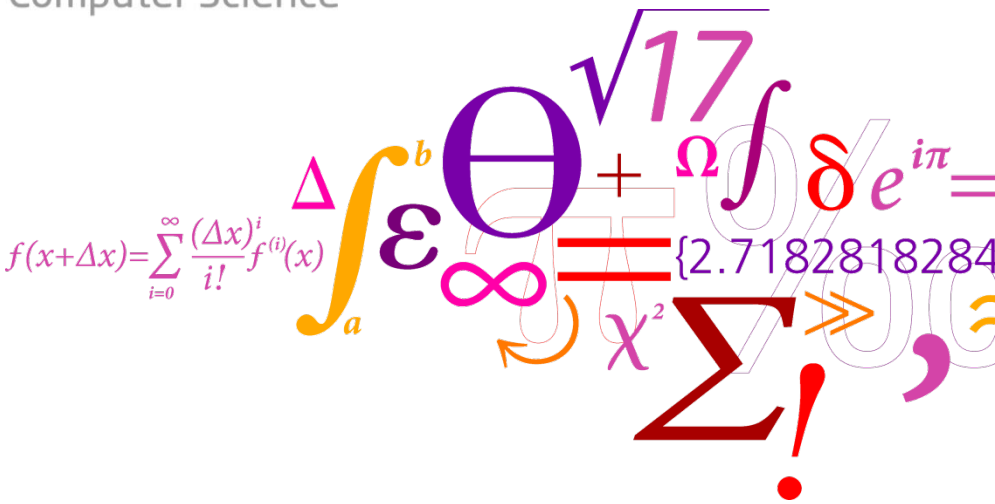
# Projekt i software-udvikling (02362)
## Forår 2022

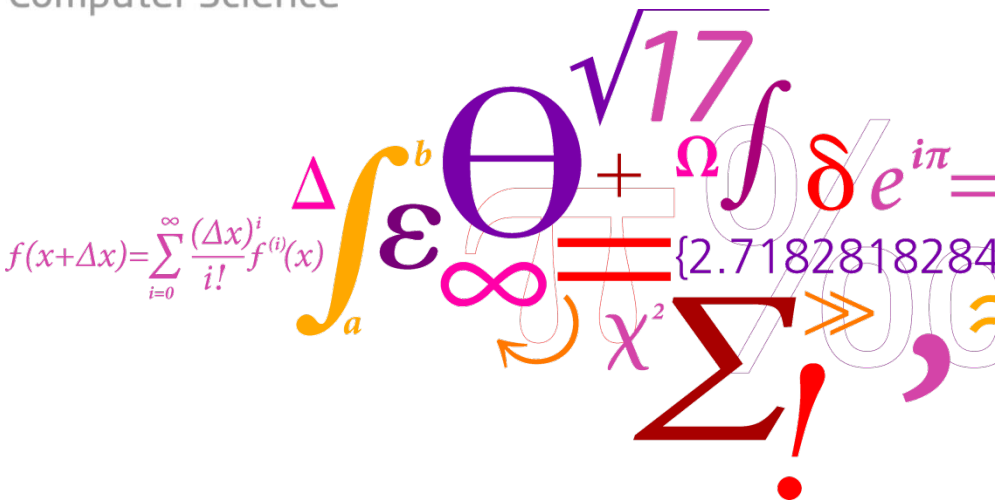
Ekkart Kindler

DTU Compute
Department of Applied Mathematics and Computer Science



# IX. Algoritmer

**DTU Compute**
Department of Applied Mathematics and Computer Science



DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

## 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

# Algoritmer

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- 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: sorterings- og søgealgoritmer

# 1. Sortering af lister

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- 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

# Bubble sort

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Lidt mere systematisk:

