Sortering: Bubble, Selection, Insert, Merge

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Høsten 2023

Oversikt

Oversikt

- Denne uken handler om *sortering*

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
- Om noen uker skal vi lære et par nye begreper, samt studere

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
- Om noen uker skal vi lære et par nye begreper, samt studere
  - Heapsort

Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
- Om noen uker skal vi lære et par nye begreper, samt studere
  - Heapsort
  - Quicksort

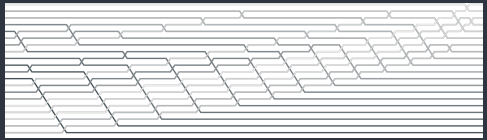
Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
- Om noen uker skal vi lære et par nye begreper, samt studere
  - Heapsort
  - Quicksort
  - Bucket sort

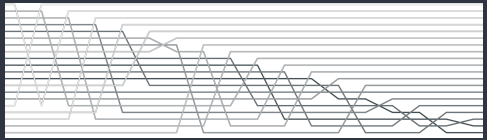
Oversikt

- Denne uken handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Merge sort
- Om noen uker skal vi lære et par nye begreper, samt studere
  - Heapsort
  - Quicksort
  - Bucket sort
  - Radix sort

Illustrasjoner



Bubble sort



Selection sort



Insertion sort



$\mathcal{O}$-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input

$\mathcal{O}$-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input

$\mathcal{O}$-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er

$\mathcal{O}$-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker

$\mathcal{O}$-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant

$\mathcal{O}$-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter

$\mathcal{O}$-notasjon handler om vekst

- En algoritme tar alltid utgangspunkt i null eller flere input
  - og produserer et output som er relatert til input
- Spørsmålet vi er interessert i er
  - hva er et øvre estimat for hvor lang tid algoritmen bruker
  - der vi kan skalere tiden med en konstant
- Konstanten kan fange opp mange ulike momenter
  - for eksempel hastigheten på maskinen og antall instruksjoner

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$

Hvordan $\mathcal{O}$ er definert

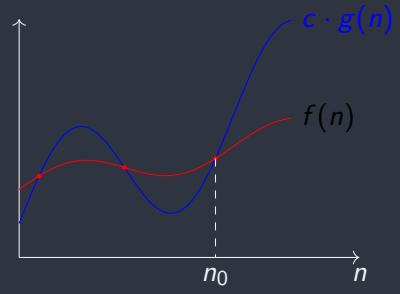
- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$

Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$

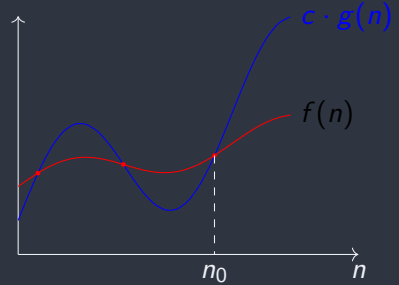
Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$



Hvordan $\mathcal{O}$ er definert

- La $f(n)$ være kjøretiden for input av størrelse n og la g være en funksjon fra heltall til positive tall
- $f(n)$ er i $\mathcal{O}(g(n))$ hvis det finnes en positiv konstant c, slik at $f(n) \leq c \cdot g(n)$ for alle tilstrekkelig store verdier av n
  - Denne sier at $f(n)$ ikke vokser noe raskere enn $g(n)$
  - $g(n)$ er en *øvre skranke* for $f(n)$
  - $f(n)$ er begrenset av $g(n)$



- $f(n) = 3n^2 + 5n + 2$ er i $\mathcal{O}(n^2)$
  - Fordi $4 \cdot n^2$ er størst så lenge $n > 6$

Klassikerene

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                    // O(1)
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                  // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← ⌊i/2⌋                                 // O(log(n))
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                  // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← ⌊i/2⌋                                 // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                // O(n)
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                  // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← ⌊i/2⌋                                 // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                // O(n)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)                                     // O(n · log(n))
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                  // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← ⌊i/2⌋                                 // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                // O(n)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)                                     // O(n · log(n))
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)                             // O(n²)
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                   // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← ⌊i/2⌋                                  // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                 // O(n)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)                                      // O(n · log(n))
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)                              // O(n^2)
```

