

Algoritmer och Komplexitet ht 09. 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 (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 algoritmer 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.

Kantmatrisprodukt På föreläsningen beskrevs hur en riktad graf $G = \langle V, E \rangle$ representeras som en kantmatris B (eng. incidence matrix) av storlek $|V| \times |E|$. Beskriv vad elementen i matrisen BB^T representerar (där B^T är B transponerad).

Bipartithet Beskriv och analysera en algoritm som avgör om en graf är bipartit. Tidskomplexiteten för algoritmen ska vara linjär i antalet kanter och antalet hörn i grafen.

Eulercykel Givet en oriktad sammanhängande graf $G = \langle V, E \rangle$ där alla hörn har jämnt gradtal. Hitta en Eulercykel, dvs en sluten stig som passerar varje kant i E exakt en gång. Algoritmen ska gå i linjär tid.

Lösningar

Lösning till Ordo

- a) Vi beräknar gränsvärdet för kvoten mellan funktionerna.

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

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)

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

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 $g_2(m) = 2^m/m$:

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

då $n \rightarrow \infty$. 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 $(2^{1/2})^{\log n} = (2^{\log n})^{1/2} = n^{1/2}$.

- 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: $n \left(\frac{2}{3}\right)^n = n / \left(\frac{3}{2}\right)^n$. Inför beteckningar för nämnare och täljare, $r(n) = n$ och $s(n) = \left(\frac{3}{2}\right)^n$, och derivera.

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

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

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

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 $\lim_{n \rightarrow \infty} 2^{\sqrt{\log n} - \frac{1}{2} \log n} = 0$.

□

Så här ser skolboksmetoden ut när 721 delas med 64:

$$\begin{array}{r}
 011 \\
 \hline
 64 \overline{) 7\cancel{2}\cancel{1}} \\
 \underline{-0} \\
 72 \\
 \underline{-64} \\
 81 \\
 \underline{-64} \\
 17
 \end{array}$$

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.

```

return  $\langle q, r \rangle$       /* Quotient and remainder */

```

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 $n \cdot c \cdot O(n) = O(n^2)$. $\square$

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 a s 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 $a_i = a_{i+2} + cb_i > a_i/2 + a_i/2 = a_i$. $\square$

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 $O(n^3)$. $\square$

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

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

Algoritmen har alltså linjär komplexitet med avseende på enhetskostnad men exponentiell komplexitet med avseende på bitkostnad! $\square$

Lösning till Kantmatrisprodukt

Diagonalelementet (i, i) anger hur många kanter som har sin ändpunkt i hörn i . Ickediagonalelementet (i, j) är det negerade antalet kanter som går mellan hörnen i och j . $\square$

Lösning till Bipartithet

Använd djupetförstökning men färga hörnen alternerande med rött och grönt. Om en kant mellan två hörn av samma färg upptäcks är grafen inte bipartit. $\square$

Lösning till Eulercykel

Idén är att börja i ett hörn v och söka sig igenom grafen tills v nås igen. Sökstigen P kommer då

antingen att ha besökt alla kanter eller så återstår det, om vi tar bort P , en eller flera sammanhängande komponenter $G_1, G_2, \dots, G_n$. I det senare fallet kan vi göra om samma sorts sökning för varje komponent så vi får en Eulercykel för varje G_i , och sedan sätter vi in alla dessa Eulercykler i P så att vi får en total Eulercykel.

För att kunna genomföra detta i linjär tid så hittar vi på en algoritm som använder två spelare P_1 och P_2 som tillsammans går igenom grafen. Båda spelarna startar i v . Spelare P_1 skapar stigen P genom att gå längs kanter hur som helst. Han märker varje kant han passerar och stannar när han kommer tillbaka till v igen. Sedan följer P_2 efter längs P , men varje gång han kommer till ett nytt hörn u så kollar han om alla kanter från u är märkta. I så fall fortsätter han längs P . Om det finns omärkta kanter (det finns alltid ett jämnt antal) så skickar han ut P_1 för att hitta en ny stig som börjar och slutar i u . P_2 går sedan denna nya stig innan han fortsätter från u . Efter att ha upprepat detta kommer P_2 till slut att ha gått längs varje kant exakt en gång och därför skapat en Eulercykel. P_1 har inte heller gått längs samma kant mer än en gång så totala körtiden blir linjär.

I algoritmen nedan kallas P_1 för *PathFinder* och P_2 för *Straggler*.

```

EulerCycle(G) =
  cycle ← {1}
  choose edge (1, t)
  mark (1, t)
  path ← PathFinder(G, 1, t)
  Straggler(path)
  return cycle

PathFinder(G, start, cur) =
  append(cur, path)
  while cur ≠ start do
    choose unmarked edge (cur, v)
    mark (cur, v)
    append(v, path)
    cur ← v
  return path

Straggler(path) =
  while path ≠ ∅ do
    u ← next(path)
    append(u, cycle)
    for all edges (u, v) do
      if unmarked (u, v) then
        mark (u, v)
        p ← PathFinder(G, u, v)
        Straggler(p)

```

□