

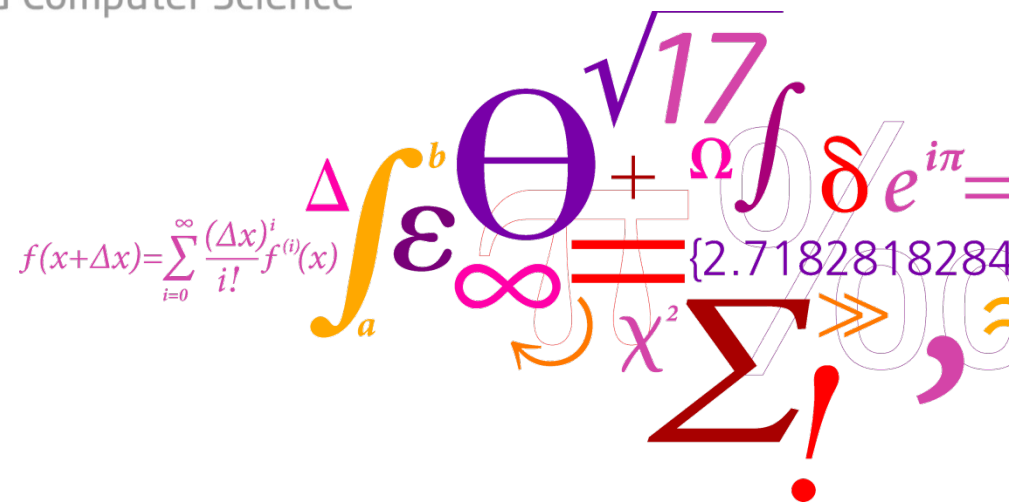
Projekt i software-udvikling (02362)

Forår 2021

Ekkart Kindler

DTU Compute

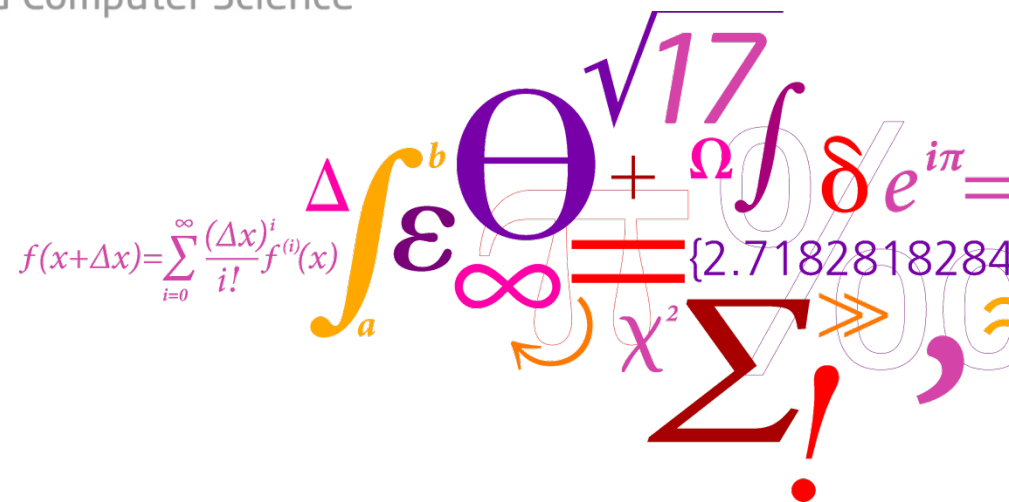
Department of Applied Mathematics and Computer Science



IX. Algoritmer

DTU Compute

Department of Applied Mathematics and Computer Science



A collage of mathematical symbols and formulas. The central formula is the Taylor series expansion: $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$. Other symbols include the integral $\int_a^b$, the Greek letter ϵ , the Greek letter θ , the square root $\sqrt{17}$, the Greek letter Ω , the Dirac delta δ , the exponential function $e^{i\pi}$, the infinity symbol ∞ , the chi-squared symbol χ^2 , the summation symbol Σ , the greater-than symbol $>$, the comma $,$, and the exclamation mark $!$.

Motivation

- Implementeringer i programmer er ofte meget teknisk
- Der er mange dataer som kan ødelægges, når man ikke bruger dem rigtigt
- Man ved ikke hvad der er vigtigt og ikke så vigtigt
- Programmet er svært at forstå
- Programmet kan nemt ødelægges

- Bagved interfaces ligger der ofte nogle mere eller mindre komplicerede algoritmer som implementerer den ønskede funktionalitet
- Ofte har andre implementeret denne funktionalitet, så at vi bare kan bruge den og kombinere det, til det funktionalitet vi har brug for

Det gælder også de algoritmer, som vi diskuterer her.

- Men for at fremme jeres sans for (den måde dataloger tænker og taler om) algoritmer diskuterer vi dem her alligevel: sorteringsalgoritmer

- Der er forskellige algoritmer til at sortere en array
- En af de mest enkle (men ikke så hurtige) algoritmer er **Bubble Sort**.

Algoritmens idé:

Sålænge der findes tal ved siden af hinanden, som ikke er i den rigtige rækkefølge:

Byt dem

Lidt mere systematisk:

Vi antager at listen ligger i et array

```
int[] list = ...
```

og

```
int size = ...
```

er den aktuelle længde af listen

```
boolean swapped;  
do {  
    swapped = false;  
    for (int i=0; i+1<size; i++) {  
        if (values[i] > values[i+1]) {  
            int v = values[i];  
            values[i] = values[i+1];  
            values[i+1] = v;  
            swapped = true;  
        }  
    }  
} while (swapped);
```

Det her kan (og skal)
gøres lidt bedre!

Observationer:

- i første omgang "vandrer" det største tal til den højeste position
- i næste omgang "vandrer" det næststørreste tal til anden højeste position
- o.s.v.
- derfor er vi ikke nødt til at køre løkken igennem hele arrayet hver gang

Første iteration

9	8	1	6	7	4	5	2	3
---	---	---	---	---	---	---	---	---



8	9	1	6	7	4	5	2	3
---	---	---	---	---	---	---	---	---



8	1	9	6	7	4	5	2	3
---	---	---	---	---	---	---	---	---



8	1	6	9	7	4	5	2	3
---	---	---	---	---	---	---	---	---



...