Vi antager at listen ligger i et array

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

og

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

er den aktuelle længde af listen

# Bubble sort (1. version)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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);
```

Inden vi diskuterer koden, skal vi diskutere ideen bagved på tavlen!!!

Det her kan (og skal) gøres lidt bedre!

# Bubble Sort

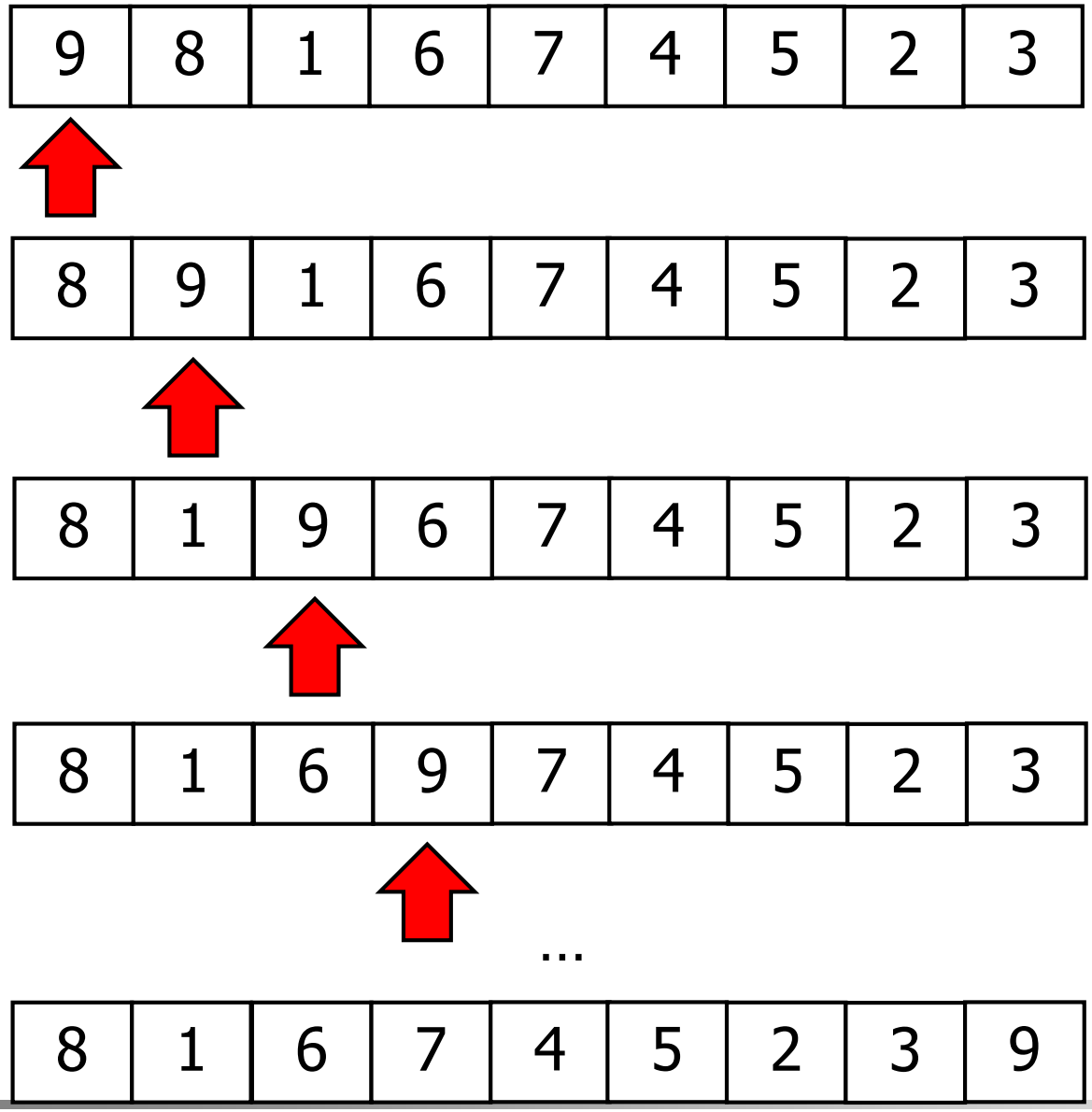
DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Observationer:

- i første omgang ”vandrer” det størrste tal (i hele array) 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

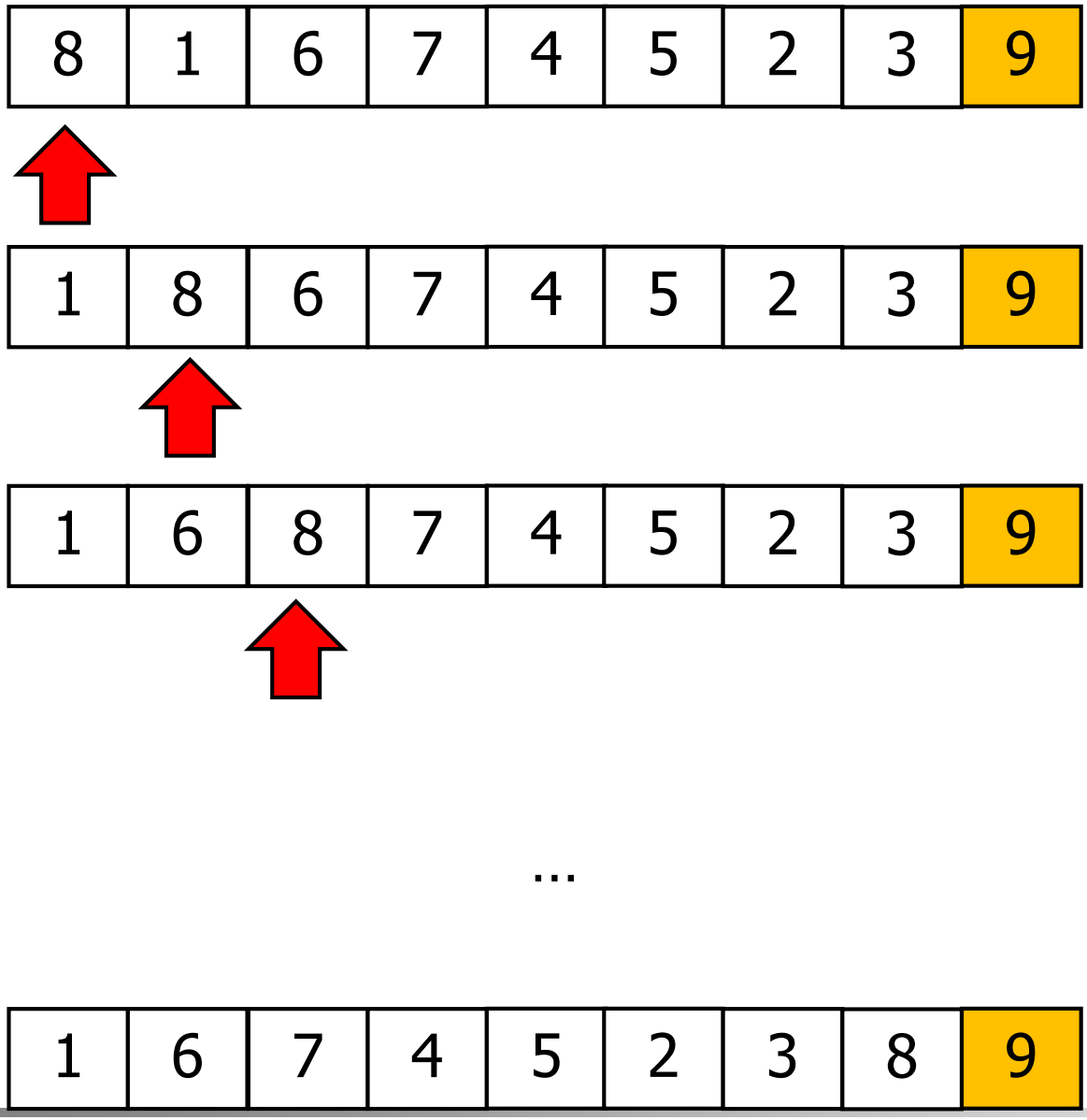
# Første iteration

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler



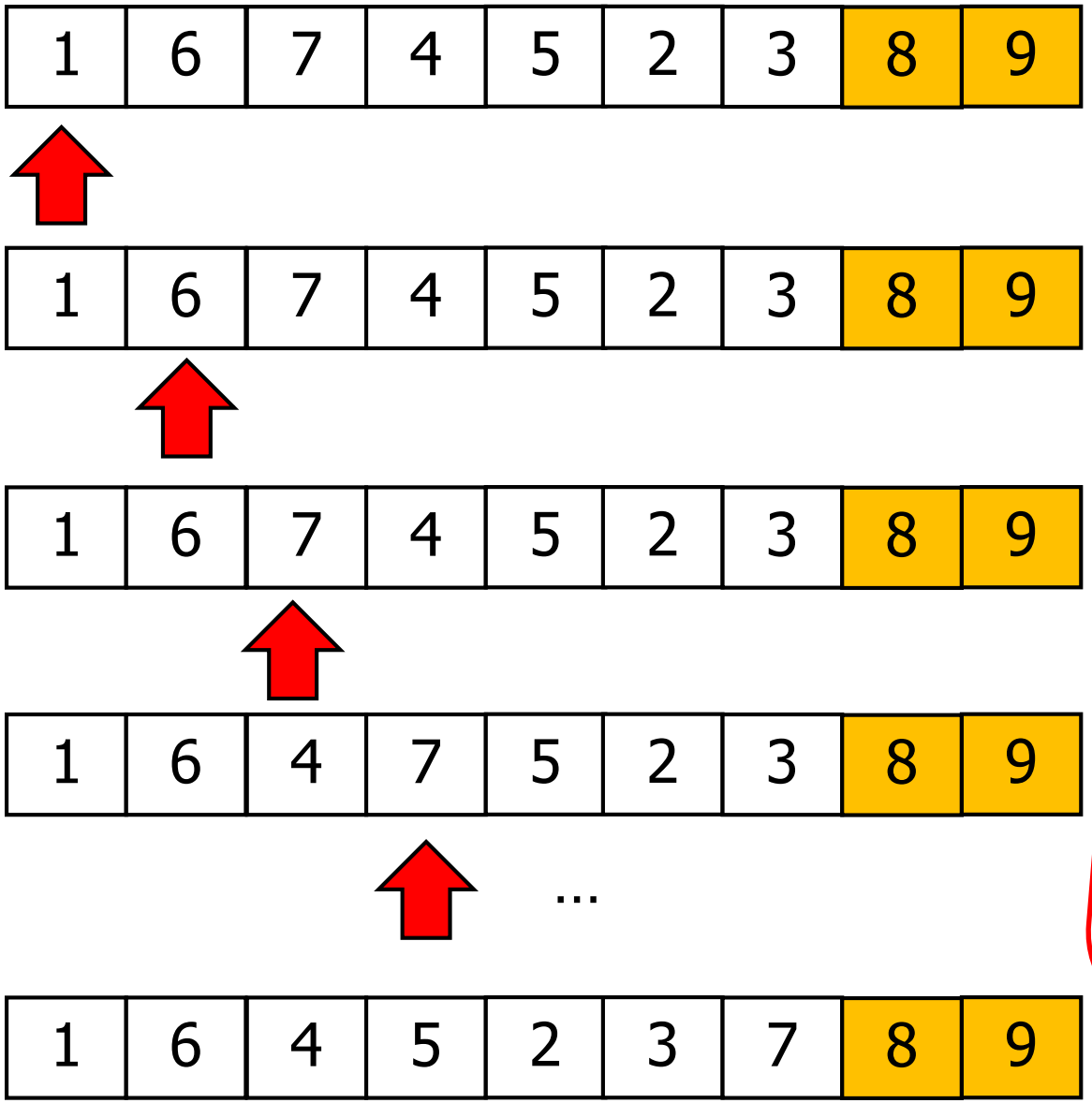
# Anden iteration

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler



# Tredje iteration

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler




I hver iteration kan vi stoppe en position tidligere! Det kan man realisere med to for-løkker i sted for en do-while- og en for-løkke eller med en tæller `j` som husker hvor langt man er nødt til at sammenligne!

# Bubble sort (2. version)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

# Bubble Sort

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- 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


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

# 2. Søgning

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- Hvordan kan man finde positionen af et element i listen (→ i lister svarer det til metoden `indexOf()`)?
- Hvor effektiv (engl. efficient) er det?

Diskussion på tavlen!
→ lineær søgning

- Kan man gøre det hurtigere, hvis listen er sorteret?

Diskussion på tavlen!

# Binær søgning

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

# Binær søgning

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Input

| | |
|---|---|
| `int[] values = …;` | Array med alle værdier som |
| `int lower = 0;` | Laveste index (typisk 0) |
| `int upper = …;` | Højeste index |

# Binær søgning (næsten!)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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!

# Binær søgning

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

# Opgave A4 (til 19. 4.)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

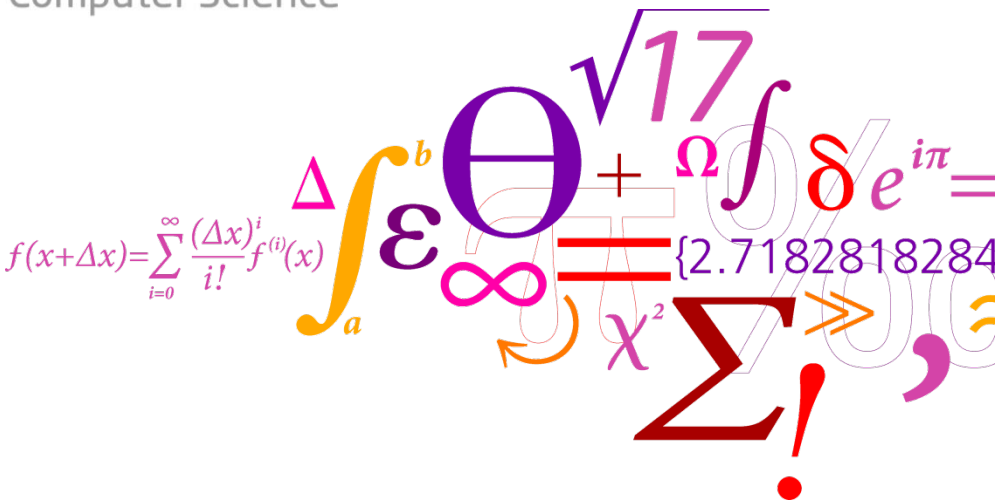
Anden **prototype** med **fokus** på:

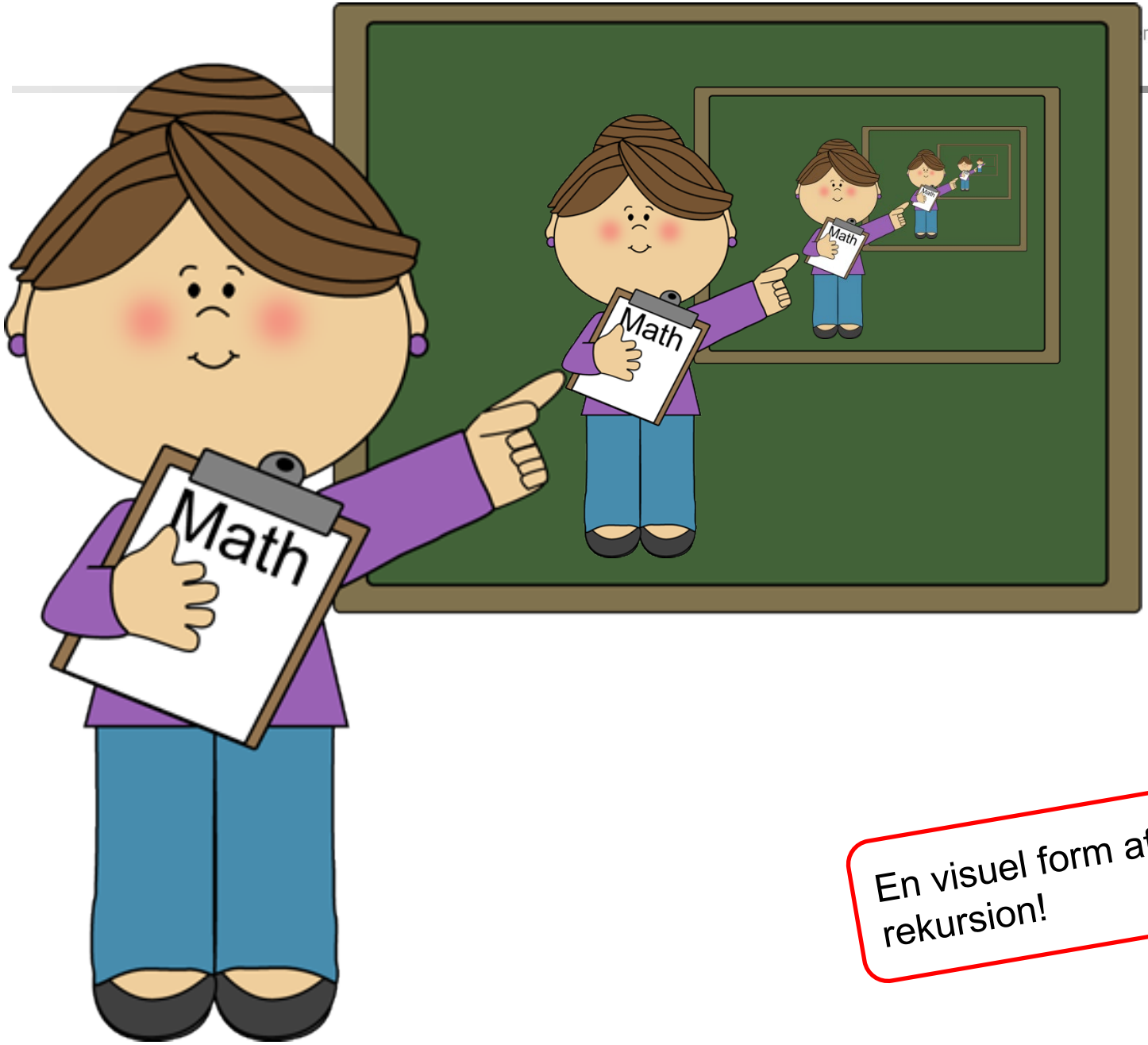
- Finpudsning af database-tilknytning (V4a) og loade spilleplader fra filer (V4b) med vægge og aktioner

- 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)

# 3. Rekursion

**DTU Compute**
Department of Applied Mathematics and Computer Science






En visuel form af en rekursion!

# Rekursion

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- ... opstår, når noget bliver definieret 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.

# Rekursion

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- ... 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!

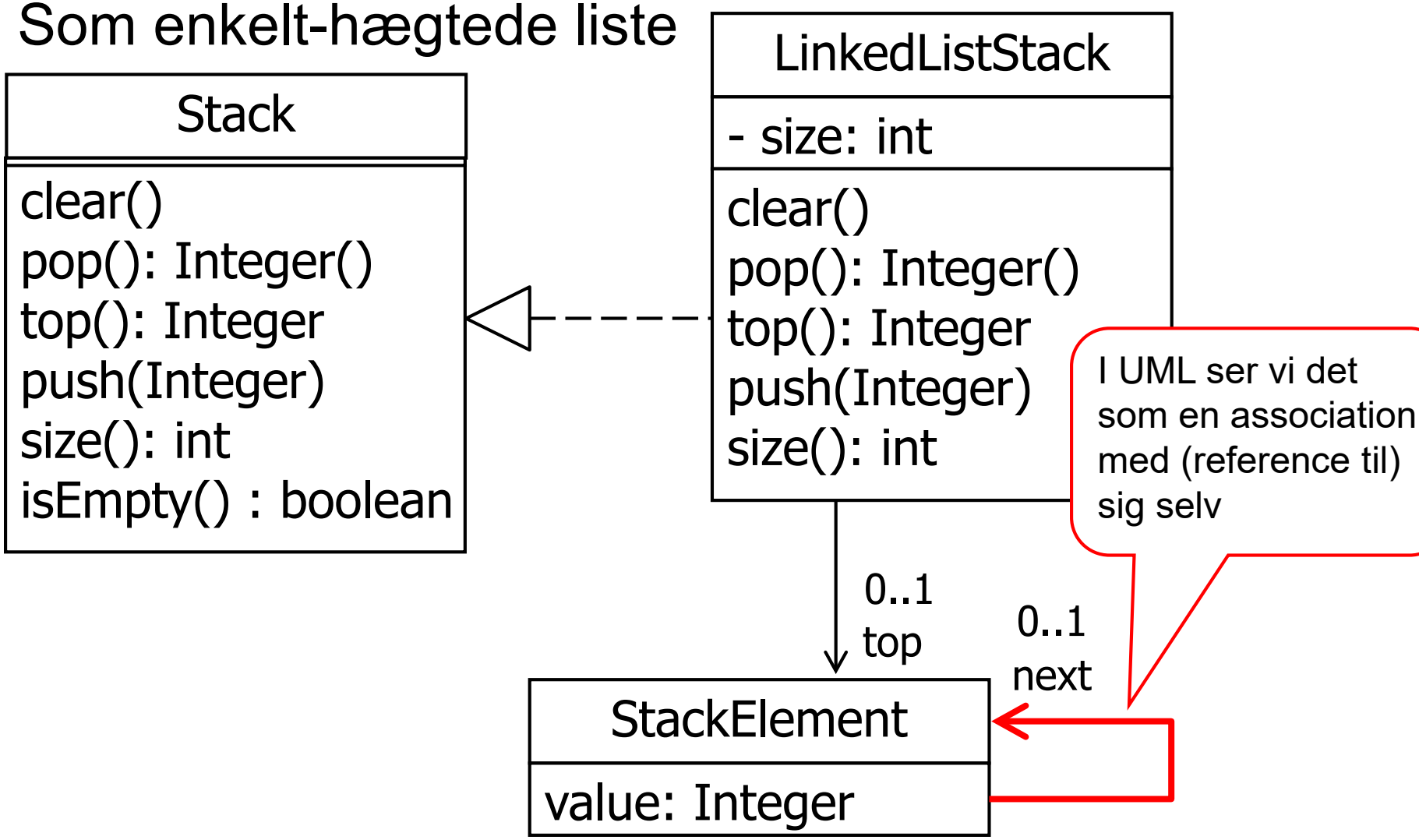
```
private class StackElement {

