TDDE22 & 725G97

Kursintroduktion och tidskomplexitet

Magnus Nielsen

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Personal

- Examinator - Filip Strömbäck
- Kursledare - Magnus Nielsen
- Assistent - Elias Leijonmarck
- Assistent - Johan Christiansson
- Assistent - Martin Jonsson
- Assistent - Mikael Lundgren
- Assistent - Simon Törnqvist
- Kursadministratör - Annelie Almquist

Litteratur

- OpenDSA
- Uppgiftssamling till lektionerna (.pdf på kurshemsidan efter varje pass)
- Labbkompendium (.pdf på kurshemsidan)

Examination

- Skriftlig tentamen (oktober, januari och augusti)
- Uppgifter i OpenDSA
- 4 + 1 laborationsuppgifter
- Basgruppsarbete (Gäller endast IT-programmet)
- Seminarie (Gäller endast 725G97).

Examination

- Skriftlig tentamen (oktober, januari och augusti)
- Uppgifter i OpenDSA
- 4 + 1 laborationsuppgifter
- Basgruppsarbete (Gäller endast IT-programmet)
- Seminarie (Gäller endast 725G97).

Viktigt

Laborationerna skall vara *redovisade och godkända* senast vid första tentamenstillfället (2023-10-26). Rapportering av laborationer och OpenDSA kan bara garanteras i samband med ordinarie kurstillfälle.

LINKÖPING UNIVERSITY

Föreläsningar

- Administration, mål, komplexitet, ordo-notation, algoritmanalys
- Enkla abstrakta datatyper (ADT): stack, kö, lista
- Hashning, skiplistor
- Binära sökträd
- AVL-träd, multivägs-sökträd
- Prioritetsköer, heapar
- Sortering I
- Sortering II
- ADTn graf
- Grafsökning, topologisk sortering, kortaste väg
- Reserv (generellt tentaföreläsning)

Lektioner

- 4 handledda lektioner (Nytt från förra året!)
  - Tidskomplexitet och Ordo-notation
  - Testning
  - AVL-träd
  - Grafer och intro till Lab4.
- Uppgifter tillhandahålls vid lektionstillfället och finns senare på kurswebbsidan.
- Var vänlig registrera er omgående i webReg.

Laborationer (Java)

- Lab1: Hashning
- Lab2: Balanserade sökträd
- Lab2.5: Hemuppgift (i par), heapar / effektivisering av algoritm.
- Lab3: Problemlösande sortering
- Lab4: Ordkedjor

Bonusproblem

- Finns på kurswebbsidan
- 0 lösta uppgifter ger 0 bonuspoäng.
- Lösta bonusuppgifter ger 0.5 bonuspoäng per uppgift, upp till max 2 (ca 5 % av totala tentabetyget), räknas bara mot högre betyg.

Din insats i kursen

- Följ undervisningen (om du vill)
- Plugga under hela kursen (viktigt!)
- Gör uppgifterna i OpenDSA
- Gör datorlabb 1–4
- IT-programmet: Vinjetter
- 725G97: Seminarie
- Hemuppgift: Lab2.5
- Lös bonusproblem för att få bonuspoäng (frivlligt)

Kursens hemsida

`http://www.ida.liu.se/~TDDE22` (ej Lisam)

LINKÖPING UNIVERSITY

Kursöversikt

Datastrukturer

Hur lagrar man data effektivt

- Teoretiskt sett effektiva datastrukturer
- Praktiskt sett effektiva datastrukturer
- Förståelse för datastrukturer för att effektivt lösa problem

Algoritmer

Hur löser man problem effektivt

- Analys av komplexitet
- Exempel på olika slags algoritmer
  - Sorteringsalgoritmer
  - Grafalgoritmer

Varför ska man plugga DALG?

Gammalt ursprung, nya möjligheter

- Studiet av algoritmer har pågått åtminstone sedan Euklides
- Formaliserades av Church och Turing på 1930-talet
- Vissa viktiga algoritmer upptäcktes av studenter i denna typ av kurser!
- Mer moderna, och tidsenliga, anledningar. Vad?

Varför ska man plugga DALG?

Gammalt ursprung, nya möjligheter

- Studiet av algoritmer har pågått åtminstone sedan Euklides
- Formaliserades av Church och Turing på 1930-talet
- Vissa viktiga algoritmer upptäcktes av studenter i denna typ av kurser!
- Mer moderna, och tidsenliga, anledningar. Vad?
  - Miljö

Varför ska man plugga DALG?

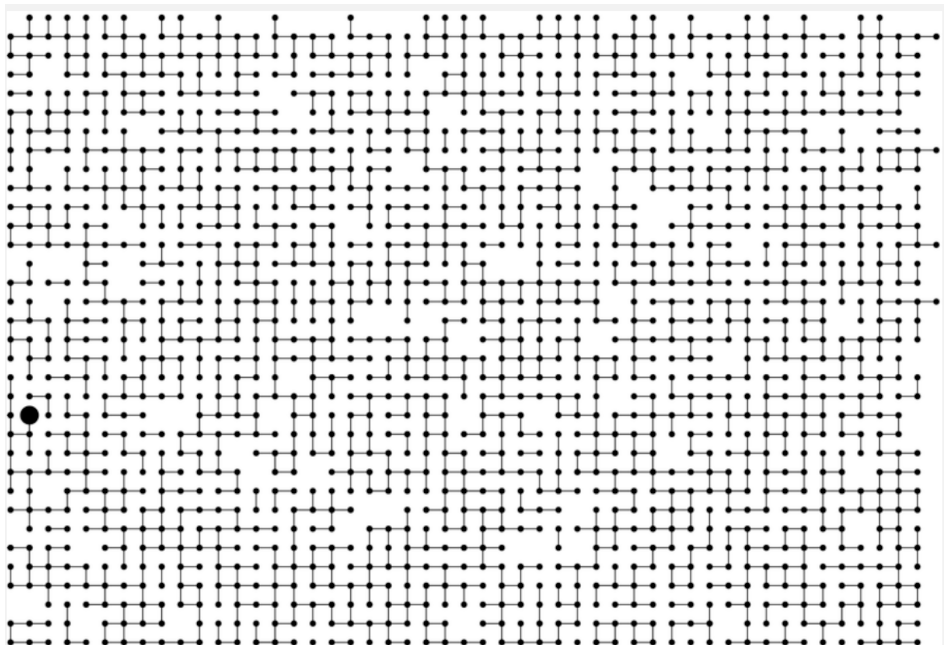
Gammalt ursprung, nya möjligheter

- Studiet av algoritmer har pågått åtminstone sedan Euklides
- Formaliserades av Church och Turing på 1930-talet
- Vissa viktiga algoritmer upptäcktes av studenter i denna typ av kurser!
- Mer moderna, och tidsenliga, anledningar. Vad?
  - Miljö
  - Ekonomi

Varför ska man plugga DALG?

