F06 - Komplexitetsanalys, problemlösningsstrategier

5DV149 Datastrukturer och algoritmer

Niclas Börlin
niclas.borlin@cs.umu.se

2024-04-09 Tis

Innehåll

Algoritmanalys

Jämföra algoritmers effektivitet

- Hur jämför vi två algoritmer för att avgöra vilken som är effektivast för stora problem?
  - Med avseende på tid
  - Med avseende på minne
- Nyckelfråga:
  - Om ett problem med n element tar tiden x sekunder och y bytes minne...
  - ...hur mycket resurser kräver ett problem med $2n$ element?
- Typfall

Konstant (1)	$2n$	$\Rightarrow$	x
Linjärt (n)	$2n$	$\Rightarrow$	$2x$
Kvadratiskt (n^2)	$2n$	$\Rightarrow$	$2^2x = 4x$
Kubiskt (n^3)	$2n$	$\Rightarrow$	$2^3x = 8x$
Exponentiellt (k^n)	$n+1$	$\Rightarrow$	kx
Logaritmiskt ($\log n$)	$2n$	$\Rightarrow$	$x+1$

Exempel

- 1 operation tar $1\mu s = 10^{-6}$s
- $n = 10^9$ element i en lista
- Två sorteringsalgoritmer:

1	$O(n^2)$	$T(n) = 10^{12}$ s	$\approx$	31000 år
2	$O(n \log n)$	$T(n) \approx 30000$ s	$\approx$	8 timmar

- 1000 ggr så snabb dator: Algoritm 1 $\approx$ 31 år
- Snabbare algoritm viktigare än snabbare dator!

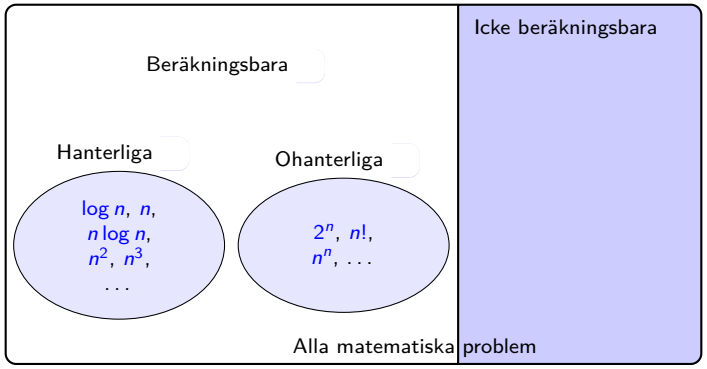
Exekveringstider — 100 000 MIPS, 10^{11} op/s

	n=10	20	50	100	300
n	$1 \cdot 10^{-10}$ s	$2 \cdot 10^{-10}$ s	$5 \cdot 10^{-10}$ s	$1 \cdot 10^{-9}$ s	$3 \cdot 10^{-9}$ s
n^2	$1 \cdot 10^{-9}$ s	$4 \cdot 10^{-9}$ s	$2 \cdot 10^{-8}$ s	$1 \cdot 10^{-7}$ s	$9 \cdot 10^{-7}$ s
n^5	$1 \cdot 10^{-6}$ s	$3 \cdot 10^{-5}$ s	$3 \cdot 10^{-3}$ s	0.1 s	24 s
2^n	$1 \cdot 10^{-8}$ s	$1 \cdot 10^{-5}$ s	3 tim	$4 \cdot 10^{11}$ år	$6 \cdot 10^{71}$ år
n^n	0.1 s	$3 \cdot 10^{7}$ år	$3 \cdot 10^{66}$ år	$3 \cdot 10^{181}$ år	$4 \cdot 10^{724}$ år

- Vad kan vi beräkna?

Beräkningsbara/hanterbara problem

- Icke beräkningsbara problem
- Beräkningsbara, ohanterliga problem — superpolynom
- Beräkningsbara, hanterliga problem — polynom



Icke beräkningsbara problem

- Problem där det inte finns någon algoritm som kan ge ett korrekt svar för varje variant av problemet
  - Ofta kan man finna lösningar för några varianter, men inte för alla
- Exempel: "Stopp-problemet" (*The Halting Problem*)
  - Givet ett program P och indata X till P.
  - Terminerar programmet P när det körs på indatat X?
- Är "Stopp-problemet" beräkningsbart?

Bevis för att stopp-problemet inte är beräkningsbart

- Antag att det finns en algoritm `Stopp(P, X)` som avgör stopp-problemet
- Skapa sedan följande program `M(P)`:
  - `if Stopp(P, P) then`
    - `enter infinite loop`
  - `else`
    - `return`
- Vad händer när `M(M)` körs?
  - Fall 1: `M(M)` terminerar
    - Då måste `Stopp(M, M)` vara falskt för att `return` ska nås — motsägelse!
  - Fall 2: `M(M)` terminerar inte
    - Då är `Stopp(M, M)` sant och programmet kommer aldrig att lämna den oändliga loopen — motsägelse!
- Bägge fallen leder till motsägelse
  - Alltså kan det inte finnas en algoritm `Stopp(P, X)` som avgör stopp-problemet
  - Alltså är stopp-problemet inte beräkningsbart

Beräkningsbarhet

- *Ett problem är beräkningsbart om och endast om det finns en Turing-maskin som löser problemet*
- Vi delar in beräkningsbara problem i två klasser
  - Hanterliga beräkningsbara problem — komplexiteten är begränsad av ett polynom
  - Ohanterliga beräkningsbara problem — komplexiteten är superpolynom, t.ex. 2^n, n^n, $n!$

Ohanterbarhet

- Många beräkningsbara, ohanterliga problem är triviala att förstå och viktiga att lösa:
  - Handelsresande-problemet
  - Schemaläggning

Hantera ohanterbarhet

- Lösa nästan rätt problem, exakt:
  - Förenkling
- Lösa exakt rätt problem, nästan rätt:
  - Approximation
- Exempel: Hitta snabbaste vägen från A till B[1]
  - Förenkling: Sök A–motorväg–B
  - Approximation: Dra "rakt streck" närmaste vägen A–B på kartan. Justera strecket så att det går på vägar.

