

# Insertion sort & Asymptotisk notation

## Algorithms and Datastructures, F25, Lecture 2

Andreas Holck Høeg-Petersen

Department of Computer Science  
Aalborg University

January 30, 2025

# Opdateringer

- Løsninger på exercises kommer på et eller andet tidspunkt
- Fra evaluering:
  - ▶ Grupper?
  - ▶ Andet?

# Outline

- 1 Insertion Sort
- 2 Loop invarianter og korrekthed
- 3 Exercises
- 4 Asymptotisk notation og analyse

# Outline

- 1 Insertion Sort
- 2 Loop invarianter og korrekthed
- 3 Exercises
- 4 Asymptotisk notation og analyse

# Sorteringsproblemet

En klassiker

**Input** En sekvens  $A$  af  $n$  tal  $(a_1, a_2, \dots, a_n)$

**Output** En permutation  $(a'_1, a'_2, \dots, a'_n)$  af  $A$  således at  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

# Sorteringsproblemet

En klassiker

**Input** En sekvens  $A$  af  $n$  tal  $(a_1, a_2, \dots, a_n)$

**Output** En permutation  $(a'_1, a'_2, \dots, a'_n)$  af  $A$  således at  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- Tallene vi sorterer kalder vi også **nøgler** (keys)

# Sorteringsproblemet

En klassiker

**Input** En sekvens  $A$  af  $n$  tal  $(a_1, a_2, \dots, a_n)$

**Output** En permutation  $(a'_1, a'_2, \dots, a'_n)$  af  $A$  således at  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- Tallene vi sorterer kalder vi også **nøgler** (keys)
  - ▶ Nøglerne er nogle gange forskellig fra den data, vi egentlig sorterer

# Sorteringsproblemet

En klassiker

**Input** En sekvens  $A$  af  $n$  tal  $(a_1, a_2, \dots, a_n)$

**Output** En permutation  $(a'_1, a'_2, \dots, a'_n)$  af  $A$  således at  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- Tallene vi sorterer kalder vi også **nøgler** (keys)
  - ▶ Nøglerne er nogle gange forskellig fra den data, vi egentlig sorterer
  - ▶ F.eks. kunne vi sortere brugere (**sattelit data**) på baggrund af deres alder (**nøgler**)



# Sorteringsproblemet

En klassiker

**Input** En sekvens  $A$  af  $n$  tal  $(a_1, a_2, \dots, a_n)$

**Output** En permutation  $(a'_1, a'_2, \dots, a'_n)$  af  $A$  således at  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- Tallene vi sorterer kalder vi også **nøgler** (keys)
  - ▶ Nøglerne er nogle gange forskellig fra den data, vi egentlig sorterer
  - ▶ F.eks. kunne vi sortere brugere (**sattelit data**) på baggrund af deres alder (**nøgler**)
- Sortering er tit et underproblem for mange andre problemer, der kan gøres nemmere ved først at sortere inputtet

# Sorteringsproblemet

En klassiker

**Input** En sekvens  $A$  af  $n$  tal  $(a_1, a_2, \dots, a_n)$

**Output** En permutation  $(a'_1, a'_2, \dots, a'_n)$  af  $A$  således at  $a'_1 \leq a'_2 \leq \dots \leq a'_n$

- Tallene vi sorterer kalder vi også **nøgler** (keys)
  - ▶ Nøglerne er nogle gange forskellig fra den data, vi egentlig sorterer
  - ▶ F.eks. kunne vi sortere brugere (**sattelit data**) på baggrund af deres alder (**nøgler**)
- Sortering er tit et underproblem for mange andre problemer, der kan gøres nemmere ved først at sortere inputtet
- Der findes mange sorteringsalgoritmer: merge sort, quicksort, bubble sort, heapsort, cocktail shaker sort, etc. . .

# Sorteringsproblemet

## En klassiker

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging [5]
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	Used in several STL algorithms.
Insertion	$n \log n$	$n \log n$	$n^2 \log n$	1	Yes	Partitioning	Highly customizable but no filling in using the "three finger sort" algorithm [6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	$n$	Yes	Merging	Variant of Heapsort.
Quicksort	$n \log n$	$n \log n$	$n^2$	1	No	Selection	When using a sort, it uses a sort.
Tree sort	$n \log n$	$n \log n$	$n \log n$	1	Yes	Insertion	Combines a block-based O(n) insertion merge algorithm [7] with a selection sort.
Block sort	$n$	$n \log n$	$n \log n$	1	Yes	Merging	An adaptive variant of Insertion sort, used in the standard library.
Smoothsort	$n$	$n \log n$	$n \log n$	1	No	Selection	Makes $n-1$ comparisons when the data is already sorted or reverse sorted.
Timsort	$n$	$n \log n$	$n \log n$	$n$	Yes	Insertion & Merging	Finds all the longest increasing subsequences in the log $n$ .
Radix sort	$n$	$n \log n$	$n \log n$	$n$	Yes	Selection	Makes $n-1$ comparisons when the data is already sorted or reverse sorted.
Countsort	$n$	$n \log n$	$n \log n$	$n$	Yes	Insertion	Quicksort is usually done in place with O(log $n$ ) stack space [8].
Quicksort	$n \log n$	$n \log n$	$n^2$	1	No	Partitioning	Stable for a general insertion sort. It requires randomly permuting the input to maintain worst-case time complexity.
Library sort	$n \log n$	$n \log n$	$n^2$	$n$	No	Insertion	Stable for a general insertion sort. It requires randomly permuting the input to maintain worst-case time complexity.
Shell sort	$n \log n$	$n^2$	$n^2$	1	No	Insertion	Small code size.
Comb sort	$n \log n$	$n^2$	$n^2$	1	No	Exchanging	Faster than bubble sort on average.
Insertion sort	$n$	$n^2$	$n^2$	1	Yes	Insertion	O(n $\log n$ ) in the worst case, but more comparisons than heap sort or mergesort.
Bubble sort	$n$	$n^2$	$n^2$	1	Yes	Exchanging	Very code size.
Cocktail shaker sort	$n$	$n^2$	$n^2$	1	Yes	Exchanging	A variant of Bubble sort which sorts with small values at end of list.
Grainsort	$n$	$n^2$	$n^2$	1	Yes	Exchanging	Very code size.
Odd-even sort	$n$	$n^2$	$n^2$	1	Yes	Exchanging	Can be run in parallel processors easily.
Simple parallel sort	$n$	$n^2$	$n^2$	1	No	Selection	A variant of selection sort that sorts recursively, instead of just swapping the two items after each selection scan.
Steady sort	$n$	$n^2$	$n^2$	$n$	Yes	Selection	Stable with O(n) extra space, when using linked lists, or when made as a variant of Insertion Sort.
Selection sort	$n^2$	$n^2$	$n^2$	1	No	Selection	Instead of swapping the two items, it swaps the two items [9].
Exchange sort	$n^2$	$n^2$	$n^2$	1	No	Exchanging	Very code size.
Cycle sort	$n^2$	$n^2$	$n^2$	1	No	Selection	Insertion with exponentially optimal number of writes.

