2D1458, Problemlösning och programmering under press

Föreläsning 1: Dekomposition, giriga algoritmer och dynamisk programmering

Datum: 2007-09-04
Skribent(er): Anders Malm-Nilsson och Niklas Nummelin
Föreläsare: Mikael Goldmann

Denna föreläsning tog upp tre av de grundläggande metoderna för att konstruera och klassifiera algoritmer. Dessa metoder benäms som dekomposition, giriga algoritmer samt dynamisk programmering.

1 Dekomposition

Dekomposition, eller ”Söndra och Härska” (från engelskans ”Divide and Conquer”) är en metod som går ut på att dela upp ett större problem i mindre problem. Detta upprepas till dess att alla delproblem har nått en lämplig komplexitetsgrad. Då alla delproblem är lösta kan resultaten från dessa kombineras till det slutgiltiga svaret på problemet. Dock förutsätter detta att delproblemen är av sådan karkatär att deras optimala lösning kan kombineras till en optimal lösning för hela problemet.

1.1 Exempel 1: Skyline

Givet n hus som projicerats på ett plan, skall deras skyline beräknas. Med skyline menas den övre kontur som kan fås ur projektionen. Varje hus representeras av en position x, bredd b och höjd h. Utdata är en skyline representerad av m tupler. Varje tupel är en position x och höjd h.

Vi börjar med att visa den pragmatiska metoden för att kunna komma fram till en bättre lösning. Den går ut på att kontrollera alla de punkter som varje hus spänner upp på tallinjen och spara undan det hitills högsta värdet hos varje punkt. Efter att detta har utförts för alla hus kan den beräknade höjdbuffern gås igenom från början för att generera en lista med utdata. Tidskomplexiteten för detta blir alltså $(n_{max} \cdot b_{max} + n_{max} + b_{max})$ eller $O(n_{max} \cdot b_{max})$. Den slutgiltiga datamängden kommer att vara sorterad efter position.

Algoritm 1: Pragmatisk lösning till Skylineproblemet.
Input: $U = [x_i, b_i, h_i]_{i=1}^{n}, n \leq 5000; x, b, h \leq 10000;$ där $n, x, b, h \in \mathbf{N}$
Output: $A = [x_i, h_i]_{i=1}^{m}$

```
SLPragmatisk(U)
(1)   H[0..20000] ← {0..0};
(2)   foreach (x_i, b_i, h_i) ∈ U
(3)     for i = x_i to x_i + b_i
(4)       if H_i < h_i
(5)         H_i ← h_i
(6)   LH ← 0
(7)   A ← ∅
(8)   foreach h_i ∈ H
(9)     if h_i ≠ LH
(10)      A ← A ∪ (i, h_i)
(11)      LH ← h_i
(12)  if LH ≠ 0
(13)    A ← A ∪ (19999, 0)
```

Då ut-datamängden skall bli sorterad av algoritmen kan man utnyttja sortering hos Mergesort för att skapa en lösning. Sorteringsalgoritmen Mergesort går i $O(n \log(n))$. Den dekompositionerade Skylinealgoritmen är en vidareutveckling av Mergesort och går därför lika fort.

Algoritm 2: Algoritm 2 - Mergefunktionen.
Input: $U_1, U_2 = [x_i, h_i]_{i=1}^{n}, n \leq 5000; x, h \leq 10000$; där $n, x, h \in \mathbf{Z}$
Output: $A = [x_i, h_i]_{i=1}^{m}$

```
SLDekompMerge(U)
(1)    A = ∅
(2)    xposition ← 0
(3)    h1 ← 0
(4)    h2 ← 0
(5)    while U1 ≠ ∅ and U2 ≠ ∅
(6)        (x1_i, h1_i) ← U1.Front
(7)        (x2_i, h2_i) ← U2.Front
(8)        if x1_i < x2_i
(9)            xposition ← x1_i
(10)           h1 ← h1_i
(11)           if h1 < h2
(12)               A ← A ∪ (xposition, h1)
(13)           else
(14)               A ← A ∪ (xposition, h2)
(15)           U1 ← U1 \ {(x1_i, h1_i)}
(16)       else
(17)           xposition ← x2_i
(18)           h2 ← h2_i
(19)           if h1 < h2
(20)               A ← A ∪ (xposition, h1)
(21)           else
(22)               A ← A ∪ (xposition, h2)
(23)           U2 ← U2 \ {(x2_i, h2_i)}
(24)       if x1_i = x2_i
(25)           U1 ← U1 \ {(x1_i, h1_i)}
```

