

# TDDI16 – Föreläsning 9

Sortering i linjär tid, beräkningsbarhet

Filip Strömbäck

# Planering

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

- 1 Hur snabbt kan vi sortera data?
- 2 Sortering i linjär tid
- 3 Tillämpningar
- 4 "Intressanta" algoritmer
- 5 Beräkningsbarhet
- 6 Sammanfattning

## Bättre än $\mathcal{O}(n \log n)$ ?

Vi vet att sorteringsproblemet avgränsas av  $\Omega(n)$  och  $\mathcal{O}(n \log n)$  (varför?).

Kan vi hitta en ännu bättre algoritm som sorterar indata med hjälp av en jämförelsefunktion?

## Bättre än $\mathcal{O}(n \log n)$ ?

Vi vet att sorteringsproblemet avgränsas av  $\Omega(n)$  och  $\mathcal{O}(n \log n)$  (varför?).

Kan vi hitta en ännu bättre algoritm som sorterar indata med hjälp av en jämförelsefunktion?

Sterlings formel:  $\ln(n!) \approx n \ln(n) - n$  för stora  $n$ .

Alltså:  $\Omega(\log(n!)) = \Omega(n \log(n))$

- 1 Hur snabbt kan vi sortera data?
- 2 Sortering i linjär tid
- 3 Tillämpningar
- 4 "Intressanta" algoritmer
- 5 Beräkningsbarhet
- 6 Sammanfattning

## Sortering i $\mathcal{O}(n)$ ?

- Jämförelsebaserade sorteringar begränsas av  $\Omega(n \log n)$

Hur sorterar vi då i  $\mathcal{O}(n)$ ?

## Sortering i $\mathcal{O}(n)$ ?

- Jämförelsebaserade sorteringar begränsas av  $\Omega(n \log n)$

Hur sorterar vi då i  $\mathcal{O}(n)$ ?

Vi skippar jämförelserna!



## Count sort – Idé

1. Skapa en array av heltal med ett index för varje möjligt tal i indata
2. Räkna förekomster av alla heltal i indata, lagra dem i arrayet
3. Skriv tillbaka talen i ordning utifrån antalet

Original:

3	1	2	3	2	0
---	---	---	---	---	---

## Count sort – Idé

1. Skapa en array av heltal med ett index för varje möjligt tal i indata
2. Räkna förekomster av alla heltal i indata, lagra dem i arrayet
3. Skriv tillbaka talen i ordning utifrån antalet

Original:

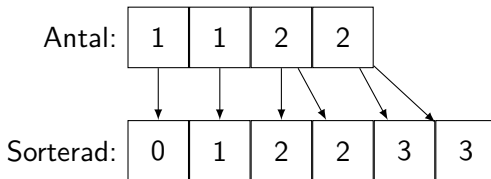
3	1	2	3	2	0
---	---	---	---	---	---

Antal:

1	1	2	2
---	---	---	---

## Count sort – Idé

1. Skapa en array av heltal med ett index för varje möjligt tal i indata
2. Räkna förekomster av alla heltal i indata, lagra dem i arrayet
3. Skriv tillbaka talen i ordning utifrån antalet



## Count sort – Implementation

```
void count_sort(vector<int> &data) {  
    vector<int> count(max_number, 0);  
    for (int x : data)  
        count[x]++;  
  
    size_t pos = 0;  
    for (int i = 0; i < int(max_number); i++)  
        for (int c = 0; c < count[i]; c++)  
            data[pos++] = i;  
}
```

# Bucket sort

En grov sortering som första steg.

- Lägg elementen i olika "lådor" baserat på deras storlek
- Sortera sedan varje låda med någon annan sorteringsalgoritm

## Bucket sort – implementation

```
void bucket_sort(vector<int> &data) {  
    vector<vector<int>> buckets(num_buckets);  
    const int divide = (max_number / num_buckets);  
    for (int x : data)  
        buckets[x / divide].push_back(x);  
  
    size_t pos = 0;  
    for (vector<int> &x : buckets) {  
        sort(x.begin(), x.end());  
        copy(x.begin(), x.end(), data.begin() + pos);  
        pos += x.size();  
    }  
}
```

## Radix sort (LSD first) – Idé

Om talen varierar mycket blir det snabbt mycket minne som krävs...