```
1 Procedure Polynomial(n)
2 |  for i_1 ← 0 to n − 1 do
3 |  |  for i_2 ← 0 to n − 1 do
  |  |  |  ⋮
4 |  |  |  ⋮
5 |  |  |  for i_k ← 0 to n − 1 do
6 |  |  |  |  Constant(i)                           // O(n^k)
```

Klassikerene

```
1 Procedure Constant(n)
2 |  return n · 3                                   // O(1)
```

```
1 Procedure Log(n)
2 |  i ← n
3 |  while i > 0 do
4 |  |  Constant(i)
5 |  |  i ← ⌊i/2⌋                                  // O(log(n))
```

```
1 Procedure Linear(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Constant(i)                                 // O(n)
```

```
1 Procedure Linearithmic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  Log(n)                                      // O(n · log(n))
```

```
1 Procedure Quadratic(n)
2 |  for i ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  Constant(i)                              // O(n^2)
```

```
1 Procedure Polynomial(n)
2 |  for i_1 ← 0 to n − 1 do
3 |  |  for i_2 ← 0 to n − 1 do
  |  |  |  ⋮
4 |  |  |  for i_k ← 0 to n − 1 do
5 |  |  |  |  Constant(i)                           // O(n^k)
```

```
1 Procedure Exponential(n)
2 |  if n = 0 then
3 |  |  return 1
4 |  a ← Exponential(n − 1)
5 |  b ← Exponential(n − 1)
6 |  return a + b                                   // O(2^n)
```

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$$\mathcal{O}(1) \qquad \text{konstant tid}$$

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$$\mathcal{O}(1) \qquad \text{konstant tid}$$

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n \log(n))$	lineæritmisk tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n \log(n))$	lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input

Hvorfor $\mathcal{O}$-notasjon er enkelt

- $\mathcal{O}$-notasjon lar oss *ignorere* konstantfaktorer
- $\mathcal{O}$-notasjon lar oss kun fokusere på det største leddet
- Dette gjør at vi slipper unna utrolig mye utregning
- I konteksten av IN2010 må vi for en inputstørrelse n lære å skille mellom

$\mathcal{O}(1)$	konstant tid
$\mathcal{O}(\log(n))$	logaritmisk tid
$\mathcal{O}(n)$	lineær tid
$\mathcal{O}(n\log(n))$	lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

- Og gjøre dette for hvert input
  - (i tilfellet hvor vi har en algoritme som er avhengig av mer enn ett input)

Sortering

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at
  - a kommer før b hvis $a \preceq b$

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at
  - a kommer før b hvis $a \preceq b$
  - alle elementer fra datastrukturen er bevart i output

Definisjon av problemet

- Å *sortere* går ut på å *ordne* elementer fra en datastruktur slik at
  - a kommer før b hvis $a \preceq b$
  - alle elementer fra datastrukturen er bevart i output
- Vi kommer til å fokusere på sortering av *arrayer*

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk
  - Dele inn, eller grupere, etter sort

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk
  - Dele inn, eller grupere, etter sort
- Det informatikere mener med sortering er egentlig å *ordne* elementer

Litt begrepsforvirring

- Begrepet sortering har en annen betydning utenfor informatikk
  - Dele inn, eller grupere, etter sort
- Det informatikere mener med sortering er egentlig å *ordne* elementer
  - Men å «ordne» er et overbelastet begrep, så man bruker heller ordet «sortere»

Problemer som løses ved å sortering

1. Samle ting som hører sammen

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
    - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
    - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
    - Dette kalles også partisjonering

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
    - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
    - Dette kalles også partisjonering
    - Kanskje det er her begrepet har fått sin betydning i informatikk fra

Problemer som løses ved å sortering

1. Samle ting som hører sammen
    - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
    - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
    - Dette kalles også partisjonering
    - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche

Problemer som løses ved å sortering

1. Samle ting som hører sammen
   - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
   - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
   - Dette kalles også partisjonering
   - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche
   - Gitt to eller flere sekvensielle strukturer, kan vi finne elementer som matcher ved å løpe over kun én gang

Problemer som løses ved å sortering

1. Samle ting som hører sammen
   - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
   - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
   - Dette kalles også partisjonering
   - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche
   - Gitt to eller flere sekvensielle strukturer, kan vi finne elementer som matcher ved å løpe over kun én gang
3. Søk

Problemer som løses ved å sortering

1. Samle ting som hører sammen
   - Hvis du har ting som faller i ulike kategorier kan vi ordne kategoriene
   - Sorterer vi etter kategoriene, så samler vi alt som faller i samme kategori
   - Dette kalles også partisjonering
   - Kanskje det er her begrepet har fått sin betydning i informatikk fra
2. Matche
   - Gitt to eller flere sekvensielle strukturer, kan vi finne elementer som matcher ved å løpe over kun én gang
3. Søk
   - Vi har lært hvordan å søke i *sorterte* arrayer er dramatisk mye raskere enn usorterte

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use. The real truth probably involves all three of these possibilities, but in any event we can see that sorting is worthy of serious study, as a practical matter.

Hvorfor sortere

Computer manufacturers of the 1960s estimated that more than 25 percent of the running time on their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms have been in common use. The real truth probably involves all three of these possibilities, but in any event we can see that sorting is worthy of serious study, as a practical matter.

– Donald Knuth, The Art of Computer Programming



Bubble sort

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi
  1. løpe over hvert par av etterfølgende elementer i arrayet

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi
  1. løpe over hvert par av etterfølgende elementer i arrayet
  2. bytte om rekkefølgen et par dersom det ikke er ordnet

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og bare fortsett sånn helt til det ikke er noen flere feil å rette opp!
- Litt mer presist skal vi
  1. løpe over hvert par av etterfølgende elementer i arrayet
  2. bytte om rekkefølgen et par dersom det ikke er ordnet
  3. gå til 1. dersom det forekom minst et bytte

Bubble sort – Implementasjon

ALGORITHM: BUBBLE SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
```