Algoritm 3: Lösning till Skylineproblemet, med dekomposition.
Input: $U = [x_i, b_i, h_i]_{i=1}^{n}, n \leq 5000; x, b, h \leq 10000;$ där $n, x, b, h \in \mathbf{Z}$
Output: $A[x_i, h_i]_{i=1}^{m}$

```
SLDekomp(U)
(1)   if |U| = 1
(2)     (x, b, h) ← U.Front
(3)     A ← ∅
(4)     A ← A ∪ (x, h)
(5)     A ← A ∪ (x + b, 0)
(6)     return A
(7)   return SLDekompMerge(SLDekomp(U[0..|U|/2]), SLDekomp(U[|U|/2+
      1]..|U|))
```

2 Giriga algoritmer

Med giriga algoritmer menas sådana problemlösningsalgoritmer som vid varje val tar det beslut som ger mest utdelning. Då denna typ av algoritm alltid strävar efter den lokalt bästa lösningen innebär det att den lösningen inte alltid blir globalt optimal. Vi kommer dock att använda giriga algoritmer för att lösa problem där vi kan visa att den giriga algoritmen faktiskt ger en optimal lösning.

2.1 Exempel 2: Intervalltäckning

Givet en mängd intervall mellan heltalspunkterna L och R på en tallinje, finn det minsta antalet intervall som behövs för att täcka alla heltalspunkter mellan L och R.

Algoritm 4: Girig algoritm för intervalltäckning.
Input: $U = [a_i, b_i]_{i=1}^{n}, L, R; \cup_{i \in A}[a_i, b_i] \supseteq [L, R] \cap \mathbf{Z}$
Output: $A \subseteq \{x_i, h_i\}$

```
IntervallTäckning(U)
(1)   A ← ∅
(2)   while L < R
(3)     longest ← 0;
(4)     foundInterval ← false
(5)     foreach (a_i, b_i) ∈ U
(6)       if (a_i ≤ L and b_i ≥ L)
(7)         if b_longest < b_i
(8)           longest ← i
(9)           foundInterval ← true
(10)          break
(11)    if foundInterval = false
(12)      return ∅
(13)    A ← A ∪ {longest}
(14)    L ← b_longest + 1
```

Tidskomplexiteten för algoritmen blir alltså $O(n^2)$, då en linjärsökning sker för varje interval. Genom att sortera listan kan sökningen göras snabbare då sökintervallet hela tiden kommer att förminskas. Man kan även utföra sökningen med hjälp av binärsökning, vilket resulterar i tidskomplexiteten $O(n \log(n))$.

2.2 Exempel 3: Pianoflyttarna

P stycken pianoflyttare har tagit på sig att flytta m pianon under n dagar (dag 1 är en måndag). Tyngden gör att två flyttare endast kan flytta ett piano per dag. Varje piano kan dessutom bara flyttas mellan två datum, a och b. Är det möjligt för dem att flytta alla pianon? Är det dessutom möjligt för dem att vara lediga på helgerna?

Algoritm 5: Girig algoritm för pianoflyttarna.
Input: $U = [a_i, b_i]_{i=1}^{m}, P, n; 1 \leq a; b \leq 100; P, m, n, a, b \in \mathbf{Z}$
Output: *boolean*

```
PIANOFLYTT(U)
(1)   for i = 0 to n
(2)     if (i mod 7 = 6) or (i mod 7 = 0)
(3)       continue
(4)     finnsP ← P/2
(5)     dag ← i
(6)     while finnsP > 0
(7)       foreach (a_j, b_j) ∈ U
(8)         if a_j > dag
(9)           continue
(10)        if b_j < dag
(11)          return false
(12)        if b_j = dag
(13)          if finnsP = 0
(14)            if dag = i then return false
(15)                       else break
(16)          else
(17)            U ← U \ {(a_j, b_j)}
(18)            if U = ∅
(19)              return true
(20)            finnsP ← finnsP − 1
(21)      dag ← dag + 1
(22)      if dag = n
(23)        return true
(24)    if dag > i
(25)      i ← dag − 1
```