8	1	6	7	4	5	2	3	9
---	---	---	---	---	---	---	---	---

Anden iteration

8	1	6	7	4	5	2	3	9
---	---	---	---	---	---	---	---	---



1	8	6	7	4	5	2	3	9
---	---	---	---	---	---	---	---	---



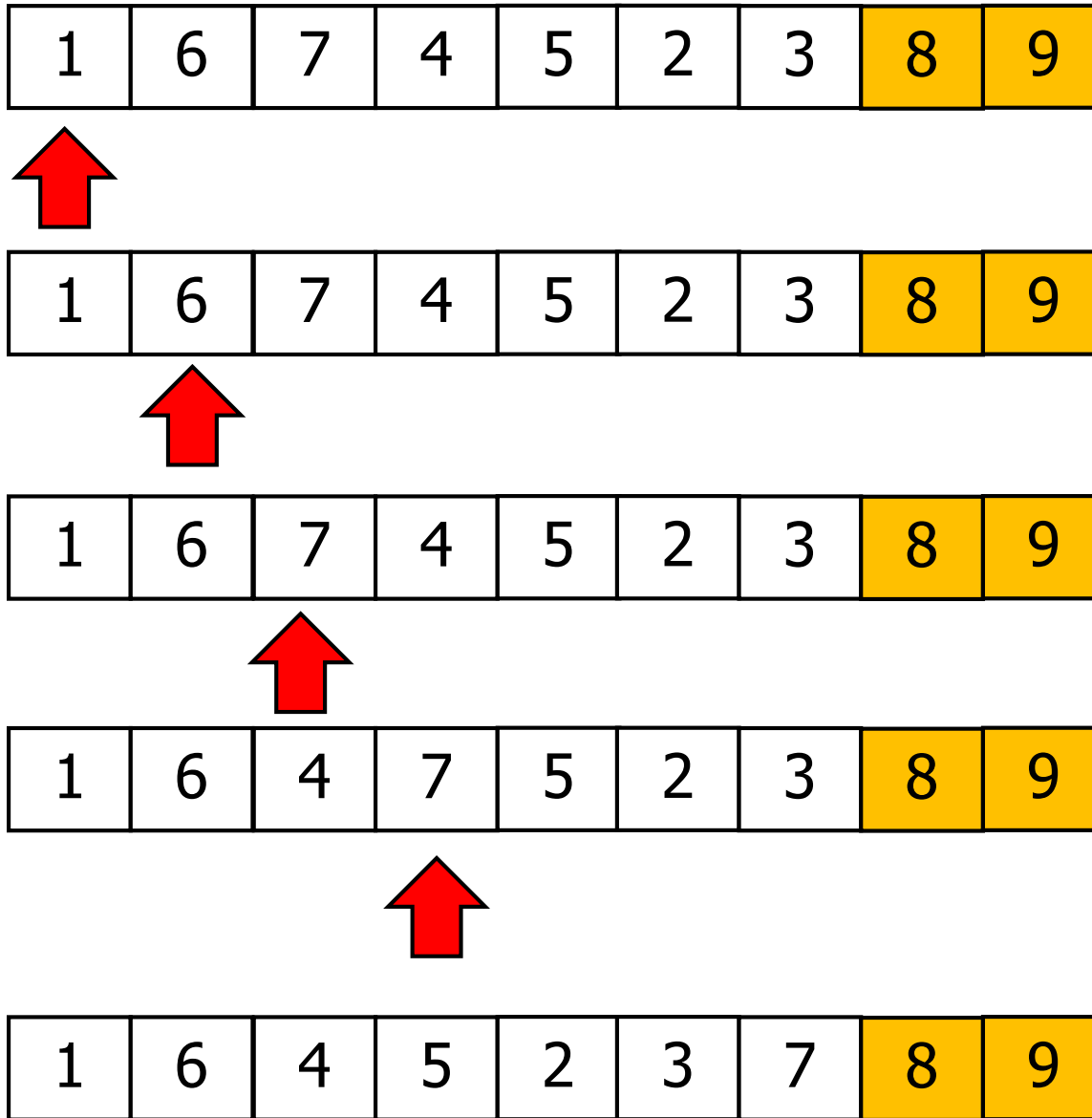
1	6	8	7	4	5	2	3	9
---	---	---	---	---	---	---	---	---



...

1	6	7	4	5	2	3	8	9
---	---	---	---	---	---	---	---	---

Tredje iteration



I hver iteration kan vi stoppe en position tidligere! Det kan man realisere med to for-løkker i stedet for en do-while- og en for-løkke!

- Hvis man drejer arrayet 90 grader, så "stiger de største tal op som bobbler"!

Derfor hedder det **bubble sort**

- Der er nogle varianter af bubble sort: f.eks **shaker sort** (hvis man skifter mellem de største og mindste bobbler hver iteration)
- Bubble Sort bruger kvadratisk tid $O(n^2)$ i arrays størrelsen

9
8
7
3
2
5
4
6
1

andre algoritmer er meget hurtigere
→ Quicksort, som vi diskuterer senere

- Hvordan kan man finde positionen af et element i listen (**indexOf()**)?
- Hvor effektiv (engl. efficient) er det?
- Kan man gøre det hurtigere, hvis listen er sorteret?

Discussion på tavlen!
→ lineær søgning

Discussion på
tavlen!

Når man har et sorteret array af tal, kan man søge efter et tal meget hurtiger:

1. "Del" arrayet i midten
2. Hvis det tal vi søger efter er mindre end tallet i midten, søg i den lave halvdel og fortsæt med skridt 1.
3. Hvis det tal vi søger efter er større end tallet i midten, søg i den høje halvdel og fortsæt med skridt 1.
4. Hvis tallet i midten er tallet vi søger efter, så har vi fundet det

Input

`int[] values = ...;` Array med alle værdiger som

`int lower = 0;` Laveste index (typisk 0)

`int upper = ...;` Højeste index

```
while (lower <= upper) {  
    int middle = (lower + upper) / 2;  
    if (value == values[middle]) {  
        return middle;  
    } else if (value < values[middle]) {  
        upper = middle;  
    } else {  
        lower = middle;  
    }  
}  
  
return -1;
```

Er der noget galt?