Figure: Screenshot fra Wikipedia

# Insertion sort

Vores første sorteringsalgoritme!

- Vi starter med at kigge på **Insertion sort**
- Simple sorteringsalgoritme, effektiv for små værdier af  $n$ 
  - ▶ Hvad mener jeg med  $n$ ?
  - ▶ Også effektiv for **næsten sorterede** sekvenser!
- Kan ligne sortering af kort:
  - ▶ Start med en tom hånd, kortbunken ligger på bordet
  - ▶ Tag et kort af gangen fra bunken
  - ▶ Søg i hånden til vi finder den korrekte position (fra højre til venstre)
  - ▶ Indsæt kortet på denne position

# Insertion-Sort

## Pseudo-kode

### Insertion-Sort( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3      // Insert  $A[i]$  into the sorted sequence  $A[1 : i - 1]$ 
4       $j = i - 1$ 
5      while  $j > 0$  and  $A[j] > key$ 
6           $A[j + 1] = A[j]$ 
7           $j = j - 1$ 
8       $A[j + 1] = key$ 
```

# Insertion sort

## Pseudo-kode

- Algoritmen tager et array  $A[1 : n]$  som input
- Vedligeholder to sub-arrays
  - ▶  $A[1 : i - 1]$  er 'kortene på hånden' (altid sorteret)
  - ▶  $A[i : n]$  er 'kortene på bordet'

### INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Insertion sort

## Pseudo-kode

- Algoritmen gør 3 ting i hver iteration:

- ▶ Find det element *key*, der skal placeres korrekt (i 'hånden')
- ▶ Gør plads i det sorterede sub-array ('hånden') ved at flytte større elementer en plads bagud
- ▶ Indsæt *key* på sin plads

### INSERTION-SORT(*A*)

```
1  for i = 2 to n
2      key = A[i]
3      j = i - 1
4      while j > 0 and A[j] > key
5          A[j + 1] = A[j]
6          j = j - 1
7      A[j + 1] = key
```

# Insertion sort

## Pseudo-kode

- Algoritmen gør 3 ting i hver iteration:

- ▶ Find det element *key*, der skal placeres korrekt (i 'hånden')
- ▶ Gør plads i det sorterede sub-array ('hånden') ved at flytte større elementer en plads bagud
- ▶ Indsæt *key* på sin plads

### INSERTION-SORT(*A*)

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```



# Insertion sort

## Pseudo-kode

- Algoritmen gør 3 ting i hver iteration:

- ▶ Find det element  $key$ , der skal placeres korrekt (i 'hånden')
- ▶ Gør plads i det sorterede sub-array ('hånden') ved at flytte større elementer en plads bagud
- ▶ Indsæt  $key$  på sin plads

### INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Insertion sort

## Pseudo-kode

- Algoritmen gør 3 ting i hver iteration:

- ▶ Find det element *key*, der skal placeres korrekt (i 'hånden')
- ▶ Gør plads i det sorterede sub-array ('hånden') ved at flytte større elementer en plads bagud
- ▶ **Indsæt *key* på sin plads**

### INSERTION-SORT(*A*)

