleutel met elementen uit de rij.

Stelling 9.4.1 *Zij C(n) het aantal vergelijkingen nodig voor het opzoeken van een sleutel in een rij van lengte n met een zoekalgoritme gebaseerd op vergelijkingen. Dan is* $C(n) = \Omega(logn)$.

Bewijs Een dergelijk zoekalgoritme in een rij kan ook worden beschreven als een reeks van **driewegsvergelijkingen**, waarvan de uitkomst een van de drie mogelijkheden 'gelijk', 'kleiner', 'groter' is. Om een dergelijk algoritme visueel voor te stellen kan een **driewegs-beslissingsboom** worden gebruikt. Dit is een ternaire boom waarbij elke interne top een driewegsvergelijking tussen de sleutel en een element in de rij voorstelt. Afhankelijk van de uitkomst van een bepaalde driewegsvergelijking, voert het algoritme een volgende vergelijking uit of neemt het een besluit; deze mogelijke volgende stappen vormen de kinderen in de boom van de corresponderende top. De bladeren in de boom geven aan welk element uit de rij gevonden werd, of bevatten informatie over de elementen waartussen de gezochte sleutel zich bevindt, in geval van een niet-geslaagde zoekbewerking.

In een voorstelling van het algoritme met een beslissingsboom komt dit slechtste geval overeen met de lengte van een langst mogelijke pad van de wortel naar een blad, m.a.w. met de hoogte van de boom. Beschouwen we nu de situatie waarin een rij a van n verschillende elementen en een sleutel x als input aan het algoritme gegeven wordt. In de beslissingsboom corresponderend met het algoritme moeten er minstens n bladeren zijn, want elk element uit de rij zou kunnen gelijk zijn aan de gezochte sleutel. een ternaire boom met minstens n bladeren heeft echter minstens hoogte log_n (zie stelling 9.1.9). Dit betekent dat een zoekalgoritme dus minstens $log_3 n$ vergelijkingen doet.

Stelling 9.4.2 *De slechtst mogelijke uitvoeringstijd van een zoekalgoritme gebaseerd op vergelijkingen is* $\Omega(logn)$.

Bewijs De slechtst mogelijke uitvoeringstijd voor een zoekalgoritme is minstens zo groot als het aantal vergelijkingen dat in het slechtste geval uitgevoerd wordt, want het algoritme doet naast de vergelijkingen ook nog ander werk (zoals variabelen aanpassen). Wegens voorgaande stelling doet een zoekalgoritme dat gebaseerd is op vergelijkingen voor een rij van n elementen, minstens $log_3 n$ vergelijkingen. De slechtst mogelijke uitvoeringstijd is dus $\Omega(logn)$.

9.4.2 Ondergrens voor de complexiteit van sorteren

Stelling 9.4.3 *Zij C(n) het aantal vergelijkingen nodig om n elementen te sorteren met een sorteeralgoritme gebaseerd op vergelijkingen. Dan is* $C(n) = \Omega(nlogn)$.

Bewijs Een binaire beslissingsboom wordt als volgt gebruikt om een sorteeralgoritme voor te stellen. Elke interne top van de boom wordt als volgt gebruikt om een sorteeralgoritme voor te stellen. Elke interne top van de boom stelt een vergelijking voor van de vorm $a_i \leq a_j$

tussen twee elementne uit de rij. Wanneer de vergelijking waar is, volgt het algoritme de linkertak, anders volgt het de rechtertak. Op de boog wordt de toestand van de te sorteren rij na de vergelijking (en eventuele verwisseling) getoond. Dit proces wordt herhaald totdat de ganse rij gesorteerd is, m.a.w. wanneer een interne top bereikt wordt, wordt een andere vergelijking uitgevoerd en vervolgens volgt het algoritme diens linker of rechtertak. Wanneer een blad in de boom bereikt wordt, is de rij gesorteerd.

In een voorstelling van het algoritme voor n elementen met een beslissingsboom T komt het slechtste geval overeen met de lengte van een zo lang mogelijk pad van de wortel naar een blad. M.a.w. $C(n) = h$, met h de hoogte van T.

Aangezien n verschillende elementen op n! manieren kunnen worden gerangschikt, moet T minstens n! bladeren hebben. Wegens Stelling 9.1.8 heeft een binaire boom met m bladeren minstens hoogte $log_2 m$, dus

$$h \geq log_2(n!).$$

Wegens Stelling 2.2.8 is

$$log_2(n!) = \Omega(nlogn),$$

en dus krijgen we dat

$$C(n) = h \geq log_2(n!) = \Omega(nlogn).$$

Hieruit volgt dat

$$C(n) = \Omega(nlogn),$$

Hetgeen het gestelde bewijst.

Stelling 9.4.4 *De slechtst mogelijke uitvoeringstijd van een sorteeralgoritme gebaseerd op vergelijkingen is* $\Omega(nlogn)$