**Idé:** Vi sorterar flera gånger!

- Sortera efter den minst signifikanta siffran
- Fortsätt sedan med nästa siffra
- ...

Det är viktigt att sorteringen är **stabil**.

Bucket sort passar bra här!

## Radix sort (LSD first) – Implementation

```
void radix_sort(vector<int> &data) {  
    for (int divide = 1;  
        divide < 10 * max_number;  
        divide *= 10) {  
  
        bucket_sort_digit(data, divide);  
    }  
}
```



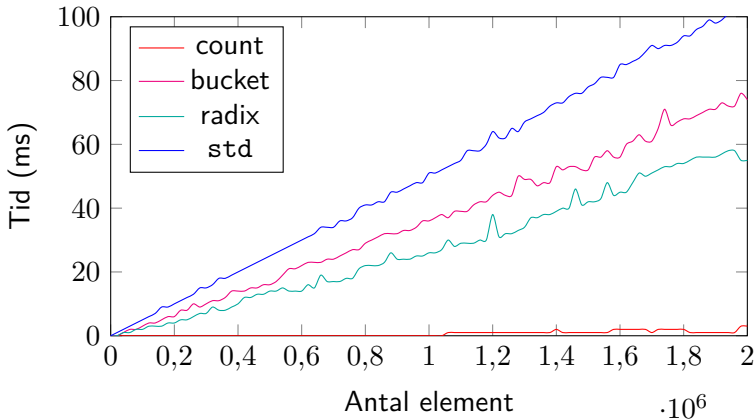
## Radix sort – modifierad bucket sort

```
void bucket_sort_digit(vector<int> &data, int div) {  
    vector<vector<int>> buckets(10);  
  
    for (int x : data)  
        buckets[(x / div) % 10].push_back(x);  
  
    size_t pos = 0;  
    for (vector<int> &x : buckets) {  
        copy(x.begin(), x.end(), data.begin() + pos);  
        pos += x.size();  
    }  
}
```

## Radix sort (MSD first)

- Vi kan också börja med den mest signifikanta siffran!
- I så fall sorterar vi de olika "partitionerna" separat i stället för samtidigt som i tidigare fallet

## Jämförelse – Element mellan 0 och 10000



## Sound of sorting

Det finns många visualiseringar av diverse sorteringsalgoritmer på internet, exempelvis:

- 15 algoritmer, en och en:  
`https://www.youtube.com/watch?v=kPRA0W1kECg`
- 6 olika algoritmer samtidigt:  
`https://www.youtube.com/watch?v=ZZuD6iUe3Pc`

- 1 Hur snabbt kan vi sortera data?
- 2 Sortering i linjär tid
- 3 **Tillämpningar**
- 4 "Intressanta" algoritmer
- 5 Beräkningsbarhet
- 6 Sammanfattning

# Fördela röster

Baserat på problem 12390 från OnlineJudge.

- Det finns  $n$  städer, vardera med  $p_i$  invånare
- Alla ska rösta på valdagen
- Du har  $b$  valurnor, hur ska du fördela dem så att den urnan med flest röster innehåller så få röster som möjligt?
- $1 \leq n \leq 500000$ ,  $n \leq b \leq 2000000$ ,  $p_i \leq 5000000$

- 1 Hur snabbt kan vi sortera data?
- 2 Sortering i linjär tid
- 3 Tillämpningar
- 4 "Intressanta" algoritmer
- 5 Beräkningsbarhet
- 6 Sammanfattning

## Sleep sort

Starta en tråd/process för varje element, låt dem sova i motsvarande antal sekunder och sedan skriva ut sin data.

```
#!/bin/bash
task() { sleep "$1"; echo "$1"; }
for i in "$@"
do
    task $i &
done
wait
```

Risk för fel...

Vad är tidskomplexiteten?



# Machine learning sort

Idé:

1. Sortera en delmängd av datan
2. Träna ett neuralt nätverk att beräkna den slutgiltiga positionen baserat på ett element
3. Använd det för resten av datan

Återigen: Risk att datan inte blir korrekt sorterad...