```
1  for i = 2 to n
2      key = A[i]
3      j = i - 1
4      while j > 0 and A[j] > key
5          A[j + 1] = A[j]
6          j = j - 1
7      A[j + 1] = key
```

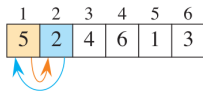
# Insertion sort

## Eksempel



# Insertion sort

## Eksempel



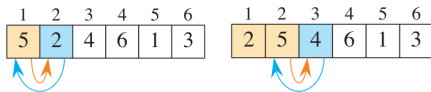
**Figure:** Insertion sort example on input [5, 2, 4, 6, 1, 3]

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Insertion sort

## Eksempel



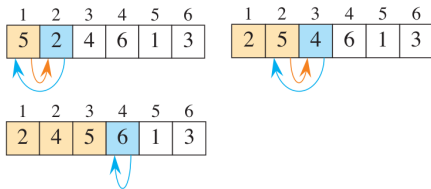
**Figure:** Insertion sort example on input [5, 2, 4, 6, 1, 3]

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Insertion sort

## Eksempel



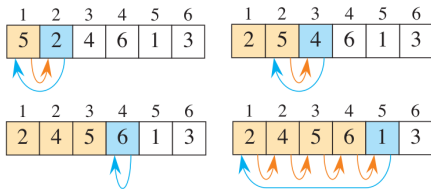
**Figure:** Insertion sort example on input [5, 2, 4, 6, 1, 3]

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Insertion sort

## Eksempel



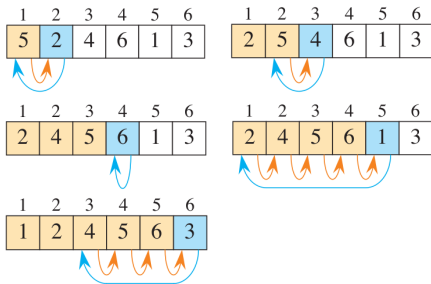
**Figure:** Insertion sort example on input [5, 2, 4, 6, 1, 3]

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Insertion sort

## Eksempel



**Figure:** Insertion sort example on input [5, 2, 4, 6, 1, 3]

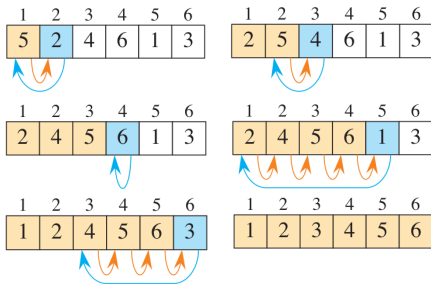
INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```



# Insertion sort

## Eksempel



**Figure:** Insertion sort example on input [5, 2, 4, 6, 1, 3]

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Insertion sort

GIF!



# Outline

- 1 Insertion Sort
- 2 Loop invarianter og korrekthed
- 3 Exercises
- 4 Asymptotisk notation og analyse



# Loop invarianter

## Introduktion

Når vi ser på algoritmer skal vi gerne kunne argumentere for, at algoritmen faktisk virker — og endnu bedre, vi skal gerne kunne **bevise** det! En type korrekthedsbevis er ved hjælp af **loop invarianter**.

- En **invariant** er en egenskab, der *ikke* varierer; altså altid er sand

# Loop invarianter

## Introduktion

Når vi ser på algoritmer skal vi gerne kunne argumentere for, at algoritmen faktisk virker — og endnu bedre, vi skal gerne kunne **bevise** det! En type korrekthedsbevis er ved hjælp af **loop invarianter**.

- En **invariant** er en egenskab, der *ikke* varierer; altså altid er sand
- Vi leder efter en egenskab, der relaterer sig til algoritmens opgave

# Loop invarianter

## Introduktion

Når vi ser på algoritmer skal vi gerne kunne argumentere for, at algoritmen faktisk virker — og endnu bedre, vi skal gerne kunne **bevise** det! En type korrekthedsbevis er ved hjælp af **loop invarianter**.

- En **invariant** er en egenskab, der *ikke* varierer; altså altid er sand
- Vi leder efter en egenskab, der relaterer sig til algoritmens opgave
- Vi vil så gerne vise, at

# Loop invarianter

## Introduktion

Når vi ser på algoritmer skal vi gerne kunne argumentere for, at algoritmen faktisk virker — og endnu bedre, vi skal gerne kunne **bevise** det! En type korrekthedsbevis er ved hjælp af **loop invarianter**.

- En **invariant** er en egenskab, der *ikke* varierer; altså altid er sand
- Vi leder efter en egenskab, der relaterer sig til algoritmens opgave
- Vi vil så gerne vise, at
  - ▶ Invarianten er sand når vi starter den første iteration (**initialization**)

# Loop invarianter

## Introduktion

Når vi ser på algoritmer skal vi gerne kunne argumentere for, at algoritmen faktisk virker — og endnu bedre, vi skal gerne kunne **bevise** det! En type korrekthedsbevis er ved hjælp af **loop invarianter**.

- En **invariant** er en egenskab, der *ikke* varierer; altså altid er sand
- Vi leder efter en egenskab, der relaterer sig til algoritmens opgave
- Vi vil så gerne vise, at
  - ▶ Invarianten er sand når vi starter den første iteration (**initialization**)
  - ▶ Hvis den er sand, når en iteration starter, er den også sand, når iterationen slutter (**maintenance**)



# Loop invarianter

## Introduktion

Når vi ser på algoritmer skal vi gerne kunne argumentere for, at algoritmen faktisk virker — og endnu bedre, vi skal gerne kunne **bevise** det! En type korrekthedsbevis er ved hjælp af **loop invarianter**.

- En **invariant** er en egenskab, der *ikke* varierer; altså altid er sand
- Vi leder efter en egenskab, der relaterer sig til algoritmens opgave
- Vi vil så gerne vise, at
  - ▶ Invarianten er sand når vi starter den første iteration (**initialization**)
  - ▶ Hvis den er sand, når en iteration starter, er den også sand, når iterationen slutter (**maintenance**)
  - ▶ Loopet terminerer på et tidspunkt, og invariantens egenskab kan nu bruges til at vise algoritmens korrekthed

# Loop invarianter

## Insertion sort



# Loop invarianter

## Insertion sort

### Initialization

- Før den første iteration er  $i = 2$

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Initialization

- Før den første iteration er  $i = 2$
- Sub-arrayet  $A[1 : i - 1]$  består kun af et element  $A[1]$

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Initialization

- Før den første iteration er  $i = 2$
- Sub-arrayet  $A[1 : i - 1]$  består kun af et element  $A[1]$
- Dette element er det samme, som oprindeligt var i  $A[1]$  (for vi har ikke ændret noget)

### INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Initialization

- Før den første iteration er  $i = 2$
- Sub-arrayet  $A[1 : i - 1]$  består kun af et element  $A[1]$
- Dette element er det samme, som oprindeligt var i  $A[1]$  (for vi har ikke ændret noget)
- Et array med kun 1 element er per definition sorteret

### INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Maintenance

- While-løkken flytter **et** element fra  $A[i]$  til dets korrekte plads i  $A[1 : i]$

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Maintenance

- While-løkken flytter **et** element fra  $A[i]$  til dets korrekte plads i  $A[1 : i]$
- Sub-arrayet indeholder nu stadig de oprindelige elementer fra  $A[1 : i]$ , stadig i sorteret orden

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.



# Loop invarianter

## Insertion sort

### Maintenance

- While-løkken flytter **et** element fra  $A[i]$  til dets korrekte plads i  $A[1 : i]$
- Sub-arrayet indeholder nu stadig de oprindelige elementer fra  $A[1 : i]$ , stadig i sorteret orden
- Når vi inkrementerer  $i$  opretholdes loop-invarianten

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Maintenance

- While-løkken flytter **et** element fra  $A[i]$  til dets korrekte plads i  $A[1 : i]$
- Sub-arrayet indeholder nu stadig de oprindelige elementer fra  $A[1 : i]$ , stadig i sorteret orden
- Når vi inkrementerer  $i$  opretholdes loop-invarianten
- NB: I teorien burde vi have lavet samme øvelse for while-løkken selv, men...

### INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Termination

- For-løkken stopper når  $i$  er større end  $n$ . Da  $i$  starter ved 2 og inkrementeres i hver iteration, vil løkken terminere når  $i = n + 1$

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Termination

- For-løkken stopper når  $i$  er større end  $n$ . Da  $i$  starter ved 2 og inkrementeres i hver iteration, vil løkken terminere når  $i = n + 1$
- Indsætter vi  $n + 1$  i loop-invarianten får vi  $A[1 : (n + 1) - 1] = A[1 : n]$

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Termination

- For-løkken stopper når  $i$  er større end  $n$ . Da  $i$  starter ved 2 og inkrementeres i hver iteration, vil løkken terminere når  $i = n + 1$
- Indsætter vi  $n + 1$  i loop-invarianten får vi  $A[1 : (n + 1) - 1] = A[1 : n]$
- Altså får vi, at  $A[1 : n]$  indeholder alle de oprindelige elementer, men nu i sorteret rækkefølge

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Loop invarianter

## Insertion sort

### Termination

- For-løkken stopper når  $i$  er større end  $n$ . Da  $i$  starter ved 2 og inkrementeres i hver iteration, vil løkken terminere når  $i = n + 1$
- Indsætter vi  $n + 1$  i loop-invarianten får vi  $A[1 : (n + 1) - 1] = A[1 : n]$
- Altså får vi, at  $A[1 : n]$  indeholder alle de oprindelige elementer, men nu i sorteret rækkefølge
- Eftersom  $A[1 : n]$  er hele arrayet, kan vi konkludere, at  $A$  er sorteret og algoritmen er **korrekt**

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### Loop invariant

Ved begyndelse af hver iteration af **for-løkken** består sub-arrayet  $A[1 : i - 1]$  af de oprindelige elementer i  $A[1 : i - 1]$  men i sorteret rækkefølge.

# Outline

- 1 Insertion Sort
- 2 Loop invarianter og korrekthed
- 3 Exercises
- 4 Asymptotisk notation og analyse



# Exercises!

Yay!



AALBORG  
UNIVERSITET



# Outline

- 1 Insertion Sort
- 2 Loop invarianter og korrekthed
- 3 Exercises
- 4 Asymptotisk notation og analyse



# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

tid  $\times$  antal gange

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$c_1 \times n$

# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$$c_1 \times n$$

$$c_2 \times n - 1$$

# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$$c_1 \times n$$

$$c_2 \times n - 1$$

$$c_3 \times n - 1$$

# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$$c_1 \times n$$

$$c_2 \times n - 1$$

$$c_3 \times n - 1$$

$$c_4 \times \sum_{i=2}^n t_i$$

# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$$c_1 \times n$$

$$c_2 \times n - 1$$

$$c_3 \times n - 1$$

$$c_4 \times \sum_{i=2}^n t_i$$

$$c_5 \times \sum_{i=2}^n (t_i - 1)$$

# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$$c_1 \times n$$

$$c_2 \times n - 1$$

$$c_3 \times n - 1$$

$$c_4 \times \sum_{i=2}^n t_i$$

$$c_5 \times \sum_{i=2}^n (t_i - 1)$$

$$c_6 \times \sum_{i=2}^n (t_i - 1)$$



# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$$c_1 \times n$$

$$c_2 \times n - 1$$

$$c_3 \times n - 1$$

$$c_4 \times \sum_{i=2}^n t_i$$

$$c_5 \times \sum_{i=2}^n (t_i - 1)$$

$$c_6 \times \sum_{i=2}^n (t_i - 1)$$

$$c_7 \times n - 1$$

# Tidskompleksitet

## Recap

Vi så i sidste uge, hvordan vi i lidt grove træk kan angive køretiden på en algoritme. Lad os prøve på INSERTION-SORT! Vi siger, at while-løkken kører  $t_i$  gange for en eller anden værdi af  $i$ :

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

$$c_1 \times n$$

$$c_2 \times n - 1$$

$$c_3 \times n - 1$$

$$c_4 \times \sum_{i=2}^n t_i$$

$$c_5 \times \sum_{i=2}^n (t_i - 1)$$

$$c_6 \times \sum_{i=2}^n (t_i - 1)$$

$$c_7 \times n - 1$$

Hmm... Hvad er best case? Hvad er worst case?

# Tidskompleksitet

## Recap

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

