Algoritmer, datastrukturer och komplexitet, hösten 2012

Uppgifter till övning 1

Algoritmanalys

Ordo Jämför följande par av funktioner med avseende på hur dom växer då n växer. Tala i varje fall om ifall $f(n) \in \Theta(g(n))$, $f(n) \in O(g(n))$ eller $f(n) \in \Omega(g(n))$.

	$f(n)$	$g(n)$
a)	$100n + \log n$	$n + (\log n)^2$
b)	$\log n$	$\log n^2$
c)	$\frac{n^2}{\log n}$	$n(\log n)^2$
d)	$(\log n)^{\log n}$	$\frac{n}{\log n}$
e)	$\sqrt{n}$	$(\log n)^5$
f)	$n2^n$	3^n
g)	$2^{\sqrt{\log n}}$	$\sqrt{n}$

Division Analysera skolboksalgoritmen för division (liggande stolen, tidigare trappdivision). Använd bitkostnad.

Euklides algoritm Analysera Euklides algoritm som hittar största gemensamma delaren mellan två heltal. Analysera både med avseende på enhetskostnad och bitkostnad. Euklides algoritm lyder på följande sätt, där vi förutsätter att $a \geq b$.

```
gcd(a, b)=
  if b|a then
    gcd ← b
  else
    gcd ← gcd(b, a mod b)
```

Potenser med upprepad kvadrering Följande algoritm beräknar tvåpotenser när exponenten själv är en tvåpotens.

Indata: $m = 2^n$
Utdata: 2^m

```
power(m) =
    pow ← 2
    for i ← 1 to log m do
        pow ← pow · pow
    return pow
```

Analysera denna algoritm både med avseende på enhetskostnad och bitkostnad.

Lösningar

Lösning till Ordo

a) Vi beräknar gränsvärdet för kvoten mellan funktionerna.

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{100n+\log n}{n+(\log n)^2}=\lim_{n\to\infty}\frac{100+(\log n)/n}{1+(\log n)^2/n}=100.$$

Eftersom gränsvärdet är en konstant drar vi slutsatsen att $f(n)\in\Theta(g(n))$.

b) Vi noterar att $\log n^2=2\log n$, så $\log n\in\Theta(\log n^2)$.

c)

$$\frac{g(n)}{f(n)}=\frac{n(\log n)^2}{n^2/\log n}=\frac{(\log n)^3}{n}\to 0 \text{ då } n\to\infty.$$

Därför är $g(n)\in O(f(n))$ och $f(n)\in\Omega(g(n))$.

d) Substituera $m=\log n$ och jämför sedan $f_2(m)=m^m$ och $g_2(m)=2^m/m$:

$$\frac{g_2(m)}{f_2(m)}=\frac{2^m}{m\cdot m^m}=\frac{1}{m}\left(\frac{2}{m}\right)^m\to 0$$

då $n\to\infty$. Därför är $f(n)\in\Omega(g(n))$ och $g(n)\in O(f(n))$.

e) Polynomiska funktioner vinner alltid över polylogaritmiska. Därför är $f(n)\in\Omega(g(n))$ och $g(n)\in O(f(n))$.

Om man vill ha ett mer formellt bevis kan man använda lemmat som säger att för alla konstanter $c>0$, $a>1$ och alla monotont växande funktioner $h(n)$ är $(h(n))^c\in O(a^{h(n)})$. Om vi väljer $h(n)=\log n$, $c=5$ och $a=\sqrt{2}$ så får vi $(\log n)^5\in O(\sqrt{2}^{\log n})=O(\sqrt{n})$ eftersom $\left(2^{1/2}\right)^{\log n}=\left(2^{\log n}\right)^{1/2}=n^{1/2}$.

