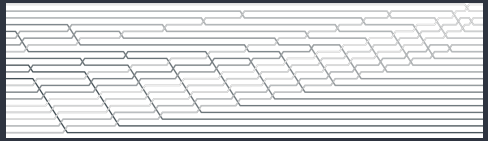
Sortering

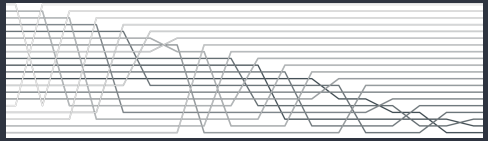
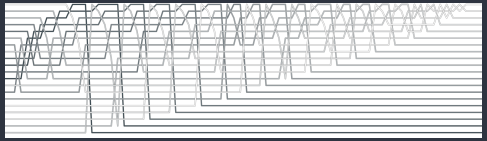
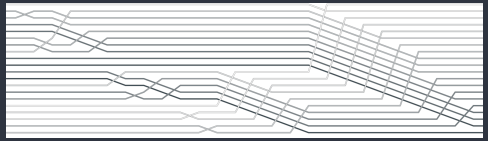
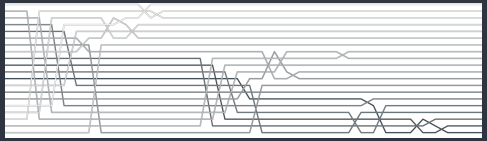
IN2010 – Algoritmer og datastrukturer

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
  - Men Merge sort er 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                 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

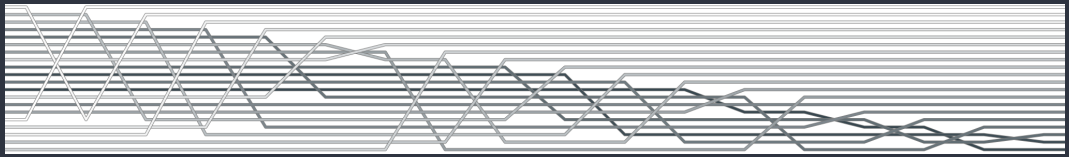
- Vi itererer fra 0 til $n-2$, som svarer til $n-1$ iterasjoner
- For hver iterasjon løper vi fra 0 til $n-i-2$
  - For hver iterasjon blir i større, så vi itererer over mindre
  - I verste tilfelle (når $i=0$) får vi $n-1$ iterasjoner
  - Men når $i=n-2$ får vi ingen iterasjoner!
- Hvis vi teller det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

Heapsort

Heapsort – Idé

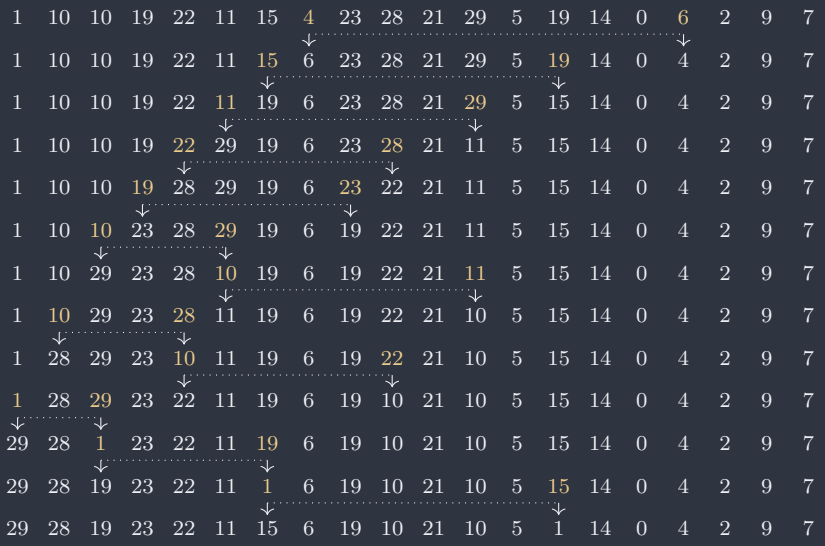


- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet
  3. pop fra max-heapen og plasser elementet på plass i
  4. senk i og gå til 3. frem til i blir 0