Løkken terminerer
ikke altid!

```
while (lower <= upper) {  
    int middle = (lower + upper) / 2;  
    if (value == values[middle]) {  
        return middle;  
    } else if (value < values[middle]) {  
        upper = middle - 1;  
    } else {  
        lower = middle + 1;  
    }  
}  
  
return -1;
```

Tjek altid om løkken
terminerer!

- Binær søgning er **meget** hurtiger (logaritmisk $O(\log(n))$ end lineær søgning (lineær $O(n)$))

Anden **prototype** med **fokus** på:

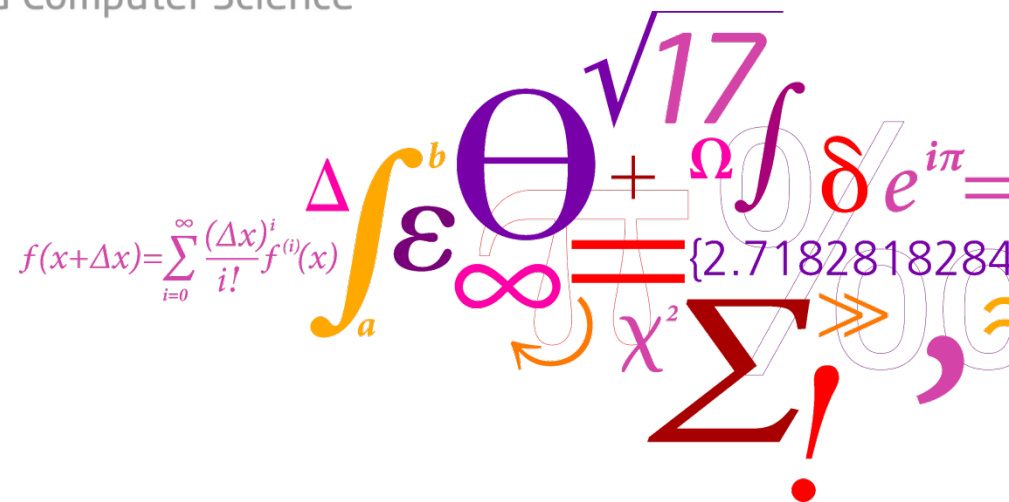
- Finpudsning af database-tilknytning (V4a) og loade spilleplader fra filer (V4b) med vægge og aktioner
- Spillelogik:
 - Man skal kunne afslutte (vinde) et spil ("checkpoints" og bogføring af hvilke "checkpoints" en spiller kom forbi allerede)
 - Flere forskellige aktioner som er tilknyttet forskellige felter
 - Evtl. robotaktioner imod andre (laser)

Hvis I vil og kan, må aflevering A4 indeholde en foreløbig rapport (med det er ikke påkrævet)! Men hvis I afleverer en foreløbig rapport, så får I også nogle kommentarer, inden I skal aflevere jeres endelige projekt!

3. Rekursion

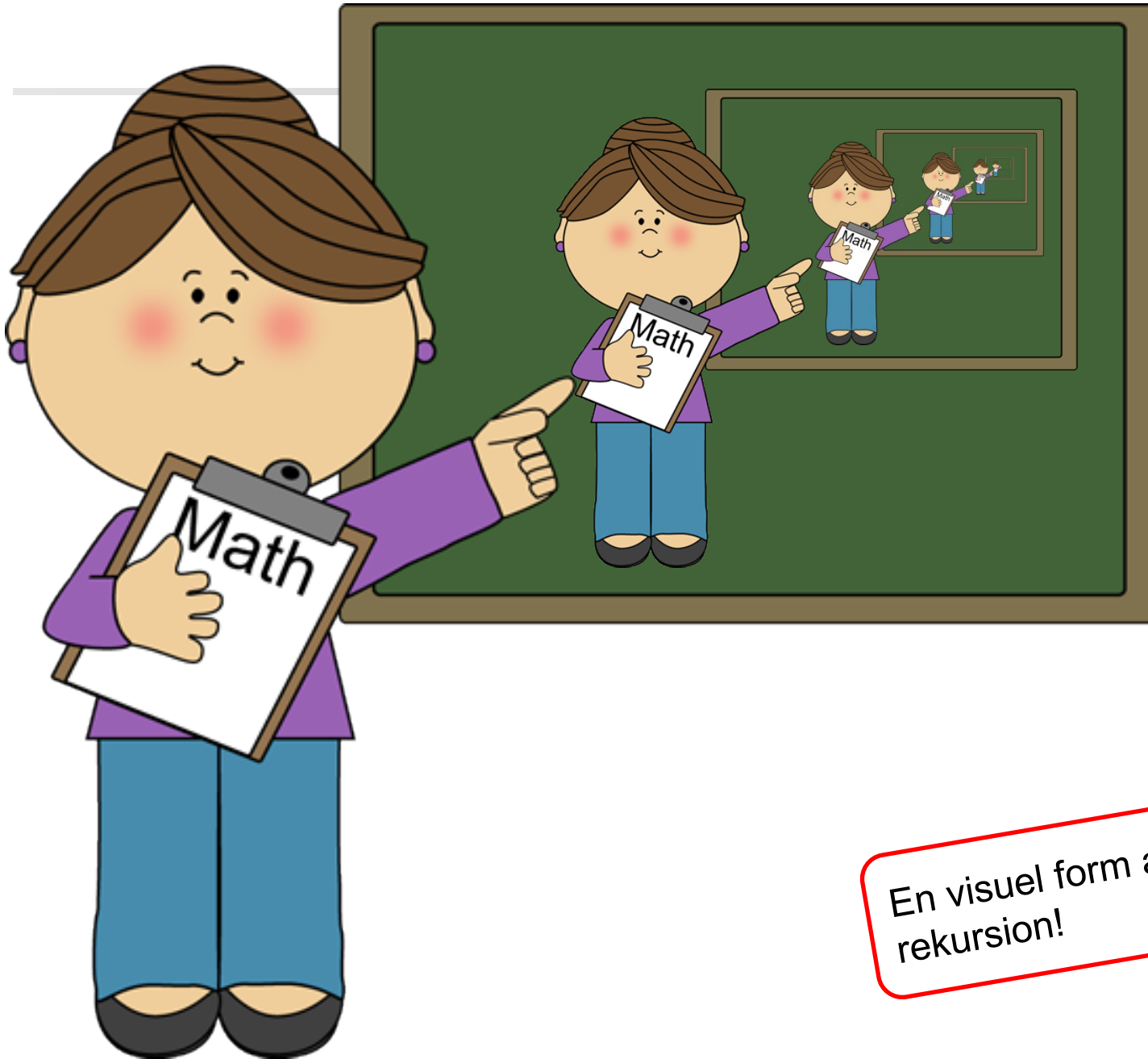
DTU Compute

Department of Applied Mathematics and Computer Science



A collage of mathematical symbols and formulas. The central element is the Taylor series expansion of the exponential function:
$$f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$$
 Surrounding this are various other mathematical symbols and expressions in different colors and sizes, including:

- Δ (pink)
- $\int_a^b$ (yellow)
- ε (purple)
- Θ (purple)
- $\sqrt{17}$ (purple)
- $+$ (pink)
- Ω (pink)
- $\int$ (purple)
- $\delta e^{i\pi} =$ (red)
- ∞ (pink)
- χ^2 (purple)
- $\sum$ (red)
- $!$ (red)
- $\{2.7182818284\}$ (purple)
- $>$ (yellow)
- $,$ (purple)



En visuel form af en
rekursion!

- ... opstår, når noget bliver defineret igennem sig selv

... bare mindre

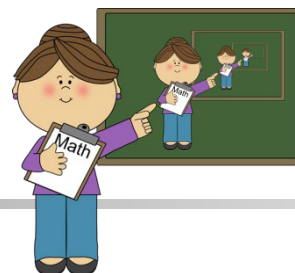
- ... kan gå grueligt galt

... hvis man ikke passer på

- ... kan løse nogle problemer meget elegant

... når man først har fundet ud af det

- ... kan være meget sjovt.



- ... bruger vi oftere end man tror

... uden at vi lægger mærke til det; især i matematik
(der kaldes det for en induktiv definition)

- ... har vi brugt allerede!

```
private class StackElement {
```

```
    private Integer value;
```

```
    private StackElement next;
```

```
    public StackElement(Integer value,
```

```
        StackElement next) {
```

```
        this.value = value;
```

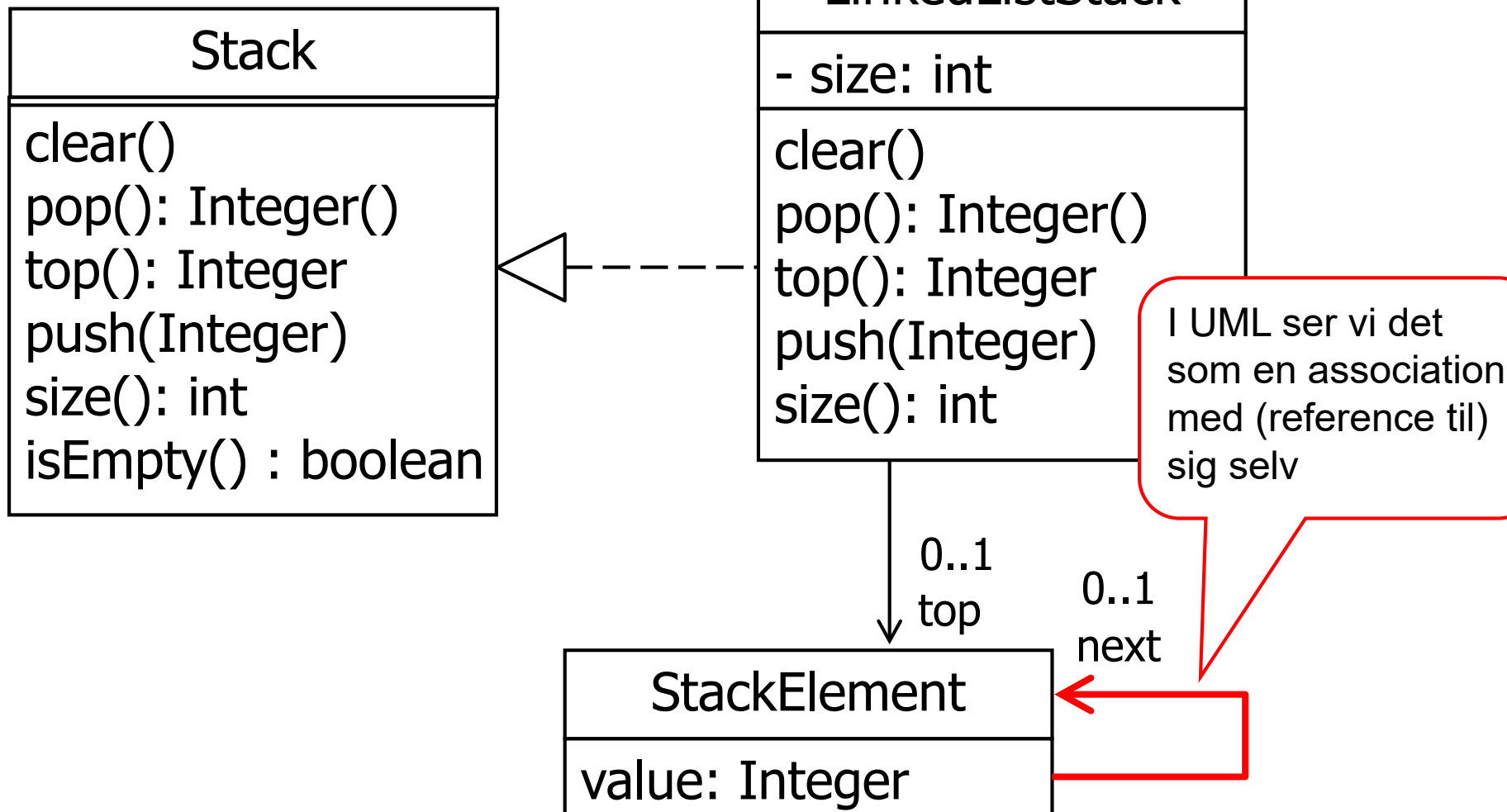
```
        this.next = next;
```

```
    }
```

```
}
```

Vi bruger klasse
StackElement i sin egen
definition: som type af
attributet **next**

- Som enkelt-hægtede liste



- ... bruger vi oftere end man tror

... uden at vi lægger mærke til det; især i matematik
(selvom de kalder det induktiv definition)

- ... har vi brugt allerede: som rekursive datatyper
(StackElement se slide 24 og 25)

Rekursive datatyper, er heller ikke noget svært eller
overraskende, når man først har vænnet sig til det!

