# Divide and Conquer & The Master Theorem

## Algorithms and Datastructures, F25, Lecture 3

Andreas Holck Høeg-Petersen

Department of Computer Science
Aalborg University

February 12, 2025

# Opdateringer



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

# Outline

# Outline

# Divide and Conquer
## Algoritmiske teknikker

AALBORG UNIVERSITET

Divide-and-conquer er en effektiv teknik til at designe effektive algoritmer til at løse komplekse problemer ved at bryde dem ned i mindre dele. Ofte giver det en asymptotisk køretid i $\Theta(n \log n)$.

Divide-and-conquer er en effektiv teknik til at designe effektive algoritmer til at løse komplekse problemer ved at bryde dem ned i mindre dele. Ofte giver det en asymptotisk køretid i $\Theta(n \log n)$.

Metoden har overordnet set 3 skridt:

# Divide and Conquer

## Algoritmiske teknikker

Divide-and-conquer er en effektiv teknik til at designe effektive algoritmer til at løse komplekse problemer ved at bryde dem ned i mindre dele. Ofte giver det en asymptotisk køretid i $\Theta(n \log n)$.

Metoden har overordnet set 3 skridt:

**Divide** Del problemet op i et eller flere sub-problemer, der er mindre instanser af det samme problem

Divide-and-conquer er en effektiv teknik til at designe effektive algoritmer til at løse komplekse problemer ved at bryde dem ned i mindre dele. Ofte giver det en asymptotisk køretid i $\Theta(n \log n)$.

Metoden har overordnet set 3 skridt:

**Divide** Del problemet op i et eller flere sub-problemer, der er mindre instanser af det samme problem

**Conquer** Løs sub-problemerne rekursivt

# Divide and Conquer

## Algoritmiske teknikker

Divide-and-conquer er en effektiv teknik til at designe effektive algoritmer til at løse komplekse problemer ved at bryde dem ned i mindre dele. Ofte giver det en asymptotisk køretid i $\Theta(n \log n)$.

Metoden har overordnet set 3 skridt:

**Divide** Del problemet op i et eller flere sub-problemer, der er mindre instanser af det samme problem

**Conquer** Løs sub-problemerne rekursivt

**Combine** Kombiner løsningerne på sub-problemerne til en løsning på det oprindelige problem

# Divide and Conquer

## Algoritmiske teknikker

Divide-and-conquer er en effektiv teknik til at designe effektive algoritmer til at løse komplekse problemer ved at bryde dem ned i mindre dele. Ofte giver det en asymptotisk køretid i $\Theta(n \log n)$.

Metoden har overordnet set 3 skridt:

**Divide** Del problemet op i et eller flere sub-problemer, der er mindre instanser af det samme problem

**Conquer** Løs sub-problemerne rekursivt

**Combine** Kombiner løsningerne på sub-problemerne til en løsning på det oprindelige problem

# Divide and Conquer

## Algoritmiske teknikker

Divide-and-conquer er en effektiv teknik til at designe effektive algoritmer til at løse komplekse problemer ved at bryde dem ned i mindre dele. Ofte giver det en asymptotisk køretid i $\Theta(n \log n)$.

Metoden har overordnet set 3 skridt:

- **Divide** Del problemet op i et eller flere sub-problemer, der er mindre instanser af det samme problem
- **Conquer** Løs sub-problemerne rekursivt
- **Combine** Kombiner løsningerne på sub-problemerne til en løsning på det oprindelige problem

Hvis problemet er småt nok (base case), løses det uden videre. Ellers (recursive case) fortsætter man rekursionen.

# Divide and Conquer

## Rekursion???

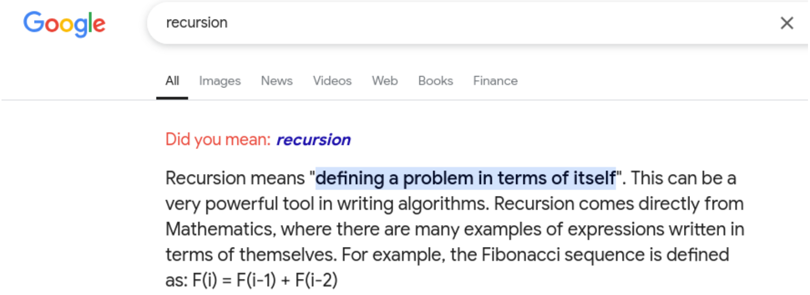



Figure: Google søgning på ‘recursion’

# Divide and Conquer

## Rekursion???

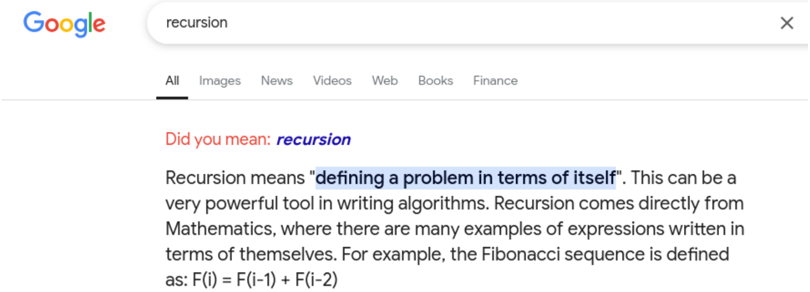



Figure: Google søgning på 'recursion'

**Example (Fibonacci-sekvensen)**

Det næste tal i Fibonacci-sekvensen er givet ved at summere de to foregående elementer. Den starter med 1, 1, 2, 3, 5, 8, 13, . . . . Men den kan dermed også defineres rekursivt, således at det $i$'ende elementer er givet ved $F(i) = F(i-1) + F(i-2)$.

# Divide and Conquer
## Algoritmisk rekursion

I algoritmisk forstand forstår vi en rekursion $T(n)$ således, at der for en tilpas stor konstant $n_0$ skal gælde følgende:

1. For alle $n < n_0$ har vi at $T(n) = \Theta(1)$ — dvs. $T(n)$ er konstant
2. For alle $n \geq n_0$ må alle stier af rekursionen ende i en defineret base case inden for et endeligt antal rekursive kald.

AALBORG UNIVERSITET

I algoritmisk forstand forstår vi en rekursion $T(n)$ således, at der for en tilpas stor konstant $n_0$ skal gælde følgende:

1. For alle $n < n_0$ har vi at $T(n) = \Theta(1)$ — dvs. $T(n)$ er konstant
2. For alle $n \geq n_0$ må alle stier af rekursionen ende i en defineret base case inden for et endeligt antal rekursive kald.

I kurset her gælder det for alle rekursioner, vi ser på, men det er værd at have in mente, hvis I selv designer algoritmer, som gør brug af rekursion.

