UPPSALA UNIVERSITET

MA1: L01

Tom Smedsaas

Introduktion till rekursion

UPPSALA UNIVERSITET

Algoritmer

Definition: En *algoritm* är en sekvens av väldefinierade instruktioner som löser ett problem eller utför en beräkning med ett ändligt antal steg.

Beståndsdelar:

- *sekvens* (en följd av instruktioner)
- *selektion* (`if`, `elif`, `else`)
- *iteration* (`for`, `while`)
- *abstraktion* (funktioner, metoder)

Exempel: Fakultetberäkning

Fakulteten kan definieras *iterativt*:

$$n! = \begin{cases} 1 & \text{om } n = 0 \\ 1 \cdot 2 \cdot 3 \cdot \ \ldots \ \cdot n & \text{om } n > 0 \end{cases}$$

vilket kan uttryckas i kod med någon form av loop:

```
def fac(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

Rekursiv fakultetsberäkning

Fakulteten kan också definieras *rekursivt*:

$$n! = \begin{cases} 1 & \text{om } n = 0 \\ n \cdot (n-1)! & \text{om } n > 0 \end{cases}$$

```
def fac(n):
    if n == 0:
        return 1
    else:
        return n*fac(n-1)
```

```
def fac(n):
    result = 1
    if n > 0:
        result =n*fac(n-1)
    return result
```

UPPSALA UNIVERSITET

Demonstration med hjälp av debuggern i Thonny

Exempel: Beräkna x^n

Operationen x^n där x reellt och n heltal:

Iterativt: $$x^n = \begin{cases} 1 & \text{om } n = 0 \\ x \cdot x \cdot x \cdot \ \ldots \ \cdot \mathrm{x} & \text{om } n > 0 \end{cases}$$

Rekursivt: $$x^n = \begin{cases} 1 & \text{om } n = 0 \\ x \cdot x^{n-1} & \text{om } n > 0 \end{cases}$$

Om n får vara negativt?

$$x^n = \begin{cases} \dfrac{1}{x^{-n}} & \text{om } n < 0 \\ 1 & \text{om } n = 0 \\ x \cdot x^{n-1} & \text{om } n > 0 \end{cases}$$

```
def power(x, n):
    if n < 0:
        return 1./power(x, -n)
    elif n == 0:
        return 1
    else:
        return x*power(x, n-1)
```

UPPSALA UNIVERSITET

Varför rekursion?

- Kraftfullt sätt att *hitta* algoritmer.
- Ett kraftfullt sätt att hitta *effektiva* algoritmer.
- *Naturligt* i många problem.

Men också

- Kraftfullt sätt att konstruera ineffektiva algoritmer

Rekursion generellt

1. Dela upp problemet i ett eller flera delproblem av samma typ.
2. Lös delproblemen (rekursivt)
3. Kombinera lösningarna till delproblemen till en lösning av ursprungsproblemet.

Det måste finnas minst ett rekursionsterminerande fall, så kallade basfall.

Fakultetsberäkningen $n!$:

- **Ett** delproblem: Beräknas $(n-1)!$.
- **Kombinera**: Multiplicera med n.
- **Basfall**: $n = 0$.

Hur hittar man delproblemen?

I ovanstående exempel definieras problemstorleken av ett tal *n* som ges som parameter: $n!$ och x^n.

Annat exempel:

`number_of_digits(x)`

som ska returnera antalet decimala siffror i heltalet x:

`number_of_digits(125)` → `3`

`number_of_digits(2341562)` → `7`

```
def number_of_digits(x):
    if x < 10:
        return 1
    else:
        return 1 + number_of_digits(x//10)
```

Ibland kan delproblemen göras på olika sätt

Skriv funktionen `reverse(lst)` som returnerar en ny lista där elementen i listan `lst` kommer i omvänd ordning.

(Vi ignorerar alla inbyggda funktioner och metoder för att reversera.)

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    else:
        mid = len(lst)//2
        return reverse(lst[mid:]) + reverse(lst[:mid])
```

Fråga: Hur gör vi samma operation på en sträng?

Ibland är rekursionen självklar

Antag att vi har följande *iterativa* funktion:

```
def reverse_list(lst):
    result = []
    for x in lst:
        result.insert(0, x)
    return result
```

In: `[1, [2, [3, 4]], [[5, 6], 7], [8, 9]]`

Ut: `[[8, 9], [[5, 6], 7], [2, [3, 4]], 1]`

Ibland är rekursionen självklar …

Antag nu att funktionen även ska vända på alla ingående dellistor dvs

```
In:   [1,  [2, [3, 4]],  [[5, 6], 7],  [8, 9]]
Ut:   [[9, 8],  [7, [6, 5]],  [[4, 3], 2],  1]
```

Enkelt! Vi har ju en funktion
som reverserar listor:

```
def reverse_list(lst):
    result = []
    for x in lst:
        result.insert(0, x)
    return result
```

```
def reverse_list(lst):
    result = []
    for x in lst:
        if type(x) is list:
            x = reverse_list(x)
        result.insert(0, x)
    return result
```

Exempel med två basfall

En funktion som tar emot två listor med tal och returnerar en ny lista där elementen består av talen talen summerade parvis:

```
summa([1,2,3], [1,2,3,4,5])   →   [2,4,6,4,5]
summa([1,2,3], [5,7])         →   [5,9,3]
```

```
def summa(x, y):
    if len(x) == 0:
        return y
    elif len(y) == 0:
        return x
    else:
        return [x[0] + y[0]] + summa(x[1:], y[1:])
```

UPPSALA UNIVERSITET

The end

.

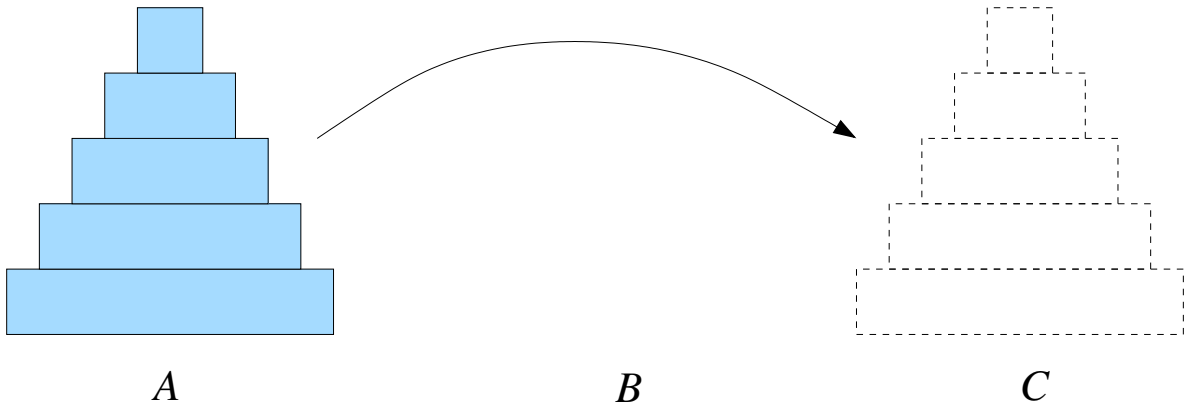
UPPSALA UNIVERSITET

MA1: L02

Tom Smedsaas

Rekursion: Två exempel.

Hanois torn



En trave brickor av olika storlek som ska flyttas från stället ***A*** till stället ***C*** med iakttagande av följande regler:

- Bara en bricka får flyttas i taget
- Man får aldrig lägga en större bricka på en mindre

För att detta ska vara möjligt behövs en ”hjälpplats” ***B***

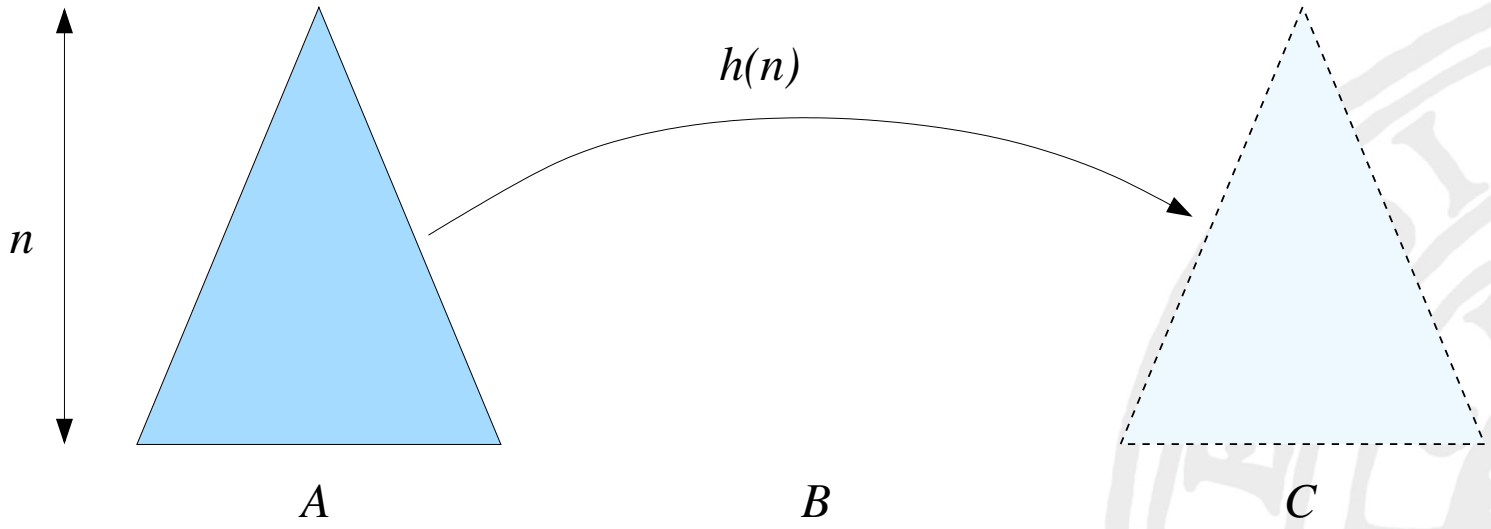
Visualisering

Tom flyttar brickor

Tháp Hà Nội

Beteckna problemet att flytta *n* brickor från ett ställe till ett annat med *h(n)*.

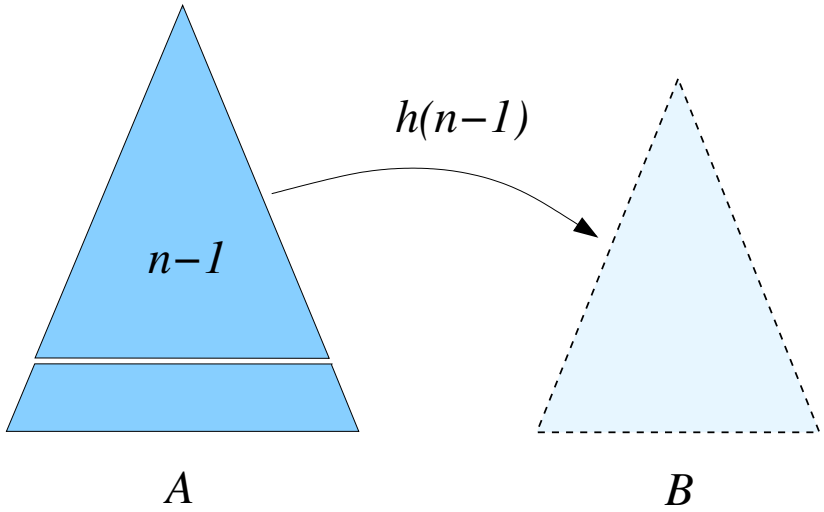
Vi vill alltså lösa problemet:



Hanois torn

För att kunna flytta den understa brickan från A till C måste alla andra brickor ligga på B.