För att kunna lösa annars olösbara problem

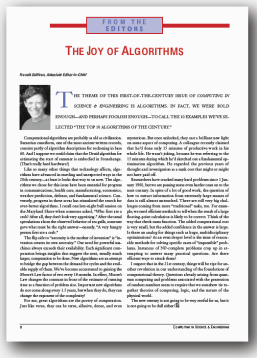
- T.ex. nätverkskonnektivitet



Varför ska man plugga DALG?

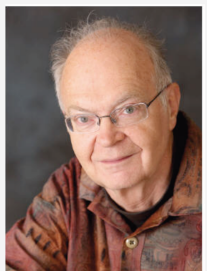
För intellektuell stimulans

> *“ For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. ”* — *Francis Sullivan*


FROM THE EDITORS

THE JOY OF ALGORITHMS

> *“ An algorithm must be seen to be believed. ”* — *Donald Knuth*



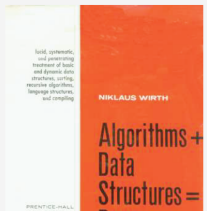
Varför ska man plugga DALG?

För att bli en kunnig programmerare (skriva effektiva program!)

" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships. "

— Linus Torvalds (creator of Linux)

" Algorithms + Data Structures = Programs. " *— Niklaus Wirth*



Varför ska man plugga DALG?

För att DALG kan hjälpa oss att lista ut saker om livet och universum

> *" Computer models mirroring real life have become crucial for most advances made in chemistry today.... Today the computer is just as important a tool for chemists as the test tube. "*
>
> — *Royal Swedish Academy of Sciences*
> *(Nobel Prize in Chemistry 2013)*

DALG – grunder

Abstrakta datatyp (ADT)

maskinoberoende högnivåbeskrivning av data och operationer på data, ex: Stack, Kö, Lista...

Datastruktur

logisk organisation av datorns minne för att lagra data

Algoritm

högnivåbeskrivning av konkreta operationer på datastrukturer

ADT

implementeras med lämpliga datastrukturer och algoritmer

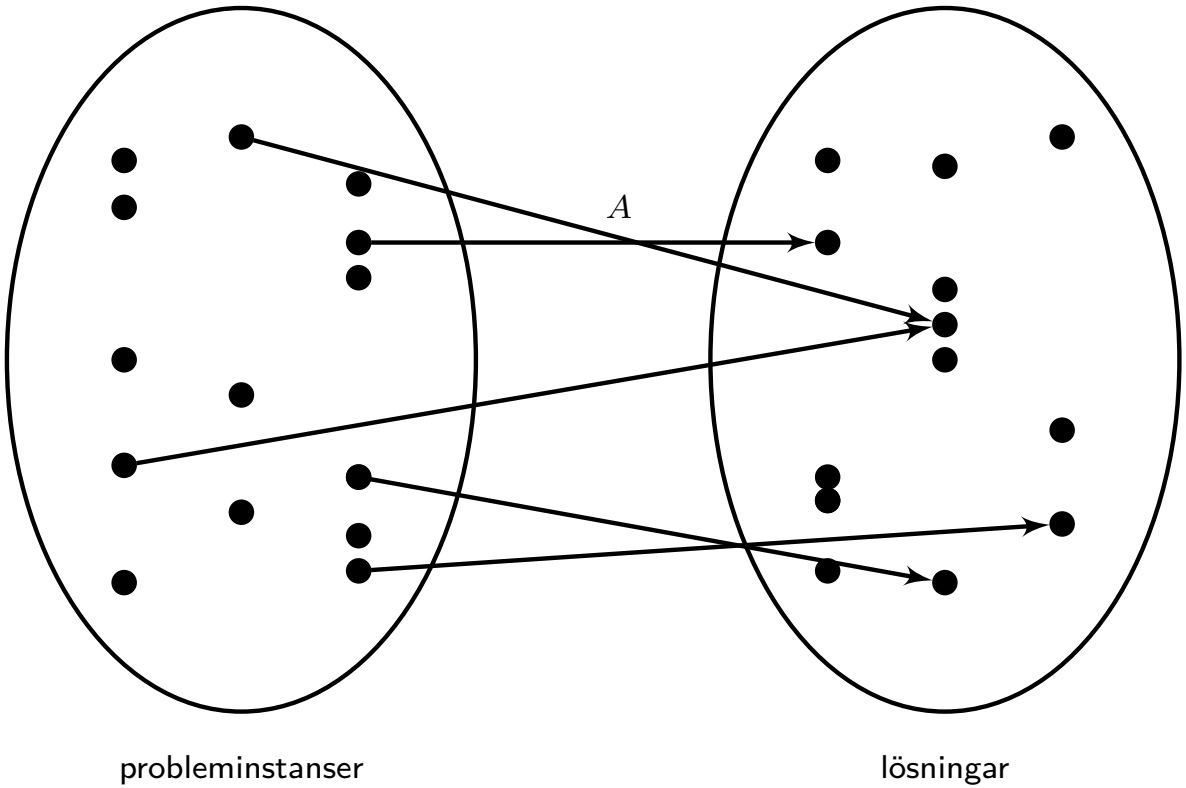
Program

implementerar algoritmer och datastrukturer i något visst programspråk

Algoritm

En *algoritm* är en ändlig beskrivning av hur man steg för steg löser ett problem.

En algoritm tar oftast *indata* som beskriver en *probleminstans* och producerar *utdata* som beskriver probleminstansens *lösning*.
En algoritm kan ses som en funktion
A : PROBLEMINSTANSER $\rightarrow$ LÖSNINGAR.



Korrekthet

att det beräknade utdatat för varje givet indata stämmer med problemets beskrivning

Algoritmanalys

tids- och minnesåtgång, skalbarhet, effektivitet, värsta fallet, bästa fallet, medelfallet, amorterad analys

Algoritmiska paradigm

vanligt förekommande problemlösningsstrategier, t.ex. rekursiv nedbrytning, dynamisk programmering, giriga algoritmer, ...

Pseudokod

algoritmer beskrivs med pseudokod, en blandning av naturligt språk och programmeringskonstruktioner

ADT – diskussion

En ADT berättar *vad* som ska göras: en uppsättning operationer på data.

För att beskriva *hur* detta ska göras

- väljer vi en datastruktur (en representation i minnet)
- konstruerar vi algoritmer som utför ADT-operationerna

Samma ADT kan implementeras

- med olika datastrukturer
- med olika algoritmer

Algoritmerna beror på den valda datastrukturen.
Välj den *mest effektiva* lösningen. Vad är effektivitet?

