TDDI16 – Föreläsning 1

Introduktion och komplexitet

Filip Strömbäck

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Resurser

- Kurshemsida: `https://www.ida.liu.se/~TDDI16/`
- Litteratur: OpenDSA, (Introduction to Algorithms)

Examinator	Erik Nilsson
Kursledare	Filip Strömbäck
Kursassistent	Magnus Nielsen
Assistenter	David Hasselquist
	Mladen Nikic
Administratör	Madeleine Häger Dahlqvist

Examination

UPG1 Uppgifter i OpenDSA, 2hp (U, G)

LAB1 Laborationer, 2hp (U, G)
4 laborationsuppgifter

DAT1 Datortentamen, 2hp (U, 3, 4, 5)
Två delar: OpenDSA-frågor (ca 30%), samt frågor om användandet av datastrukturer och algoritmer.
Extrauppgifter ger upp till 10% bonus mot högre betyg.

OpenDSA – Digital kursbok

- Digital kursbok med interaktiva övningar
- Skapa ett konto – använd **LiU-id** som användarnamn
- För att klara UPG1 ska ni innan kursens slut ha löst **alla** interaktiva övningar
- Avklarade kapitel markeras med en bock
- Klicka på ert namn för att kontrollera vad som är kvar

OpenDSA fungerar bäst i Chrome eller Chromium

Föreläsningar

- Fokus på hur info från OpenDSA kan *användas*
- Efter varje föreläsning finns 2 extrauppgifter i KATTIS
  - Relaterade till det som tagits upp
  - Ett enklare och ett svårare
  - Löses individuellt
- Ställ hemskt gärna frågor!

Insiktsnivå

- Jag ställer inga frågor
  - Jag har inte förstått
- Jag ställer grundläggande frågor
  - Jag har förstått att jag inte kan och vill lära mig
- Jag ställer kontrollfrågor
  - Jag har börjat förstå och vill bekräfta det jag kan
- Jag ställer djupa frågor
  - Jag har förstått och vill lära mig mer

Laborationer

- Arbete sker parvis
- Tre klasser efter program:
  - A: DI2
  - B: DI2
  - C: IP2
- Anmälan sker i Webreg (länk finns på kurshemsidan)
- Anmälan öppen till och med 5/9
- Innehåll:
  1. AVL-träd
  2. Knäcka lösenord
  3. Ordkedjor
  4. Mönsterigenkänning

Planering

Vecka	Fö	Lab
36	Komplexitet, Linjära strukturer	----
37	Träd, AVL-träd	1---
38	Hashning, meet-in-the-middle	12--
39	Grafer, graftraversering	-23-
40	Grafer, kortaste vägen	-23-
41	Sortering	--34
42	Repetition	---4

LINKÖPING UNIVERSITY

Liknelse

Du vill gräva en stor grop.

- Utan verktyg: 2 dagar
- Med spade: 5 timmar
- Med grävskopa: 1 timme
- ...

Om du har tillgång till dynamit kan du dessutom lösa ett svårare problem: att gräva en ”grop” i en berghäll.

I programmering...

Du vill lösa ett svårt problem.

- Utan DALG-kunskap: 1 månad
- Kan använda datastrukturer: 1 vecka
- Känner till lämpliga algoritmer: 1 dag

Om du dessutom vet hur verktygen fungerar, kan du dessutom anpassa dem så att du kan lösa mer komplicerade problem, och så att lösningen blir mer effektiv.

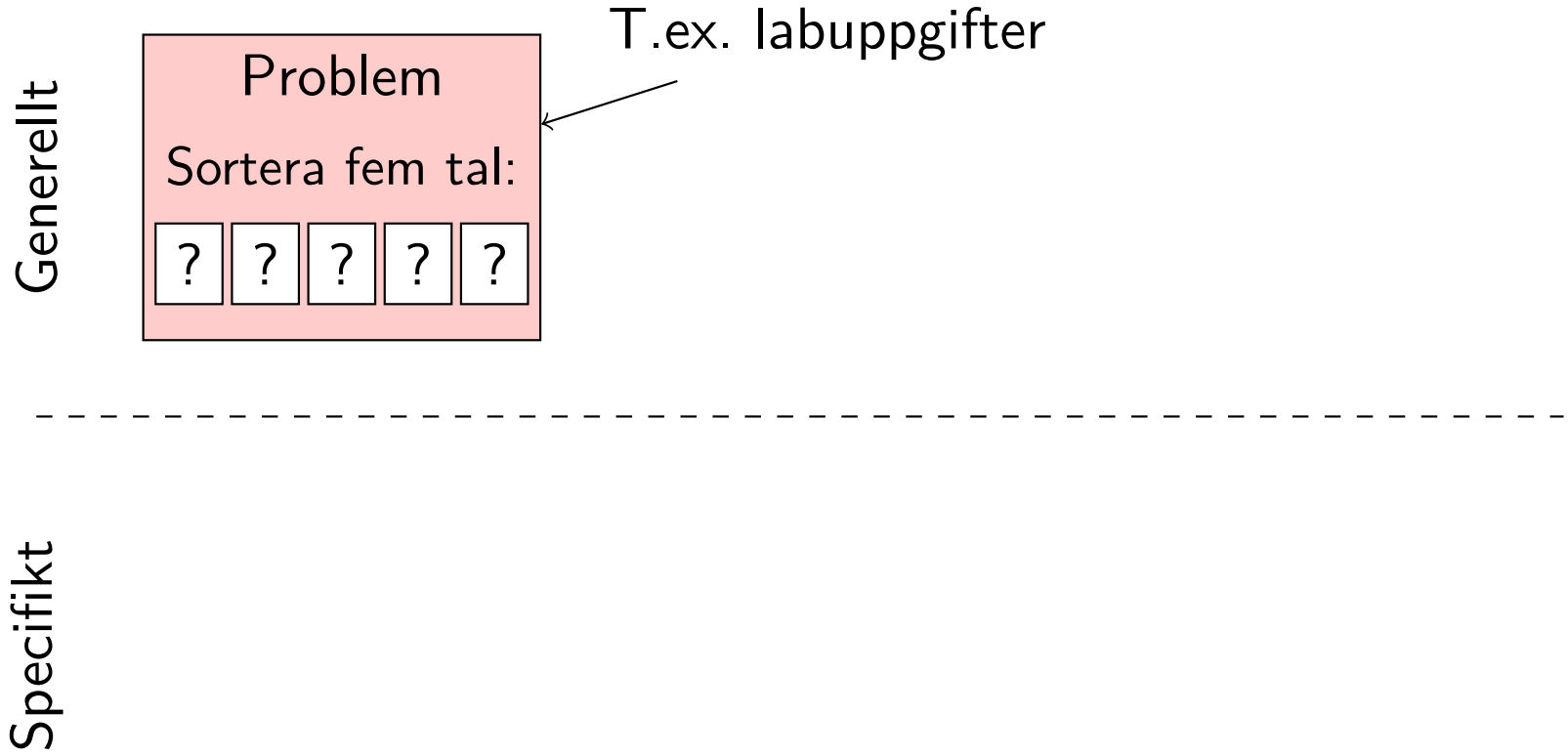
Varför DALG?

- *Veta* vilka verktyg som finns
- *Kunna använda* verktygen som finns tillgängliga
  - ...för att kunna *implementera* lösningar smidigt
  - ...för att kunna *uttrycka* sig bättre
  - ...för att kunna *resonera* på en högre abstraktionsnivå
- *Kunna välja* rätt verktyg
  - *Kunna analysera* och *värdera* olika lösningar
- *Kunna anpassa* standardalgoritmer- eller datastrukturer så att de kan lösa ditt specifika problem.

