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)

Kursledare	Filip Strömbäck
Assistenter	Malte Nilsson
	Edvin Dyremark
	Alexander Engström
	David Warnquist
Administratör	Annelie Almquist

Examination

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

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

DAT1 Datortentamen, 2hp (U, 3, 4, 5)
Frågor om användandet av datastrukturer och algoritmer.
Extrauppgifter och deadline på laborationer ger upp till 10% bonus mot högre betyg.

OpenDSA – Digital kursbok

- Digital kursbok med interaktiva övningar
- Logga in med LiU-id, dubbelkolla rubrik
- 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

Föreläsningar

- Fokus på hur info från OpenDSA kan *användas*
- Slides finns på kurshemsidan, men är *inte* tänkta att kunna läsas i isolation
- Efter varje föreläsning finns 2 extrauppgifter, ger extrapoäng på tentan
  - Relaterade till det som tagits upp
  - En enklare och en svårare
  - Löses individuellt
- Ställ hemskt gärna frågor!

Laborationer

- Parvis
- Anmälan sker i Webreg (länk på kurshemsidan)
  - 3 grupper: DI.A, DI.B, IP
  - Separata grupper för de som vill hitta labbpartner
- Innehåll:
  1. AVL-träd
  2. Hashning
  3. Ordkedjor
  4. Mönsterigenkänning
- Notera: Givna testfall är inte heltäckande. Skriv också egna testfall (med verktyg på kurshemsidan)

Planering

Vecka	Fö	Lab
36	Komplexitet, Linjära strukturer	----
37	Träd, AVL-träd	1---
38	Hashning	1---
39	Grafer och kortaste vägen	12--
40	Fler grafalgoritmer	-23-
41	Sortering	--3-
42	Mer sortering, beräkningsbarhet	--34
43	Tentaförberedelse	---4

Se kommentarer i TimeEdit för deadlines!

Ändringar från förra året

- Reviderad labb 4.
- Reviderad formulering av sista frågan i labb 2.

LINKÖPING UNIVERSITY

Hur man löser alla problem enligt Richard Feynman:

1. Write down the problem
2. Think really hard
3. Write down the answer

Hur man löser alla problem enligt Richard Feynman:

1. Write down the problem
2. Think really hard
3. Write down the answer

DALG hjälper oss med steg 2!

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 bergshäll.

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 anpassa dem så att du kan lösa mer komplicerade problem, och så att lösningen blir mer effektiv.

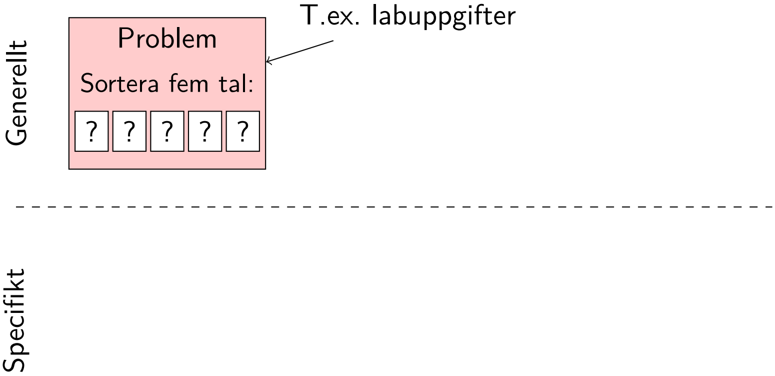
Varför DALG?