Analys av algoritmer

Vad ska vi analysera?

- Korrekthet (bara praktiskt i denna kurs)
- Terminering (inte i denna kurs)
- Effektivitet, resursförbrukning, komplexitet

Tidskomplexitet — hur lång tid tar algoritmen i värsta fallet?

- som funktion av vad?
- vad är ett tidssteg?

Minneskomplexitet — hur stort minne behöver algoritmen i värsta fallet?

- som funktion av vad?
- mätt i vad?
- tänk på att funktions- och proceduranrop också tar minne

Hur man kan jämföra effektivitet

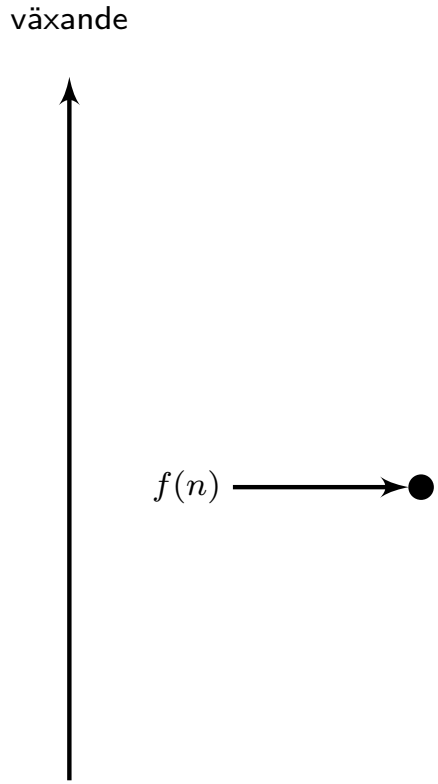
- Studera exekveringstid (eller minnesåtgång) som en funktion av storleken på indata.
- När är två algoritmer "lika effektiva"?
- När är en algoritm bättre än en annan?

Jämförelse med några elementära funktioner

n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	1	2	2	4	4
16	4	16	64	256	$6.5 \cdot 10^4$
64	6	64	384	4096	$1.84 \cdot 10^{19}$

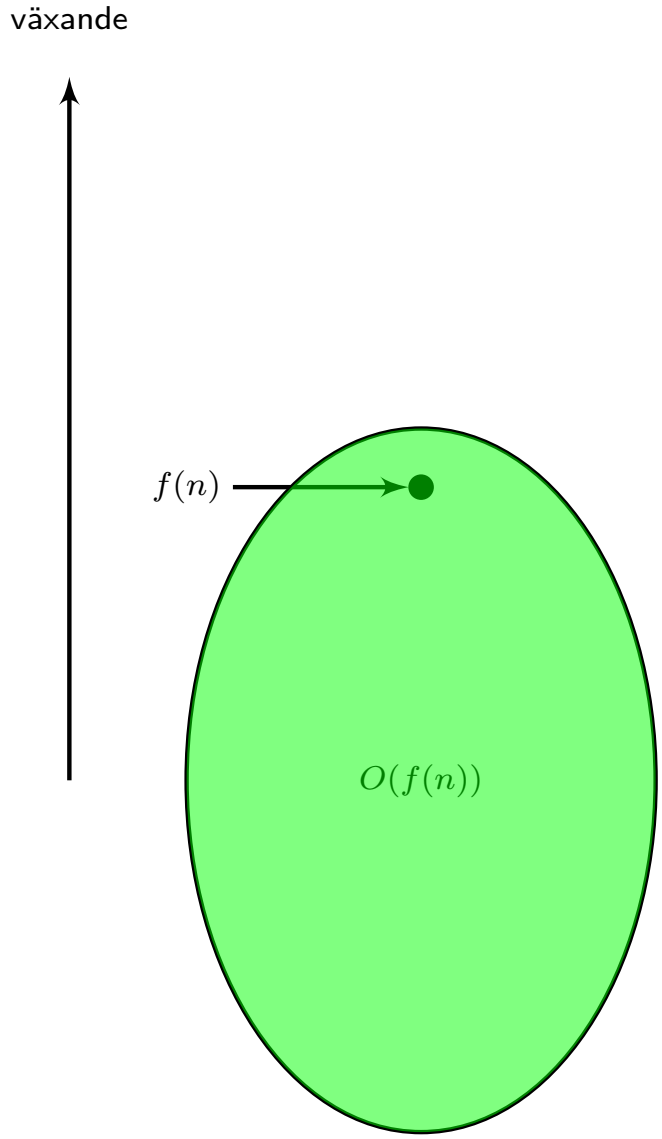
$1.84 \cdot 10^{19} \mu$sekunder $= 2.14 \cdot 10^8$ dagar $= 583.5$ årtusenden

Hur komplexitet kan anges



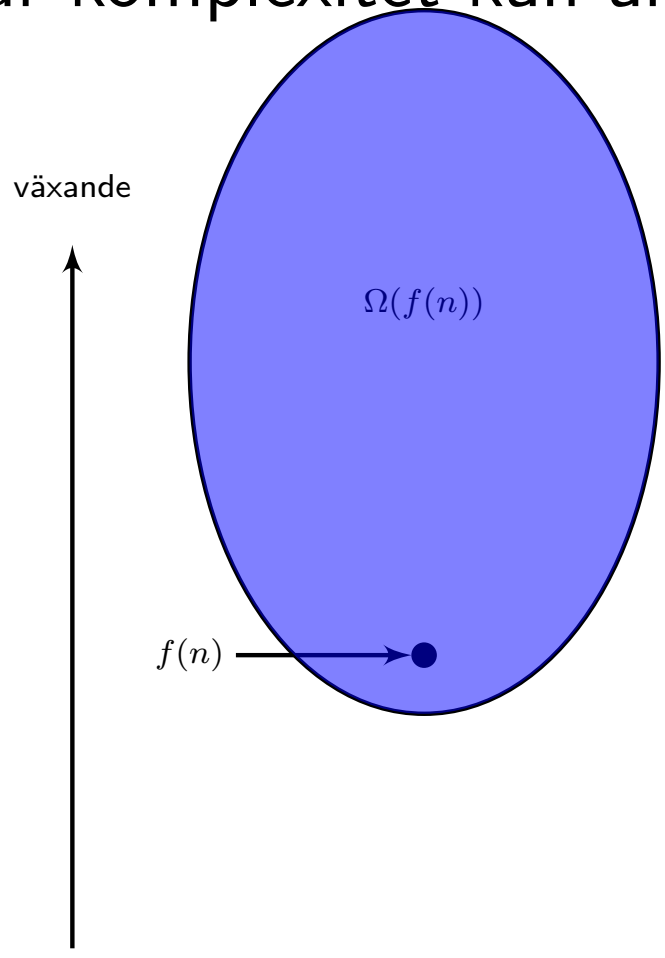
- Hur ändras komplexiteten för växande storlek n på indata?
- Asymptotisk komplexitet — vad händer när n växer mot oändligheten?
- Mycket enklare om vi bortser från konstanta faktorer.