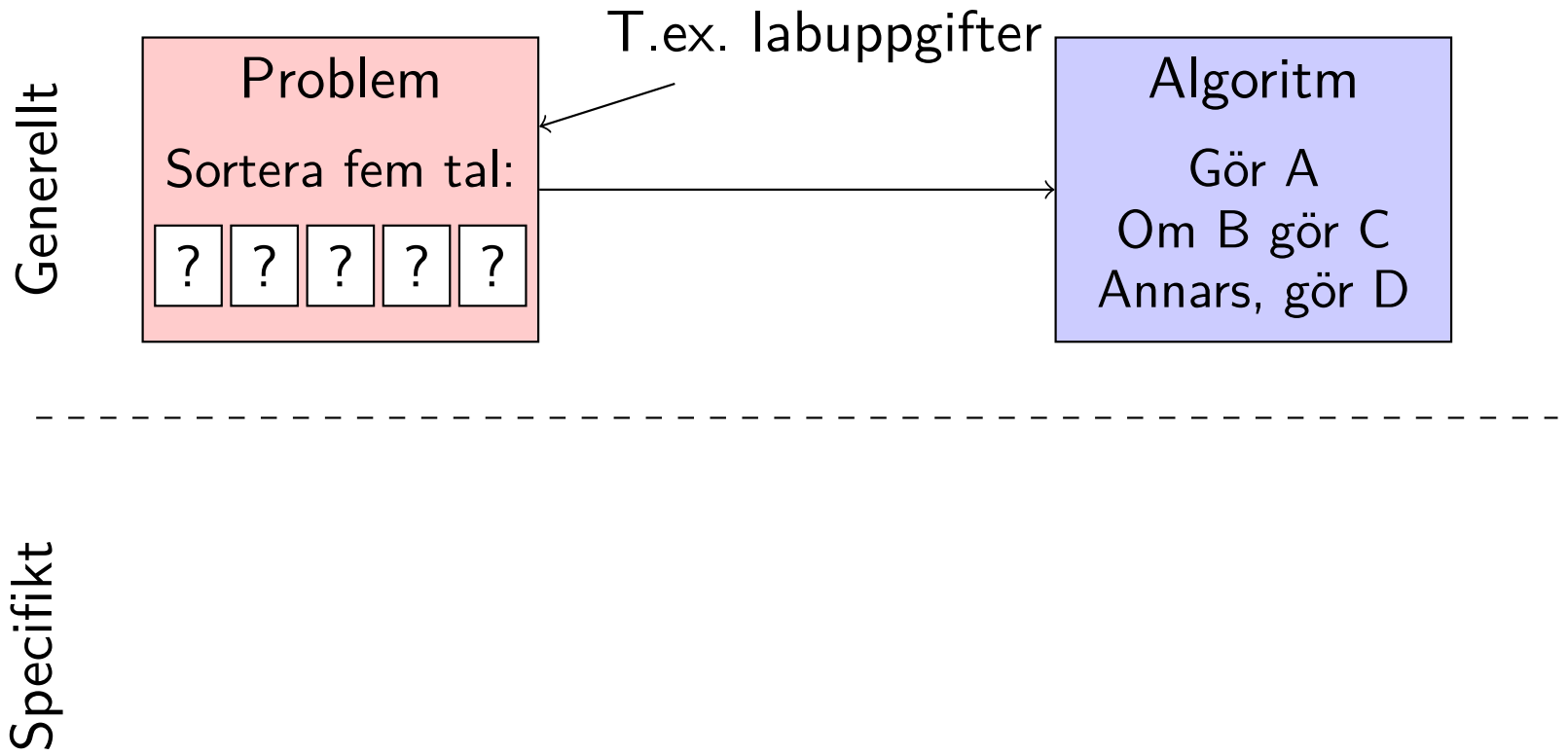
För att effektivt kunna lösa svåra problem, eller problem med stora mängder data.

