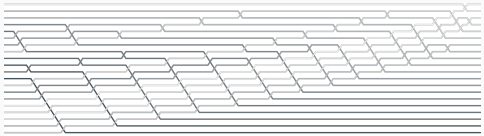
Sortering

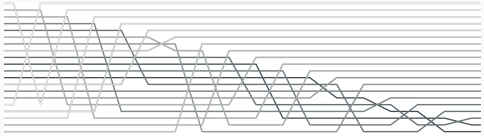
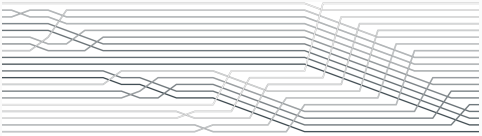
IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Sortering



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*

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

Stabilitet og in-place

Stabilitet

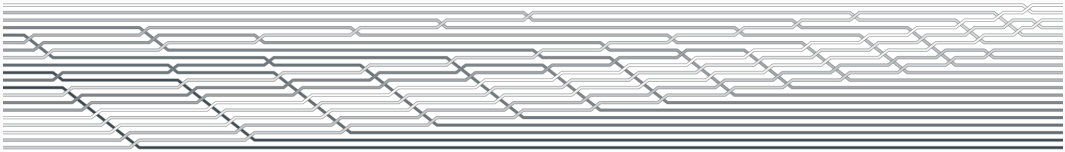
- Ofte sorterer vi på *nøkler*
  - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom
  - for alle elementer x, y med samme nøkkel k
  - hvis x forekom før y før sortering
  - så forekommer x før y etter sortering
- Om en sorteringsalgoritme er stabil kan være implementasjonsavhengig

In-place

- En algoritme er in-place dersom den ikke bruker ekstra datastrukturer
- Å mellomlagre resultater i en annen datastruktur er ikke in-place
- Av algoritmene vi skal se i dette kurset er de fleste algoritmene in-place
  - Merge sort er et eksempel på en algoritme som *ikke* in-place

Bubble sort

Bubble sort – Idé



- Idéen bak bubble sort løpe gjennom et array og «rette opp» feil
- … og 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)
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                 swap A[j] and A[j + 1]
```

- Merk at vi kunne brutt ut av den *ytre* løkken dersom det ikke forekommer noen bytter i den *indre* løkken

Bubble sort – Kjøretidsanalyse

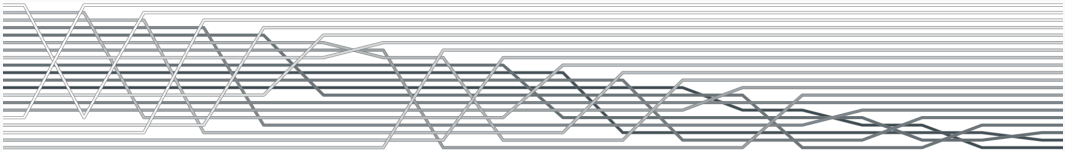
- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får kun én iterasjon
- Hvis vi teller det totale 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                 swap A[j] and A[j + 1]
```

Selection sort

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)
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             swap A[i] and A[k]
```

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

Selection sort – Kjøretidsanalyse

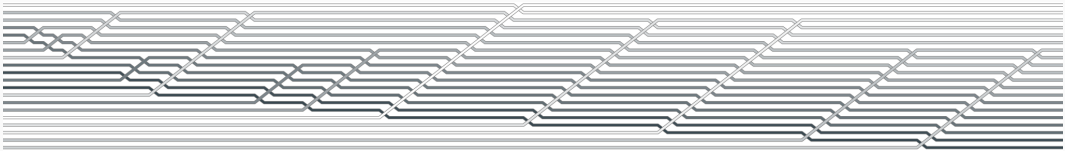
- Analysen her blir tilnærmet lik den for bubble sort
- Den ytre løkken kjører $\mathcal{O}(n)$ ganger
- Den indre løkken kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Selection sort kan ikke bryte ut av løkken tidlig
- Selection sort vil maksimalt gjøre $n - 1$ bytter!
- Selection sort egner seg spesielt godt hvis bytter er 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             swap A[i] and A[k]
```

Insertion sort

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)
2     for i ← 1 to n − 1 do
3         j ← i
4         while j > 0 and A[j − 1] > A[j] do
5             swap A[j − 1] and A[j]
6             j ← j − 1
```

Insertion sort – Kjøretidsanalyse

- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre løkken kjører $\mathcal{O}(n)$ ganger
- Den indre løkken kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre løkken
- 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             swap A[j − 1] and A[j]
6             j ← j − 1
```

Merge sort

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)
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)
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 $A[0], A[1], \ldots, A[i-1]$

Merge sort – Kjøretidsanalyse (`Merge`)

- 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 $|A_1| + |A_2| = |A| = n$
- Da har vi at `Merge` er i $\mathcal{O}(n)$

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
```

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 $\mathcal{O}(n \cdot \log(n))$

Algorithm: Merge sort

```
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)
```