Hur komplexitet kan anges



- Hur ändras komplexiteten för växande storlek n på indata?
- Asymptotisk komplexitet — vad händer när n växer mot oändligheten?
- Mycket enklare om vi bortser från konstanta faktorer.
- $O(f(n))$ – växer högst lika snabbt som $f(n)$

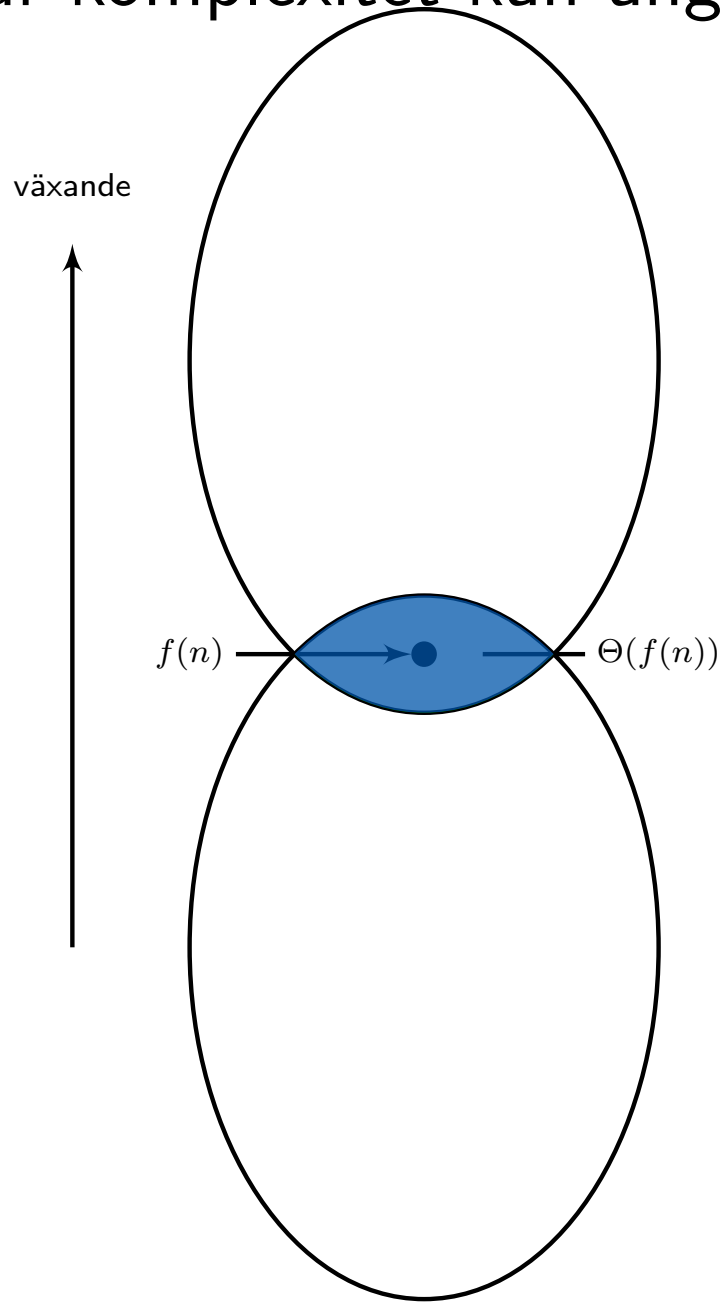
Hur komplexitet kan anges



- Hur ändras komplexiteten för växande storlek n på indata?
- Asymptotisk komplexitet — vad händer när n växer mot oändligheten?
- Mycket enklare om vi bortser från konstanta faktorer.

- $\Omega(f(n))$ – växer minst lika snabbt som $f(n)$

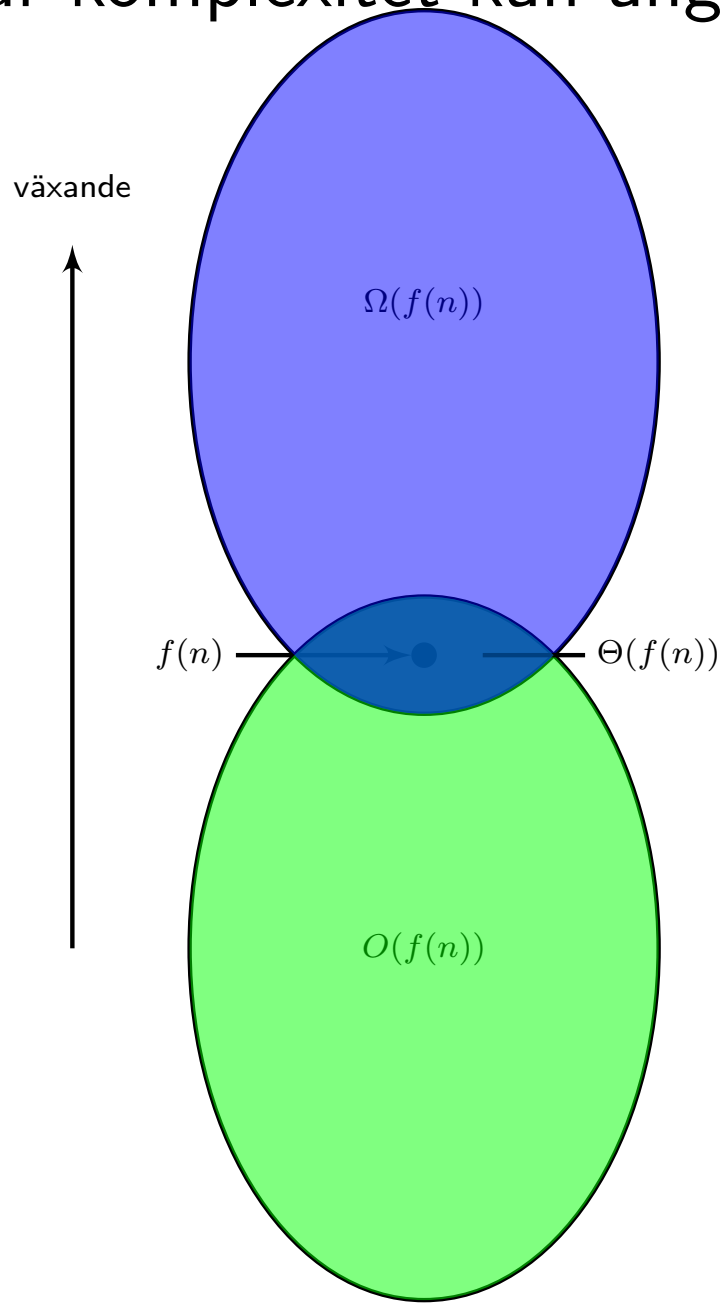
Hur komplexitet kan anges



- Hur ändras komplexiteten för växande storlek n på indata?
- Asymptotisk komplexitet — vad händer när n växer mot oändligheten?
- Mycket enklare om vi bortser från konstanta faktorer.