  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`**

# Implementeringer

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- Som enkelt-hægtede liste



# Rekursion

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- ... 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 25 og 26)

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)

# Eksempel

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

→ Diskussion på tavlen

# Størrelsen af stak

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

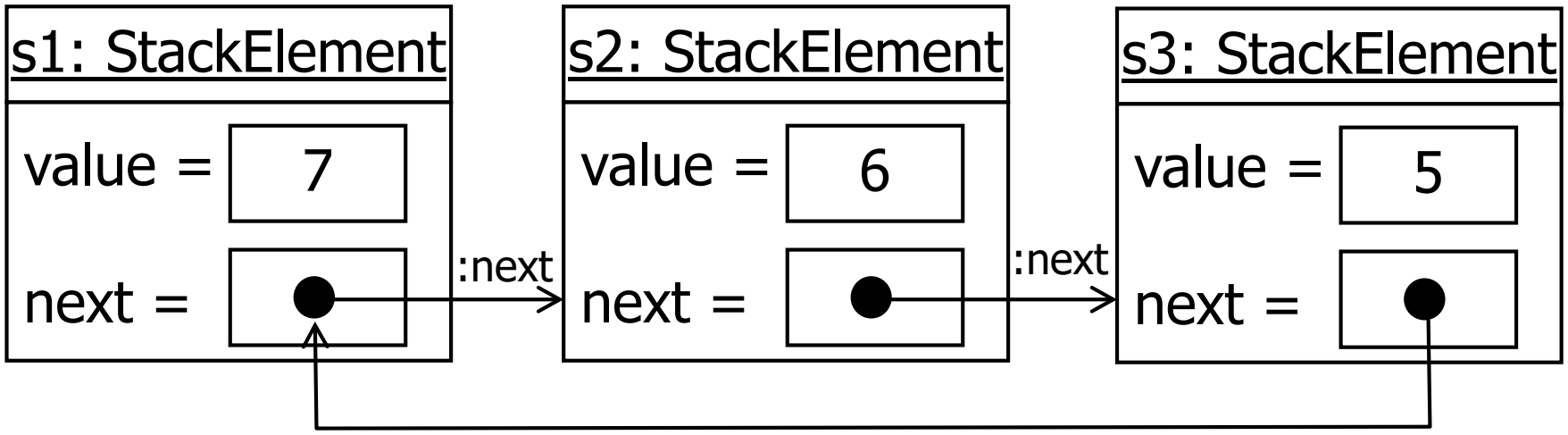
Må man det?

# Instance ("Objectdiagram")

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

med "tre elementer"

Hvad ville ske, når vi kalder **`size(s1)`** på dettet struktur?



# Call stak af `size(s1)`

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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`!

# Rekursiv stak størrelse

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- 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`**!

# Fakultet-funktion

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Definition: *factorial(n)* =

*n! = n * (n-1) * ... * 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

# Rekursiv definition *n!*

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

Her er der **rekursion** igen

# Fakultet-funktion i Java

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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?

# Fakultet-funktion i Java

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

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

# Fakultet-funktion i Java

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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. Men til factorial vil jeg heller ikke anbefale at bruge float eller double!

# Fakultet-funktion terminerer

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Fakultets-funktion terminerer for alle ikke negative heltal

… og pga defensiv programmering også for negative tal!

**Bevis**: Matematisk induktion:

Induktionsbasis:

- **`factorial(0)`** terminerer:
  body med n=0 returnerer 1 og terminerer
- Induktionsantagelse:
  **`factorial(n)`** terminerer
- Induktionsskridt: n → n+1
  **`factorial(n)`** terminerer pga. antagelsen; body returnerer **`(n+1) * factorial(n)`** og terminerer

# Observationer

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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!