LINKÖPING UNIVERSITY

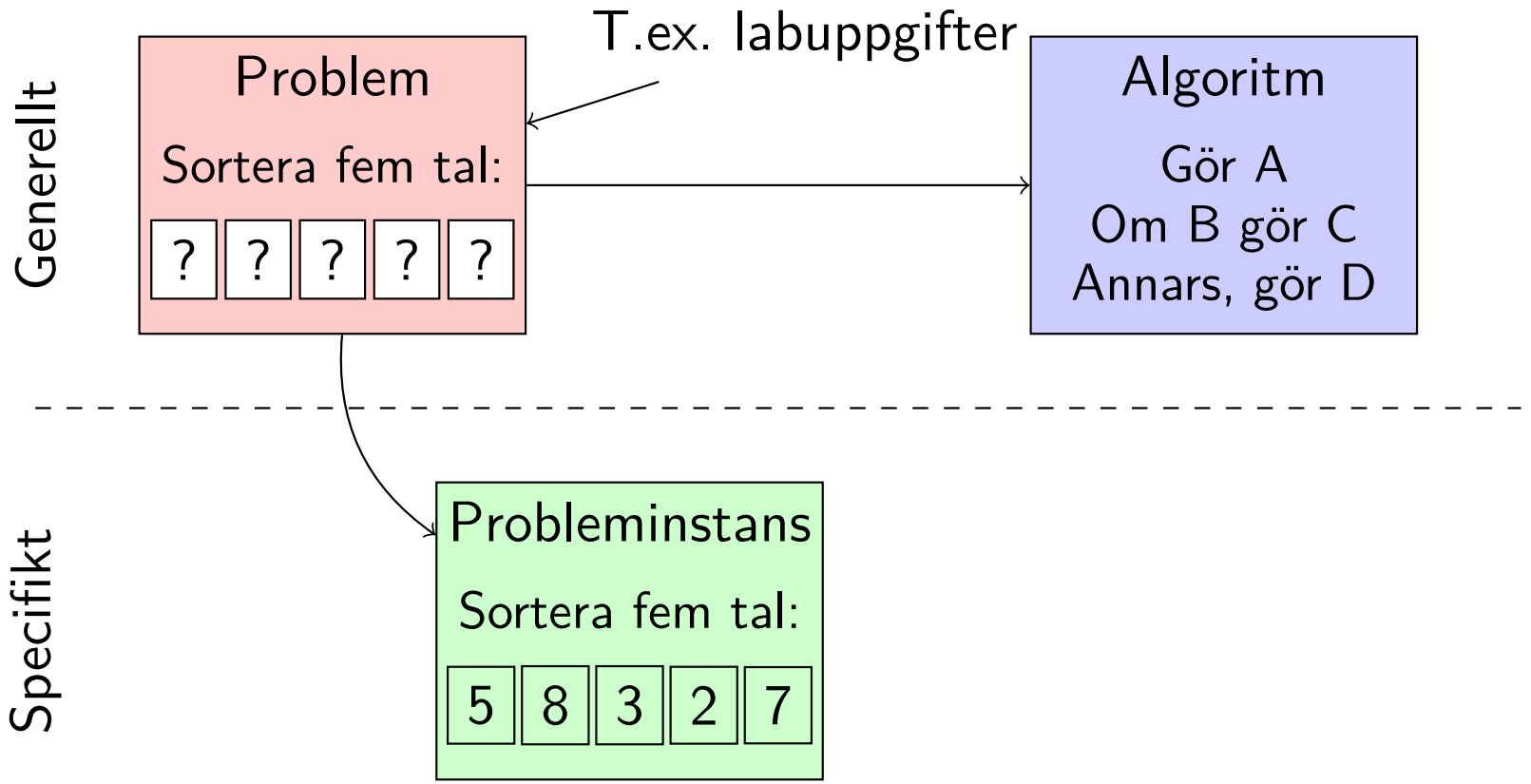
Problem, program och algoritmer



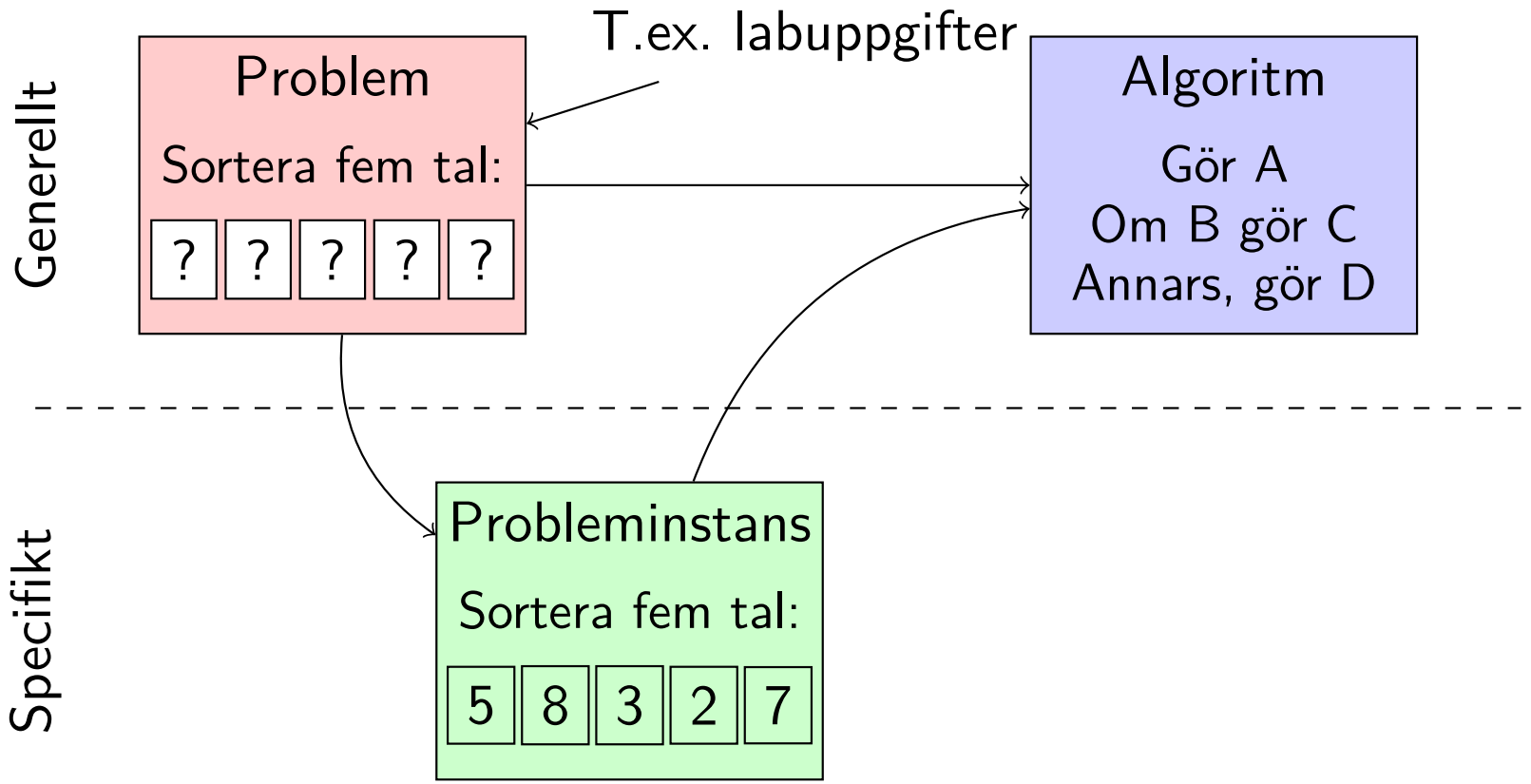
Problem, program och algoritmer



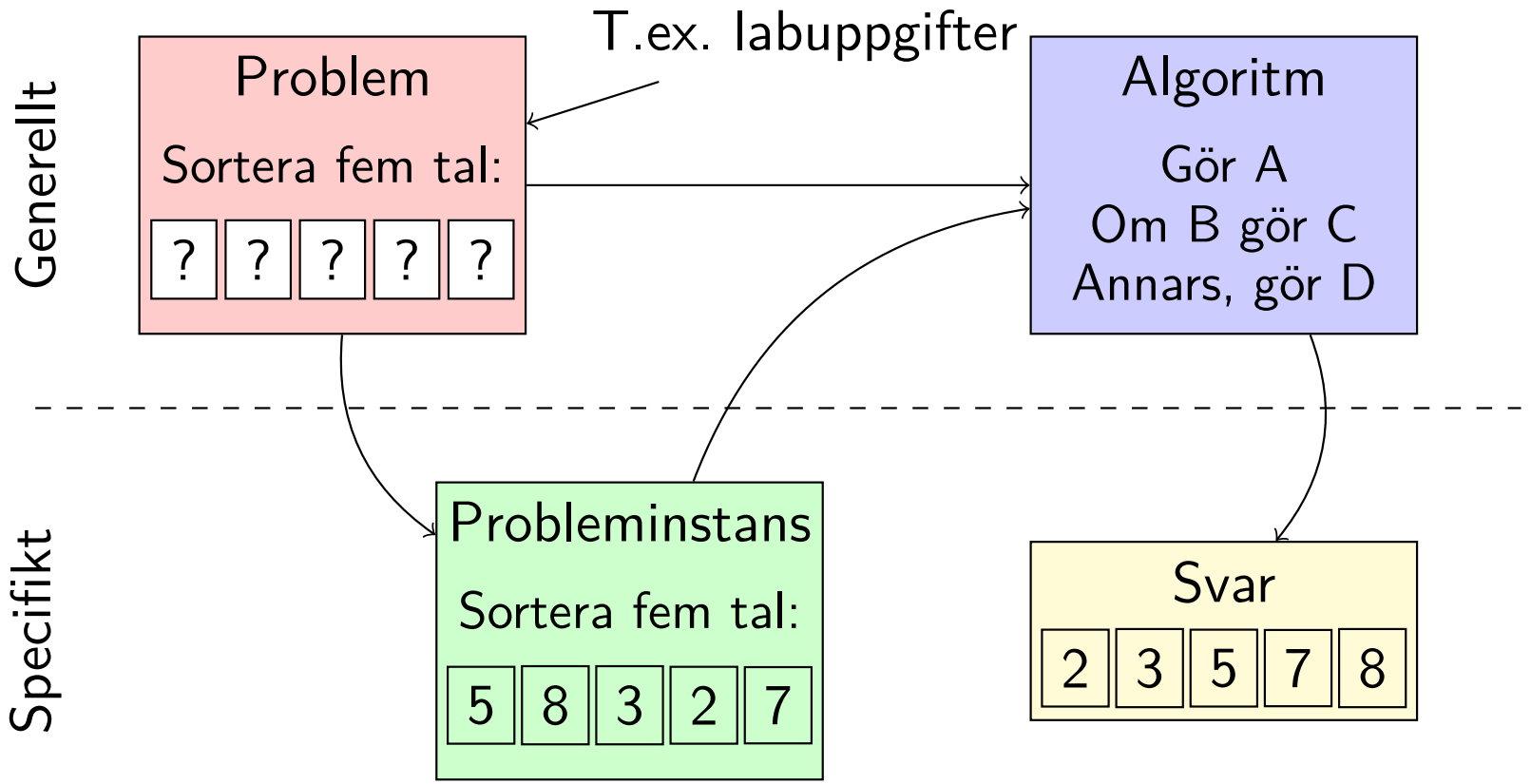
Problem, program och algoritmer



Problem, program och algoritmer



Problem, program och algoritmer



Algoritmanalys?

Det finns oftast flera olika algoritmer som löser ett givet problem. Vilken ska vi välja?

Algoritmanalys?

Det finns oftast flera olika algoritmer som löser ett givet problem. Vilken ska vi välja?

- Den som är snabbast (den som alltid terminerar)
- Den som använder minst minne
- Den som kan köras parallellt
- Den som gör minst antal frågor till externa tjänster
- ...

Algoritmanalys!

Exempel: Algoritm som beräknar fibonaccital

1, 1, 2, 3, 5, 8, 13, 21, ...

$$f(n) = \begin{cases} 1, & \text{om } n = 1 \\ 1, & \text{om } n = 2 \\ f(n-1) + f(n-2) & \text{annars} \end{cases}$$

Vilken implementation är snabbast?

