Algoritmes en complexiteit

Definities en grote-O-notatie

beOI Training

OLYMPIADE BELGE D'INFORMATIQUE
BELGISCHE INFORMATICA-OLYMPIADE

8 octobre 2016

Inhoudstafel

Wat is een algoritme ?

- Een manier om een resultaat te berekenen
- Een idee om een probleem op te lossen
- Een opeenvolging van instructies
- De beschrijving van een programma

Wat is een *goed* algoritme?

Aandachtspunten voor de programmeur

- Crasht niet
- Eindigt
- Geeft het goede antwoord

Aandachtspunten voor de competitieve programmeur

- Snelheid
- Weinig geheugengebruik
- **Wordt geaccepteerd in een wedstrijd**

Inhoudstafel

De efficiëntie meten

Ideëen

- De tijd opmeten
- Het RAM geheugen dat gebruikt wordt opmeten

Maar varieert volgens

- Taal
- Implementatie
- Machine
- Uur van de dag

grote-O-notatie

Geeft een *intrinsieke* notie van efficiëntie weer :

- Door de input te vergroten
- Door te kijken hoe de snelheid evolueert

Voorbeeld : bereken $1 + \cdots + n$.Als n vermenigvuldigd wordt met 2, dan zou het volgende kunnen gebeuren met de tijd :

- Blijft constant : $O(1)$
- Wordt vermenigvuldigd met 2 : $O(n)$
- Wordt vermenigvuldigd met 4 : $O(n^2)$

Dit is onafhankelijk van constante factoren !

Constante tijd

Probleem : Bereken de som : $1 + 2 + \cdots + n$.

Oplossing 1 : Een simpele berekening

```
int sum = n * (n+1) / 2;
```

- Tijd verandert niet als n verdubbelt
- "constante" tijd
- $O(1)$ complexiteit
- Een tijd "proportioneel met 1"

Lineaire tijd

Oplossing 2 : Een lus

```
int sum = 0;
for (int i = 1; i <= n; i++)
    sum += i;
```

- Tijd verdubbelt als n verdubbelt
- "lineaire" tijd
- $O(n)$ complexiteit
- Een tijd "proportioneel met n"

Kwadratische tijd

Oplossing 3 : Twee lussen (dom !)

```
int sum = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= i; j++)
        sum++;
```

- Tijd wordt vier keer groter als n verdubbelt
- "Kwadratische" tijd
- $O(n^2)$ complexiteit
- Een tijd "proportioneel met n^2"

Machten

Definitie :

- Herhaald vermenigvuldigen
- “3 tot de macht *n*”
- $3^n = \underbrace{3 \times \cdots \times 3}_{n \text{ keer}}$

Voorbeelden :

- $3^0 = 1$ (dit is zo gedefinieerd)
- $3^1 = 3$
- $3^2 = 3 \times 3 = 9$ (kwadraat)
- $3^3 = 3 \times 3 \times 3 = 27$ (derde macht)

Logaritmes : intuïtie (1)

Spel met twee spelers :

- Alice kiest een getal tussen 1 en 16
- Om de beurt gebeurt het volgende :
  - Bob geeft Alice één of meerdere getallen
  - Alice zegt of haar getal tussen die getallen zit
- Wanneer Bob het getal gevonden heeft, wint hij
- Hoe kan hij winnen in zo weinig mogelijk zetten ?

Logaritmes : intuïtie (2)

Strategie : Geef de helft van de mogelijke getallen

- Eerst 8 getallen uit 16
- Vervolgens 4 getallen uit de overgebleven 8
- Vervolgens 2 getallen uit de overgebleven 4
- Vervolgens 1 getal uit de overgebleven 2
- Gevonden !

Dus 4 vragen zijn voldoende.

Logaritmes : intuïtie (3)

In het algemeen, als we starten met n getallen, hoeveel vragen zijn er dan nodig ?

Hoe vaak kan je in 2 helften splitsen ?

- Als $n = 2$, één keer
- Als $n = 4$, twee keer
- Als $n = 8$, drie keer
- Als $n = 16$, vier keer

Basis 2 logaritme

De functie die het antwoord biedt op deze vraag is $\log_2$: De basis 2 logaritme. Bijvoorbeeld

- $\log_2(2) = 1$
- $\log_2(4) = 2$
- $\log_2(8) = 3$
- $\log_2(16) = 4$

De strategie van Bob is $O(\log_2(n)) = O(\log n)$.

De basis 2 logaritme is dus de macht waartoe je 2 moet verheffen om n te krijgen :

$$x = \log_2(n) \;\Leftrightarrow\; 2^x = n$$

Logaritmes in het algemeen (extra)

Dit geldt niet enkel voor 2 ! De logaritme in basis a, is het aantal keer dat je kan delen door a. Bijvoorbeeld :

- $\log_3(27) = 3$
- $\log_4(16) = 2$
- $\log_5(5) = 1$

De basis a logaritme is dus de macht waartoe je a moet verheffen om n te krijgen :

$$x = \log_a(n) \Leftrightarrow a^x = n$$

Je kan het een beetje zien als "de inverse" van machten.

Zoeken in een gesorteerde tabel (1)

We krijgen een tabel die gesorteerd is volgens stijgende volgorde :

1	4	6	9	15	23	24

Ga na of een getal x zich erin bevindt.

Oplossing 1 : Alles overlopen, lineaire tijd, $O(n)$

```
bool isIn(int tab[], int n, int x)
{
    for (int i = 0; i < n; i++)
        if (tab[i] == x)
            return true;
    return false;
}
```

Zoeken in een gesorteerde tabel (2)

We zoeken 7.

Idee : kijk in het midden en vergelijk :

1	4	6	**9**	15	23	24

Te groot ($9 > 7$), dus we gaan naar links :

1	**4**	6	9	15	23	24

Te klein ($4 < 7$), dus we gaan naar rechts :

1	4	**6**	9	15	23	24

Maar $6 \neq 7$ dus 7 zit niet in de tabel.

Zoeken in een gesorteerde tabel (3)

We splitsen telkens in 2, dus $\Rightarrow \log_2(n)$ pogingen nodig.

Solution 2 : binair zoeken, logaritmische tijd, $O(\log n)$

```
bool isIn(int tab[], int n, int x)
{
    int left = 0, right = n-1;
    while (left <= right)
    {
        int mid = (left+right) / 2;
        if (x < tab[mid]) right = mid - 1;
        else if (x > tab[mid]) left = mid + 1;
        else return true;
    }
    return false;
}
```

Veel sneller !

Praktische limieten

Limieten voor n om uit te voeren in enkele seconden :

Complexiteit	Limiet voor n	Voorbeeld
$O(1)$, $O(\log n)$	$\leq 10^{18}$	(Ongeveer de limiet voor een long
$O(n)$	$\leq 100\,\mathrm{M}$	Een rijl overlopen
$O(n \log n)$	$\leq 1\,\mathrm{M}$	Een rij sorteren
$O(n^2)$	$\leq 10\,\mathrm{k}$	Een geneste lus (een lus binnen ee

Samengevat : kijk naar de tweede kolom en deze geeft ongeveer de hoogst mogelijke complexiteit weer die je programma mag hebben.