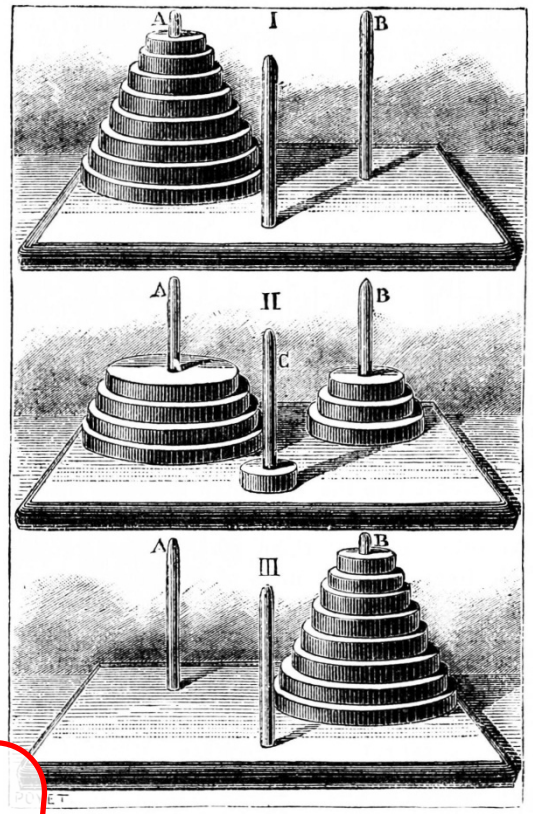
# Andet Eksempel



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



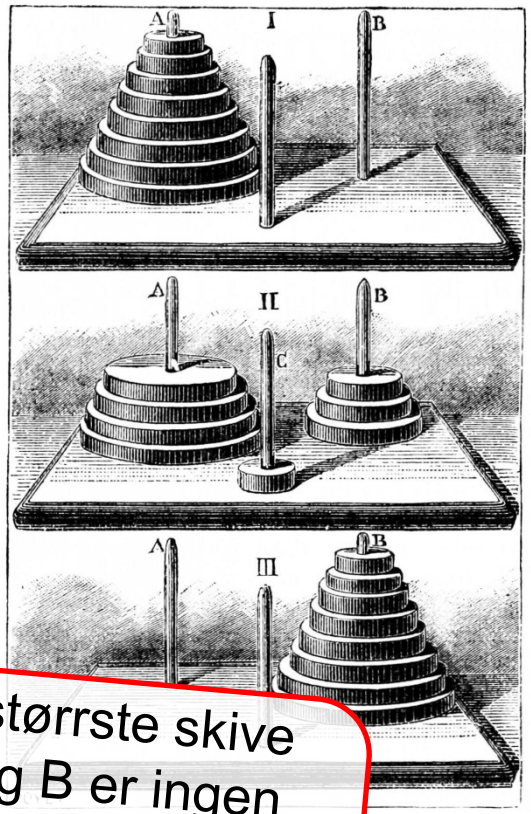

Man må gerne ”parkere” skiveren på den tredje pin *C*; men regel med størrelsen gælder også der

# Løsningsidé:

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Flyt *n* skiver fra *A* til *B*

1. Flyt de øverste *n-1* skiver fra *A* til *C* (ifølge reglerne)
2. Så flyt den nedereste (største) 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

# Rekursiv løsning:

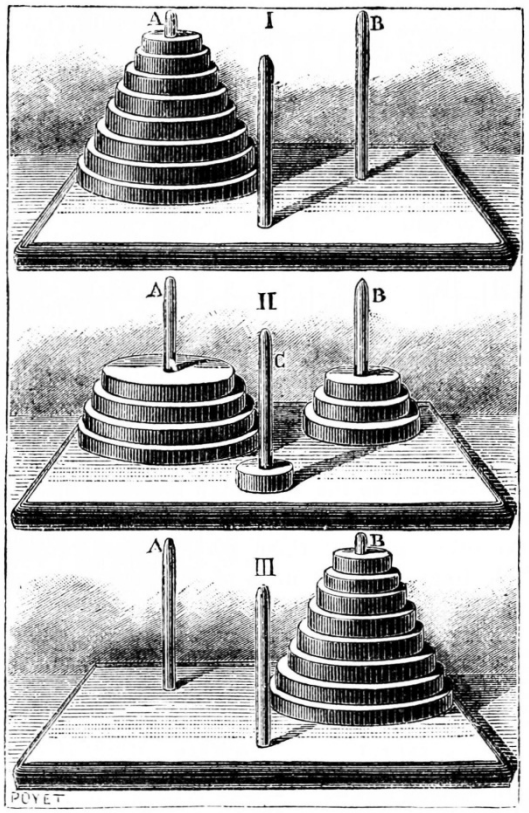
DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Flyt ***n*** skiver fra *A* til *B*

1. Flyt de øverste ***n-1*** skiver fra *A* til *C*
   (iføge reglerne)
2. Så flyt den nedereste (størrste)
   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!

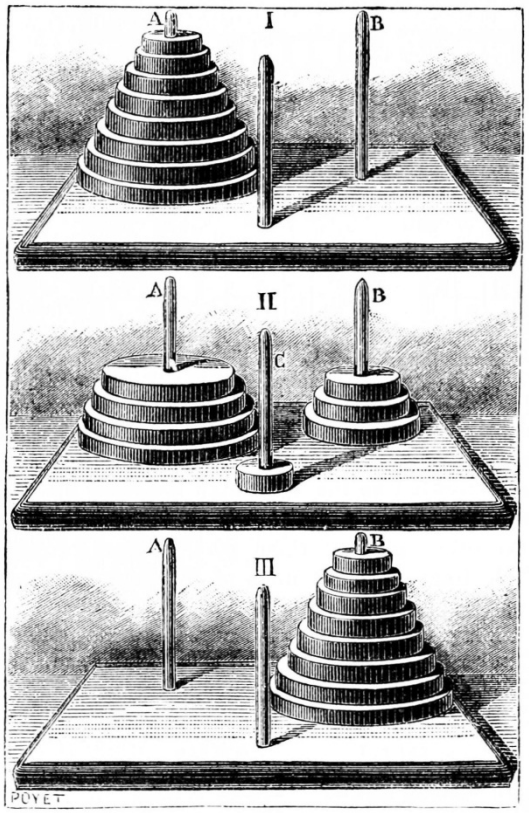


# Rekursiv løsning:

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

Flyt ***n*** skiver fra *A* til *B*

1. Flyt de øverste ***n-1*** skiver fra *A* til *C*
   (iføge reglerne)
2. Så flyt den nedereste (størrste)
   skive *n* fra *A* til *B*
   (som er OK ifølge reglerne)
3. Nu flyt de ***n-1*** skiver fra *C* til *B*



# Løsning i Java

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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);
  }
}
```

# Løsning i Java

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

