

# Föreläsning 1

## Introduktion

TDDC91, TDDE22, 725G97: DALG

Utskriftsversion av föreläsning i *Datastrukturer och algoritmer* 3 september 2018

Magnus Nielsen, IDA, Linköpings universitet

1.1

### 1 Administrativ information

#### Lärare och personal

- Examinator: Erik Nilsson [erik.nilsson@liu.se](mailto:erik.nilsson@liu.se)
- Kursledare, föreläsningar: Magnus Nielsen [magnus.nielsen@liu.se](mailto:magnus.nielsen@liu.se)
- Kursassistent, lektioner, labbar: Kimberley French [kimberley.french@liu.se](mailto:kimberley.french@liu.se)
- Assistent, lektioner, labbar: Alojz Milicevic [alozj.milicevic@liu.se](mailto:alozj.milicevic@liu.se)
- Assistent, lektioner, labbar: Daniel Thorén [danth343@student.liu.se](mailto:danth343@student.liu.se)
- Assistent, lektioner, labbar: Gustav L'estrade [gustav.lestrade@liu.se](mailto:gustav.lestrade@liu.se)
- Assistent, lektioner, labbar: Henrik Fredriksson [henrik.fredriksson@liu.se](mailto:henrik.fredriksson@liu.se)
- Assistent, lektioner, labbar: Oskar Holmström [oskar.holmstrom@liu.se](mailto:oskar.holmstrom@liu.se)
- Kursadministratör: Jenny Lönn [jenny.lonn@liu.se](mailto:jenny.lonn@liu.se)

1.2

#### Litteratur

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

1.3

#### Examination

- Skriftlig tentamen (oktober, januari och augusti)
- Uppgifter i OpenDSA
- 4 + 1 (Gäller ej TDDC91) laborationsuppgifter
- Basgruppsarbete (Gäller endast TDDC91)

Laborationerna skall vara redovisade senast vid första tentamenstillfället. Rapportering av laborationer och OpenDSA kan bara garanteras i samband med ordinarie kurstillfälle.

1.4

### 1.1 Upplägg

#### 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
- Graftökning, topologisk sortering, kortaste väg
- Reserv

1.5

## Lektioner

- 3 handledda lektioner
- Uppgifter tillhandahålls vid lektionstillfället och finns senare på kurswebbsidan.

1.6

## Laborationer

- Hashning (Java)
- Binära sökträd (Java)
- Quick-sort (Java)
- Ordkedjor (Java)
  
- Var vänlig registrera er senast 10 sep i webReg och i Kattis

1.7

## Bonusproblem

- Finns på kurswebbsidan
- 0 lösta uppgifter ger 0 bonuspoäng.
- Lösta bonusuppgifter ger 1 bonuspoäng per uppgift, upp till max 4 (ca 10 % av totala tentabetyget), räknas bara mot högre betyg.

1.8

## Din insats i kursen

- Följ undervisningen (om du vill)
- Plugga under hela kursen (viktigt!)
- Gör uppgifterna i OpenDSA
- Gör datorlabb 1–4
- TDCC91: Vinjetter
- 725G97/TDDE22: Labb5
- Lös bonusproblem för att få bonuspoäng

## Kursens hemsida

<http://www.ida.liu.se/~TDCC91,TDDE22,725G97/>

1.9

## 2 DALG – introduktion

### 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

1.10

### Varför ska man plugga DALG?

#### Gammalt ursprung, nya möjligheter

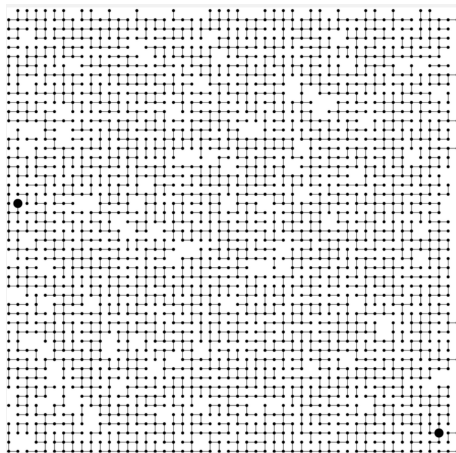
- Studiet av algoritmer har pågått åtminstone sedan Euklides
- Formaliserades av Church och Turing på 1930-talet
- Vissa viktiga algoritmer upptäcktes av studenter i liknande kurser som denna!

