Samenvatting Gegevensstructuren en Algoritmen

Bert De Saffel

2017-2018

Inhoudsopgave

Deel I

Theorie

Hoofdstuk 1

Inleiding

Een algoritme is een verzameling van opeenvolgende instructies die uitgevoerd worden. Bij een het oplossen van een probleem zijn er twee zaken belangrijk:

- De juiste aanpak gebruiken (Het algoritme).
- Een goede efficiëntie (De gegevensstructuren).

De focus van deze cursus ligt op de discrete wiskunde. Voor de continue wiskunde zijn er numerieke algoritmen nodig die niet behandeld worden in deze cursus.

Hoofdstuk 2

Efficëntie van algoritmen

2.1 Analyse van algoritmen

Probleem: Er bestaat een vector v met n getallen. Er moet gezocht worden of er dubbele waarden in deze vector bestaan. Een eerste oplossing zou zijn:

```
1 bool doubles = false;
2  for(int i = 0; i < n; i++){
3   for(int j = 0; j < n; j++){
4     if(i != j && v[i] = v[j]{
5       doubles = true;
6     }
7   }
8 }
```

Deze oplossing heeft volgende nadelen:

- Indien er een dubbele waarde is gevonden, wordt er nog altijd verder gezocht.
- Als gevolg heeft dit dat de vector tweemaal wordt doorlopen.
- Het if statement op lijn 4 wordt uitgevoerd als bijvoorbeeld $i = 5$ en $j = 27$, maar ook als $i = 27$ en $j = 5$. Er is dus sprake van redundantie.

Een betere oplossing zou kunnen zijn:

```
1 int i = 0;
2 int j = 1;
```

```
3 while(i < n && v[i] != v[j]){
4   j++;
5   if(j == n){
6     i++;
7     j = i + 1;
8   }
9 }
```

De eerste keer wordt de while n keer uitgevoerd, de tweede keer $n-1$ keer tot uiteindelijk de while nog maar 1 keer uitgevoerd wordt.
Het beste algoritme blijkt het volgende:

```
1 vector<bool> zitErIn(n, false);
2 int i = 0;
3 while(i < n && !zitErIn[v[i]]){
4   zitErIn[v[i]] = true;
5   i++;
6 }
```

Het voordeel van bovenstaand algoritme is dat de while lus hoogstens n keer uitgevoerd wordt.

2.1.1 Tijdscomplexiteit

Het is nuttig om te bekijken welke operaties een impact hebben op een algoritme. Beschouw volgende implementatie van het *insertion sort* algoritme:

```
1  void insertion_sort(vector<T> & v){
2    for(int i = 0; i < v.size() ; i ++){
3      T el = move(v[i]);
4      int j = i - 1;
5      while(j ≥ 0 && el < v[i]){
6        v[j + 1] = move(v[j]);
7        j--;
8      }
9      v[j + 1] = move(el);
10   }
11 }
```

Elke relevante operatie werd omkaderd en heeft een bepaalde uitvoeringstijd t. Voor elke operatie kan de uitvoeringstijd apart beschouwd worden in zijn slechtste en beste geval:

Operatie	Tijd	Aantal(best)	Aantal(slechtst)
h	t_1	1	1
i < v.size()	t_2	n	n
i++	t_3	n - 1	n - 1
T el = move(v[i])	t_4	n - 1	n - 1
int j = i - 1	t_5	n - 1	n - 1
j ≥ 0	t_6	n - 1	(n + 2)(n - 1)/2
h ¡ v[j]	t_7	n - 1	n(n - 1)/2
v[j + 1] = v[j]	t_8	0	n(n - 1)/2
j–	t_9	0	n(n - 1)/2
v[j + 1]	t_10	n - 1	n - 1)

- Beste geval:

$$(t_1 + t_8 + t_9) * 1 + (t_2 + t_3 + t_4 + t_5 + t_6 + t_7 + t_{10}) * n$$

  Er kan ook gezegd worden dat de tijdscomplexiteit in het beste geval gelijk is aan $O(n)$

- Slechtste geval:

$$(t_1) * 1 + (t_2 + t_3 + t_4 + t_5 + t_{10}) * n + (t_6 + t_7 + t_8 + t_9) * n^2$$

  Er kan ook gezegd worden dat de tijdscomplexiteit in het beste geval gelijk is aan $O(n^2)$

Het bewijs dat enkel de hoogste term de complexiteit bepaalt staat op pagina 10 van de cursus.