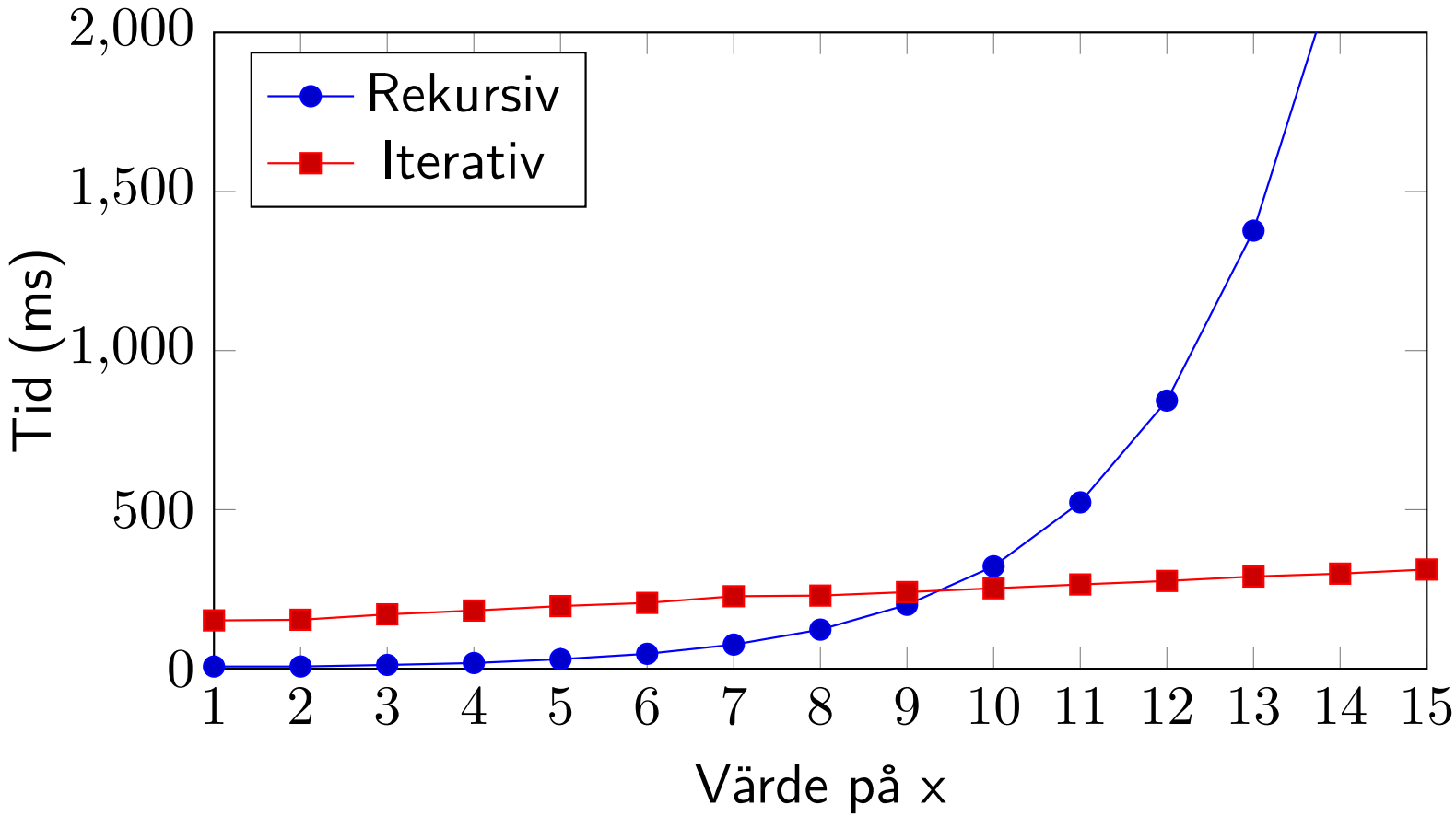
```
int fib1(int n) {
  if (n == 1 || n == 2) {
    return 1;
  } else {
    return fib1(n - 1)
      + fib1(n - 2);
  }
}
```

```
int fib2(int n) {
  vector<int> res(n+1, 1);

  for (int i=3; i<=n; i++) {
    res[i] = res[i - 1]
        + res[i - 2];
  }

  return res[n];
}
```

Vi mäter! 1 000 000 körningar per indata



Vad är intressant?

Tiden för **små** indata är oftast väldigt liten, knappt mätbar. Alltså:

- Vi är intresserade av vad som händer för **stora** indata
- Vi vill kunna jämföra olika algoritmer
- Vi är intresserade av **helheten**

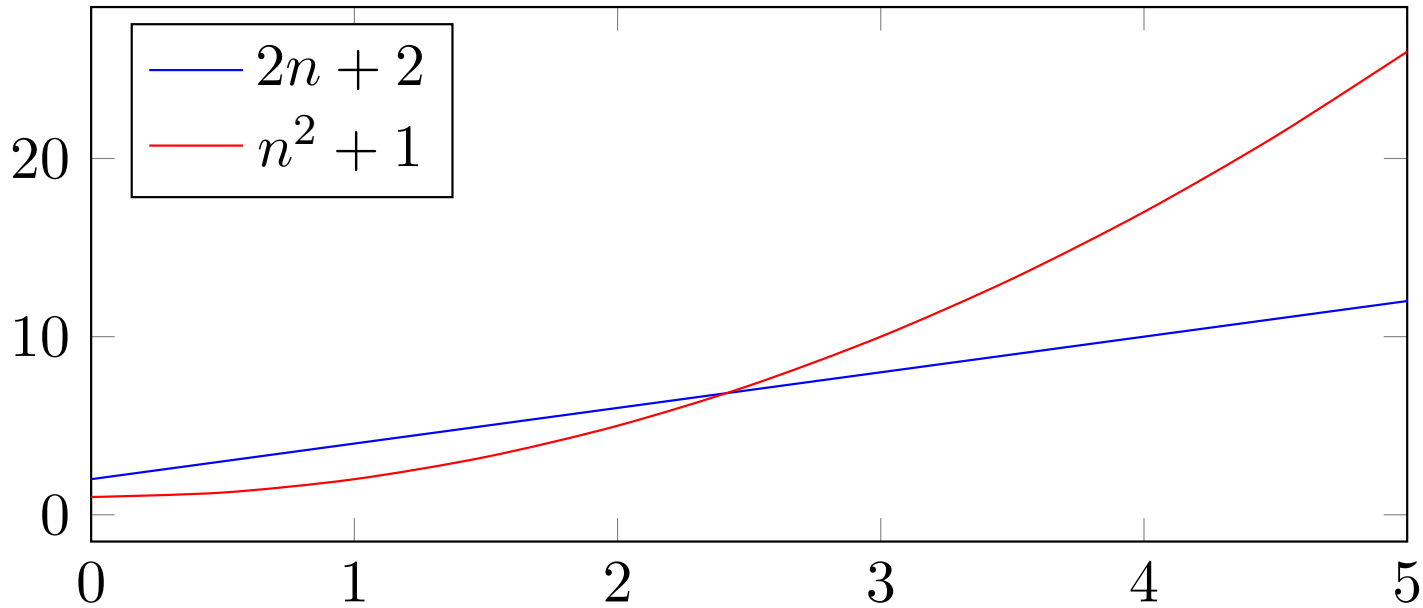
Vi behöver ett sätt att resonera om detta!

LINKÖPING UNIVERSITY

Idé

- Vi delar in funktioner i olika grupper, där varje grupp växer ungefär lika snabbt för stora n.
- För att veta vilka grupperna är behöver vi kunna jämföra funktioner. Vi säger att $f(n) \in \mathcal{O}(g(n))$ omm det finns några $0 \leq c < \infty$ och $0 \leq n_0 < \infty$ så att $f(n) \leq cg(n)$ för alla $n \geq n_0$.
- Detta innebär att $f(n)$ inte växer snabbare än $g(n)$. Man kan tänka sig att $f(n) \leq g(n)$ gäller för tillräckligt stora n (även om det inte riktigt stämmer).