AALBORG UNIVERSITET

I dag skal vi se på to eksempler på divide-and-conquer-algoritmer:

- Merge sort
- Quicksort

# Outline

# Merge sort

## Den kender I jo!

- En af de mest berømte og benyttede sorteringsalgoritmer — og en af de første til at blive implementeret i en computer (ca. 1945 af John von Neumann)
- Ide:

| | |
|---|---|
| Divide | Opdel sekvensen i to lige store sub-sekvenser og kald algoritmen rekursivt |
| Conquer | Når algoritmen modtager en sekvens med kun et element, returner det trivielt sorterede element |
| Combine | Kombiner de sorterede sub-sekvenser, så sorteringsrækkefølgen overholdes |

# Merge sort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$

```
Merge-Sort(A, p, r)
if p ≥ r
        return
q = ⌊(p + r)/2⌋
Merge-Sort(A, p, q)
Merge-Sort(A, q + 1, r)
Merge(A, p, q, r)
```

# Merge sort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså Merge-Sort$(A, 1, n)$

```
Merge-Sort(A, p, r)
if p ≥ r
        return
q = ⌊(p + r)/2⌋
Merge-Sort(A, p, q)
Merge-Sort(A, q + 1, r)
Merge(A, p, q, r)
```

# Merge sort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså Merge-Sort$(A, 1, n)$
- I linie 3 finder vi midtpunktet mellem $p$ og $r$

```
Merge-Sort(A, p, r)
1  if p ≥ r
2       return
3  q = ⌊(p + r)/2⌋
4  Merge-Sort(A, p, q)
5  Merge-Sort(A, q + 1, r)
6  Merge(A, p, q, r)
```

# Merge sort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså Merge-Sort$(A, 1, n)$
- I linie 3 finder vi midtpunktet mellem $p$ og $r$
- I linie 4 og 5 kalder vi rekursivt for den ene og anden halvdel af sekvensen

```
Merge-Sort(A, p, r)
1  if p ≥ r
2        return
3  q = ⌊(p + r)/2⌋
4  Merge-Sort(A, p, q)
5  Merge-Sort(A, q + 1, r)
6  Merge(A, p, q, r)
```

# Merge sort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså Merge-Sort$(A, 1, n)$
- I linie 3 finder vi midtpunktet mellem $p$ og $r$
- I linie 4 og 5 kalder vi rekursivt for den ene og anden halvdel af sekvensen
- I linie 5 kombinerer ('merger') vi de to halvdele sammen

```
Merge-Sort(A, p, r)
1  if p ≥ r
2      return
3  q = ⌊(p + r)/2⌋
4  Merge-Sort(A, p, q)
5  Merge-Sort(A, q + 1, r)
6  Merge(A, p, q, r)
```

# Merge sort

## Merge-operationen

Merge-operationen er et rædselsfuldt monster i CLRS. . . !

```
MERGE(A, p, q, r)
n_L = q − p + 1            // length of A[p : q]
n_R = r − q                // length of A[q + 1 : r]
let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
for i = 0 to n_L − 1       // copy A[p : q] into L[0 : n_L − 1]
    L[i] = A[p + i]
for j = 0 to n_R − 1       // copy A[q + 1 : r] into R[0 : n_R − 1]
    R[j] = A[q + j + 1]
i = 0                      // i indexes the smallest remaining element in L
j = 0                      // j indexes the smallest remaining element in R
k = p                      // k indexes the location in A to fill
// As long as each of the arrays L and R contains an unmerged element,
//      copy the smallest unmerged element back into A[p : r].
while i < n_L and j < n_R
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
    k = k + 1
// Having gone through one of L and R entirely, copy the
//      remainder of the other to the end of A[p : r].
while i < n_L
    A[k] = L[i]
    i = i + 1
    k = k + 1
while j < n_R
    A[k] = R[j]
    j = j + 1
    k = k + 1
```

Figure: Ew!!

# Merge sort

## Merge-operationen

En lidt mere venlig version kunne se sådan her ud:

```
Merge(A, p, q, r)
let B[0 : r − p] be a new array with 0-index
for i = 0 to r − p
    B[i] = A[i + p]
i = 0, j = (r − q)
for k = p to r
    if (i + p) > q
        A[k] = B[j]
        j = j + 1
    elseif (j + q) > r
        A[k] = B[i]
        i = i + 1
    elseif B[j] < B[i]
        A[k] = B[j]
        j = j + 1
    else
        A[k] = B[i]
        i = i + 1
```

# Merge sort

## Merge-operationen

Og her endda med forståelige navne:

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be a new array with 0-index
for i = 0 to B.length
    B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
    if (i + low) ≥ mid
        A[k] = B[j]
        j = j + 1
    elseif (j + low) ≥ high
        A[k] = B[i]
        i = i + 1
    elseif B[j] < B[i]
        A[k] = B[j]
        j = j + 1
    else
        A[k] = B[i]
        i = i + 1
```

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
      a new array with 0-index
for i = 0 to B.length
      B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
      if (i + low) ≥ mid
            A[k] = B[j]
            j = j + 1
      elseif (j + low) ≥ high
            A[k] = B[i]
            i = i + 1
      elseif B[j] < B[i]
            A[k] = B[j]
            j = j + 1
      else
            A[k] = B[i]
            i = i + 1
```

- Vi lader $low = p$, $mid = q + 1$ og $high = r + 1$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
  low = p, mid = q + 1, high = r + 1
  let B[0 : high − low] be
          a new array with 0-index
  for i = 0 to B.length
          B[i] = A[i + low]

  i = 0, j = (mid − low)
  for k = low to high
          if (i + low) ≥ mid
                  A[k] = B[j]
                  j = j + 1
          elseif (j + low) ≥ high
                  A[k] = B[i]
                  i = i + 1
          elseif B[j] < B[i]
                  A[k] = B[j]
                  j = j + 1
          else
                  A[k] = B[i]
                  i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$
- Hvis $i$ er forbi midten, tag næste element fra $B[j : r - p]$ og inkrementer $j$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$
- Hvis $i$ er forbi midten, tag næste element fra $B[j : r - p]$ og inkrementer $j$
- Hvis $j$ er forbi slutningen, tag næste element fra $B[0 : q - p]$ og inkrementer $i$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$
- Hvis $i$ er forbi midten, tag næste element fra $B[j : r - p]$ og inkrementer $j$
- Hvis $j$ er forbi slutningen, tag næste element fra $B[0 : q - p]$ og inkrementer $i$
- Hvis $B[j]$ er lavere end $B[i]$, sæt $A[k]$ til $B[j]$ og inkrementer $j$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$
- Hvis $i$ er forbi midten, tag næste element fra $B[j : r - p]$ og inkrementer $j$
- Hvis $j$ er forbi slutningen, tag næste element fra $B[0 : q - p]$ og inkrementer $i$
- Hvis $B[j]$ er lavere end $B[i]$, sæt $A[k]$ til $B[j]$ og inkrementer $j$
- Ellers, sæt $A[k]$ til $B[i]$ og inkrementer $i$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$
- Hvis $i$ er forbi midten, tag næste element fra $B[j : r - p]$ og inkrementer $j$
- Hvis $j$ er forbi slutningen, tag næste element fra $B[0 : q - p]$ og inkrementer $i$
- Hvis $B[j]$ er lavere end $B[i]$, sæt $A[k]$ til $B[j]$ og inkrementer $j$
- Ellers, sæt $A[k]$ til $B[i]$ og inkrementer $i$
- Vi har nu lagt elementerne fra $B$ tilbage i $A$ i sorteret rækkefølge!

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p, mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$
- Hvis $i$ er forbi midten, tag næste element fra $B[j : r - p]$ og inkrementer $j$
- Hvis $j$ er forbi slutningen, tag næste element fra $B[0 : q - p]$ og inkrementer $i$
- Hvis $B[j]$ er lavere end $B[i]$, sæt $A[k]$ til $B[j]$ og inkrementer $j$
- Ellers, sæt $A[k]$ til $B[i]$ og inkrementer $i$
- Vi har nu lagt elementerne fra $B$ tilbage i $A$ i sorteret rækkefølge!
- Forskellen fra CLRS er, at vi samler $L$ og $R$ i et enkelt array $B$

# Merge sort

## Merge-operationen

```
Merge(A, p, q, r)
low = p, mid = q + 1, high = r + 1
let B[0 : high − low] be
        a new array with 0-index
for i = 0 to B.length
        B[i] = A[i + low]

i = 0, j = (mid − low)
for k = low to high
        if (i + low) ≥ mid
                A[k] = B[j]
                j = j + 1
        elseif (j + low) ≥ high
                A[k] = B[i]
                i = i + 1
        elseif B[j] < B[i]
                A[k] = B[j]
                j = j + 1
        else
                A[k] = B[i]
                i = i + 1
```

- Vi lader $low = p$, $mid = q + 1$ og $high = r + 1$
- Kopier $A[p : r]$ til $B[0 : r - p]$
- $i$ og $j$ peger på første og anden del af $B$
- $k$ løber igennem alle indicies i $A[p : r]$
- Hvis $i$ er forbi midten, tag næste element fra $B[j : r - p]$ og inkrementer $j$
- Hvis $j$ er forbi slutningen, tag næste element fra $B[0 : q - p]$ og inkrementer $i$
- Hvis $B[j]$ er lavere end $B[i]$, sæt $A[k]$ til $B[j]$ og inkrementer $j$
- Ellers, sæt $A[k]$ til $B[i]$ og inkrementer $i$
- Vi har nu lagt elementerne fra $B$ tilbage i $A$ i sorteret rækkefølge!
- Forskellen fra CLRS er, at vi samler $L$ og $R$ i et enkelt array $B$
- …og at vi klarer resten i et enkelt loop (fremfor 3, eew!)