Bubble sort – Implementasjon

ALGORITHM: BUBBLE SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
```

Bubble sort – Implementasjon

ALGORITHM: BUBBLE SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
```

Bubble sort – Implementasjon

ALGORITHM: BUBBLE SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Implementasjon

Algorithm: Bubble sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

- Merk at denne varianten ikke er optimalisert

Bubble sort – Implementasjon

ALGORITHM: BUBBLE SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

- Merk at denne varianten ikke er optimalisert
  - Man kan bryte ut av den ytre loopen dersom det ikke forekommer noen bytter i den indre loopen

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n - i - 2$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n - i - 2$
  - For hver iterasjon blir i større, så vi itererer over mindre

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n - i - 2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i = 0$) får vi $n - 1$ iterasjoner

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n - i - 2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i = 0$) får vi $n - 1$ iterasjoner
  - Men når $i = n - 2$ får vi ingen iterasjoner!

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n - 2$, som svarer til $n - 1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n - i - 2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i = 0$) får vi $n - 1$ iterasjoner
  - Men når $i = n - 2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n - 1 + n - 2 + \cdots + 1$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi

$$n-1+n-2+\cdots+1=1+2+\cdots+n-1$$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi

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

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1 = 1+2+\cdots+n-1 = \frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2}) = \mathcal{O}(\frac{n^2-n}{2})$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})=\mathcal{O}(n^2-n)$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})=\mathcal{O}(n^2-n)=\mathcal{O}(n^2)$

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1=1+2+\cdots+n-1=\frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2})=\mathcal{O}(\frac{n^2-n}{2})=\mathcal{O}(n^2-n)=\mathcal{O}(n^2)$
- Merk at optimaliseringen nevnt på forrige slide *ikke* påvirker *verste* tilfellet