- $\Theta(f(n))$ – växer lika snabbt som $f(n)$

Hur komplexitet kan anges



- Hur ändras komplexiteten för växande storlek n på indata?
- Asymptotisk komplexitet — vad händer när n växer mot oändligheten?
- Mycket enklare om vi bortser från konstanta faktorer.
- $O(f(n))$ – växer högst lika snabbt som $f(n)$
- $\Omega(f(n))$ – växer minst lika snabbt som $f(n)$
- $\Theta(f(n))$ – växer lika snabbt som $f(n)$

Ordo-notation

f, g: växande funktioner från $\mathbb{N}$ till $\mathbb{R}^+$

- $f \in O(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \leq c \cdot g(n)$ för alla $n \geq n_0$
  Intuition: Bortsett från konstanta faktorer växer f inte snabbare än g

Ordo-notation

f, g: växande funktioner från $\mathbb{N}$ till $\mathbb{R}^+$

- $f \in O(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \leq c \cdot g(n)$ för alla $n \geq n_0$
  Intuition: Bortsett från konstanta faktorer växer f inte snabbare än g
- $f \in \Omega(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \geq c \cdot g(n)$ för alla $n \geq n_0$
  Intuition: Bortsett från konstanta faktorer växer f minst lika fort som g

Ordo-notation

f, g: växande funktioner från $\mathbb{N}$ till $\mathbb{R}^+$

- $f \in O(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \leq c \cdot g(n)$ för alla $n \geq n_0$
  Intuition: Bortsett från konstanta faktorer växer f inte snabbare än g
- $f \in \Omega(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \geq c \cdot g(n)$ för alla $n \geq n_0$
  Intuition: Bortsett från konstanta faktorer växer f minst lika fort som g
- $f(n) \in \Theta(g(n))$ omm $f(n) \in O(g(n))$ och $g(n) \in O(f(n))$
  Intuition: Bortsett från konstanta faktorer växer f och g lika snabbt

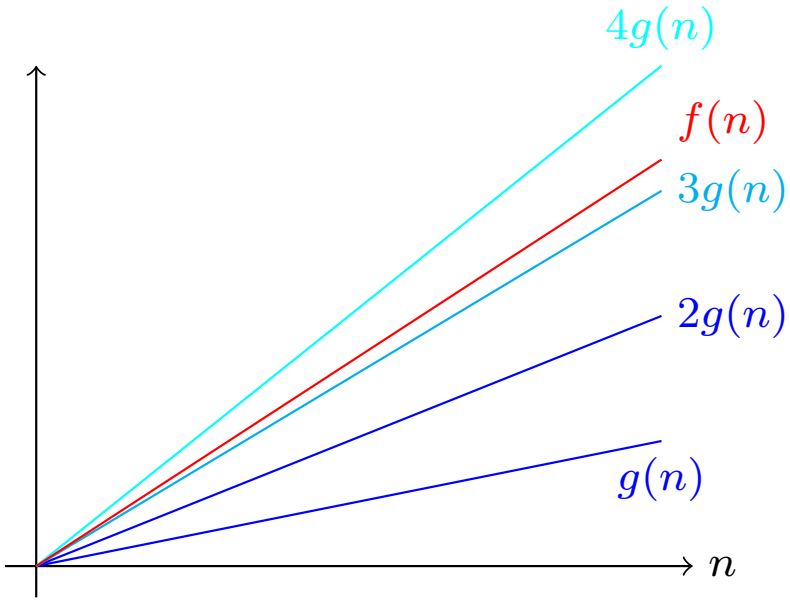
Ordo-notation

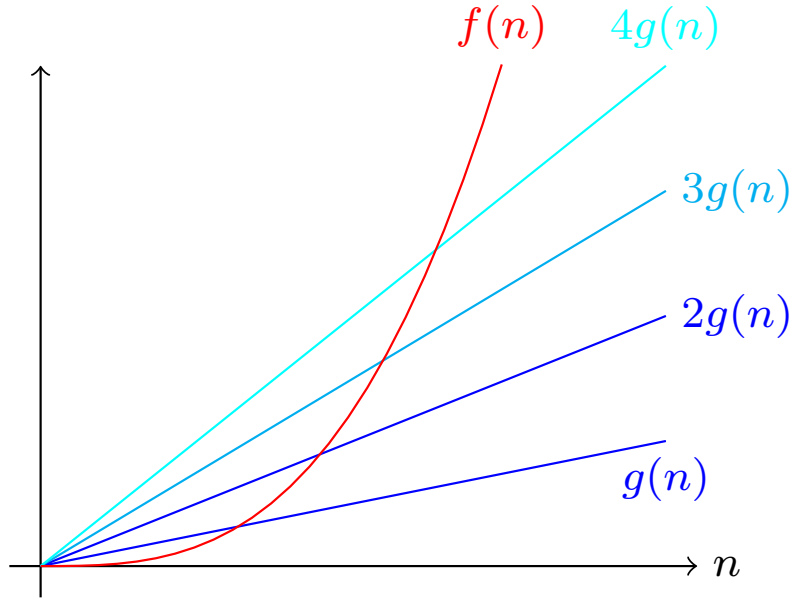
f, g: växande funktioner från $\mathbb{N}$ till $\mathbb{R}^+$