2.2 Asymptotische benadering

Een asymptotische benadering wil zeggen dat de uitvoeringstijd van een algoritme, vanaf voldoende grote waarden voor n, benadert kunnen worden met functies (Een begrenzende functie). Om deze begrenzende functie voor te stellen bestaan er drie notaties, namelijk: O, Θ en Ω. Onthou dat in de volgende voorbeelden de letter n het aantal elementen voorstelt.

2.2.1 O-notatie

De O-notatie stelt een afschatting naar boven voor. Dit wil zeggen dat een functie (het algoritme) niet sneller groeit dan een andere bepaalde functie, wat dus de bovengrens vormt. Zo wil de uitdrukking $O(n^2)$ zeggen dat het algoritme zeker niet sneller (maar wel gelijk) kan groeien dan de functie n^2.

2.2.2 Omega-notatie

De Ω-notatie stelt een afschatting naar onder voor. Dit wil zeggen dat een functie (ook weer het algoritme) niet trager groeit dan een andere bepaalde functie, wat de ondergrens vormt. Zo wil de uitdrukking $\Omega(n^2)$ zeggen dat het algoritme zeker niet traag (maar wel gelijk) kan groeien dan de functie n^2

2.2.3 Theta-notatie

Indien de ondergrens gelijk is aan de bovengrens wordt de Θ-notatie gebruikt.

2.2.4 Voorbeeld: sorteren

Indien een tabel met n *verschillende* elementen gegeven is, bestaan er $n!$ verschillende permutaties van n elementen. Wat is het efficiëntste sorteeralgoritme? begin vanaf de ongelijkheid $n! < n^n$.

$$n! < n^n$$
$$\log(n!) = O(n \log n)$$
$$\frac{n}{2} * (\frac{n}{2} + 1)...n$$
$$n! > (\frac{n}{2})^{n/2}$$
$$\log n! > \frac{n}{2} \log \frac{n}{2}$$
$$\log n! = \Theta(n \log n)$$

2.3 Afschatten van recursiebetrekkingen

Soms is het mogelijk een functie f af te schatten door een recursieve betrekking.

2.3.1 Machten van n

$$\begin{aligned} f(n) &\leq Cn^\alpha + f(n-1) \\ &\leq Cn^\alpha + C(n-1)^\alpha + f(n-2) \\ &\leq C(n^\alpha + (n-1)^\alpha) + ...1^\alpha) + f(0) \end{aligned}$$

De term tussen haakjes kan geschreven worden als de integraal:

$$n^\alpha + (n-1)^\alpha) + ...1^\alpha = \int_0^n \lceil x \rceil^\alpha \, dx$$

2.3.2 Logaritmische afschattingen

$$\begin{aligned} f(n) &= f\left(\frac{n}{2}\right) + C \\ &= f\left(\frac{n}{4}\right) + C + C \\ &= f\left(\frac{n}{2^k}\right) + kC \end{aligned}$$

indien k groot genoeg wordt

$$f(1) + kC \text{ met } k = \lceil \log n \rceil$$

2.3.3 Afschatting met sommen

$$\begin{aligned} &(x-1)(1 + x + x^2 + ... + x^p) \\ &= x + x^2 + x^3 + ... + x^{p+1} - 1 - x - x^2 - ... - x^p \\ &= -1 + x^{p+1} \end{aligned}$$

Gevolg:

$$\begin{aligned} 1 + x + ... + x^p &= \frac{x^{p+1}}{x-1} = S(x,p) \\ x^p < S(x,p) &< x^{p+1} \text{ voor } x \geq 2 \\ x + 2x^2 + 3x^3 + ... + px^p &< px^{p+1} \end{aligned}$$

2.3.4 Bernouilliverdeling

Iets heeft een kans p om te mislukken.