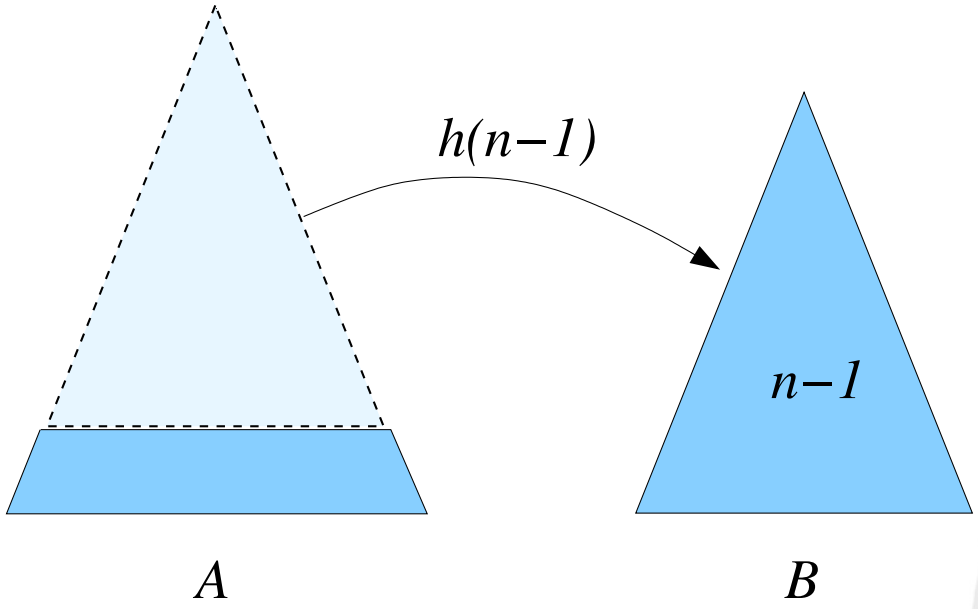
Vi måste alltså börja med att lösa problemet att flytta *n*-1 brickor från A till B:



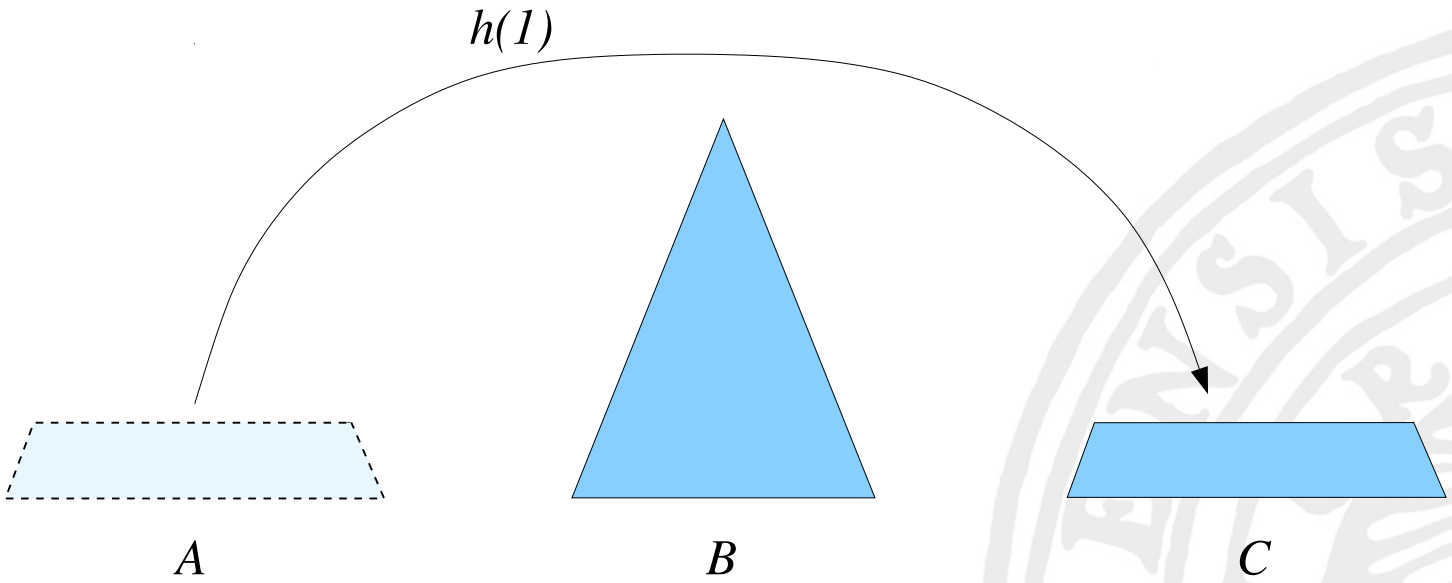
1. Flytta *n*-1 brickor från *A* till *B* med hjälp av *C*

.

Hanois torn



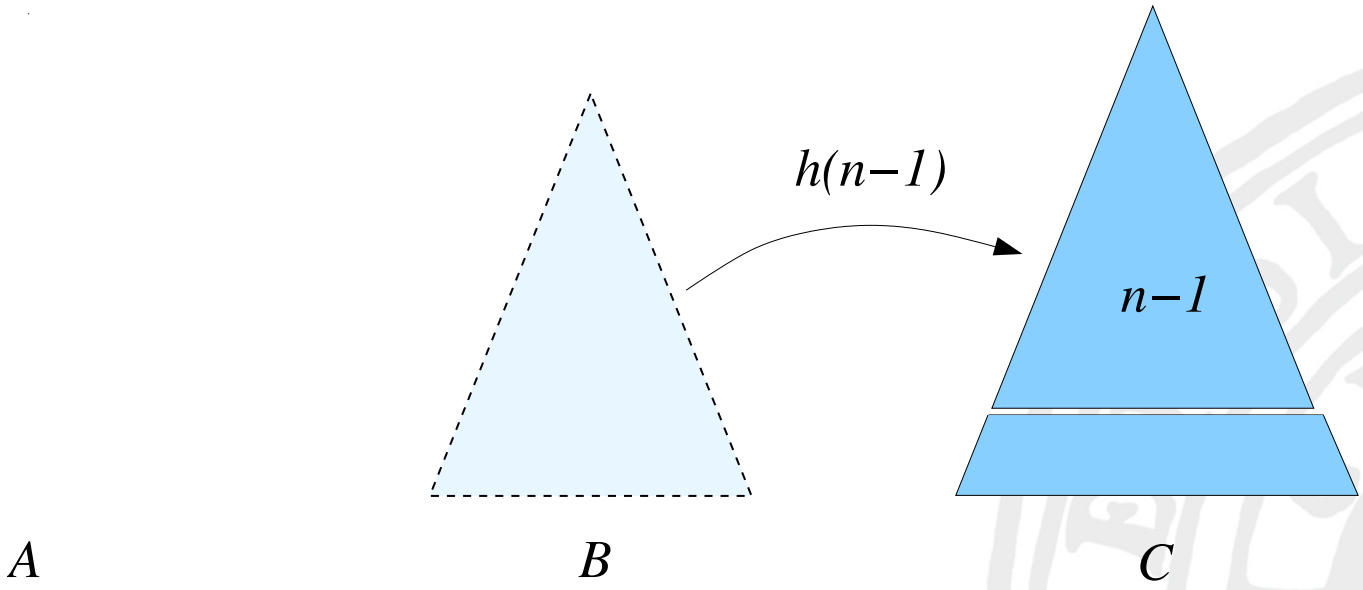
Hanois torn



2. Flytta 1 bricka från *A* till *C*.

.

Hanois torn



3. Flytta n – 1 brickor från *B* till *C* med hjälp av *A*

Hanois torn

Algoritm:

1. Flytta *n* - 1 brickor från *A* till *B* med hjälp av *C.*
2. Flytta 1 bricka från *A* till *C.*
3. Flytta n – 1 brickor från *B* till *C* med hjälp av *A.*

Observera att det är två rekursiva anrop samt att uppgifterna om från, till och med hjälp av varierar. Platserna *A*, *B* och *C* har byter alltså roller.

Tips 1: Enklast kod får man om man använder 0 som basfall.

Tips 2: Det räcker med 5 rader kod inklusive `def`, `if` och `else` för att lösa uppgiften!

UPPSALA UNIVERSITET

Växlingsproblemet

UPPSALA UNIVERSITET

Växlingsproblemet

Antag att vi har mynt/sedlar med följande värden: 1, 5, 10, 50, 100.
På hur många olika sätt kan vi växla ett givet belopp?

Exempel: Beloppet 12 kan växlas på 4 olika sätt:

1x10 + 2x1, 2x5 + 2x1, 1x5 + 7x1 och 12x1

Vi vill skriva en `funktion`

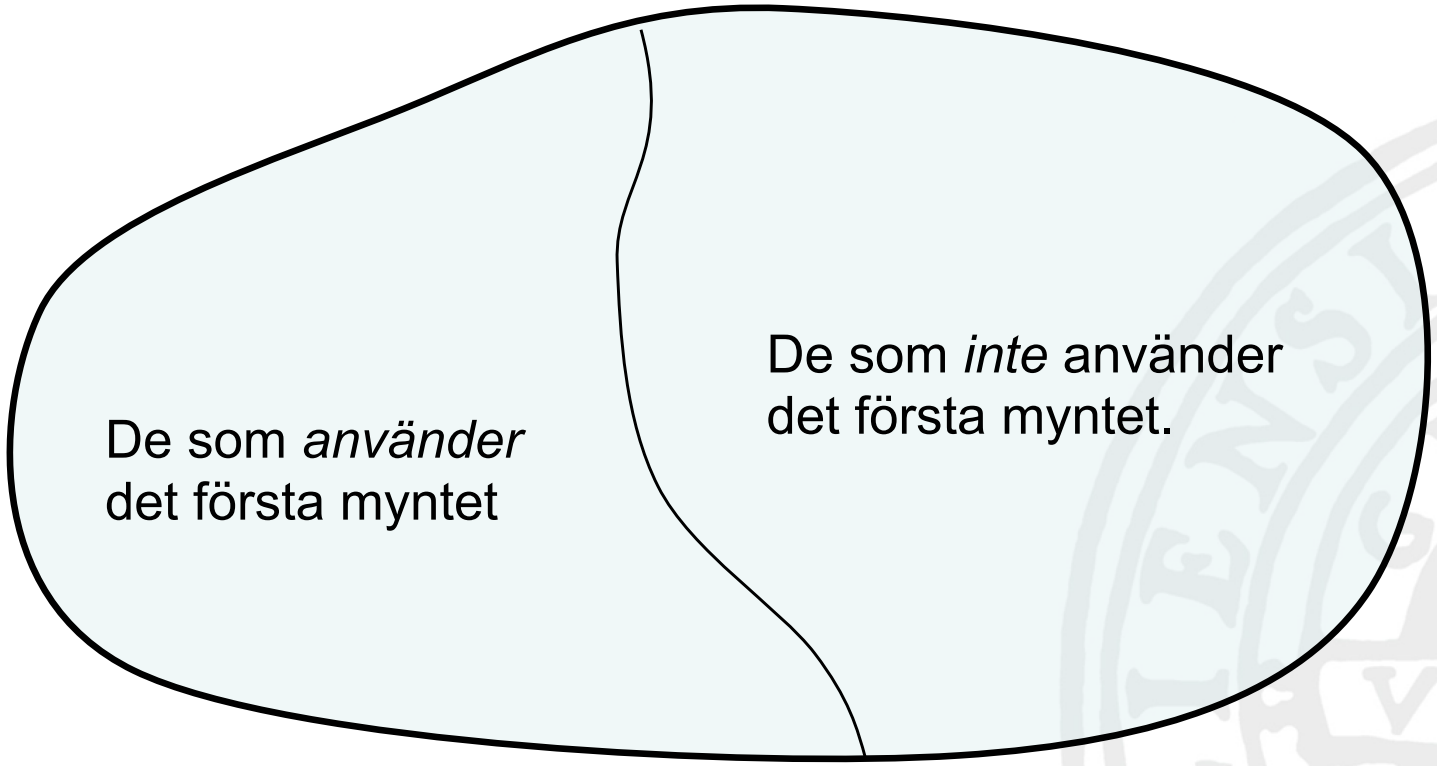
```
exchange(a, coins)
```

som returnerar antalet sätt som man kan växla beloppet a med de valörer som finns i `coins`.

