Sortering: Bubble, Selection, Insert, Heap

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

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

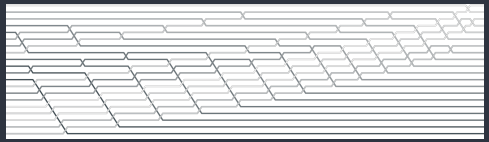
Høsten 2022

Oversikt uke 37–38

Oversikt uke 37–38

- Denne og neste uke handler om *sortering*
- Denne uken skal vi først definere og motivere problemet
- Så skal vi lære
  - Bubble sort
  - Selection sort
  - Insertion sort
  - Heapsort
- Neste uke skal vi lære
  - Merge sort
  - Quicksort
  - Bucket sort
  - Radix sort

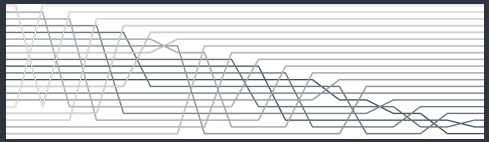
Illustrasjoner



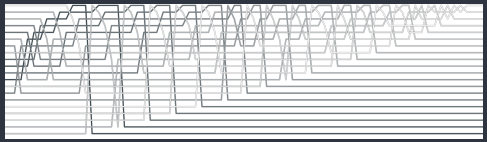
Bubble sort



Insertion sort



Selection sort



Heapsort

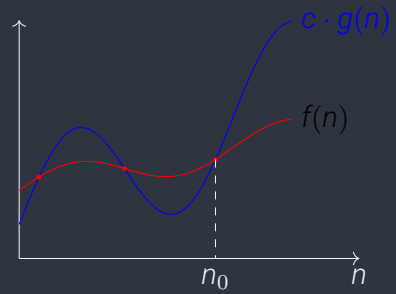


$\mathcal{O}$-NOTASJON HANDLER OM VEKST

- 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
- $f(n)$ 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

```
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 ← 0 to n − 1 do
3 |  |  for j ← 0 to n − 1 do
4 |  |  |  ··· for k ← 0 to n − 1 do
5 |  |  |  |  Constant(i)                          // O(n^k)
```

```
1 Procedure Exponential(n)
2 |  if n = 0 then
3 |  |  return
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
- $\mathcal{O}$-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
$\mathcal{O}(n\log(n))$	logaritmisk 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
  - *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
  - 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
   - 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. 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
- … 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)
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 ingen den indre loopen ikke gjør noen bytter

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!
- 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
- 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

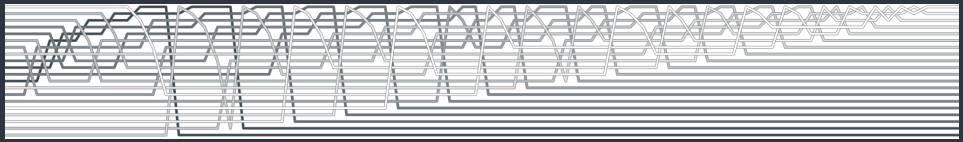
- Analysen her blir tilnærmet lik den for bubble- og selection sort
- Den ytre loopen kjører $\mathcal{O}(n)$ ganger
- Den indre loopen kjører $\mathcal{O}(n)$ ganger
- Da får vi $\mathcal{O}(n^2)$ kjøretidskompleksitet
- Insertion sort bryter «ofte» ut av den indre loopen
- Den er spesielt rask på «nesten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

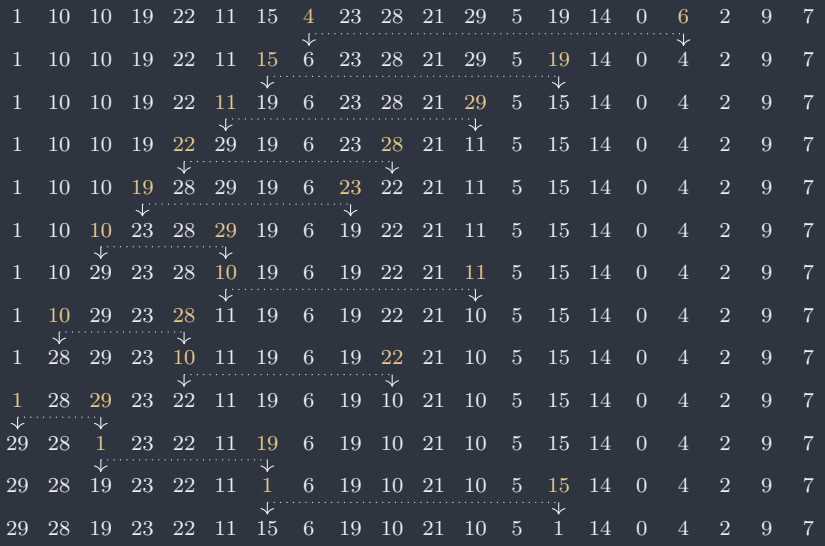


- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet
  3. pop fra max-heapen og plasser 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 ligger på plass $2i + 1$
  - og høyre barn ligger på plass $2i + 2$
- `BuildMaxHeap` gjør et array om til en max-heap

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 ← left, largest
8
9      if right < n and A[largest] < A[right] then
10         largest, right ← right, largest
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)
```

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
- Husk a $n \leq 2^{h+1} - 1$
  - der h er høyden av treet
  - og h er $\mathcal{O}(\log(n))$
- 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 ← left, largest
8  |
9  |  if right < n and A[largest] < A[right] then
10 |  |  largest, right ← right, largest
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 $\mathcal{O}(n \log(n))$
  - 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 $\mathcal{O}(n \log(n))$ 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)
```