Föreläsning 24

Metoder för algoritmdesign

TDDD86: DALP

Utskriftsversion av föreläsning i *Datastrukturer, algoritmer och programmeringsparadigm*
7 december 2015

Tommy Färnqvist, IDA, Linköpings universitet

Innehåll

Innehåll

1 Dekomposition

Divide and conquer, söndra och härska

- Ett allmänt paradigm för algoritmdesign:
  - Dela upp indata S i två eller fler disjunkta delmängder $S_1, S_2, \ldots$
  - Lös delproblemen rekursivt
  - Kombinera lösningarna till delproblemen till en lösning till S
- Basfallen är delproblem av konstant storlek
- Analysen kan genomföras m.h.a. rekurrensrelationer

Exempel: Matrismultiplikation

$$C = AB \Leftrightarrow \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Antag att vi har $n \times n$-matriser och att $n = 2^k$ för något k

function MUL(A, B, n)
  if $n = 1$ **then return** $A \cdot B$
  $C_{11} =$ADD(MUL$(A_{11}, B_{11}, \frac{n}{2})$,MUL$(A_{12}, B_{21}, \frac{n}{2}), \frac{n}{2})$
  $C_{12} =$ADD(MUL$(A_{11}, B_{12}, \frac{n}{2})$,MUL$(A_{12}, B_{22}, \frac{n}{2}), \frac{n}{2})$
  $C_{21} =$ADD(MUL$(A_{21}, B_{11}, \frac{n}{2})$,MUL$(A_{22}, B_{21}, \frac{n}{2}), \frac{n}{2})$
  $C_{22} =$ADD(MUL$(A_{21}, B_{12}, \frac{n}{2})$,MUL$(A_{22}, B_{22}, \frac{n}{2}), \frac{n}{2})$
  return C

Analys

$$\left.\begin{array}{l} T(1) = 1 \\ T(n) = 8T\left(\frac{n}{2}\right) + 4 \cdot \left(\frac{n}{2}\right)^2 \end{array}\right\} \Rightarrow T(n) \in O(n^3)$$

Exempel: Matrismultiplikation (Strassen)

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Kan beräknas genom $m_1 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$ $m_2 = (a_{12} + a_{22}) \cdot (b_{11} + b_{22})$ $m_3 = (a_{11} - a_{21}) \cdot (b_{11} + b_{12})$ $m_4 = (a_{11} + a_{12}) \cdot b_{22}$ $m_5 = a_{11} \cdot (b_{12} - b_{22})$ $m_6 = a_{22} \cdot (b_{21} - b_{11})$ $m_7 = (a_{21} + a_{22}) \cdot b_{11}$
$c_{11} = m_1 + m_2 - m_4 + m_6$ $c_{12} = m_4 + m_5$ $c_{21} = m_6 + m_7$ $c_{22} = m_2 - m_3 + m_5 - m_7$

Analys
Sammanlagt 7 multiplikationer och 18 additioner/subtraktioner

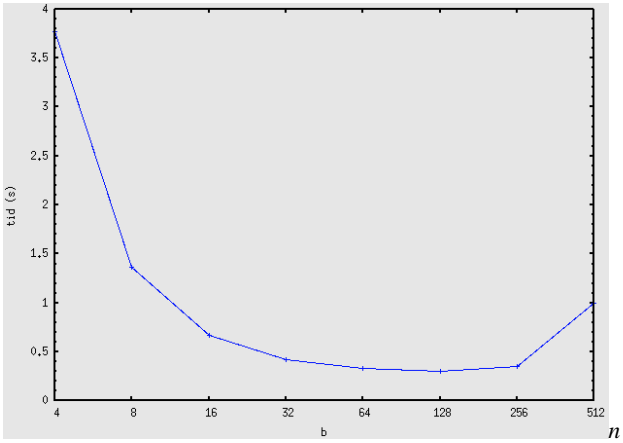
Multiplikation av två $n \times n$-matriser tar tid

$$\left.\begin{aligned} T(1) &= 1 \\ T(n) &= 7T\left(\tfrac{n}{2}\right) + 18 \cdot \left(\tfrac{n}{2}\right)^2 \end{aligned}\right\} \Rightarrow T(n) \sim n^{2.81}$$

24.5

Strassen i praktiken

- Använd Strassens matrismultiplikationsalgoritm för stora matriser ($n > b$).
- Använd vanlig matrismultiplikation för mindre matriser ($n \leq b$)