<https://arxiv.org/abs/1805.04272>

## Bogosort

```
void bogosort(vector<int> &v) {  
    while (!sorted(v))  
        random_shuffle(v.begin(), v.end(), generator);  
}  
  
bool sorted(const vector<int> &v) {  
    for (size_t i = 1; i < v.size(); i++)  
        if (v[i - 1] > v[i])  
            return false;  
  
    return true;  
}
```

## Bogobogosort

```
void bogobogosort(iter begin, iter end) {  
    while (!sorted(begin, end))  
        random_shuffle(begin, end, generator);  
}  
  
bool sorted(iter begin, iter end) {  
    if (begin + 1 == end)  
        return true;  
    vector<int> x(begin, end);  
    bogobogosort(begin, end - 1);  
    return *(end - 2) <= *(end - 1);  
}
```

<http://www.dangermouse.net/esoteric/bogobogosort.html>

# Worstsor

1. Generera alla permutationer av indata, spara i en lista
2. Sortera listan lexiografiskt med hjälp av bubblesort
3. Ta det första elementet

Komplexitet:  $\Omega((n!)^2)$

`https://sites.math.northwestern.edu/~mlerma/  
papers-and-preprints/inefficient\_algorithms.pdf`

## Worstsort – even worse

1. Generera alla permutationer av indata, spara i en lista
2. Sortera listan lexiografiskt med hjälp av **worstsort**, om vi inte har nått ett förutbestämt djup ännu...
3. Ta det första elementet

Komplexitet:  $\Omega(((n!) \dots!)^2) = \Omega((n!^{(k)})^2)$

[https://sites.math.northwestern.edu/~mlerma/papers-and-preprints/inefficient\\_algorithms.pdf](https://sites.math.northwestern.edu/~mlerma/papers-and-preprints/inefficient_algorithms.pdf)

# Dropsort

Iterera genom listan, börja med det andra elementet:

- Jämför elementet med det förra.
- Om det var mindre än det förra, ta bort det.

*Dropsort is what is known in the computer science field as a lossy algorithm...*

<https://www.dangermouse.net/esoteric/dropsort.html>

## Andra intressanta algoritmer

- *Ineffective Sorts*  
<https://xkcd.com/1185/>
- *Stacksort*:  
<https://gkoberger.github.io/stacksort/>
- Med flera...

- 1 Hur snabbt kan vi sortera data?
- 2 Sortering i linjär tid
- 3 Tillämpningar
- 4 "Intressanta" algoritmer
- 5 **Beräkningsbarhet**
- 6 Sammanfattning



## Kan vi lösa alla problem? – The Halting Problem

Vi har en funktion  $h(f, x)$  som returnerar true om anropet  $f(x)$  terminerar.

## Kan vi lösa alla problem? – The Halting Problem

Vi har en funktion  $h(f, x)$  som returnerar true om anropet  $f(x)$  terminerar.

Vi skriver en funktion,  $q(f)$ :

```
void q(f) {  
    if (h(f, f))  
        while (true) {}  
}
```

Terminerar  $q(q)$ ?

## Kan vi lösa alla problem? – The Halting Problem

Vi har en funktion  $h(f, x)$  som returnerar true om anropet  $f(x)$  terminerar.

Vi skriver en funktion,  $q(f)$ :

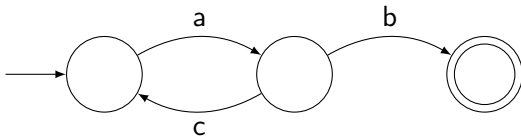
```
void q(f) {  
    if (h(f, f))  
        while (true) {}  
}
```

Terminerar  $q(q)$ ?

Om  $h(q, q)$  är sant:  $q$  terminerar ej  $\Rightarrow$  motsägelse

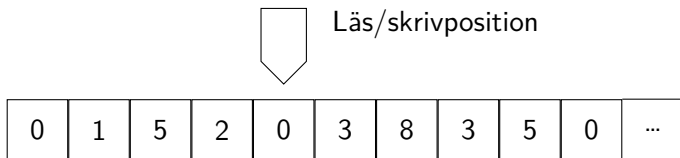
Om  $h(q, q)$  är falskt:  $q$  terminerar  $\Rightarrow$  motsägelse

## Modell – Finit Automata



- Läser indata, går till nästa tillstånd
- Finns två varianter: deterministisk (DFA), ickedeterministisk (NFA)
- DFA och NFA är lika starka