- *Veta* vilka verktyg som finns, och hur de fungerar
- *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.
- *Känna till* gränserna för vad som är möjligt att göra

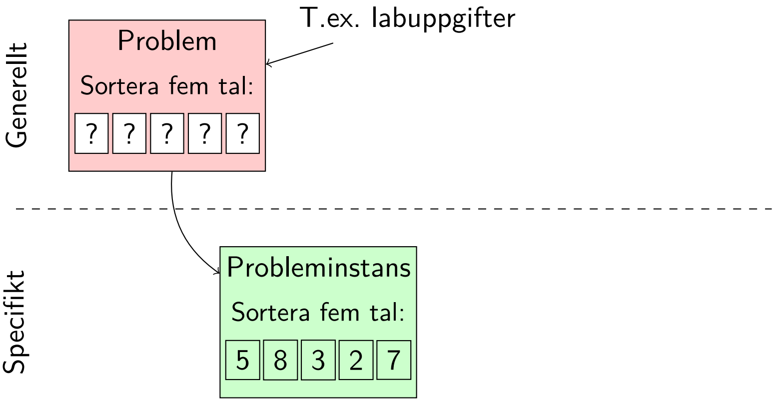
För att effektivt kunna lösa svåra problem, eller problem med stora mängder data.

LINKÖPING UNIVERSITY

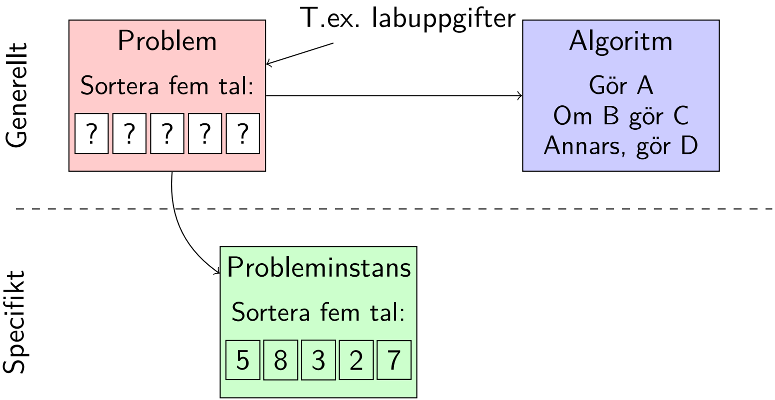
Problem, program och algoritmer



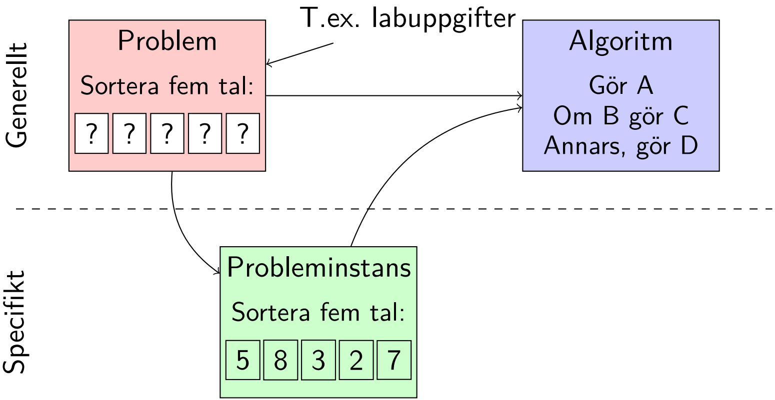
Problem, program och algoritmer



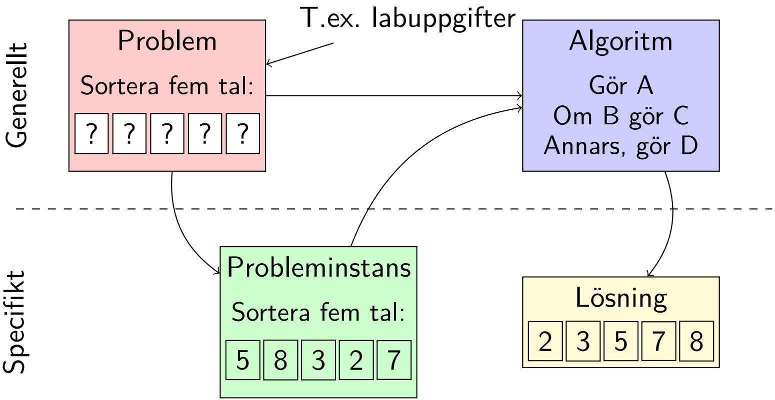
Problem, program och algoritmer



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 räknar förekomster av olika tal i en array:

```
vector<int> count(const vector<int> &input, int max_value)
```

```
count({1, 2, 1, 3}, 4) ⇒ {0, 2, 1, 1}
```

Vilken implementation är snabbast?