Hur ska brytpunkten väljas? $n = 512$

24.6

Exempel: Multiplikation av binära tal

$$x = \underbrace{x_{n-1}x_{n-2}\ldots x_{\frac{n}{2}}}_{a}\underbrace{x_{\frac{n}{2}-1}\ldots x_1x_0}_{b} = a \cdot 2^{\frac{n}{2}} + b$$

$$y = \underbrace{y_{n-1}y_{n-2}\ldots y_{\frac{n}{2}}}_{c}\underbrace{y_{\frac{n}{2}-1}\ldots y_1y_0}_{d} = c \cdot 2^{\frac{n}{2}} + d$$

$$x \cdot y = a \cdot c \cdot 2^n + (a \cdot d + b \cdot c)2^{\frac{n}{2}} + b \cdot d$$

Antag att $n = 2^k$

function MULT(x, y, k)
  if $k = 1$ **then return** $x * y$
  else
    $[a, b] \leftarrow x$
    $[c, d] \leftarrow y$
    $p_1 \leftarrow$MULT$(a, c, k-1)$
    $p_2 \leftarrow$MULT$(b, d, k-1)$
    $p_3 \leftarrow$MULT$(a, d, k-1)$
    $p_4 \leftarrow$MULT$(b, c, k-1)$
    return $p_1 \cdot 2^n + (p_3 + p_4) \cdot 2^{\frac{n}{2}} + p_2$

Analys

$$T(1) = \Theta(1) \text{ och}$$
$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n)$$
$$\Rightarrow T(n) \in O(n^{\log_2 4})$$

24.7

Exempel: Smartare multiplikation (Karatsuba)

$A \leftarrow a \cdot c$ $B \leftarrow b \cdot d$ $C \leftarrow (a+b) \cdot (c+d)$ $D \leftarrow A \cdot 2^n + (C-A-B) \cdot 2^{\frac{n}{2}} + B$ $D = a \cdot c \cdot 2^n + (a \cdot d + b \cdot c) \cdot 2^{\frac{n}{2}} + b \cdot d = x \cdot y$

function SMARTMULT(x, y, k)
  if $k \leq 4$ **then return** $x * y$
  else
    $[a,b] \leftarrow x$
    $[c,d] \leftarrow y$
    $A \leftarrow$SMARTMULT$(a,c,k-1)$
    $B \leftarrow$SMARTMULT$(b,d,k-1)$
    $C \leftarrow$SMARTMULT$(a+b,c+d,k-1)$
    return $A \cdot 2^n + (C-A-B) \cdot 2^{\frac{n}{2}} + B$

Analys

$$\left.\begin{array}{l} T(4) = \Theta(1) \\ T(n) = 3 \cdot T\left(\frac{n}{2}\right) + \Theta(n) \end{array}\right\} \Rightarrow T(n) \in O(n^{\log_2 3}) \in O(n^{1,58})$$

24.8

2 Totalsökning

Exhaustive search, totalsökning

- Gå igenom alla tänkbara lösningar och kolla om det är den sökta lösningen.Detta görs med fördel rekursivt.

Ofta är antalet tänkbara lösningar exponentiellt många och då går det bara att använda för små n. Totalsökning är en metod man tar till i sista hand.

Det svåraste med totalsökning är normalt att se till att man går igenom varje tänkbar lösning en (och helst inte mer än en) gång. Att sedan kolla om det är den sökta (eller optimala) lösningen brukar vara lätt.

Ibland kan man redan innan en tänkbar lösning är färdigkonstruerad se att den inte är en möjlig lösning. Då kan man strunta i att gå vidare med den och istället gå tillbaka och konstruera nästa möjliga lösning. Detta kallas backtracking.

24.9

TSP – handelsresandeproblemet

Hitta kortaste turen som passerar alla städer en gång.

Olika varianter av problemet:

- Generell TSPprobleminstans: graf med kantvikter
- Euklidisk TSP i dimension $d$probleminstans: städerna givna som koordinater i $\mathbb{R}^d$

24.10

Exempel: Lösning av generell TSP med totalsökning

function TSP$(n, d[1 \ldots n, 1 \ldots n])$
  $minlen \leftarrow \infty$
  for $i \leftarrow 1$ **to** n **do** $visited[i] \leftarrow$ **false**
  for $i \leftarrow 1$ **to** n **do**
    $perm[1] \leftarrow i$; $visited[i] \leftarrow$ **true**
    CHECKPERM$(2,0)$
    $visited[i] \leftarrow$ **false**
  return $minlen$
procedure CHECKPERM$(k, length)$
  if $k > n$ **then**
    $totalLength \leftarrow length + d[perm[n], perm[1]]$

```
        if totalLength < minlength then
            minlength ← totalLength
    else
        for i ← 1 to n do
            if ¬visited[i] then
                perm[k] ← i; visited[i] ← true
                CHECKPERM(k+1, length+d[perm[k−1], i])
                visited[i] ← false
```