- Rekursive datatyper bruges ofte sammen med
rekursive funktioner (metoder)

- Beregn størrelse af en stak som er realiseret som enkelt-hægtede liste

→ Diskussion på tavlen

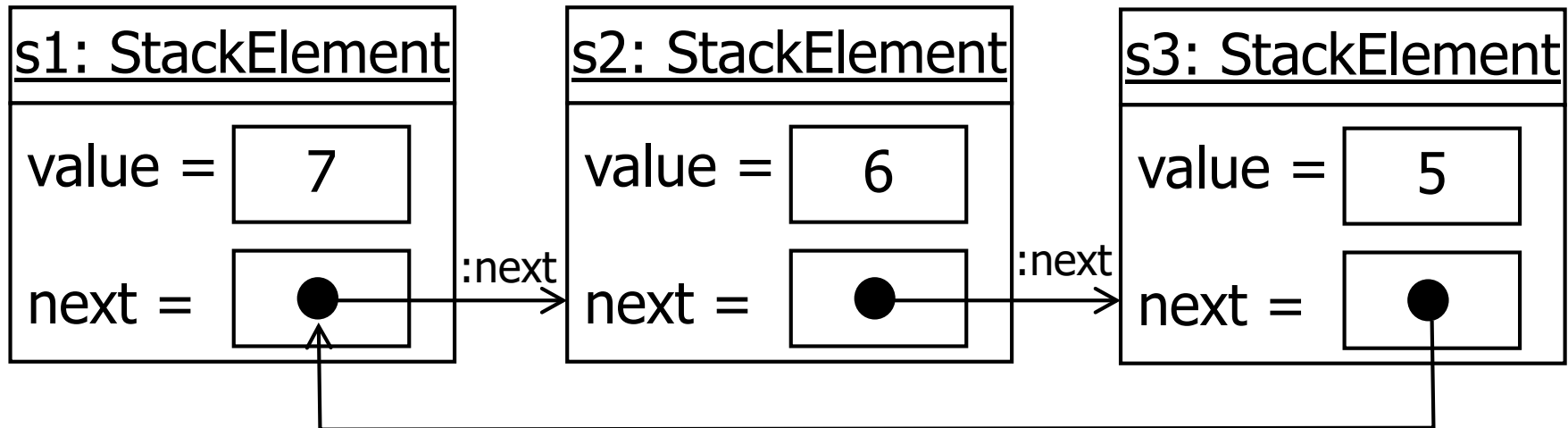
```
static int size(StackElement element) {  
    if (element == null) {  
        return 0;  
    } else {  
        return size(element.next) + 1;  
    }  
}
```

Må man det?

Instance ("Objectdiagram")

med "tre elementer"

Hvad ville ske, når
vi kalder
size(s1) på dette
struktur?



Call stak af `size(s1)`

```
size(s1) ->  
  size(s2) ->  
    size(s3) ->  
      size(s1) ->  
        size(s2) ->
```

Lige som vores cykliske
liste forstætter opkald af
`size()` uendeligt ...

... indtil det stopper med en
`StackOverflowError!`

- Vi skal passe meget på, at vores enkelt-hægtede lister ikke kan blive cykliske!
- Ellers ville vores rekursive kald af `size` aldrig (undtaget `StackOverflowError`) stoppe: ikke terminere!

Men hvis vi implementerede `size()` med en **while**-løkke ville vi have samme problem! Faktisk ville være det værre, da denne uendelige løkke ikke ville stoppe med en **StackOverflowError**!

Definition: $factorial(n) =$

$$n! = n * (n-1) * \dots * 3 * 2 * 1$$

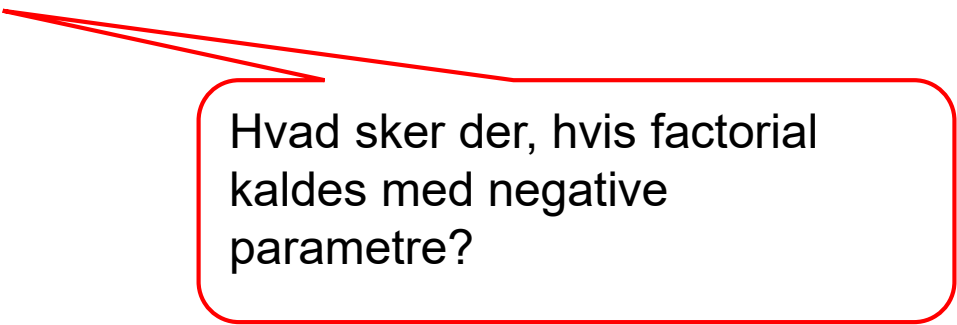
På engelsk hedder
fakultet **factorial**

Når vi bruger ... for at
definere noget, så er der
næsten altid en **rekursiv**
(matematisk: **induktiv**)
definition bagved

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

Her er der **rekursion** igen

```
private static int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    } else {  
        return n * factorial(n-1);  
    }  
}
```



Hvad sker der, hvis factorial kaldes med negative parametre?

```
private static int factorial(int n) {  
    if (n == 0) {  
        return 1;  
    } else if (n > 0) {  
        return n * factorial(n-1);  
    } else {  
        return -1;  
    }  
}
```

Defensive programming; det ville være endnu bedre at kaste en undtagelse (`IllegalArgumentException`).

```
private static double factorial(int n) {  
    if (n == 0) {  
        return 1;  
    } else if (n > 0) {  
        return n * factorial(n-1);  
    } else {  
        return -1;  
    }  
}
```

Nogle detaljer om **primitive datatyper** i Java:
int og **long**: når tal bliver for store/små, bliver de forkerte (uden advarsel)!

Det starter allerede, at gå galt med **int** på **factorial(14)**!

Når man bruger **double** og **float** eksisterer værdier som **Infinity** og **NaN** (Not a Number), som viser, at resultatet er for stort eller forkert.

Fakultets-funktion terminerer for alle ikke negative heltal

Bevis: Matematisk induktion:

Induktionsbasis:

- **factorial(0)** terminerer:
body med $n=0$ returnerer 1 og terminerer
- Induktionsantagelse:
factorial(n) terminerer
- Induktionsskridt: $n \rightarrow n+1$
factorial(n) terminerer pga. antagelsen; body returnerer $(n+1) * \text{factorial}(n)$ og terminerer

... og pga defensiv programmering også for negative tal!