```
 $c_1 \times n$ 
 $c_2 \times n - 1$ 
 $c_3 \times n - 1$ 
 $c_4 \times \sum_{i=2}^n t_i$ 
 $c_5 \times \sum_{i=2}^n (t_i - 1)$ 
 $c_6 \times \sum_{i=2}^n (t_i - 1)$ 
 $c_7 \times n - 1$ 
```

# Tidskompleksitet

## Recap

**Best case**  $A$  er allerede sorteret, og vi kommer aldrig ind i while-løkken  $\Rightarrow t_i = 0$  for alle  $i = 2 \dots n$

### INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

### tid $\times$ antal gange

```
 $c_1 \times n$ 
 $c_2 \times n - 1$ 
 $c_3 \times n - 1$ 
 $c_4 \times n - 1$ 
 $c_5 \times 0$ 
 $c_6 \times 0$ 
 $c_7 \times n - 1$ 
```

# Tidskompleksitet

## Recap

**Best case**  $A$  er allerede sorteret, og vi kommer aldrig ind i while-løkken  $\Rightarrow t_i = 0$  for alle  $i = 2 \dots n$

**Worst case**  $A$  er omvendt sorteret, og vi skal helt i bund hver gang  $\Rightarrow t_i = i$  for alle  $i = 2 \dots n$

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

```
 $c_1 \times n$ 
 $c_2 \times n - 1$ 
 $c_3 \times n - 1$ 
 $c_4 \times \sum_{i=2}^n i$ 
 $c_5 \times \sum_{i=2}^n (i - 1)$ 
 $c_6 \times \sum_{i=2}^n (i - 1)$ 
 $c_7 \times n - 1$ 
```

# Tidskompleksitet

## Recap

**Best case**  $A$  er allerede sorteret, og vi kommer aldrig ind i while-løkken  $\Rightarrow t_i = 0$  for alle  $i = 2 \dots n$

**Worst case**  $A$  er omvendt sorteret, og vi skal helt i bund hver gang  $\Rightarrow t_i = i$  for alle  $i = 2 \dots n$

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

tid  $\times$  antal gange

