Algoritmer, datastrukturer och komplexitet, hösten 2013

Uppgifter till övning 3

Dekomposition och dynamisk programmering

Max och min med dekomposition I vektorn $v[1..n]$ ligger n tal. Konstruera en dekompositionsalgoritm som tar reda på det största och det minsta talet i v. Algoritmen ska använda högst $\lceil 3n/2 \rceil - 2$ jämförelser mellan v-element. Antalet tal i v behöver inte vara en tvåpotens.

Matrismultiplikation Strassens algoritm multiplicerar två $n \times n$-matriser i tid $O(n^{2.808})$ genom dekomposition i 2×2-blockmatriser. Anledningen till att Strassens algoritm går snabbare än $O(n^3)$ är att den gör sju multiplikationer istället för åtta för att bilda produktmatrisen.

En annan idé är att göra dekomposition i 3×3-blockmatriser istället. Viggo försökte för ett par år sedan hitta det minimala antalet multiplikationer som krävs för att multiplicera två 3×3-matriser. Han lyckades nästan komma fram till 22 multiplikationer. Om han hade lyckats, vilken tidskomplexitet hade det gett för multiplikation av två $n \times n$-matriser?

Majoritet med dekomposition Indata är i denna uppgift en array A med n element. Konstruera och analysera en algoritm som tar reda på om något element i arrayen A är i majoritet, det vill säga förekommer mer än $n/2$ gånger, och i så fall returnerar det. Algoritmen ska vara en dekompositionsalgoritm och ha tidskomplexiteten $O(n \log n)$. Enda tillåtna jämförelseoperationen på element i A är $=$. Det finns alltså ingen ordningsrelation mellan elementen.

Mobil Konstruera en algoritm som balanserar en mobil! Mobilen beskrivs som ett binärträd. Löven motsvarar kulor som hänger underst i mobilen. Inre noder motsvarar tunna raka pinnar i vars ändar är fastknutet trådar som sönerna hänger i. Indata är ett binärträd med vikter i noderna. Vikten i varje löv är motsvarande kulas massa. Vikten i varje inre nod är motsvarande pinnes längd i cm. Resultatet av algoritmen ska vara samma binärträd men där det i varje inre nod också står hur många centimeter från vänstra ändpunkten tråden som fäster pinnen i ovanförliggande pinne ska sitta för att mobilen ska bli balanserad.

Talföljder Anta att du har fått rekursionen för en talföljd presenterad för dig. Skriv pseudokoden för en dynamisk programmeringsalgoritm för att beräkna S_n om

a)

$$S_n = \begin{cases} 1 & \text{om } n = 0, \\ 2(S_{n-1}) + 1 & \text{annars.} \end{cases}$$

b)

$$S_n = \begin{cases} 4 & \text{om } n = 1, \\ 5 & \text{om } n = 2, \\ \max\left(S_{n-1}, S_{n-2}, S_{n-1} - S_{n-2} + 7\right) & \text{annars.} \end{cases}$$

c)

$$S_n = \begin{cases} 4 & \text{om } n = 1, \\ 5 & \text{om } n = 2, \\ \max\left(S_{n-2}, S_{n-1} - S_{n-2} + 7\right) & \text{annars.} \end{cases}$$

Skriv ned de 7 första talen. Vad behöver man spara under beräkningens gång?

Generaliserade talföljder

a) Tvådimensionell rekursion utan indata

Givet följande rekursion, konstruera en algoritm som beräknar $M[i,j]$ med dynamisk programmering. Argumentera för att algoritmens beräkningsordning fungerar.

$$M[i,j] = \begin{cases} 0 & \text{om } i = 0 \text{ eller } j = 0, \\ M[i-1,j-1]+i & \text{annars.} \end{cases}$$

b) 2D-rekursion med indata

Givet två strängar a och b av längd n respektive m, låt a_i vara tecknet på position i i sträng a och på motsvarande sätt b_j vara tecknet på position j i b, hitta en beräkningsordning och skriv en algoritm som utför följande rekursion med hjälp av dynamisk programmering.