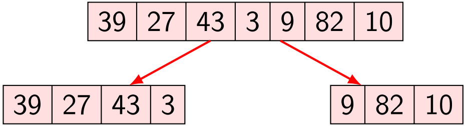
# Merge sort

## Example

| | | | | | | |
|---|---|---|---|---|---|---|
| 39 | 27 | 43 | 3 | 9 | 82 | 10 |

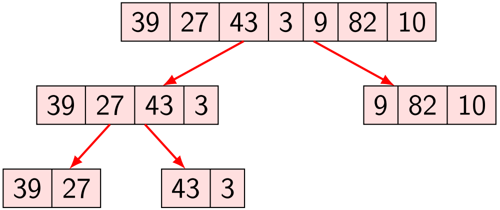
# Merge sort

## Example



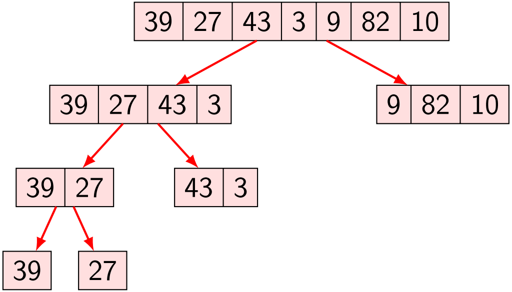
# Merge sort

## Example



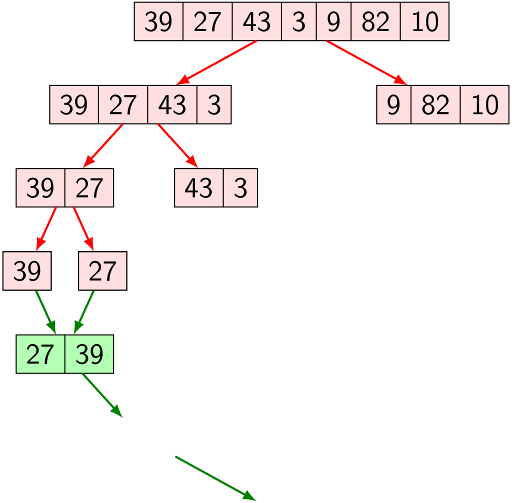
# Merge sort

## Example



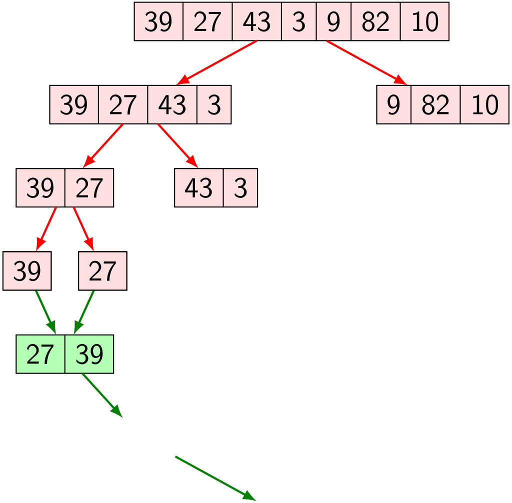
# Merge sort

## Example



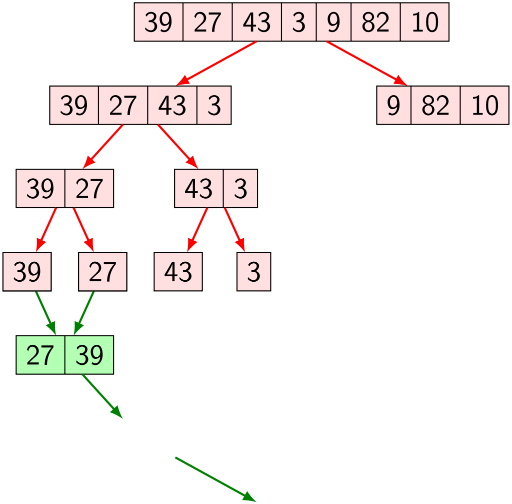
# Merge sort

## Example



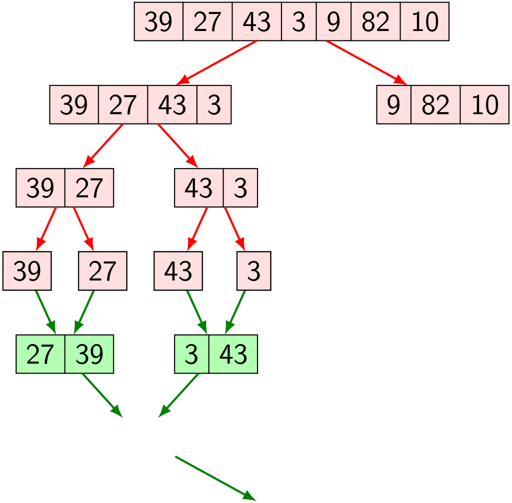
# Merge sort

## Example

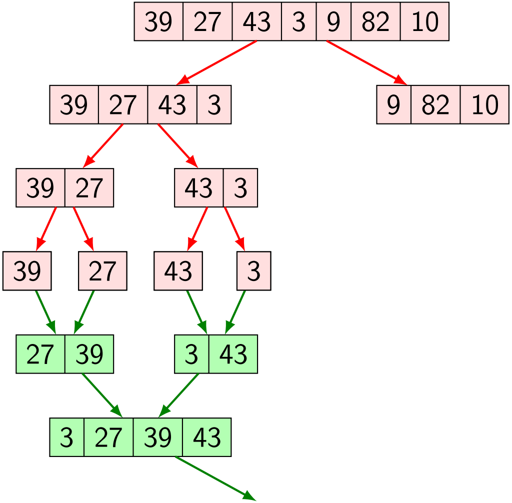


# Merge sort

## Example

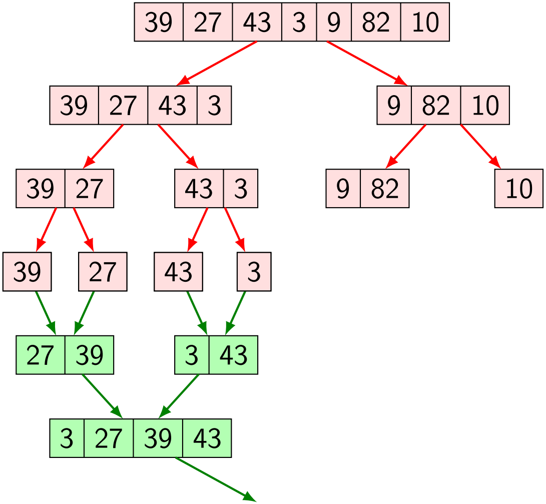


# Merge sort
## Example



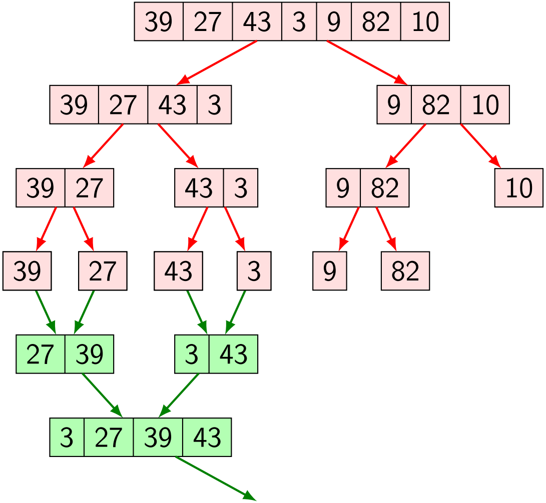
# Merge sort

## Example



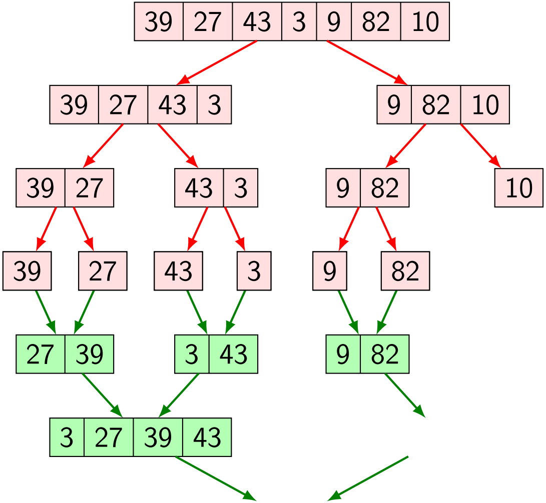
# Merge sort

## Example



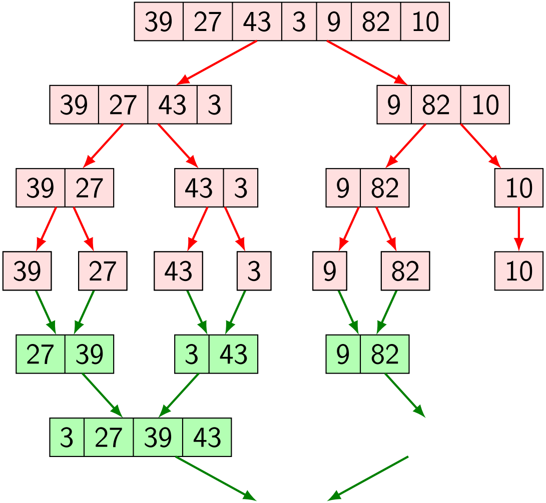
# Merge sort

## Example



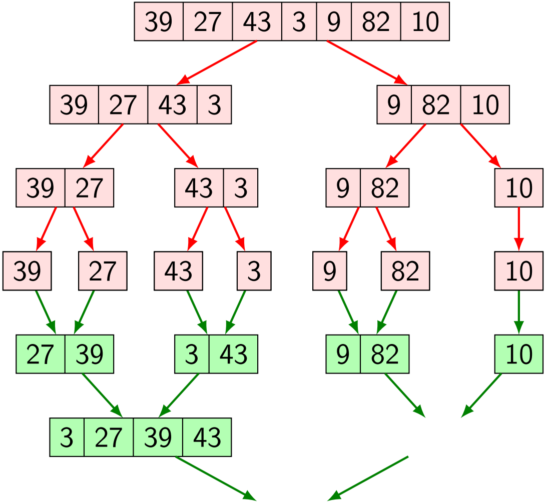
# Merge sort

## Example



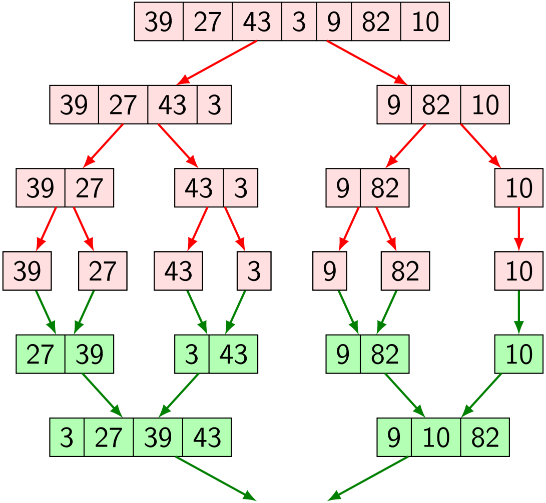
# Merge sort

## Example



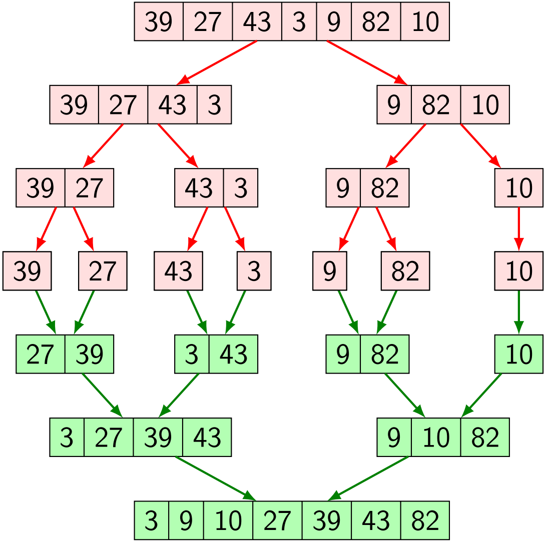
# Merge sort

## Example



# Merge sort

## Example



# Merge sort

## Intuitiv analyse

I næste del af forelæsningen lærer vi om et mere generelt trick til at analysere køretid for rekursive algoritmer, men til en start ser vi på en intuitiv måde at gå til analysen på.

# Merge sort

## Intuitiv analyse

I næste del af forelæsningen lærer vi om et mere generelt trick til at analysere køretid for rekursive algoritmer, men til en start ser vi på en intuitiv måde at gå til analysen på.