```
 $c_1 \times n$ 
 $c_2 \times n - 1$ 
 $c_3 \times n - 1$ 
 $c_4 \times \frac{n(n+1)}{2} - 1$ 
 $c_5 \times \frac{n(n-1)}{2}$ 
 $c_6 \times \frac{n(n-1)}{2}$ 
 $c_7 \times n - 1$ 
```

Og husk, vi er primært interesseret i worst case. Men det efterlader os så med...

$$T(N) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left( \frac{n(n+1)}{2} - 1 \right) \\ + c_5 \left( \frac{n(n-1)}{2} \right) + c_6 \left( \frac{n(n-1)}{2} \right) + c_7(n-1)$$

Og husk, vi er primært interesseret i worst case. Men det efterlader os så med...

$$\begin{aligned}T(N) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left( \frac{n(n+1)}{2} - 1 \right) \\&\quad + c_5 \left( \frac{n(n-1)}{2} \right) + c_6 \left( \frac{n(n-1)}{2} \right) + c_7(n-1) \\&= \left( \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right) n^2 + \left( c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 \right) n \\&\quad - (c_2 + c_3 + c_4 + c_7)\end{aligned}$$



Og husk, vi er primært interesseret i worst case. Men det efterlader os så med...

$$\begin{aligned}T(N) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left( \frac{n(n+1)}{2} - 1 \right) \\&\quad + c_5 \left( \frac{n(n-1)}{2} \right) + c_6 \left( \frac{n(n-1)}{2} \right) + c_7(n-1) \\&= \left( \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right) n^2 + \left( c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 \right) n \\&\quad - (c_2 + c_3 + c_4 + c_7) \\&= an^2 + bn + c\end{aligned}$$

# Asymptotisk analyse

## Order of growth

Det leder os videre til en ny måde at tale om kompleksitet på, nemlig i termer af **order of growth**.

- Den eksakte køretid er sjældent særligt relevant — vi vil hellere abstrahere
- For små inputs er kørtiden også irrelevant (computere er hurtige!)
- For store inputs er konstanter og små termer irrelevante — det essentielle er, hvordan kørtiden **udvikler sig** som en funktion af  $n$
- Vi studerer derfor **asymptotisk køretid**
  - ▶ Hvordan vokser kørtiden, når inputtet bliver større?

# Asymptotisk notation

Big-Oh, Big-Omega, Big-Theta

Målet med asymptotisk analyse er forenkle udtrykket for køretiden ved at abstrahere irrelevante og svært forudsigelige faktorer væk og istedet fange 'essensen' af  $T(n)$  — nemlig den dominerende term, når  $n$  går mod  $\infty$ .

# Asymptotisk notation

Big-Oh, Big-Omega, Big-Theta

Målet med asymptotisk analyse er forenkle udtrykket for køretiden ved at abstrahere irrelevante og svært forudsigelige faktorer væk og istedet fange 'essensen' af  $T(n)$  — nemlig den dominerende term, når  $n$  går mod  $\infty$ .

Vi har 3 notationer, vi bruger:

Big-Oh,  $O$  Asymptotisk **upper bound**

Big-Omega,  $\Omega$  Asymptotisk **lower bound**

Big-Theta,  $\Theta$  Asymptotisk **tight bound**

Både  $O$ ,  $\Omega$  og  $\Theta$  definerer **sæt af funktioner**, som en funktion for tidskompleksiteten kan høre til.

# Asymptotisk notation

## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

# Asymptotisk notation

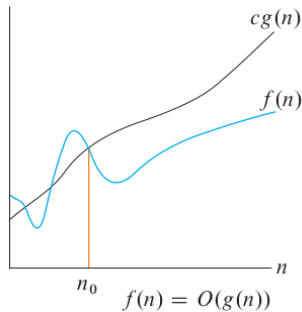
## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = O(g(n))$  hvis  $T(n) \in O(g(n))$



# Asymptotisk notation

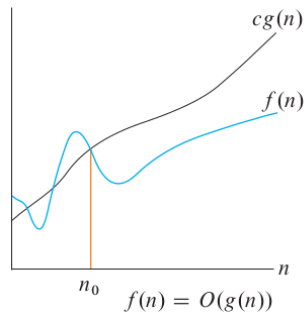
## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = O(g(n))$  hvis  $T(n) \in O(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk langsommere end  $g(n)$



# Asymptotisk notation

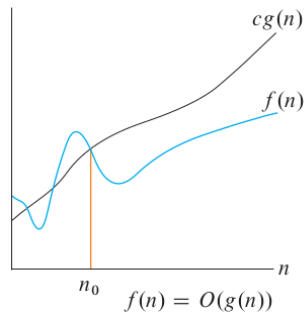
## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = O(g(n))$  hvis  $T(n) \in O(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk langsommere end  $g(n)$
- Eksempler:





# Asymptotisk notation

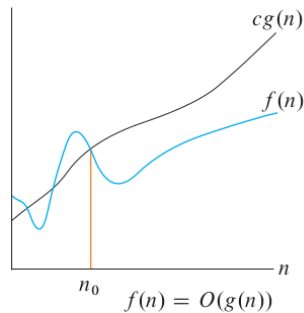
## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = O(g(n))$  hvis  $T(n) \in O(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk langsommere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = O(n^3)$



# Asymptotisk notation

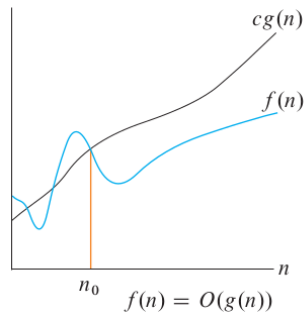
## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = O(g(n))$  hvis  $T(n) \in O(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk langsommere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = O(n^3)$
  - ▶  $T(n) = 23n^3 + 1000n = O(n^4)$



# Asymptotisk notation

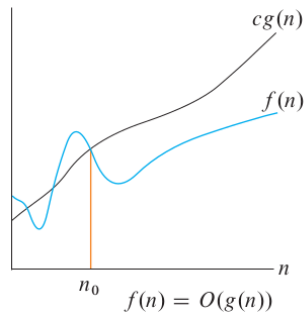
## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = O(g(n))$  hvis  $T(n) \in O(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk langsommere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = O(n^3)$
  - ▶  $T(n) = 23n^3 + 1000n = O(n^4)$
  - ▶  $T(n) = 2^n + 41n^27 = O(2^n)$



# Asymptotisk notation

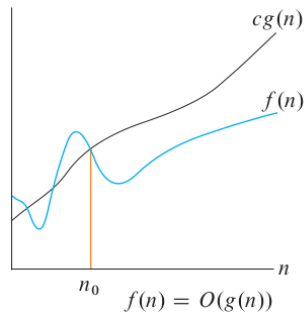
## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = O(g(n))$  hvis  $T(n) \in O(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk langsommere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = O(n^3)$
  - ▶  $T(n) = 23n^3 + 1000n = O(n^4)$
  - ▶  $T(n) = 2^n + 41n^27 = O(2^n)$
  - ▶  $T(n) = 100 = O(1)$



# Asymptotisk notation

## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

Eksempel:

# Asymptotisk notation

## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

Eksempel:

- Vi vil vise, at funktionen  $f(n) = n^2 + 1000n + 500 = O(n^2)$

# Asymptotisk notation

## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

Eksempel:

- Vi vil vise, at funktionen  $f(n) = n^2 + 1000n + 500 = O(n^2)$