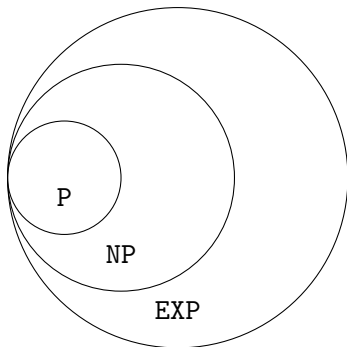
## Modell – Turingmaskiner



- Kan *läsa*, *skriva*, *flytta* huvudet fram eller tillbaka, och *terminera*.
- Ekvivalent med "vanliga" datorer (men kräver oftast fler steg)
- Finns också deterministisk och ickedeterministisk
- Enkel att göra bevis för

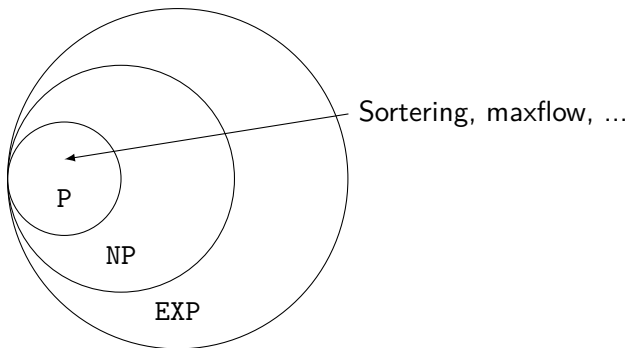
# Komplexitetsklasser

Olika problem är olika “svåra”:



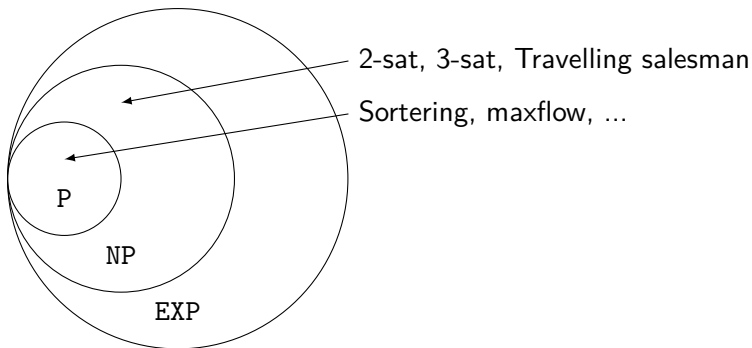
# Komplexitetsklasser

Olika problem är olika “svåra”:



# Komplexitetsklasser

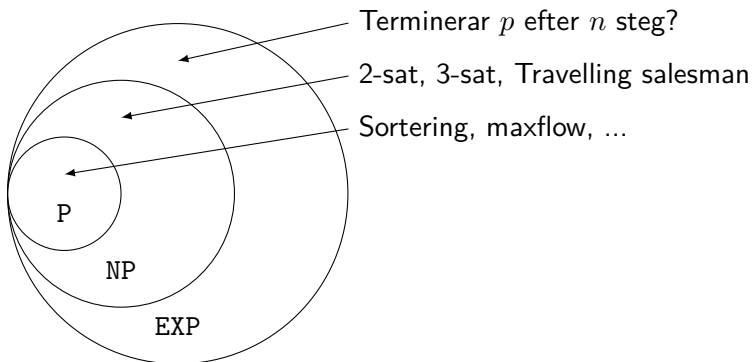
Olika problem är olika “svåra”:





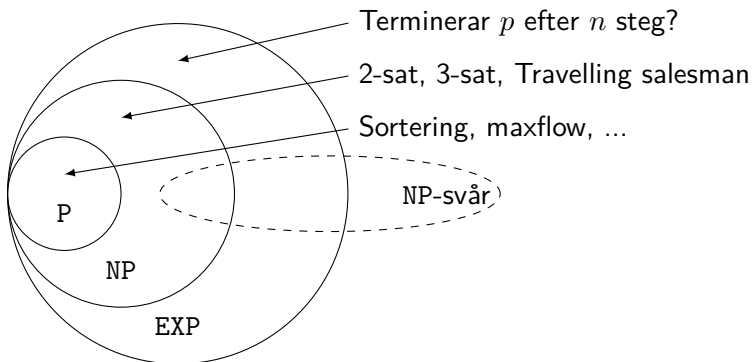
# Komplexitetsklasser

Olika problem är olika “svåra”:



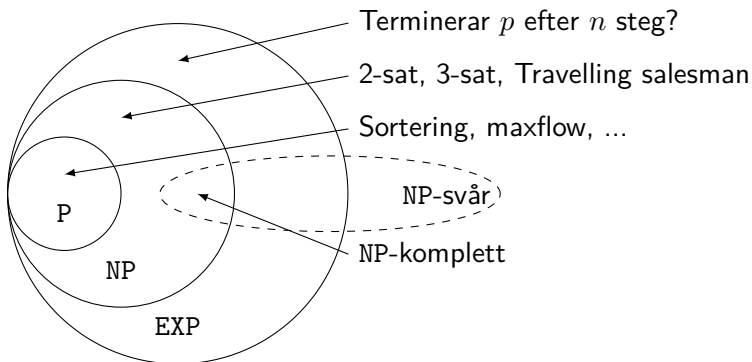
# Komplexitetsklasser

Olika problem är olika “svåra”:



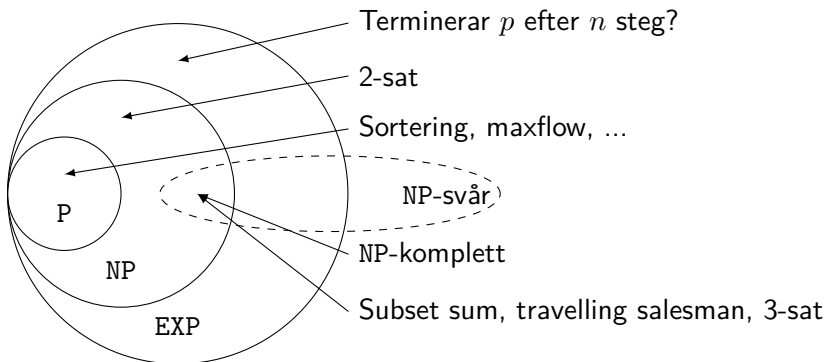
# Komplexitetsklasser

Olika problem är olika “svåra”:



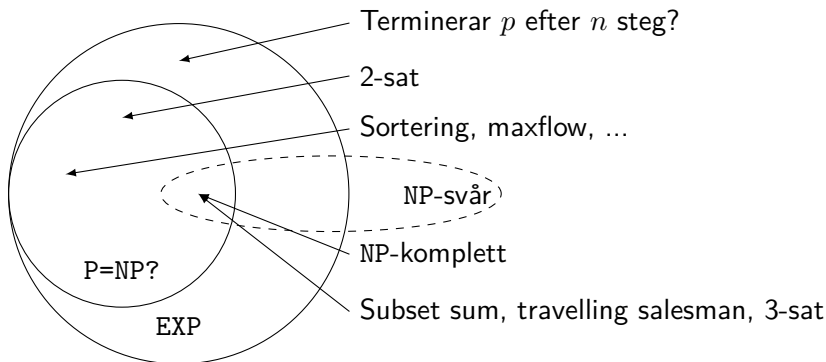
# Komplexitetsklasser

Olika problem är olika “svåra”:



# Komplexitetsklasser

Olika problem är olika “svåra”:



- 1 Hur snabbt kan vi sortera data?
- 2 Sortering i linjär tid
- 3 Tillämpningar
- 4 "Intressanta" algoritmer
- 5 Beräkningsbarhet
- 6 Sammanfattning

# I kursen framöver

- Nästa vecka
  - Tentaförberedelse
- Extrauppgifter
  - 11991 (enklare)  
Hitta tal effektivt. Kan vi använda sortering för att effektivisera lösningen?
  - 10369 (svårare)  
Anslut baser med radiokommunikation, minimera hur kraftig radiolänk som behövs. Kan binärsökning förenkla problemet?

## Påbyggnadskurser (vissa på masternivå)

TDIU16 Process- och operativsystem

Multitrådning och operativsystem

TDDD95 Algoritmisk problemlösning

Bygg ditt eget bibliotek med användbara  
algoritmer för att lösa Kattisproblem

TDDC78 Programmering av paralleldatorer

Analysera parallella algoritmer, programmera  
superdatorer

TDDD56 Multicore- och GPU-programmering

Programmera och analysera parallella  
algoritmer, delvis på GPU



Filip Strömbäck

[www.liu.se](http://www.liu.se)