- Vi noterer os, at Merge operationen er $\Theta(n)$

# Merge sort

## Intuitiv analyse

I næste del af forelæsningen lærer vi om et mere generelt trick til at analysere køretid for rekursive algoritmer, men til en start ser vi på en intuitiv måde at gå til analysen på.

- Vi noterer os, at Merge operationen er $\Theta(n)$
- De to rekursitve kald halverer begge input-størrelsen, altså har vi to kald med $n/2$

# Merge sort

## Intuitiv analyse

I næste del af forelæsningen lærer vi om et mere generelt trick til at analysere køretid for rekursive algoritmer, men til en start ser vi på en intuitiv måde at gå til analysen på.

- Vi noterer os, at Merge operationen er $\Theta(n)$
- De to rekursitve kald halverer begge input-størrelsen, altså har vi to kald med $n/2$
- I base case, hvor $n \leq 1$ og inputtet er trivielt sorteret, er køretiden $\Theta(1)$ (konstant)

# Merge sort

## Intuitiv analyse

I næste del af forelæsningen lærer vi om et mere generelt trick til at analysere køretid for rekursive algoritmer, men til en start ser vi på en intuitiv måde at gå til analysen på.

- Vi noterer os, at Merge operationen er $\Theta(n)$
- De to rekursitve kald halverer begge input-størrelsen, altså har vi to kald med $n/2$
- I base case, hvor $n \leq 1$ og inputtet er trivielt sorteret, er køretiden $\Theta(1)$ (konstant)
- Vi har altså:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

# Merge sort

## Intuitiv analyse

I næste del af forelæsningen lærer vi om et mere generelt trick til at analysere køretid for rekursive algoritmer, men til en start ser vi på en intuitiv måde at gå til analysen på.