Anropet `exchange(12, [1, 5, 10, 50, 100])` ska alltså returnera 4.

Växlingsproblemet

Mängden möjliga växlingar:



Växlingsproblemet

Det ger följande kod:

```
def exchange(a, coins):
    return \
        exchange(a, coins[1:]) + \
        exchange(a-coins[0], coins)
```

Basfall?

- a = 0 Lyckad. Räkna 1
- a < 0 Misslyckad. Räkna 0
- coins tom Misslyckad. Räkna 0

Växlingsproblemet

Slutlig version:

```
def exchange(a, coins):
    if a == 0:
        return 1
    elif a < 0 or len(coins) == 0:
        return 0
    else:
        return \
            exchange(a, coins[1:]) + \
            exchange(a-coins[0], coins)
```

Två utmatningar (frivilliga):

1. Skriv en funktion som listar de gjorda växlingarna.
2. Lista möjliga växlingar givet ett begränsat antal av varje mynt.

UPPSALA UNIVERSITET

The end

.

UPPSALA UNIVERSITET

MA1: L03

Tom Smedsaas

Introduktion till komplexitetsanalys

Beräkna x^n iterativt

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ x \cdot x \cdot x \cdot \ \ldots \ \cdot x & \text{om } n > 0 \end{cases}$$

```
def power(x, n):
    result = 1
    for i in range(1, n+1):
        result *= x
    return result
```

Algoritmen gör *n* multiplikationer så tiden växer linjärt med *n* oberoende av *x*.

$\frac{10}{4}$ $10 \cdot \frac{1}{4}$

Beräkna x^n *rekursivt*

```
def power(x, n):
    if n == 0:
        return 1
    else:
        return x*power(x, n-1)
```

Om vi låter *t*(*n*) stå för tiden anropet `power(x, n)` tar så gäller

$$t(n) = \begin{cases} c & \text{om } n = 0 \\ d + t(n-1) & \text{om } n > 0 \end{cases}$$

$$t(n) = d + t(n-1) = d + d + t(n-2) = d + d + d + t(n-3) = \ \ldots$$

$$\ldots = d \cdot n + t(0) = d \cdot n + c$$

Effektivare beräkning av x^n

Antag att vi vill beräkna x^{16}.

Om vi börjar med att beräkna x^8 så räcker en kvadrering för att få x^{16}

och x^8 kan beräknas med att kvadrera x^4

och x^4 kan beräknas med att kvadrera x^2

och x^2 kan beräknas med en multiplikation.

Alltså, x^{16} kan beräkna med 4 multiplikationer i stället för 16.

Hur kan man formulera denna idé till en generell algoritm?

Effektivare beräkning av x^n

Antag att *n* är jämnt och ≥ 0:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ (x^{n/2})^2 & \text{om } n > 0 \end{cases}$$

Om *n* är udda så är *n* -1 jämnt:

$$x^n = \begin{cases} 1 & \text{om } n = 0 \\ (x^{n/2})^2 & \text{om } n > 0,\ n \text{ jämnt} \\ x \cdot (x^{(n-1)/2})^2 & \text{om } n > 0,\ n \text{ udda} \end{cases}$$

Effektivare beräkning av x^n

```
def power(x, n):

    def sqr(x):
        return x*x

    if n<0:
        return 1./power(x, -n)
    elif n==0:
        return 1
    elif n%2==0:
        return sqr(power(x, n//2))
    else:
        return x*sqr(power(x, (n-1)//2))
```

$$\begin{cases} 1 \\ (x^{n/2})^2 \\ x \cdot (x^{(n-1)/2})^2 \end{cases}$$

Hur många multiplikationer gör denna algoritm?

Om *n* är en jämn 2-potens, dvs om $n = 2^k$?

Krävs $k = \log_2 n$ multiplikationer

Om *n* inte är en jämn 2-potens, dvs om $n = 2^k$?

Krävs högst en extra multiplikation innan problemet halveras
dvs högst $2 \log_2 n$ multiplikationer.

Ungefär hur många multiplikationer krävs för att beräkna
x^{1000} respektive $x^{1000000}$?

Sökning i en lista

Att söka efter ett värde i en Python-lista görs normalt med operatorn in t. ex. med uttrycket `if x in lst`

Även om vi inte ser det så måste det bakom scenen finnas kod liknande den här:

```
def search(x, lst):
    for e in lst:
        if e == x:
            return True
    return False
```

Arbetet är således, åtminstone i värsta fall, proportionellt mot listans längd.

Operatorn `in` och funktionen `search` har samma *komplexitet* även om operatorn in säkert är mycket snabbare.

UPPSALA UNIVERSITET

Effektivare sökning

Om data i listan är sorterade i storleksordning kan sökningen göras väsentligt effektivare:

om x < värdet i mitten:
sök i vänster halva
annars:
sök i höger halva

Metoden som kallas binär sökning är enkel i implementera både rekursivt och iterativt.

Komplexitet för binär sökning

Metoden halverar sökmängden för varje iteration eller rekursivt anrop.

$$n \longrightarrow \frac{n}{2} \longrightarrow \frac{n}{2^2} \longrightarrow \frac{n}{2^3} \longrightarrow \ldots \longrightarrow \frac{n}{2^k}$$

När är $\frac{n}{2^k} = 1$?

$$n = 2^k \Leftrightarrow k = \log_2 n$$

Hur många iterationer/funktionsanrop behövs för att söka i en lista med $10^6, 10^9$ och 10^{12} element?

Algoritm- eller komplexitetsanalys

Studerar hur tiden för en algoritm beror på indata.

Resultat av typen:

tiden växer proportionellt mot kvadraten på antalet element

eller

tiden är konstant oberoende av input

eller

tiden växer exponentiellt med problemstorleken

UPPSALA UNIVERSITET

The end

.

UPPSALA UNIVERSITET

MA1: L04

Tom Smedsaas

Asymptotisk notation
Ordo ($\mathcal{O}$), Omega (Ω) och Theta (Θ)

UPPSALA UNIVERSITET

Asymptotisk notation

Exempel på formuleringar:

- *Den enkla algoritmen för att beräkna x^n är $\Theta(n)$*
- *Kvadreringsalgorimen för att beräkna x^n är $\Theta(\log n)$*
- *Binärsökningsalgoritmen är $\Theta(\log n)$*

- *Linjärsökningsalgoritmen är $\mathcal{O}(n)$ alltid*
- *Linjärsökningsalgoritmen är $\mathcal{O}(1)$ i bästa fall*
- *Linjärsökningsalgoritmen är $\mathcal{O}(n)$ i genomsnitt*

Asymptotisk notation - definition

En funktion $t(n)$ sägs vara $\mathcal{O}(f(n))$ om det existerar två konstanter c och n_0 sådana att

$$|t(n)| < c|f(n)| \quad \text{för alla} \quad n > n_0$$

Observera att detta är en *matematisk* definition!

Asymptotisk notation

Påstående: $f(n) = n^2 + 10n + 100$ är $\mathcal{O}(n^2)$

Varför?

Om $n > 1$ så

$$|f(n)| = n^2 + 10n + 100 < n^2 + 10n \cdot n + 100 \cdot n \cdot n = 111n^2$$

Alltså duger konstanterna $c = 111$ och $n_0 = 1$

Observera att det är *existensen* som är viktig – inte värdena!

UPPSALA UNIVERSITET

Asymptotisk notation

Eftersom $\mathcal{O}$ är en *övre* gräns behöver det inte innebära så mycket.
Funktionen på förra bilden är t.ex. automatiskt $\mathcal{O}(n^3)$ och $\mathcal{O}(2^n)$.

För att ange en *undre* gräns används Ω en liknande definition med > i stället för <.
Exempelfunktionen är alltså också $\Omega(n^2)$.

En funktion som är både $\mathcal{O}(f(n))$ och $\Omega(f(n))$ sägs vara $\Theta(f(n))$.

I datavetenskapen är det vanligt att säga $\mathcal{O}$ när man menar Θ.

UPPSALA UNIVERSITET

Användning i algoritmanalys

Funktionerna som vi uppskattar är *tid* eller *antal gånger* någon central operation utförs.

Exempel: Antal multiplikationer som utförs i kvadreringsalgoritmen för att beräkna x^n är $\Theta(\log n)$.

Behöver vi inte ange bas i logaritmen?

Nej, eftersom

$$\log_a x = \frac{1}{\log_b a} \cdot \log_b x$$

Användning i algoritmanalys

Exempel:

$\mathcal{O}(1)$	indexering i array (Python-lista)
$\Theta(\log n)$	binär sökning, kvadreringsalgoritmen
$\Theta(n)$	`insert(0, x)` i en lista med n lagrade värden
$\Theta(n \log n)$	snabba sorteringsmetoder
$\Theta(n^2)$	enkla sorteringsmetoder
$\Theta(n^3)$	multiplikation av $n \times n$ - matriser

UPPSALA UNIVERSITET

I algoritmanalys

När man analyserarar algoritmer kan man behöva skilja på bästa, värsta och genomsnittlig komplexitet.

Exempel:

Att sortera n element med *insticksortering* kräver

- $\Theta(n)$ operationer i bästa fall
- $\Theta(n^2)$ operationer i värsta fall
- $\Theta(n^2)$ operationer i genomsnitt

Vad menar man med ”*i genomsnitt*”?

I algoritmanalys

När man diskuterar sökning behöver man ofta skilja mellan ”lyckad” och ”misslyckad” sökning.

Till exempel, funktionen

```
def search(x, lst):
    for e in lst:
        if e == x:
            return True
    return False
```

kräver för *lyckad sökning*

- $\Theta(1)$ operationer i bästa fall
- $\Theta(n)$ operationer i genomsnitt
- $\Theta(n)$ operationer i värsta fall

medan misslyckad sökning alltid kräver $\Theta(n)$ operationer.

Tidsuppskattningar

Antag att vi vet att tiden för en viss algoritm är $\Theta\big(f(n)\big)$. Då kan vi uppskatta tiden för stora värden på n med funktionen

$$t(n) = c \cdot f(n)$$

Konstanten c beror på många saker (dator, os, programmerare, …).

För en viss implementation och dator kan den uppskattas genom att göra en eller flera tidsmätningar.