[1] Detta problem är hanterbart, men är ändå ett bra exempel.

NP-kompletta problem

- En speciell klass av ohanterliga problem:
  1. Det existerar en brute force-algoritm som kan hitta en lösning
  2. Korrektheten hos en lösning kan verifieras "snabbt" (i polynomisk tid)
- Exempel:
  - Givet en mängd M av heltal, finns det en icke-tom delmängd vars summa är noll?
  - Heltalsfaktorisering: Givet ett heltal r som är produkten av två primtal p och q, bestäm p och q
- Ekvivalenta:
  - Transformeras till varandra
  - Saknar bevis för ohanterbarhet
- Ett bevis att NP-kompletta problem är NP eller P (super-polynomiska eller polynomiska) är ett stort olöst problem inom datavetenskap och matematik
  - Ett av sju s.k. *Millennium Prize Problems*

Stora Ordo

Stora Ordo (kap 12.2), definition

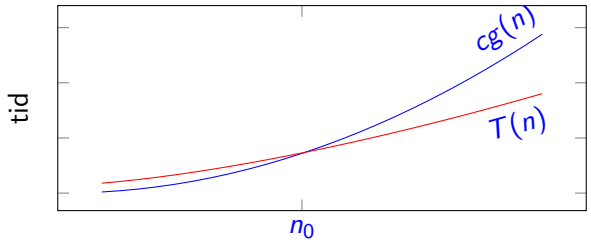
- Vi definierar $T(n)$ att vara $O(g(n))$ ("stort ordo av g av n") *om och endast om* det existerar konstanter $n_0, c > 0$ sådana att[2]

$$|T(n)| \leq cg(n),\ \forall n \geq n_0$$

- Formellt: För $n \geq n_0$ så är $|T(n)|$ uppåt begränsad av $cg(n)$
- Informellt: Över $n = n_0$ så växer $T(n)$ inte snabbare än $cg(n)$

[2] Boken använder K och N i stället för c och n_0.

Stora Ordo (kap 12.2), illustration



$T(n)$ är $O(g(n))$

- Man kan säga

$$T(n) \in O(g(n)),$$

  då $O(g(n))$ är en mängd av funktioner
- Vi kommer att säga/skriva att

$$T(n) \text{ är } O(g(n))$$

  eller

$$T(n) = O(g(n))$$

- Eng: $T(n)$ *is (of)* *order* $g(n)$

Stora Ordo, exempel 1 (1)

- Påstående: $T(n) = 10n + 7$ är $O(n)$, dvs. $g(n) = n$
  - Hitta ett c och n_0 för att avgöra om påståendet är sant
- Man hittar c med hjälp av gränsvärdet

$$c = \lim_{n\to\infty}\left(\frac{T(n)}{g(n)}\right)$$

- I detta fall

$$c = \lim_{n\to\infty}\left(\frac{10n+7}{n}\right) = \lim_{n\to\infty}\left(10 + \frac{7}{n}\right) = 10$$

Stora Ordo, exempel 1 (2)

- För att finna n_0 måste man finna det n där

$$T(n) \le cg(n)$$

  börjar gälla
- Här: Från vilket $n > 0$ gäller att:

$$10n + 7 \le 10n$$

  - Svar: Aldrig!
  - Slutsats: c är för litet
  - Avrunda c uppåt till $c = 11$ och sök n_0! [3]
- Från vilket $n > 0$ gäller att:

$$10n + 7 \le 11n$$

  - Svar: För $n \ge 7$, dvs. $n_0 = 7$
- Slutsats: $\boxed{T(n) \text{ är } O(n) \text{ med } c = 11 \text{ och } n_0 = 7}$

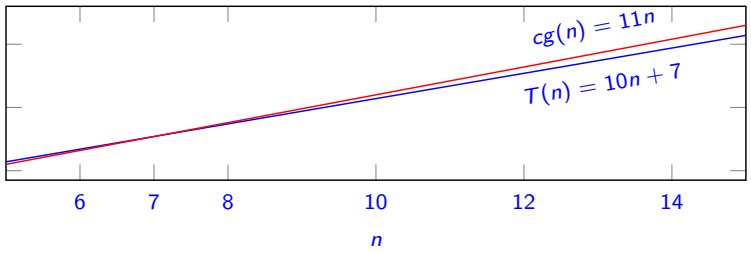
[3] Vi återkommer till hur mycket vi ska avrunda uppåt.

Stora Ordo, exempel 1 (3)

- Genväg:
  - Om den näst mest dominerande termen (här: 7) är positiv, lägg till 1 till gränsvärdet på en gång:

$$c = \lim_{n\to\infty}\left(\frac{10n+7}{n} + 1\right) = \lim_{n\to\infty}\left(10 + \frac{7}{n} + 1\right) = 11$$

Stora Ordo, exempel 1 (4)



Vilket ordo ska man välja?