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

Bubble sort – Kjøretidsanalyse

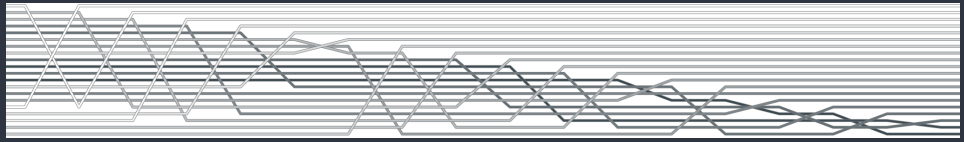
- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det totale antall iterasjoner får vi $n-1+n-2+\cdots+1 = 1+2+\cdots+n-1 = \frac{n(n-1)}{2}$
- Og $\mathcal{O}(\frac{n(n-1)}{2}) = \mathcal{O}(\frac{n^2-n}{2}) = \mathcal{O}(n^2-n) = \mathcal{O}(n^2)$
- Merk at optimaliseringen nevnt på forrige slide *ikke* påvirker *verste* tilfellet
- Altså har bubble sort *kvadratisk* kjøretidskompleksitet

Algorithm: Bubble sort

```
1 Procedure BubbleSort(A)
2     for i ← 0 to n − 2 do
3         for j ← 0 to n − i − 2 do
4             if A[j] > A[j + 1] then
5                 A[j], A[j + 1] ← A[j + 1], A[j]
```

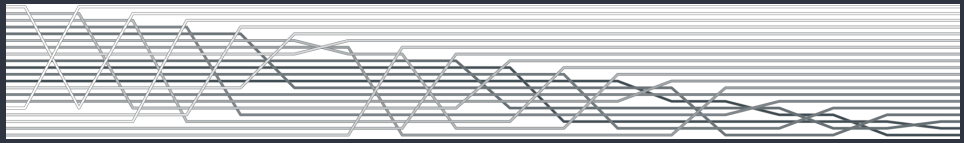
Selection sort

Selection sort – Idé



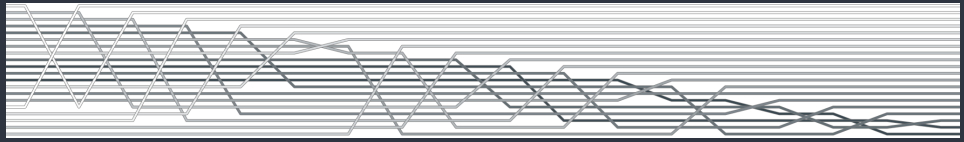
- Idéen bak selection sort er å finne det minste i resten og plassere det først

Selection sort – Idé



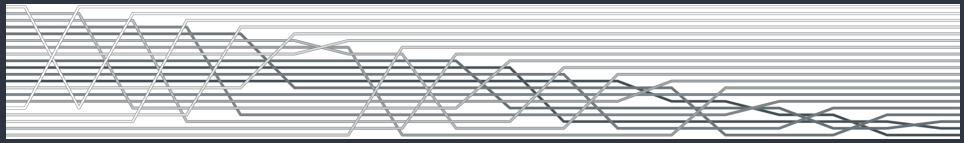
- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi

Selection sort – Idé



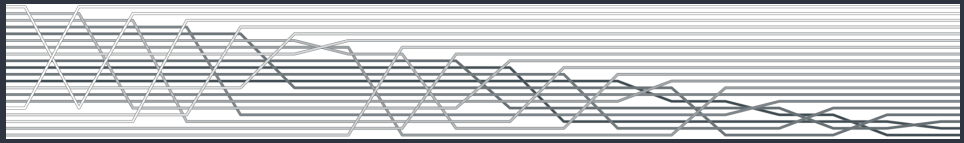
- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0
  2. finn hvor det minste elementet fra *i* og utover ligger

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0
  2. finn hvor det minste elementet fra *i* og utover ligger
  3. bytt ut elementet på plass *i* med det minste (hvis nødvendig)

Selection sort – Idé



- Idéen bak selection sort er å finne det minste i resten og plassere det først
- Litt mer presist skal vi
  1. la *i* være 0
  2. finn hvor det minste elementet fra *i* og utover ligger
  3. bytt ut elementet på plass *i* med det minste (hvis nødvendig)
  4. øk *i* og gå til 2. frem til *i* når størrelsen av arrayet