A:

```
vector<int> solution_a(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < max_val; i++) {
    result[i] = std::count(input.begin(), input.end(), i);
  }
  return result;
}
```

Vilken implementation är snabbast?

B:

```
vector<int> solution_b(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < input.size(); i++) {
    int value = input[i];
    result[value] += 1;
  }
  return result;
}
```

Kompilatoroptimeringar?

Med flagga -O2 eller -O3 kan vi be kompilatorn att optimera vår kod.

Hur påverkar det körtiden?

Varför behöver jag som programmerare tänka på att skriva effektiv kod när jag kan låta kompilatorn optimera den?

Kompilatoroptimeringar?

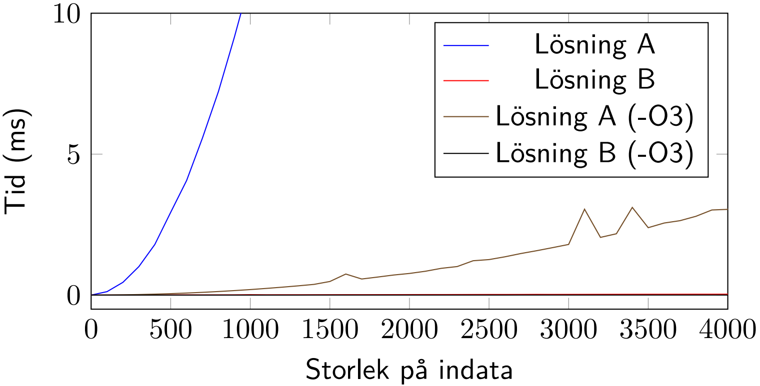
Med flagga `-O2` eller `-O3` kan vi be kompilatorn att optimera vår kod.

Hur påverkar det körtiden?

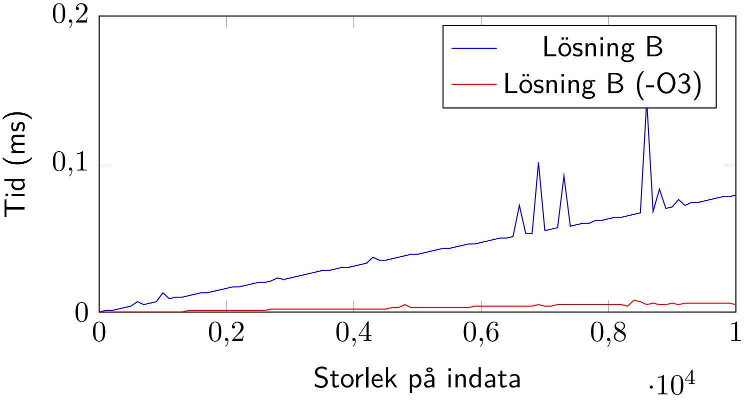
Varför behöver jag som programmerare tänka på att skriva effektiv kod när jag kan låta kompilatorn optimera den?

$\Rightarrow$ Optimeringar i kompilatorn innebär bara att den spenderar mer tid på att generera bra maskinkod. Den ändrar (nästan) aldrig vilken algoritm som används, och hur *data* representeras!

Vad händer för olika storlek på indata?



Vad händer för olika storlek på indata?



Vad är intressant?

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

- Vi är intresserade av vad som händer när indata **växer** – om jag lägger till ett till element, hur mycket dyrare blir det då?
- Vi vill kunna jämföra olika algoritmer
- Vi är intresserade av **helheten**

Vi behöver ett sätt att resonera om detta!

Vad är intressant?

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

- Vi är intresserade av vad som händer när indata **växer** – om jag lägger till ett till element, hur mycket dyrare blir det då?
- Vi vill kunna jämföra olika algoritmer
- Vi är intresserade av **helheten**

Vi behöver ett sätt att resonera om detta!

