Datastrukturer och algoritmer

Föreläsning 5

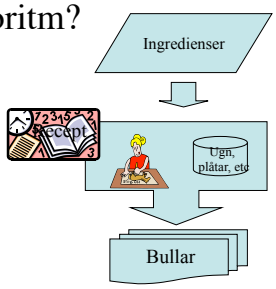
Algoritmer & Analys av Algoritmer

Innehåll

- Algoritmer
  - Vad är det?
  - Mer formellt om algoritmer
    - beräkningsbarhet
  - Att beskriva algoritmer
  - Analysera algoritmer
    - Exekveringstid, minnesåtgång….

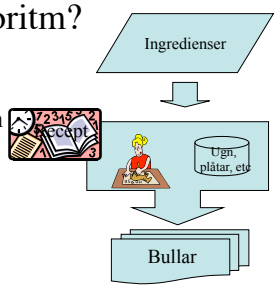
Algoritm?

- Recept som man följer för att lösa ett givet problem på ett strukturerat sätt
- Ändlig stegvis beskrivning av en ändlig process



Algoritm?

- Texten som beskriver algoritmen är fix stor
- Processen kan variera i storlek
- Kornighet i en algoritm



Exempel

- Antag att vi har en lista på alla anställda på ett företag: Namn, Pnr, lön, etc. Vi vill räkna ut lönekostanden för företaget.

```
1. Skriv ner talet 0
2. Gå igenom listan, och för varje anställd
   så adderar man du personens lön till det
   skrivna talet
3. När man nått slutet på
   listan så är det ned
   skrivna talet resultatet
```


Algoritmer mer formellt

Definition:

> Algoritm är en noggrann plan,
> en metod för att stegvis utföra något

Krav på algoritmer


23. Donald Knuth

- Ändlighet
  - Algoritmen måste sluta
- Bestämdhet
  - Varje steg måste vara entydigt
- Indata
  - Måste ha noll eller flera indata
- Utdata
  - Måste ha ett eller flera utdata
- Effektivitet/Genomförbarhet
  - Varje steg i algoritmen måste gå att utföra på ändlig tid

Att (be)skriva algoritmer

- Vi behöver ett språk som:
  - Är strukturerat och formellt
  - Mindre formellt än programmeringsspråk
    - Ingen typning
    - Dynamisk bindning
    - Räckvidd

Pseudokod

- Mix av naturligt språk och programmeringsspråk
- Influenser från matematisk notation
  - ← används för tilldelning
  - = används för likhetsrelationen
- Funktionsdeklaration
  - **`Algorithm`** `name(param1,param2)`

Olika sätt att beskriva en algoritm

- **Naturligt språk** - man förklarar problemlösningen med vanlig text med införda variabel- och funktionsnamn.
- **Blockdiagram** - man visar en grov struktur av problemlösningen i form av ritade "boxar". Varje box kan vara ett delproblem.
- **Flödesschema/flödesdiagram,** strukturdiagram - man ritar algoritmen med olika symboler, som visar när och hur saker skall ske i programmet. Finare indelning än blockdiagram, t.ex. kan varje block beskrivas som flödesschema för ett delproblem.
- **Pseudokod** - man skriver en blandning av programmeringsspråk och vanlig text, dvs man har variabler, funktioner, kontrollstrukturer etc

Pseudokod – programkonstruktioner

- Besluts strukturer:
  `if`…**`then`**…**`[else`**…**`]`**
- Villkorsloopar:
  `while`…**`do`**
  `repeat` … **`until`**…
- Räkneloopar:
  `for`…**`do`**
- Arrayindexering:
  `A[`*`i`***`]`**
- Anrop:
  `method(`*`args`***`)`**
  `object metod(`*`args`***`)`**
- Returnera värden:
  `return` *`value`*

Pseudokod – exempel

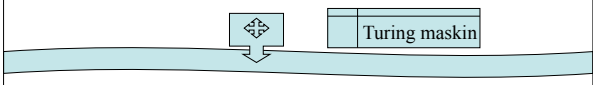
```
Algorithm arrayMax(A,n)
   input: An array A storing n integers
   output: The maximum element in A
currentMax ← A[0]
for i ← 1 to n-1 do
   if currentMax < A[i] then
      currentMax ← A[i]
return currentMax
```

Pseudokod

- Vi använder oss av pseudokod för att beskriva algoritmer
- Det finns inget universellt språk utan många dialekter
- Alla döljer mycket av programspråkens designval, dvs. pseodokoden är programspråksoberoende