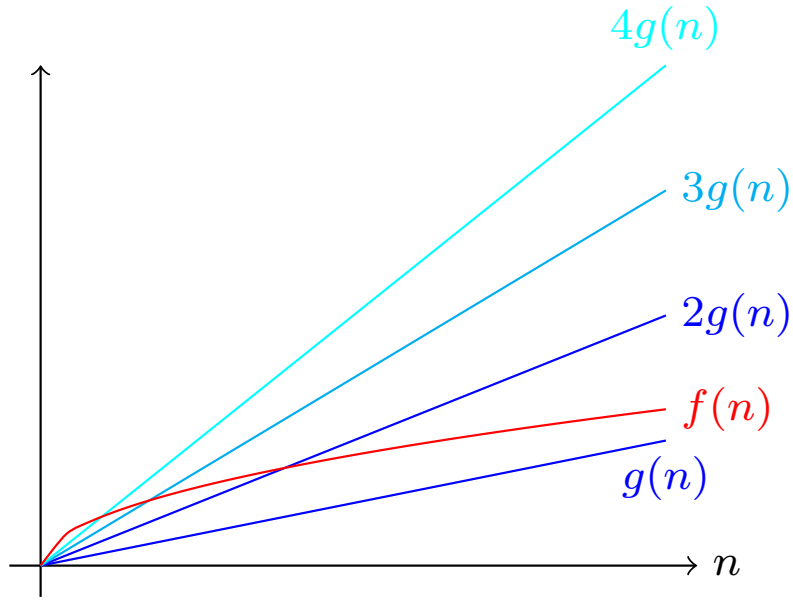
- $f \in O(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \leq c \cdot g(n)$ för alla $n \geq n_0$
  Intuition: Bortsett från konstanta faktorer växer f inte snabbare än g
- $f \in \Omega(g)$ omm det existerar $c > 0, n_0 > 0$ sådana att $f(n) \geq c \cdot g(n)$ för alla $n \geq n_0$
  Intuition: Bortsett från konstanta faktorer växer f minst lika fort som g
- $f(n) \in \Theta(g(n))$ omm $f(n) \in O(g(n))$ och $g(n) \in O(f(n))$
  Intuition: Bortsett från konstanta faktorer växer f och g lika snabbt

NOTERA: Ω är motsatsen till O, dvs $f \in \Omega(g)$ omm $g \in O(f)$.

Jämförelser av tillväxt







$\Omega(g), \Theta(g), O(g) \ldots ?$

"Regler" för ordo-notation

- Om $f(n)$ är ett polynom av grad d så gäller $f(n) \in O(n^d)$, dvs
  - Strunta i lägre ordningens termer
  - Strunta i konstanta faktorer
- Använd minsta möjliga klass av funktioner
  – säg $2n \in O(n)$ i stället för $2n \in O(n^2)$
- Använd enklast möjliga representant för klassen
  – säg $3n + 5 \in O(n)$ i stället för $3n + 5 \in O(3n)$

Analys av problem

Ringa in ett problems komplexitet!

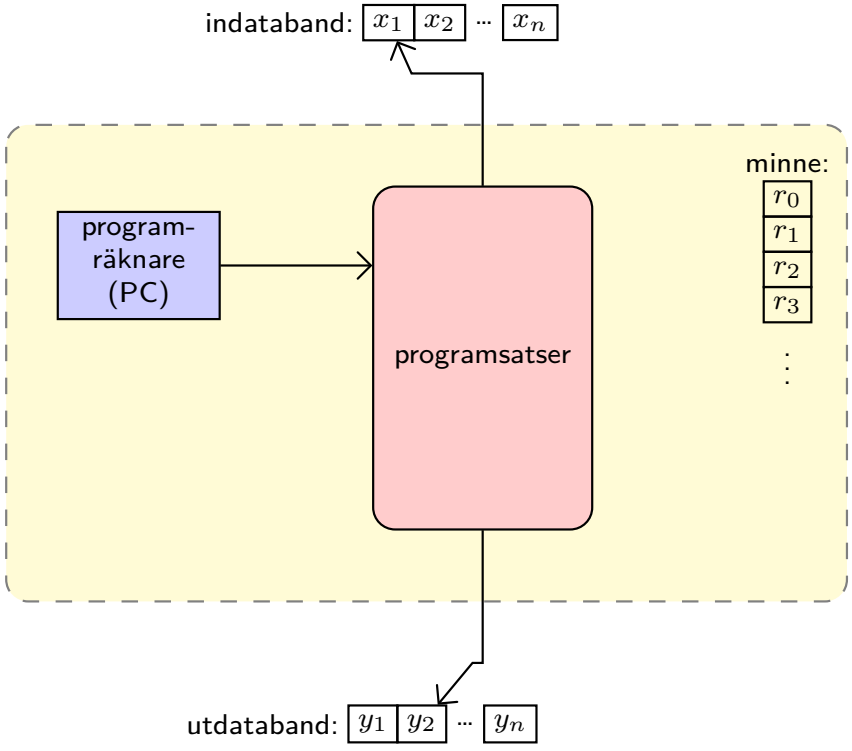
Övre gräns:

- Ge en algoritm som löser problemet.
- Algoritmens komplexitet är en övre gräns för problemets komplexitet.

Undre gräns:

- Ofta svårt att ange.
- Egenskaper hos problemet måste användas.
  - måste titta på alla indata $\Rightarrow \Omega(n)$
  - måste producera hela utdata
  - beslutsträd — ett visst antal olika fall måste särskiljas

RAM (Random Access Machine)



Programmet består av vanliga satser som utförs sekvensiellt (inte parallellt). Varje sats kan bara läsa och påverka ett konstant antal minnesplatser. Bara en symbol kan läsas/skrivas i taget. Varje sats tar konstant tid.

På grund av $O()$-notationens robusthet kan vi strunta i vilka värden konstanterna har.

Kostnadsmått

Enhetskostnad

- varje operation tar en tidsenhet
- varje variabel tar upp en minnesenhet

beräkningsmodell: RAM

Bitkostnad

- varje bitoperation tar en tidsenhet
- varje bit tar upp en minnesenhet

beräkningsmodeller: RAM med begränsad ordlängd, turingmaskin

Kostnadsmått

Enhetskostnad

- varje operation tar en tidsenhet
- varje variabel tar upp en minnesenhet

beräkningsmodell: RAM

Bitkostnad

- varje bitoperation tar en tidsenhet
- varje bit tar upp en minnesenhet

beräkningsmodeller: RAM med begränsad ordlängd, turingmaskin

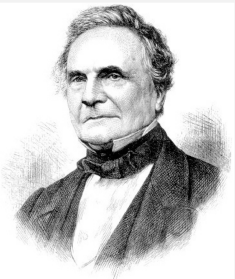
Exempel: Addition av två n-bitars heltal

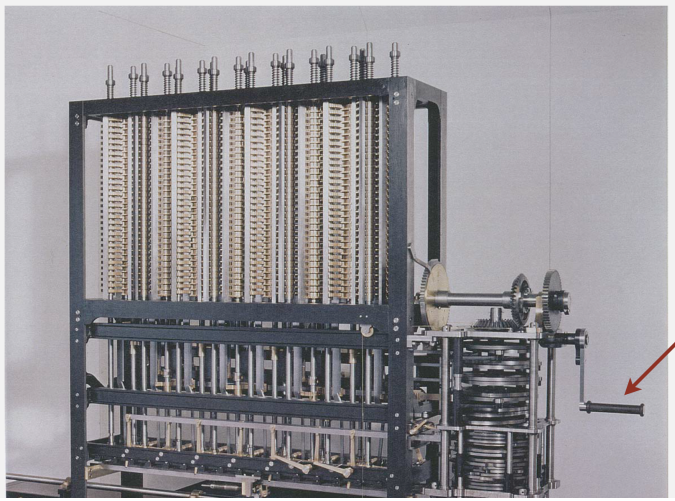
- tid $O(1)$ med enhetskostnad
- tid $O(n)$ med bitkostnad

LINKÖPING UNIVERSITY

Exekveringstid

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*





hur många varv behöver veven dras runt?