For termineringen af rekursion er det vigtigt, at

- der er et eller flere basistilfælde (engl. base case) hvor rekursionen terminerer umiddelbart:

```
if (n == 0) { return 1; }
```

- i alle andre tilfælde (engl. cases) kommer en rekursiv opkald lidt nærmere til basistilfælde

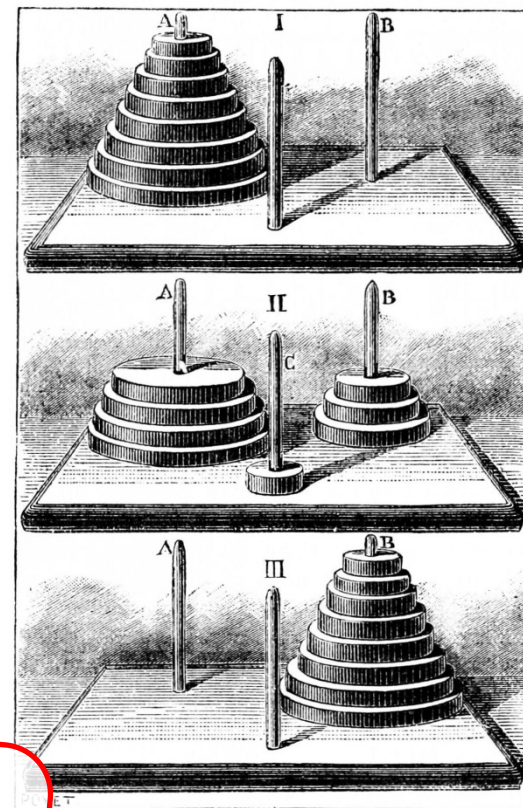
```
else { return n * factorial(n-1); }
```

Hvis man passer på det, er rekursion ikke så farligt og bliver noget smukt!

engl. Towers of Hanoi

Hanois tårne: Matematisk "puslespil":

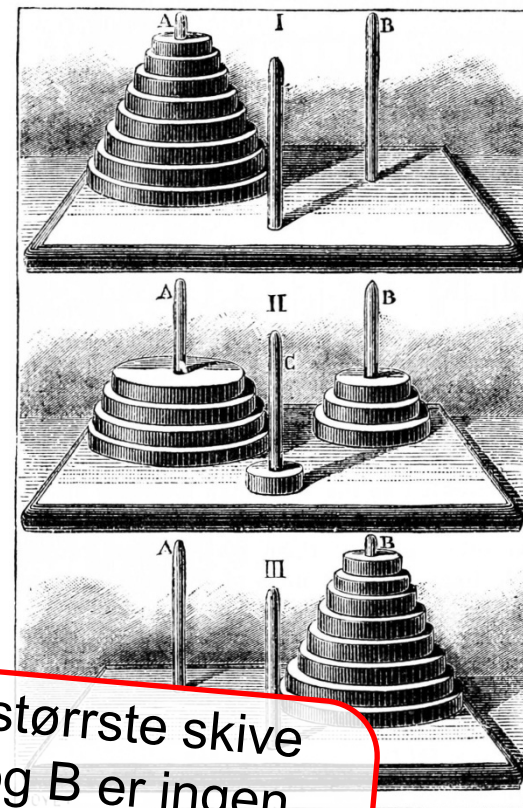
- Stativ med tre pinde (A , B og C)
- På pind A er der n skiver med aftagende størrelse (set nedefra)
- **Opgave:**
Flyt alle n skiver fra A til B , men
 - kun en skive ad gangen
 - en skive må aldrig ligge på en skive som er mindre end den selv



Man må gerne "parkere" skiveren på den tredje pin C ; men regel med størrelsen gælder også der

Flyt n skiver fra A til B

1. Flyt de øverste $n-1$ skiver fra A til C (ifølge reglerne)
2. Så flyt den nederste (størreste) skive n fra A til B (som er OK ifølge reglerne)
3. Nu flyt de $n-1$ skiver fra C til B



Sammen med rekursion er det løsningen fordi 1. og 3. er samme problemstilling bare for $(n-1)$ skiver

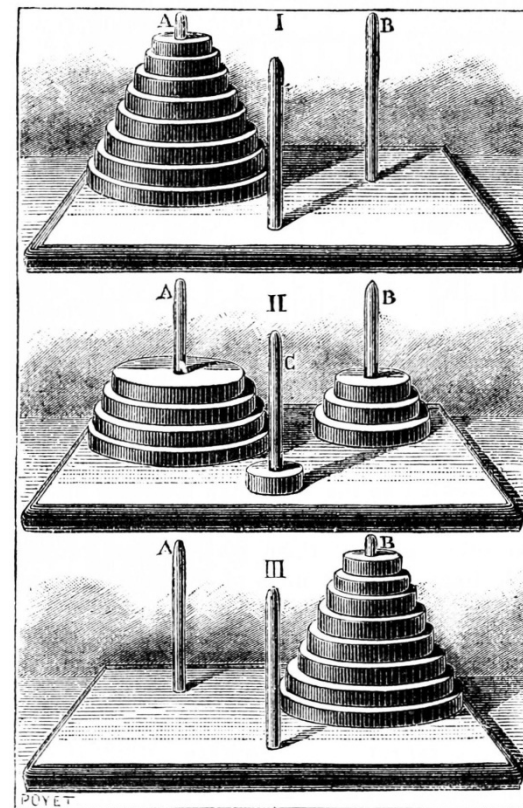
Bemærk at den største skive på bunden af A og B er ingen forhindring, da den er jo størrest og alle skiver må ligge på den

Flyt n skiver fra A til B

1. Flyt de øverste $n-1$ skiver fra A til C
(ifølge reglerne)
2. Så flyt den nederste (størreste) skive n fra A til B
(som er OK ifølge reglerne)
3. Nu flyt de $n-1$ skiver fra C til B