Heapsort – Bygge en max-heap

- Vi trenger å kunne gjøre et array om til en max-heap
- En max-heap er en heap der hver node er *større* enn begge barna
  - I motsetning til en min-heap der hver node er *mindre* enn begge barna
- En node svarer til en posisjon i arrayet
  - der roten ligger på plass 0,
  - venstre barn for noden på plass i ligger på plass $2i + 1$
  - og høyre barn for noden på plass i ligger på plass $2i + 2$
- `BuildMaxHeap` gjør et array om til en max-heap
  - Den starter i midten av arrayet og jobber seg mot venstre
  - Den bytter plass på foreldre- og barnenoder rekursivt der foreldrenoden er mindre enn (minst) en av barnenodene
  - Etter siste iterasjon vil det største elementet ligge i roten, og alle foreldrenoder er større (eller lik) begge barnenoder

Heapsort – Bygge en max-heap (eksempel)



Heapsort – Bygge en max-heap (implementasjon)

Algorithm: Hjelpeprosedyre for å bygge en max-heap

Input: En (uferdig) heap A med n elementer der i er roten
Output: En mindre uferdig heap

```
1  Procedure BubbleDown(A, i, n)
2      largest ← i
3      left ← 2i + 1
4      right ← 2i + 2
5
6      if left < n and A[largest] < A[left] then
7          largest ← left
8
9      if right < n and A[largest] < A[right] then
10         largest ← right
11
12     if i ≠ largest then
13         A[i], A[largest] ← A[largest], A[i]
14         BubbleDown(A, largest, n)
```

Algorithm: Bygg en max-heap

Input: Et array A med n elementer
Output: A som en max-heap

```
1  Procedure BuildMaxHeap(A, n)
2      for i ← ⌊n/2⌋ down to 0 do
3          BubbleDown(A, i, n)
```

Heapsort – Implementasjon

Algorithm: Heapsort

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

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
6     return A
```

Heapsort – Kjøretidsanalyse (`BubbleDown`)

- Merk at linje 2–13 er konstanttidsoperasjoner
- Vi bryr oss bare om antall rekursive kall
- Algoritmen terminerer garantert når $i \geq \frac{n}{2}$
- Hvert rekursive kall øker i til $2i + 1$ eller $2i + 2$
- Altså *dobles* i for hvert rekursive kall
- Hvis en heap har høyde h inneholder den mindre enn 2^{h+1}
  - h er i $\mathcal{O}(\log(n))$, siden en heap er et *komplett* binærtre
- Dermed gjør vi maksimalt h rekursive kall
- Så `BubbleDown` er i $\mathcal{O}(\log(n))$

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
1  Procedure BubbleDown(A, i, n)
2  |  largest ← i
3  |  left ← 2i + 1
4  |  right ← 2i + 2
5  |
6  |  if left < n and A[largest] < A[left] then
7  |  |  largest ← left
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest ← right
11 |
12 |  if i ≠ largest then
13 |  |  A[i], A[largest] ← A[largest], A[i]
14 |  |  BubbleDown(A, largest, n)
```

Heapsort – Kjøretidsanalyse (`BuildMaxHeap`)

- Vi gjør $\frac{n}{2}$ kall på `BubbleDown`
- Siden `BubbleDown` er $\mathcal{O}(\log(n))$ og vi gjør $\frac{n}{2}$ kall
  - *virker* dette som $\mathcal{O}(n \cdot \log(n))$
  - Men det *er faktisk* $\mathcal{O}(n)$!
  - Å vise dette er utenfor pensum
- Intuitivt er det fordi
  - $\frac{n}{2}$ noder vil ikke ha noen kall på `BubbleDown`
  - 2^{h-1} noder vil ha kun ett kall på `BubbleDown`
  - 2^{h-2} noder vil ha kun to kall på `BubbleDown`
  - ...
  - kun rotnoden vil kunne treffe verste tilfellet til `BubbleDown`

Algorithm: Bygg en max-heap

```
1 Procedure BuildMaxHeap(A, n)
2     for i ← ⌊n/2⌋ down to 0 do
3         BubbleDown(A, i, n)
```

Heapsort – Kjøretidsanalyse (`HeapSort`)

- Vi vet at `BuildMaxHeap` er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på `BubbleDown`
  - som er i $\mathcal{O}(\log(n))$
  - Her vil vi alltid kalle `BubbleDown` fra rotnoden
- Dette gir $\mathcal{O}(n \cdot \log(n))$ i kjøretidskompleksitet

Algorithm: Heapsort

```
1 Procedure HeapSort(A)
2     BuildMaxHeap(A, n)
3     for i ← n − 1 down to 0 do
4         A[0], A[i] ← A[i], A[0]
5         BubbleDown(A, 0, i)
6     return A
```