- Vi noterer os, at Merge operationen er $\Theta(n)$
- De to rekursitve kald halverer begge input-størrelsen, altså har vi to kald med $n/2$
- I base case, hvor $n \leq 1$ og inputtet er trivielt sorteret, er køretiden $\Theta(1)$ (konstant)
- Vi har altså:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

- Spørgsmålet er så, hvor mange gange kan vi halvere $n$ før, at vi når til base case?

# Merge sort

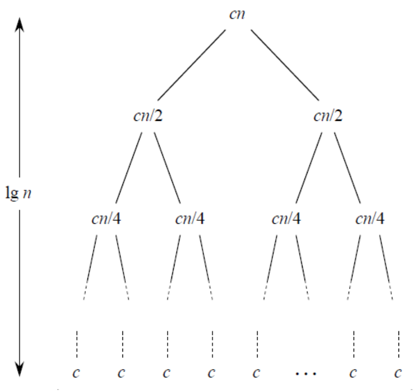
## Intuitiv analyse

I næste del af forelæsningen lærer vi om et mere generelt trick til at analysere køretid for rekursive algoritmer, men til en start ser vi på en intuitiv måde at gå til analysen på.

- Vi noterer os, at Merge operationen er $\Theta(n)$
- De to rekursitve kald halverer begge input-størrelsen, altså har vi to kald med $n/2$
- I base case, hvor $n \leq 1$ og inputtet er trivielt sorteret, er køretiden $\Theta(1)$ (konstant)
- Vi har altså:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

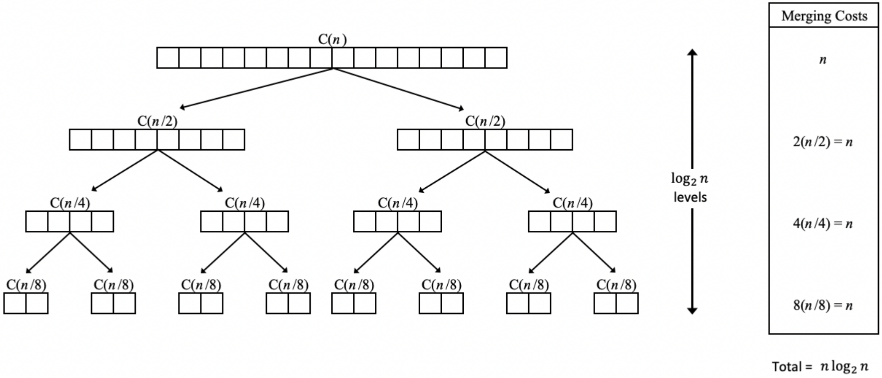
- Spørgsmålet er så, hvor mange gange kan vi halvere $n$ før, at vi når til base case?
- Dette er faktisk selve definitionen på base-2 logaritmen, $\log_2$!



# Merge sort

## Intuitiv analyse

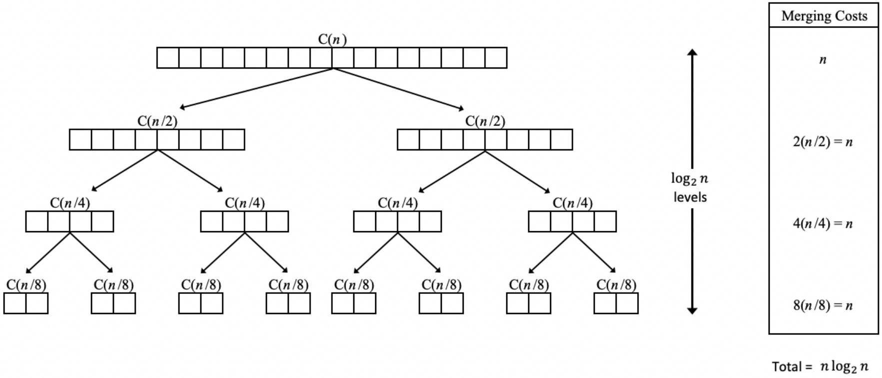
På hvert 'niveau' i træet — som der er $\log_2 n$ af — skal vi samlet set foretage $\Theta(n)$ operationer. F.eks., når vi er på niveau 2, har vi halveret $n$ to gange, så vi har 4 lister af størrelse $n/(2*2) = n/4$, og tydeligvis er $4(n/4) = n$.



# Merge sort

## Intuitiv analyse

På hvert 'niveau' i træet — som der er $\log_2 n$ af — skal vi samlet set foretage $\Theta(n)$ operationer. F.eks., når vi er på niveau 2, har vi halveret $n$ to gange, så vi har 4 lister af størrelse $n/(2 * 2) = n/4$, og tydeligvis er $4(n/4) = n$.




Dermed kan vi sige, at køretiden for Merge-Sort er $T(n) = \Theta(n \log_2 n)$!

# Vi renser lige hovedet. . .

. . . inden vi går til næste algoritme!

# Outline

1. Divide and Conquer
2. Merge sort
3. Quicksort
4. Exercises
5. The Master Theorem

# Quicksort

## Endnu en klassiker

Quicksort er en anden meget populær sorteringsalgoritme, der ligeledes følger divide-and-conquer-metoden. I worst-case er dens køretid $\Theta(n^2)$, men i praksis er den typisk hurtigere end de fleste andre alternativer — og med en simpel modifikation, kan man (næsten) sikre sig, at køretiden er $\Theta(n \log n)$. Derudover er dens pladsforbrug mindre end for merge sort, og implementationen er noget simplere.

# Quicksort

## Divide-and-conquer

De tre dele af divide-and-conquer-metoden for quicksort er:

De tre dele af divide-and-conquer-metoden for quicksort er:

**Divide** Vælg et pivot element $p$, og del input sekvensen op i en 'lav' del og en 'høj' del således, alt i den lave del er mindre end $p$ og alt i den høje er større end eller lig med $p$. Indsæt $p$, så den skiller de to.

# Quicksort

## Divide-and-conquer

De tre dele af divide-and-conquer-metoden for quicksort er:

**Divide** Vælg et pivot element $p$, og del input sekvensen op i en 'lav' del og en 'høj' del således, alt i den lave del er mindre end $p$ og alt i den høje er større end eller lig med $p$. Indsæt $p$, så den skiller de to.

**Conquer** Kald quicksort rekursivt på de to halvdele.

De tre dele af divide-and-conquer-metoden for quicksort er:

**Divide** Vælg et pivot element $p$, og del input sekvensen op i en 'lav' del og en 'høj' del således, alt i den lave del er mindre end $p$ og alt i den høje er større end eller lig med $p$. Indsæt $p$, så den skiller de to.