- Om $T(n)$ är $O(n)$, så är $T(n)$ också $O(n^2)$ och $O(n^2 + n)$, $O(n^3)$, $O(2^n)$, ...
- Underförstått att man väljer så "bra" begränsning som möjligt
  - Viktigast för $g(n)$
  - Vanligen mindre viktigt för c och n_0
- Vilket c ska man välja?
  - Vid teoretisk analys av algoritmer vanligt med heltal
  - Vid experimentell analys: "avrunda rimligt uppåt"
    - $c \approx 2.68 \cdot 10^{-6} \Rightarrow 3 \cdot 10^{-6}$ rimligt
    - $c \approx 2.68 \cdot 10^{-6} \Rightarrow 10 \cdot 10^{-6} = 1 \cdot 10^{-5}$ troligen rimligt
    - $c \approx 2.68 \cdot 10^{-6} \not\Rightarrow 1000 \cdot 10^{-6} = 1 \cdot 10^{-3}$ troligen onödigt

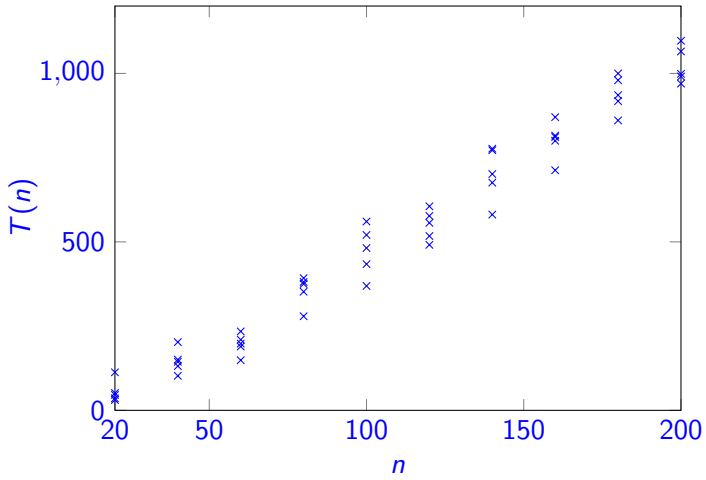
Komplexitetsanalys

- Experimentell
  1. Kör programmet för olika problemstorlekar
  2. Mät tiden
  3. Uppskatta trenden
- Asymptotisk
  1. Analysera algoritmen teoretiskt
  2. Undersök vad som händer då n blir stort

Experimentell analys

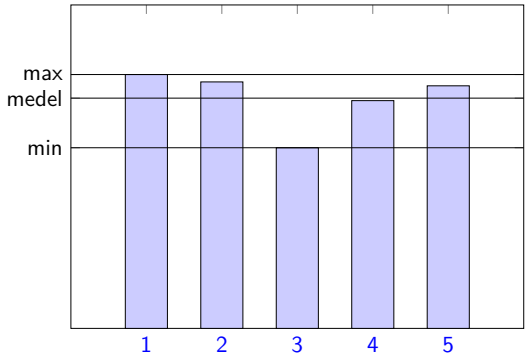
1. Implementera algoritmen
2. Kör programmet med varierande datamängd
   - Storlek
   - Sammansättning
3. Mät tiden $T(n)$ då programmet körs
4. Plotta $T(n)$
   - 4.1 Ansätt en hypotes, t.ex. $g(n) = n^2$
   - 4.2 Plotta $f(n) = T(n)/g(n)$
   - 4.3 Om $f(n)$ går mot positiv konstant så är hypotesen troligen korrekt
   - 4.4 Om inte, ansätt en annan hypotes, t.ex. $g(n) = n$

Exempel på en plot



Bästa, värsta, medel

- $T(n)$ för $n = 80$:

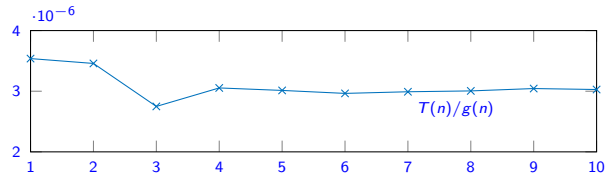


- Beroende på datats sammansättning kan algoritmen fungera olika bra
  - Bubblesort för redan sorterad lista är $O(n)$
    - I medel- och värsta fall $O(n^2)$

Kontrollera din slutsats

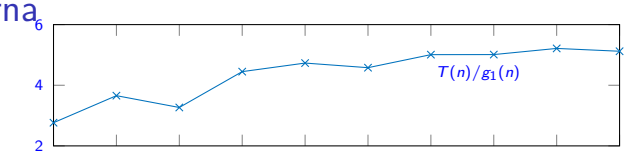
- Mät tiden för ett antal n
- Gissa $O(n)$, ev. med stöd från teori
- Plotta uppmätt tid/hypotetisk ($T(n)/g(n)$ enligt ordo-def)
- Borde gå mot positiv konstant för stora värden för korrekt gissning

n	1	2	3	4	5	6	7	8	9	10
$g(n) = n^2$	1	4	9	16	25	36	49	64	81	100
$T(n)$ ($\cdot 10^{-6}$)	3.54	14	25	49	75	107	147	192	247	303
$T(n)/g(n)$	3.54	3.46	2.75	3.05	3.01	2.96	2.99	3.01	3.04	3.03



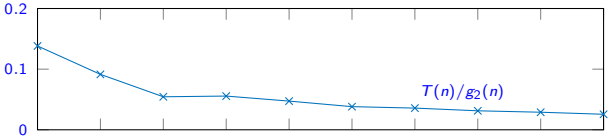
Testa hypoteserna

$g_1(n) = n$



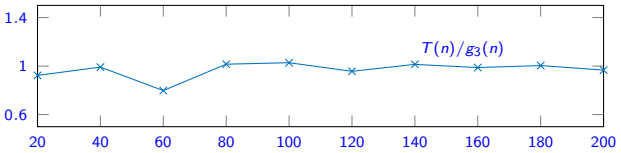
Fortsätter växa. g_1 växer för långsamt!

