DD1331

Grundläggande programmering för F1

Några bilder till föreläsning 5

Innehåll

- Rekursion – viktig metodik i datalogin

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion
  - Flera små exempel

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion
  - Flera små exempel
  - Hur går det till?

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion
  - Flera små exempel
  - Hur går det till?
  - Inte alltid bra!

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion
  - Flera små exempel
  - Hur går det till?
  - Inte alltid bra!
- Algoritmer och algoritmanalys

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion
  - Flera små exempel
  - Hur går det till?
  - Inte alltid bra!
- Algoritmer och algoritmanalys
  - Introduktion
  - $\mathcal{O}$ – notationen

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion
  - Flera små exempel
  - Hur går det till?
  - Inte alltid bra!
- Algoritmer och algoritmanalys
  - Introduktion
  - $\mathcal{O}$ – notationen
- Två enkla sökalgoritmer

Innehåll

- Rekursion – viktig metodik i datalogin
  - Introduktion
  - Flera små exempel
  - Hur går det till?
  - Inte alltid bra!
- Algoritmer och algoritmanalys
  - Introduktion
  - $\mathcal{O}$ – notationen
- Två enkla sökalgoritmer
  - Linjärsökning
  - Binärsökning

Rekursion

Rekursion (självreferens)

Rekursion (självreferens)

Rekursiv tanke:

Basfall:

Rekursion (självreferens)

Rekursiv tanke:

Formulera lösningen med hjälp av
lösningen till ett eller flera likadana men mindre problem

Basfall:

Rekursion (självreferens)

Rekursiv tanke:

Formulera lösningen med hjälp av
lösningen till ett eller flera likadana men mindre problem

Basfall:

Det måste finnas minst ett fall där problemet
har en icke-rekursiv lösning.

Rekursionsexempel: Rita trianglar (som i labb 2)

Rekursionsexempel: Rita trianglar (som i labb 2)

```
***********
 *********
  *******
   *****
    ***
     *
```

triangelBASUPP(avst, bredd)

6 11

Rekursionsexempel: Rita trianglar (som i labb 2)

```
***********
 *********
  *******
   *****
    ***
     *
```

triangelBASUPP(avst, bredd)

6 11

```
     *
    ***
   *****
  *******
 *********
```

triangelBASNER(avst, bredd)

2 9

triangelBASUPP(avst, bredd)

triangelBASUPP(avst, bredd)

```
*********
 *******
  *****
   ***
    *
```

triangelBASUPP(avst, bredd)

```
*********
 *******
  *****
   ***
    *
```

Rekursiv tanke:

Basfall:

triangelBASUPP(avst, bredd)

```
*********
 *******
  *****
   ***
    *
```

Rekursiv tanke:

Rita **bredd** st * , **avst** från kanten

Basfall:

triangelBASUPP(avst, bredd)

```
*********
 *******
  *****
   ***
    *
```

Rekursiv tanke:

Rita **bredd** st * , **avst** från kanten

Gör **triangelBASUPP(avst + 1, bredd-2)**

Basfall:

triangelBASUPP(avst, bredd)

```
*********
 *******
  *****
   ***
    *
```

Rekursiv tanke:

Rita **bredd** st * , **avst** från kanten

Gör **triangelBASUPP(avst + 1, bredd-2)**

Basfall:

Om **bredd == 1**

triangelBASUPP(avst, bredd)

```
*********
 *******
  *****
   ***
    *
```

Rekursiv tanke:

Rita **bredd** st * , **avst** från kanten

Gör **triangelBASUPP(avst + 1, bredd-2)**

Basfall:

Om **bredd == 1**

Rita en * , **avst** från kanten

triangelBASUPP(avst, bredd)

```
*********
 *******
  *****
   ***
    *
```

Rekursiv tanke:

Rita **bredd** st * , **avst** från kanten

Gör **triangelBASUPP(avst + 1, bredd-2)**

Basfall:

Om **bredd == 1**

Rita en * , **avst** från kanten

Alternativt basfall: Om **bredd** $<$ **1** , gör ingenting

triangelBASNER(avst, bredd)

triangelBASNER(avst, bredd)

```
    *
   ***
  *****
 *******
*********
```

triangelBASNER(avst, bredd)

```
    *
   ***
  *****
 *******
*********
```

Rekursiv tanke:

Basfall:

triangelBASNER(avst, bredd)

```
    *
   ***
  *****
 *******
*********
```

Rekursiv tanke:

Gör **triangelBASNER(avst + 1, bredd-2)**

Basfall:

triangelBASNER(avst, bredd)

```
    *
   ***
  *****
 *******
*********
```

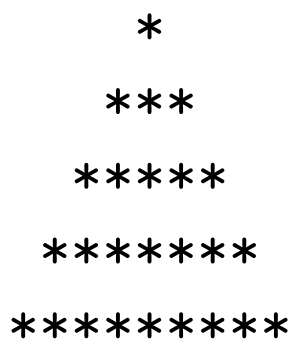
Rekursiv tanke:

Gör **triangelBASNER(avst + 1, bredd-2)**

Rita **bredd** st * , **avst** från kanten

Basfall:

triangelBASNER(avst, bredd)



Rekursiv tanke:

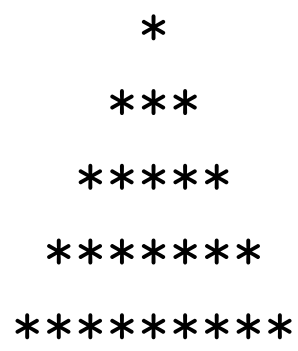
Gör **triangelBASNER(avst + 1, bredd-2)**

Rita **bredd** st * , **avst** från kanten

Basfall:

Om **bredd == 1**

triangelBASNER(avst, bredd)



Rekursiv tanke:

Gör **triangelBASNER(avst + 1, bredd-2)**

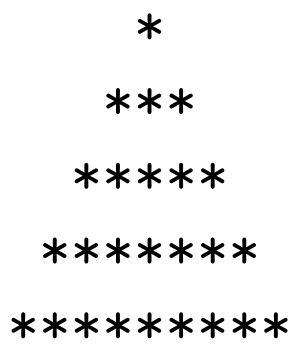
Rita **bredd** st * , **avst** från kanten

Basfall:

Om **bredd == 1**

Rita en * , **avst** från kanten

triangelBASNER(avst, bredd)



Rekursiv tanke:

Gör **triangelBASNER(avst + 1, bredd-2)**

Rita **bredd** st * , **avst** från kanten

Basfall:

Om **bredd == 1**

Rita en * , **avst** från kanten

Alternativt basfall: Om **bredd** $<$ **1** , gör ingenting

Skriv de rekursiva funktionerna själv!