$$M[i,j] = \begin{cases} 0 & \text{om } i = 0 \text{ eller } j = 0, \\ M[i-1,j-1]+1 & \text{om } a_i = b_j, \\ 0 & \text{annars.} \end{cases}$$

Lösningar

Lösning till Max och min med dekomposition
När man bara har två tal räcker det med en enda jämförelse för att både hitta det största och det minsta talet.

```
MinMax(v,i,j)=
  if i=j then return (v[i],v[i])
  else if i+1=j then
     if v[i]<v[j] then return (v[i],v[j])
     else return (v[j],v[i])
  else
     m:=Floor((j-i)/2)
     if Odd(m) then m:=m+1;
     (min1,max1):=MinMax(v,i,i+m-1);
     (min2,max2):=MinMax(v,i+m,j);
     min:=(if min1<min2 then min1 else min2);
     max:=(if max1>max2 then max1 else max2);
     return (min,max);
```

Beräkningsträdet kommer att få $\lceil n/2 \rceil$ löv och $\lceil n/2 \rceil - 1$ inre noder. Om n är jämnt är alla $n/2$ löv tvåelementsföljder och gör därför en jämförelse. Om n är udda är ett löv (det högraste) ett enstaka element som inte kräver någon jämförelse. I löven görs alltså $\lfloor n/2 \rfloor$ jämförelser. I varje inre nod görs två jämförelser, alltså sammanlagt $2\lceil n/2 \rceil - 2$ stycken. Totalt får vi $\lfloor n/2 \rfloor + 2\lceil n/2 \rceil - 2 = \lceil 3n/2 \rceil - 2$ stycken jämförelser.

Det går faktiskt att visa att problemet inte kan lösas med färre jämförelser. □

Lösning till Matrismultiplikation
Rekursionsekvationen blir $T(n) = 22 \cdot T(n/3) + O(n^2)$. Mästarsatsen ger lösningen $T(n) = O(n^{\log_3 22}) = O(n^{2.814})$. □

Lösning till Majoritet med dekomposition
Om det finns ett majoritetselement måste det vara i majoritet i åtminstone ena halvan av arrayen. Rekursiv tanke: Kolla majoritet rekursivt i vänstra och högra halvan och räkna sedan hur många gånger halvarraysmajoritetselementen förekommer i hela arrayen. Om något element är i total majoritet returneras det.

```
Majority(A[1..n]) =
    if n = 1 then return A[1]
    mid← ⌈(n + 1)/2⌉
    majInLeft← Majority(A[1..mid−1])
    majInRight← Majority(A[mid..n])
    if majInleft=majInRight then return majInLeft
    noOfMajInLeft← 0
    noOfMajInRight← 0
    for i ← 1  to n do
       if A[i] =majInLeft then noOfMajInLeft←noOfMajInLeft+1
       else if A[i] =majInRight then noOfMajInRight←noOfMajInRight+1
    if noOfMajInLeft≥ m then return majInLeft
    if noOfMajInRight≥ m then return majInRight
    else return NULL
```

Tidskomplexitet: Två rekursiva anrop med halva arrayen följt av efterarbetet $O(n)$ ger med mästarsatsens hjälp tidskomplexiteten $O(n \log n)$. □

Lösning till Mobil
Betrakta en pinne av längd l i mobilen. Anta att vikten v hänger i vänstra änden och vikten w i den högra. Låt x vara avståndet från pinnens vänstra ände till tråden som fäster pinnen i ovanförliggande pinne. För att momenten ska ta ut varandra krävs enligt mekaniken att $xv = (l - x)w$, det vill säga att $x = lw/(v + w)$. Vi kan nu beräkna x för varje pinne rekursivt nerifrån och upp i trädet. Låt den rekursiva funktionen returnera vikten av mobilen som hänger i den aktuella pinnen. Algoritmen tar linjär tid.

```
Balance(p)=
        if p.left =NIL then return p.num
        left ← Balance(p.left)
        right ← Balance(p.right)
        p.x ← p.num · right/(left + right)
        return left + right
```