```
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

# Diskussion

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

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

Bevis gennem matematisk induktion

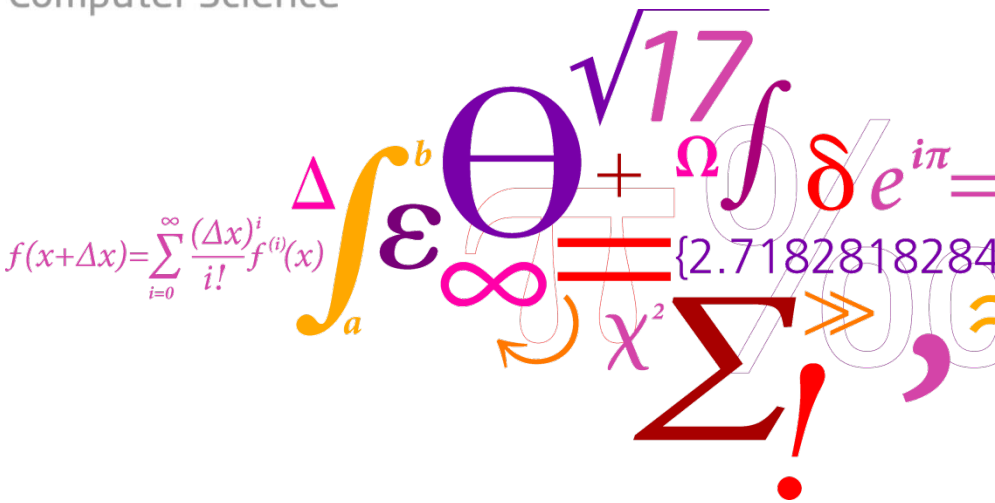
- Rekursion ser harmløs ud:
  men det kan blive til rimelig mange opkald

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:

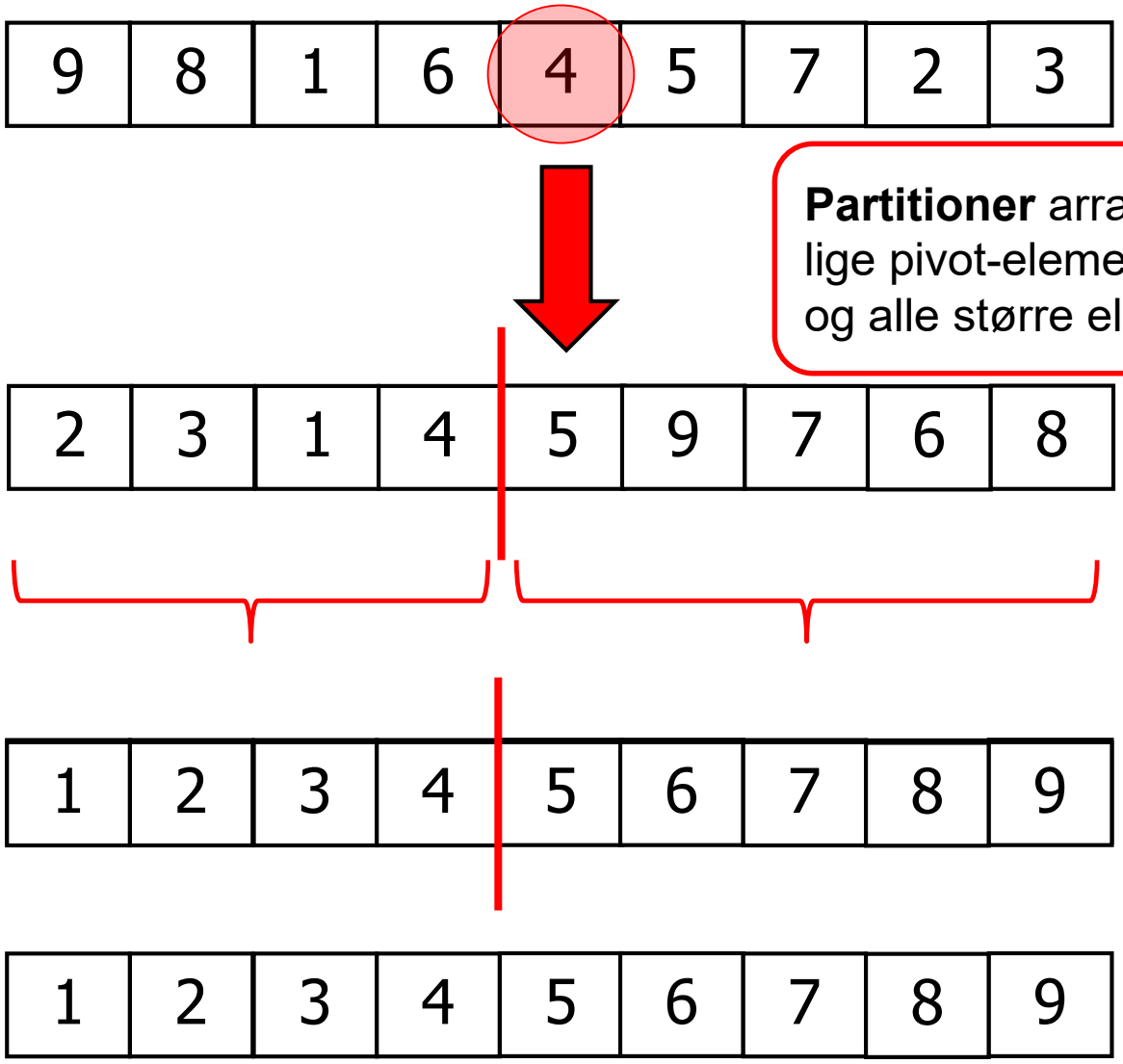
# 4. Quicksort
# (Lister, Sortering, og Søgning)