Notera: Små fall kan också vara viktiga, men börja med en effektiv algoritm och optimera den vid behov.

LINKÖPING UNIVERSITY

Hur körs ett program?

Idé: räkna antalet ”operationer” som krävs:

- Aritmetiska operationer
- Tilldelingar
- Läsning/skrivning av minnet
- ...

Vi antar att alla ”enkla” operationer tar lika lång tid

Tidsåtgång – exempel

```
int a(int n) {
  return n * 2;
}
```

Tidsåtgång – exempel

```
int a(int n) {
  return n * 2;
}
```

$$\Rightarrow t(n) = 2$$

Tidsåtgång – exempel

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

Tidsåtgång – exempel

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

$\Rightarrow t(n) = 4 + 3n$

Tidsåtgång – exempel

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

Tidsåtgång – exempel

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

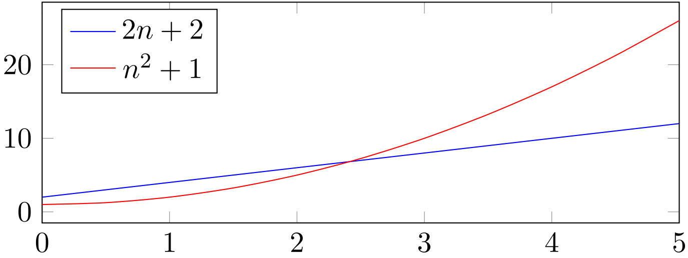
$\Rightarrow t(n) = 4 + 4n$

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

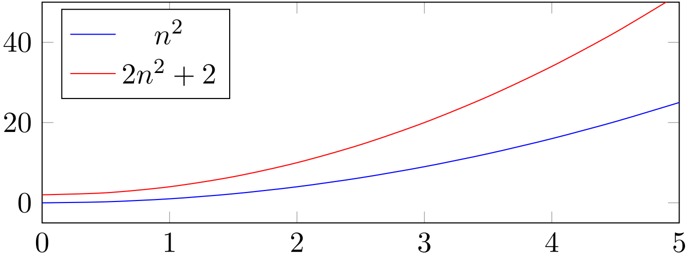


$2n + 2 \leq c(n^2 + 1)$ för $c = 1$ och $n \geq 3$

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

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

Exempel

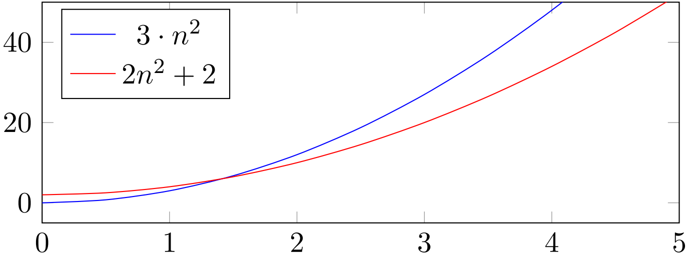


$n^2 \in \mathcal{O}(2n^2 + 2)$ (enkelt att se)

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

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

Exempel

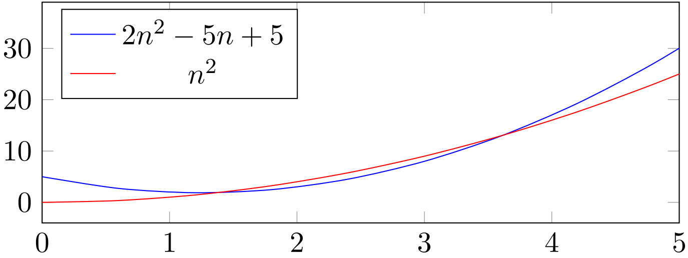


$n^2 \in \mathcal{O}(2n^2 + 2)$ (enkelt att se)