f) $\frac{f(n)}{g(n)}=\frac{n2^n}{3^n}=n\left(\frac{2}{3}\right)^n$. En exponentialfunktion som $(2/3)^n$ vinner alltid över en polynomisk funktion (som n), men låt oss bevisa det med l'Hôpitals regel. Formulera först om uttrycket: $n\left(\frac{2}{3}\right)^n=n/\left(\frac{2}{3}\right)^{-n}$. Inför beteckningar för nämnare och täljare, $r(n)=n$ och $s(n)=\left(\frac{2}{3}\right)^{-n}$, och derivera.

$$r'(n)=1,\qquad s'(n)=\left(\frac{2}{3}\right)^{-2n}\left(\frac{2}{3}\right)^n\ln\left(\frac{3}{2}\right)=\frac{\ln(3/2)}{\left(\frac{2}{3}\right)^n}$$

Vi kan använda l'Hôpital eftersom $r(n)\to\infty$, $s(n)\to\infty$ och $s'(n)\neq 0$.

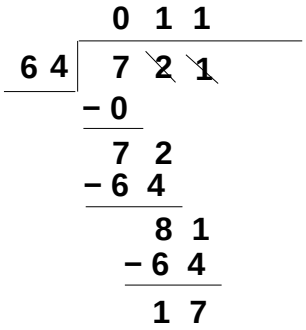
$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{r(n)}{s(n)}=\lim_{n\to\infty}\frac{r'(n)}{s'(n)}=\frac{\left(\frac{2}{3}\right)^n}{\ln\left(\frac{3}{2}\right)}\to 0$$

och vi har visat att $f(n)\in O(g(n))$.

g) $\sqrt{n}\in\Omega(2^{\sqrt{\log n}})$. Notera bara att $\sqrt{n}=2^{\frac{1}{2}\log n}$ och att $\lim_{n\to\infty}2^{\sqrt{\log n}-\frac{1}{2}\log n}=0$.

□

Lösning till Division
Så här ser skolboksmetoden ut när 721 delas med 64. Uppställningen kallas trappan. Den som är van vid liggande stolen skriver istället nämnaren på högra sidan av täljaren.



Algoritmen börjar med att kolla hur många gånger 64 går i 7, den mest signifikanta siffran i 721. Eftersom $64 > 7$ är det 0 gånger. Vi har $0 \cdot 64 = 0$ så vi subtraherar 0 från 7 och får 7. Vi multiplicerar 7 med 10 (basen), flyttar ner nästa siffra i 721, nämligen 2, och fortsätter att dividera 72 med 64 för att få nästa siffra i kvoten. Vi fortsätter på detta sätt, subtraherar antalet gånger 64 går i varje tvåsiffrigt tal, flyttar ner nästa siffra och slutar när alla heltalssiffror i kvoten har beräknats. Talet 17 längst ner är divisionens rest.

Vi kan formulera detta i följande algoritm där vi beräknar $q = b/a$ där a, b är n-bitstal ($a = a_{n-1}a_{n-2}\cdots a_0$, $b = b_{n-1}b_{n-2}\cdots b_0$) i basen B. Låt $x \ll y$ vara x vänsterskiftat y steg. För att det ska vara lättare att bevisa att algoritmen är korrekt har ingångsvillkor, utgångsvillkor och invarianter satts ut.

$Div(a, b, n) =$
PRE: $a > 0, b \geq 0$, a och b lagras med n bitar.
POST: $qa + r = b, 0 \leq r < a$
$r \leftarrow 0$
for $i \leftarrow n - 1$ **to** 0 **do**
INV: $(q_{n-1}...q_{i+1}) \cdot a + r = (b_{n-1}...b_{i+1}), 0 \leq r < a$
$r \leftarrow (r \ll 1) + b_i$ /* Byt till nästa siffra */
$q' \leftarrow 0$
$a' \leftarrow 0$
while $a' + a \leq r$ **do** /* Hitta max q' så att $q'a \leq r$ */
INV: $a' = q'a \leq r$
$a' \leftarrow a' + a$
$q' \leftarrow q' + 1$
$q_i \leftarrow q'$
$r \leftarrow r - a'$
return $\langle q, r \rangle$ /* kvot och rest */