$g_2(n) = n^2$



Fortsätter avta. g_2 växer för snabbt!

$g_3(n) = n \log n$



Konvergerar. g_3 växer lagom snabbt! ^{n}

Experimentell analys

- Fördelar:
  - Behöver inte källkoden
  - Behöver "bara" ett körbart program
- Begränsningar:
  - Måste implementera och testa algoritmen
  - Experimenten kan endast utföras på en begränsad (liten) mängd data
  - Man kan missa viktiga testdata (specialfördelningar)
  - Hård- och mjukvaran måste vara densamma för alla körningar
  - Modern, "intelligent", strömsparande mjuk- och hårdvara kan variera hastigheten på processorn

Asymptotisk analys

- Utgå från högnivåbeskrivning av algoritmer istället för implementation
- Oberoende av hårdvaran och mjukvaran
- Kan beräkna teoretiska bästa- och värsta-fallen
- Utgå från pseudokoden:
  1. Räkna operationer
  2. Ställ upp ett tidsuttryck $T(n)$ för totala antalet operationer som en funktion av problemstorleken n
  3. Förenkla tidsuttrycket $T(n)$
  4. Ta fram en funktion $g(n)$ och konstanter c, n_0 som uppfyller Ordo-definitionen

Analys av algoritmer (1)

- Primitiva operationer:
  - Lågnivå-beräkningar som är oberoende av programspråk
  - Kan definieras i termer av pseudokod
  - Vi väljer att räkna dessa operationer som primitiva:
    - Anropa en metod/funktion
    - Returnera från en metod/funktion
    - Utföra en aritmetisk operation (*, -, . . .)
    - Jämföra två tal, etc.
    - Referera till (läs av eller tilldela) en variabel eller objekt
    - Indexera i en array
  - Varje operation antas ta samma tid
- Inspektera pseudokoden och räkna antalet primitiva operationer!

Analys av algoritmer (2)

- Kraftig abstraktion:
  - Vi bortser från hårdvaran (tid per operation)
  - Vi bortser från att olika operationer tar olika lång tid
- Alternativet är att titta på de verkliga tiderna för de olika operationerna, ex. långsam indexering eller $\sqrt{\cdot}$ långsamt
  - Ger en maskin-beroende analys

Exempel 1 — while-loop (1)

```
Algorithm Sum1(n)
    Sum all numbers 1..n (while version)

sum ← 0                          1
i ← 1                            1
while i <= n do                  (n + 1) · 3 + n · [ ]
    sum ← sum + i                    4
    i ← i + 1                        3
return sum                       2
```

- Testet i lådan körs $n + 1$ gånger
- Loopen körs n gånger

Exempel 1 — while-loop (2)

- Summera alla rader till $T(n)$

$$\begin{aligned} T(n) &= 1 + 1 + (n+1) \cdot 3 + n(4+3) + 2 \\ &= 2 + 3n + 3 + 7n + 2 = 10n + 7 \end{aligned}$$

- För beräkning av c och n_0, se tidigare exempel
- $T(n)$ är $O(n)$ med $c = 11$ och $n_0 = 7$

Exempel 2 — for-loop (1)

```
Algorithm Sum2(n)
    Sum all numbers 1..n (for version)

sum ← 0                          1
for i ← 1 to n do                1 + (n + 1) · 3 + 3n + n · [ ]
    sum ← sum + i                    4
return sum                       2
```

- Initial tilldelning
- Testet $i \leq n$ syns inte i pseudokoden, körs $n + 1$ gånger
- Uppräkningen $i \leftarrow i + 1$ syns inte i pseudokoden, körs n gånger
- Loopen körs n gånger

Exempel 2 — for-loop (2)

- Summera alla rader till $T(n)$

$$T(n) = 1 + 1 + (n+1) \cdot 3 + n(4+3) + 2 = 10n + 7$$

- Samma som while-exemplet! (Slump?)

Exempel 3 — summera jämna tal

```
Algorithm SumAllEven(n)
    Sum all even numbers 2..n

sum ← 0                          1
i ← 2                            1
while i <= n do                  (n/2 + 1) · 3 + n/2 · [ ]
    sum ← sum + i                    4
    i ← i + 2                        3
return sum                       2
```

- Loopen körs $n/2$ gånger

$$T(n) = 1 + 1 + 3(n/2 + 1) + n/2(4 + 3) + 2 = 5n + 7$$

$$g(n) = n$$

$$c = \lim_{n\to\infty}\left(\frac{T(n)}{g(n)} + 1\right) = \lim_{n\to\infty}\left(\frac{5n+7}{n} + 1\right) = 6$$

$5n + 7 \leq 6n$ för $n \geq 7 \rightarrow n_0 = 7$

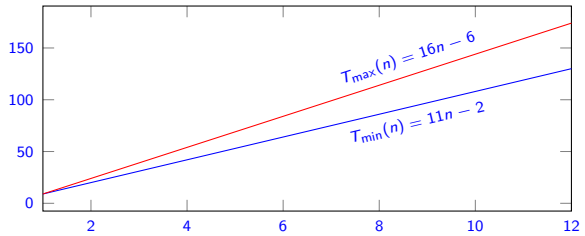
Exempel 4 — max i array (1)

```
Algorithm ArrayMax(A,n)
    input: An array A storing n integers
    output: The maximum element of A
currentMax ← A [ 0 ]                    3
for i ← 1 to n-1 do                     1 + 3n + 3(n − 1) + (n − 1) · [ ]
    if currentMax < A [ i ] then            5
        currentMax ← A [ i ]                4
return currentMax                       2
```