- Vi skal dermed finde  $c$  og  $n_0$  således, at  $n^2 + 1000n + 500 \leq cn^2$  for alle  $n \geq n_0$

# Asymptotisk notation

## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

Eksempel:

- Vi vil vise, at funktionen  $f(n) = n^2 + 1000n + 500 = O(n^2)$
- Vi skal dermed finde  $c$  og  $n_0$  således, at  $n^2 + 1000n + 500 \leq cn^2$  for alle  $n \geq n_0$
- Vi dividerer begge sider med  $n^2$ , hvilket giver  $1 + 1000/n + 500/n^2 \leq c$



# Asymptotisk notation

## Big-Oh

### Definition (Big-Oh, $O$ )

For en given funktion  $g(n)$  er  $O(g(n))$  det sæt af funktioner, således at

$$O(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq f(n) \leq cg(n) \text{ for alle } n \geq n_0\}$$

Eksempel:

- Vi vil vise, at funktionen  $f(n) = n^2 + 1000n + 500 = O(n^2)$
- Vi skal dermed finde  $c$  og  $n_0$  således, at  $n^2 + 1000n + 500 \leq cn^2$  for alle  $n \geq n_0$
- Vi dividerer begge sider med  $n^2$ , hvilket giver  $1 + 1000/n + 500/n^2 \leq c$
- Her skulle det være nemt at se, at jo større  $n_0$ , jo mindre et  $c$  kan vi klare os med — f.eks. ved  $n_0 = 2$  bliver venstresiden af uligheden 629, og vi kan vælge et hvilket som helst  $c \geq 629$ . Hvis vi vælger  $n_0 = 100$  kan vi vælge et  $c \geq 11.05$

# Asymptotisk notation

## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

# Asymptotisk notation

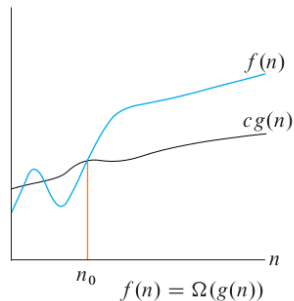
## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = \Omega(g(n))$  hvis  $T(n) \in \Omega(g(n))$



# Asymptotisk notation

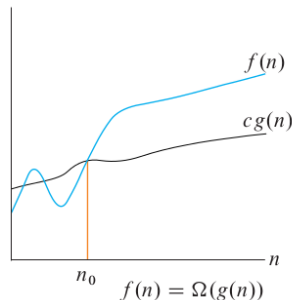
## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = \Omega(g(n))$  hvis  $T(n) \in \Omega(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk hurtigere end  $g(n)$



# Asymptotisk notation

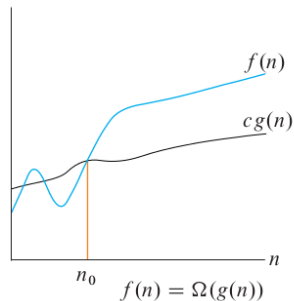
## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = \Omega(g(n))$  hvis  $T(n) \in \Omega(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk hurtigere end  $g(n)$
- Eksempler:



# Asymptotisk notation

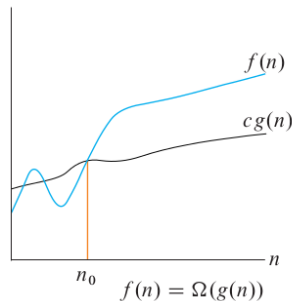
## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = \Omega(g(n))$  hvis  $T(n) \in \Omega(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk hurtigere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = \Omega(n^3)$



# Asymptotisk notation

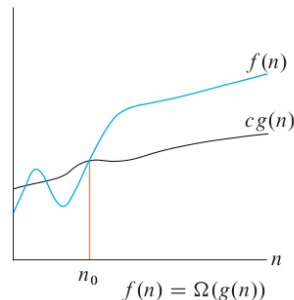
## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = \Omega(g(n))$  hvis  $T(n) \in \Omega(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk hurtigere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = \Omega(n^3)$
  - ▶  $T(n) = 23n^3 + 1000n = \Omega(n)$



# Asymptotisk notation

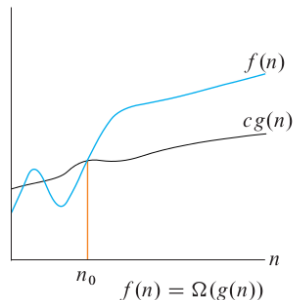
## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = \Omega(g(n))$  hvis  $T(n) \in \Omega(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk hurtigere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = \Omega(n^3)$
  - ▶  $T(n) = 23n^3 + 1000n = \Omega(n)$
  - ▶  $T(n) = 2^n + 41n^{27} = \Omega(2^n)$





# Asymptotisk notation

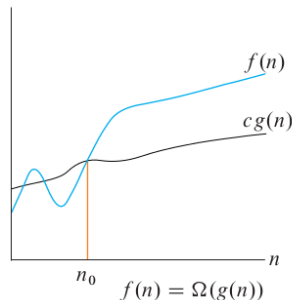
## Big-Omega

### Definition (Big-Omega, $\Omega$ )

For en given funktion  $g(n)$  er  $\Omega(g(n))$  det sæt af funktioner, således at

$$\Omega(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c \text{ og } n_0 \\ \text{sådan at } 0 \leq cg(n) \leq f(n) \text{ for alle } n \geq n_0\}$$

- Vi skriver  $T(n) = \Omega(g(n))$  hvis  $T(n) \in \Omega(g(n))$
- Intuition:  $T(n)$  vokser asymptotisk hurtigere end  $g(n)$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = \Omega(n^3)$
  - ▶  $T(n) = 23n^3 + 1000n = \Omega(n)$
  - ▶  $T(n) = 2^n + 41n^{27} = \Omega(2^n)$
  - ▶  $T(n) = 100 = \Omega(1)$



# Asymptotisk notation

## Big-Theta

### Definition (Big-Theta, $\Theta$ )

For en given funktion  $g(n)$  er  $\Theta(g(n))$  det sæt af funktioner, således at

$$\Theta(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \\ \text{sådan at } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for alle } n \geq n_0\}$$

# Asymptotisk notation

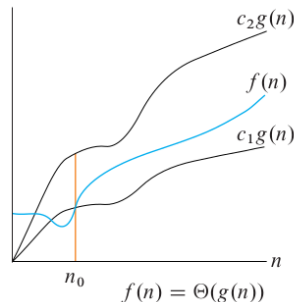
## Big-Theta

### Definition (Big-Theta, $\Theta$ )

For en given funktion  $g(n)$  er  $\Theta(g(n))$  det sæt af funktioner, således at

$$\Theta(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \\ \text{sådan at } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for alle } n \geq n_0\}$$

- Intuition:  $g(n)$  er et asymptotisk tight bound for  $T(n)$



# Asymptotisk notation

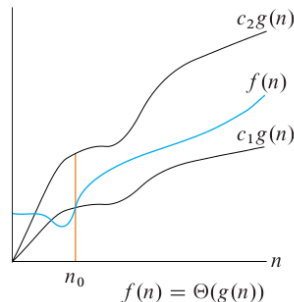
## Big-Theta

### Definition (Big-Theta, $\Theta$ )

For en given funktion  $g(n)$  er  $\Theta(g(n))$  det sæt af funktioner, således at

$$\Theta(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \\ \text{sådan at } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for alle } n \geq n_0\}$$

- Intuition:  $g(n)$  er et asymptotisk tight bound for  $T(n)$
- Theorem: for to funktioner  $f(n)$  og  $g(n)$  har vi at  $f(n) = \Theta(g(n))$  hvis og kun hvis  $f(n) = \Omega(g(n))$  og  $f(n) = O(g(n))$



# Asymptotisk notation

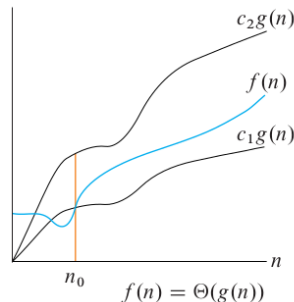
## Big-Theta

### Definition (Big-Theta, $\Theta$ )

For en given funktion  $g(n)$  er  $\Theta(g(n))$  det sæt af funktioner, således at

$$\Theta(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \\ \text{sådan at } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for alle } n \geq n_0\}$$