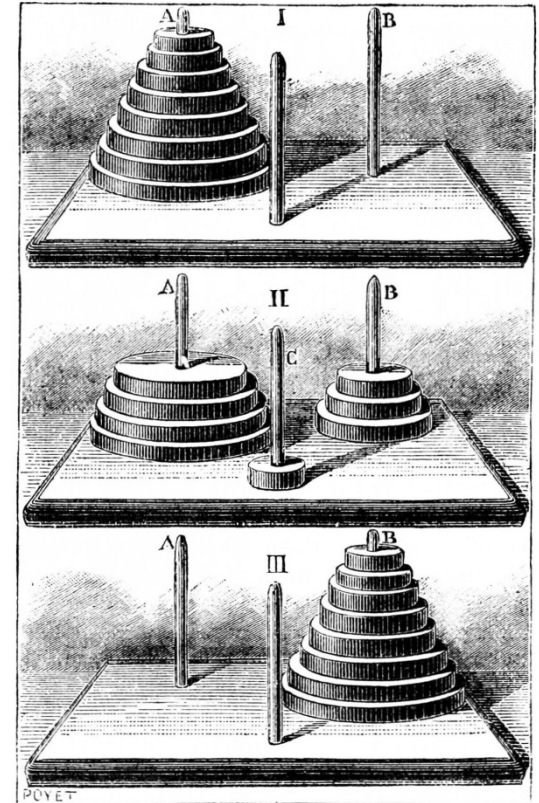
Nu kan B bruges til at parkere skiverne!

Nu kan A bruges til at parkere skiverne!



Flyt n skiver fra A til B

1. Flyt de øverste $n-1$ skiver fra A til C
(ifølge reglerne)
2. Så flyt den nederste (størreste) skive n fra A til B
(som er OK ifølge reglerne)
3. Nu flyt de $n-1$ skiver fra C til B



```
public static void moveTower(  
    int n, Position a, Position b,  
    Position c) {  
    if (n == 1) {  
        moveDisc(1, a, b);  
    } else {  
        moveTower(n-1, a, c, b);  
        moveDisc(n, a, b);  
        moveTower(n-1, c, b, a);  
    }  
}
```

```
public static void moveTower (  
    int n, Position a, Position b,  
    Position c) {  
    if (n <= 0) { return; }  
    else if (n == 1) {  
        moveDisc(1, a, b);  
    } else {  
        moveTower(n-1, a, c, b);  
        moveDisc(n, a, b);  
        moveTower(n-1, c, b, a);  
    }  
}
```

Lidt defensiv
programmering igen
(termineringen, når
n ikke er positiv)

Base case

Rekursion

- For n skiver bruger man $2^n - 1$ flyttetræk

Bevis gennem matematisk induktion

- Rekursion ser harmløs ud:
men det kan blive til rimelig mange opkald

For $n=64$ og et sekund per træk (det er jo fysiske skiver som skal flyttes) tager det meget mere end **universets alder** at komme igennem hele flytning.

Brug af rekursion til noget mere anvendeligt:

4. Quicksort (Lister, Sortering, og Søgning)

DTU Compute

Department of Applied Mathematics and Computer Science

A collage of mathematical symbols and formulas. It includes the Taylor series expansion $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$, an integral $\int_a^b \epsilon$, a large purple Θ , a square root $\sqrt{17}$, a plus sign $+$, a Greek letter Ω , a delta δ , an exponential $e^{i\pi}$, an equals sign $=$, a set of curly braces $\{2.7182818284\}$, a Greek letter χ^2 , a summation symbol Σ , a greater-than sign $>$, and an exclamation mark $!$.

Quicksort (→ Hoare 1959)

9	8	1	6	4	5	7	2	3
---	---	---	---	---	---	---	---	---

Vælg et tilfældigt **pivot** element



Partitioner arrayet, så at alle tal lavere eller lige pivot-elementet ligger på venstre side og alle større eller lige på højre side

2	3	1	4	5	9	7	6	8
---	---	---	---	---	---	---	---	---



Sorter partitionerne uafhængigt af hinanden **rekursiv**

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---

Så er hele arrayet sorteret

```
private void quicksort(int lower, int upper) {  
    if (lower >= upper) {  
        return;  
    }  
  
    int partition = partition(lower, upper);  
  
    quicksort(lower, partition);  
  
    quicksort(partition+1, upper);  
}
```

Bemærk at det er vigtigt at begge partitioner bliver mindre! Begge skal have mindst et element

Det er faktisk den svære del af opgaven!

Partition: Idé

Vælg et tilfældigt **pivot** element

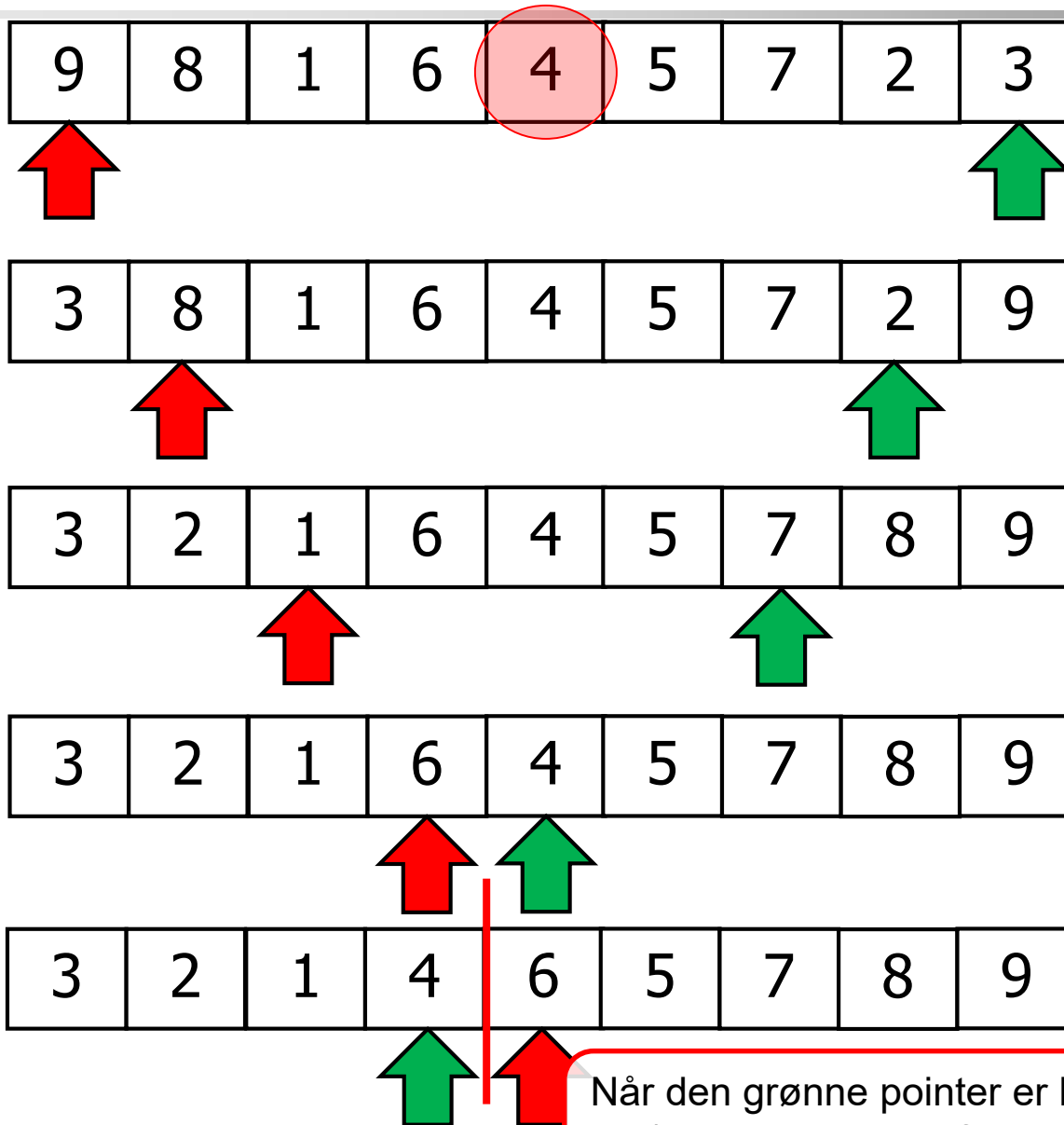
Flyt pointere ind indtil de rammer et element $\geq$ (left) or $\leq$ (right) pivot

