

# **Datalogi C + Datastrukturer og Algoritmer**

Velkommen til DatC'erne

Dagens emne: Hvad er D&A, mål for effektivitet

Kursuslærer: *Henning Christiansen*

*henning@ruc.dk*, <http://www.ruc.dk/~henning>

Hjælpelærer på DatC: *Lars Mogensen*, [lamo@ruc.dk](mailto:lamo@ruc.dk)

Kursushjemmeside: [http://www.ruc.dk/~henning/DatC\\_DsAlg/kursusplan.html](http://www.ruc.dk/~henning/DatC_DsAlg/kursusplan.html)

# Dagens program

- Kort intro til Dat C'erne
- Datastrukturer & Algoritmer?
- Kompleksitet: Mål for tids- (og plads-) forbrug:
  - Teoretisk model
  - Eksempler

# Praktisk:

- Evaluering for Dat-OB'erne: Skriftlig eksamen
- Evaluering for DatC'erne: Afleveringsopgaver løbende
  - 80% skal godkendes; aflevering efter frist ikke mulig; genaflevering ikke mulig.
  - Opgaverne annonceres sammen med øvrige øvelsesopgaver på [www](http://www.ruc.dk/~henning/DatC_DsAlg/kursusplan.html)
- Lærebog: Weiss: Data Structures & problem Solving using Java
- Kom til øvelserne!!!!
  - DatC: 42.1 (Datalogi), fredage kl.13, første gang 17/9
  - Dat-OB: 43.2 (Usability Lab.), tirsdage kl. 13, næste 24/9

Find detaljer på [http://www.ruc.dk/~henning/DatC\\_DsAlg/kursusplan.html](http://www.ruc.dk/~henning/DatC_DsAlg/kursusplan.html)

## **Kursusindhold:**

- at bruge ((Javas) OOP-) prog.sprogs-faciliteter til at strukturere
- klassiske datastrukturer og algoritmer
- metoder til at vurdere effektivitet...
- designe og impl. solide datastrukturer og algoritmer

### **Hvad vi *ikke* kommer ind på:**

- tråde og parallelle algoritmer
- formel semantik af programmeringssprog og verifikation
- systematisk afestning
- styring af store projekter (mange personer/grupper roder med kode og struktur samtidigt)

# Krav til programmer/programdele

- veldefineret formål og korrekthed
- robusthed og sikkerhed
- effektivitet, store datamængder
- nemt at vedligeholde
- kan genbruges

## Kan understøttes ved rigtig brug af objektorienterede begreber:

Læs kapitel 1-4; hvis ønske fra DatC'erne: korte intro til øvelserne (eller andet tidspunkt)

Kan ses stort set uafhængigt af det øvrige stof

# Opgørelse af effektivitet, eksempel

$a=a+1$

*Tid = 1 (af en slags)*

```
for(int i=1; i<=n; i++)
```

$a=a+1;$

*Tid = n*

```
for(int i=1; i<=n; i++)
```

```
for(int j=1; j<=m; j++)
```

$a=a+1;$

*Tid =  $n*m$*

```
for(int i=1; i<=n; i++)
```

```
for(int j=1; j<=n; j++)
```

```
for(int k=1; j<=n; k++)
```

$a=a+1;$

*Tid =  $n^3$*

## Generelt:

- funktion af input ( $n, m$ )

## Mere kompliceret ved:

- while-løkker
- if(betingelse) omkring indre løkke
- indre løkke afh. af ydre

# Matematik til at karakterisere effektivitet

- Normalt ikke interesseret i eksakt tal, men asymptotisk opførsel (dvs.  $n \rightarrow \infty$ )
- Hvis en funktion  $T(n)$  står for eksakt tidsforbrug, bruges  $O(F(n))$  til at karakterisere øvre mål for "opførsel"

**Definition** (alternativ til bogen):

$T(n)$  er  $O(F(n))$  hviss  $T(n)/F(N) \leq \text{konstant}$   
når  $n \rightarrow \infty$

# Illustration af def. af $O(-)$

```
init; // tid = 25
for(i=1;i<=n;i++) // tid = n*17
  noget;
for(i=1;i<=n;i++) // tid = n^2*3
  for(j=1;j<=n;j++)
    noget_andet;
```