Selection sort – Implementasjon

Algorithm: Selection sort

Input: Et array A med *n* elementer
Output: Et *sortert* array med de samme *n* elementene

```
1 Procedure SelectionSort(A)
```

Selection sort – Implementasjon

ALGORITHM: SELECTION SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
```

Selection sort – Implementasjon

ALGORITHM: SELECTION SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
```

Selection sort – Implementasjon

ALGORITHM: SELECTION SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
```

Selection sort – Implementasjon

ALGORITHM: SELECTION SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Implementasjon

ALGORITHM: SELECTION SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

- Merk at vi *ikke* kan bryte ut av den ytre loopen tidlig slik vi kunne med bubble sort

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!
- Å bytte om to verdier i et array er forholdsvis dyrt

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Selection sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av loopen tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!
- Å bytte om to verdier i et array er forholdsvis dyrt
  - Så selection sort er som regel raskere enn bubble sort

Algorithm: Selection sort

```
1 Procedure SelectionSort(A)
2     for i ← 0 to n − 1 do
3         k ← i
4         for j ← i + 1 to n − 1 do
5             if A[j] < A[k] then
6                 k ← j
7         if i ≠ k then
8             A[i], A[k] ← A[k], A[i]
```

Insertion sort

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi
  1. la *i* være 1

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi
  1. la *i* være 1
  2. dra det *i*-te elementet mot venstre som ved sortert innsetting

Insertion sort – Idé



- Idéen bak insertion sort er å plassere alle elementene sortert inn i en liste
- Dette er antageligvis slik du sorterer kort
- Vi lar alt til venstre for en gitt posisjon *i* være sortert
- Litt mer presist skal vi
  1. la *i* være 1
  2. dra det *i*-te elementet mot venstre som ved sortert innsetting
  3. øk *i* og gå til 2. frem til *i* når størrelsen av arrayet

Insertion sort – Implementasjon

Algorithm: Insertion sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure InsertionSort(A)
```

Insertion sort – Implementasjon

ALGORITHM: INSERTION SORT

Input: Et array A med n elementer

Output: Et *sortert* array med de samme n elementene

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
```

Insertion sort – Implementasjon

Algorithm: Insertion sort

Input: Et array A med n elementer

Output: Et *sortert* array med de samme n elementene

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
```

Insertion sort – Implementasjon

Algorithm: Insertion sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre loopen

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre loopen
- Den er spesielt rask på «nesten sorterte» arrayer

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

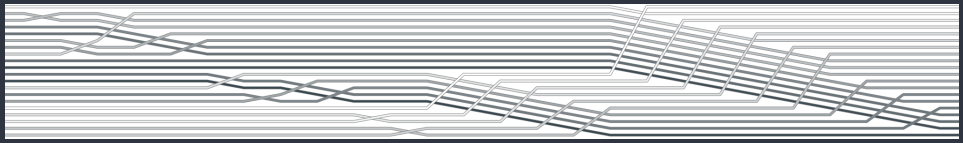
- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre loopen
- Den er spesielt rask på «nesten sorterte» arrayer
- Dette gjør at den er blant de raskeste algoritmene for *små* arrayer

Algorithm: Insertion sort

```
1 Procedure InsertionSort(A)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j-1] > A[j] do
5             A[j-1], A[j] ← A[j], A[j-1]
6             j ← j − 1
```

Merge sort

Merge sort – Idé



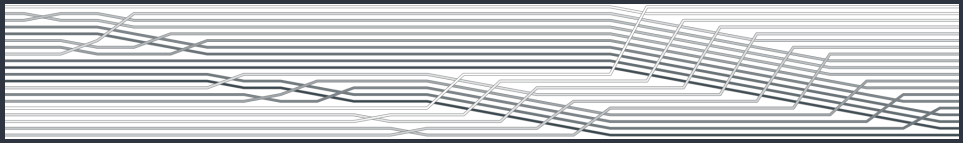
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler

Merge sort – Idé



- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene

Merge sort – Idé



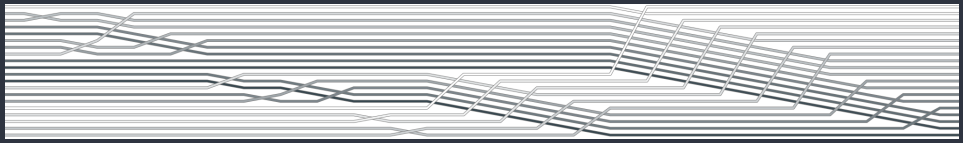
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen

Merge sort – Idé



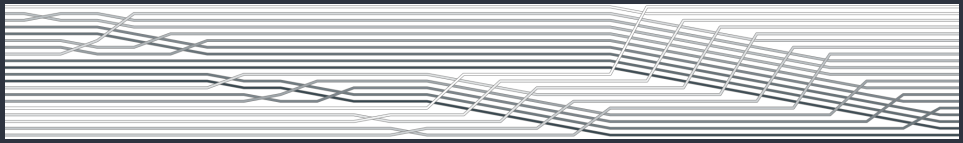
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:

Merge sort – Idé



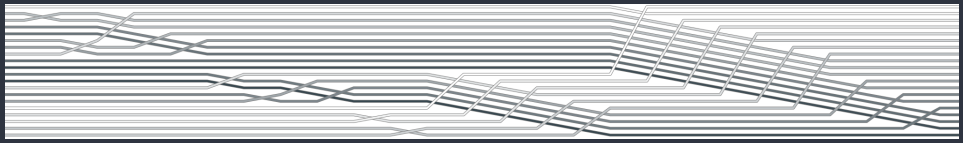
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet `A`

Merge sort – Idé



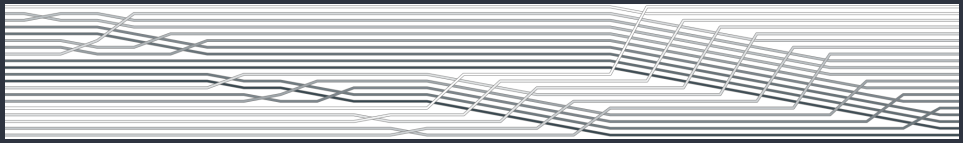
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`

Merge sort – Idé



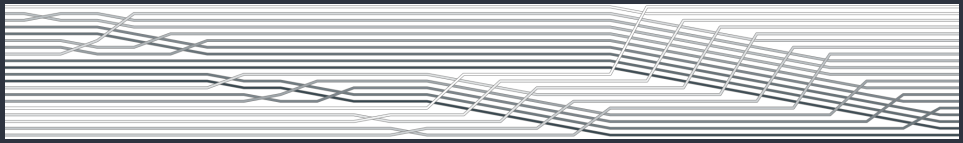
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`
  - la $i = \lfloor \frac{n}{2} \rfloor$

Merge sort – Idé



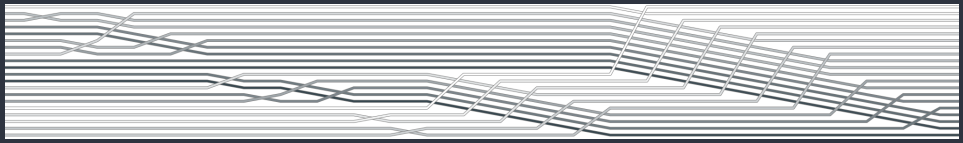
- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`
  - la $i = \lfloor \frac{n}{2} \rfloor$
  - splitt arrayet i to deler `A[0..i-1]` og `A[i..n-1]`

Merge sort – Idé



- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`
  - la $i = \lfloor \frac{n}{2} \rfloor$
  - splitt arrayet i to deler `A[0..i-1]` og `A[i..n-1]`
  - anvend merge sort rekursivt på `A[0..i-1]` og `A[i..n-1]`