Byt elementerne på pointere og flyt dem en position videre ind

⋮

Flyt pointere indtil de rammer et element $\geq$ (left) or $\leq$ (right) pivot

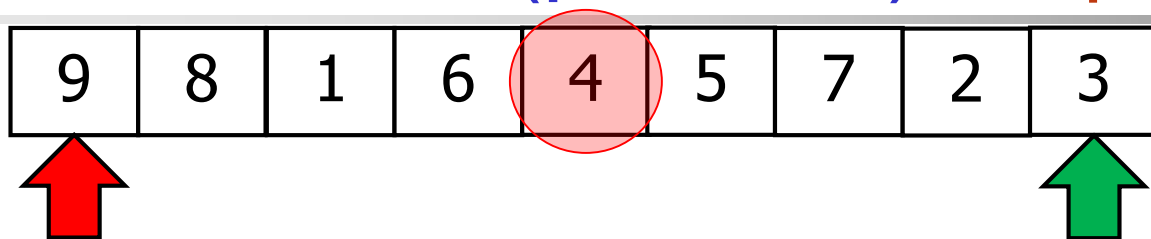
Byt elementerne på pointere og flyt dem en position videre ind



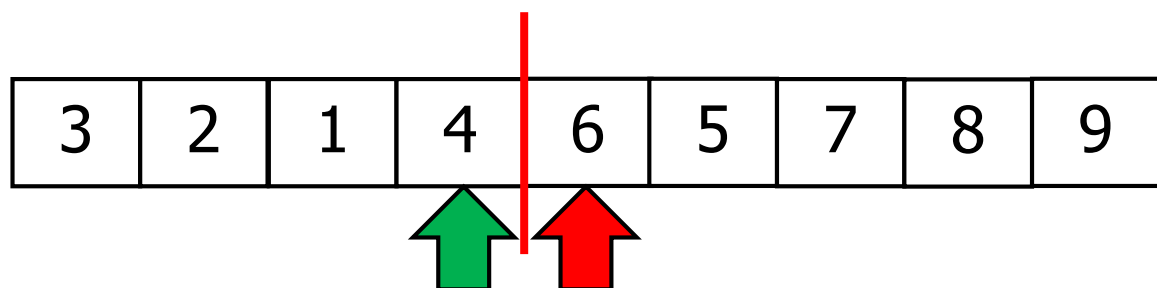
Når den grønne pointer er lavere (eller lige) end den røde, afbryd, og returner den grønne som partitionsgrænse

- Mange versioner af Quicksort vælger det første eller det sidste element som pivot element; men det er meget ueffektiv, hvis arrayet er sorteret allerede
- Man skal helst sørge for at begge partitioner bliver næsten samme størrelse (vælg enten midten som pivot eller helt tilfældigt) → Men se næste slide!!
- Hvis man gør det, så er den gennemsnitlige kompleksitet af Quicksort $O(n * \log(n))$
- Men worst case kompleksitet er $O(n^2)$ og det rammer, når man har sorterede arrays og vælger det første eller sidste element som pivot element

Diskussion (partition)



⋮



lower

p

upper

- Det er vigtigt at $lower \leq p < upper$
- Derfor skal pivot elementet ikke være det sidste i arrayet; men det er nemt, byt pivot elementet med det første

Ellers terminerer
Quicksort muligvis ikke!

Som sagt skal man helst ikke
vælge det første som pivot,
men det er i orden at bytte.

```
private void quicksort(Comparable<E>[] arr, int lower, int upper) {  
    if (lower < upper) {  
  
        int i = lower, j = upper;  
        E pivot = (E) arr[(i+j)/2];  
        do {  
            while(arr[i].compareTo(pivot) < 0) i++;  
            while(pivot.compareTo((E) arr[j]) < 0) j--;  
            if(i <= j) {  
                E temp = (E) arr[i];  
                arr[i] = arr[j];  
                arr[j] = temp;  
                i++;  
                j--; }  
        } while (i <= j);  
  
        quicksort(arr, lower, j);  
        quicksort(arr, i, upper);  
    } }  
}
```

En mulig implementering
(som vi ikke diskuterer i
detaljer her)!

- Der er andre sorteringsalgoritmer, som har også en worst case kompleksitet som er $O(n * \log(n))$:
fx. *merge sort* eller *heap sort*

Men de er lidt sværere at forklare! Derfor bruger vi Quicksort som eksempel her.