$$p = \frac{1}{10e}$$

Een bepaald algoritme moet een slaagkans hebben van 80%. Zal dit lukken met de kans p? De kans dat er bij m pogingen het j keer misgaat is:

$$\sum_{k=0}^{k=j} C_m^k (1-p)^{m-k} p^k < \sum_{k=0}^{k=j} \left(\frac{emp}{k}\right)^k$$

2.4 Optimalisatie

Stel een algoritme a met 2 afschattingen.

$$\begin{cases} f_1(n) = \Omega(g_1(n)) \rightarrow \text{Actie } f_1(n) \text{ keer} \\ f_2(n) = \Omega(g_2(n)) \rightarrow \text{efficiëntie van 1 actie} \end{cases}$$

De afschatting wordt $\Omega(g_1(n)g_2(n))$ Een voorbeeld is een quizwedstrijd. Bij een quiz mag elke ploeg een prijs kiezen, en dit in volgorde van de uitslag. Stel dat deze uitslag gegeven is in een tabel `uitslag` zodang dat `uitslag[i]` aangeeft wat de rangschikking is van ploeg i. Er bestaan twee manieren om ploegen in volgorde aan te spreken.

```
1  for(int i = 0; i < n ; i++){
2   int j = 0;
3   while ( uitslag[j] != i){
4     j++;
5   }
6   kies(j);
7  }
```

De forlus ($g_1(n)$) wordt n keer uitgevoerd. De whilelus ($g_2(n)$) wordt ook n keer uitgevoerd. De afschatting wordt dus

$$\Omega(g_1(n)g_2(n)) = \Omega(n^2)$$

Een betere methode:

```
1 vector<int> invers(n);
2 for(int i = 0; i < n; i++){
3  invers[uitslag[i]] = i;
4 }
5 for(int j = 0; j < n; j++){
6  kies[invers[j]);
7 }
```

Dit voorbeeld heeft geen geneste lus, wat ervoor zorgt dat de afschatting $\Omega(n)$ wordt.

Hoofdstuk 3

Rangschikking

Belangrijk bij sorteeralgoritmes:

- Het aantal gegevens
- De aard van die gegevens
- De mate waarin de gegevens reeds geordend zijn
- De plaats waar de gegevens tijdens het rangschikken opgeslagen zijn (inwendig of uitwendig geheugen)

Er wordt aandacht besteedt aan drie zaken:

- Tijdsefficiëntie
- Geheugengebruik
- Stabiliteit

Veronderstellingen:

- Alle gegevens bevinden zich in het inwendig geheugen.
- De gegevens worden ondergebracht in een één dimensionale tabel.
- De gegevens zijn klein en bevatten geen bijhorende informatie.
- Er wordt gerangschikt in stijgende volgorde.

3.1 Insertion Sort

```
1  void insertion_sort(vector<T> & v){
2    for(int i = 0; i < v.size(); i++){
3      T el = move(v[i]);
4      int j = i - 1;
5      while(j <= 0 && el < v[i]){
6        v[j + 1] = move(v[j]);
7        j--;
8      }
9      v[j + 1] = move(el);
10   }
11 }
```

3.1.1 Werking

Insertion sort is een eenvoudig algoritme dat element per element werkt. Vergelijk een element met de opeenvolgende elementen in de tabel. Schuif de grotere elementen op en zet het element op zijn plaats. De gerangschikte en nog te rangschikken zones hebben geen overlap.

Slechtste geval

Wanneer de tabel in omgekeerde volgorde gesorteerd staat.

Gemiddelde geval

Beste geval

3.1.2 Voordelen

3.1.3 Nadelen

straight from notities
O(n + aantal inversies)
tabel v
vomgekeerd[i] = v[n - 1 - i];
vomgekeerd[n - 1 - i] ¿ omgekeerd[n - 1 - j];

inversies in v + #inversies in vomgekeerd = $\frac{n(n-1)}{2}$
random tabel

3.2 Shell Sort

.....