Algoritmer mer formellt

- Algoritmiska problem & beräkningsbarhet
  - En klass av problem
  - Beräkningsbar omm det finns en Turingmaskin som löser problemet



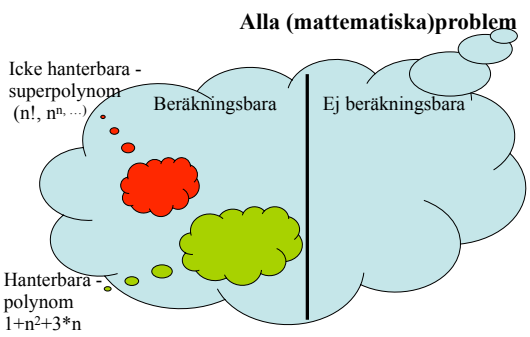
Analys av algoritmer

- Vad kan analyseras?
  - Exekveringstid
  - Minnesåtgång
  - Implementationskomplexitet
  - Förståelighet
  - Korrekthet
  -
  -

Varför analysera algoritmer?

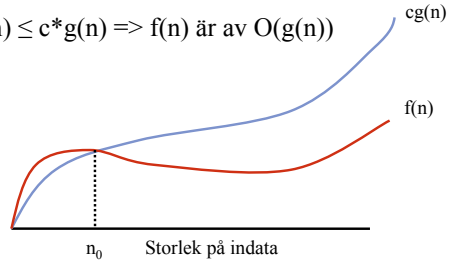
- Exekveringstid/minnesåtgång
  - Är algoritmen praktiskt körbar
  - Vi vill ha den snabbaste!
    - Att implementera
    - Att köra

Beräkningsbar/hanterbar



Stora Ordo

$f(n) \leq c*g(n) \Rightarrow f(n)$ är av $O(g(n))$



Litet räkneexempel

- 1 operation tar 1μs
- $1*10^9$ element i en lista
- Kvadratisk sorteringsalgoritm n^2
  - 31000år
- Logaritmisk sorteringsalgoritm n*log(n)
  - 30000s ≈ 1 arbetsdag
- n^2 och dubbelt så snabb => 15500 år
- n^2 och1000 gånger så snabb => 31år

Exekveringstider - en dator med 1 MIPS, $1*10^6$ op/sek

	10	20	50	100	300
N^2	1/10000	1/2500	1/400	1/100	9/100
N^5	1/10	3.2 sek.	5.2 min.	2.8 tim.	28.1 dag.
2^N	1/1000	1 s	35.7 år	40000 billioner år	7500 siffror år
N^N	2.8 tim.	3.3 billioner år	7000 siffror år	18500 siffror år	76800 siffror år

✓ Drygt 10 miljarder μs på en dag
✓ $1*10^{24}$ μs sedan ”Big Bang”

Ohanterbarhet

- Många triviala att förstå och viktiga att lösa
  - Schemaläggning
  - Handelsresande
- Moore’s lag förändrar den situationen?
- Hur hanterar vi ohanterbarhet?

Hantera ohanterbarhet

- Heuristik
  - Lösa nästan rätt problem
    - Förenkling
  - Lösa problemet nästan rätt
    - Approximation

NP-kompletta problem

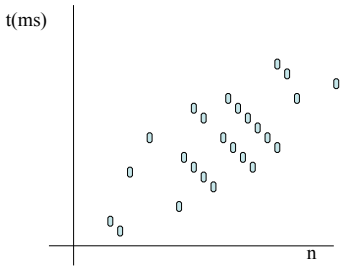
- En speciell klass av ohanterliga problem
- Har problem X en lösning med egenskaperna Y
- Ekvivalenta:
  - Transformeras
  - Högst exponentiella
  - Saknar bevis för ohanterbarhet

Icke hanterbara - superpolynom ($n!$, n^n, …)

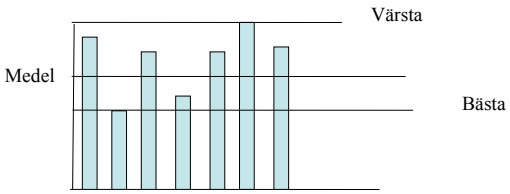
Mäta tidsåtgången

- Hur ska vi mäta tidsåtgången?
  - Experimentell analys
    - Implementera algoritmen
    - Kör programmet med varierande datamängd
      - Storlek
      - Sammansättning
    - Använd metoder för tidtagning så som
      - `System.currentTimeMillis()`
    - Plotta uppmätt data