24.11

3 Dynamisk programmering

Beräkning av Fibonaccital

- Fibonaccitalen har en rekursiv definition:$F_0 = 0 F_1 = 1 F_i = F_{i-1} + F_{i-2}$ för $i > 1$
- Som rekursiv algoritm (första försöket):

```
function BINARYFIB(k)
    if k < 2 then
        return k
    else
        return BINARYFIB(k−1)+BINARYFIB(k−2)
```

24.12

Analys av Fibonacci med binär rekursion

- Låt n_k beteckna antalet rekursiva anrop gjorda av BinaryFib(k)
  - $n_0 = n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
- Värdet åtminstone fördubblas för vartannat värde på n_k. D.v.s $n_k > 2^{k/2}$. Antalet anropet växer exponentiellt!

24.13

En bättre algoritm för Fibonaccital

- Använd dynamisk programmering istället:

```
function DYNPROGFIB(k)
    int F[0..k]
    F[0] = 0
    F[1] = 1
    for i = 2 to k do
        F[i] = F[i-2] + F[i-1]
    return F[k]
```

- Går i tid $O(k)$

24.14

Dynamisk programmering

- Tillämpbart på problem som vid första påseende verkar kräva mycket tid (möjligtvis exponentiellt) förutsatt att vi har:
  - Enkla delproblem: delproblemen kan definieras i termer av några få variabler
  - Optimalitet hos delproblemen: det globalt optimala värdet kan definieras i termer av optimala delproblem
  - Överlapp hos delproblemen: delproblemen är inte oberoende, istället överlappar de (och borde därför konstrueras botten-upp)

24.15

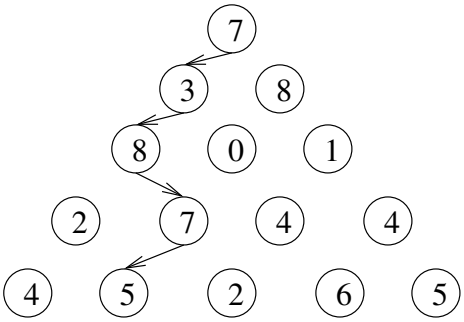
Dynamisk programmering

- Bestäm strukturen hos optimal lösning
- Ställ upp rekursion för optimalvärdet
- Beräkna delproblemens optimalvärden från små till stora
- Konstruera optimallösningen (ev. med extra information som lagras i steg 3)

24.16

Exempel: Maximal triangelstig

Problem: hitta stigen från toppen till botten som maximerar summan av de ingående talen.



$n =$ antalet rader finns 2^{n-1} stigar att pröva $a_{ij} =$ element nr j på rad i

24.17

Dynamisk programmeringslösning

- Optimala lösningens strukturstig uppbyggd av delstigar
- RekursionLåt $V[i,j] =$ värdet på bästa stigen från a_{ij} ner till rad n

$$V[i,j] = \begin{cases} a_{ij} & \text{om i=n (sista raden)} \\ a_{ij} + \max(V[i+1,j], V[i+1,j+1]) & \text{annars} \end{cases}$$

- Beräkning

for $j = 1$ **to** n **do**
  $V[n,j] \leftarrow a_{nj}$
for $i = n-1$ **downto** 1 **do**
  for $j = 1$ **to** i **do**
    $V[i,j] \leftarrow a_{ij} + \max(V[i+1,j], V[i+1,j+1])$
return $V[1,1]$

Tidskomplexitet: $O(n^2)$

24.18

Delsekvenser

- En *delsekvens* av en sträng $x_0x_1x_2\ldots x_{n-1}$ är en sträng på form $x_{i_1}x_{1_2}\ldots x_{i_k}$, där $i_j < i_{j+1}$
- Inte samma sak som delsträng!