Exempel

Antag att tiden för ett program som implementerar en $\Theta(n \log n)$ - algoritm har uppmätts till 1 sek för en $n = 10^3$.
Hur lång tid kommer programmet då ta för $n = 10^6$?

$$t(n) = c\, n \log n$$

$$t(10^3) = c \cdot 10^3 \log 10^3 = 1$$

$$c = 1/3000$$

$$t(10^6) = \frac{1}{3000} \cdot 10^6 \log 10^6 = 2000 \text{ sek } \approx 33 \text{ min}$$

Exempel

Antag samma mätning men att det är en $\Theta(n^2)$ - algoritm.
Hur lång tid kommer programmet då ta för $n = 10^6$?

$$t(n) = c \cdot n^2$$

$$t(10^3) = c \cdot (10^3)^2 = 1$$

$$c = 10^{-6}$$

$$t(10^6) = 10^{-6} \cdot (10^6)^2 = 10^6 \text{ sek} \approx 12 \text{ dagar}$$

Slutsats: En $\Theta(n \log n)$ - algoritm är *mycket* snabbare än en $\Theta(n^2)$ - algoritm för stora värden på n.

UPPSALA UNIVERSITET

The end

.

UPPSALA UNIVERSITET

Insticks- och samsortering

Tom Smedsaas

Två sorteringsalgoritmer:
instickssortering och samsortering
(mergesort)

UPPSALA UNIVERSITET

Sortering

Vanliga algoritmer att ta upp är

- Insticksortering
- Urvalssortering
- Bubbelsortering

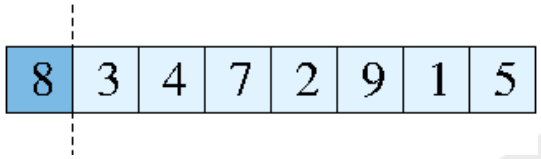
Egenskaper:

- Enkla att förstå
- Enkla att programmera
- Alla $\Theta(n^2)$ i genomsnitt

Insticksortering

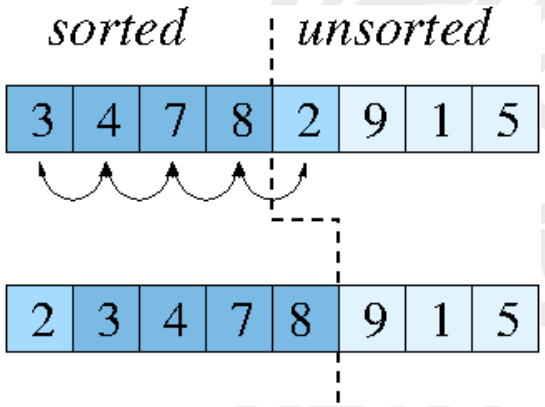
Ett vanligt sätt att beskriv insticksorteringen är att säga att listan består av två delar – en sorterad och en osorterad.

Från början innehåller den sorterade delen bara det första elementet:



För varje steg utvidgas den sorterade delen med det som står först i den osorterade delen

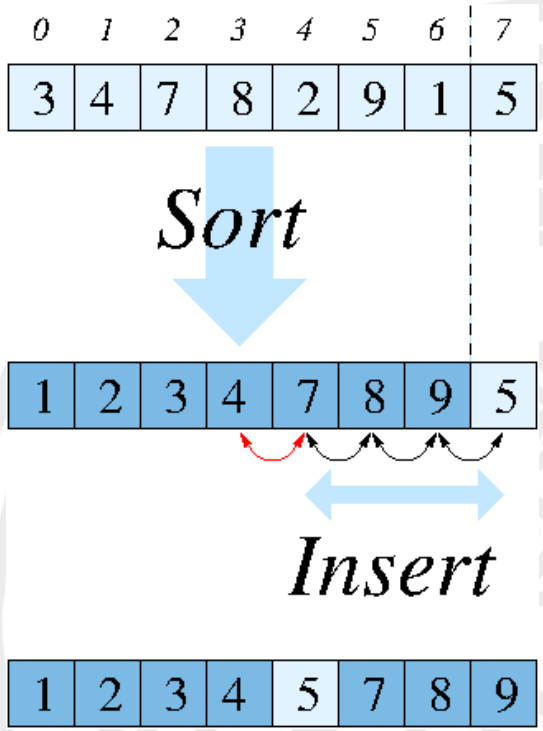
Så här kan det se ut vid den fjärde utvidgningen:



Insticksortering rekursivt

För att sortera en lista med n element sorterar vi först de $n - 1$ första elementen varefter vi infogar det sista elementet så att sorteringen behålls.

```
def ins_sort(lst, n):
    if n <= 1:
        return
    ins_sort(lst, n-1)
    i = n-1
    while i>0 and lst[i] < lst[i-1]:
        lst[i-1], lst[i] = \
            lst[i], lst[i-1]
        i -= 1
```



Insticksortering – bästa fall

```
def ins_sort(lst, n):
    if n <= 1:
        return
    ins_sort(lst, n-1)
    i = n-1
    while i>0 and lst[i] < lst[i-1]:
        lst[i-1], lst[i] = lst[i], lst[i-1]
        i -= 1
```

Låt $t(n)$ vara antalet gånger while-villkoret beräknas.

I *bästa* fall (om värdena read är sorterade):

$$t(n) = t(n-1) + 1 = (t(n-2) + 1) + 1 = \cdots = n$$

Insticksortering – värsta fall

```
def ins_sort(lst, n):
    if n <= 1:
        return
    ins_sort(lst, n-1)
    i = n-1
    while i>0 and lst[i] < lst[i-1]:
        lst[i-1], lst[i] = lst[i], lst[i-1]
        i -= 1
```

I *värsta* fall (om sorterat i omvänd ordning):

$$t(n) = t(n-1) + n \;\; = \big(t(n-2) + (n-1)\big) + n \quad = \cdots$$

$$= 1 + 2 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

Insticksorteringen – genomsnitt

```
def ins_sort(lst, n):
    if n <= 1:
        return
    ins_sort(lst, n-1)
    i = n-1
    while i>0 and lst[i] < lst[i-1]:
        lst[i-1], lst[i] = lst[i], lst[i-1]
        i -= 1
```

I *genomsnitt* tänker vi oss att elementen reser halva vägen:

$$t(n) = t(n-1) + n/2 \; = (t(n-2) + (n-1)/2) + n/2 \quad = \cdots \quad = \frac{n \cdot (n+1)}{4}$$

Således $\Theta(n^2)$ både i genomsnitt och i värsta fall.

Balansering av algoritmen

I stället för att infoga elementen ett och ett kan man ta flera åt gången.

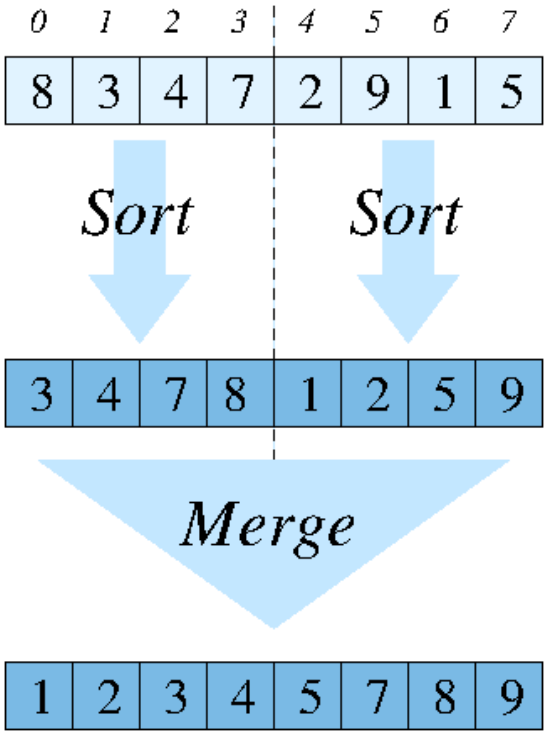
8	3	4	7	2	9	1	5

Bäst blir det om man delar i mitten:

1. Dela mängden i två lika stora delar.
2. Sortera delarna var för sig.
3. Sammanfoga delarna.

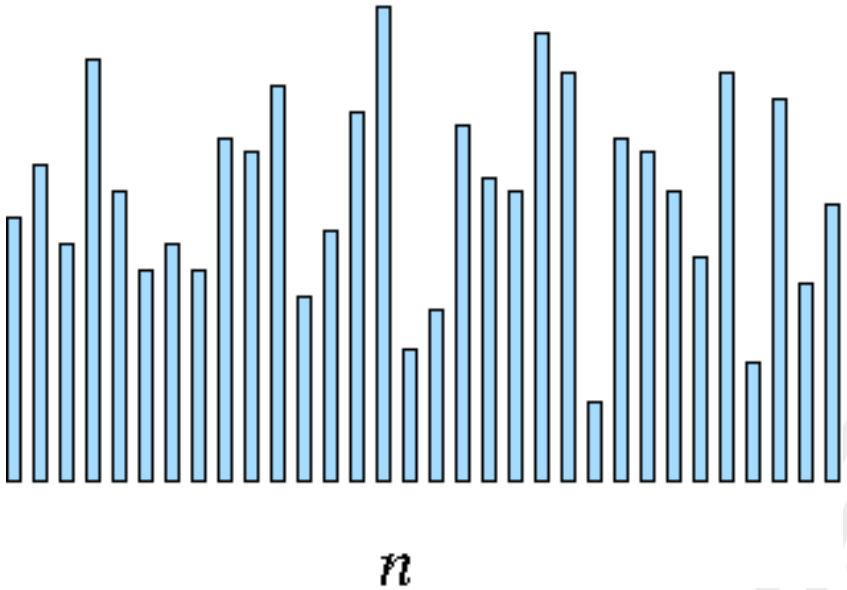
UPPSALA UNIVERSITET

Mergesort



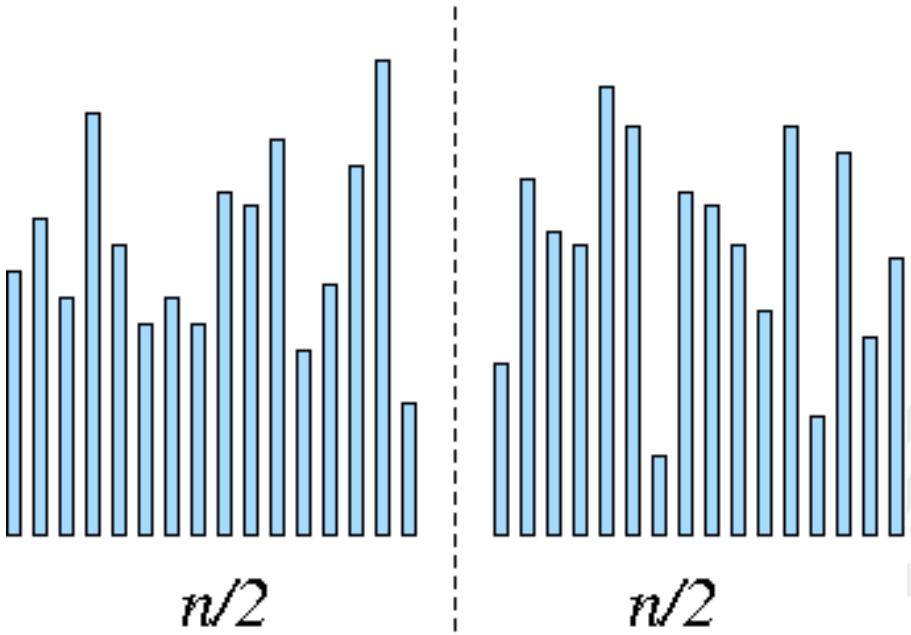
UPPSALA UNIVERSITET

Mergesort - illustration

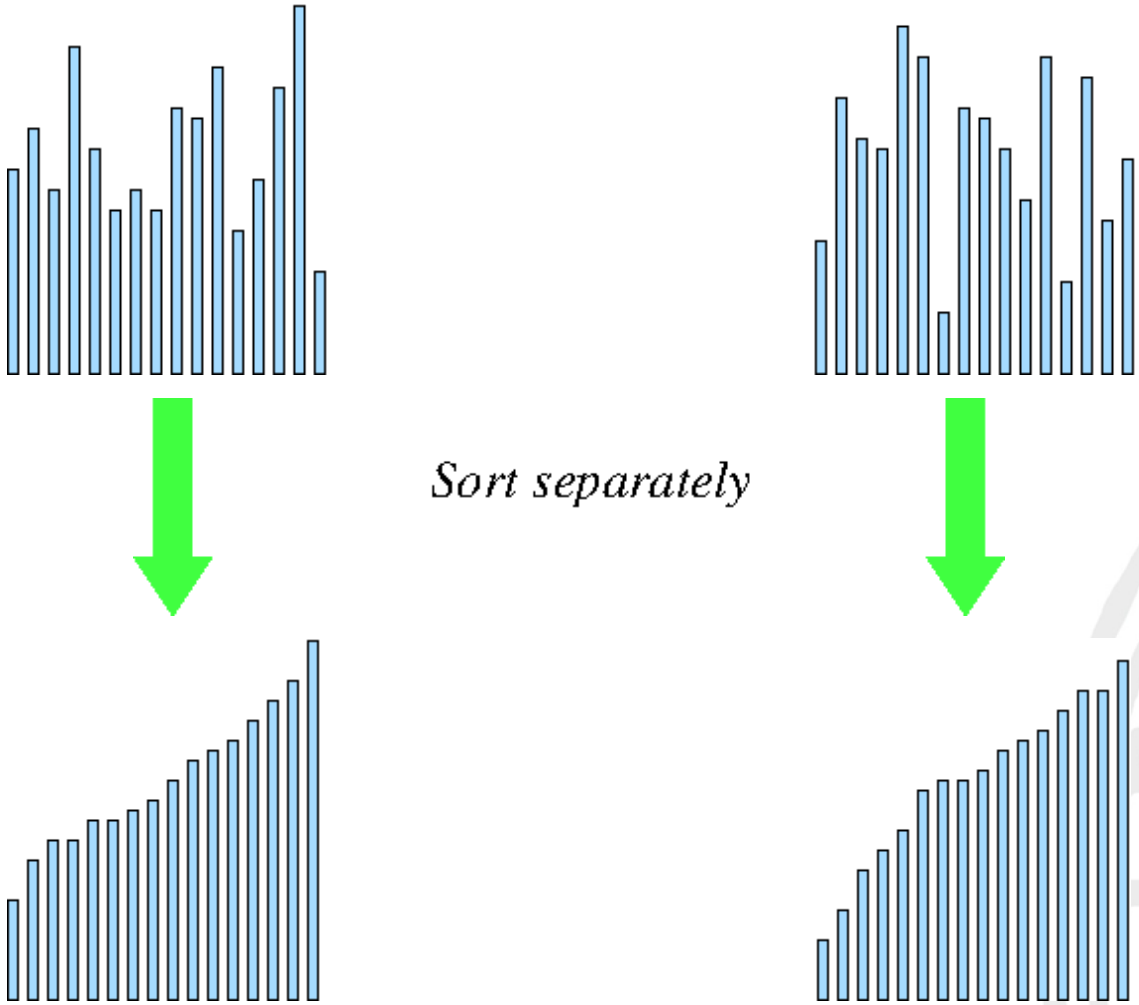


UPPSALA UNIVERSITET

Mergesort-illustration

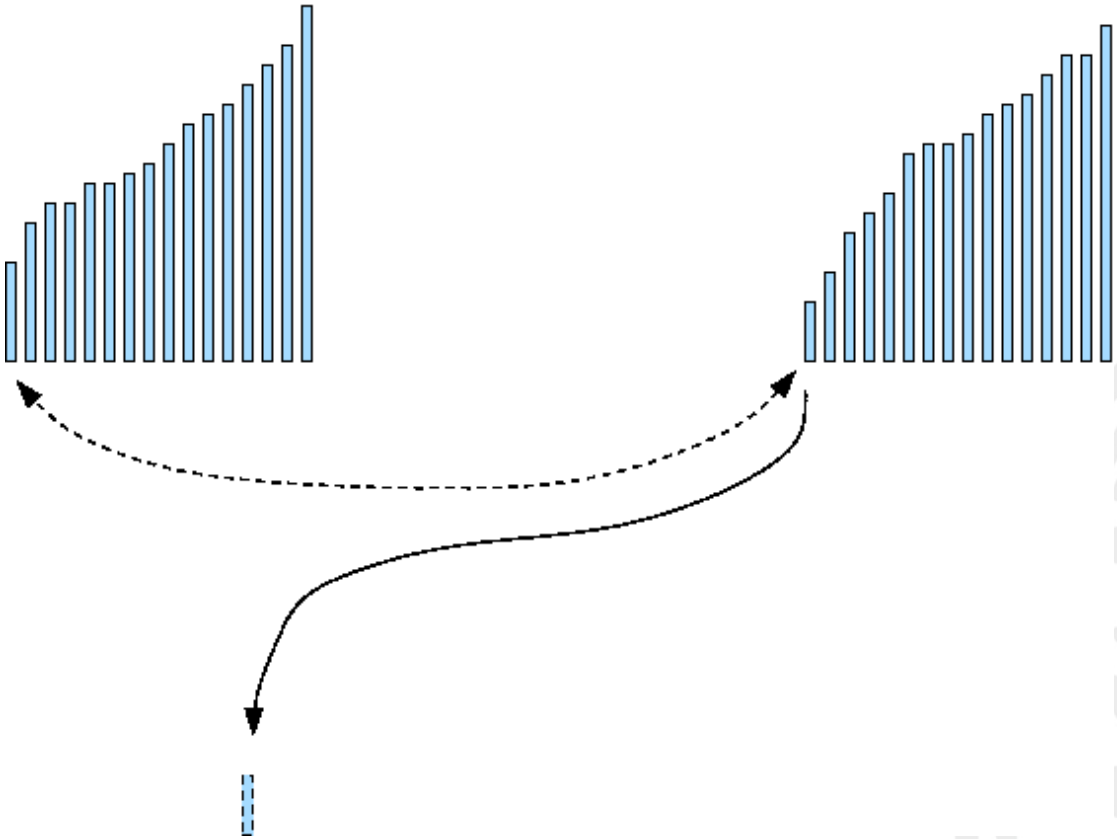


Mergesort-illustration



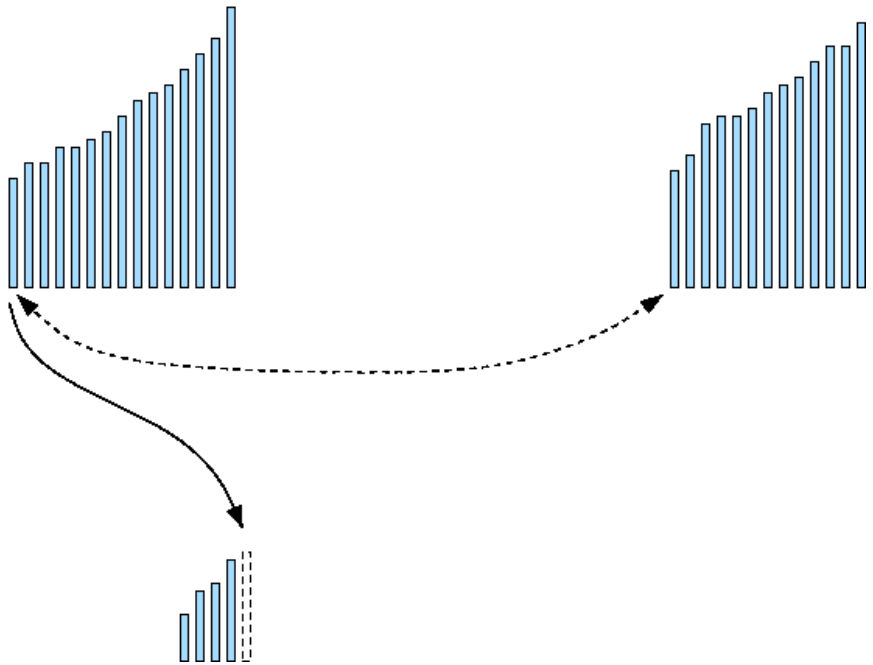
UPPSALA UNIVERSITET

Mergesort-illustration



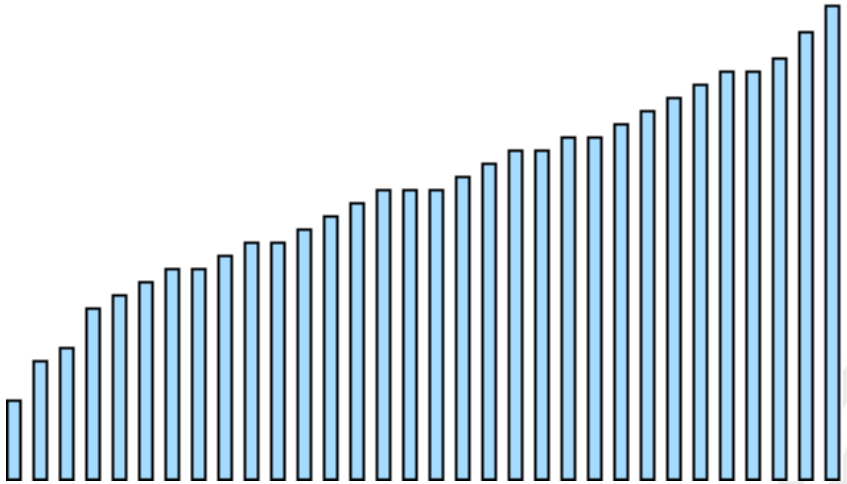
UPPSALA UNIVERSITET

Mergesort-illustration



UPPSALA UNIVERSITET

Mergesort-illustration



Analys av mergesort

Vi löser ett problem av storlek n genom att lösa *två* problem av storlek $n/2$ samt en sammanfogning av de två lösningarna.

Tiden för att sammanfoga de två lösningarna är proportionell mot n.

Låt $t(n)$ vara tiden det tar att sortera n element. Då gäller

$$t(n) = \begin{cases} c & \text{om } n = 0 \\ 2t(n/2) + d \cdot n & \text{om } n > 0 \end{cases}$$

där c och d är obekanta konstanter.

Analys av mergesort

Om n är en jämn 2-potens, $n = 2^k$, så gäller

$$t(n) = 2t(n/2) + dn =$$

$$= 2\left(2t(n/4) + \frac{dn}{2}\right) + dn =$$

$$= 4t(n/4) + dn + dn =$$

$$= 2^k t(n/2^k) + kdn = nt(1) + dn\log_2 n$$

Således är algoritmen komplexitet $\Theta(n \log n)$.

Detta gäller även om n inte är en jämn tvåpotens.

