TDDI16 Datastrukturer och algoritmer

Algoritmanalys

LINKÖPINGS UNIVERSITET

Översikt

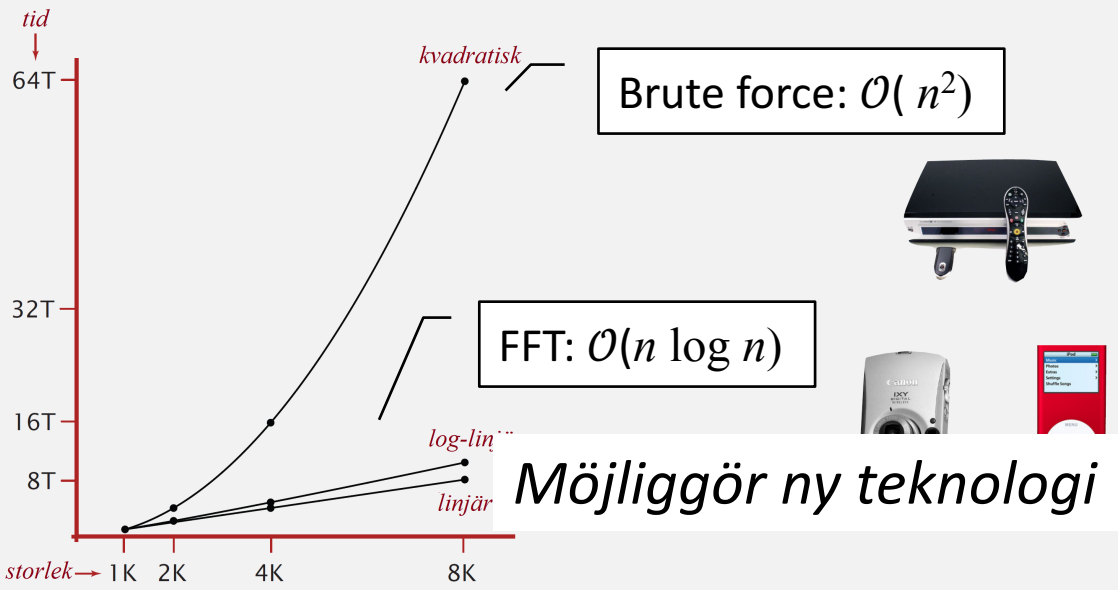
- Skäl för att analysera
- Olika fall att tänka på (bästa, värsta, medelfall)
- Metoder för analys

Skäl till att analysera algoritmer

- Ge garantier
- Förstå teoretisk grund
- Jämföra algoritmer
- Förutsäga prestanda

Varför är prestanda viktigt?

- Diskret Fourier-transform
- Bryt ned vågform bestående av n samplingar i periodiska komponenter



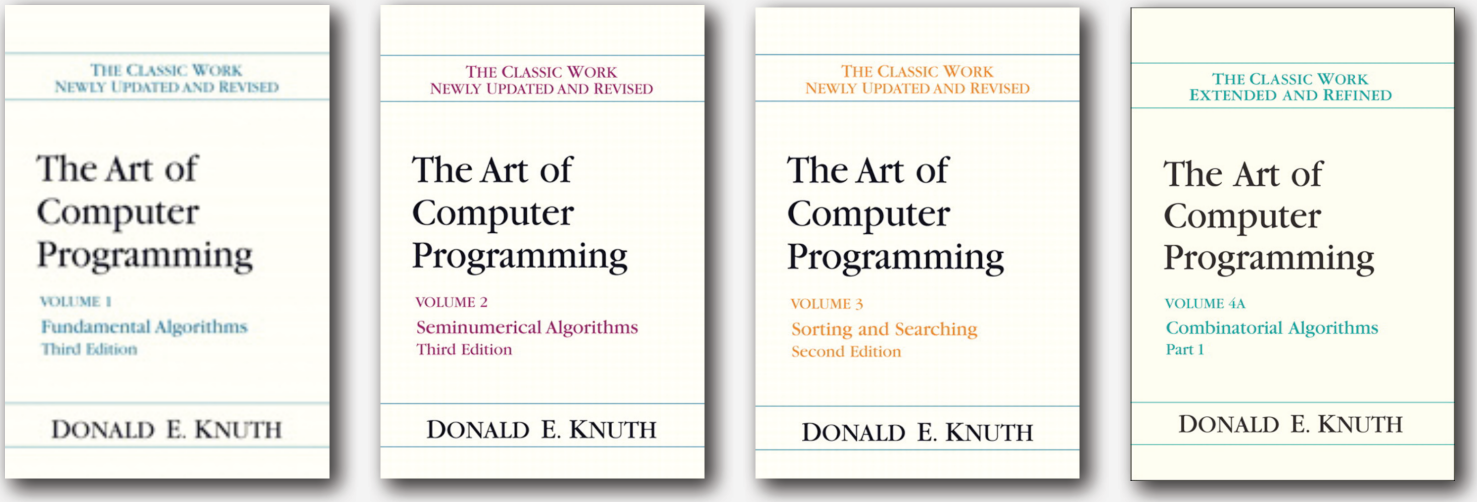
Utmaningen


Varför är mitt program så långsamt?
Varför får det slut på minne?