Vad är tidskomplexiteten? For-slingan går n varv. I varje varv har vi ett konstant antal tilldelningar, jämförelser, additioner och subtraktioner samt en while-slinga som går högst B varv (eftersom B är basen och talet $B \cdot a \geq r$). Eftersom basen kan anses vara en konstant så innehåller varje varv i for-slingan ett konstant antal operationer. Och eftersom all aritmetik görs med n-bitstal så tar varje jämförelse, addition och subtraktion tiden $O(n)$. Totalt får vi $n \cdot c \cdot O(n) = O(n^2)$. □

Lösning till Euklides algoritm
Eftersom algoritmen är rekursiv är det inte uppenbart att den alltid tar slut (terminerar), så vi börjar med att bevisa det. I termineringsbevis använder man oftast en potentialvariabel som har en undre gräns (till exempel noll) och vid varje anrop minskar med ett heltalssteg. I denna algoritm kan vi använda a som potentialvariabel. Eftersom a alltid är minst lika stor som b och algoritmen terminerar så fort $a = b$ eller $b = 1$ så har vi en undre gräns för a och vi kan se att a minskar i varje anrop. Antalet anrop beror tydligen på parametrarnas storlek, så låt oss beräkna hur den största parametern, a, minskar. Låt a_i vara as värde i anrop nummer i i algoritmen.

Lemma 1 $a_{i+2} \leq a_i/2$.

Bevis. Vi vet att $a_{i+2} = b_{i+1}$ och $b_{i+1} = a_i \bmod b_i$, varför $a_{i+2} = a_i \bmod b_i$. Anta nu att $a_{i+2} > a_i/2$. Detta innebär att $b_i \geq a_i/2$, vilket ger en motsägelse, eftersom $a_i = a_{i+2} + cb_i > a_i/2 + a_i/2 = a_i$. □

Med hjälp av lemmat kan vi visa att

$$\lceil \log a_{i+2} \rceil \leq \left\lceil \log \frac{a_i}{2} \right\rceil = \lceil \log a_i - \log 2 \rceil = \lceil \log a_i \rceil - 1.$$

Det betyder att i vartannat anrop av funktionen minskas parametrarnas storlek med (minst) en bit. Därför kan antalet anrop vara högst $2\lceil \log a \rceil$.

I varje rekursivt anrop görs bara en modulooperation och med enhetskostnad tar det konstant tid. Därmed har vi kommit fram till att tidskomplexiteten är $2\lceil \log a \rceil = 2n \in O(n)$.

När vi analyserar algoritmen med avseende på bitkostnad måste vi vara noggrannare. En division mellan två n-bits heltal tar (som vi har sett) $O(n^2)$, och modulooperationen tar därför $O(n^2)$. Så om vi gör $O(n)$ anrop och varje anrop tar $O(n^2)$ så blir den totala bitkomplexiteten $O(n^3)$. □

Lösning till Potenser med upprepad kvadrering
Slingan går $\log m = n$ varv. Varje varv tar konstant tid med enhetskostnad. Komplexiteten blir därför $O(n)$.

Om vi använder bitkomplexitet måste vi undersöka vad varje multiplikation i slingan kostar. Vi antar att kostnaden för att multiplicera ett l-bitstal med sig själv är $O(l^2)$ (vilket är en överdrift – det finns snabbare sätt). I varv i var variabeln $pow = pow_i$ värdet 2^{2^i} så varje multiplikation kostar $O((\log pow_i)^2) = O((\log 2^{2^i})^2) = O(2^{2i})$. Om vi summerar över alla varv får vi

$$\sum_{i=1}^{\log m} c2^{2i} = c \sum_{i=1}^{\log m} 4^i = 4c \sum_{i=0}^{\log m - 1} 4^i = 4c\frac{4^{\log m} - 1}{4 - 1} \in O(4^{\log m}) = O(4^n).$$

Algoritmen har alltså linjär komplexitet med avseende på enhetskostnad men exponentiell komplexitet med avseende på bitkostnad! □