UPPSALA UNIVERSITET

Två frågor

- Antag att det tar 1 sekund att sortera 10^3 tal den enkla insticksorteringen. Hur lång tid kommer det då att ta att sortera 10^6 tal?
- Samma fråga för mergesort.

The end

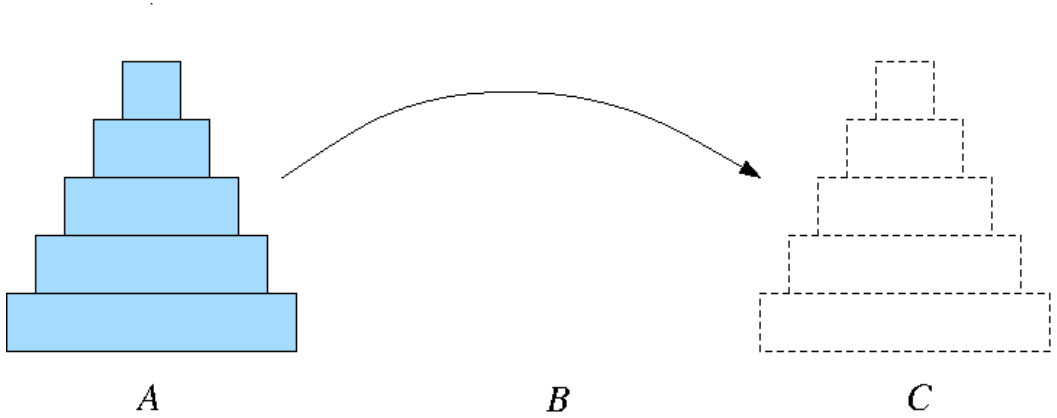
UPPSALA UNIVERSITET

Hanois torn

Tom Smedsaas

Komplexitet för problemet Hanois torn

Hanois torn

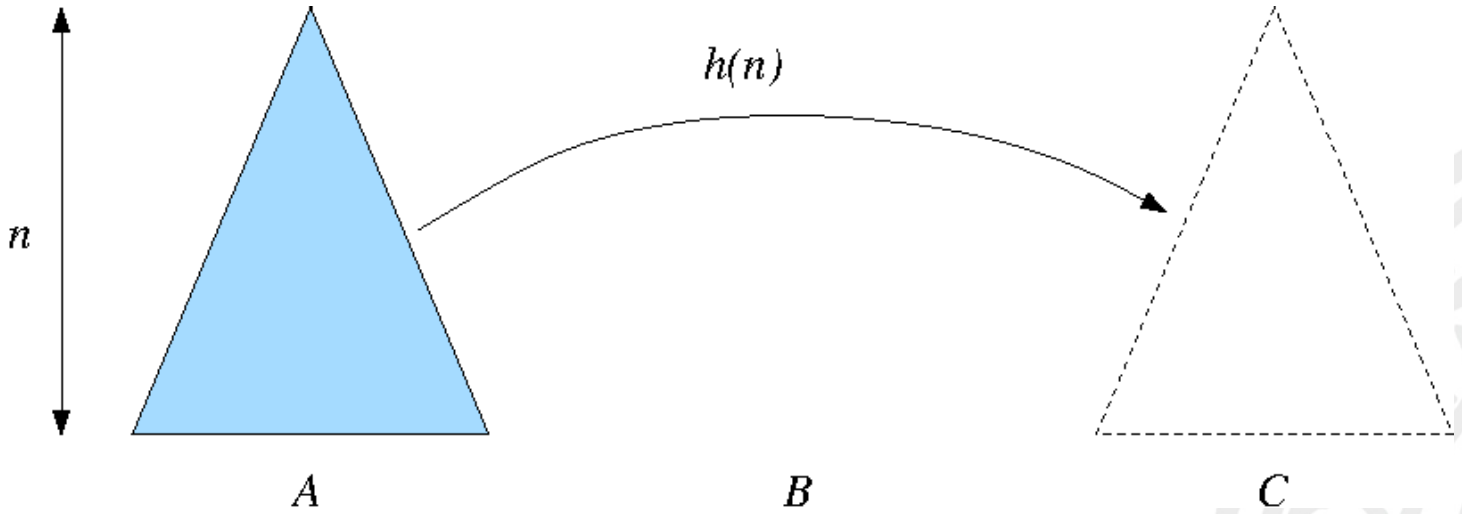


En hög med brickor ska flyttas från A till C med följande regler:

1. Endast en bricka får flyttas åt gången
2. En större bricka får aldrig läggas på en mindre.

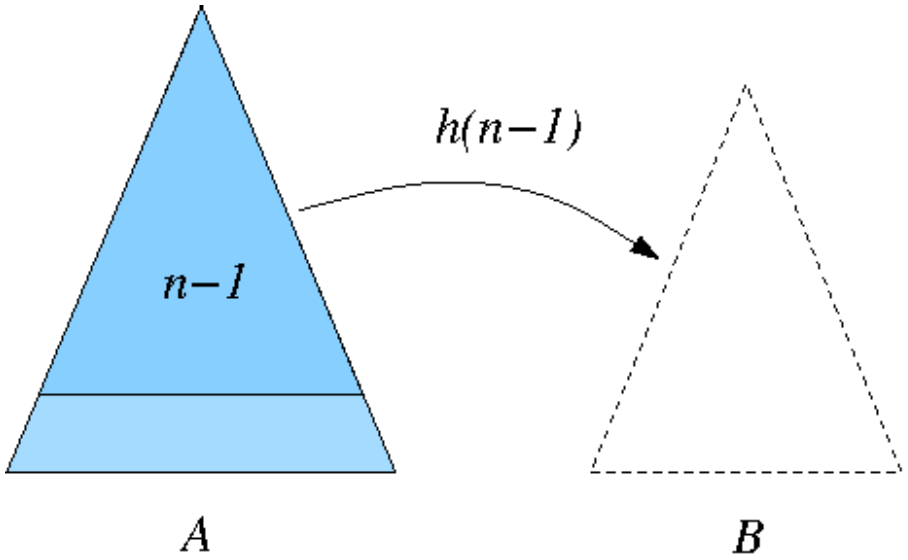
Hanois torn

Låt $h(n)$ stå för problemet att flytta n brickor.