Exempel

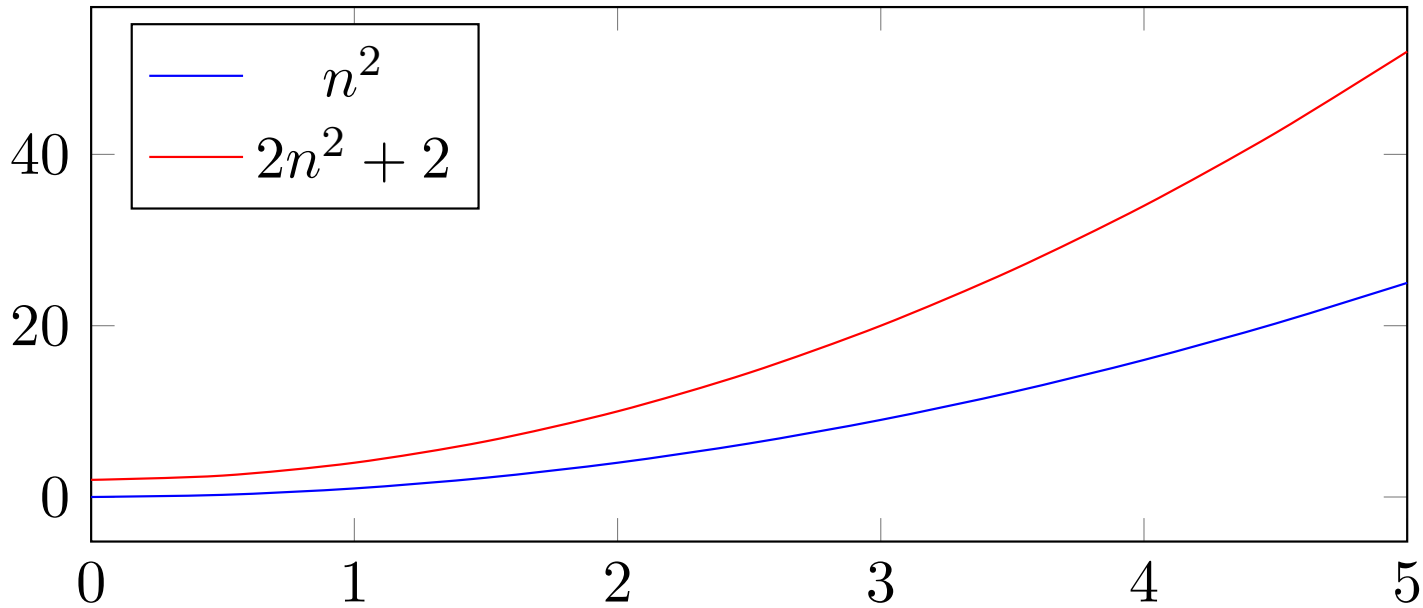


$2n + 2 \leq c(2n + 2)$ för $c = 1$ och $n \geq 3$

Alltså är $2n + 2 \in \mathcal{O}(n^2 + 1)$

Dock är $n^2 + 1 \notin \mathcal{O}(2n + 2)$

Exempel

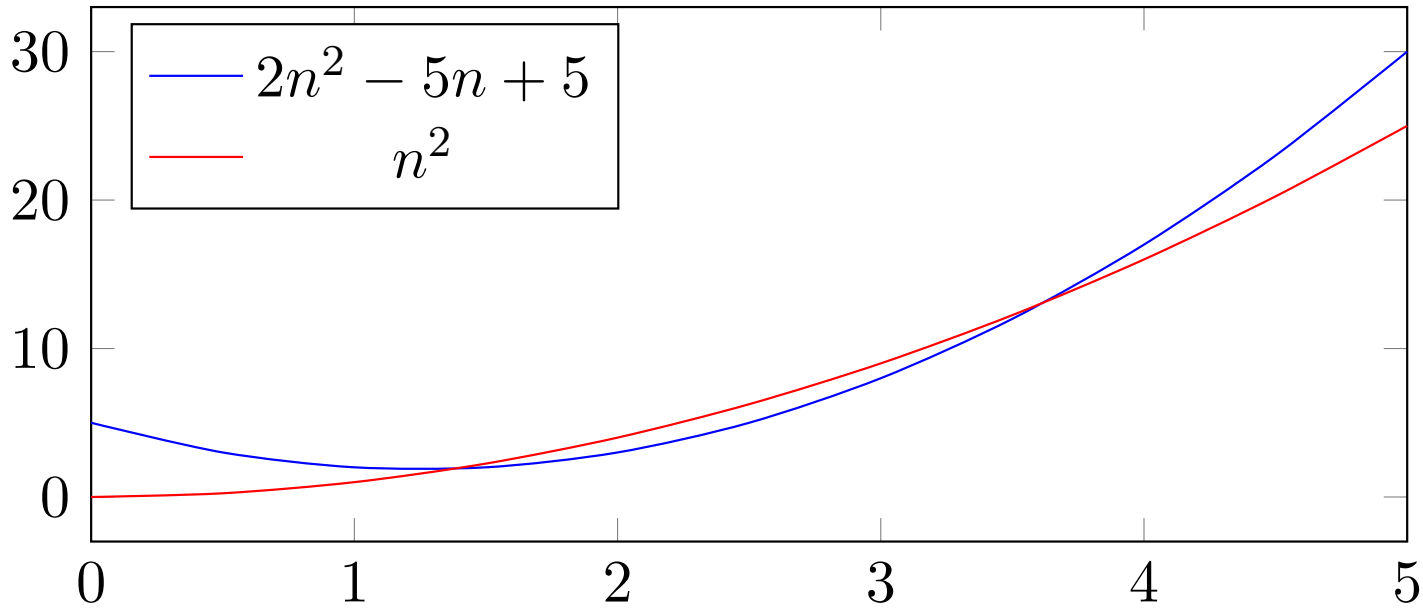


$n^2 \in \mathcal{O}(2n^2 + 2)$ (enkelt att se)

$2n^2 + 2 \in \mathcal{O}(n^2)$ gäller också. Hur?

Alltså: $\mathcal{O}(n^2) = \mathcal{O}(2n^2 + 2)$

Exempel



$n^2 \in \mathcal{O}(2n^2 - 5n + 5)$ (enkelt att se)

$2n^2 - 5n + 5 \in \mathcal{O}(n^2)$ gäller också. Hur?

Alltså: $\mathcal{O}(2n^2 - 5n + 5) = \mathcal{O}(n^2)$

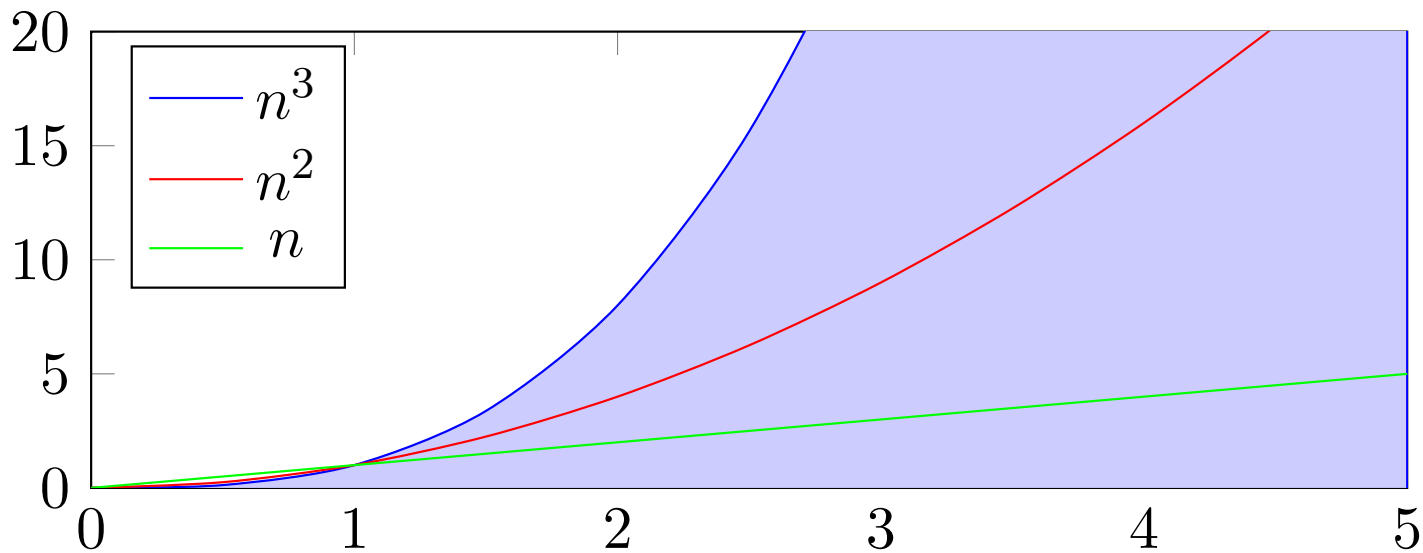
Observationer – Regler

- I en summa av termer kan vi förenkla bort alla termer förutom den snabbast växande termen
  Ex: $n^2 + n + 1 \in \mathcal{O}(n^2)$
- Konstanter, både konstanta termer (ex. $n + 5$) och konstanter framför termer (ex. $5n$) kan förenklas bort
- Om $f(n) \in \mathcal{O}(g(n))$, och $g(n) \in \mathcal{O}(f(n))$ så är $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$.
- Alltså kan vi representera våra olika ”grupper” i form av den mest förenklade formeln