Total tidsforbrug

$$T(n) = 25 + n*17 + n^2*3$$

Påstand:  $T(n)$  er  $O(n^2)$

Udtales også:

"algoritmen er af orden  $O(n^2)$ "

"algoritmen er kvadratisk"

**Bevis:**  $T(n)/F(n) = T(n)/n^2$

$$= 25/n^2 + n*17/n^2 + n^2*3/n^2$$

$$= 25/n^2 + 17/n + 3$$

$$\square \quad 3 \text{ når } n \square \infty$$

**Generelt:**

- dominerende led bestemmer  $O$
- konstant uinteressant til generel karakteristik
- størrelsesorden god til estimerer (eksempel...)
- snyder ved små  $n$

# Eksempel: estimat fra $n=1000$ til $n=10000$

$$T(n) = 25 + n \cdot 17 + n^2 \cdot 3 \quad \text{og} \quad F(n) = n^2$$

$$T(10\,000)/T(n) \approx F(10\,000)/F(n), \text{ dvs. } T(10\,000) \approx 100 \cdot T(n)$$

Antag  $T(1000) = 10$  sek,

dvs.  $T(1000) = 3.001.725$  t (hvor t er en passende brøkdelen af et sek.)

Eksakt værdi for  $T(10.000) = 300.170.025$  t

Estimeret ved " $\cdot 100$ ":  $T(10.000) = 100$ sek

$T(10.000)$  omregnet fra "t" til sek:  $300.170.025/3.001.725 \cdot 10$  sek = **99,99 sek**

Fejl  $\approx$  **0,1%**

# Regneregler om O

**Definition:**  $F(n)$  dominerer over  $G(n)$ ,  $F(n) \gg G(n)$  såfremt

$$G(n)/F(n) \square 0 \text{ når } n \square \infty$$

**I så fald:**  $O(F(n)+G(n)) = O(F(n))$

hvor  $O(H(n)) = O(J(n))$  betyder  $H(n)/J(n) \square c \gg 0$  når  $n \square \infty$