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

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

Exempel

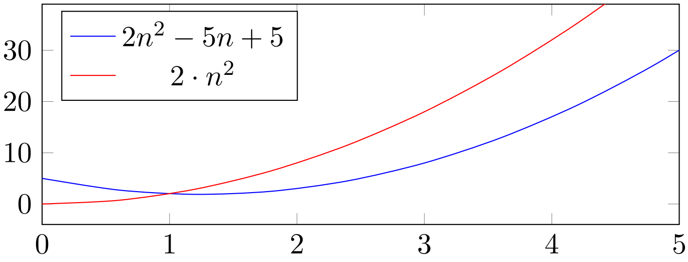


$n^2 \in \mathcal{O}(2n^2 - 5n + 5)$ (enkelt att se)

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

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

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

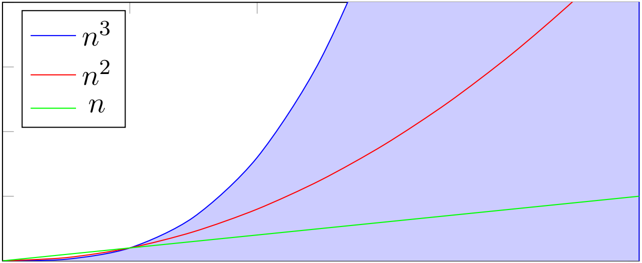
Observationer – Regler

- 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 $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$. Då gäller även $f(n) \in \Theta(g(n))$ samt $g(n) \in \Theta(f(n))$.
- Alltså kan vi representera våra olika ”grupper” i form av den mest förenklade formeln

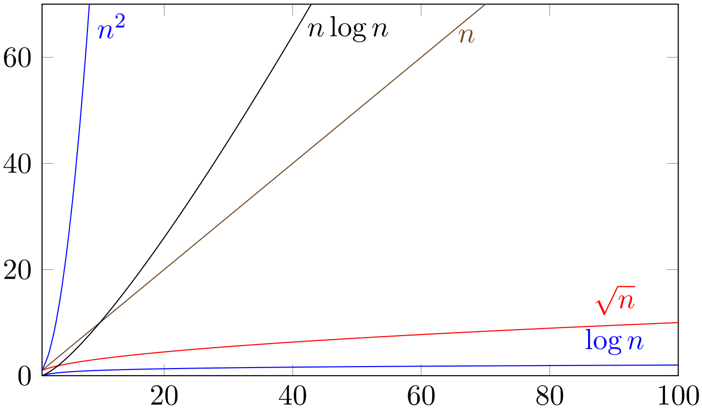
Förenkling med regler

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

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

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

⇒ Vi kan anta att varje operation tar 1 tidsenhet

Exempel

```
int a(int n) {
  return n * 2;
}
```

$$\Rightarrow t(n) = 2 \in \mathcal{O}(1)$$

Exempel

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

$$\Rightarrow t(n) = 4 + 3n \in \mathcal{O}(n)$$

Exempel

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

$$\Rightarrow t(n) = 4 + 4n \in \mathcal{O}(n)$$

Exempel

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

Exempel

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

$\Rightarrow t(n) \in \mathcal{O}(n^2)$

Tidsåtgång

```
vector<int> solution_b(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < input.size(); i++) {
    int value = input[i];
    result[value] += 1;
  }
  return result;
}
```

Tidsåtgång

```
vector<int> solution_b(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < input.size(); i++) {
    int value = input[i];
    result[value] += 1;
  }
  return result;
}
```

$\Rightarrow t(n) = ?$

Vad är n?

- Beror på vad vi vill analysera
- Välj n som beskriver indata på lämpligt sätt
- Ibland behöver vi flera olika

Här:

- n - storlek på array
- m - maximalt värde
- I det här fallet är det inte orimligt att anta att $n \approx m$, men detta beror på innehållet i arrayen!