Matematiska modeller av exekveringstid

- Behöver analysera program för att hitta uppsättning operationer
- Kostnad beror på maskin, kompilator, etc
- Frekvens beror på algoritm, indata

Finns ofta matematiska modeller



Donald Knuth
1974 Turing Award

Matematiska modeller av exekveringstid

kostnad (beror på maskin, kompilator)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

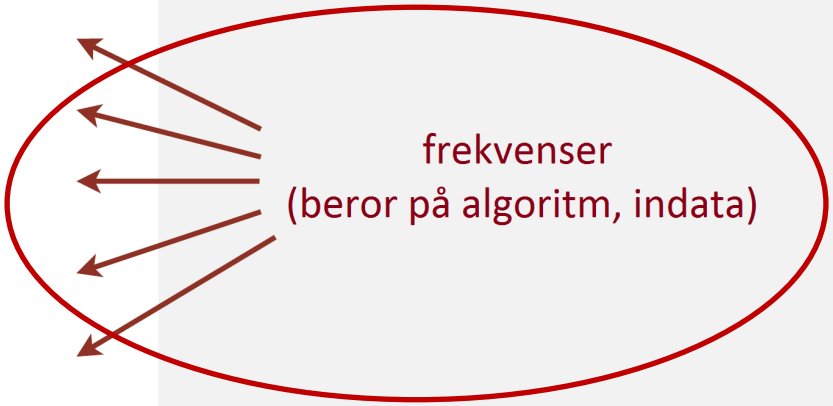
$A =$ array access

$B =$ integer add

$C =$ integer compare

$D =$ increment

$E =$ variable assignment



Exempel

```
for i = 0; i < A.size(); i++
    sum = sum + A[i];
end
```

- För A med n element, kommer loopen att köras n varv: $T(n) = n,\ T(n) \in \mathcal{O}(n)$
- Vi bortser ifrån (konstant) kostnad för addition, variabeltilldelning, uppräkning av i, etc.

Fall att analysera

- Bästa fall
  - T.ex. alla if-satser “faller igenom” med ett nej → minst arbete igenom algoritmen
  - T.ex. indata råkar redan vara sorterad
- Värsta fall
- Genomsnittsfall
- Amorterat

Metoder

- Algebraiskt
  - Räkna varv (iterationer)
- Lös rekurrensrelationer
  - För rekursiva algoritmer
- Probabilistisk analys
  - Lista ut beteendet i förväntade fallet

Algebraisk analysmetod

Exempel: Värstafallsanalys

- Beskriv resursförbrukningen (tid/minne) som en icke avtagande funktion av indatastorlek, n
  - Ignorera konstanta faktorer
- Resulterar i $T(n)$, $M(n)$
- Studera *asymptotiskt beteende* för stora indata
  - Ignorera konstanter och lägre ordningens termer
- Kan då bestämma vilken $\mathcal{O}$–kategori $T(n)$ tillhör

Värstafallsanalys

- Räkna operationer som upprepas i algoritmen
  - aritmetisk operation
  - jämföra två tal
  - följa en objektreferens
  - anropa en funktion/metod/procedur
- n upprepningar $\rightarrow n \cdot$ kostnad

Värstafallsanalys

- Loop (for. . . och while. . .): tiden av villkoret plus kroppen gånger antalet iterationer n

$$t_{while} = n \cdot (t_{cond} + t_{body})$$

Värstafallsanalys

- Villkorssats (if…then…else): tiden för att evaluera villkoret plus den maximala tiden av de två grenarna

$$t_{if} = t_{cond} + \max(t_{then}, t_{else})$$

Exempel: for-loop

```
function FIND(A[1,...,n], t)
    for i ← 1 to n do
        if A[i] == t then
            return true
    return false
```

- Hur ser värstafallsinstansen ut?
- Hur mycket tid går åt i värsta fallet?
- Vad är tidskomplexiteten?

Exempel: Två loopar

```
function FIND(A[1,...,n], B[1,...,n], t)
    for i ← 1 to n do
        if A[i] == t then
            return true
    for i ← 1 to n do
        if B[i] == t then
            return true
    return false
```

- Hur ser värstafallsinstansen ut?
- Hur mycket tid går åt i värsta fallet?
- Vad är tidskomplexiteten?

Exempel: Två nästlade loopar

```
function COMMON(A[1,...,n], B[1,...,n])
    for i ← 1 to n do
        for j←1 to n do
            if A[i] == B[ j] then
                return true
    return false
```

- Hur ser värstafallsinstansen ut?
- Hur mycket tid går åt i värsta fallet?
- Vad är tidskomplexiteten?