Intresserade aktörer

Programmeraren måste utveckla en fungerande lösning

Studenten kan spela en eller flera av dessa roller någon dag

Kunden vill lösa problemet effektivt

Teoretikern vill uppnå förståelse

Skäl till att analysera algoritmer

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

Skäl till att analysera algoritmer

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

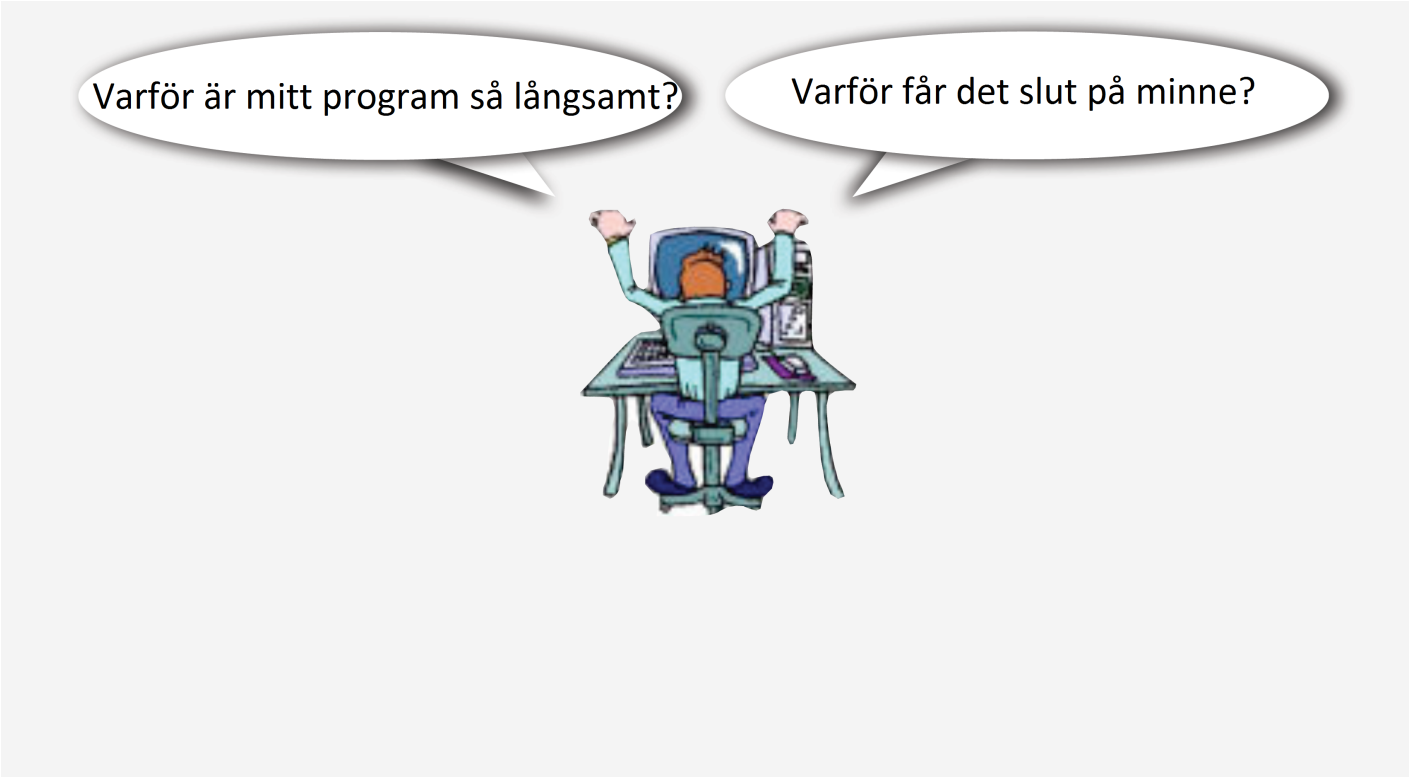
Primära praktiska skälet

Undvika buggar som påverkar prestanda

klienten får dålig prestanda på grund av att programmeraren inte förstår prestanda-karakteristiken hos sitt program

Utmaningen

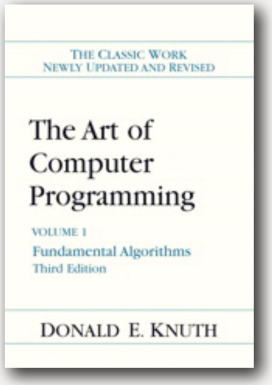
Fråga: Kommer mitt program att kunna hantera stora indata?

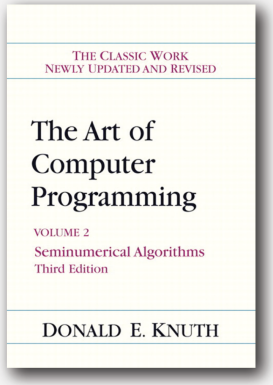


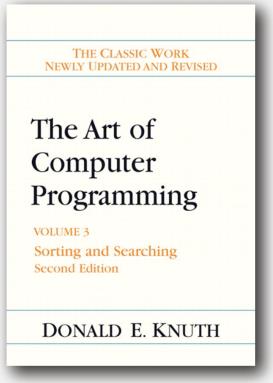
Matematiska modeller för exekveringstid

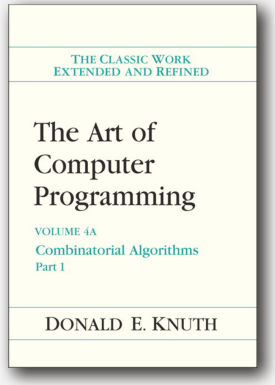
Total exekveringstid: summa av [kostnad $\times$ frekvens] för alla operationer

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









Donald Knuth
1974 Turing Award

I princip har vi tillgång till bra matematiska modeller

Matematiska modeller för exekveringstid

I princip har vi tillgång till bra matematiska modeller

I praktiken

- Formlerna kan bli komplicerade
- Avancerad matematik kan krävas
- Exakta modeller är bäst att lämna åt forskarna

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

frekvenser
(beror på algoritm, indata)

Slutsats: Vi använder approximativa modeller i kursen: $T(N) \in \mathcal{O}(f(N))$

Olika tekniker

Frågan gäller: tillväxttakten hos...

