Programkonstruktion och Datastrukturer

VT 2012

Tidskomplexitet

Elias Castegren
elias.castegren.7381@student.uu.se

Problem och algoritmer

- Ett problem är en uppgift som ska lösas.
  - Beräkna n! givet n>0
  - Räkna antalet vokaler i en textsträng
  - Lös ett kryptokorsord, givet en ordlista med möjliga ord
- En algoritm är en steg-för-steg-beskrivning för hur man löser ett problem.

  1. Sätt p till 1
  2. Är n = 0? Returnera p, annars gå till steg 3
  3. Sätt p := p•n och n := n-1. Gå till steg 2

Tidskomplexitet

- Tidskomplexitet är ett mått på hur körtiden för en algoritm förändras när storleken på problemet ändras.
- Teoretiskt mått på hur komplicerat ett problem är beräkningsmässigt. Inte låst till programmering.
- Säger ingenting om hur lång tid det tar att köra algoritmen!

Exempelproblem

- Utgå från en lista av godtyckliga heltal och beräkna samma lista med alla dubletter borttagna.
  - [1,1,2,2,2,3] ==> [1,2,3]
  - [3,1,4,1,5,9,2,6,5,3,5,8,9] ==> [3,1,4,5,9,2,6,8]
- Algoritm 1

  1. Finns det element i listan? Om nej är man klar.
  2. Ta bort varje förekomst av det första elementet utom den första.
  3. Börja om från steg 1 med alla element förutom det första.

Exempelproblem

- Algoritm 1, analys
  - Om det finns n element i listan måste vi i värsta fall jämföra det första elementet med (n-1) element, det andra med (n-2) osv.
  - Körtiden T för n element blir blir:

$$T(n) = (n-1)k_1 + (n-2)k_1 + \cdots + 2k_1 + 1k_1 = (n-1)\frac{n}{2}k_1$$

där k_1 är tiden för att jämföra och eventuellt ta bort ett element.

Exempelproblem

- Algoritm 2
  1. Skapa en tom hash-tabell h.
  2. Finns det element i listan? Om nej är man klar.
  3. Finns det första elementet i h? Om nej, lägg in det i h. Om ja, ta bort elementet ur listan.
  4. Börja om från steg 2 med resten av listan.

Exempelproblem

- Algoritm 2, analys
  - För varje element gör vi en sökning och eventuellt en insättning i en hashtabell. Båda operationerna tar konstant tid.
  - Körtiden T för n element blir:

    $$T(n) = n \cdot k_2$$

    där k_2 är tiden för att söka efter och eventuellt sätta in ett element i en hashtabell, samt eventuellt ta bort ett element ur listan.

Två exempel på komplexitet

$$T_1(2n) = (2n-1)\frac{2n}{2}k_1 = 2n^2 + n \approx 4T_1(n)$$

$$T_2(2n) = 2n \cdot k_2 = 2T_2(n)$$

För en dubbelt så lång lista tar det ungefär fyra gånger så lång tid med algoritm 1, men dubbelt så lång tid med algoritm 2.

- Algoritm 1 har alltså högre komplexitet än algoritm 2.
- Hur ska man uttrycka den här skillnaden?

  Matematiken ger oss Theta-notationen!

Theta-notationen

- Uttrycker begränsningar för tillväxten hos matematiska funktioner.

$$f(x) = \Theta(g(x)) \implies \text{Det f nns } k_1, k_2, x_0 \text{ så att}$$

$$0 < k_1 g(x) \leq f(x) \leq k_2 g(x)$$

$$\text{för } x \geq x_0$$

- Informellt: För tillräckligt stora x växer $f(x)$ ungefär lika snabbt som $g(x)$
- ”För tillräckligt stora x”: Asymptotisk analys

Theta-notationen

- ”För tillräckligt stora x”: Asymptotisk analys
  - Bara den snabbast växande termen i $f(x)$ är intressant, de andra kan strykas.
  - Om $f(x)$ är ett polynom är man bara intresserad av termen med högst grad
- Koeff cienter påverkar bara k_1 och k_2 (se föregående slide) och kan också strykas:

$$f(x) = \cancel{3}x^3 + \cancel{20}x^2 + \cancel{713}x + \cancel{100} = \Theta(x^3)$$