Exempel



Bästa, värsta & medel



Experimentell analys…

- Begränsningar med metoden
  - Måste implementera och testa algoritmen
  - Svårt att veta om programmet har stannat eller fast i beräkningarna.
    T ex. 2^n; n=100 => 40000 billioner år
  - Experimenten kan endast utföras på en begränsad mängd av data, man kan missa viktiga testdata
  - Hårdvaran och mjukvaran måste vara den samma för alla implementationer

Generellare metod behövs

- Som använder en högnivåbeskrivning av algoritmerna istället för en implementation av den
- Tar hänsyn till alla möjliga indata
- Analys oberoende av hårdvaran och mjukvaran
- Asymptotisk analys

Datastrukturer och algoritmer

Föreläsning 6

Asymptotisk analys

Innehåll

- Asymptotisk analys
  - Lite matte
  - Analysera pseudokoden
  - O-notation
    - Strikt
    - ”Okulärbesiktning”

Asymptotisk analys

- Utgår från pseudokoden
- Räkna operationer
  - Ställ upp ett uttryck för antalet operationer beroende av problemstorleken
- Förenkla tidsuttrycket
- Ta fram en funktion som begränsar tidsuttrycket ovanifrån…

Analys av algoritmer

- Primitiva operationer
  - Lågnivå beräkningar som är i stort sett oberoende av programspråk och kan definieras i termer av pseudokod:
    - Anropa en metod/funktion
    - Returnera från en metod/funktion
    - Utföra en aritmetisk operation (+, -, …)
    - Jämföra två tal, etc.
    - Referera till en/ett variabel/objekt
    - Indexera i en array

Mer analys av algoritmer…

- Inspektera pseudokoden och räkna antalet primitiva operationer.
- Väldig abstraktion, vi bortser från hårdvaran, och att olika operationer tar olika lång tid, …
- Alternativet är att titta på de verkliga tiderna för de olika operationerna
  - Ger en maskinberoende analys

Exempel

```
Algorithm arrayMax(A,n)
   input: An array A storing n integers
   output: The maximum element in A
currentMax ← A[0]                          //1+1
for i ← 1 to n-1 do                        //1+n(1+1)+(n-1)*([]+1)
   if currentMax < A[i] then               //1+1+1
      currentMax ← A[i]                    //1+1
return currentMax                          //1
```

$T_{max}(n) = 3+2n+(n-1)*6+1 = 8n-2$

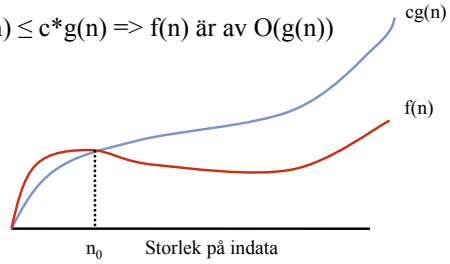
$T_{min}(n) = 3+2n+(n-1)*4+1 = 6n$

Jämföra T(n)

- Rita kurvor för T(n) och jämför…svårt…
- Alternativet är asymptotisk notation/analys
  - Förenkla jämförelsen genom att avrunda T(n)
    - $1.000001 \approx 1$
    - $3n^2 \approx n^2$

Stora Ordo

$f(n) \leq c*g(n) \Rightarrow f(n)$ är av $O(g(n))$



O(definition)

Definition:

Givet funktionerna f(n) och g(n) säger vi att f(n) är O(g(n)) omm $f(n) \leq c*g(n)$ för $n \geq n_0$ och $c > 0$ och $n_0 \geq 1$

f(n) **är** O(g(n))

- Varför inte $f(n) \leq O(g(n))$ eller $f(n) \approx O(g(n))$
- Borde vara
  - f(n) **tillhör** O(g(n)) ty O(g(n)) är en mängd funktioner,
    - Se bilden.