1.11

## Varför ska man plugga DALG?

### För att kunna lösa annars olösbara problem

- T.ex. nätverkskonnektivitet



1.12

## 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*



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



1.13

## Varför ska man plugga DALG?

### För att bli en kunnig programmerare

*“ 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*



1.14

## Varför ska man plugga DALG?

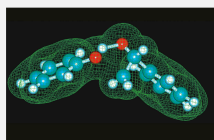
### För att DALG kan hjälpa oss att lista ut saker om livet och universum

“ 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)



Martin Karplus, Michael Levitt, and Arieh Warshel



Datorbaserade modeller ersätter matematiska modeller alltmer i dagens vetenskap

1.15

## DALG – grunder

### Abstrakta datatyp (ADT)

maskinberoende högnivåbeskrivning av data och operationer på data t.ex. Stack, Kö, Mängd, ...

### 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

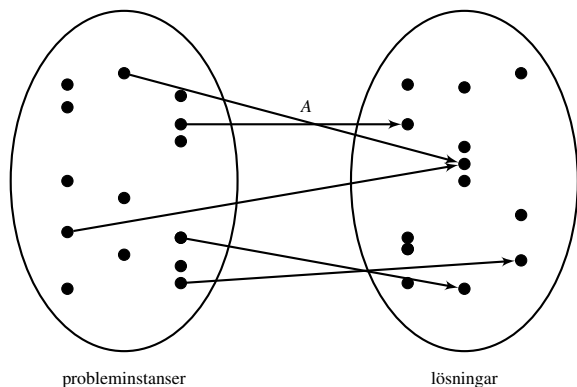
implementerar algoritmer och datastrukturer i något visst programspråk

1.16

**Definition 1** (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  $A : \text{PROBLEMINSTANSER} \rightarrow \text{LÖSNINGAR}$ .



1.17

## Algoritm

### 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 paradig

vanligt förekommande problemlösningstrategier, t.ex. rekursiv nedbrytning, dynamisk programmering, gigriga algoritmer, ...

### Pseudokod

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

1.18

## 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*?

1.19

## 2.1 Algoritmanalys

### Analys av algoritmer

#### Vad ska vi analysera?

- Korrekthet (inte 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

1.20

### Hur man kan jämföra effektivitet

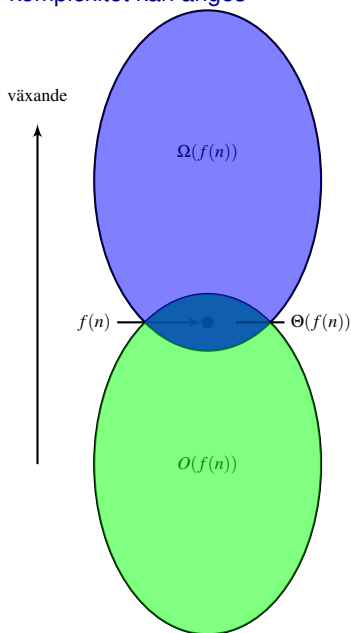
- Studera exekveringstid (eller minnesåtgång) som en funktion av storleken på indata.
- När är två algoritmer ”lika effektiva”?
- När är en algoritm bättre än en annan?

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

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

1.21

## 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  $f(n)$

1.22

## Ordo-notation

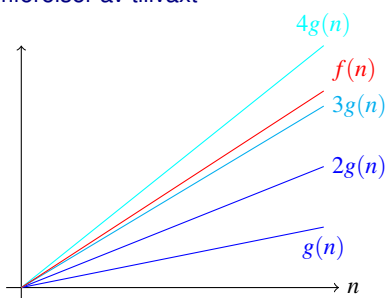
$f, g$ : växande funktioner från  $\mathbb{N}$  till  $\mathbb{R}^+$