Skriv de rekursiva funktionerna själv!

För svårt?

Skriv de rekursiva funktionerna själv!

För svårt?

Gå på veckans övning!

Där ges fullständig förklaring och Pythonkod.

Nedräkningsexemplet med vanlig funktion

Nedräkningsexemplet med vanlig funktion

```
import time

def nedrakning(n):
    tid = n
    while tid > 0:
        print(tid)
        time.sleep(1)
        tid -=1
    print("0 and GOOOOOOO")

nedrakning(8)
```

Nedräkning med rekursion:

Nedräkning med rekursion:

Rekursiv tanke:

Basfall:

Nedräkning med rekursion:

Rekursiv tanke:

Basfall:

Nedräkning från 0:

Nedräkning med rekursion:

Rekursiv tanke:

Basfall:

Nedräkning från 0:

Skriv ut `0 and GOOOOOOO`

Nedräkning med rekursion:

Rekursiv tanke:

Nedräkning från n, om > 0:

Basfall:

Nedräkning från 0:

Skriv ut `0 and GOOOOOOO`

Nedräkning med rekursion:

Rekursiv tanke:

Nedräkning från n, om > 0:

Skriv ut n

Basfall:

Nedräkning från 0:

Skriv ut `0 and GOOOOOOO`

Nedräkning med rekursion:

Rekursiv tanke:

Nedräkning från n, om > 0:

Skriv ut n

Vänta en sekund

Basfall:

Nedräkning från 0:

Skriv ut `0 and GOOOOOOO`

Nedräkning med rekursion:

Rekursiv tanke:

Nedräkning från n, om > 0:

- Skriv ut n
- Vänta en sekund
- Gör nedräkning från n-1

Basfall:

Nedräkning från 0:

- Skriv ut `0 and GOOOOOOO`

Rekursiva nedräkningsfunktionen med anrop

Rekursiva nedräkningsfunktionen med anrop

```
import time

def nedrakningR(n):
    if n==0:                                # basfall
        print('0 and GOOOOOOO')
        return
    print(n, '\tR')
    time.sleep(1)
    nedrakningR(n-1)                        # rekursion

# anropsexempel
nedrakningR(10)
```

Beräkna n! med iteration

Beräkna n! med iteration

```
def fak(n):
    res = 1
    for k in range(1, n+1):
        res = res*k     # res = 1*2*3*...*n
    return res

# anropa fak for nagra n
for n in [3, 7, 11]:
    print(str(n)+"! = ", fak(n))
```

Utskrift:

```
3! = 6
7! = 5040
11! = 39916800
```

Rekursion som returnerar värde

Rekursion som returnerar värde

Beräkna n! rekursivt:

Rekursion som returnerar värde

Beräkna n! rekursivt:

- **Rekursiv tanke:**

Rekursion som returnerar värde

Beräkna n! rekursivt:

- **Rekursiv tanke:**
  För $n > 0$ definieras n!:

Rekursion som returnerar värde

Beräkna n! rekursivt:

- **Rekursiv tanke:**

  För n $>$ 0 definieras n!:

  n $*$ (n-1)!

Rekursion som returnerar värde

Beräkna n! rekursivt:

- **Rekursiv tanke:**
  För n $>$ 0 definieras n!:
  n $*$ (n-1)!
- **Basfall:**

Rekursion som returnerar värde

Beräkna n! rekursivt:

- **Rekursiv tanke:**
  För n $>$ 0 definieras n!:
  n $*$ (n-1)!
- **Basfall:**
  För n=0 och n=1 definieras n!:

Rekursion som returnerar värde

Beräkna n! rekursivt:

- **Rekursiv tanke:**
  För n $>$ 0 definieras n!:
  n $*$ (n-1)!
- **Basfall:**
  För n=0 och n=1 definieras n!:
  1

Rekursion som returnerar värde

Beräkna n! rekursivt:

- **Rekursiv tanke:**
  För n $>$ 0 definieras n!:
  n $*$ (n-1)!
- **Basfall:**
  För n=0 och n=1 definieras n!:
  1

(vi struntar i n $<$ 0)

Rekursiva funktionen:

```
def fakR(n):
    if n <= 1:
        return 1
    return n * fakR(n-1)
```

Rekursiva funktionen:

```
def fakR(n):
    if n <= 1:
        return 1
    return n * fakR(n-1)
```

Anropsexempel:

```
for n in [4, 9, 13]:
    print(str(n)+"! = ", fakR(n))
```

Rekursiva funktionen:

```
def fakR(n):
    if n <= 1:
        return 1
    return n * fakR(n-1)
```

Anropsexempel:

```
for n in [4, 9, 13]:
    print(str(n)+"! = ", fakR(n))
```

Utskrift:

4! = 24

9! = 362880

13! = 6227020800

Hur kan rekursion fungera?

Hur kan rekursion fungera?

Aktiveringsposter

Hur kan rekursion fungera?

Aktiveringsposter

- Varje funktionsanrop får en aktiveringspost på stacken i minnet

Hur kan rekursion fungera?

Aktiveringsposter

- Varje funktionsanrop får en aktiveringspost på stacken i minnet
- Där lagras parametrar och lokala variabler för varje anrop

Hur kan rekursion fungera?

Aktiveringsposter

- Varje funktionsanrop får en aktiveringspost på stacken i minnet
- Där lagras parametrar och lokala variabler för varje anrop
- När funktionsanropet exekverat klart återlämnas minnet

Hur kan rekursion fungera?