Theta-notationen

- Variationer: Omega och Omicron (”Big Oh”)

$$f(x) = \Theta(g(x)) \implies 0 < k_1 g(x) \leq f(x) \leq k_2 g(x)$$

$$f(x) = \Omega(g(x)) \implies 0 < kg(x) \leq f(x)$$

$$f(x) = O(g(x)) \implies 0 < f(x) \leq kg(x)$$

- Informellt: För tillräckligt stora x begränsas $f(x)$ underifrån, respektive ovanifrån av $g(x)$

Tillbaka till dublettborttagningen

- Algoritm 1

$$T(n) = (n-1)\frac{n}{2} \cdot k_1 = \Theta(n^2)$$

  - Algoritmen har *kvadratisk* komplexitet

- Algoritm 2

$$T(n) = n \cdot k_2 = \Theta(n)$$

  - Algoritmen har *linjär* komplexitet

Olika komplexiteter

För problem av storlek n.

k är en konstant.

Komplexitetsfunktion	**Tillväxthastighet**	
1	Konstant	Sub-linjär
log n	Logaritmisk	
$\log^2 n$	”log-squared”	
n	Linjär	Polynomisk
n log n		
n^2	Kvadratisk	
n^3	Kubisk	
k^n	Exponentiell	Exponentiell
n!		Superexponentiell
n^n		

Källa: Pierre Flener

Från kod till komplexitet

- Antag att vi har en rekursiv funktion vars tidskomplexitet vi vill analysera.
- Avgör vilket/vilka argument som påverkar körtiden T. Storleken på detta argument är problemstorleken n.
  - Värdet av ett heltal
  - Längden på en sträng/lista
  - Höjden av ett träd
  - ...

Från kod till komplexitet

- Avgör hur körtiden för varje rekursivt steg beror av problemstorleken n.
  - En lista av längd n traverseras
  - Vägen till ett löv i ett träd hittas
  - Problemstorleken spelar ingen roll (konstant körtid)
  - ...
- Avgör antalet rekursiva anrop och problemstorleken för dessa
  - Ett anrop med en sträng ett tecken kortare (T(n-1))
  - Två anrop med hälften av alla element i listan (2T(n/2))
  - ...

Från kod till komplexitet

- Ställ upp en rekursion som beskriver körtiden för funktionen givet problemstorleken n.
  - Har ofta formen

$$T(n) = \begin{cases} \text{Körtiden för basfallet, } n=0 \\ \text{Körtiden för det allmänna fallet plus de rekursiva anropen, } n>0 \end{cases}$$

- Hitta den slutna formen för rekursionen T och bevisa att den stämmer.
  - Gissa/Prova/Rita/Slå upp formler (Alla medel tillåtna!)
  - Master Theorem
  - ...

Från kod till komplexitet

- Betrakta nedanstående funktion som räknar antalet förekomster av `x` i en lista `l`:

```
fun countOccurences(_, []) = 0
  | countOccurences(x, (f::r)) =
      if x = f then
          1 + countOccurences(x, r)
      else
          countOccurences(x, r);
```

Från kod till komplexitet

- Hitta problemstorleken
  - En längre lista tar längre tid att genomsöka
  - Värdet på `x` spelar ingen roll för körtiden
  - n = längden på listan `l`

```
fun countOccurences(_, []) = 0
  | countOccurences(x, (f::r)) =
      if x = f then
          1 + countOccurences(x, r)
      else
          countOccurences(x, r);
```

Från kod till komplexitet

- Hur påverkar n körtiden i varje steg?
  - Funktionen genomför en heltalsjämförelse och eventuellt en addition.
  - Den nuvarande längden på listan spelar ingen roll
  - Körtiden i varje rekursivt steg är alltså konstant

```
fun countOccurences(_, []) = 0
  | countOccurences(x, (f::r)) =
      if x = f then
          1 + countOccurences(x, r)
      else
          countOccurences(x, r);
```

Från kod till komplexitet

- Undersök de rekursiva anropen
  - Det allmänna fallet gör exakt ett rekursivt anrop
  - I varje rekursivt anrop är listan ett element kortare
  - Varje anrop där n>0 ger alltså upphov till exakt ett anrop med körtiden T(n-1)

```
fun countOccurences(_, []) = 0
  | countOccurences(x, (f::r)) =
      if x = f then
          1 + countOccurences(x, r)
      else
          countOccurences(x, r);
```