**DTU Compute**
Department of Applied Mathematics and Computer Science



# Quicksort (→ Hoare 1959)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler




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

**Sorter** partitionerne uafhængigt af hinanden **rekursiv**

Så er hele arrayet sorteret

# Quicksort

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

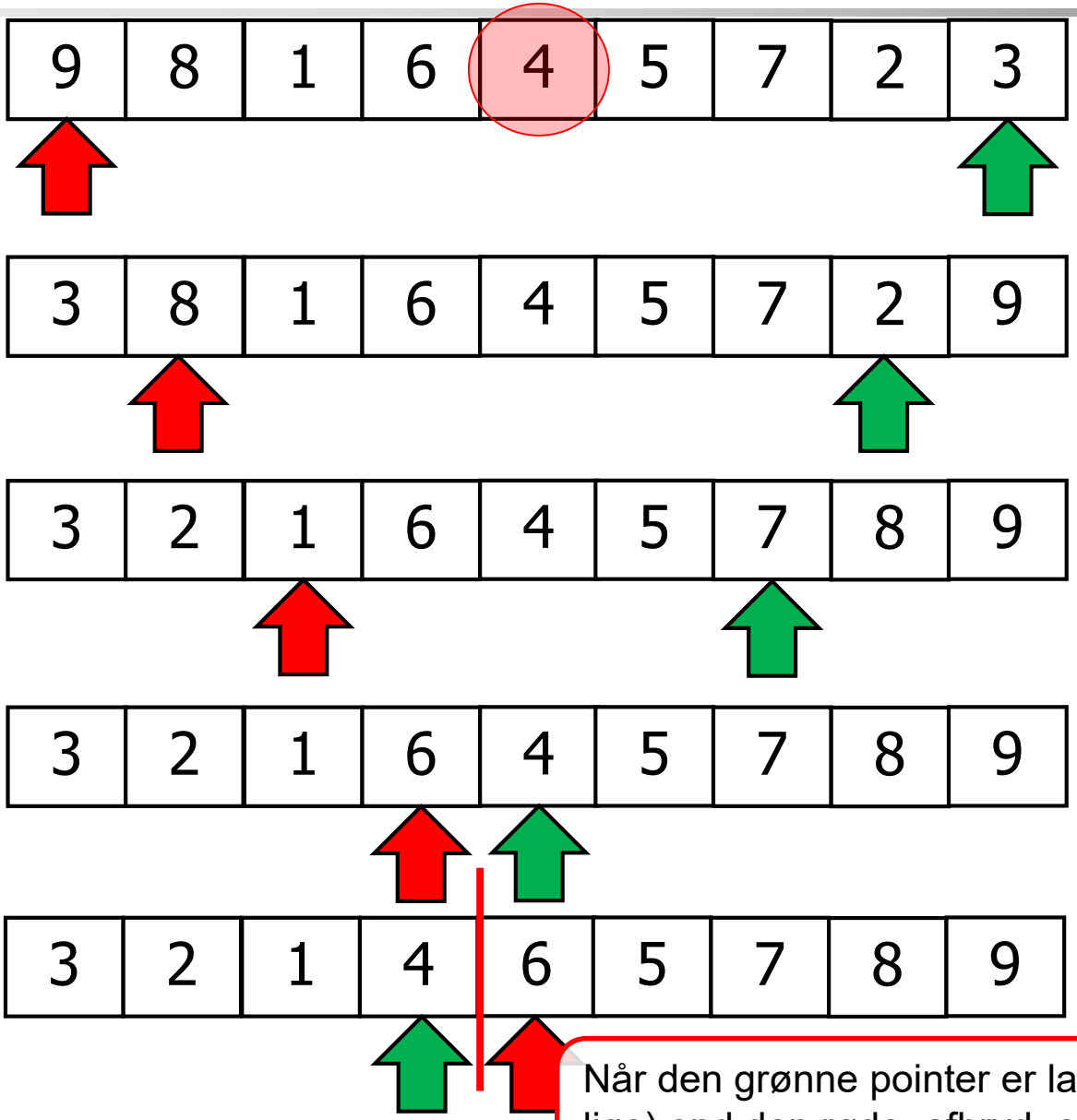
```
private void quicksort(int lower, int upper) {
  if (lower >= upper) {
    return;
  }

  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 >= (left) or <= (right) pivot

Byt elementerne på pointere og flyt dem en position videre ind

⋮

Flyt pointere indtil de rammer et element >= (left) or <= (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

# Diskussion (Quicksort)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

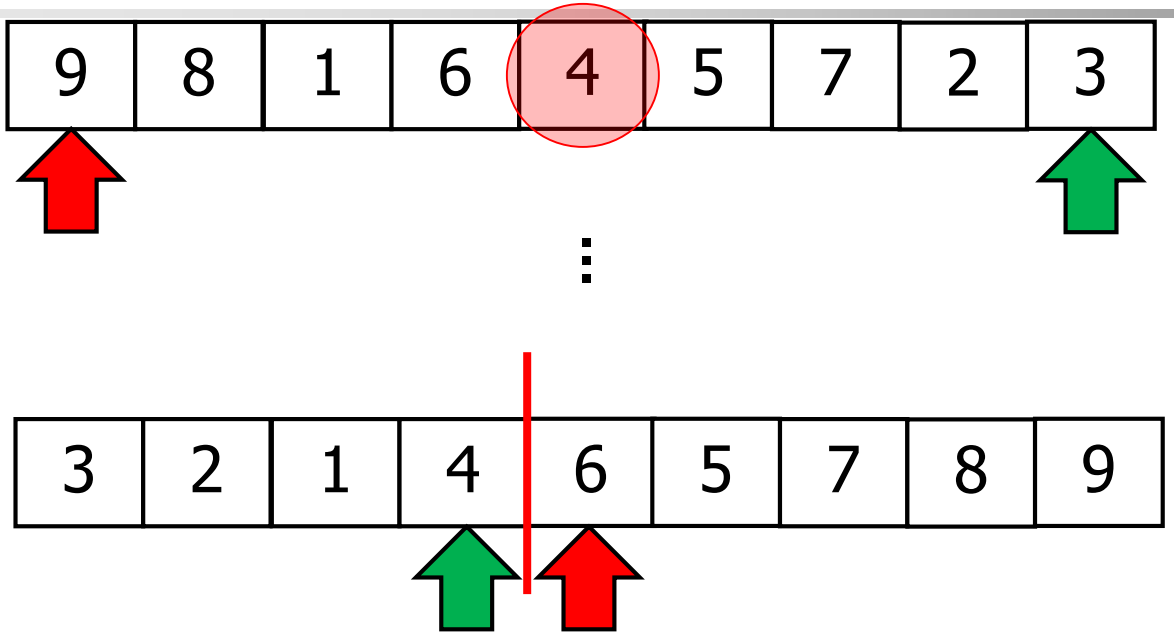
- Mange versioner af Quicksort vælger det første eller det sidste element som pivot element; men det er meget ueffektiv, hvis arrayet er sorteret allerede
- Man skal helst sørge for at begge partitioner bliver næsten samme størrelse (vælg enten midten som pivot eller helt tilfædigt)

→ 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)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler




- Det er vigtigt at *lower* ≤ *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)!

# Diskussion (andre)

DTU Compute
Department of Applied Mathematics and Computer Science
Ekkart Kindler

- 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.