Tidsåtgång

```
vector<int> solution_b(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < input.size(); i++) {
    int value = input[i];
    result[value] += 1;
  }
  return result;
}
```

Tidsåtgång

```
vector<int> solution_b(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < input.size(); i++) {
    int value = input[i];
    result[value] += 1;
  }
  return result;
}
```

$\Rightarrow t(n, m) \in \mathcal{O}(n + m)$

Om $n \approx m : \mathcal{O}(n)$

Tidsåtgång

```
vector<int> solution_a(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < max_val; i++) {
    result[i] = std::count(input.begin(), input.end(), i);
  }
  return result;
}
```

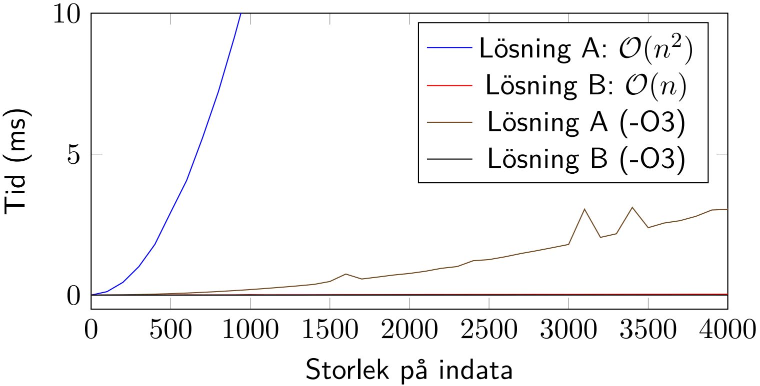
Tidsåtgång

```
vector<int> solution_a(
    const vector<int> &input,
    int max_val)
{
  vector<int> result(max_val, 0);
  for (int i = 0; i < max_val; i++) {
    result[i] = std::count(input.begin(), input.end(), i);
  }
  return result;
}
```

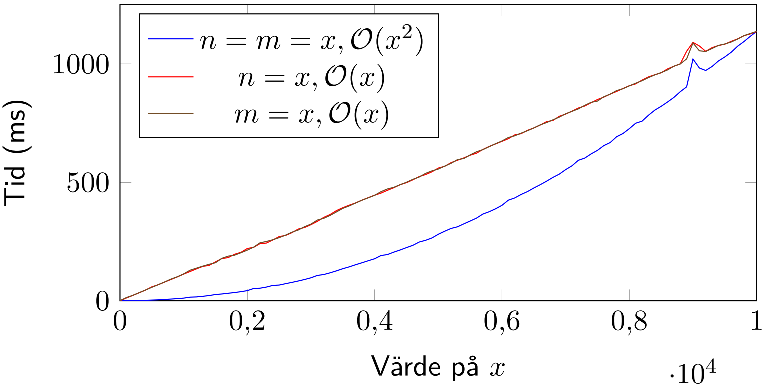
$\Rightarrow t(n, m) \in \mathcal{O}(nm)$

Om $n \approx m : \mathcal{O}(n^2)$

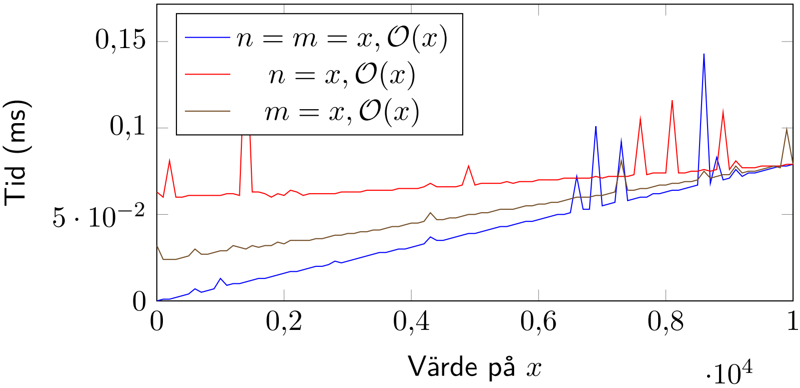
Åter till mätdatan – Stämmer beräkningarna?



Variation i endast en dimension (Lösning A)



Variation i endast en dimension (Lösning B)



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

- 272 (enkel)
  Övning på att använda systemet.
- 10340 (svårare)
  Tänk på tidskomplexiteten vid sökning i strängar!

Filip Strömbäck

www.liu.se

LINKÖPING UNIVERSITY