**Eksempel:**  $O(25 + n*17 + n^2*3) = O(n^2)$

**Eksempler på regler:**

$$O(c*F(n)) = O(F(n))$$

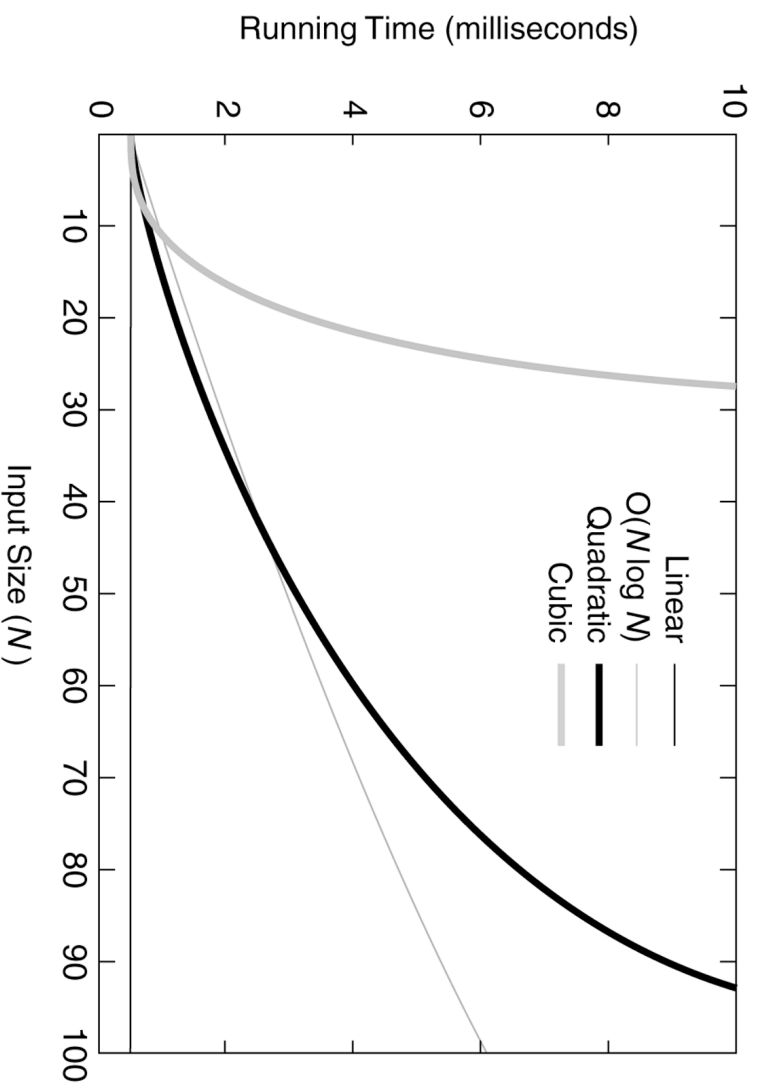
$$O(n*F(n)) \gg O(F(n))$$

$$O(V(n)*F(n)) \gg O(F(n)) \text{ såfremt } O(V(n)) \gg O(1)$$

# Ofte forekommende størrelsesordner

$$O(1) < O(\log n) < O((\log n)^2) < O(n) < O(n \log n)$$

$$< O(n^2) < O(n^3) < \dots < O(2^n)$$



## Små drilske led:

$$O(n + 2^n/1.000.000)$$

n	T
10	10
20	21
30	1.104
40	1.099.541

Hvem # kan finde den %φ"} lille tidrøver inden afleveringsprøven i morgen formiddag?

## Små drilske led, med endnu mindre konstant:

$$O(n + 2^n/1.000.000.000)$$

n	T
10	10
20	20
30	31
40	$\approx 1.000$
50	$\approx 1.000.000$

## Eksempler på polynomielle alg. $n^3$ , $n^2$ , $n$

Find maksimal sum af delsekvens

Eksempel:

-2	<b>11</b>	<b>-4</b>	<b>13</b>	-5	2
----	-----------	-----------	-----------	----	---

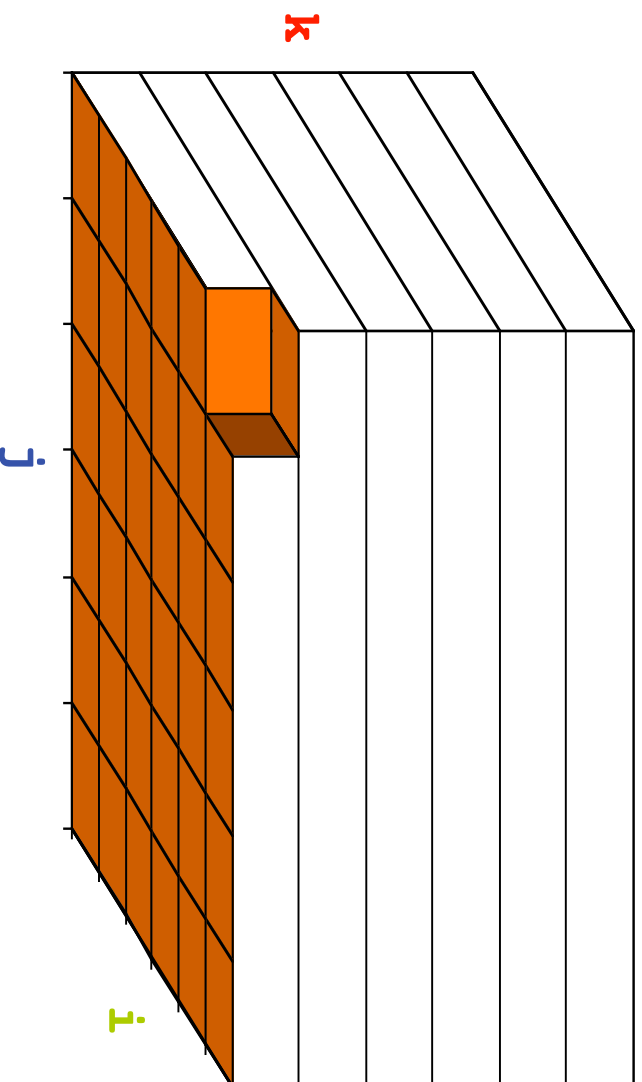
Den enkle algoritme: Generer samtlige mulige summer og hold rede på den hidtil største.