Pianoflyttarna arbetar på det sättet att de alltid tar de mest kritiska jobben, d.v.s. de flyttar de pianon som är närmast sitt slutdatum. Först flyttas de pianon som måste flyttas idag, sen de som måste flyttas imorgon och så vidare tills dess att det inte finns några pianoflyttare kvar idag. Genom att man alltid tar de mest kritiska jobben först så får man garanterat en optimal lösning. Tidskomplexiteten för detta blir $O(n^2)$. Om listan med pianoflyttningsjobb sorteras efter slutdatum kan analysen istället göras i linjär tid. Tidskomplexiteten blir då $O(n \log(n))$.

3 Dynamisk programmering

Dynamisk programmering är en metod som går ut på att ta till vara resultatet från tidigare lösta delproblem, så att dessa inte behöver lösas igen. Vi skall nu studera ett problem som enkelt kan tidsoptimeras med hjälp av dynamisk programmering.

3.1 Exempel 4: Binomialtermen

Vi vill beräkna binomialtermen. Med matematisk notation kan den rekursiva algoritmen beskrivas som:

$$\binom{n}{k} = \begin{cases} 1 & \text{om } k = 0 \text{ eller } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{annars} \end{cases}$$

Pseudokoden för den rekursiva algoritmen blir då:

Algoritm 6: Rekursiv algoritm för att finna $\binom{n}{k}$.
Input: $n, k \in \mathbf{Z}$ där n $\geq k \geq 0$
Output: $\binom{n}{k}$
BINOM(n, k)
(1) **if** $k = 0$ or $k = n$
(2) **return** 1
(3) **else**
(4) **return** BINOM$(n-1, k-1)$ + BINOM$(n-1, k)$

Denna kräver alltså $O\left(\binom{n}{k}\right)$ operationer, då alla basfall, d.v.s. alla 1:or, är de som skall summeras till resultatet. Det är $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ operationer. Det värsta fallet är alltså $k = floor(\frac{n}{2})$ då $k \leq n$ och $k \geq 0$, d.v.s. O($\frac{n!}{(floor(\frac{n}{2})!)\cdot(ceil(\frac{n}{2})!)}$). Med hjälp av dynamisk programmering kan detta problem optimeras till $O(nk)$ operationer. Man utnyttjar att det, då rekursionsträdet expanderas, finns flera överlappande grenar, vilka därför bara behöver expanderas och beräknas en gång. Denna teknik brukar vanligtvis kallas för dynamisk programmering och kan göras på två olika sätt, med sina för och nackdelar:

Algoritm 7: Algoritm för att finna $\binom{n}{k}$, top-down med memoisering.
Input: $n, k \in \mathbf{Z}$ där n $\geq k \geq 0$
Output: $\binom{n}{k}$
BINOM(n, k)
(1) **if** $k = 0$ or $k = n$
(2) **return** 1
(3) **else if** computed$[n, k] \neq true$
(4) computed$[n, k] = true$
(5) tab$[n, k]$ = BINOM$(n-1, k-1)$ + BINOM$(n-1, k)$
(6) **return** tab[n,k]

”Top-down”-metoden fungerar så att värden beräknas och sparas undan i en tabell först när de efterfrågas och inte redan har beräknats. Metoden att spara undan resultatet från funktionsanrop för återanvändning brukar vanligtvis kallas ”memoisering” (från engelskans ”memoization”). Detta är bra om man har en stor spridning på indata och inte vet vilka tal som kommer att behöva beräknas.