- Intuition:  $g(n)$  er et asymptotisk tight bound for  $T(n)$
- Theorem: for to funktioner  $f(n)$  og  $g(n)$  har vi at  $f(n) = \Theta(g(n))$  hvis og kun hvis  $f(n) = \Omega(g(n))$  og  $f(n) = O(g(n))$
- Eksempler:



# Asymptotisk notation

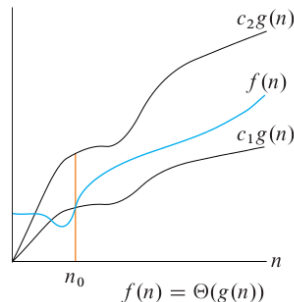
## Big-Theta

### Definition (Big-Theta, $\Theta$ )

For en given funktion  $g(n)$  er  $\Theta(g(n))$  det sæt af funktioner, således at

$$\Theta(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \text{ sådan at } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for alle } n \geq n_0\}$$

- Intuition:  $g(n)$  er et asymptotisk tight bound for  $T(n)$
- Theorem: for to funktioner  $f(n)$  og  $g(n)$  har vi at  $f(n) = \Theta(g(n))$  hvis og kun hvis  $f(n) = \Omega(g(n))$  og  $f(n) = O(g(n))$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = \Theta(n^3)$



# Asymptotisk notation

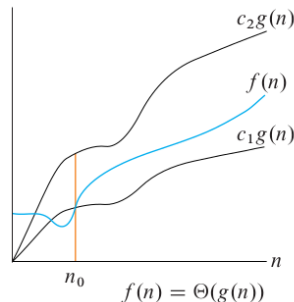
## Big-Theta

### Definition (Big-Theta, $\Theta$ )

For en given funktion  $g(n)$  er  $\Theta(g(n))$  det sæt af funktioner, således at

$$\Theta(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \\ \text{sådan at } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for alle } n \geq n_0\}$$

- Intuition:  $g(n)$  er et asymptotisk tight bound for  $T(n)$
- Theorem: for to funktioner  $f(n)$  og  $g(n)$  har vi at  $f(n) = \Theta(g(n))$  hvis og kun hvis  $f(n) = \Omega(g(n))$  og  $f(n) = O(g(n))$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = \Theta(n^3)$
  - ▶  $T(n) = 2^n + 41n^{27} = \Theta(2^n)$



# Asymptotisk notation

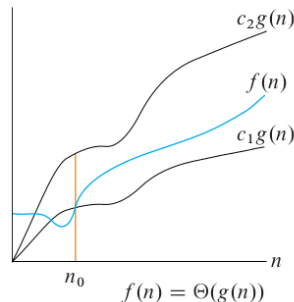
## Big-Theta

### Definition (Big-Theta, $\Theta$ )

For en given funktion  $g(n)$  er  $\Theta(g(n))$  det sæt af funktioner, således at

$$\Theta(g(n)) = \{f(n) : \text{der eksisterer positive konstanter } c_1, c_2 \text{ og } n_0 \\ \text{sådan at } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for alle } n \geq n_0\}$$

- Intuition:  $g(n)$  er et asymptotisk tight bound for  $T(n)$
- Theorem: for to funktioner  $f(n)$  og  $g(n)$  har vi at  $f(n) = \Theta(g(n))$  hvis og kun hvis  $f(n) = \Omega(g(n))$  og  $f(n) = O(g(n))$
- Eksempler:
  - ▶  $T(n) = 23n^3 + 1000n = \Theta(n^3)$
  - ▶  $T(n) = 2^n + 41n^{27} = \Theta(2^n)$
  - ▶  $T(n) = 100 = \Theta(1)$





# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$
- Ignorer mindre termer

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$
- Ignorer mindre termer
  - ▶  $T(n) = n^3 + 1000n^2 - n \log n + 13n = \Theta(n^3)$

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$
- Ignorer mindre termer
  - ▶  $T(n) = n^3 + 1000n^2 - n \log n + 13n = \Theta(n^3)$
- Hvordan identificerer man mindre termer?

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$
- Ignorer mindre termer
  - ▶  $T(n) = n^3 + 1000n^2 - n \log n + 13n = \Theta(n^3)$
- Hvordan identificerer man mindre termer?
  - ▶  $c < \log n < n < n \log n < n^a < b^n < n!$

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$
- Ignorer mindre termer
  - ▶  $T(n) = n^3 + 1000n^2 - n \log n + 13n = \Theta(n^3)$
- Hvordan identificerer man mindre termer?
  - ▶  $c < \log n < n < n \log n < n^a < b^n < n!$
  - ▶ Konstant, logaritmisk, log linear, polynomial, eksponentiel, fakultet



# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$
- Ignorer mindre termer
  - ▶  $T(n) = n^3 + 1000n^2 - n \log n + 13n = \Theta(n^3)$
- Hvordan identificerer man mindre termer?
  - ▶  $c < \log n < n < n \log n < n^a < b^n < n!$
  - ▶ Konstant, logaritmisk, log linear, polynomial, eksponentiel, fakultet

# Asymptotisk notation

## Tips og tricks

Denne nemme måde ('ingeniørmetoden') til at bruge asymptotisk notation:

- Ignorer indledende konstanter
  - ▶  $T(n) = 1000n^2 = \Theta(n^2)$
- Ignorer mindre termer
  - ▶  $T(n) = n^3 + 1000n^2 - n \log n + 13n = \Theta(n^3)$
- Hvordan identificerer man mindre termer?
  - ▶  $c < \log n < n < n \log n < n^a < b^n < n!$
  - ▶ Konstant, logaritmisk, log linear, polynomial, eksponentiel, fakultet

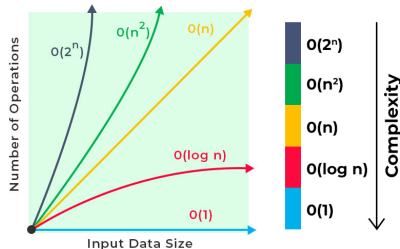


Figure: Source: <https://www.geeksforgeeks.org/what-is-logarithmic-time-complexity/>

# Asymptotisk analyse

## Insertion sort

Vi slutter, hvor vi startede — med Insertion-Sort. Nu da vi kender til asymptotisk analyse og notation, kan vi så gribe vores analyse lidt lettere an?

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

# Asymptotisk analyse

## Insertion sort

Vi slutter, hvor vi startede — med Insertion-Sort. Nu da vi kender til asymptotisk analyse og notation, kan vi så gribe vores analyse lidt lettere an?

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

- Vi ser, at hele algoritmen er pakket ind i en for-løkke, der kører  $\Theta(n)$  gange

# Asymptotisk analyse

## Insertion sort

Vi slutter, hvor vi startede — med Insertion-Sort. Nu da vi kender til asymptotisk analyse og notation, kan vi så gribe vores analyse lidt lettere an?

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

- Vi ser, at hele algoritmen er pakket ind i en for-løkke, der kører  $\Theta(n)$  gange
- Vi ser, at der i for-løkken er en while-løkke, der i worst case selv kører  $\Theta(n)$  gange

# Asymptotisk analyse

## Insertion sort

Vi slutter, hvor vi startede — med Insertion-Sort. Nu da vi kender til asymptotisk analyse og notation, kan vi så gribe vores analyse lidt lettere an?

INSERTION-SORT( $A$ )