Merge sort – Idé



- Idéen bak merge sort er å splitte arrayet i to ca. like store deler
  - sortere de to mindre arrayene
  - så flette (eller «merge») de to sorterte arrayene sammen
- Litt mer presist:
  - la n angi størrelsen på arrayet `A`
  - hvis $n \leq 1$, returner `A`
  - la $i = \lfloor \frac{n}{2} \rfloor$
  - splitt arrayet i to deler `A[0..i-1]` og `A[i..n-1]`
  - anvend merge sort rekursivt på `A[0..i-1]` og `A[i..n-1]`
  - flett sammen `A[0..i-1]` og `A[i..n-1]` sortert

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$

Output: Et sortert array A med elementene fra A_1 og A_2

1 **Procedure** Merge(A_1, A_2, A)

Merge sort – Merge

Algorithm: Sortert fletting av to arrayer

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$

Output: Et sortert array A med elementene fra A_1 og A_2

```
1 Procedure Merge(A_1, A_2, A)
2     i ← 0
3     j ← 0
```

Merge sort – Merge

ALGORITHM: SORTERT FLETTING AV TO ARRAYER

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$

Output: Et sortert array A med elementene fra A_1 og A_2

```
1 Procedure Merge(A_1, A_2, A)
2     i ← 0
3     j ← 0
4     while i < |A_1| and j < |A_2| do
```

Merge sort – Merge

ALGORITHM: SORTERT FLETTING AV TO ARRAYER

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$

Output: Et sortert array A med elementene fra A_1 og A_2

```
1 Procedure Merge(A_1, A_2, A)
2     i ← 0
3     j ← 0
4     while i < |A_1| and j < |A_2| do
5         if A_1[i] ≤ A_2[j] then
6             A[i + j] ← A_1[i]
7             i ← i + 1
8         else
```

Merge sort – Merge

ALGORITHM: SORTERT FLETTING AV TO ARRAYER

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$

Output: Et sortert array A med elementene fra A_1 og A_2

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4      while i < |A_1| and j < |A_2| do
5          if A_1[i] ≤ A_2[j] then
6              A[i + j] ← A_1[i]
7              i ← i + 1
8          else
9              A[i + j] ← A_2[j]
10             j ← j + 1
```

Merge sort – Merge

ALGORITHM: SORTERT FLETTING AV TO ARRAYER

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4      while i < |A_1| and j < |A_2| do
5          if A_1[i] ≤ A_2[j] then
6              A[i + j] ← A_1[i]
7              i ← i + 1
8          else
9              A[i + j] ← A_2[j]
10             j ← j + 1
11     while i < |A_1| do
12         A[i + j] ← A_1[i]
13         i ← i + 1
```

Merge sort – Merge

ALGORITHM: SORTERT FLETTING AV TO ARRAYER

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$
Output: Et sortert array A med elementene fra A_1 og A_2

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4      while i < |A_1| and j < |A_2| do
5          if A_1[i] ≤ A_2[j] then
6              A[i + j] ← A_1[i]
7              i ← i + 1
8          else
9              A[i + j] ← A_2[j]
10             j ← j + 1
11     while i < |A_1| do
12         A[i + j] ← A_1[i]
13         i ← i + 1
14     while j < |A_2| do
15         A[i + j] ← A_2[j]
16         j ← j + 1
17     return A
```

Merge sort – Implementasjon

ALGORITHM: MERGE SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
```

Merge sort – Implementasjon

Algorithm: Merge sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
```

Merge sort – Implementasjon

Algorithm: Merge sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
```