- Loopen körs $n - 1$ gånger
- Uppräkningen av i körs $n - 1$ gånger
- Testet utförs n gånger
  - Jag har antagit att loop-testet görs som `i<n`, dvs. 3 operationer per test

Exempel 4 — max i array (2)

- Vi får två fall:
  - Bästa: `if`-testet alltid falskt (det första talet i fältet är störst)
    - $T_{\min}(n) = 3 + 1 + 6n - 3 + (n-1)5 + 2 = 11n - 2$
  - Sämsta: `if`-testet alltid sant (listan sorterad i stigande ordning)
    - $T_{\max}(n) = 3 + 1 + 6n - 3 + (n-1)9 + 2 = 15n - 6$
- Båda fallen $O(n)$ men med olika konstanter c
  - Oftast är n_0 också olika



Typalgoritmer (1)

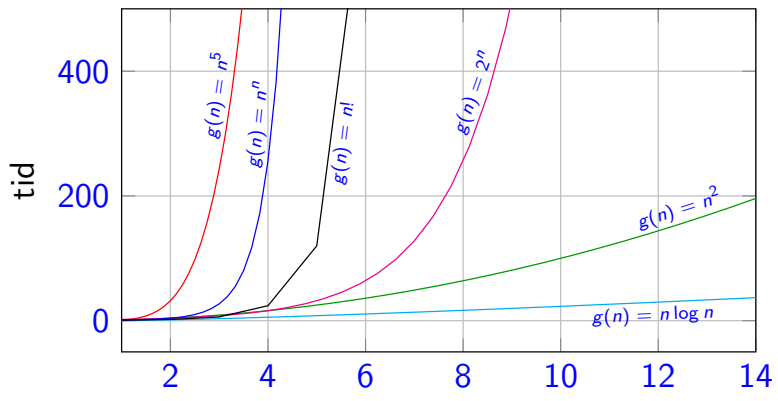
- Speciella klasser av algoritmer/problem:

Konstanta:	$O(1)$
Logaritmiska:	$O(\log n)$
Linjära:	$O(n)$
Kvadratiska:	$O(n^2)$
Polynoma:	$O(n^k), k \geq 1$
Exponentiella:	$O(a^n), a > 1$
Kombinatoriska:	$O(n!)$

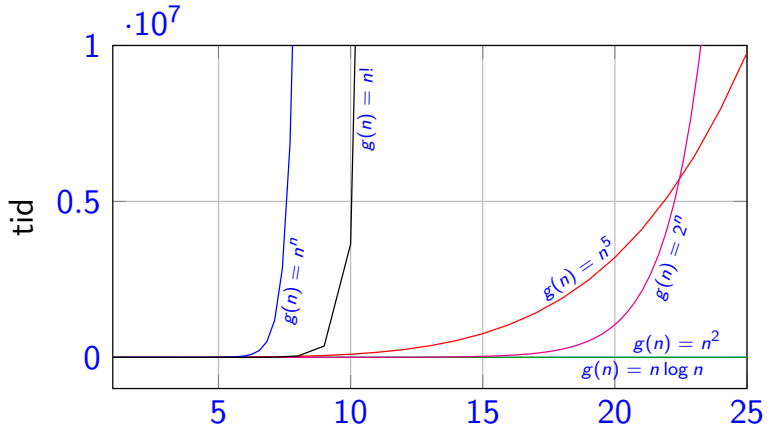
- Typalgoritmerna är ordnade enligt:

$$1 \ll \log n \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n$$

Typalgoritmer (2)



Typalgoritmer (3)



Förenklad asymptotisk analys

- Rita kurvor för $T(n)$ och jämföra är svårt och behövs ofta inte
- Oftast så räcker med en förenklad asymptotisk analys:
  - Ignorera allt utom den dominerande termen, dvs. lägre ordningens termer och konstanter
  - Använd inga koefficienter i $g(n)$
- Exempel:
  - $T(n) = 10n + 7$ är $O(n)$
  - $T(n) = 8n^3 + 5n^2 + n - 10$ är $O(n^3)$
  - $T(n) = 8n^2 \log n + 10n^2$ är $O(n^2 \log n)$

Sammanfattning

- $O(\cdot)$ används för att uttrycka antalet primitiva operationer som utförs som en funktion av storleken på indata n
- Det är en övre gräns för tillväxt
- En algoritm som körs på $O(n)$ är snabbare än en $O(n^2)$, men $O(\log n)$ är snabbare än $O(n)$ (för stora n)

$$1 \ll \log n \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n! \ll n^n$$

Varning!

- Var aktsam, stora konstanter c ställer till det:
  - $T_1(n) = 1000000n$ är en linjär algoritm $O(n)$
  - $T_2(n) = 2n^2$ är en kvadratisk algoritm $O(n^2)$
  - $T_2(n)$ är snabbare för "små" datamängder, $n < n_0 = 5 \cdot 10^{13}$
- O-notationen är en förenkling och en övre gräns
- O-notationen har tagit bort kopplingen till hårdvaran
  - T.ex. kan cache-minne, *CPU throttling*, etc. påverka analysen

Genväg, grovanalys

- Man kan många gånger skippa vägen över $T(n)$ helt
- Väldigt grov uppskattning av tillväxten
- Man gör en okulärbesiktning av algoritmen:
  - Initiera en array är $O(n)$
  - Enkelloop $O(n)$
  - Dubbelloop $O(n^2)$
  - Nästlade loopar är $O(n) \cdot O(n) \cdots O(n) = O(n^k)$

Grovanalys, exempel 1

```
Algorithm PrefixAv1(X,n)
    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]

A ← CreateArray(n)
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
```