Algoritm 8: Algoritm för att finna $\binom{n}{k}$, bottom-up.
Input: $n, k \in \mathbf{Z}$ där n $\geq k \geq 0$
Output: $\binom{n}{k}$
BINOM(n, k)
(1) **for** $i = 0$ **to** n
(2) tab$[i, 0]$ = tab$[i, i] = 1$
(3) **for** $i = 2$ **to** n
(4) **for** $j = 1$ **to** min(i-1, k)
(5) tab$[i, j]$ = tab$[i - 1, j - 1]$ + tab$[i - 1, j]$
(6) **return** tab$[n, k]$

Med ”Bottom-up”-metoden förberäknar man istället hela tabellen. Detta är bra då man är intresserad av att få ut många eller alla av de värden som blir beräknade i tabellen. Ibland kan det även vara en bra metod att använda om många eller alla värden i tabellen behövs för att nå resultatet. Observera att tabellen bara behövs beräknas en gång. Sedan kan värdena hämtas i konstant tid.

3.2 Exempel 5: Kretskortskonstruktion

Två chip med n ben är konstruerade så att varje chip har alla sina ben liggandes mot det andra chipets ben. Benen på båda kretsarna är numrerade men tillverkaren av den ena kretsen lyckades inte lägga benen i ordning. Tillverkaren som nu skall använda kretsarna kan inte dra ledningarna mellan benen i kors utan måste lägga korsande ledningar som olika lager. Vi ska undersöka hur man kan gå tillväga för att hitta det största antalet kopplingar som kan finnas i ett lager (längsta växande delsekvens).

Två algoritmer skall nu visas som båda använder sig utav kretsuppsättningen nedan.

y	x
1	1
2	3
3	6
4	4
5	2
6	5

Man kan skapa en växande delsekvens med hjälp av en girig algoritm, där man tar de kopplingar som fungerar om man börjar uppifrån i listan och går neråt. Denna algoritm finner endast en växande delsekvens. Det man kan konstatera är att om man finner den längsta växande delsekvensen har man funnit det maximala antalet kopplingar som kan finnas i lagret. Om y styr kopplingarna ger den giriga algoritmen kopplingar mellan $(1, 2, 5)$ medan x ger $(1, 3, 6)$. Då man undersöker tabellen finner man att den längsta växande delsekvensen för denna krets är $(1, 3, 4, 5)$, så ingen av lösningarna gav den längsta delsekvensen. Den giriga algoritmen gav alltså ingen optimal lösning.

För att kunna finna den längsta växnade delsekvensen utan att behöva göra en totalsökning, så använder vi oss utav dynamisk programmering.

Man använder sig utav en vektor last[] där last[i] är det lägsta värdet som kan avsluta en delsekvens som är i lång. last[0] initieras till $-\infty$ medan resterande poster i last initieras till ∞. För varje x[i] måste man nu finna det första j så att last[$j-1$] $<$ x[i] $\leq$ last[j]. De värden som last[] är initierad till gör att, för varje x[], en sådan alltid kommer att finnas. När denna är funnen sätter man last[j] = x[i]. Tabellen nedan visar hur algoritmen fungerar. Algoritmen lägger antingen elementet efter det sista non-∞ elementet i listan eller byter ut ett element i listan. I kolumnerna för 1, 3 och 6 byter algoritmen ut ∞ värden då värdet x[i] är större än raden ovanför men $<\infty$. 6:an får sedan byta plats med 4:an då $3 < 4 < 6$ och i nästa kolumn får 2:an gå in i stället för 3:an då $1 < 2 < 3$. I sista kolumen läggs sedan in en 5:a i slutet. Om binärsökning används för att finna det element som skall skrivas last-vektorn blir algoritmens tidskomplexitet O(n log(n)).

	start	1	3	6	4	2	5
last[0]	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
last[1]	∞	1	1	1	1	1	1
last[2]	∞	∞	3	3	3	2	2
last[3]	∞	∞	∞	6	4	4	4
last[4]	∞	∞	∞	∞	∞	∞	5
last[5]	∞	∞	∞	∞	∞	∞	∞
last[6]	∞	∞	∞	∞	∞	∞	∞

Tabellen ger oss nu antalet element i den maximalt växande delsekvensen. Detta ges av att räkna antalet element i sista kolumnen som inte är ∞ eller -∞. Genom att använda denna tabell kan även man ta fram den längsta växande delsekvensen, vilket inte visas här.