- minnesanvändning
- exekveringstid

Situationer att analysera:

- värsta fallet, bästa fallet, förväntade fallet,
- amorterat: en sekvens av anrop till algoritmen

Tekniker:

- algebraiskt (räkna iterationer)
- lös rekurrensrelationer (rekursiva algoritmer)

LINKÖPING UNIVERSITY

Algoritmanalys — hur då?

En algoritm bör (normalt) fungera för indata av godtycklig storlek.

Beskriv resursförbrukningen (tid/minne) som en *icke avtagande funktion* av *indatastorlek*.

Fokusera på beteendet i *värsta fallet*!

Ignorera *konstanta faktorer*

- analysen bör vara maskinoberoende;
- kraftfullare CPU $\Rightarrow$ uppsnabbning med konstant faktor.

Studera *skalbarhet/asymptotiskt beteende* för stora problemstorlekar:
ignorera lägre ordningens termer, fokusera på dominerande termer.

Exekveringstid för iterativa algoritmer

- Elementära operationer: begränsade av konstant tid
  - tilldela ett värde till en variabel
  - anropa en funktion/metod/procedur
  - utföra en aritmetisk operation
  - jämföra två tal
  - indexera i en array
  - följa en objektreferens
  - returnera från en funktion/metod/procedur
- Sekvenser av operationer: summan av komponenterna
- Loop (for...och while...):
  tiden (i värsta fallet) av villkoret plus kroppen gånger antalet iterationer (i värsta fallet) N:
  $t_{while} = N \cdot (t_{cond} + t_{body})$
- Villkorssats (if...then...else):
  tiden (i värsta fallet) för att evaluera villkoret plus den maximala tiden (i värsta fallet) av de två grenarna
  $t_{if} = t_{cond} + \max(t_{then}, t_{else})$

Algoritmanalys — hur i praktiken?

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

- Hur ser värstafallsinstansen ut?
- Hur mycket tid går åt i värsta fallet?
- Vad är “tidskomplexiteten” för den här funktionen?

Exempel: Två loopar

```
1: function FIND(A[1,...,n], B[1,...,n], t)
2:     for i ← 1 to n do
3:         if A[i] == t then
4:             return true
5:     for i ← 1 to n do
6:         if B[i] == t then
7:             return true
8:     return false
```

- Hur ser värstafallsinstansen ut?
- Hur mycket tid går åt i värsta fallet?
- Vad är "tidskomplexiteten" för den här funktionen?

Exempel: Två nästlade loopar

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

- Hur ser värstafallsinstansen ut?
- Hur mycket tid går åt i värsta fallet?
- Vad är "tidskomplexiteten" för den här funktionen?

Exempel: Två andra nästlade loopar

```
1: function DUPLICATE(A[1, ..., n])
2:     for i ← 1 to n do
3:         for j ← i + 1 to n do
4:             if A[i] == A[j] then
5:                 return true
6:     return false
```

- Hur ser värstafallsinstansen ut?
- Hur mycket tid går åt i värsta fallet?
- Vad är "tidskomplexiteten" för den här funktionen?

Exempel: Binärsökning

- Trivialt att implementera?

Exempel: Binärsökning

- Trivialt att implementera?
  - Första algoritmen publicerad 1946; första buggfria publicerad 1962.
  - Java-bugg i `Arrays.binarysearch()` upptäckt 2006.

Exempel: Binärsökning

- Trivialt att implementera?
  - Första algoritmen publicerad 1946; första buggfria publicerad 1962.
  - Java-bugg i `Arrays.binarysearch()` upptäckt 2006.

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
   }
   return -1;
}
```

Exempel: Binärsökning

- Trivialt att implementera?
  - Första algoritmen publicerad 1946; första buggfria publicerad 1962.
  - Java-bugg i `Arrays.binarysearch()` upptäckt 2006.

```
public static int binarySearch(int[] a, int key)
{
    int lo = 0, hi = a.length-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo) / 2;
        if      (key < a[mid]) hi = mid - 1;
        else if (key > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

Analys

Värstafallstid: $c_1 + maxit \cdot c_2 + c_3$, där maxit är maximalt antal iterationer av while-loopen.

LINKÖPING UNIVERSITY

Medelfallsanalys

Betrakta TableSearch: sekvensiell sökning i en tabell

- Indata: ett av tabellelementen,
- antag att det är valt med *likformig sannolikhet* över alla element.

function TableSearch($T[0,\ldots,n-1]$,K)
  for $i \leftarrow 0$ **to** $n-1$ **do**
    if $T[i] = K$ **then return** i
      $\vdots$

Medelfallsanalys

Betrakta TableSearch: sekvensiell sökning i en tabell

- Indata: ett av tabellelementen,
- antag att det är valt med *likformig sannolikhet* över alla element.

function TableSearch($T[0,\ldots,n-1]$,K)
  for $i \leftarrow 0$ **to** $n-1$ **do**
    if $T[i] = K$ **then return** i
      $\vdots$

Förväntad söktid

$$\frac{t_c + 2t_c + 3t_c + \ldots + nt_c}{n} = \frac{(1+2+3+\ldots+n)t_c}{n} =$$

$$= \frac{n(n+1)}{2n} t_c = \frac{n+1}{2} t_c \in O(n)$$

En utökningsbar array

Vi vill ha en ny typ av array som ökar sin storlek när den blivit full (antalet element insatta, n, är samma som kapaciteten, N). Antag att insättning hela tiden sker i arrayens första lediga position. Vi använder följande strategi:

- Allokera en ny array B med kapacitet $2N$
- Kopiera $A[i]$ till $B[i]$, för $i = 0, \ldots, N-1$
- Låt $A = B$, dvs vi låter B ta över rollen A har haft.

I termer av effektivitet kanske det här sättet att utöka arrayen verkar långsamt. Men, tiden för att lägga till ett nytt element blir

- $O(1)$ i de flesta fallen
- $O(n)$ för kopiering av n element och $O(1)$ för insättning när reallokering behövs.

Amorterad analys

Genom att använda amortering kan vi visa att körtiden för en sekvens av operationen att lägga till ett element till vår utökningsbara array faktiskt är ganska effektiv.

Proposition

Låt S vara en tabell implementerad med hjälp av en utökningsbar array A, som ovan. Den totala tiden för att sätta in n element i S, med start från en tom tabell S (vilket betyder att A har kapacitet $N = 1$) är $O(n)$.

Nästa gång:
Några grundläggande ADTer; stackar, köer och listor...

www.liu.se

LINKÖPING UNIVERSITY