## Föreläsning 1. Introduktion

### Vad är en algoritm?

Några exempel på "algoritmer".

1. Häll 1 dl havregryn och ett kryddmått salt i 2 $\frac{1}{2}$ dl kallt vatten. Koka upp och kocka gröten ca 3 minuter. Rör om då och då. Servera t.ex med mjölk eller hallonsylt.

2. Olika arter får leva tillsammans med begränsade resurser. De enstaka individerna fortplantar sig och dör. Nya arter uppstår genom förändringar i arvsmassan. Förändringar som uppstår kommer att överleva om de klarar konkurrensen med de andra arterna.

3.

```
INSÄTTNINGSSORTERING(A)
(1)   for j = 2 to length (A)
(2)       key ← A[j]
(3)       i ← j − 1
(4)       while i > 0 och A[i] > key
(5)           A[i + 1] ← A[i]
(6)           i ← i − 1
(7)       A[i + 1] ← key
(8)   return A
```

Utmärkande drag:

Indata/Utdata

Informell beskrivning:
En algoritm är en ändlig beskrivning av hur man steg för steg löser ett problem. Algoritmen tar indata, som beskriver ett problem och producerar utdata, en lösning till problemet.

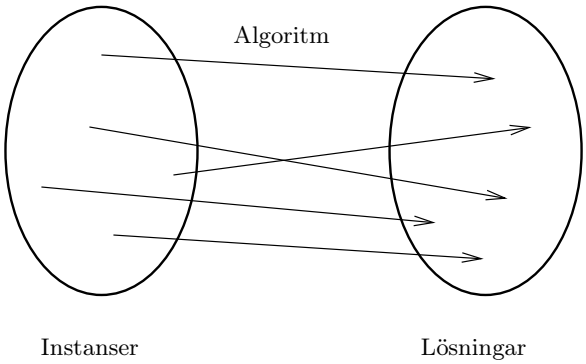
**Tre egenskaper hos "idealiserade" algoritmer**

1. De är plattformsoberoende. De skall kunna simuleras på olika sätt.

2. De är helt mekaniska och ointelligenta.

3. De ger alltid korrekt resultat.

En algoritm kan ses som en funktion:
A: Probleminstanser → Lösningar




Denna beskrivning är rent matematisk och tar inte hänsyn till hur effektiv en algoritm är.
Algoritmer antas vara presenterade på en form som kan översättas till ett dataprogram.
Ofta kommer vi att beskriva algoritmer i s.k. pseudokod. (Mer om detta senare.)

Mer detaljerad beskrivning:

**Beräkningsmodeller**

- Matematisk beskrivning av beräkningar.
- Turingmaskiner, RAM, automater.

Beräkningsmodeller ger en precisering av hur programkroppen i en algoritm kan fungera. Används mest för att bevisa begränsningar hos algoritmer.

**Vad är en effektiv algoritm?**

Standardmåttet som används är **tidskomplexitet**.
Ex: Vi sorterar $n$ st tal med insättningssortering. Det tar tid $\sim cn^2$ för stora $n$. Konstanten $c$ är beroende av implementation och man försöker att tänka bort den genom att använda s.k. ordo-klasser.

### Tillväxt hos funktioner

Komplexiteten beskrivs ofta i $O(\cdot)$-notation. Den bortser från konstanta faktorer och anger vad som händer då $n \to \infty$ .

- $f(n) \in O(g(n))$ om $f(n)$ växer högst lika snabbt som $g(n)$.

  D.v.s. det finns $c > 0$ och $n_0$ så att $f(n) \leq cg(n)$ för alla $n \geq n_0$.
- $f(n) \in \Theta(g(n))$ om $f(n)$ och $g(n)$ växer lika snabbt.
- $f(n) \in \Omega(g(n))$ om $f(n)$ växer minst lika snabbt som $g(n)$.

Ett annat komplexitetsmått är **minneskomplexitet**:
Storleken på minnet som krävs för att utföra en beräkning.
Tidskomplexitet anses vara det viktigaste komplexitetsmåttet.

### Tre problem med definition av komplexitet

1. Tiden är proportionell mot antalet **operationer**. Vad räknas som en operation?
ex: Beräkna $521 \cdot 394 = 205274$ med "vanlig" algoritm. Hur många operationer krävs?
Två möjliga svar:
a. En operation!
b. 521 har 9 bitar. 394 har 8 bitar. Totalt krävs $9 \cdot 8 = 72$
De två typerna av kostnad är:

**Enhetskostnad:**

- Varje grundläggande operation tar en tidsenhet.
- Varje variabel upptar en minnesenhet.

(Beräkningsmodell: RAM)
**Bitkostnad:**

- Varje operation på en enskild bit tar en tidsenhet.
- En bit upptar en minnesenhet.

Bägge dessa mått på kostnaden används.

2. Indatas storlek.
Antag att ett komplicerat objekt är indata. Vad är då indatas storlek? I princip är det storleken lika med antalet bitar i den effektivaste representationen av objektet i en datastruktur. I praktiken kan det ibland vara svårt att avgöra vad det är.