Aktiveringsposter

- Varje funktionsanrop får en aktiveringspost på stacken i minnet
- Där lagras parametrar och lokala variabler för varje anrop
- När funktionsanropet exekverat klart återlämnas minnet

Aktiveringsposter för fakR(4)

Hur kan rekursion fungera?

Aktiveringsposter

- Varje funktionsanrop får en aktiveringspost på stacken i minnet
- Där lagras parametrar och lokala variabler för varje anrop
- När funktionsanropet exekverat klart återlämnas minnet

Aktiveringsposter för fakR(4)

`fakR(4)`	`fakR(3)`	`fakR(2)`	`fakR(1)`
n: 4	n: 3	n: 2	n: 1
			`ret 1`
`ret 4*fakR(3)`	`ret 3*fakR(2)`	`ret 2*fakR(1)`	basfall

fakR(4)	fakR(3)	fakR(2)	fakR(1)
n: 4	n: 3	n: 2	n: 1 ret 1
ret 4*fakR(3)	ret 3*fakR(2)	ret 2*fakR(1)	basfall

rekursionen vänder, minnet återlämnas

fakR(4)	fakR(3)	fakR(2)	fakR(1)
n: 4	n: 3	n: 2	n: 1
			ret 1
ret 4*fakR(3)	ret 3*fakR(2)	ret 2*fakR(1)	basfall

rekursionen vänder, minnet återlämnas

fakR(4)	fakR(3)	fakR(2)
n: 4	n: 3	n: 2
ret 4*fakR(3)	ret 3*fakR(2)	ret 2*1

fakR(4)	fakR(3)	fakR(2)	fakR(1)
n: 4	n: 3	n: 2	n: 1
			ret 1
ret 4*fakR(3)	ret 3*fakR(2)	ret 2*fakR(1)	basfall

rekursionen vänder, minnet återlämnas

fakR(4)	fakR(3)	fakR(2)
n: 4	n: 3	n: 2
ret 4*fakR(3)	ret 3*fakR(2)	ret 2*1

fakR(4)	fakR(3)
n: 4	n: 3
ret 4*fakR(3)	ret 3*2

fakR(4)	fakR(3)	fakR(2)	fakR(1)
n: 4	n: 3	n: 2	n: 1
			ret 1
ret 4*fakR(3)	ret 3*fakR(2)	ret 2*fakR(1)	basfall

rekursionen vänder, minnet återlämnas

fakR(4)	fakR(3)	fakR(2)
n: 4	n: 3	n: 2
ret 4*fakR(3)	ret 3*fakR(2)	ret 2*1

fakR(4)	fakR(3)
n: 4	n: 3
ret 4*fakR(3)	ret 3*2

Till sist returneras $4*6 = 24$ som är svaret från fakR(4).

Konstruera binära representationen av ett heltal b, som textsträng:

Konstruera binära representationen av ett heltal b, som textsträng:

$$
\begin{array}{lcl}
0 & \rightarrow & \text{'0'} \\
1 & \rightarrow & \text{'1'} \\
8 & \rightarrow & \text{'1000'} \\
11 & \rightarrow & \text{'1011'} \\
1729 & \rightarrow & \text{'11011000001'}
\end{array}
$$

Konstruera binära representationen av ett heltal b, som textsträng:

$$
\begin{array}{lcl}
0 & \rightarrow & \text{'0'} \\
1 & \rightarrow & \text{'1'} \\
8 & \rightarrow & \text{'1000'} \\
11 & \rightarrow & \text{'1011'} \\
1729 & \rightarrow & \text{'11011000001'}
\end{array}
$$

Basfall:

Rekursiv tanke:

Konstruera binära representationen av ett heltal b, som textsträng:

$$
\begin{array}{lcl}
0 & \rightarrow & \text{'0'} \\
1 & \rightarrow & \text{'1'} \\
8 & \rightarrow & \text{'1000'} \\
11 & \rightarrow & \text{'1011'} \\
1729 & \rightarrow & \text{'11011000001'}
\end{array}
$$

Basfall:

Om b $=$ 0 eller b $=$ 1:

Rekursiv tanke:

Konstruera binära representationen av ett heltal b, som textsträng:

$$
\begin{array}{lcl}
0 & \rightarrow & \text{'0'} \\
1 & \rightarrow & \text{'1'} \\
8 & \rightarrow & \text{'1000'} \\
11 & \rightarrow & \text{'1011'} \\
1729 & \rightarrow & \text{'11011000001'}
\end{array}
$$

Basfall:

Om b $=$ 0 eller b $=$ 1:

returnera "0" respektive "1" `str(b)`

Rekursiv tanke:

Konstruera binära representationen av ett heltal b, som textsträng:

$$
\begin{array}{lcl}
0 & \rightarrow & \text{'0'} \\
1 & \rightarrow & \text{'1'} \\
8 & \rightarrow & \text{'1000'} \\
11 & \rightarrow & \text{'1011'} \\
1729 & \rightarrow & \text{'11011000001'}
\end{array}
$$

Basfall:

Om b $=$ 0 eller b $=$ 1:

returnera "0" respektive "1" `str(b)`

Rekursiv tanke:

Om b $>$ 1:

Konstruera binära representationen av ett heltal b, som textsträng:

$$
\begin{array}{lcl}
0 & \rightarrow & \text{'0'} \\
1 & \rightarrow & \text{'1'} \\
8 & \rightarrow & \text{'1000'} \\
11 & \rightarrow & \text{'1011'} \\
1729 & \rightarrow & \text{'11011000001'}
\end{array}
$$

Basfall:

Om b $=$ 0 eller b $=$ 1:

returnera "0" respektive "1" `str(b)`

Rekursiv tanke:

Om b $>$ 1:

Konstruera bin-repr för b//2 `binr(b//2)`

Konstruera binära representationen av ett heltal b, som textsträng:

0	$\rightarrow$	'0'
1	$\rightarrow$	'1'
8	$\rightarrow$	'1000'
11	$\rightarrow$	'1011'
1729	$\rightarrow$	'11011000001'