Från kod till komplexitet

- Ställ upp en rekursion och finn en sluten formel

$$T(n) = \begin{cases} k_0 & , n = 0 \\ T(n-1) + k_1 & , n > 0 \end{cases}$$

$$T(n) = nk_1 + k_0 = \Theta(n)$$

```
fun countOccurences(_, []) = 0
  | countOccurences(x, (f::r)) =
      if x = f then
          1 + countOccurences(x, r)
      else
          countOccurences(x, r);
```

Från kod till komplexitet

- Ett till exempel (funktionens syfte är oviktigt)

```
fun gizmo [] = 0
  | gizmo (f::r) =
      let
        val v = countOccurences(f,r)
      in
        v + gizmo r
      end;
```

Från kod till komplexitet

- Problemstorleken n är listans längd
- I varje steg anropas `countOccurences` (som vi vet har linjär komplexitet) med en lista av längd n-1. Arbetsmängden i varje steg är alltså $\Theta(n-1) = \Theta(n)$.
- Ett rekursivt anrop T(n-1)

```
fun gizmo [] = 0
  | gizmo (f::r) =
      let
        val v = countOccurences(f,r)
      in
        v + gizmo r
      end;
```

Från kod till komplexitet

- Rekursion och sluten formel

$$T(n) = \begin{cases} O(1) & , n = 0 \\ T(n-1) + \Theta(n), & n > 0 \end{cases}$$

$$T(n) = n \bullet \Theta(n) + O(1) = \Theta(n^2)$$

```
fun gizmo [] = 0
  | gizmo (f::r) =
      let
        val v = countOccurences(f,r)
      in
        v + gizmo r
      end;
```

Från kod till komplexitet

- Avgör vilket/vilka argument som utgör problemstorleken
- Avgör hur körtiden för varje rekursivt steg beror av problemstorleken n.
- Avgör antalet rekursiva anrop och problemstorleken för dessa
- Ställ upp en rekursion T(n)
- Hitta en sluten form för T(n) och bevisa den

Master Theorem

- Löser rekursioner på formen:

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

- Den intressanta termen är $n^{\log_b a}$
- Tre fall:
  - $\frac{n^{\log_b a}}{f(n)} = \Omega(n^\epsilon) \implies T(n) = n^{\log_b a}, \quad \epsilon > 0$
  - $n^{\log_b a} = \Theta(f(n)) \implies T(n) = n^{\log_b a} \log n$
  - $\frac{f(n)}{n^{\log_b a}} = \Omega(n^\epsilon) \implies T(n) = f(n), \quad \epsilon > 0$
    (+ regularity condition)

Master Theorem

$$\frac{n^{\log_b a}}{f(n)} = \Omega(n^\epsilon) \implies T(n) = n^{\log_b a}, \quad \epsilon > 0$$

- Vad betyder allt det här då?

  Jo:
  ”Om $\frac{n^{\log_b a}}{f(n)}$ alltid är större än n^ϵ för något ϵ och tillräckligt stora n, så är den slutna formen för rekursionen $T(n) = n^{\log_b a}$ ”

  - Vi säger att $n^{\log_b a}$ *dominerar* $f(n)$.
- Tänk på att kvoten mellan $n^{\log_b a}$ och $f(n)$ måste vara polynomial (alltså i $\Omega(n^\epsilon)$, för något ϵ), annars är inte MT applicerbar!

Master Theorem

- Löser rekursioner på formen:

$$T(n) = aT(\tfrac{n}{b}) + f(n)$$

- Den intressanta termen är $n^{\log_b a}$
- Tre fall:
  - $n^{\log_b a}$ dominerar $f(n)$: $T(n) = n^{\log_b a}$
  - $n^{\log_b a}$ växer som $f(n)$: $T(n) = n^{\log_b a} \log n$
  - $f(n)$ dominerar $n^{\log_b a}$: $T(n) = f(n)$
    (+ regularity condition)

Övningar

- Hitta tidskomplexiteten för följande funktioner:

```
fun foo([]) = 0
  | foo(f::r) = 1 + foo(r);

fun bar([]) = 2
  | bar(f::r) = foo(r) + bar(r);

fun baz(0) = 42
  | baz(n) = baz(n div 2) + baz(n div 2);
```