"Indlysende" kubisk, dvs.  $n^3$

# Kubisk algoritme for max-sum-delsekvens

```
maxSum = 0;
for(i=1; i<=n; i++) {
  for(j=i; j<=n; j++) {
    sum = 0;
    for(k=i; k<=j; k++)
      sum += a[k];
    if (sum>maxSum)
      maxSum = sum;
  }
}
```

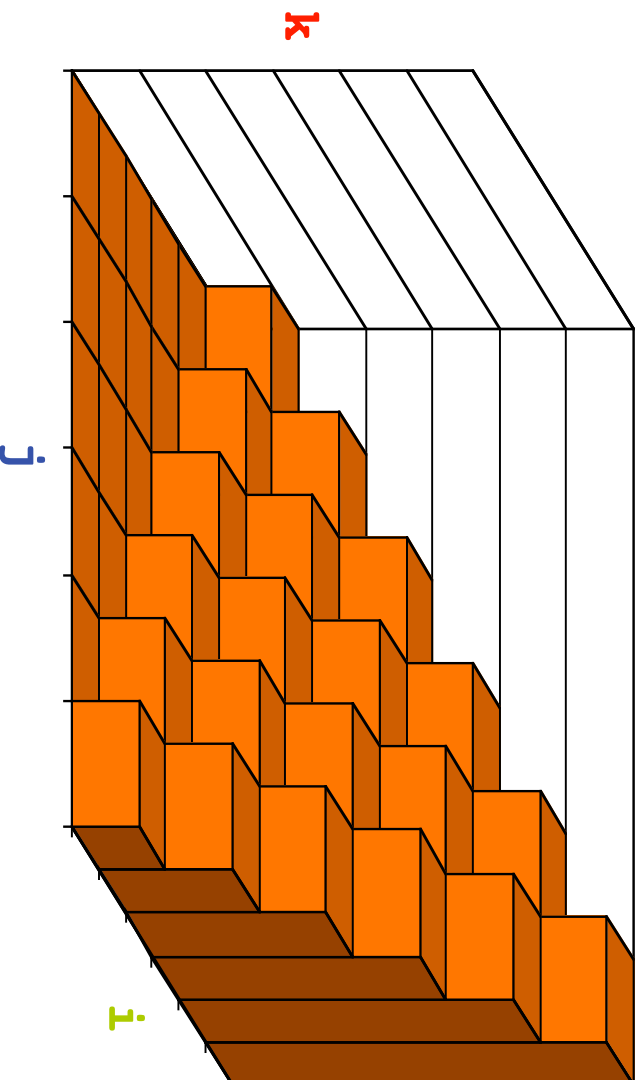
<b>k</b>	□					
<b>j</b>	□					
<b>i</b>	□					
	-2	11	-4	13	-5	2



# Kubisk algoritme for max-sum-delsekvens

```
maxSum = 0;  
for (i=1; i<=n; i++) {  
    for (j=i; j<=n; j++) {  
        sum = 0;  
        for (k=i; k<=j; k++)  
            sum += a[k];  
        if (sum>maxSum)  
            maxSum = sum;    }  
}
```

<b>k</b>							□
<b>j</b>							□
<b>i</b>							□
	-2	11	-4	13	-5		2



$$1 + (1+2) + (1+2+3) + \dots$$
$$+ (1+2+\dots+n)$$
$$= n * (n+1) * (n+2) / 6$$

**dvs.  $O(n^3)$**

# Kvadratisk algoritme for max-sum-delsekvens

Udnyt at “næste sum” = “gammel sum” + “næste led”