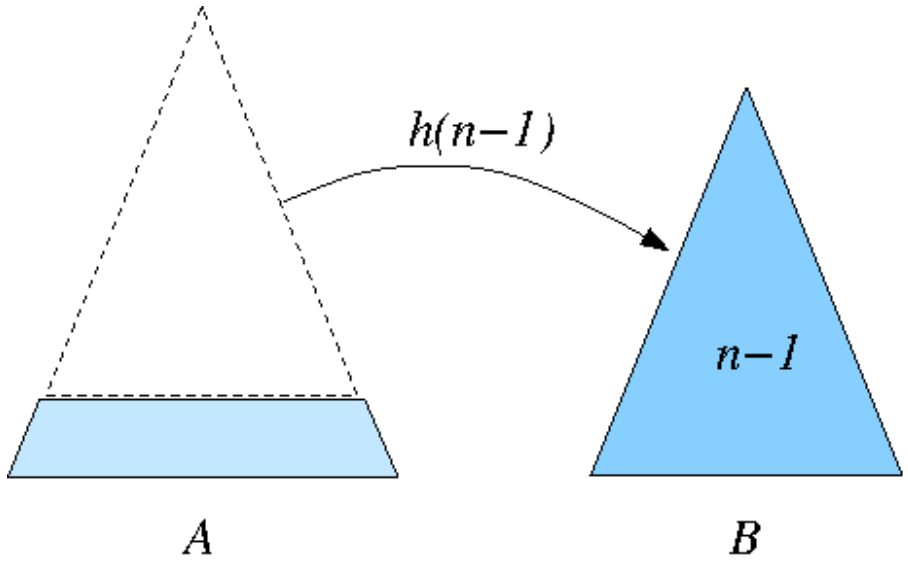


UPPSALA UNIVERSITET

Hanois torn

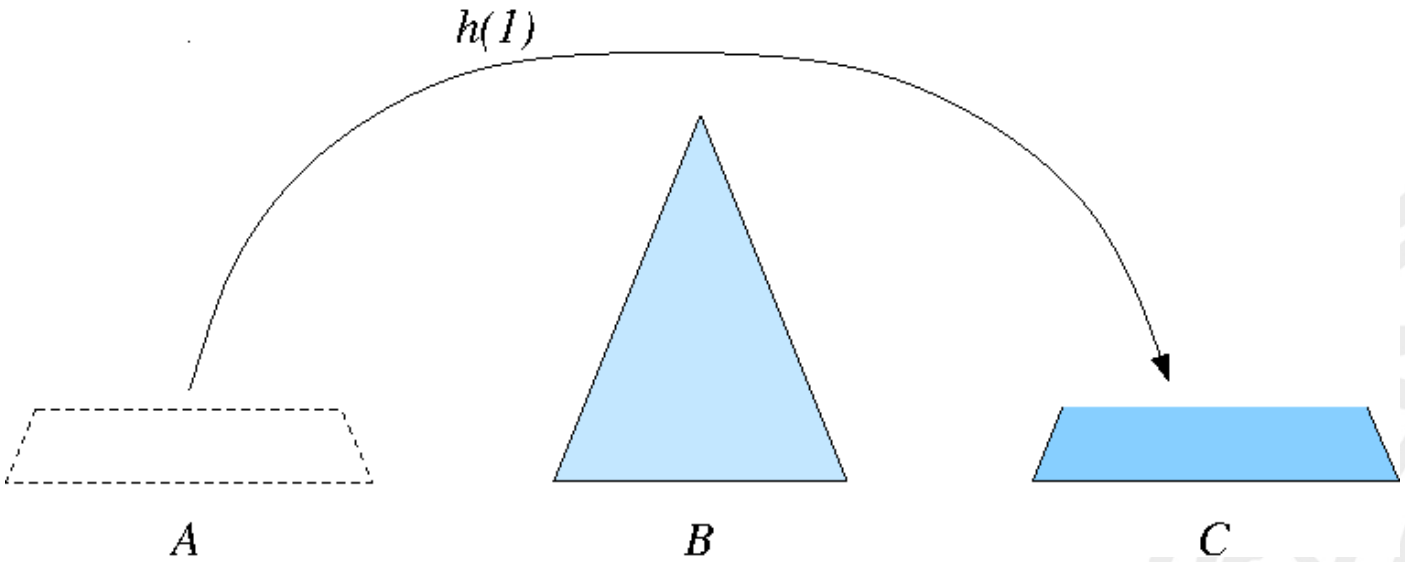


Hanois torn

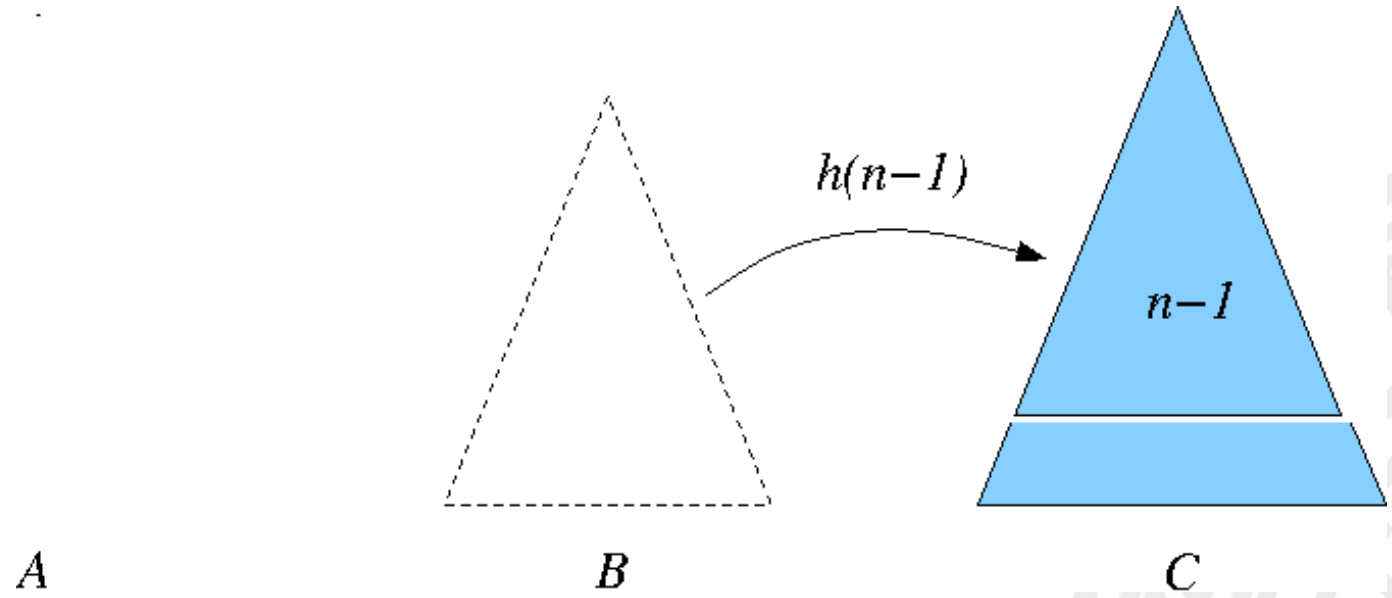


UPPSALA UNIVERSITET

Hanois torn



Hanois torn



Analys av komplexiteten

Vi löser ett problem av storleken n genom att lösa två problem av storleken $n-1$ och ett problem av storleken 1.

Om $h(n)$ står för antalet brickförflyttningar blir då

$$h(n) = 2h(n-1) + 1 =$$

$$= 2(2h(n-2) + 1) + 1 = 4h(n-2) + 2 + 1 = \cdots =$$

$$= 2^k h(n-k) + 2^{k-1} + 2^{k-2} + \cdots + 2 + 1 =$$

$$= 2^{n-1} h(1) + 2^{n-2} + \cdots + 2 + 1 =$$

$$= 2^{n-1} \cdot 1 + 2^{n-2} + \cdots + 2 + 1 = 2^n - 1$$

Hopplöst för stora n.

UPPSALA UNIVERSITET

The end

UPPSALA UNIVERSITET

Fibonacci-tal

Tom Smedsaas

Komplexitet vid rekursiv beräkning av Fibonacci-tal.

Vi ska också introducera tekniken “memoization” för att effektivisera vissa rekursiva program.

Rekursiv Fibonacci-funktion

Fibonacci-talen kan definieras rekursivt:

$$F_n = \begin{cases} 0 & \text{om } n = 0 \\ 1 & \text{om n} = 1 \\ F_{n-1} + F_{n-2} & \text{om } n > 1 \end{cases}$$

och beräknas med en rekursiv funktion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

UPPSALA UNIVERSITET

Demo 1

Komplexitet för `fib`-funktionen

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Låt $t(n)$ vara antalet additioner som görs av anropet `fib(n)`.

$$t(n) = t(n-1) + t(n-2) + 1$$

Kan uppskattas uppåt:

$$t(n) = t(n-1) + t(n-2) + 1 < 2t(n-1) + 1 = 2^n - 1$$

Kan uppskattas nedåt:

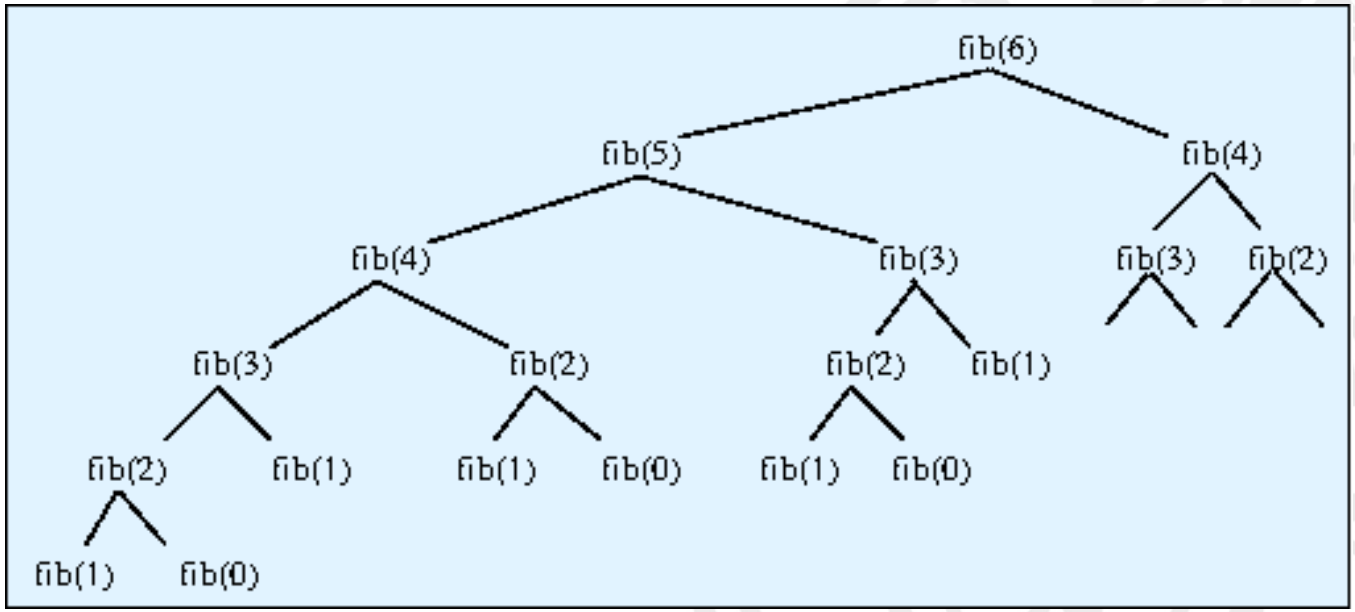
$$t(n) = t(n-1) + t(n-2) > 2t(n-2) + 1 \approx \sqrt{2}^n$$

Man kan visa att $t(n) \approx c \cdot 1.618^n$ Hopplöst för stora n !

Varför blir det så ineffektivt?

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Vad händer vid anropet `fib(6)` ?



UPPSALA UNIVERSITET

”Memoization”

Spara beräknade värden och använd dessa i första hand.

```
memory = {0:0, 1:1}

def fib(n):
    if n not in memory:
        memory[n] = fib(n-1) + fib(n-2)
    return memory[n]
```

Kan nu beräkna väldigt stora Fibonacci-tal!

UPPSALA UNIVERSITET

Demo 2

Kan man dölja variabeln memory?

Variabeln memory är nu global – inte så vackert!
Den används ju bara i funktionen fib.

```
memory = {0:0, 1:1}

def fib(n):
    if n not in memory:
        memory[n] = fib(n-1) + fib(n-2)
    return memory[n]
```

Så här?

```
def fib(n):
    memory = {0:0, 1:1}
    if n not in memory:
        memory[n] = fib(n-1) + fib(n-2)
    return memory[n]
```

UPPSALA UNIVERSITET

Demo 3

Bättre försök att gömma memory

Låt fib vara en ”omslagsfunktion” till den rekursiva funktionen och placera memory i den:

```
def fib(n):

    memory = {0:0, 1:1}

    def _fib(n):
        if n not in memory:
            memory[n] = _fib(n-1) + _fib(n-2)
        return memory[n]

    return _fib(n)
```

UPPSALA UNIVERSITET

Demo 4

UPPSALA UNIVERSITET

Alltså

- Variabler skapade i en funktion är lokala i den funktionen.
- Funktionsdefinitioner kan innehålla funktionsdefinitioner. Sådana funktioner är lokala till den omslutande funktionen.
- I en lokal funktion är variablerna i den omslutande funktionen tillgängliga.
- Variabler deklarerade på högsta nivån (utanför alla funktioner) är global Undvik att använda globala variabler!

Frivillig övning: Skriv om funktionen `exchange` med memoization.

UPPSALA UNIVERSITET

Pythonspecialitet

```
from functools import lru_cache

@lru_cache()
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

UPPSALA UNIVERSITET

Demo 5

UPPSALA UNIVERSITET

The end

.

UPPSALA UNIVERSITET

Sortering

Tom Smedsaas

Experiment med några sorteringsmetoder.

Avsikten är att visa hur man kan verifiera teori med experiment samt att peka på när rekursion är bra och när den är dålig.

UPPSALA UNIVERSITET

Insticksortering igen

```
def ins_sort_rec(lst):
    return _ins_sort_rec(lst, len(lst))


def _ins_sort_rec(lst, n):
    if n > 1:
        _ins_sort_rec(lst, n-1)
        x = lst[n-1]
        i = n - 2
        while i >= 0  and lst[i] > x:
            lst[i+1] = lst[i]
            i -= 1
        lst[i+1] = x
    return lst
```

UPPSALA UNIVERSITET

Mergesort igen

```
def merge_sort(lst):
    if len(lst) <= 1: return lst
    else:
        n = len(lst)//2
        l1 = lst[:n]
        l2 = lst[n:]
        l1 = merge_sort(l1)
        l2 = merge_sort(l2)
        return merge(l1, l2)

