2D1458, Problemlösning och programmering under press

Föreläsning 1: Dynamisk programmering och giriga algoritmer

Datum: 2006-09-04
Skribent(er): Dmitry Palagin och Henrik Ygge
Föreläsare: Mikael Goldmann

Denna föreläsning började med en kort genomgång av domarsystemet Kattis som används i kursen och regler för samarbete vid laborationer och hemuppgifter. Sedan diskuterade vi algoritmkonstruktionmetoder såsom giriga algoritmer och dynamisk programmering.

1 Administration

1.1 Kattis

Domarsystemet Kattis finns på http://kattis.csc.kth.se. När man skickar in en lösning på ett problem till domaren kan man få följande svar:

- Accepted - Lösningen godkänd.
- Submission error - Problemet okänt eller felaktigt format på inskickningen.
- Compilation error - Kattis kunde inte kompilera filen.
- Illegal function - Vissa funktioner är inte tillgängliga i Kattis av säkerhetsskäl, t.ex. att läsa en fil från disken etc. Försöker man det får man detta fel.
- Runtime error - Fel vid körning.
- Memory limit exceeded - Programmet använder mer minne än vad som är tillåtet. Kan också visa sig som ett Runtime error.
- Output limit exceeded - Programmet genererar alldeles för mycket output till standard out.
- Time limit exceeded - Programmet hinner inte exekvera klart inom den förbestämda tidsgränsen. Observera att Kattis stänger av programmet i så fall.
- Wrong answer - Programmet svarar fel på testdata.

1.2 Betygsgränser

Observera att maximala antalet poäng på OVN1 och LAB1 är 80p respektive 108p. För betyg 3/4/5 krävs 24/36/48 poäng. Mikael var noga med att poängtera att även om maxpoängen är hög jämfört med betygsgränserna är det ändå inte lätt att få bra betyg i kursen, och att man inte behöver satsa på att ta alla poäng, men att det är viktigt att jobba kontinuerligt med kursen.

1.3 Samarbete

- Hemtal: Ska lösas individuellt. Att dela idéer är okej, men inte kod!
- Anteckningar: Skrivs i grupp med 2-3 personer.
- Problemsessioner: 2 pers/grupp, är det ojämnt antal studenter i tid till en session blir det en grupp med 3 personer, ingen ska alltså behöva göra en problemsession själv.
- Labbar: Löses i grupper om 1-3 personer, men alla måste skicka in en separat lösning där koden innehåller en kommentar som anger gruppmedlemmar.

Det finns en nyhetsgrupp för kursen där studenter kan diskutera. Den heter nada.kurser.popup06 och finns på inn.nada.kth.se.

2 Algoritmkonstruktion

2.1 Dekomposition

Går ut på att dela upp problemet i mindre delproblem, lösa dem och sedan sätta ihop resultatet. Exempel är Quicksort och Mergesort. Detta är förmodligen bekant från tidigare kurser.

2.2 Giriga algoritmer

Dessa är typiska för att hitta en lösning som är "billigast" eller "bäst". Idén går ut på att utvidga en partiell lösning stegvis, lokalt optimalt, och se till att valet inte förstör möjligheten att utvidga till en optimal lösning på slutet. Ett exempel är Kruskals algoritm för *minimum spanning tree*. Den väljer i varje steg den billigaste kanten att utvidga som inte bildar en cykel med de redan valda kanterna.

Ett annat exempel är *täckning med intervall* [1].

> **Algoritm 1:** Täckning med intervall
> **Input:** $[L, R]$, $[a_1, b_1]$, $[a_2, b_2]$, ... , $[a_n, b_n]$; där $L, R, a_i, b_i \in \mathbf{Z}$, $a_i \leq b_i$ och $L \leq R$.
> **Output:** En minimalt stor delmängd $A \subseteq \{1, 2, ..., n\}$ så att $\cup_{i \in A}[a_i, b_i] \supseteq \mathbf{Z} \cap [L, R]$. Mata ut "fail" om det inte finns någon sådan delmängd.
>
> (1) Bland alla $[a_i, b_i]$ där $a_i \leq L \leq b_i$ välj det i där b_i är störst. Om det inte finns $[a_i, b_i]$ som täcker L **return** fail
> (2) $A \leftarrow A \cup \{i\}$
> (3) $L \leftarrow b_i + 1$
> (4) **if** $L > R$
> (5) **return** A
> (6) goto (1)

[1] Ofta vill man täcka hela det reella intervallet $[L, R]$, inte bara heltalspunkterna, och då behöver man modifiera den givna algoritmen.

För att övertyga oss om att algoritmen ovan är korrekt bevisar vi följande påstående: Första gången vi väljer intervall $[a_i, b_i]$ girigt väljs ett intervall som kan ingå i en optimal lösning.

Antag att vi väljer $[a_j, b_j]$ först och att det existerar en optimal lösning A som inte innehåller detta intervall. Då $\exists i \in A$ så att $L \in [a_i, b_i]$, alltså finns det ett annat intervall i i den optimala lösningen A som täcker L.

Betrakta

$$A' = (A \setminus \{i\}) \cup \{j\}$$

I och med att j har det största övre gränsen b av alla intervall som täcker L är det lätt att se att

$$[L, R] \cap [a_i, b_i] \in [L, R] \cap [a_j, b_j]$$

alltså förlorar vi ingenting på att välja $[a_j, b_j]$ framför $[a_i, b_i]$.

Vi kan lätt klara av det här i $O(n \log n)$ tid genom att först sortera alla intervall så att $a_1 \leq a_2 \leq ... \leq a_n$. Sen går vi igenom den sorterade listan och väljer det intervall med högst b_i där $a_i \leq L$. Sorteringen tar $O(n \log n)$ tid, och sedan stegar vi genom listan i $O(n)$ tid. Totalt alltså $O(n \log n)$ tid.