3. Medelvärde/Värsta fall.

Ex: Quicksort är en känd algoritm för att sortera en lista tal. I medeltal kräver den $O(n \log n)$ operationer. I värsta fall kan den dock kräva $O(n^2)$ operationer. Vanligen används **värsta fall** som mått.

Skäl: Det är ofta svårt att känna till sannolikhetsfördelningen på möjliga indata för en korrekt medeltalsberäkning. Värsta fall är enklare att beräkna och är i många fall ungefär lika med medeltal.

### Definition av effektiv algoritm

En algoritm som arbetar i tid $O(n^k)$ för något $k$. Kallas för **polynomiell** algoritm.

### Korrekthetsanalys

Med korrekthetsanalys menas ett bevis för att algoritmen gör det den är tänkt att göra. Generella principer för korrekthetsanalys är mycket komplicerade. Ett knep som är omtyckt är **invarianter**. De används vid analys av loopar i algoritmer. En invariant är ett påstående som är sant innan man går in i loopen, är sant medan man kör loopen och när man går ut ur loopen. (Problemet är att visa att något är en invariant.) Invarianten används sedan för att visa att algoritmen uppnår rätt resultat.
Ex:

```
GRAFSÖKNING(G, s, t)
(1)   U ← {s}
(2)   while t ∉ U och det finns kant (v, w) med v ∈ U
      och w ∉ U
(3)       U ← U ∪ {w}
(4)   if t ∈ U
(5)       return Ja
(6)   return Nej
```

Den naturliga invarianten här är att $U$ hela tiden är en **sammanhängande** mängd noder som innehåller $s$. När man går ur slingan så finns antingen $t$ i $U$ eller så är $U$ en maximal sammanhängande mängd utan $t$.

### Komplexitetsteori

Handlar om analys av problem som inte går att lösa effektivt med någon algoritm.

### Vad är ett problem?

En abstrakt beskrivning är att ett problem består av Indata/Mål. Givet indata skall man skall man ha reda på svaret, kallat Målet, på en fråga.
Ex:
Indata: En graf $G$.
Mål: En färgning av grafen med $k$ färger (om en sådan finns).

Ett problem definieras med **variabel** indata. Som t.ex.
SORTERING
Indata: Lista $\{a_1, a_2, ..., a_n\}$ av tal.
Mål: En sortering $\{a_{i_1}, a_{i_2}, ..., a_{i_n}\}$ så att $a_{i_1} \leq a_{i_2} \leq ....$

Ett specialfall av ett problem kallas för en **instans** av problemet.
Instans av SORTERING.
Indata: $\{7, 3, 9, 2, 4\}$
Mål: $\{2, 3, 4, 7, 9\}$
En algoritm löser ett problem om den löser alla instanser.

## Olika typer av problem

Beslutsproblem: Givet indata är utdata "Ja" eller "Nej". Vanligen är problemfrågan: Givet indata, finns en lösning eller ej.

Optimeringsproblem: Givet indata är utdata ett tal. Vanligen presenteras storleken på en optimal lösning till problemet.

Konstruktionsproblem: Givet indata är utdata en struktur. Vanligen presenteras lösningen till problemet.

## Svåra problem

1. Problem som inte kan lösas med **någon algoritm**.
2. Problem som inte kan lösas med **någon effektiv algoritm**.
Det går att visa att problem av båda typerna finns. Det stora problemet inom komplexitetsteori är att avgöra vilka problemen av typ 2 är.
Vanligt redskap: **Reduktioner**
Om ett problem A kan reduceras till ett problem B så är inte A svårare att lösa än B. Om A är svårt måste också B vara svårt.
En viktig klass av **förmodat svåra problem**: NP-fullständiga problem.

## Klassen NP

Antag att ett beslutsproblem är av följande slag:
Givet indata $x$, finns det en lösning?
Antag att det finns en polynomiell algoritm A som givet data $y$ avgör om $y$ är en lösning till problemet.
$A(x, y)$ = Ja/Nej.
Då tillhör problemet klassen NP.
De flesta "svåra" NP-problemen verkar ha exponentiell lösningstid.
P $\subseteq$ NP
Öppet problem: P=NP ?

## Kursöversikt

**Beräkningsmodeller**

- Matematisk beskrivning av beräkningar.
- Turingmaskiner, RAM, automater.

**Konstruktion och analys av effektiva algoritmer**

Viktiga speciella algoritmer

- grafalgoritmer
- flödesalgoritmer
- sorteringsalgoritmer
- linjärprogrammering

Viktiga generella metoder

- dekomposition
- giriga algoritmer strut
- dynamisk programmering

Komplexitetsberäkningar och korrekthetsanalys.

**Komplexitetsteori**

- Vilka beräkningsresurser kräver ett problem?
- Problem som inte kan lösas av någon dator.
- Problem som saknar effektiva algoritmer.
- Reduktioner mellan problem.
- Komplexitetsklasser.

## Förkunskaper från INDA/TILDA

- Sorterings- och sökalgoritmer (t.ex. QuickSort resp. binärsökning).
- Datastrukturer: Listor, stackar, träd, heapar, hashtabeller.
- Algoritmanalys med $O(\cdot)$-notation.
- Motivering av *hur* och *varför* en algoritm fungerar.