def merge(l1, l2):
    if len(l1) == 0:
        return l2
    elif len(l2) == 0:
        return l1
    elif l1[0] <= l2[0]:
        return [l1[0]] + merge(l1[1:], l2)
    else:
        return [l2[0]] + merge(l1, l2[1:])
```

Hur kommer tiden växa för dessa metoder?

Vi ska titta på hur tiderna förändras när storleken dubblas.

Insticksorteringen är en $\Theta(n^2)$ – metod.

Dubblas storleken växer tiden med en faktor $\frac{c\cdot(2n)^2}{c\cdot n^2} = 4$

Samsorteringen är en $\Theta(n \log n)$ – metod.

Dubblas storleken växer tiden med en faktor

$$\frac{c \cdot 2n \log(2n)}{c \cdot n \log n} = 2 \cdot \frac{\log 2 + \log n}{\log n} = 2 \cdot (1 + \frac{1}{\log_2 n})$$

Om $n > 1000$ är faktorn mindre än 2.1.

Testkod

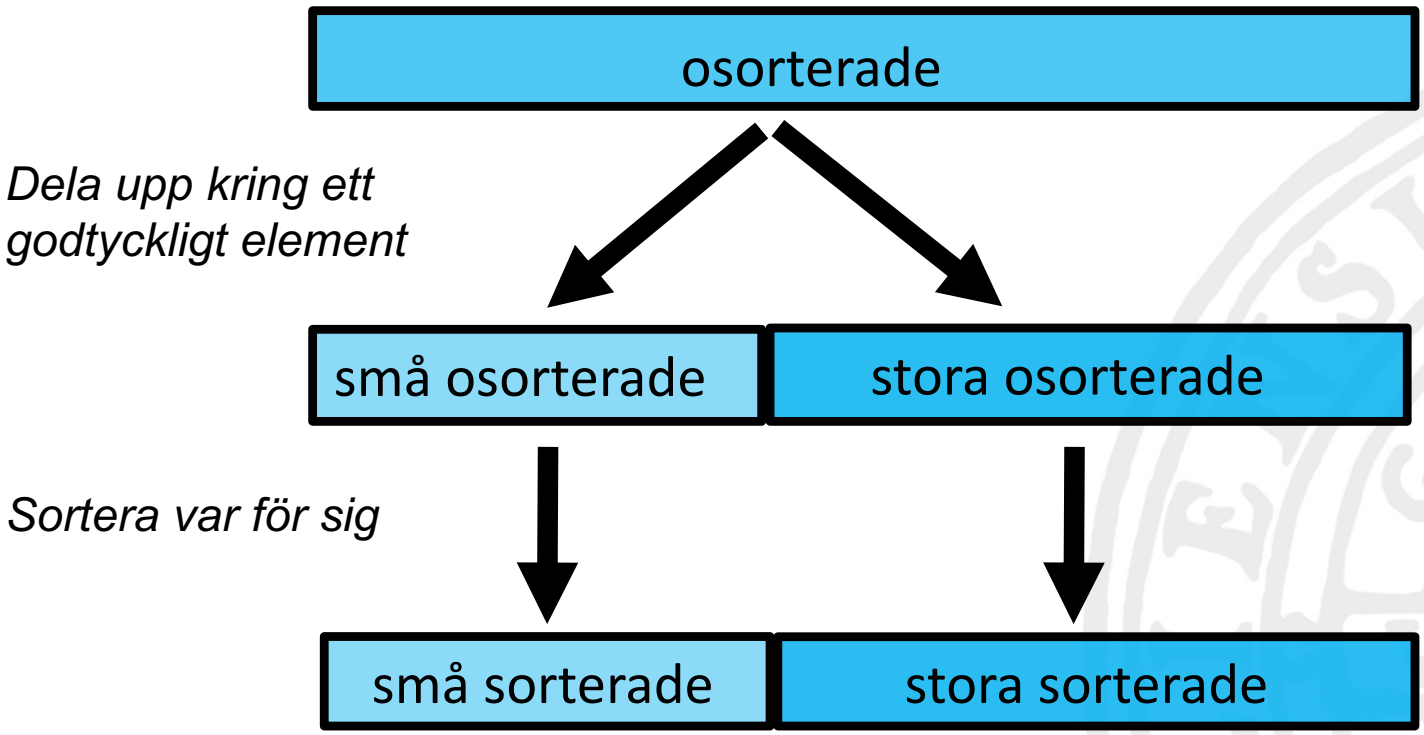
```
sort_functions = [ins_sort_rec, merge_sort]


for sort in sort_functions:
    print('\n', sort.__name__)
    for n in [1000, 2000, 4000, 8000]:
        lst = []
        for i in range(n):
            lst.append(random.random())
        tstart = time.perf_counter()
        lst = sort(lst)
        tstop = time.perf_counter()
        print(f" Time for {n}\t : {tstop - tstart:4.2f}")
```

UPPSALA UNIVERSITET

Testkörning

En annan sorteringsmetod

Idé: Börja med att ordna om elementen så att ”små” ligger (osorterade) till vänster och ”stora” ligger (osorterade) till höger.



UPPSALA UNIVERSITET

Partition sort

```
def psort(lst):
    _psort(lst, 0, len(lst)-1)
    return lst

def _psort(lst, n, m):
    if m > n:
        ip = partition(lst, n, m)
        _psort(lst, n, ip-1)
        _psort(lst, ip+1, m)
```

Likheter och skillnader mot mergesort?

UPPSALA UNIVERSITET

Partitioneringen

```
def partition(lst, n, m):
    if m>n:
        p = lst[n]
        i = n
        j = m
        while j > i:
            while j>i and lst[j] > p:
                j -= 1
            lst[i] = lst[j]
            while i < j and lst[i] < p:
                i += 1
            lst[j] = lst[i]
            j -= 1
        lst[i] = p
        return i
```

Anmärkning: Metoden kallas ofta för *quicksort.*

Demo

UPPSALA UNIVERSITET

Sammanfattning

- Tidmätningar kan användas för att verifiera teoretiska resultat.
- De teoretiska resultaten för instickssortering och samsortering stämmer mycket bra i praktiken.
- $\Theta(n \log n)$ är mycket bättre än $\Theta(n^2)$!
- Bra att balansera algoritmerna.
- Rekursera inte över långa listor.
- Inga problem med rekursiondjupet i mergesort och quicksort.

UPPSALA UNIVERSITET

The end

.

UPPSALA UNIVERSITET

Sammanfattning av modul MA1

UPPSALA UNIVERSITET

Rekursion

- Dela upp problemet i (ett eller flera) delproblem av samma slag men mindre.
- Lös delproblemen.
- Kombinera lösningarna av delproblemen till en lösning av ursprungsproblemet.

Det måste finnas ett eller flera basfall.

Det blir vanligtvis bäst kod om man använder de enklaste basfallen (t.ex. 0 i stället för 1 i Hanois torn eller en tom lista i stället för en lista med ett element).

UPPSALA UNIVERSITET

Varför rekursion

- Generell teknik för problemlösning.
- Lätt att hitta lösningen.
- Lätt att hitta effektiva lösningar.
- Naturligt i många sammanhang.
- Särskilt bra vid rekursivt definierade strukturer.

Men

- Lätt att producera hopplöst långsamma program.
- Problem med rekursionsdjup.

UPPSALA UNIVERSITET

Vad är avgörande?

Avgörande är *antalet* delproblem och *storleken* på delproblemen.

- Om vi har två eller fler delproblem med *nästan samma storlek* som ursprungsproblemet så har vi en exponentiell tillväxt.
  Exempel: Hanois torn, Fibonacci.

- Oftast bättre med *två* delproblem som har *halva storleken* är *ett* delproblem som är *nästan lika stort* som ursprungsproblemet.
  Exempel: mergesort – insticksortering

- I regel bra att *balansera* storleken på delproblemen.

- Undvik rekursion över långa strukturer – problem med stackdjupet.

UPPSALA UNIVERSITET

Asymptotisk notation

Ett sätt att beskriva hur tiden för en algoritm växer med problemstorleken oberoende av dator, programmeringsspråk etc.

När använder man $\mathcal{O}, \Theta,$ repektive Ω ?

- Θ ger mest information.
- $\mathcal{O}$ är en *övre* gräns (ej nödvändigtvis ”tät”).
- Ω är en *undre* gräns.

UPPSALA UNIVERSITET

Ordo, Omega eller Theta?

- Om du har en ”bra” $\mathcal{O}$-funktion kan den användas för att säga att en algoritm är bra.
  Exempel: Om du hittat på en sorteringsalgoritm är det bra att kunna säga att den är $\mathcal{O}(n \log n)$ men meningslöst att säga att den är $\mathcal{O}(n^2)$.

- Om du vill säga att en algoritm är dålig kan du använda Ω.
  Exempel: Om någon kommer med en sorteringsalgoritm kan du säga att den är inte så bra om den är $\Omega(n^2)$ och att den är usel om den är $\Omega(n^3)$.
  Det är dock meningslöst att säga att den är $\Omega(n \log n)$.

- Θ är mest informativ. Använd om möjligt.

UPPSALA UNIVERSITET

Vad är bra och vad är dåligt?

- $\Theta(a^n)$ är *dåligt* om $a > 1$.
- $\Theta(\log n)$ är *mycket bättre* än $\Theta(n)$.
- $\Theta(n \log n)$ är *mycket bättre* än $\Theta(n^2)$.
- $\Theta(n)$ är *inte så mycket bättre* än $\Theta(n \log n)$.

UPPSALA UNIVERSITET

Tiduppskattningar

Om vi vet att en algoritm är $\Theta(f(n))$ så kan vi uppskatta tidsåtgången $t(n)$ för stora problem med uttrycket

$$t(n) = c \cdot f(n)$$

och uppskatta konstanten genom att mäta tiden för något n.

Man bör verifiera modellen genom att mäta tiden för några olika n.

Använd inte för små n – det blir säkrare uppskattningar ju större värden man mäter för.

UPPSALA UNIVERSITET

Exempel

För att verifiera komplexiteten för en viss algoritm är det bra att ha strategi för hur n ska varieras.

Att dubbla värdet passar bra om man har (tror sig ha) polynomial komplexitet:

- För en $\Theta(n)$-algoritm så bör tiden dubblas om n dubblas.
- För en $\Theta(n^2)$-algoritm så gäller $\frac{t(2n)}{t(n)} = \frac{c \cdot (2n)^2}{c \cdot n^2} = 4$
- För en $\Theta(n^3)$-algoritm så gäller $\frac{t(2n)}{t(n)} = \frac{c \cdot (2n)^3}{c \cdot n^3} = 8$

UPPSALA UNIVERSITET

Exempel

- För en $\Theta(\log n)$-algoritm så är det bra att kvadrera: $\frac{\log n^2}{\log n} = 2$

- För en $\Theta(n \log n)$-algoritm är en dubblering användbar:

$$\frac{t(2n)}{t(n)} = \frac{c{\cdot}2n \log 2n}{c{\cdot}n \log n} = 2 \cdot \frac{\log 2 + \log n}{\log n} = 2 + \frac{1}{\log n} \approx 2 \text{ för stora } n.$$

- För en exponentiell komplexitet, dvs $\Theta(a^n)$ så passar n och $n+1$:

$$\frac{c \cdot a^{n+1}}{c \cdot a^n} = a$$

Behöver vi bry oss?

Ja! Det finns fortfarande många problem där datorkraften begränsar oss.

Ett axplock:

- fysik och teknik: aerordynamik, väderprognoser, …
- biologi: bioinfomatik, genomsekvensering, …
- realtidssystem: robotar, självkörande bilar, …
- animation: datorspel, filmindustri, …
- infomationssökning: google, …

UPPSALA UNIVERSITET

Men

Citat från ”*The elements of programming style*” av Kernighan och Ritchie:

- Correctness is much more important than speed!
- Do not sacrifice clarity for small efficiency gains!
- Do not sacrifice simplicity for small efficiency gains!
- Do not sacrifice modifiability for small efficiency gains!
- If the program is too slow: Find a better algorithm!

Några nya Python-detaljer

Man kan deklarera funktioner inuti funktioner.

Första exemplet:

```
def power(x, n):
    def sqr(x):
        return x*x
    if n<0:
        return 1./power(x, -n)
    elif n==0:
        return 1
    elif n%2==0:
        return sqr(power(x, n//2))
    else:
        return x*sqr(power(x, (n-1)//2))
```

UPPSALA UNIVERSITET

Nya Python-detaljer

Andra exemplet:

```
def fib(n):
    memory = {0:0, 1:1}

    def _fib(n):
        if n not in memory:
            memory[n] = _fib(n-1) + _fib(n-2)
        return memory[n]

    return _fib(n)
```

Nya Python-detaljer

Funktioner är *objekt* som kan lagras i variabler, listor, lexikon, …

```
sort_functions = [ins_sort_iter, merge_sort, psort, sorted]

for sort in sort_functions[1:]:
    print(f'\n ***{sort.__name__}***')
    for n in [100000, 200000, 400000, 800000]:
        lst = []
        for i in range(n):
            lst.append(random.random())
        tstart = time.perf_counter()
        lst = sort(lst)
        tstop = time.perf_counter()
        print(f" Time for {n}\t : {tstop - tstart:4.2f}")
```

UPPSALA UNIVERSITET

The end

.