Basfall:

Om b $=$ 0 eller b $=$ 1:

returnera "0" respektive "1" `str(b)`

Rekursiv tanke:

Om b $>$ 1:

Konstruera bin-repr för b//2 `binr(b//2)`

Om b är udda, lägg till '1',
annars lägg till '0' `str(b%2)`

Konstruera binära representationen av ett heltal b, som textsträng:

0	$\rightarrow$	'0'
1	$\rightarrow$	'1'
8	$\rightarrow$	'1000'
11	$\rightarrow$	'1011'
1729	$\rightarrow$	'11011000001'

Basfall:

Om b $=$ 0 eller b $=$ 1:

returnera "0" respektive "1" `str(b)`

Rekursiv tanke:

Om b $>$ 1:

Konstruera bin-repr för b//2 `binr(b//2)`

Om b är udda, lägg till '1',
annars lägg till '0' `str(b%2)`

Pythonkoden kommer på kurshemsidan

Varning för rekursion!

Varning för rekursion!

Fibonacci–talen

Varning för rekursion!

Fibonacci–talen

Definition

Varning för rekursion!

Fibonacci–talen

Definition

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$ för $n \geq 2$

Varning för rekursion!

Fibonacci–talen

Definition

- $f_0 = 0$
- $f_1 = 1$
- $f_n = f_{n-1} + f_{n-2}$ för $n \geq 2$

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Elegant lösning ???

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

De **två** rekursiva anropen är orsak till katastrofen.

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

De **två** rekursiva anropen är orsak till katastrofen.

Kallas för **lavin** av funktionsanrop

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

De **två** rekursiva anropen är orsak till katastrofen.

Kallas för **lavin** av funktionsanrop

`fibo(5)` → 15 anrop

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

De **två** rekursiva anropen är orsak till katastrofen.

Kallas för **lavin** av funktionsanrop

`fibo(5)` → 15 anrop

`fibo(6)` → 25 anrop

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

De **två** rekursiva anropen är orsak till katastrofen.

Kallas för **lavin** av funktionsanrop

`fibo(5)` → 15 anrop
`fibo(6)` → 25 anrop
`fibo(7)` → 41 anrop

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

De **två** rekursiva anropen är orsak till katastrofen.

Kallas för **lavin** av funktionsanrop

`fibo(5)` → 15 anrop
`fibo(6)` → 25 anrop
`fibo(7)` → 41 anrop
`fibo(17)` → 5167 anrop

Elegant ?

```
def fibo(n):
    if n <=1:
        return n
    return fibo(n-1) + fibo(n-2)
```

Nej, katastrofalt resurskrävande lösning!

Exponentiell tillväxt av onödiga funktionsanrop!

De **två** rekursiva anropen är orsak till katastrofen.

Kallas för **lavin** av funktionsanrop

`fibo(5)` $\rightarrow$ 15 anrop
`fibo(6)` $\rightarrow$ 25 anrop
`fibo(7)` $\rightarrow$ 41 anrop
`fibo(17)` $\rightarrow$ 5167 anrop

Jämför: for-sats för f_{17} kräver 16 repetitioner

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

0 1 1 2 3 5 8 13 21 34 55 ...

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

0 1 1 2 3 5 8 13 21 34 55 ...

Här är det fel med algoritm där arbetet växer exponentiellt

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

0 1 1 2 3 5 8 13 21 34 55 ...

Här är det fel med algoritm där arbetet växer exponentiellt

Sierpinski-triangel:

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

0 1 1 2 3 5 8 13 21 34 55 ...

Här är det fel med algoritm där arbetet växer exponentiellt

Sierpinski-triangel:

Exempel på problem där arbetet faktiskt växer exponentiellt.

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

0 1 1 2 3 5 8 13 21 34 55 ...

Här är det fel med algoritm där arbetet växer exponentiellt

Sierpinski-triangel:

Exempel på problem där arbetet faktiskt växer exponentiellt.
Då är det rätt med flera rekursiva anrop.

Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

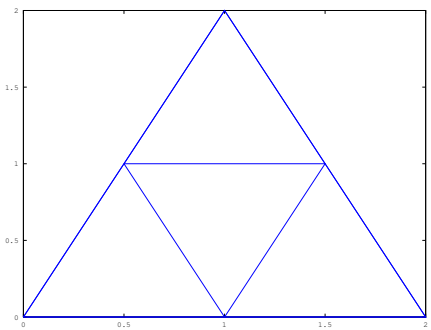
$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

0 1 1 2 3 5 8 13 21 34 55 ...

Här är det fel med algoritm där arbetet växer exponentiellt

Sierpinski-triangel:

Exempel på problem där arbetet faktiskt växer exponentiellt.
Då är det rätt med flera rekursiva anrop.



Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

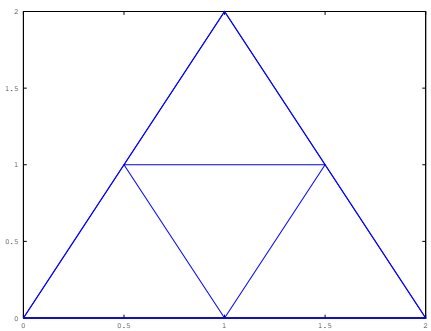
$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

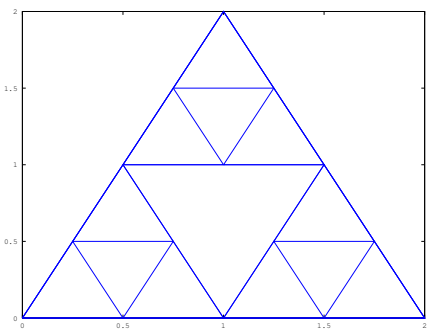
0 1 1 2 3 5 8 13 21 34 55 ...

Här är det fel med algoritm där arbetet växer exponentiellt

Sierpinski-triangel:

Exempel på problem där arbetet faktiskt växer exponentiellt.
Då är det rätt med flera rekursiva anrop.





Fibonnaci:

Problemet har en iterativ, linjär karaktär.