- En initiering, två nästlade loopar:
  $T(n) = O(n) + O(n)O(n) = \boxed{O(n^2)}$
- Detaljerad analys ger $T(n) = 6n^2 + 38n + 6 = \boxed{O(n^2)}$

Grovanalys, exempel 2

```
Algorithm PrefixAv2(X,n)
    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]

A ← CreateArray(n)
s ← 0
for i ← 0 to n-1 do
    s ← s + X [ i ]
    A [ i ] ← s / ( i + 1 )
return A
```

- En initiering, en loop: $T(n) = O(n) + O(n) = \boxed{O(n)}$
- Detaljerad analys ger $T(n) = 20n + 5 = \boxed{O(n)}$

Rumskomplexitet (minnesåtgång)

- Vi kan använda metoder liknande asymptotisk komplexitetsanalys
  - I stället för primitiva operationer så räknar vi hur mycket minne algoritmen behöver
  - Glöm inte minnet för lokala variabler, etc., som läggs upp på stacken vid rekursion
- Ofta är minnet en hård begränsning medan tid är en mjukt
  - Åtkomsttiden när primärminnet tar "slut" (disk används i stället) ökar några tiopotenser
  - Ansatsen blir att räkna ut "Vilket är det största problem som ryms i minnet"?
  - Ofta relativt enkelt att räkna ut c (hur många bytes per element som behövs)
- Mer komplex analys tar hänsyn till många olika begränsningar
  - tid, minne, filåtkomst, kommunikation (nätverk, mellan CPU:er), osv.

OU2

- Quiz i Canvas
  - 3 försök före första deadline, högsta poängen räknas
- Till deadline #2 är det ett annat quiz

Problemlösningsstrategier

Design av algoritmer

- Problemlösningsstrategier:
  - Top-down:
    - Börja med en övergripande bild av problemet
    - Bryt ner problemet i mindre delar
    - Lös delarna var för sig eller dela upp ännu mer
  - Bottom-up:
    - Börja med smådelarna
    - Bygg ihop till större lösningar
- Typer av algoritmer (lösningstekniker)
  1. *Brute force* ("råstyrka")
  2. Giriga algoritmer (*Greedy algorithms*)
  3. Söndra och härska (*Divide-and-Conquer*)
  4. Dynamisk programmering
  5. Backtracking

Brute force

- Rättfram, "naiv", ansats
  - Utgå direkt från problemställningen
- Om problemet är kombinatoriskt — gör en fullständig sökning!
  - Generera och enumerera alla tänkbara lösningar
  - Testa varje lösning
    - Välj den bästa lösningen
- Bra metod att starta med
  - Ofta enkla algoritmer
  - Garanterar en korrekt lösning om en sådan finns
  - Men. . . garanterar inte effektivitet

Brute force, exempel

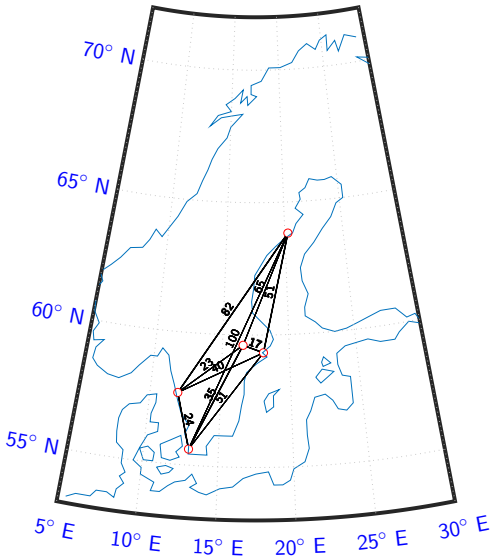
- Linjärsökning i lista:
  - Börja från början och kolla varje element
  - När listan är slut, meddela om det sökta elementvärdet finns
- Max värde i ett fält

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


Brute force, exempel — Handelsresande-problemet

Travelling Salesman Problem — TSP

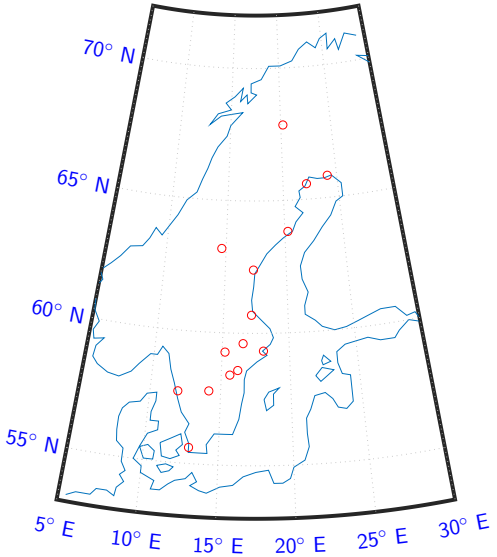
- Givet n städer, finn den kortaste rutten som besöker varje stad exakt en gång
- Komplexitet?
- För $n = 5$, $\boxed{12}$ alternativ:
  1. 1-2-3-4-5-1: 247 mil
  2. 1-2-3-5-4-1: 216 mil
  3. 1-2-4-3-5-1: 274 mil
  4. 1-2-4-5-3-1: 273 mil
  5. 1-2-5-3-4-1: 249 mil
  6. 1-2-5-4-3-1: 280 mil
  7. 1-3-2-4-5-1: 240 mil
  8. 1-3-2-5-4-1: 215 mil
  9. 1-3-4-2-5-1: 274 mil
  10. 1-3-5-2-4-1: 242 mil
  11. 1-4-2-3-5-1: 210 mil
  12. 1-4-3-2-5-1: 216 mil