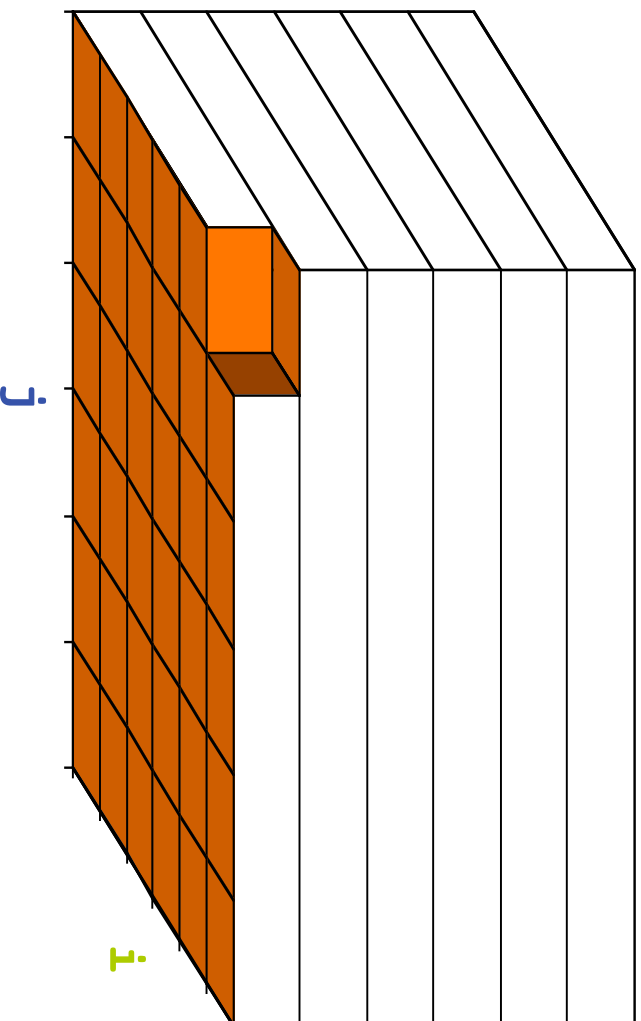
Dvs.  $Sum_{i,j+1} = Sum_{i,j} + a[j+1]$

```
maxSum = 0;
for (i=1; i<=n; i++) {
    sum = 0;
    for (j=i; j<=n; j++) {
        sum += a[j];
        if (sum>maxSum)
            maxSum = sum;
    }
}
```

# Kvadratisk algoritme for max-sum-delsekvens

```
maxSum = 0;
for (i=1; i<=n; i++) {
    sum = 0;
    for (j=i; j<=n; j++) {
        sum += a[j];
        if (sum>maxSum)
            maxSum = sum;
    }
}
```

j	□					
i	□					
	-2	11	-4	13	-5	2

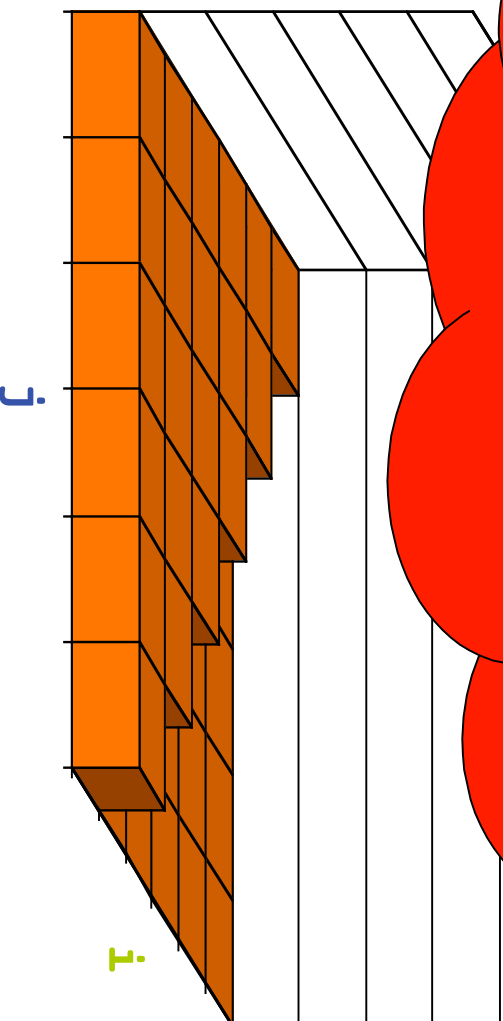


# Kvadratisk algoritme for max-sum-delsekvens

```
maxSum = 0;  
for (i=1; i<=n; i++) {  
    sum = 0;  
    for (j=i; j<=n; j++) {  
        sum += arr[i][j];  
    }  
    maxSum = max(maxSum, sum);  
}
```

j						
i	1	2	3	4	5	6
1						□
2						□
3						□
4						□
5						□
6						□

Læs selv om lineær algoritme i bog:  
Eksempel på at optimerede algoritmer  
kan være svært gennemskuelig!