Från $f_0 = 0$, $f_1 = 1$ kan f_2, f_3, ... enkelt beräknas med

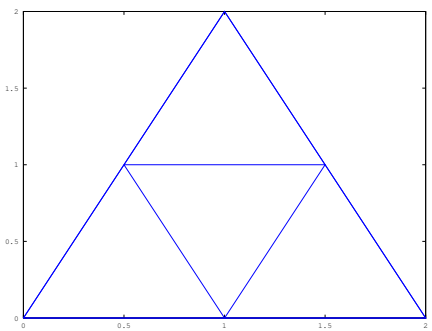
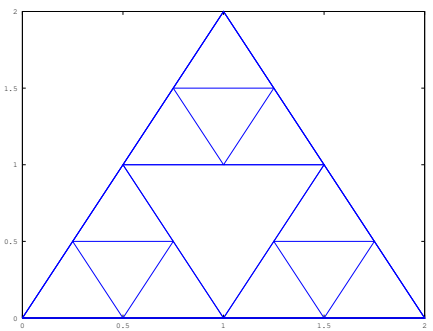
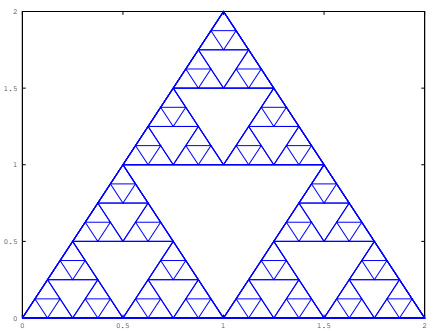
$f_n = f_{n-1} + f_{n-1}$ för n=2,3,...

0 1 1 2 3 5 8 13 21 34 55 ...

Här är det fel med algoritm där arbetet växer exponentiellt

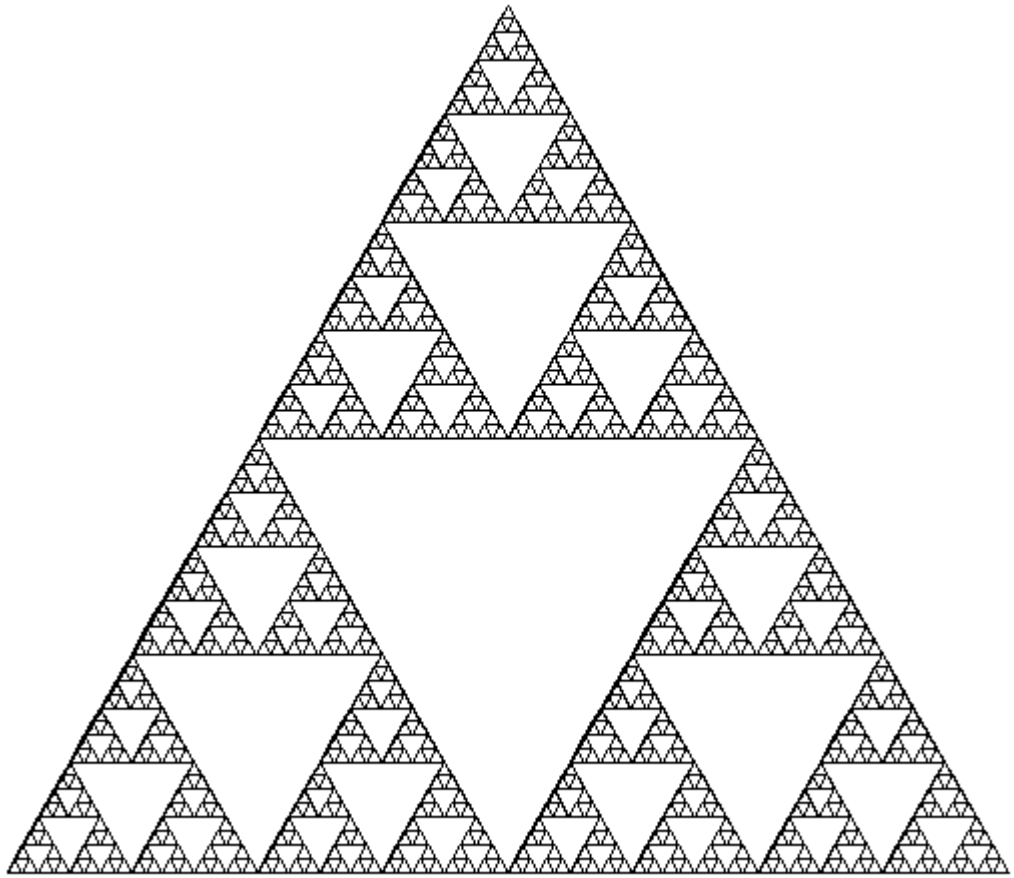
Sierpinski-triangel:

Exempel på problem där arbetet faktiskt växer exponentiellt.
Då är det rätt med flera rekursiva anrop.