Brute force, exempel — Handelsresande-problemet

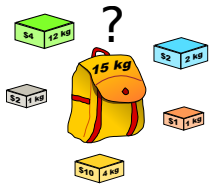
Travelling Salesman Problem — TSP

- Komplexitet: $(n-1)!/2$
- För $n = 5$, $\boxed{12}$ alternativ
- För $n = 15$: $\boxed{4.4 \cdot 10^{10}}$ alternativ
- Brute force är inte en hållbar strategi för större kombinatoriska problem



Brute force, exempel — *The 0-1 Knapsack Problem* (1)

- Givet en mängd med n element där element i har värde $v_i > 0$ och en vikt $w_i > 0$:
  - *Välj element med maximalt värde utan att den totala vikten blir mer än W.*
- Låt $x_i \in \{0, 1\}$:
  - Om $x_i = 1$ så är elementet med
  - Om $x_i = 0$ så låter vi elementet vara

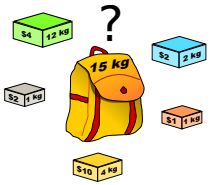


Brute force, exempel — *The 0-1 Knapsack Problem* (2)

- Matematisk formulering:

$$\max_{x_i \in \{0,1\}} \sum_{i=1}^{n} x_i v_i \text{ med begränsningen } \sum_{i=1}^{n} x_i w_i \leq W$$

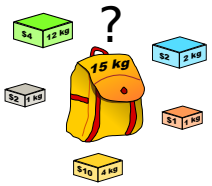
- Komplexitet?
  - För varje extra n fördubblas antalet möjligheter



Brute force, exempel — *The 0-1 Knapsack Problem*

Element	Värde	Vikt
1	4	12
2	2	2
3	2	1
4	1	1
5	10	4

Maxvikt: $W = 15$



i	x_1	x_2	x_3	x_4	x_5	$\sum x_i w_i$	$\sum x_i v_i$
0	0	0	0	0	0	0	0
1	0	0	0	0	1	4	10
2	0	0	0	1	0	1	1
3	0	0	0	1	1	5	11
4	0	0	1	0	0	1	2
5	0	0	1	0	1	5	12
6	0	0	1	1	0	2	3
7	0	0	1	1	1	6	13
8	0	1	0	0	0	2	2
9	0	1	0	0	1	6	12
10	0	1	0	1	0	3	3
11	0	1	0	1	1	7	13
12	0	1	1	0	0	3	4
13	0	1	1	0	1	7	14
14	0	1	1	1	0	4	5
15	0	1	1	1	1	8	15
16	1	0	0	0	0	12	4
17	1	0	0	0	1	16	14
18	1	0	0	1	0	13	5
19	1	0	0	1	1	17	15
20	1	0	1	0	0	13	6
21	1	0	1	0	1	17	16
22	1	0	1	1	0	14	7
23	1	0	1	1	1	18	17
24	1	1	0	0	0	14	6
25	1	1	0	0	1	18	16
26	1	1	0	1	0	15	7
27	1	1	0	1	1	19	17
28	1	1	1	0	0	15	8
29	1	1	1	0	1	19	18
30	1	1	1	1	0	16	9
31	1	1	1	1	1	20	19

Brute force, summering

- Många problem saknar känd bättre lösning
- Ger ofta hög tidskomplexitet
- Går många gånger att effektivisera de naiva algoritmerna
  - Testa alternativen i någon speciell ordning
    - Ex. tyngsta objekten först
  - Avbryta tidigt om vägen omöjligt leder till en lösning
    - Ex. om maxvikt uppnåtts efter två element, ingen idé kontrollera resten
- Eller om vi kan tänka oss att modifiera målet
  - Avsluta när vi funnit en lösning
    - Ex. första lösningen med maxvikt
  - Avslutar när vi funnit en lösning som är nästan optimal
    - Ex. första lösningen över 90% av teoretiskt bästa
  - Eller så relaxerar vi problemet (släpper på någon begränsning)
    - Ex. kan ta med en del av objekt X

Giriga (*Greedy*) algoritmer

- Metod:
  - I varje steg, titta på alla möjliga nästa steg och välj det som ger störst förbättring
- Bra för vissa optimeringsproblem
  - Om den optimala lösningen kan nås via stegvisa lokala förändringar av starten
- Giriga algoritmer specialfall av heuristiska (tumregelsbaserade)
  - Tumregel: Ta så mycket som möjligt så fort som möjligt!
- Kan vara bra alternativ till *brute force*-algoritmer

Giriga algoritmer, exempel

- Problem: Lämna tillbaka växel med så få mynt som möjligt
  - Heuristik:
    - Ta alltid myntet med högst värde i varje iteration
- Grafalgoritmer
  - Minimalt uppspännande träd
    - Kruskals algoritm (senare)
    - Prims algoritm (senare)
  - Kortaste vägen
    - Dijkstras algoritm (senare)
- Huffman-kodning (senare)

The 0-1 Knapsack Problem, girig algoritm, exempel

- Garanterar inte en optimal lösning
- Exempel:
  - Ryggsäcken tål max 4 kg
  - Vi har följande element:

Element	Värde	Vikt
A	1.65	3
B	1	2
C	1	2

  - Regeln "välj den värdefullaste först" skulle göra att vi valde A och sen stopp
  - Lösningen B+C är optimal med värdet 2