**Conquer** Kald quicksort rekursivt på de to halvdele.

**Combine** Her behøver vi ikke gøre noget, for når vi når til bunds i rekursionen, så er begge sub-arrays sorterede.

# Quicksort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$

QuickSort($A, p, r$)

**if** $p < r$
    $q$ = Partition($p, r$)
    QuickSort($A, p, q-1$)
    QuickSort($A, q+1, r$)

# Quicksort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså QuickSort$(A, 1, n)$

```
QuickSort(A, p, r)
if p < r
    q = Partition(p, r)
    QuickSort(A, p, q − 1)
    QuickSort(A, q + 1, r)
```

# Quicksort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså QuickSort$(A, 1, n)$
- I linie 2 kalder vi proceduren Partition, som deler $A$ i to og returnerer indexet på pivot-elementet

```
QuickSort(A, p, r)
1  if p < r
2        q = Partition(p, r)
3        QuickSort(A, p, q − 1)
4        QuickSort(A, q + 1, r)
```

# Quicksort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså QuickSort$(A, 1, n)$
- I linie 2 kalder vi proceduren Partition, som deler $A$ i to og returnerer indexet på pivot-elementet
- I linie 3 og 4 kalder vi rekursivt for den ene og anden del af sekvensen

```
QuickSort(A, p, r)
1  if p < r
2        q = Partition(p, r)
3        QuickSort(A, p, q − 1)
4        QuickSort(A, q + 1, r)
```

# Quicksort

## Pseudo-kode del 1

- Input: en sekvens $A[1:n]$ og to indicies $p, r$ hvor $1 \leq p \leq r \leq n$
- Ved første kald er $p = 1$ og $r = n$, altså QuickSort$(A, 1, n)$
- I linie 2 kalder vi proceduren Partition, som deler $A$ i to og returnerer indexet på pivot-elementet
- I linie 3 og 4 kalder vi rekursivt for den ene og anden del af sekvensen
- Og så behøver vi ikke gøre mere!

```
QuickSort(A, p, r)
1  if p < r
2        q = Partition(p, r)
3        QuickSort(A, p, q − 1)
4        QuickSort(A, q + 1, r)
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r]$

```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r]$
- $i$ er det sidste index i den lave del, $j$ er det første index i den høje del

Partition$(A, p, r)$

$x = A[r]$
$i = p - 1$
**for** $j = p$ **to** $r - 1$
    **if** $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exhange $A[i + 1]$ with $x$
**return** $i + 1$

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r]$
- $i$ er det sidste index i den lave del, $j$ er det første index i den høje del
- Vi løber igennem alle elementer, bortset fra pivot-elementet

```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r]$
- $i$ er det sidste index i den lave del, $j$ er det første index i den høje del
- Vi løber igennem alle elementer, bortset fra pivot-elementet
- I linie 4 tjekker vi, om $A[j]$ hører til i den lave del (er mindre end $x$)

```
Partition(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              exchange A[i] with A[j]
7  exhange A[i + 1] with x
8  return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r]$
- $i$ er det sidste index i den lave del, $j$ er det første index i den høje del
- Vi løber igennem alle elementer, bortset fra pivot-elementet
- I linie 4 tjekker vi, om $A[j]$ hører til i den lave del (er mindre end $x$)
- I linie 5 'gør vi plads' i den lave ende ved at inkrementere $i$ (bemærk at på dette tidspunkt er $A[i] > x$)

```
Partition(A, p, r)
1   x = A[r]
2   i = p − 1
3   for j = p to r − 1
4       if A[j] ≤ x
5           i = i + 1
6           exchange A[i] with A[j]
7   exhange A[i + 1] with x
8   return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r]$
- $i$ er det sidste index i den lave del, $j$ er det første index i den høje del
- Vi løber igennem alle elementer, bortset fra pivot-elementet
- I linie 4 tjekker vi, om $A[j]$ hører til i den lave del (er mindre end $x$)
- I linie 5 'gør vi plads' i den lave ende ved at inkrementere $i$ (bemærk at på dette tidspunkt er $A[i] > x$)
- I linie 6 bytter vi $A[j]$ (som er mindre end $x$) ud med $A[i]$ (som er højere end $x$)

```
Partition(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exhange A[i + 1] with x
8  return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r]$
- $i$ er det sidste index i den lave del, $j$ er det første index i den høje del
- Vi løber igennem alle elementer, bortset fra pivot-elementet
- I linie 4 tjekker vi, om $A[j]$ hører til i den lave del (er mindre end $x$)
- I linie 5 'gør vi plads' i den lave ende ved at inkrementere $i$ (bemærk at på dette tidspunkt er $A[i] > x$)
- I linie 6 bytter vi $A[j]$ (som er mindre end $x$) ud med $A[i]$ (som er højere end $x$)
- Efter loopet flytter vi $x$ til den første plads i den høje del, $A[i+1]$ — dermed er alt i $A[p:i] \leq x$ og alt i $A[i+2:r] \geq x$

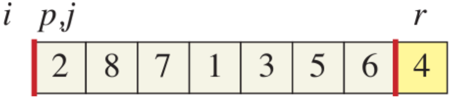
Partition$(A, p, r)$

1 $x = A[r]$
2 $i = p - 1$
3 **for** $j = p$ **to** $r - 1$
4     **if** $A[j] \leq x$
5         $i = i + 1$
6         exchange $A[i]$ with $A[j]$
7 exhange $A[i+1]$ with $x$
8 **return** $i + 1$

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 0$ og $j = 1$
- Vi sammenligner $A[j] = 2$ med pivot-elementet og ser, at det er mindre, så vi inkrementerer $i$ og bytter $A[i]$ med $A[j]$ (men da $i = j = 1$ sker der ingenting lige nu). Når loopet fortsætter, inkrementeres $j$

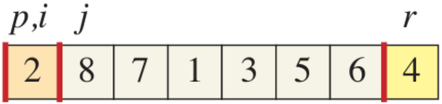



```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 1$ og $j = 2$
- Vi sammenligner $A[j] = 8$ med 4 og ser, at det er større, så vi gør intet andet end at inkrementere $j$

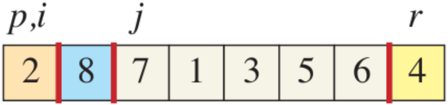



```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 1$ og $j = 3$
- Igen, vi ser $A[j] = 7$ er større end 4 og inkrementerer blot $j$

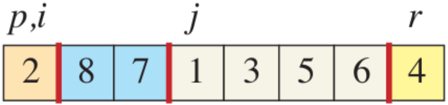



```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 1$ og $j = 4$
- Nu ser vi $A[j] = 1$ som er mindre end vores pivot-element, så vi inkrementerer $i$ og bytter plads på $A[i]$ og $A[j]$ inden $j$ inkrementeres



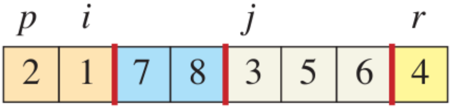

Partition($A, p, r$)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 2$ og $j = 5$
- $A[j] = 3$ er mindre end 4, så vi øger $i$ med 1 og bytter 7 og 3 (altså $A[3]$ og $A[5]$)

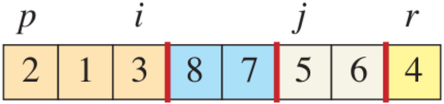



```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 3$ og $j = 6$
- For $j = 6$ er $A[j] = 5$, hvilket er større end 4, så loopet kan fortsætte

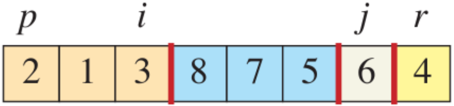



```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 3$ og $j = 7$
- . . . og igen, 6 er større end 4, så vi gør ikke noget




Partition($A, p, r$)

```
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 3$ og $j = 8$
- Til sidst bytter vi $A[i+1] = 8$ med vores pivot-element, og dermed er alt i $A[p:i]$ mindre end (eller lig med) 4, mens alt i $A[i+2:r]$ er større end 4

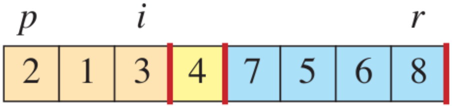
| $p$ | | $i$ | | | | | $r$ |
|---|---|---|---|---|---|---|---|
| 2 | 1 | 3 | 8 | 7 | 5 | 6 | 4 |

```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Quicksort

## Pseudo-kode del 2

- Vælg pivot-elementet $x$ til at være det sidste i sekvensen, $A[r] = 4$
- Vi har $i = 3$ og $j = 8$
- Bum! Nu kan vi returnere indexet på pivot-elementet og gentage proceduren rekursivt for de to sub-arrays