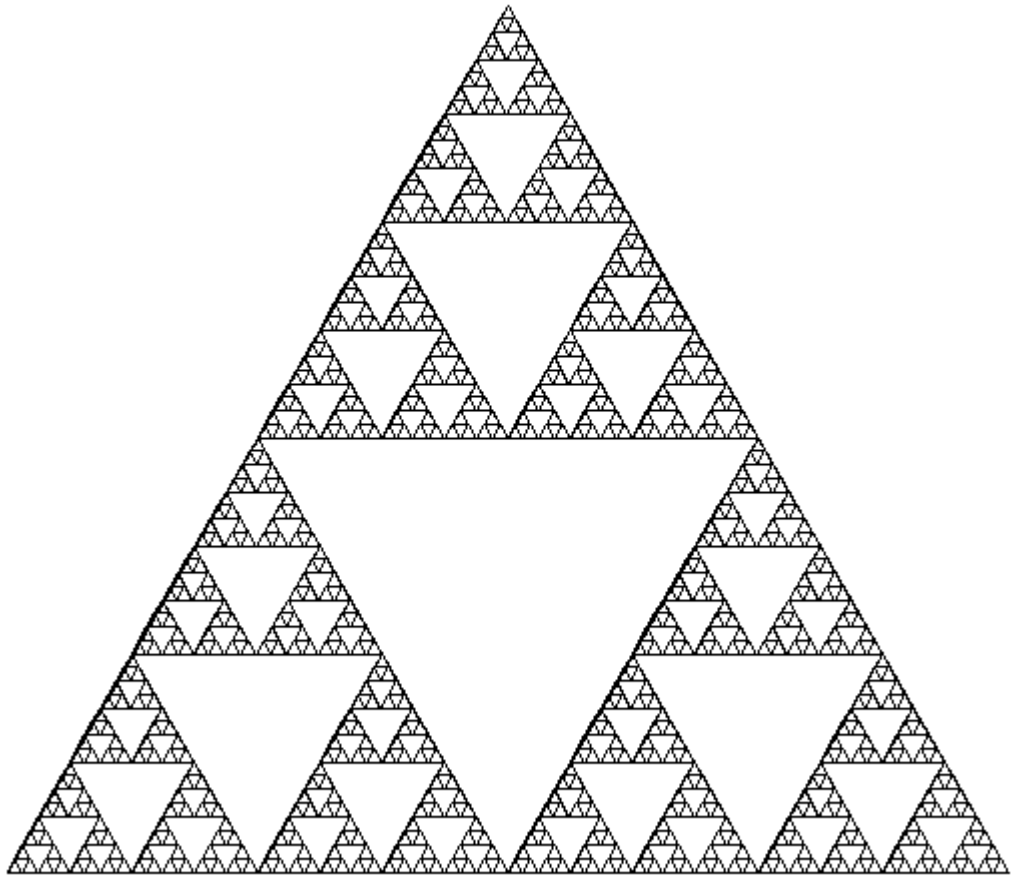
Flera rekursiva anrop är OK här

Flera rekursiva anrop är OK här



Sierpinskitriangel

Flera rekursiva anrop är OK här



Sierpinskitriangel

Ni får rita den som Matlab-uppgift

Algoritm =

Algoritm $=$

instruktioner för hur ett problem ska lösas

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Hur lång tid tar algoritmen?

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Hur lång tid tar algoritmen?

Hur mycket minne krävs?

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Hur lång tid tar algoritmen?

Hur mycket minne krävs?

Hur beskrivs algoritmers tidseffektivitet?

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Hur lång tid tar algoritmen?

Hur mycket minne krävs?

Hur beskrivs algoritmers tidseffektivitet?

Räkna primitiva karakteristiska operationer, ofta något av:

Algoritm $=$

instruktioner för hur ett problem ska lösas

Algoritmanalys $=$

Hur lång tid tar algoritmen?
Hur mycket minne krävs?

Hur beskrivs algoritmers tidseffektivitet?

Räkna primitiva karakteristiska operationer, ofta något av:

- Aritmetiska operationer $(+, -, *, /)$

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Hur lång tid tar algoritmen?
Hur mycket minne krävs?

Hur beskrivs algoritmers tidseffektivitet?

Räkna primitiva karakteristiska operationer, ofta något av:

- Aritmetiska operationer $(+, -, *, /)$
- Tester/jämförelser $(==, <, >)$

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Hur lång tid tar algoritmen?
Hur mycket minne krävs?

Hur beskrivs algoritmers tidseffektivitet?

Räkna primitiva karakteristiska operationer, ofta något av:

- Aritmetiska operationer $(+, -, *, /)$
- Tester/jämförelser $(==, <, >)$
- Tilldelningar

Algoritm =

instruktioner för hur ett problem ska lösas

Algoritmanalys =

Hur lång tid tar algoritmen?
Hur mycket minne krävs?

Hur beskrivs algoritmers tidseffektivitet?

Räkna primitiva karakteristiska operationer, ofta något av:

- Aritmetiska operationer $(+, -, *, /)$
- Tester/jämförelser $(==, <, >)$
- Tilldelningar
- Annat ...

Hur växer antalet operationer med problemets storlek?

Antal operationer kan bero på aktuella data

Antal operationer kan bero på aktuella data

- Bästa fallet

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast
  - Ibland onödigt pessimistiskt

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast
  - Ibland onödigt pessimistiskt
- Medelfallet

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast
  - Ibland onödigt pessimistiskt
- Medelfallet
  - Intressant

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast
  - Ibland onödigt pessimistiskt
- Medelfallet
  - Intressant
  - Kan vara svårt att beräkna

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast
  - Ibland onödigt pessimistiskt
- Medelfallet
  - Intressant
  - Kan vara svårt att beräkna
  - Kräver information om indatas statistiska fördelning

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast
  - Ibland onödigt pessimistiskt
- Medelfallet
  - Intressant
  - Kan vara svårt att beräkna
  - Kräver information om indatas statistiska fördelning

Antal operationer skrivs som funktion av n.

Antal operationer kan bero på aktuella data

- Bästa fallet
  - Oftast inte så intressant
- Sämsta fallet
  - Används oftast
  - Ibland onödigt pessimistiskt
- Medelfallet
  - Intressant
  - Kan vara svårt att beräkna
  - Kräver information om indatas statistiska fördelning

Antal operationer skrivs som funktion av n.
n = problemets storlek

Hur jämförs algoritmers effektivitet ?

Hur jämförs algoritmers effektivitet ?

$\mathcal{O}$–notation: (Ordo)

Hur jämförs algoritmers effektivitet ?

$\mathcal{O}$–notation: (Ordo)

- Bortser (oftast) från konstanta skalfaktorer

Hur jämförs algoritmers effektivitet ?

$\mathcal{O}$–notation: (Ordo)

- Bortser (oftast) från konstanta skalfaktorer
- Mäter vad som händer för stora problemstorlekar (n)

Hur jämförs algoritmers effektivitet ?

$\mathcal{O}$–notation: (Ordo)

- Bortser (oftast) från konstanta skalfaktorer
- Mäter vad som händer för stora problemstorlekar (n)
- Anger snabbast växande term

Hur jämförs algoritmers effektivitet ?