Relaxering — *The Fractional Knapsack Problem*

- Här får man ta en del (*fraction*) av varje element
- Mål: Välj element med maximal förtjänst utan att den totala vikten blir mer än den maximala vikten W
- Låt $x_i \in [0, 1]$ vara andelen vi tar av element i
- Matematisk formulering:

$$\max_{x_i \in [0,1]} \sum_{i=1}^{n} x_i v_i \text{ med begränsningen } \sum_{i=1}^{n} x_i w_i \leq W$$

- Regel: För varje gång, ta elementet med maximalt *värde per viktenhet* v_i / w_i
- Kan lösas i $O(n \log n)$ tid (sortering)

The Fractional Knapsack Problem, exempel

Element	Värde	Vikt	Värde/vikt
5	10	4	2.5
3	2	1	2
2	2	2	1
4	1	1	1
1	4	12	0.33

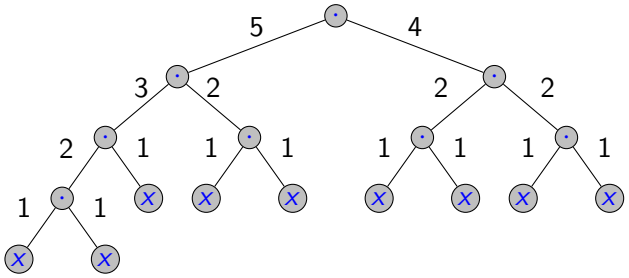
i	x_1	x_2	x_3	x_4	x_5	$\sum x_i w_i$	$\sum x_i v_i$
1					1	4	10
2			1		1	5	12
3		1	1		1	7	14
4		1	1	1	1	8	15
~~5~~	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	~~20~~	~~19~~
6	$\frac{7}{12}$	1	1	1	1	15	$17\frac{1}{3}$

Söndra och härska (*Divide-and-Conquer*)

- Metod:
  - Söndra: Dela upp problemet i två eller flera delar som löses rekursivt
    - Delarna bör vara ungefär lika stora
  - Härska: Kombinera dellösningarna till en slutlösning
- Leder till rekursiva algoritmer
  - Kan vara en bra lösning om det är svårt hitta iterativa lösningar.
  - Är ibland effektivare även om det finns iterativ lösning.
  - Ibland skapas en dellösning många gånger (= ineffektivt)
- Komplexitet $O(n \log n)$ är vanligt
- *Merge-sort* och *Quick-sort* är bra exempel

Söndra och härska, exempel: Beräkna x^n (dåligt exempel)

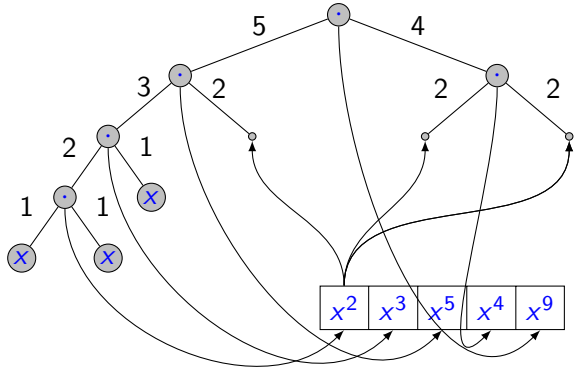
- Beräkna $f(x) = x \cdot x \cdots x$ iterativt ger en algoritm som är $O(n)$
- *Divide-and-Conquer*: Vi kan bryta ner problemet och beräkna $x^{\lceil n/2 \rceil} \cdot x^{\lfloor n/2 \rfloor}$ rekursivt
- Exempel: Beräkna x^9



- Fast det ger inget!
- I de rekursiva anropen så beräknar vi ofta samma värden
  - Kan vi utnyttja detta och vinna något?

Dynamisk programmering

- Använder lite minne till att undvika att lösa samma delproblem flera gånger
- Metod:
  - Ställ upp en tabell som lagrar redan kända lösningar
  - För varje nytt anrop, kolla om delproblemet är redan löst
  - Om inte, lös det och sätt in lösningen i tabellen



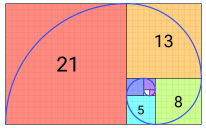
Andra exempel

- En-dimensionellt:
  - Fibonacci-sekvensen:

$$F(n) = F(n-1) + F(n-2),$$
$$F(0) = 0,$$
$$F(1) = 1$$

  - Sekvensen blir
    - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

    och den dyker upp på många ställen i naturen[4]



- Multi-dimensionell dynamisk programmering:
  - 0-1 Knapsack
  - Matrisbaserad shortest path (Floyd) (senare)

[4] https://en.wikipedia.org/wiki/Fibonacci_number

Backtracking

- Backtracking är en algoritmstrategi för att finna en eller alla lösningar till ett *constraint satisfaction*-problem
  - *Constraint satisfaction*-problem är matematiska problem som är definierade som en mängd objekt vars tillstånd måste uppfylla ett antal begränsningar
- Bygger kandidater till lösningar steg för steg och överger ("backtracks") varje partiell kandidat så snart den kommer fram till att den inte kan leda fram till en giltig lösning ("återvändsgränd")

Exempel — Sudokulösare

```
Algorithm sudoku-solver(g: Grid)
if board-filled(grid) then
  // success!
  return (True, grid)
// Find a free position
(row, col) ← find-unassigned-square(grid)
// Try to input 1 through 9 at the position
for num ← 1 to 9 do
  // If we will not break any rules...
  if is-safe(grid, row, col, num) then
    // ...set the value
    grid ← set-value(grid, row, col, num)
    // Can we find a remaining solution?
    if sudoku-solver(grid) then
      // Yes, we're done!
      return (True, grid)
    else
      // Nope, reset position
      grid ← set-value(grid, row, col, 0)
// No solution found, give up (backtrack)
return (False, None)
```