3.3 Selection Sort

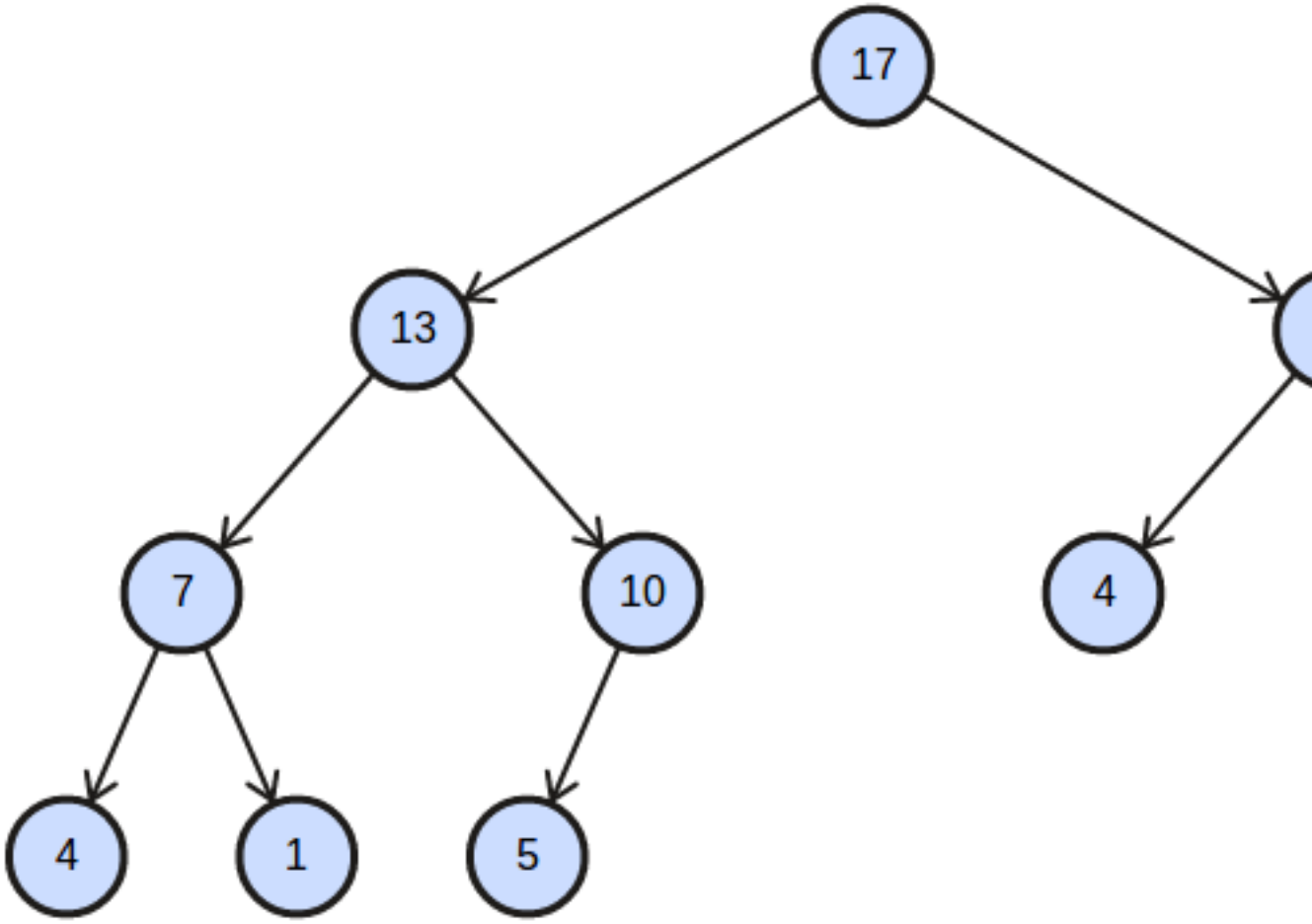
.....

Hoofdstuk 4

Efficiënte Methodes

4.1 Heaps (= Binaire boom)

Een heap lijkt sterk op een gelinkte lijst, maar elke knoop heeft slechts 1 ouder en potentieël 2 kinderen. Een heap loopt van boven naar beneden. Het topelement wordt de *wortel* genoemd. Een knoop zonder kinderen wordt een *blad* genoemd. Een heap moet voldoen aan de *heapvoorwaarde*: Elke sleutel van een knoop heeft een hogere prioriteit dan de sleutels van zijn kinderen.



Toevoegen van een knoop (14)

4.2 Quicksort