$\mathcal{O}$–notation: (Ordo)

- Bortser (oftast) från konstanta skalfaktorer
- Mäter vad som händer för stora problemstorlekar (n)
- Anger snabbast växande term

Definition:

Hur jämförs algoritmers effektivitet ?

$\mathcal{O}$–notation: (Ordo)

- Bortser (oftast) från konstanta skalfaktorer
- Mäter vad som händer för stora problemstorlekar (n)
- Anger snabbast växande term

Definition:

En funktion $f(n)$ är $\mathcal{O}(g(n))$ *om*

$$f(n) \leq c \cdot g(n) \qquad \forall n > n_0$$

för något n_0 och något $c > 0$

Algoritmers effektivitet

Vanliga klasser av algoritmer

Algoritmers effektivitet

Vanliga klasser av algoritmer

Konstant körtid $\mathcal{O}(1)$

Algoritmers effektivitet

Vanliga klasser av algoritmer

Konstant körtid	$\mathcal{O}(1)$
Logaritmisk	$\mathcal{O}(log\ n)$
Linjär	$\mathcal{O}(n)$
Linlog	$\mathcal{O}(n\ log\ n)$

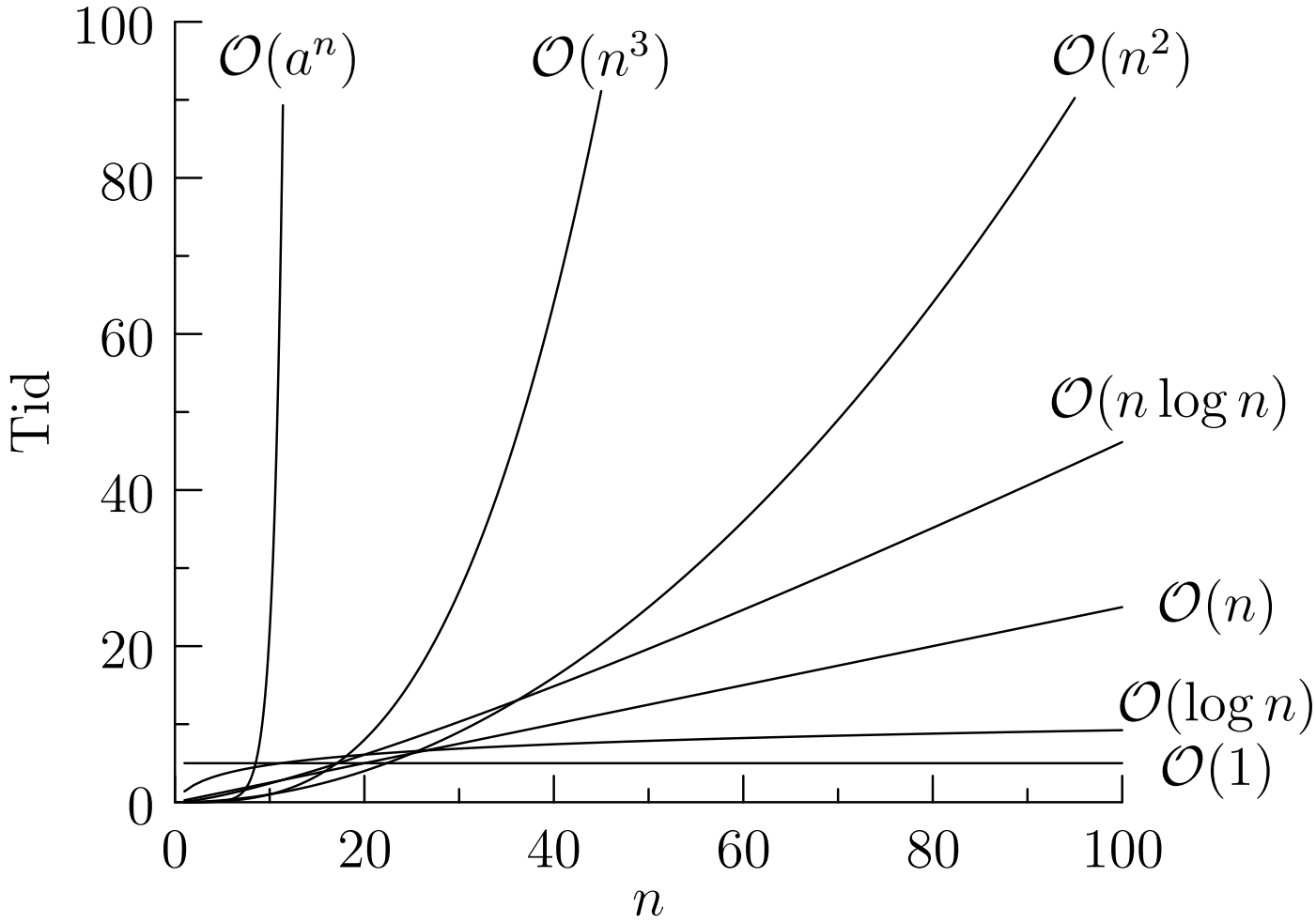
Algoritmers effektivitet

Vanliga klasser av algoritmer

Konstant körtid	$\mathcal{O}(1)$
Logaritmisk	$\mathcal{O}(log\ n)$
Linjär	$\mathcal{O}(n)$
Linlog	$\mathcal{O}(n\ log\ n)$
Kvadratisk	$\mathcal{O}(n^2)$
Polynomisk	$\mathcal{O}(n^k)$
Exponentiell	$\mathcal{O}(a^n)$

Algoritmers effektivitet – Ordo-kurvor

Algoritmers effektivitet – Ordo-kurvor



Hur växer körtiden?

	$n = 2$	$n = 16$	$n = 128$	$n = 1024$
$log\ n$	1	4	7	10
n	2	16	128	1024
$n \cdot logn$	2	64	896	10240
n^2	4	256	16384	1048576
n^3	8	4096	2097152	1073741824
2^n	4	65536	$3.4 \cdot 10\ 38$	$1.74 \cdot 10^{308}$

Hur stort kan n få vara?