$$1 + 2 + \dots + n$$
$$= n * (n-1) / 2$$

altså  $O(n^2)$

# Logaritmisk, typisk for del-og-hersk algoritmer

Eksempel: Binær søgning for at finde element i sorteret sekvens  
~ a la telefonbog

Princippet:

- at *finde*  $x$  i sekvens af lgd. 1: test " $=x$ " ja/nej
- ellers, hvis  $x \leq$  midt-element, så *find*  $x$  i venstre halvdel  
ellers *find*  $x$  i højre halvdel

Tidsforbrug

- hvert skridt (" $\bullet$ " ovenfor) konstant
- hvert skridt halverer længden af sekvensen

Total tid for lgd.  $n$ : så mange gange  $n$  skal halveres for at blive "1"

$$\approx \log_2 n$$



Hvordan var det nu?

# Lidt om logaritmefunktionen

Defineret som omvendt til eksponentialfunktionen:

$$\text{hvis } 2^x = m \text{ så } \log_2 m = x$$

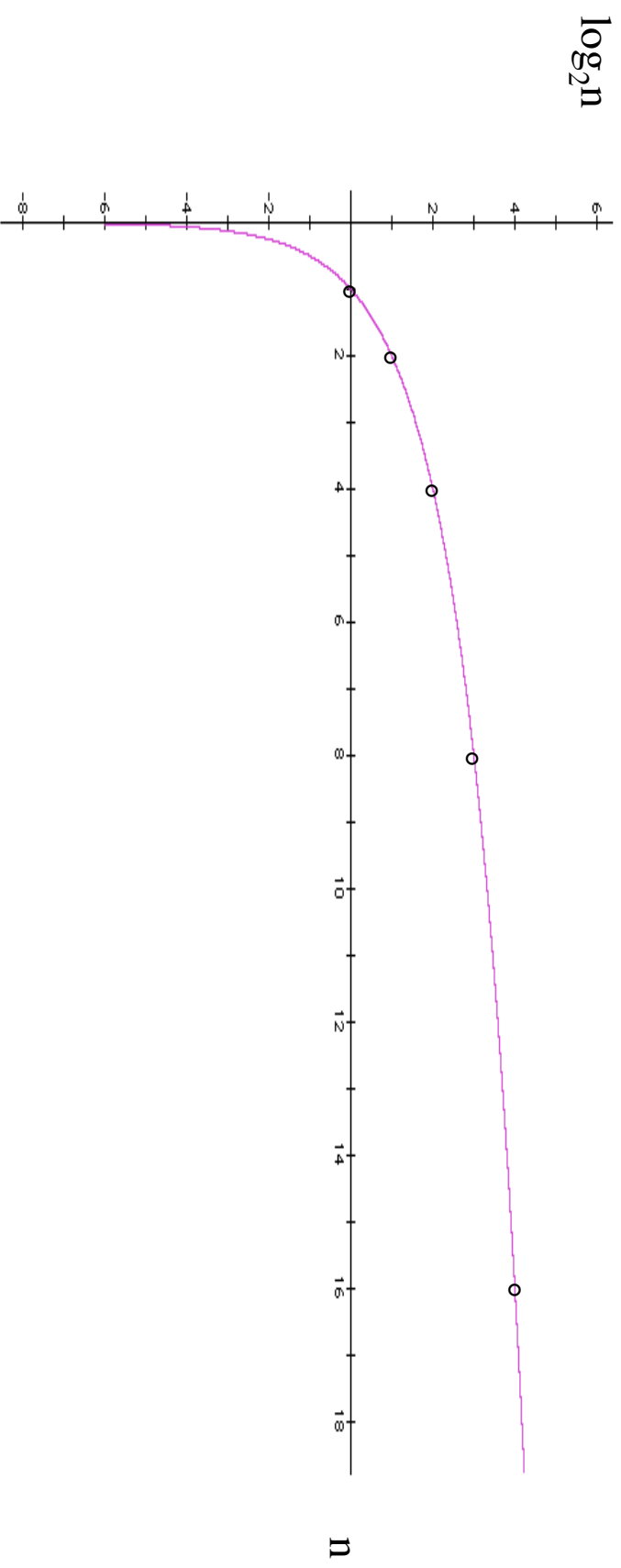
Eksempler:

n	binært	$\log_2 n$	Halvinger
1	1	0	-
2	10	1	□ 1
4	100	2	□ 2 □ 1
8	1000	3	□ 4 □ 2 □ 1
16	10000	4	□ 8 □ 4 □ 2 □ 1

Egenskaber:

- fordobles argumentet stiger logaritmen med 1
- $\log_2 n \approx$  antal tegn i n's binære repræsentation
- $\log_2 n \approx$  antal gange n skal halveres for at blive 1

# Funktionskurven für $\log_2 n$

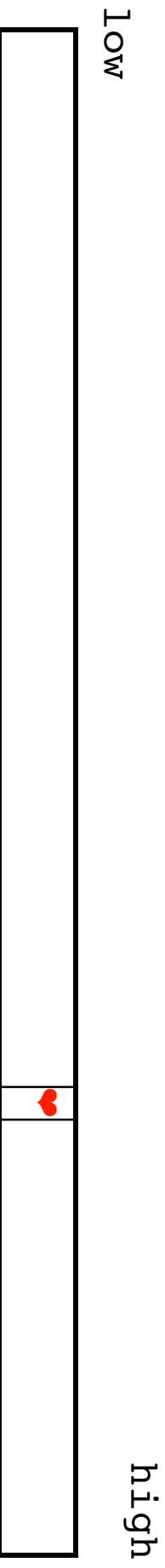


# Binær søgning i Java I: Rekursiv for "int []"

```
Kaldes: binarySearch(a, x, 0, a.length-1)
int binarySearch(int low, int high, Comparable[] a, Comparable x) {
    int mid;
    if (low > high) return -1;
    mid = (low + high) / 2
    if ( a[mid] < x )
        binarySearch(mid+1, high)
    else if ( a[mid] > x )
        binarySearch(low, mid-1);
    else return mid; } } // a[mid] == x
```

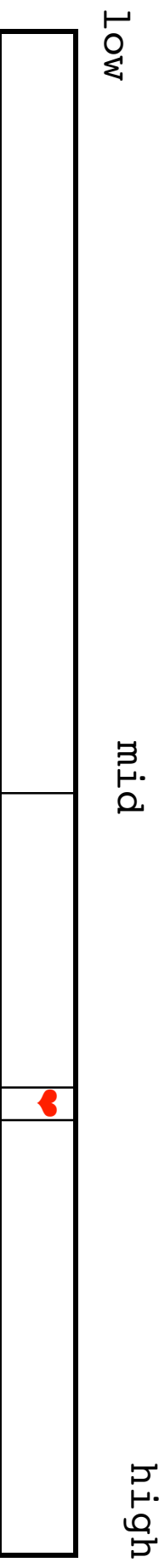
# Binær søgning i Java I: Rekursiv for "int []"

```
Kaldes: binarySearch(a, x, 0, a.length-1)
int binarySearch(int low, int high, Comparable[] a, Comparable x) {
    int mid;
    if (low > high) return -1;
    mid = (low + high) / 2
    if ( a[mid] < x )
        binarySearch(mid+1, high)
    else if ( a[mid] > x )
        binarySearch(low, mid-1);
    else return mid; } // a[mid] == x
```