Exempel: Binär sökning

- "Jfr med roten, om mindre: sök i vänstra delträd, annars i högra delträd"
- Trivialt att implementera?
  - Första algoritmen publicerad 1946
  - Första buggfria publicerad 1962
  - Java-bugg i Arrays.binarysearch() upptäckt 2006

Java-implementationen med bugg

```
public static int binarySearch(int[] a, int key) {
     int low = 0;
     int high = a.length -

     while (low <= high) {
          int mid = (low + high) / 2;
          int midVal = a[mid];
          if (midVal < key)
               low = mid + 1
          else if (midVal > key)
               high = mid - 1;
          else
               return mid; // key found
     }
     return -(low + 1); // key not found
}
```

Hur väl funkar koden för riktigt stora tal?

Java-implementationen rätt

```
public static int binarySearch(int[] a, int key) {
     int low = 0;
     int high = a.length - 1;

     while (low <= high) {
          int mid = low + ((high - low) / 2);
          int midVal = a[mid];
          if (midVal < key)
               low = mid + 1
          else if (midVal > key)
               high = mid - 1;
          else
               return mid; // key found
     }
     return -(low + 1); // key not found
}
```

Värstafallstid för binärsökning

- Max antal varv = antal ggr som input kan halveras
- $\mathrm{T}(n) \in \mathcal{O}(\log n)$

Bästafallstid för binärsökning

- Min antal varv = direktträff i mitten av input = 1 varv
- $\mathrm{T}(n) \in \mathcal{O}(1)$, dvs. kontsant tid

Medelfallstid för binärsökning

- Medel av max antal varv och minsta tänkbara antal varv, dvs.
- $T(n) = ((\log n) + 1) / 2 \in \mathcal{O}(\log n)$

Lös rekurrens-ekvationer

Rekursiva algoritmer

- Anropar sig själva med en delmängd av input data
- Efterbearbetning av delresultat från retur från de rekursiva anropen ger också kostnad att ta hänsyn till
  - T.ex. kostnad för att sammanfoga två dellistor som har sorterats
- Brukar behöva skriva ut en serie av summor, och lösa detta

Binärsökning – rekursiv form

```
function BINSEARCH(v[a,...,b], x)
    if a < b then
        m←⌊(a+b)/2⌋
        if v[m].key < x then
            return BINSEARCH(v[m+1,...,b], x)
        else return BINSEARCH(v[a,...,m], x)
    if v[a].key = x then return a
    else return 'not found'
```

Svansrekursivt: delsvar vidarebefordras utan efterbearbetning

Basfall: ett element i vektorn

Värstafallsanalys av rekursiv binärsökning

- Test för basfallet sker vid varje varv (if-satsen): konstant kostnad c_0
- Rekursiva fallet: konstant kostnad för uppdelning c

$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + c$ om $n > 1$

$$T(n) = \begin{cases} c_0 & \text{om } n = 1 \\ T\left(\left\lceil \frac{n}{2} \right\rceil\right) + c & \text{om } n > 1 \end{cases}$$

Värstafallsanalys av rekursiv binärsökning

- Vi räknar uppdelningskostnaden c för varje anrop
- Sätt in def av T(n) gång på gång, anta $n \approx 2^m$

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + c = T\left(\left\lceil \frac{n}{4} \right\rceil\right) + 2c = \cdots = T\left(\left\lceil \frac{n}{2^m} \right\rceil\right) + mc$$
$$= T(1) + mc = c_0 + mc = c_0 + (\log n)c$$

- Eftersom analysen är ”exakt” kan vi använda Θ
  $T(n) \in \Theta(\log n)$

Master theorem

- Om $a \geq 1$, $b > 1$, $d > 0$ och $T(n)$ är monotont ökande

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n), \text{där } f(n) \in \Theta\left(n^k\right), n > 1 \\ d \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad n = 1 \end{cases}$$

Efterbearbetning och kombinering av delsvar

- $T(n) = \Theta(n^{\log_b a})$, om $a > b^k$
- $T(n) = \Theta(n^k \log n)$, om $a = b^k$
- $T(n) = \Theta(n^k)$, om $a > b^k$

Probabilistisk analys

Medelfallsanalys

- Vanligt att man antar jämn sannolikhetsfördelning mellan värstafallsbeteende och bästa fall
- Medel blir då “mittemellan”

Amorterat

- Tar hänsyn till att viss kostsam operation utförs sällan (var x:te gång)
- Extra kostnaden kan fördelas över x körningar
- Ger mer realistisk uppskattning vid ojämnt “overhead” i algoritmen

Exempel på amorterad (fördelad) analys

- Problem: Vill ha automatisk utökning av array när den har blivit full
- Algoritm:
  - Sätt in och ta bort som vanligt
  - Ifall array:en blivit full → skapa en dubbelt så stor array och kopiera över gamla innehållet

Exempel på amorterad (fördelad) analys

- Del av algoritmen
  - Kopiera över gammalt till dubbelt så stor array
    - Kostnad för att kopiera: $\mathcal{O}(n)$
    - Behöver bara göras vid fåtal tillfällen
    - Fördela extra kostnaden på de n tidigare operationerna
  - Amorterad kostnad för att skriva/läsa element:

$$\frac{n}{n} + c = (1 + c) \in \mathcal{O}(1)$$

Nästa gång: …..

www.liu.se

LINKÖPINGS UNIVERSITET