□

Lösning till Talföljder
a) 1, 3, 7, 15, 31, 63, 127

För att beräkna ett tal behöver man bara känna till talet innan. Beräkningsordningen blir från basfallet och vidare ”uppåt”.

```
Talfoljda(n) =
    v = 1
    for i = 1 to n do
      v = 2*v+1
    return v
```

b) 4, 5, 8, 10, 10, 10, 10

För att beräkna ett tal behöver man känna till de två föregående talen. Beräkningsordningen blir från basfallet och vidare ”uppåt”.

```
Talfoljdb(n) =
  if n = 1 then return 4
  elif n = 2 then return 5
  else
    v0 = 4
    v1 = 5
    for i = 2 to n do
      v = max(v1, v0, v1-v0+7)
      v0 = v1
      v1 = v
    return v
```

c) 4, 5, 8, 10, 9, 10, 9

För att beräkna ett tal behöver man känna till de två föregående talen. Beräkningsordningen blir från basfallet och vidare ”uppåt”.

```
Talfoljdc(n) =
  if n = 1 then return 4
  elif n = 2 then return 5
  else
    v0 = 4
    v1 = 5
    for i = 2 to n do
      v = max(v0, v1-v0+7)
      v0 = v1
      v1 = v
    return v
```

□

Lösning till Generaliserade talföljder

a) 2D-rekursion utan indata

Eftersom vi inte vet vad vi letar efter, låter vi algoritmen returnera hela M. Det är inte vad man typiskt skulle göra om algoritmen skulle användas för att svara på någon specifik fråga. Oavsett vad frågan skulle vara, är det (första) vi behöver kunna göra just att beräkna M. När hela matrisen är ifylld kan vi svara på olika frågor om den, som vilket det största värdet i matrisen är, eller hur många gånger det största värdet förekommer.

Vi måste börja med ett basfall, men om vi tänker oss en matris M som ska fyllas i med rätt värden, så har hela översta raden (där $i = 0$) och hela vänstra kolumnen (där $j = 0$) värdet 0, som inte beräknas rekursivt. En typisk beräkningsordning för ett program skulle vara en rad i taget uppifrån eller en kolumn i taget från vänster till höger. Vi väljer rad för rad, och sätter värdena på första raden. Därefter loopar vi över alla andra rader och sätter värdena kolumn för kolumn. Beräkningsordningen fungerar, eftersom den information vi behöver ha för att beräkna ett nytt värde utom basfallen bara är vad som stått på tidigare rader, samt indexet för den rad vi befinner oss på. Vi kommer att kunna beräkna hela M. Här är vår algoritm:

for $j \leftarrow 0$ **to** n
  $M[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** m

```
    M[i,0] ← 0
    for j ← 1  to n
        M[i,j] ← M[i-1,j-1] + i
return M
```

Tiden domineras av den nästlade for-slingan och är alltså $\Theta(nm)$.

b) 2D-rekursion med indata
Vi börjar med basfallen. Alla element på översta raden och alla i kolumnen längst till vänster uppfyller villkoret för basfall, och kan därför sättas till 0. Vi väljer att börja med första raden. Därefter beräknar vi värdena på alla följande rader uppifrån och ner, kolumn för kolumn. Enligt rekursionen är värdet 0 om $a_i \neq b_j$, och annars beräknas det med hjälp av tidigare uträknade värden. Vi kan alltid titta på föregående rad $(i-1)$, eftersom vi beräknar en rad i taget. På den raden kan vi titta på vilka element vi vill, eftersom hela raden är ifylld, speciellt är det tillåtet att titta på det i kolumn $j-1$. Beräkningsordningen fungerar alltså. Algoritmen kan se ut så här:

```
for j ← 0  to n
    M[0,j] ← 0
for i ← 1  to m
    M[i,0] ← 0
    for j ← 1  to n
        if a_i = b_j then
            M[i,j] ← M[i-1,j-1] + 1
        else M[i,j] ← 0
return M
```

Tiden blir precis som i det förra exemplet $\Theta(nm)$. □