  - Men vi skriver
    f(n) **är** O(g(n))

Mer ordo

- f(n) = 7n -3 hitta en funktion som begränsar f(n)?
  - Oändligt många, hitta den ”minsta”
  - Droppa allt utom den ledande termen dvs. lägre ordningens termer och konstanter
    - 7n-3 är O(n)
    - $8n^2\log(n) + 5n^2+n$ är $O(n^2\log(n))$
- Konstanterna c och n_0?
  - Vi återkommer

Specialla klasser av algoritmer

- Logaritmiska O(log(n))
- Linjära O(n)
- Kvadradiska $O(n^2)$
- Polynoma $O(n^k)$; $k \geq 1$
- Exponentiella $O(a^n)$; $n \geq 1$
- $\log(n) << n << n^2 << n^3 << 2^n$

Exempel

- F(n) = 7n-3 är O(n)
  - Hitta c

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} + 1$$

$$\lim_{n\to\infty}\left(\frac{7n-3}{n}\right)+1 = \lim_{n\to\infty}\left(7-\frac{3}{n}\right)+1 \Rightarrow c = 8$$

- n_0?
  - 7n-3≤8n; $n_0 = 1$?
  - Ok!
- Alternativ
  - Addera de positiva termerna

O(sammanfattning)

- *O(n)* används för att utrycka antalet primitiva operationer som utförs som en funktion av storleken på indata
- En övre gräns för tillväxt
- **arrayMax** är en linjär algoritm dvs *O(n)*
- En algoritm som körs på *O(n)* är bättre än en $O(n^2)$, men *O(log(n))* är bättre än *O(n)*
  - $\log(n) << n << n^2 << n^3 << 2^n$

O(varning)

- Var aktsam, stora konstanter ställer till det
  - $T(n)$=1000000n är en linjär algoritm O(n), men i många fall mycket mindre effektiv på data mängder än en algoritm med
    $T(n) = 2n^2$ som är $O(n^2)$
  - O-notationen är en stor förenkling, dvs en övre gräns, det finns släktingar som begränsar nedåt
  - Vi har också tagit bort kopplingen till hårdvaran.

O(genväg)

- Man kan många gånger skippa vägen över T(n)
  - Väldigt grov uppskattning av tillväxten
  - Man gör en okulärbesiktning av algoritmen
    - Initiera en array **är** O(n)
    - Nästlade loopar
      är O(n)*O(n)*…*O(n)≈O(n^k)

O(exempel 1)

```
Algorithm prefixAv1(X);
   Input: An n-element Array of numbers
   Output: An n-element Array of numbers such that A[i]
   is the average of X[0],…,X[i].

   Let A be an array of numbers
   for i ← 0 to n-1 do
       a ← 0
       for j ← 0 to i do
             a ← a+X[j]
       A[i] ←a/(i+1)
   return A
```

Analys: $T_b(n) = ?$ $T_w(n) = ?$
Men algoritmen är av $O(n^2)$

O(exempel 2)

```
Algorithm prefixAv1(X);
   Input: An n-element Array of numbers
   Output: An n-element Array of numbers such that A[i] is the
   average of X[0],…,X[i].

   Let A be an array of numbers
   s ← 0
   for i ← 0 to n-1 do
       s ← s+X[i]
       A[i] ←s/(i+1)
   return A
```

Analys: T(n) = ?
Men algoritmen är av O(n)

Fler definitioner

- $T(n)=\Omega(g(n))$ omm det finns positiva konstanter c och n_0 så att $T(n)\geq cg(n)$ när $n\geq n_0$
- $T(n)=\Theta(h(n))$ omm $T(n)=O(h(n))$ och $T(n)=\Omega(h(n))$
- $T(n)=o(p(n))$ omm $T(n) = O(p(n))$ och $T(n)\neq\Theta(p(n))$

Lite matematik behövs…

- Logaritmer
  - $\log_b(xy) = \log_b(x) + \log_b(y)$
  - $\log_b(x/y) = \log_b(x) - \log_b(y)$
  - $\log_b(x^\alpha) = \alpha \log_b(x)$
  - $\log_b(a) = \log_x(a)/\log_x(b)$

Kanske lite mer…

- Exponenter

$$a^{(b+c)} = a^b a^c \qquad a^{bc} = (a^b)^c$$

$$a^b / a^c = a^{(b-c)}$$

$$b = a^{\log_a(b)} \qquad b^c = a^{c*\log_a(b)}$$

Och ännu mer matte…

- Summor är bra att kunna…
  - Generell definition

$$\sum_{i=s}^{t} f(i) = f(s) + f(s+1) + f(s+2) + \ldots f(t)$$

  - Geometrisk utveckling (n≥0 och 0<a ≠ 1)

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + a^3 + \ldots + a^n = \frac{1 - a^{n+1}}{1 - a}$$

    - Växer exponentiellt (om a>1)

Sista matten för denna gång…

- Aritmetisk tillväxt, summera alla tal från 1 t o m 100

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \frac{n^2 + n}{2}$$