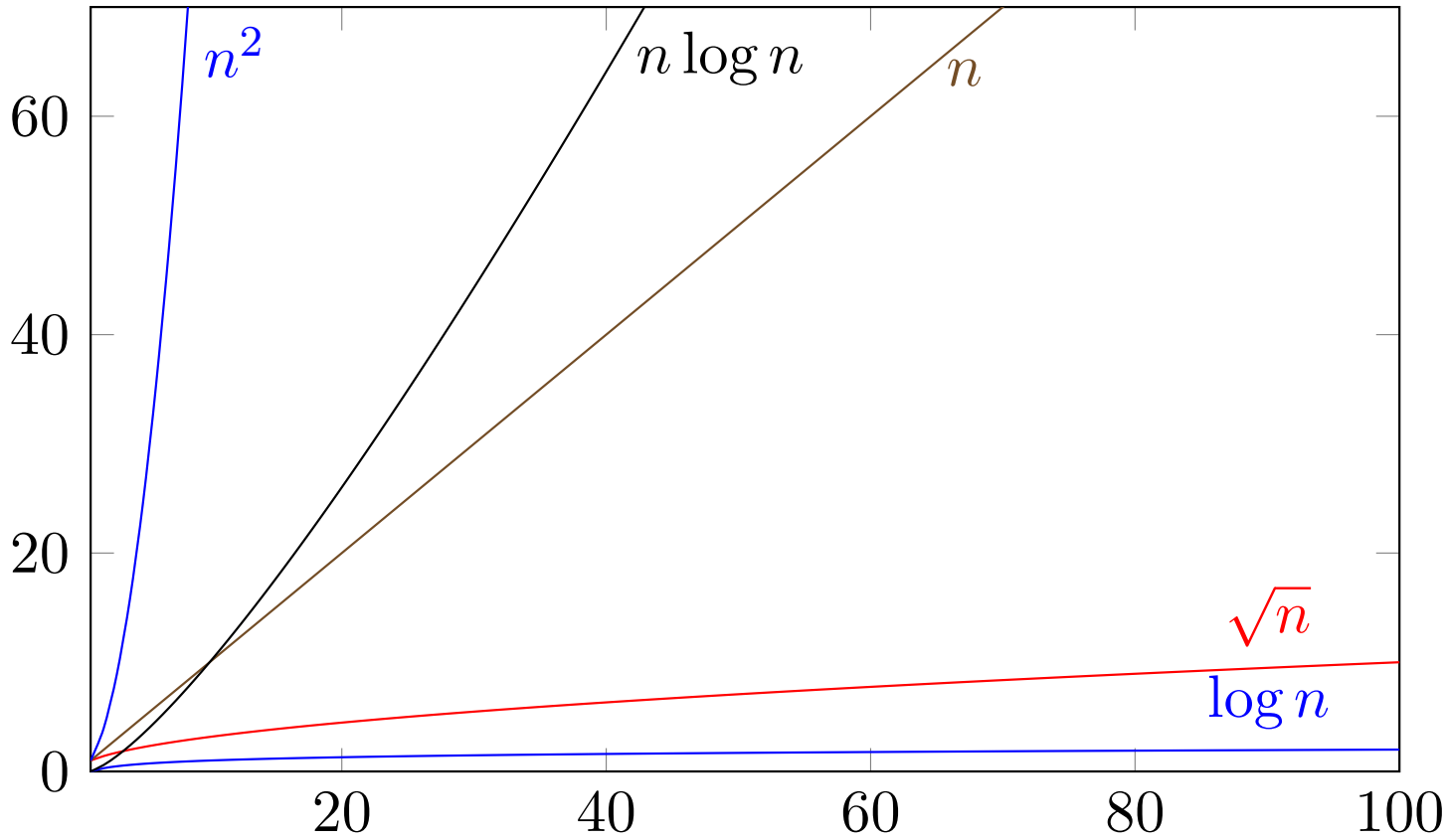
Förenkling med regler

Med hjälp av observationerna kan vi enklare få en uppfattning om förhållandet mellan olika tillväxtfunktioner.

Här kan vi se att $n, n^2 \in \mathcal{O}(n^3)$:



Vanliga uttryck för tidskomplexitet



LINKÖPING UNIVERSITY

Idé

- Vi vill ”mäta” tiden det tar att köra en algoritm
- Vi såg att konstanter i uttryck inte spelar någon roll

$\Rightarrow$ Den exakta körtiden spelar ingen ingen roll, vi är bara intresserade av **hur fort den växer**

$\Rightarrow$ Vi kan anta att varje operation tar 1 tidsenhet

Exempel 1

```
int a(int n) {
  cout << n << endl;
  cout << (n + 1) << endl;
  cout << (n + 2) << endl;
  return 0;
}
```

Exempel 2

```
int b(int n) {
  int result = 1;
  for (int i = 1; i < n; i++)
    result *= i;
  return result;
}
```

Exempel 3

```
int c(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++)
    sum += b(i);
  return sum;
}
```

Fibonacci sedan tidigare

```
int fib2(int n) {
  vector<int> res(n+1, 1);

  for (int i=3; i<=n; i++) {
    res[i] = res[i - 1]
        + res[i - 2];
  }

  return res[n];
}
```

Den rekursiva varianten

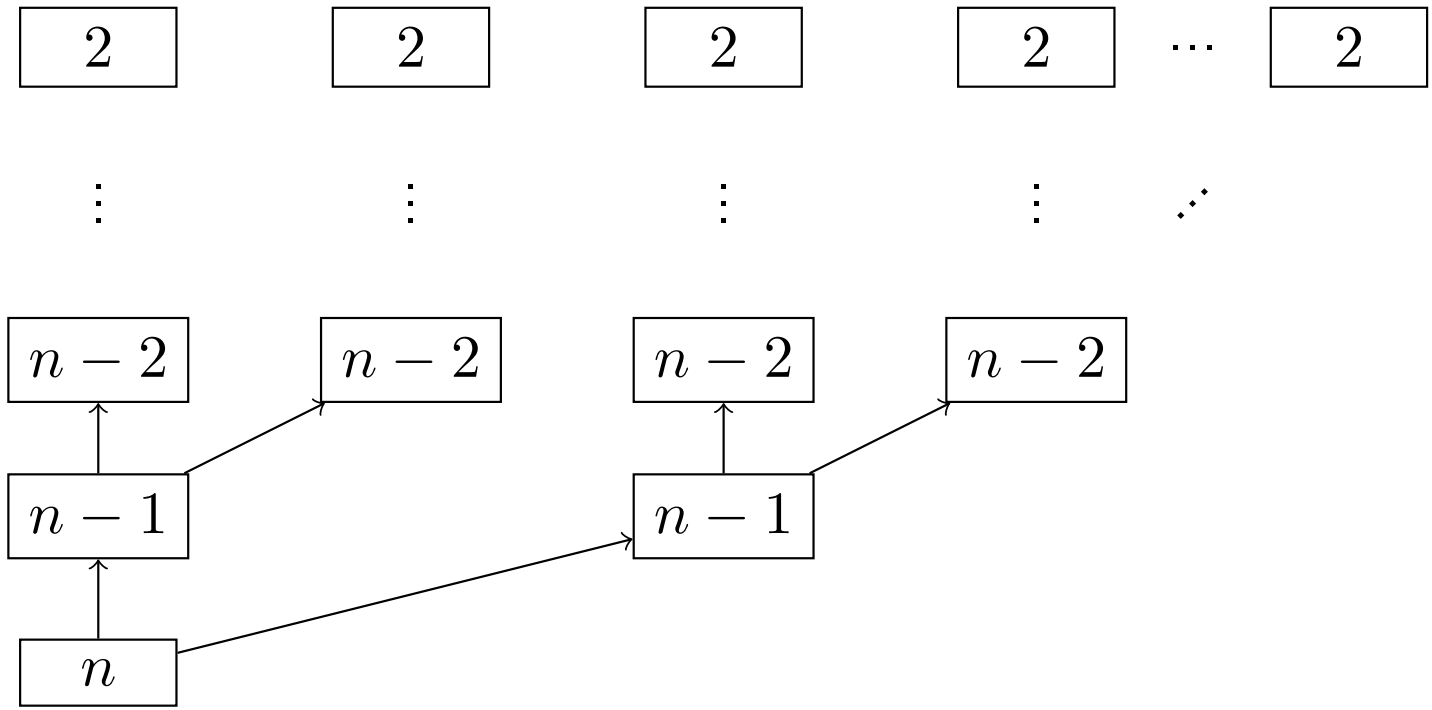
```
int fib1(int n) {
  if (n == 1 || n == 2) {
    return 1;
  } else {
    return fib1(n - 1)
      + fib1(n - 2);
  }
}
```

