Samenvatting Algoritmen en Datastructuren

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

April 7, 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

Gebruik van stapels en (prioriteitswachtlijnen)

3.1 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.

3.2 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 4

Recursie

4.1 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.

4.2 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.

4.2.1 Iteratiemethode

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

4.2.2 Substitutiemethode

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

4.2.3 Recursiebomen

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

4.2.4 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 $T(n) = \Theta(f(n))$

4.3 Wanneer recursie af te raden is

4.3.1 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 5

Verdeel-en-heers-algoritmen

5.1 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.

5.2 Zoeken in rijen

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

5.2.1 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 $\Theta(n)$

5.2.2 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.

5.3 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.

5.4 Het probleem van het dichtste puntenpaar

Chapter 6

Sorteeralgoritmen

6.1 Kwadratische sorteeralgoritmen

6.1.1 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 $T(n) = \Theta(n^2)$

6.1.2 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 $T(n) = \Theta(n^2)$

6.1.3 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.

6.2 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.

6.3 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.

6.3.1 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.

6.3.2 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 $\Theta(nlogn)$.

6.3.3 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.

6.3.4 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.

6.4 Lineaire sorteeralgoritmen

6.4.1 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.

6.4.2 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 $T(n) = \Theta(n)$.