# Binær søgning i Java I: Rekursiv for "int []"

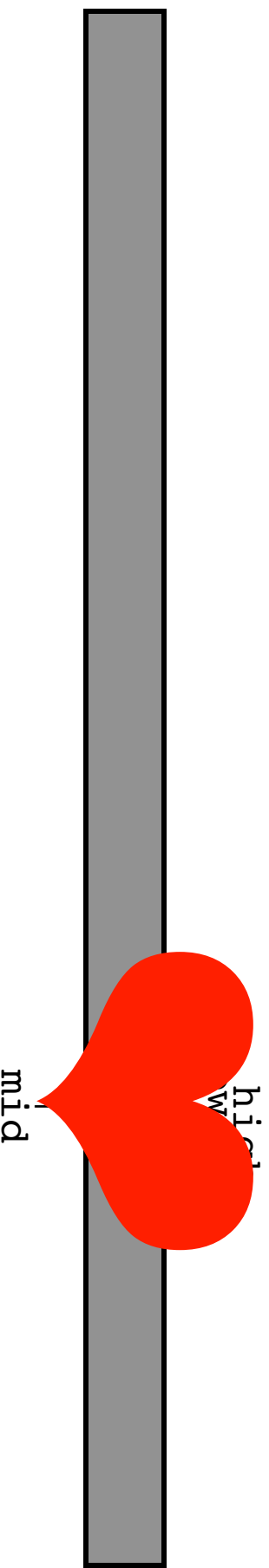
```
Kaldes: binarySearch(a, x, 0, a.length-1)
int binarySearch(int low, int high, Comparable[] a, Comparable x) {
    int mid;
    if (low > high) return -1;
    mid = (low + high) / 2
    if ( a[mid] < x )
        binarySearch(mid+1, high)
    else if ( a[mid] > x )
        binarySearch(low, mid-1);
    else return mid; } // a[mid] == x
```



# Binær søgning i Java I: Rekursiv for "int []"

**Kaldes:** `binarySearch(a, x, 0, a.length-1)`

```
int binarySearch(int low, int high, Comparable[] a, Comparable x) {
    int mid;
    if (low > high) return -1;
    mid = (low + high) / 2
    if ( a[mid] < x )
        binarySearch(mid + 1, high)
    else if ( a[mid] > x )
        binarySearch(low, mid - 1);
    else return mid; } // a[mid] == x
```



# Binær søgning i Java II: Rekursiv, generisk

```
package java.lang;

public interface Comparable {in compareTo(Object other)};

public static int binarySearch(Comparable [] a, Comparable x){
    binarySearch(0,a.length-1,a,x)}

public static int binarySearch1(int low,int high,
                               Comparable [] a, Comparable x){
    int mid;
    if(low>high) return -1;
    mid = (low+high)/2
    if(a[mid].compareTo(x)<0)
        binarySearch1(mid+1,high)
    else if(a[mid].compareTo(x)>0)
        binarySearch1(low,mid-1);
    else return mid; }}
```

```
class Elephant {...; public
int compareTo(...){...} }

zoo = new Elephant []{...};
dumbo = new Elephant;
int where_is_dumbo =
binarySearch(zoo,dumbo);
```

# Binær søgning i Java III: Iterativ, generisk

Lærebogens version:

```
int binarySearch(Comparable[] a,
int low = 0;
int high = a.length-1;
int mid;
while ( low<=high){
    mid = (low+high)/2;
    if(a[mid].compareTo(x)<0)
        low = mid+1;
    else if(a[mid].compareTo(x)
        high = mid-1;
    else return mid; }
return -1; }
```

Iterativ vs. rekursiv:  
En smagssag, men rekursive algs. ofte simple for af-natur-rek. problemer

Hvad er mest effektivt?  
Er rekursion ikke meget dyrt?

Det afhænger af compileren!  
Med en god Java compiler, identiske køretider!

# Afsluttende om $O$ -notation

- udvide med flere parametre, f.eks.  $O(m \cdot 2^n)$
- $O$ -notation benyttes også for pladsforbrug
  - ofte trade-off plads- vs. tidskompleksitet
- Algoritmer har bedste, værste og gennemsnitligt tidsforbrug
  - f.eks. quicksort, værst  $O(n^2)$ , gennemsnit  $O(n \log n)$
- " $T(n)$  er  $O(F(n))$ " angiver en overgrænse for  $T(n)$ 's asymptotiske opførsel; alternative karakteristikker:
  - " $T(n)$  er  $\square(F(n))$ " angiver undergrænse
  - " $T(n)$  er  $\square(F(n))$ " angiver eksakt karakteristik
  - " $T(n)$  er  $o(F(n))$ " angiver karakteristik som er klart for høj