```
Partition(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
        if A[j] ≤ x
                i = i + 1
                exchange A[i] with A[j]
exhange A[i + 1] with x
return i + 1
```

# Outline

# Exercises!

## Yay!

# Outline

# Analyse af rekursioner

## 3 metoder

Vi har set et par eksempler på divide-and-conquer-algoritmer, og hvordan de bruger rekursion til at løse deres opgave. Rekursioner kan godt være tricky at analysere og forstå, hvorfor der findes en række metoder til at gå til dem mere eller mindre formelt:

# Analyse af rekursioner

## 3 metoder

Vi har set et par eksempler på divide-and-conquer-algoritmer, og hvordan de bruger rekursion til at løse deres opgave. Rekursioner kan godt være tricky at analysere og forstå, hvorfor der findes en række metoder til at gå til dem mere eller mindre formelt:

**The substitution method** I denne starter vi med et kvalificeret gæt, og så bruger vi matematisk induktion til at vise, at vi har ret. Kan være den nemmeste metode, men kræver lidt erfaring.

# Analyse af rekursioner

## 3 metoder

Vi har set et par eksempler på divide-and-conquer-algoritmer, og hvordan de bruger rekursion til at løse deres opgave. Rekursioner kan godt være tricky at analysere og forstå, hvorfor der findes en række metoder til at gå til dem mere eller mindre formelt:

**The substitution method** I denne starter vi med et kvalificeret gæt, og så bruger vi matematisk induktion til at vise, at vi har ret. Kan være den nemmeste metode, men kræver lidt erfaring.

**The recursion-tree method** Vi kan tegne det træ, der fremkommer af rekursionen og bruge det visuelle til at understøtte vores forståelse af, hvad der sker. Altid en god øvelse!

# Analyse af rekursioner

## 3 metoder

Vi har set et par eksempler på divide-and-conquer-algoritmer, og hvordan de bruger rekursion til at løse deres opgave. Rekursioner kan godt være tricky at analysere og forstå, hvorfor der findes en række metoder til at gå til dem mere eller mindre formelt:

**The substitution method** I denne starter vi med et kvalificeret gæt, og så bruger vi matematisk induktion til at vise, at vi har ret. Kan være den nemmeste metode, men kræver lidt erfaring.

**The recursion-tree method** Vi kan tegne det træ, der fremkommer af rekursionen og bruge det visuelle til at understøtte vores forståelse af, hvad der sker. Altid en god øvelse!

**The Master Method** For rekursioner, der kan skrives op på formen

$$T(n) = aT(n/b) + f(n)$$

er dette den letteste og sikreste måde at garantere kompleksiteten af algoritmen. Og det er den metode, vi kigger nærmere ind i nu!

# Master method

## Introduktion

The master method giver os et værktøj, der kan løse rekursioner af formen

$$T(n) = aT(n/b) + f(n)$$

hvilket ofte er den form, som divide-and-conquer giver.

- $a$ og $b$ er konstanter, og vi kræver, at $a \geq 1$, $b > 1$
- Vi kalder $f(n)$ the driving function, og kræver blot at den er en asymptotisk positiv funktion (dvs. ikke-negativ for alle tilpas store værdier af $n$)
- Med det kan vi benytte master teoremet!

# Master theorem

## Oh boy!

**Theorem (Master theorem)**

*Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by*

$$T(n) = aT(n/b) + f(n)$$

*where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:*

# Master theorem

## Oh boy!

**Theorem (Master theorem)**

*Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by*

$$T(n) = aT(n/b) + f(n)$$

*where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:*

1. *If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$*

# Master theorem

## Oh boy!

### Theorem (Master theorem)

*Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by*

$$T(n) = aT(n/b) + f(n)$$

*where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:*

1. *If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$*
2. *If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$ then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$*

# Master theorem

## Oh boy!

### Theorem (Master theorem)

*Let* $a > 0$ *and* $b > 1$ *be constants, and let* $f(n)$ *be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence* $T(n)$ *on* $n \in \mathbb{N}$ *by*

$$T(n) = aT(n/b) + f(n)$$

*where* $aT(n/b)$ *actually means* $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ *for some constants* $a' \geq 0$ *and* $a'' \geq 0$ *satisfying* $a = a' + a''$. *Then the asymptotic behavior of* $T(n)$ *can be characterized as follows:*

1. *If there exists a constant* $\epsilon > 0$ *such that* $f(n) = O(n^{\log_b a - \epsilon})$, *then* $T(n) = \Theta(n^{\log_b a})$
2. *If there exists a constant* $k \geq 0$ *such that* $f(n) = \Theta(n^{\log_b a} \lg^k n)$ *then* $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$
3. *If there exists a constant* $\epsilon > 0$ *such that* $f(n) = \Omega(n^{\log_b a + e})$ *and if* $f(n)$ *aditionally satisfies the* regularity condition $af(n/b) \leq cf(n)$ *for some constant* $c < 1$ *and all sufficiently large* $n$, *then* $T(n) = \Theta(f(n))$

# Master theorem

## Lets break it down

### Theorem (Master theorem (simplified))

*For a recurrence of the form* $T(n) = aT(n/b) + f(n)$*, the asymptotic behavior of* $T(n)$ *can be characterized as follows:*

$f(n)$ kaldes en driving function og $n^{\log_b a}$ kaldes en watershed function. Vi sammenligner disse på følgende måder:

# Master theorem

## Lets break it down

### Theorem (Master theorem (simplified))

*For a recurrence of the form $T(n) = aT(n/b) + f(n)$, the asymptotic behavior of $T(n)$ can be characterized as follows:*

1. *If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$*

$f(n)$ kaldes en driving function og $n^{\log_b a}$ kaldes en watershed function. Vi sammenligner disse på følgende måder:

Case 1 $f(n)$ er asymptotisk mindre (jf. $O$) end $n^{\log_b a}$ med en faktor af $n^{\epsilon}$ (eftersom $n^{\log_b a - \epsilon} = \frac{n^{\log_b a}}{n^{\epsilon}}$) — i så fald er rekursionen $T(n) = \Theta(n^{\log_b a})$

# Master theorem

## Lets break it down



### Theorem (Master theorem (simplified))

*For a recurrence of the form $T(n) = aT(n/b) + f(n)$, the asymptotic behavior of $T(n)$ can be characterized as follows:*

1. *If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$*
2. *If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$ then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$*

$f(n)$ kaldes en driving function og $n^{\log_b a}$ kaldes en watershed function. Vi sammenligner disse på følgende måder:

**Case 2** $f(n)$ er asymptotisk ækvivalent (jf. $\Theta$) med $n^{\log_b a} \lg^k n$ for $k \geq 0$ — i så fald er rekursionen $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$

- Generelt kan $k = 0$ antages, hvormed udtrykket bliver noget simplere!

# Master theorem

## Lets break it down

### Theorem (Master theorem (simplified))

*For a recurrence of the form $T(n) = aT(n/b) + f(n)$, the asymptotic behavior of $T(n)$ can be characterized as follows:*

1. *If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$*
2. *If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$ then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$*
3. *If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a+\epsilon})$ and if $f(n)$ aditionally satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$*