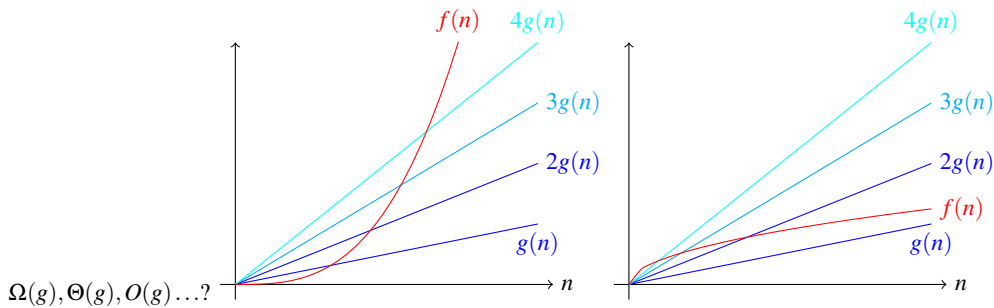
- $f \in O(g)$  omm det existerar  $c > 0, n_0 > 0$  sådana att  $f(n) \leq c \cdot g(n)$  för alla  $n \geq n_0$  Intuition: Bortsett från konstanta faktorer växer  $f$  inte snabbare än  $g$
- $f \in \Omega(g)$  omm det existerar  $c > 0, n_0 > 0$  sådana att  $f(n) \geq c \cdot g(n)$  för alla  $n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

1.23

## Jämförelser av tillväxt





1.24

### "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  $3n + 5 \in O(3n)$

1.25

### Ett sätt att jämföra tillväxt

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

$$L = \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$$

- $f \in O(g)$  om  $L < \infty$
- $f \in \Omega(g)$  om  $L > 0$
- $f \in \Theta(g)$  om  $0 < L < \infty$

1.26

## 2.2 Övre och undre gränser

### 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.

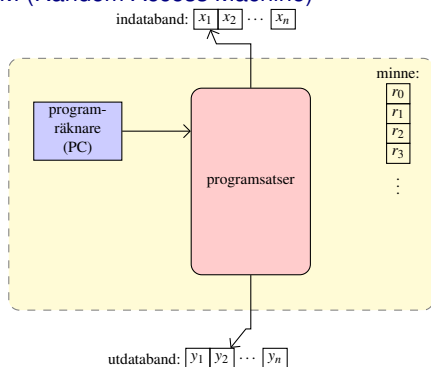
#### Undre gräns:

- 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

1.27

## 2.3 En beräkningsmodell

### 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 värdena konstanterna har.

1.28

## Kostnadsmaat

### 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  $n$ -bitsheltal*

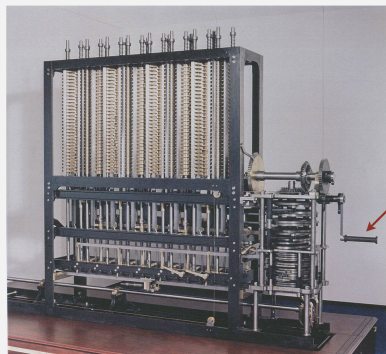
- tid  $O(1)$  med enhetskostnad
- tid  $O(n)$  med bitkostnad

1.29

## 3 Introduktion till algoritmanalys

### 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)*



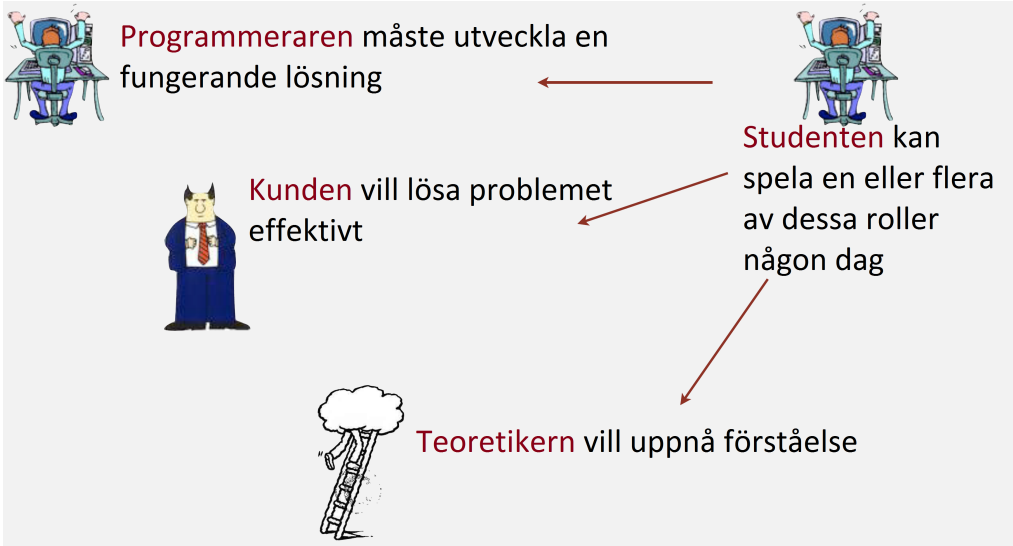
hur många varv behöver  
veven dras runt?