```
1  for  $i = 2$  to  $n$ 
2       $key = A[i]$ 
3       $j = i - 1$ 
4      while  $j > 0$  and  $A[j] > key$ 
5           $A[j + 1] = A[j]$ 
6           $j = j - 1$ 
7       $A[j + 1] = key$ 
```

- Vi ser, at hele algoritmen er pakket ind i en for-løkke, der kører  $\Theta(n)$  gange
- Vi ser, at der i for-løkken er en while-løkke, der i worst case selv kører  $\Theta(n)$  gange
- Resten af linierne er konstanter, altså har vi  $T(n) = \Theta(n) \cdot \Theta(n) = \Theta(n^2)$

# Dagens temaer

## Opsummering

- Vi har mødt vores første sorteringsalgoritme — Insertion-Sort!
  - ▶ Simpel at implementere og forstå
  - ▶ God til næsten sorterede sekvenser
  - ▶ Den asymptotiske worst case køretid er kvadratisk
- Loop invarianter og korrekthed
  - ▶ Initialization, maintenance og termination
- Asymptotisk analyse og notation
  - ▶  $O, \Omega, \Theta$

# Tak for i dag!

Flere exercises..

Den bedste måde ikke at snyde sig selv på er lave exercises!



AALBORG  
UNIVERSITET