Merge sort – Implementasjon

Algorithm: Merge sort

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

Merge sort – Implementasjon

ALGORITHM: MERGE SORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A_1 ← MergeSort(A[0..i-1])
6     A_2 ← MergeSort(A[i..n-1])
7     return Merge(A_1, A_2, A)
```

- Her angir `A[0..i-1]` å lage *et nytt* array med elementene `A[0]`, `A[1]`, ..., `A[i-1]`

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A₁, A₂, A)
2      i ← 0
3      j ← 0
4
5      while i < |A₁| and j < |A₂| do
6          if A₁[i] ≤ A₂[j] then
7              A[i + j] ← A₁[i]
8              i ← i + 1
9          else
10             A[i + j] ← A₂[j]
11             j ← j + 1
12
13     while i < |A₁| do
14         A[i + j] ← A₁[i]
15         i ← i + 1
16
17     while j < |A₂| do
18         A[i + j] ← A₂[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A₁, A₂, A)
2      i ← 0
3      j ← 0
4
5      while i < |A₁| and j < |A₂| do
6          if A₁[i] ≤ A₂[j] then
7              A[i + j] ← A₁[i]
8              i ← i + 1
9          else
10             A[i + j] ← A₂[j]
11             j ← j + 1
12
13     while i < |A₁| do
14         A[i + j] ← A₁[i]
15         i ← i + 1
16
17     while j < |A₂| do
18         A[i + j] ← A₂[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker i eller j med én

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker i eller j med én
  - Og at vi terminerer når $i = |A_1|$ og $j = |A_2|$

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker i eller j med én
  - Og at vi terminerer når $i = |A_1|$ og $j = |A_2|$
- Videre er $|A_1| + |A_2| = |\mathsf{A}| = n$

Merge sort – Kjøretidsanalyse (`Merge`)

Algorithm: Sortert fletting av to arrayer

```
1  Procedure Merge(A_1, A_2, A)
2      i ← 0
3      j ← 0
4
5      while i < |A_1| and j < |A_2| do
6          if A_1[i] ≤ A_2[j] then
7              A[i + j] ← A_1[i]
8              i ← i + 1
9          else
10             A[i + j] ← A_2[j]
11             j ← j + 1
12
13     while i < |A_1| do
14         A[i + j] ← A_1[i]
15         i ← i + 1
16
17     while j < |A_2| do
18         A[i + j] ← A_2[j]
19         j ← j + 1
20
21     return A
```

- Vi analyserer hvor mange iterasjoner som gjøres totalt
- Merk at hver iterasjon øker i eller j med én
  - Og at vi terminerer når $i = |A_1|$ og $j = |A_2|$
- Videre er $|A_1| + |A_2| = |\mathtt{A}| = n$
- Da har vi at `Merge` er i $\mathcal{O}(n)$

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A₁ ← MergeSort(A[0..i-1])
6     A₂ ← MergeSort(A[i..n-1])
7     return Merge(A₁, A₂, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2   if n ≤ 1 then
3     return A
4   i ← ⌊n/2⌋
5   A₁ ← MergeSort(A[0..i-1])
6   A₂ ← MergeSort(A[i..n-1])
7   return Merge(A₁, A₂, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A₁ ← MergeSort(A[0..i-1])
6     A₂ ← MergeSort(A[i..n-1])
7     return Merge(A₁, A₂, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall
- Dybden på rekursjonen er $\mathcal{O}(\log(n))$

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A₁ ← MergeSort(A[0..i-1])
6     A₂ ← MergeSort(A[i..n-1])
7     return Merge(A₁, A₂, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall
- Dybden på rekursjonen er $\mathcal{O}(\log(n))$
  - fordi det er så mange ganger vi kan halvere input

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A₁ ← MergeSort(A[0..i-1])
6     A₂ ← MergeSort(A[i..n-1])
7     return Merge(A₁, A₂, A)
```

Merge sort – Kjøretidsanalyse (`MergeSort`)

- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $\mathcal{O}(n)$ operasjoner for hvert kall
- Dybden på rekursjonen er $\mathcal{O}(\log(n))$
  - fordi det er så mange ganger vi kan halvere input
- Da har vi at `MergeSort` er i $\mathcal{O}(n \log(n))$

Algorithm: Merge sort

```
1 Procedure MergeSort(A)
2     if n ≤ 1 then
3         return A
4     i ← ⌊n/2⌋
5     A₁ ← MergeSort(A[0..i-1])
6     A₂ ← MergeSort(A[i..n-1])
7     return Merge(A₁, A₂, A)
```