Antag att varje operation tar $1\mu s$

	1 s	1 min	1 h
$400n$	2500	150000	9000000
$20n \cdot logn$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Hur stort kan n få vara?

Antag att varje operation tar $1\mu s$

	1 s	1 min	1 h
$400n$	2500	150000	9000000
$20n \cdot logn$	4096	166666	7826087
$2n^2$	707	5477	42426
n^4	31	88	244
2^n	19	25	31

Observation:

En mycket snabbare dator ger endast marginella förbättringar för "dåliga" algoritmer.

Sökalgoritmer

Sökalgoritmer

- Linjärsökning
- Binärsökning

Linjärsökning

Linjärsökning

- Antag lista med **n** st element
- Leta efter värdet **t**

Linjärsökning

- Antag lista med **n** st element
- Leta efter värdet **t**
- Jämför ett listelement i taget med **t**

Linjärsökning

- Antag lista med **n** st element
- Leta efter värdet **t**
- Jämför ett listelement i taget med **t**
- Sluta när **t** hittats eller listan är slut

Linjärsökning

- Antag lista med **n** st element
- Leta efter värdet **t**
- Jämför ett listelement i taget med **t**
- Sluta när **t** hittats eller listan är slut

Hur många jämförelser görs?

- Bästa fall: 1

Linjärsökning

- Antag lista med **n** st element
- Leta efter värdet **t**
- Jämför ett listelement i taget med **t**
- Sluta när **t** hittats eller listan är slut

Hur många jämförelser görs?

- Bästa fall: 1
- Sämsta fall: n

Linjärsökning

- Antag lista med **n** st element
- Leta efter värdet **t**
- Jämför ett listelement i taget med **t**
- Sluta när **t** hittats eller listan är slut

Hur många jämförelser görs?

- Bästa fall: 1
- Sämsta fall: n
- Medelfall: $\frac{n}{2}$

Komplexitet för linjärsökning: $\mathcal{O}(n)$

Linjärsökning, algoritmen

Leta efter **thing** i listan **alist**

Linjärsökning, algoritmen

Leta efter **thing** i listan **alist**

found anger med True/False om man hittat eller ej

```
found = False
for elem in alist:
    if elem == thing:
        found = True
        break
```

Linjärsökning, algoritmen

Leta efter **thing** i listan **alist**

found anger med True/False om man hittat eller ej

```
found = False
for elem in alist:
    if elem == thing:
        found = True
        break
```

I Python kan linjärsökning göras ännu enklare:

```
found = thing in alist
```

Binärsökning

Binärsökning

- Antag lista med **n** st **sorterade** element

Binärsökning

- Antag lista med **n** st **sorterade** element
- Jämför med elementet i mitten, index **n//2**

Binärsökning

- Antag lista med **n** st **sorterade** element
- Jämför med elementet i mitten, index **n//2**
- Leta vidare i ena halvan

Binärsökning

- Antag lista med **n** st **sorterade** element
- Jämför med elementet i mitten, index **n//2**
- Leta vidare i ena halvan

Hur många jämförelser m behövs?

Binärsökning

- Antag lista med **n** st **sorterade** element
- Jämför med elementet i mitten, index **n//2**
- Leta vidare i ena halvan

Hur många jämförelser m behövs?

Hur många gånger måste vi dela **n** med 2 för att få 1?

Binärsökning

- Antag lista med **n** st **sorterade** element
- Jämför med elementet i mitten, index **n//2**
- Leta vidare i ena halvan

Hur många jämförelser m behövs?

Hur många gånger måste vi dela **n** med 2 för att få 1?

Antalet element att leta bland: $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...2, 1$

Binärsökning

- Antag lista med **n** st **sorterade** element
- Jämför med elementet i mitten, index **n//2**
- Leta vidare i ena halvan

Hur många jämförelser m behövs?

Hur många gånger måste vi dela **n** med 2 för att få 1?

Antalet element att leta bland: $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...2, 1$

$$\frac{n}{2^m} = 1$$

Binärsökning

- Antag lista med **n** st **sorterade** element
- Jämför med elementet i mitten, index **n//2**
- Leta vidare i ena halvan

Hur många jämförelser m behövs?

Hur många gånger måste vi dela **n** med 2 för att få 1?

Antalet element att leta bland: $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, ...2, 1$

$$\frac{n}{2^m} = 1$$

Antal jämförelser $= m = log_2(n)$

Komplexitet: $\mathcal{O}(log(n))$

Binärsökning, algoritmen i Python

Leta efter x i sorterade listan alist

```
def binsok(alist, x):
    lo = 0
    hi = len(alist)-1
    while lo <= hi:
        mid = (lo+hi)//2
        if x < alist[mid]:
            hi = mid - 1
        elif x > alist[mid]:
            lo = mid + 1
        else:
            return True
    return False
```

Binärsökning, algoritmen i Python

Leta efter x i sorterade listan alist

```
def binsok(alist, x):
    lo = 0
    hi = len(alist)-1
    while lo <= hi:
        mid = (lo+hi)//2
        if x < alist[mid]:
            hi = mid - 1
        elif x > alist[mid]:
            lo = mid + 1
        else:
            return True
    return False
```

Om x finns i alist returneras

Binärsökning, algoritmen i Python

Leta efter x i sorterade listan alist

```
def binsok(alist, x):
    lo = 0
    hi = len(alist)-1
    while lo <= hi:
        mid = (lo+hi)//2
        if x < alist[mid]:
            hi = mid - 1
        elif x > alist[mid]:
            lo = mid + 1
        else:
            return True
    return False
```

Om x finns i alist returneras **True** annars

Binärsökning, algoritmen i Python

Leta efter x i sorterade listan alist

```
def binsok(alist, x):
    lo = 0
    hi = len(alist)-1
    while lo <= hi:
        mid = (lo+hi)//2
        if x < alist[mid]:
            hi = mid - 1
        elif x > alist[mid]:
            lo = mid + 1
        else:
            return True
    return False
```

Om x finns i alist returneras **True** annars **False**