Förenklat

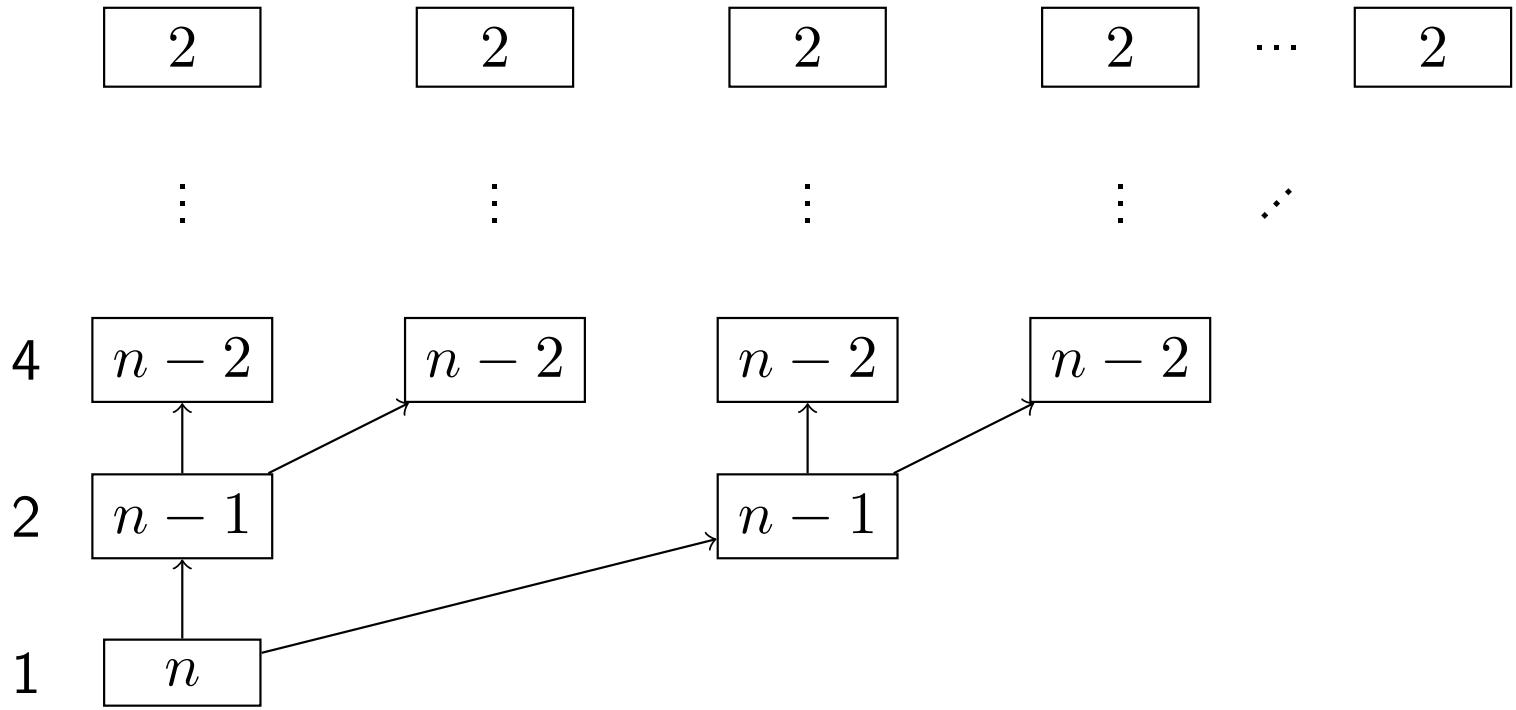
```
int fib1(int n) {
  if (n == 1 || n == 2) {
    return 1;
  } else {
    return fib1(n - 1)
      + fib1(n - 1);
  }
}
```

Detta är okej, vi gör algoritmen **långsammare** (och felaktig)! Vårt $\mathcal{O}$-uttryck kommer vara lite för högt, men det kommer fortfarande stämma.