Analytic Engine

1.30

### Intrasserade aktörer





1.31

### 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



1.32

### Utmaningen

Fråga: Kommer mitt program att kunna hantera stora indata?



**Insikt:** [Knuth 1970-talet] Använd *vetenskaplig metod* för att förstå prestanda

1.33

### Matematiska modeller för exekveringstid

**Total exekveringstid: summa av [kostnad × frekvens] för alla operationer**

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



*I princip* har vi tillgång till bra matematiska modeller

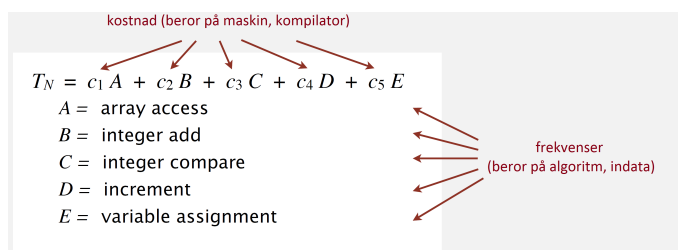
1.34

### 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



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

1.35

#### 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)

1.36

## 4 Analys av värsta fallet

### 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.

1.37

### 4.1 Iterativa algoritmer

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

1.38

### Algoritmanalys — hur i praktiken?

```
1: function FIND( $A[1, \dots, n], t$ )
2:   for  $i \leftarrow 1$  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?

1.39

### Exempel: Två loopar

```
1: function FIND( $A[1, \dots, n], B[1, \dots, n], t$ )
2:   for  $i \leftarrow 1$  to  $n$  do
3:     if  $A[i] == t$  then
4:       return true
5:   for  $i \leftarrow 1$  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?

1.40

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?
  - 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;
}

```

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

4.2 Vanliga tillväxttakter

Vanliga tillväxttakter

tillväxttakt	typisk kod	beskrivning	exempel	$T(2n)/T(n)$
1	a = b + c	sats	lägga ihop två tal	1
$\log_2 n$	while (n > 1) { n = n / 2; ... }	dela på hälften	binärsökning	$\approx 1$
n	for (int i = 0; i < n, i++) { ... }	loop	hitta maximum	2
$n \log_2 n$	se föreläsning om mergesort	divide and conquer	mergesort	$\approx 2$
$n^2$	for (int i = 0; i < n, i++) for (int j = 0; j < n, j++) { ... }	dubbel loop	kolla alla par	4
$n^3$	for (int i = 0; i < n, i++) for (int j = 0; j < n, j++) for (int k = 0; k < n, k++) { ... }	trippel-loop	kolla alla tripplar	8
$2^n$	se sista föreläsningen	total-sökning	kolla alla delmängder	$T(n)$

## 5 Analys av medelfallet

### 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( $T[0, \dots, n-1], K$ )
```

```
  for  $i \leftarrow 0$  to  $n-1$  do
    if  $T[i] = K$  then return  $i$ 
  ⋮
```

Förväntad söktid

$$\begin{aligned} \frac{t_c + 2t_c + 3t_c + \dots + nt_c}{n} &= \frac{(1+2+3+\dots+n)t_c}{n} = \\ &= \frac{n(n+1)}{2n} t_c = \frac{n+1}{2} t_c \in O(n) \end{aligned}$$

1.45

### 5.1 Amorterad analys

#### 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:

- Allokerar en ny array  $B$  med kapacitet  $2N$
- Kopiera  $A[i]$  till  $B[i]$ , för  $i = 0, \dots, N-1$
- 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.

1.46

#### 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 2.** 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  $O(n)$ .

1.47