$f(n)$ kaldes en driving function og $n^{\log_b a}$ kaldes en watershed function. Vi sammenligner disse på følgende måder:

Case 3 $f(n)$ er asymptotisk større (jf. $\Omega$) end $n^{\log_b a}$ med en faktor $n^\epsilon$ (igen, $n^{\log_b a+\epsilon} = n^{\log_b a} * n^\epsilon$) og lidt ekstra holder — i så fald er rekursionen $T(n) = \Theta(f(n))$

- Regularitetsbetingelsen holder for det meste, og I behøver ikke hænge jer i den

# Master theorem

## Hvordan bruges det?

Følg denne opskrift:

1. Identificer om rekursionen har formen $T(n) = aT(n/b) + f(n)$
2. Gør udtrykket $n^{\log_b a}$ simplere ved at indsætte $a$ og $b$
3. Undersøg hvilket af de 3 cases, der holder, altså
   - $f(n) = O(n^{\log_b a - \epsilon}) = O(n^{\log_b a}/n^{\epsilon})$, hvor $\epsilon > 0$
   - $f(n) = \Theta(n^{\log_b a} \lg^k n)$, for $k >= 0$
   - $f(n) = \Omega(n^{\log_b a + \epsilon}) = \Omega(n^{\log_b a} * n^{\epsilon})$, hvor $\epsilon > 0$
4. Konklusionen kan stadig se lidt kompliceret ud, så reducer yderligere

# Master theorem

## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = \qquad b = \qquad f(n) =$$

$$n^{\log_b a} =$$

# Master theorem

## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = \qquad f(n) =$$

$$n^{\log_b a} =$$

# Master theorem

## Eksempel



Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = 2 \quad f(n) =$$
$$n^{\log_b a} =$$

# Master theorem
## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$$

$$n^{\log_b a} =$$

# Master theorem

## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$$
$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Nu tjekker vi så, om vi kan få det til at passe på nogle af teoremets cases:

# Master theorem

## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$$
$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Nu tjekker vi så, om vi kan få det til at passe på nogle af teoremets cases:

- Er $f(n) = O(n^{1-\epsilon})$ for nogen $\epsilon > 0$?

# Master theorem

## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$$
$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Nu tjekker vi så, om vi kan få det til at passe på nogle af teoremets cases:

- Er $f(n) = O(n^{1-\epsilon})$ for nogen $\epsilon > 0$?
- Er $f(n) = \Theta(n^1 \lg^k n)$ for nogen $k >= 0$?

# Master theorem

## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$$
$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Nu tjekker vi så, om vi kan få det til at passe på nogle af teoremets cases:

- Er $f(n) = O(n^{1-\epsilon})$ for nogen $\epsilon > 0$?
- Er $f(n) = \Theta(n^1 \lg^k n)$ for nogen $k >= 0$?
  - Ja! For $k = 0$ har vi $\Theta(n^1 \lg^0 n) = \Theta(n) = f(n)$.

# Master theorem

## Eksempel

Vi identificerede tidligere, at vi kunne skrive Merge-Sorts køretid op som rekursionen $T(n) = 2T(n/2) + \Theta(n)$.

$$a = 2 \quad b = 2 \quad f(n) = \Theta(n)$$
$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Nu tjekker vi så, om vi kan få det til at passe på nogle af teoremets cases:

- Er $f(n) = O(n^{1-\epsilon})$ for nogen $\epsilon > 0$?
- Er $f(n) = \Theta(n^1 \lg^k n)$ for nogen $k >= 0$?
  - Ja! For $k = 0$ har vi $\Theta(n^1 \lg^0 n) = \Theta(n) = f(n)$.
  - Dette er case 2 i teoremet, som fortæller os at $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n$, hvilket for $a = 2$, $b = 2$ og $k = 0$ giver at $T(n) = \Theta(n \lg n)$

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a =$ , $b =$ og $f(n) =$

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) =$

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$

# Master theorem

## Endnu et eksempel



Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$
- Vi indsætter og forenkler: $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$ (ish)

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$
- Vi indsætter og forenkler: $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$ (ish)

Og så tjekker vi cases:

1. Er $f(n) = O(n^{0.792-\epsilon})$ for nogen $\epsilon > 0$?

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$
- Vi indsætter og forenkler: $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$ (ish)

Og så tjekker vi cases:

1. Er $f(n) = O(n^{0.792-\epsilon})$ for nogen $\epsilon > 0$?
   - Nej, for $n^{0.792} < n$, hvilket stadig må gælde, hvis vi fratrækker en positiv konstant i eksponenten, og da $f(n) = n \lg n \geq n$ kan $n^{0.792-\epsilon}$ altså ikke være et upper bound.

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$
- Vi indsætter og forenkler: $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$ (ish)

Og så tjekker vi cases:

1. Er $f(n) = O(n^{0.792-\epsilon})$ for nogen $\epsilon > 0$?
   - Nej, for $n^{0.792} < n$, hvilket stadig må gælde, hvis vi fratrækker en positiv konstant i eksponenten, og da $f(n) = n \lg n \geq n$ kan $n^{0.792-\epsilon}$ altså ikke være et upper bound.
2. Er $f(n) = \Theta(n^{0.792} \lg^k n)$ for nogen $k >= 0$?

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$
- Vi indsætter og forenkler: $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$ (ish)

Og så tjekker vi cases:

1. Er $f(n) = O(n^{0.792-\epsilon})$ for nogen $\epsilon > 0$?
   - Nej, for $n^{0.792} < n$, hvilket stadig må gælde, hvis vi fratrækker en positiv konstant i eksponenten, og da $f(n) = n \lg n \geq n$ kan $n^{0.792-\epsilon}$ altså ikke være et upper bound.
2. Er $f(n) = \Theta(n^{0.792} \lg^k n)$ for nogen $k >= 0$?
   - Nej, for hvis $k <= 1$ vokser $n^{0.792} \lg^k n$ for langsomt, og hvis $k > 1$ vokser det for hurtigt

# Master theorem

## Endnu et eksempel

Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$
- Vi indsætter og forenkler: $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$ (ish)

Og så tjekker vi cases:

1. Er $f(n) = O(n^{0.792-\epsilon})$ for nogen $\epsilon > 0$?
   - Nej, for $n^{0.792} < n$, hvilket stadig må gælde, hvis vi fratrækker en positiv konstant i eksponenten, og da $f(n) = n \lg n \geq n$ kan $n^{0.792-\epsilon}$ altså ikke være et upper bound.
2. Er $f(n) = \Theta(n^{0.792} \lg^k n)$ for nogen $k >= 0$?
   - Nej, for hvis $k <= 1$ vokser $n^{0.792} \lg^k n$ for langsomt, og hvis $k > 1$ vokser det for hurtigt
3. Er $f(n) = \Omega(n^{0.792+\epsilon})$ for nogen $\epsilon > 0$?

# Master theorem

## Endnu et eksempel



Vi prøver endnu et eksempel

- Vi har $T(n) = 3T(n/4) + n \lg n$
- Dermed har vi $a = 3$, $b = 4$ og $f(n) = n \lg n$
- Vi indsætter og forenkler: $n^{\log_b a} = n^{\log_4 3} = n^{0.792}$ (ish)

Og så tjekker vi cases:

1. Er $f(n) = O(n^{0.792-\epsilon})$ for nogen $\epsilon > 0$?
   - Nej, for $n^{0.792} < n$, hvilket stadig må gælde, hvis vi fratrækker en positiv konstant i eksponenten, og da $f(n) = n \lg n \geq n$ kan $n^{0.792-\epsilon}$ altså ikke være et upper bound.
2. Er $f(n) = \Theta(n^{0.792} \lg^k n)$ for nogen $k >= 0$?
   - Nej, for hvis $k <= 1$ vokser $n^{0.792} \lg^k n$ for langsomt, og hvis $k > 1$ vokser det for hurtigt
3. Er $f(n) = \Omega(n^{0.792+\epsilon})$ for nogen $\epsilon > 0$?
   - Ja! $n^x$ er et lower bound for $n \lg n$, så længe $x$ er mindre end 1. Dvs., at denne case holder for alle $\epsilon < 0.2$ (ish)
   - Case 3 giver os at $T(n) = \Theta(f(n)) = \Theta(n \lg n)$

# 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!