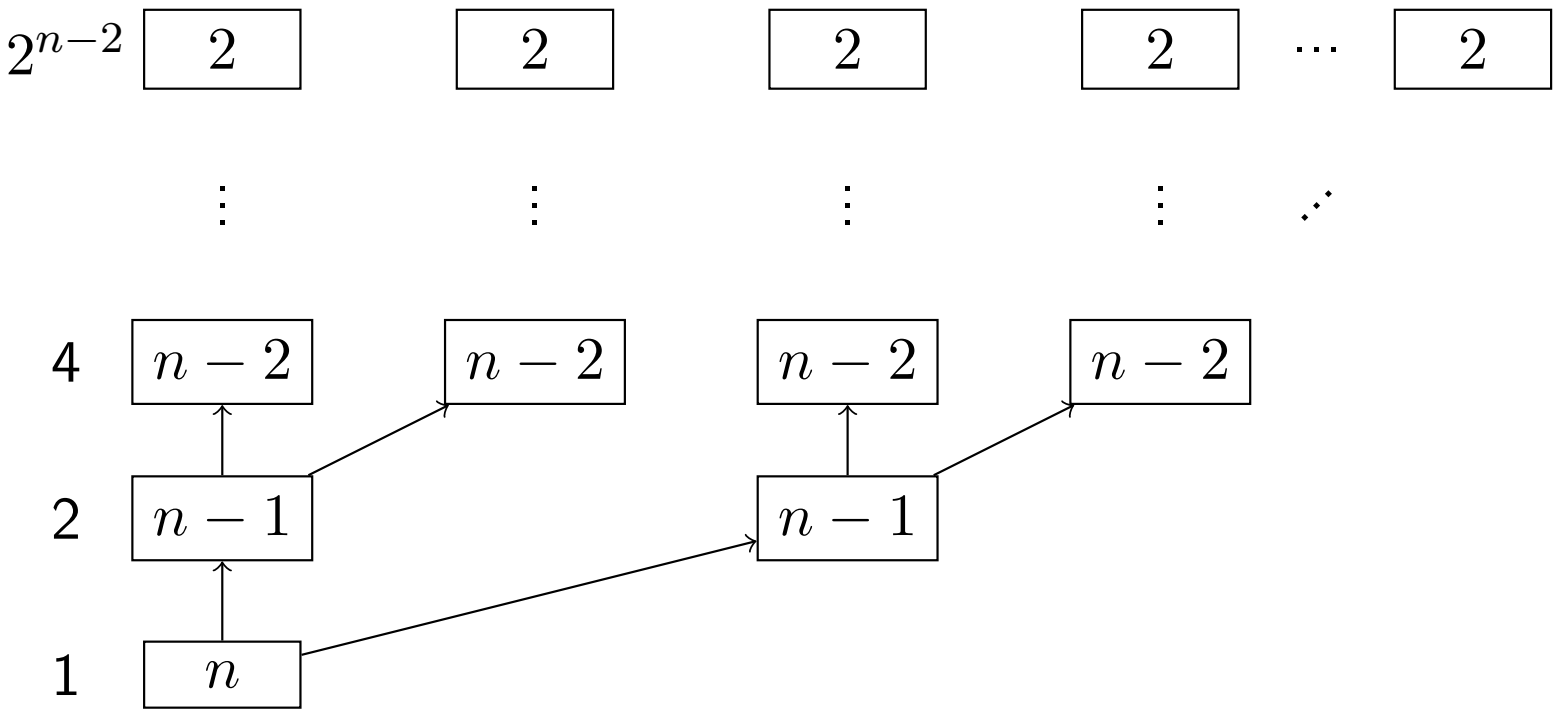
Analys



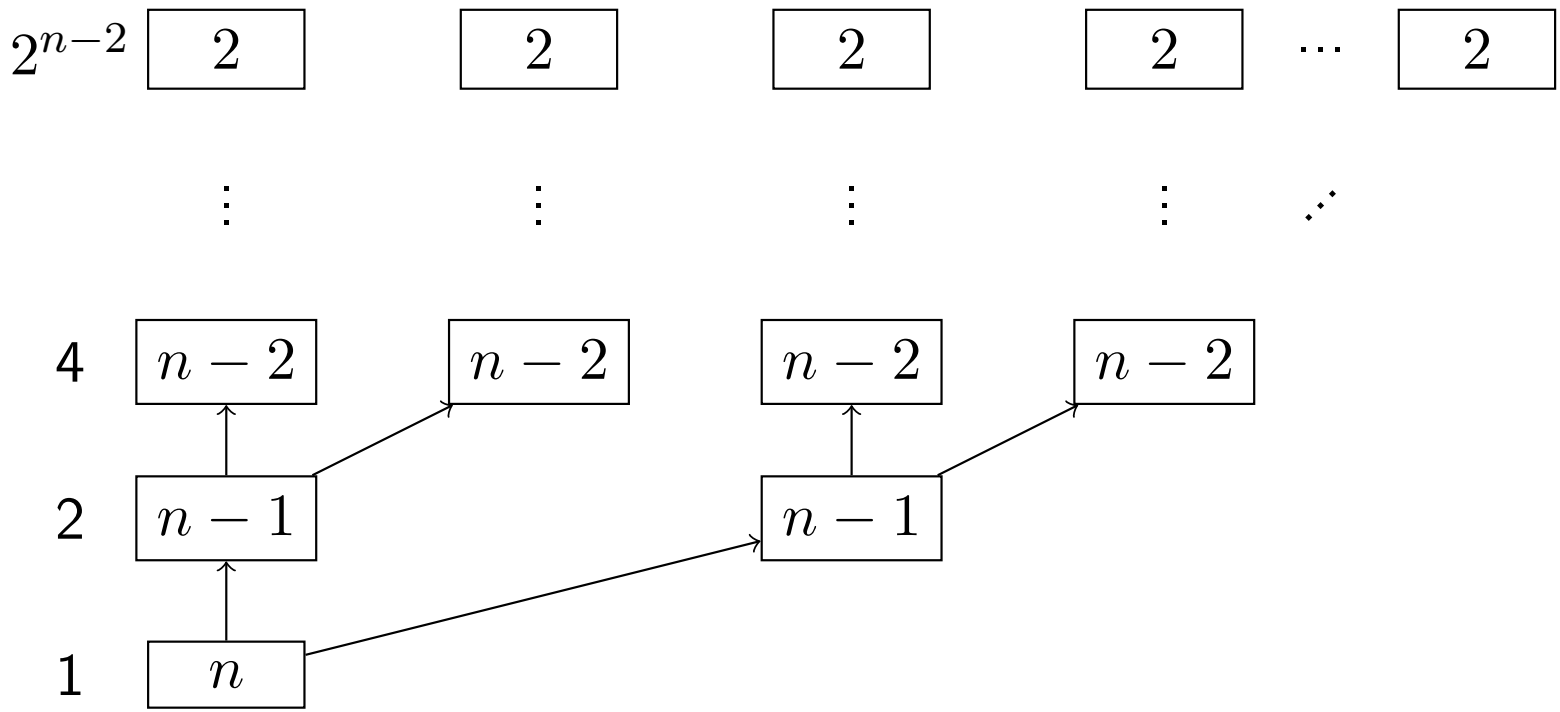
Analys



Analys

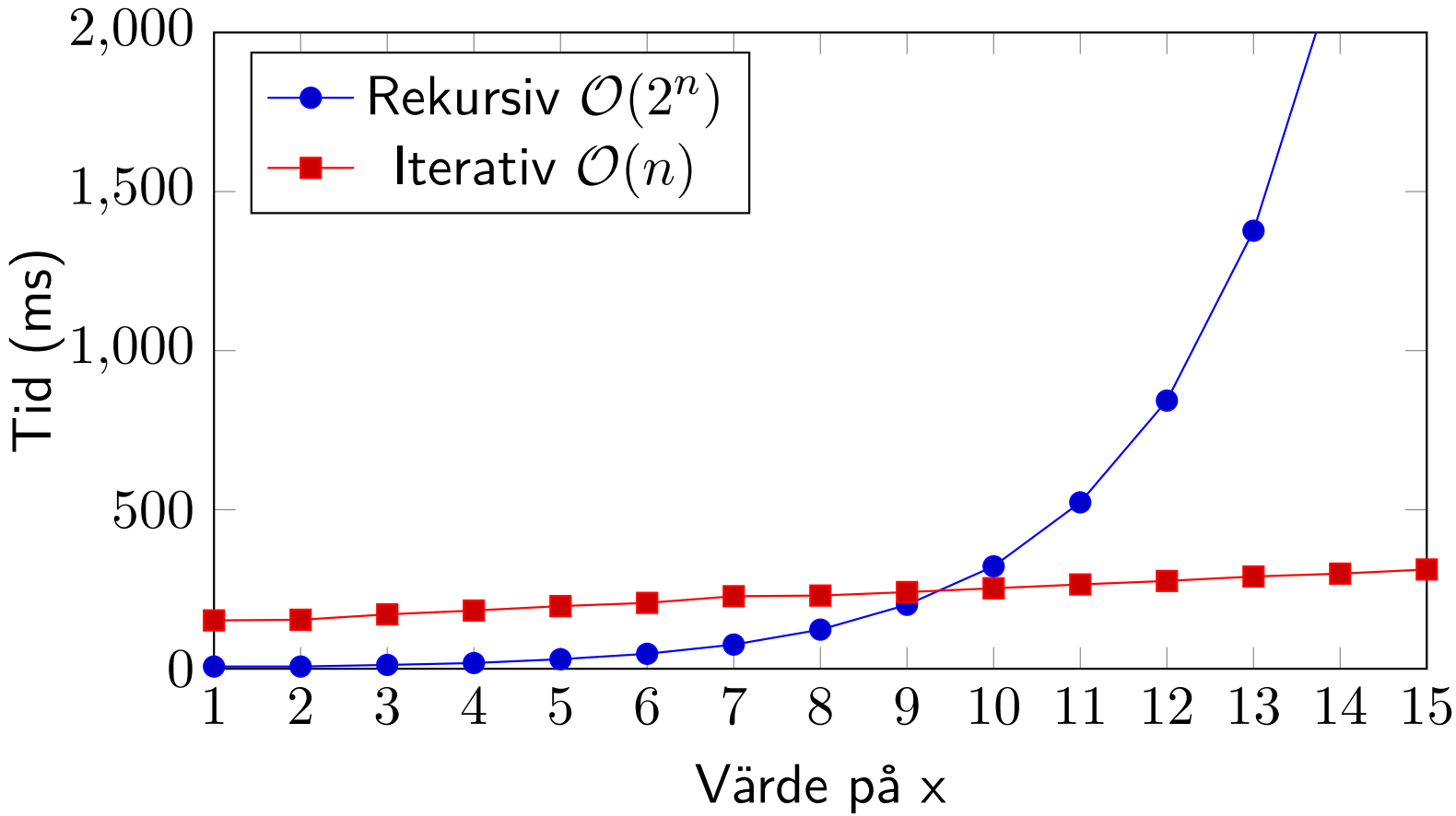


Analys



Totalt: $2^{n-1} - 1 \in \mathcal{O}(2^n)$ anrop

Åter till mätdatan – Stämmer beräkningarna?



LINKÖPING UNIVERSITY

Nästa gång

Nu har vi grunderna för att analysera linjära strukturer!

- Abstrakta datatyper (ADT)
- Array, lista
- Analys av dessa (mer övning på tidskomplexitet, samt nya koncept)

Extrauppgifter på Kattis

- `erase` (enkel)
  Introduktion till Kattis
- `howmanyzeros` (svårare)
  Tänk på tidskomplexiteten på er lösning. Räkna med att ni kan exekevera 1 000 000 000 000 operationer. Vilken tidskomplexitet måste er lösning då ha?

Fördjupningsfrågor

- Hur mycket minne använder båda algoritmerna för fibonacci uttryckt i $\mathcal{O}$-notation?
- Kan vi förbättra den iterativa lösningen på något sätt?
- Kan vi beräkna `fibonacci(n)` på kortare tid än $\mathcal{O}(n)$?

Filip Strömbäck

www.liu.se

LINKÖPING UNIVERSITY