2.3 Dynamisk programmering

Idén med dynamisk programmering är att få ner den, ofta exponentiella, tiden som krävs för beräkna ett rekursivt uttryck genom att spara framräknade värden och använda dessa om och om igen istället för att beräkna dem på nytt.

Ett exempel är att beräkna $\binom{n}{k}$. Formeln är

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

Den vanliga rekursiva lösningen är långsam. Den tar $O(\binom{n}{k})$ tid att exekvera. Vi kan göra samma sak på $O(nk)$ genom att använda dynamisk programmering.

Algoritm 2: Variant 1 - top-down med memoisering
Input: $n, k \in \mathbf{Z}$ där $n \geq k \geq 0$
Output: $\binom{n}{k}$

```
BIN(n, k)
(1)   if (n==k || k==0)
(2)     return 1
(3)   if (!computed[n,k])
(4)     tab[n, k] ← BIN(n − 1,k − 1)+BIN(n − i, k)
(5)     computed[n, k] = true
(6)   return tab[n, k]
```

Algoritm 3: Variant 2 - bottom-up, vanlig dynamisk programmering
Input: $n, k \in \mathbf{Z}$ där $n \geq k \geq 0$
Output: $\binom{n}{k}$

```
BIN(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]+tab[i − 1, j − 1];
(6)   return tab[n, k];
```

Vilken version som lämpar sig bäst varierar från fall till fall. Memoisering är bättre om man inte tror att man kommer att fylla hela tabellen, eftersom den bara räknar ut de värden som kommer att behövas. "Bottom-up" fyller alltid hela tabellen, men gör det effektivare än memoisering så om hela eller nästa hela tabellen behövs så är "bottom-up" ofta snabbare. Vilken version som är lättast att koda varierar också från fall till fall så det får man ta med i beräkningen när man bestämmer sig för den ena algoritmen framför den andra.

Ett till exempel är *maximum sum* som går ut på att hitta den största summan av bredvidliggande tal i en array. Vi har $x_1, x_2, ..., x_n \in \mathbf{Z}$. Vi vill hitta

$$\max_{l \leq r} f(l, r), \text{ där } f(l, r) = \sum_{i=l}^{r} x_i$$

Den naiva lösningen att beräkna enligt formeln ovan tar kubisk tid. Men om vi inför en vektor prefix[] enligt

$$\text{prefix}[r] = \sum_{k=1}^{r} x_k$$

så kan vi beräkna varje $f(l, r)$ i konstant tid

$$f(l, r) = \text{prefix}[r] - \text{prefix}[l - 1]$$

och bästa summan kan beräknas i totalt kvadratisk tid.

Vi kan även lösa problemet i linjär tid genom att för varje r beräkna den största summan som slutar i r. Detta kan göras i linjär tid genom att beräkna

$$\begin{aligned} m(1) &= x_1 \\ m(r+1) &= x_{r+1} + \max(0, m(r)) \end{aligned}$$

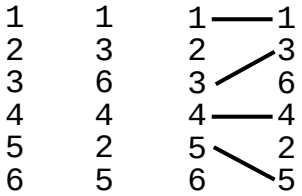
Bästa summan är då $\max_{1 \leq r \leq n} m(r)$.

2.4 Längsta växande delsekvens

Två elektroniska kretsar ligger intill varandra och varje har n stycken kontakter. Dessa är numrerade, på den ena är de i ordning $1, 2, ..., n$ medans på den andra finns samma nummer men inte i ordning. Vi vill förbinda kontakterna med samma

markering. Frågan är hur många förbindelser vi maximalt kan dra utan att de korsar varandra.

Betrakta följande exempel:



Det maximala antalet förbindelser är i detta fall 4 och det bästa sättet att förbinda chipen syns ovan.

Vi betecknar kontakterna på andra chipet som $x[i]$. Efter att ha tittat noga på bilden ovan konstaterar vi att det vi söker är egentligen längden av den längsta växande delsekvensen av $x[1...n]$.

Det kan vara lockande att försöka lösa detta problem med en girig algoritm. Vi skulle kunna gå igenom $x[]$ och i varje steg lägga till $x[i]$ till delföljden om det är större än det största hittills tillagda elementet. Med denna metod får vi följande:

	start	1	3	6	4	2	5
längden av delföljden	0	1	2	3	3	3	3
största tillagda element	$-\infty$	1	3	6	6	6	6

Detta ger som synes inte optimalt svar (maximala längden blir 3 istället för 4). Ett bättre tillvägagångssätt är att använda dynamisk programmering.

Vi skapar en vektor last[] där last[i] är det lägsta värde som kan avsluta en växande sekvens av längd i. I början sätter vi last[0] $= -\infty$ och last[i] $= \infty$ för $i = 1, ..., n$. Vi går igenom $x[]$ och för varje i söker vi j så att last[$j-1$] $< x[i] \leq$ last[j] och sätter last[j] $= x[i]$. För varje $x[i]$ kommer det att finnas en plats j eftersom vi initialiserat vektorn som vi gjorde. Så här ser det ut för vårt exempel:

	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]	∞	∞	∞	∞	∞	∞	∞

Svaret är då största talet i last[] som inte är ∞. Att hitta rätt plats i last[] för $x[i]$ kan göras med binärsökning, eftersom last[] hela tiden är sorterad i stigande ordning. Antalet element i $x[]$ är n, och tidskomplexiteten för hela algoritmen blir då $O(n \log n)$.