Exempel: ABCDEFGHIJK

- Delsträng: CDEF
- Delsekvens: ACEGIJK
- Delsekvens: DFGHK
- Inte delsekvens: DAGH

24.19

Längsta gemensamma delsekvensproblemet (LCS)

- Givet två strängar X och Y är längsta gemensamma delsekvensproblemet (LCS) att hitta den längsta delsekvensen gemensam för X och Y
- Har tillämpningar vid testning av överensstämmelse mellan DNA-strängar (alfabetet är {A,C,G,T})

Exempel
ABCDEFG och XZACKDFWGH har ACDFG som en längsta gemensam delsekvens

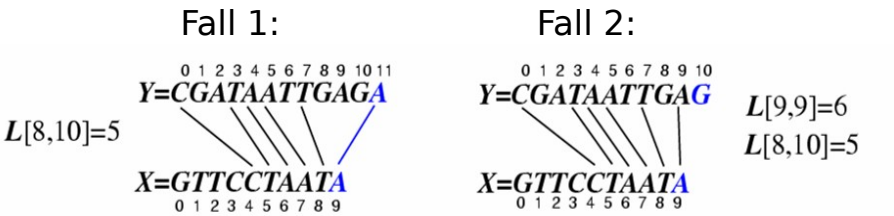
24.20

Ett dåligt sätt att närma sig LCS-problemet

- En brute-forcelösning
  - Enumerera alla delsekvenser av X
  - Testa vilka som också är delsekvenser av Y
  - Välj den längsta
- Analys
  - Om X har längd n innehåller den 2^n delsekvenser
  - Det här en algoritm som kräver exponentiell tid!

Dynamisk programmering och LCS-problemet

- Låt $L[i,j]$ vara längden av den längsta gemensamma delsekvensen av $X[0\ldots i]$ och $Y[0\ldots j]$
- Tillåt -1 som index, så att $L[-1,k]=0$ och $L[k,-1]=0$ för att indikera att *null*-delen av X eller Y inte alls matchar den andra strängen
- Då kan vi definiera $L[i,j]$ i det allmänna fallet som följer:
  - Om $X[i]=Y[j]$ är $L[i,j]=L[i-1,j-1]+1$ (vi kan lägga till den här träffen)
  - Om $X[i]\neq Y[j]$ är $L[i,j]=\max\{L[i-1,j],L[i,j-1]\}$ (vi har ingen träff här)



En algoritm för LCS

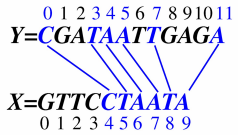
function LCS($X,Y,|X|=n,|Y|=m$)
  for $i=-1$ **to** n-1 **do**
    $L[i,-1]=0$
  for $j=0$ **to** m-1 **do**
    $L[-1,j]=0$
  for $i=0$ **to** $n-1$ **do**
    for $j=0$ **to** $m-1$ **do**
      if $X[i]=Y[j]$ **then**
        $L[i,j]=L[i-1,j-1]+1$
      else
        $L[i,j]=\max\{L[i-1,j],L[i,j-1]\}$
  return array L

Visualisering av LCS-algoritmen

m

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

n



24.24

Analys av LCS-algoritmen

- Vi har två nästlade loopar
  - Den yttre itererar n ggr
  - Den inre itererar m ggr
  - I varje iteration av innerloopen utförs en konstant mängd arbete
  - Alltså är den totala exekveringstiden $O(nm)$
- Svaret finns i $L[n,m]$ (och delsekvensen kan återskapas utifrån tabellen L)

24.25

4 Giriga algoritmer

Giriga algoritmer

Algoritmer som löser en bit av problemet i taget. I varje steg görs det som ger bäst utdelning/kostar minst.

- Den giriga metoden är ett allmänt paradigm för algoritmdesign som bygger på följande:
  - konfigurationer: olika val, samlingar eller värden att hitta
  - målfunktion: konfigurationer tilldelas en poäng som vi vill maximera eller minimera
- Det fungerar bäst applicerat på problem med *greedy-choice*-egenskapen:
  - en globalt optimal lösning kan alltid hittas genom en serie lokala förbättringar utgående från en begynnelsekonfiguration

För många problem ger giriga algoritmer inte optimala lösningar utan kanske hyfsade approximativa lösningar.

24.26

Ge växel

- Du behöver ge x cent i växel, genom att använda mynt med valörerna 1, 5, 10 och 25 cent. Vilket är det minsta antalet mynt som behövs för att ge växeln?
- Girig approach:
  - Ta så många 25-centsmynt som möjligt, sedan
  - ta så många 10-centmynt som möjligt, sedan
  - ta så många 5-centsmynt som möjligt, sedan
  - ta så många 1-centsmynt som behövs för att bli färdig.
- Exempel: 99 cent = $3*25$ cent + $2*10$ cent + $0*5$ cent + $4*1$ cent
- Är detta optimalt?
- 1, 5, 10 och 25 cent: girig algoritm ger optimum
- 1, 6 och 10 cent: girig algoritm ger inte optimum

24.27