Sortering: Merge, Quick, Bucket, Radix

IN2010 – Algoritmer og Datastrukturer

Lars Tveito og Daniel Lupp

Høsten 2020

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

Litt mer om sortering

Litt mer om sortering

- I forrige uke så vi fire sorteringsalgoritmer
- Denne uken skal vi se fire nye
- Men først skal vi lære to nye begreper

Stabilitet

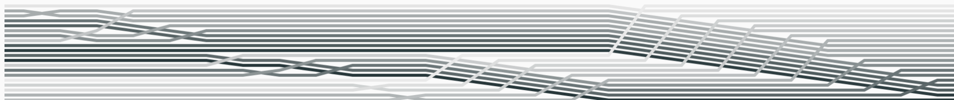
- Ofte sorterer vi på *nøkler*
    - for eksempel kan man sortere et person-objekt etter navn
- En sorteringsalgoritme kalles *stabil* dersom
    - for alle elementer x, y med samme nøkkel k
    - hvis x forekom før y før sortering
    - så forekommer x før y etter sortering
- Om en sorteringsalgoritme er stabil eller ikke er noen ganger 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
- I konteksten av sorteringsalgoritmer betyr det at
  - algortimen bruker $O(1)$ minne
- Alle algoritmene vi så forrige uke var in-place
  - men heapsort kan naturlig implementeres uten å være in-place
  - (lag en ny heap, push alle elementer på, pop dem av)

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[$0..i\text{-}1$] og A[$i..n\text{-}1$]
    - anvend merge sort rekursivt på A[$0..i\text{-}1$] og A[$i..n\text{-}1$]
    - flett sammen A[$0..i\text{-}1$] og A[$i..n\text{-}1$] sortert

Merge sort – Merge

Algorithm 1: 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(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         end
13     end
14     while i < |A1| do
15         A[i + j] ← A1[i]
16         i ← i + 1
17     end
18
19     while j < |A2| do
20         A[i + j] ← A2[j]
21         j ← j + 1
22     end
23
24     return A
```

Merge sort – Implementasjon

Algorithm 2: 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 3: Sortert fletting av to arrayer

```
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     end
12   end
13   while i < |A_1| do
14     A[i + j] ← A_1[i]
15     i ← i + 1
16   end
17   while j < |A_2| do
18     A[i + j] ← A_2[j]
19     j ← j + 1
20   end
21   return A
```

- Vi trenger bare analysere 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 $O(n)$

Merge sort – Kjøretidsanalyse (MergeSort)

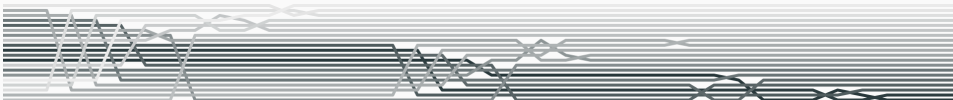
- Merk at vi får to rekursive kall for hvert kall
- Samtidig halverer vi input i hvert kall
- Vi gjør en $O(n)$ operasjoner for hvert kall
- Dybden på rekursjonen er $O(\log(n))$
  - fordi det er så mange ganger vi kan halvere input
- Da har vi at **MergeSort** er i $O(n \log(n))$

Algorithm 4: 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)
```

Quicksort

Quicksort – Idé



- Idéen bak quicksort er å velge et element, så
  - samle alt som er mindre enn elementet til venstre for det
  - samle alt som er større enn elementet til høyre for det
  - gjør dette rekursivt
- Litt mer presist:
  - vi velger en $0 \leq i < n$ som kalles pivot-elementet
    - søk fra venstre mot høyre etter et element som er større enn A[i]
    - søk fra høyre mot venstre etter et element som er mindre enn A[i]
    - bytt plass på disse, og søk etter nye som kan byttes
    - avslutt når høyre og venstre søkene krysser
  - fortsett rekursivt på alle som er hhv. til venstre og høyre for pivot

Quicksort – Haskell (ikke pensum)

```
qs []     = []
qs (x:xs) = qs ys ++ [x] ++ qs zs
    where (ys, zs) = partition (<x) xs
```

- En enkel (men ikke *veldig* rask) implementasjon av quicksort i Haskell
- Første linje sier at quicksort (`qs`) av en tom liste gir en tom liste
- Andre linje sier at
  - `x` er første elementet i en liste, og `xs` er resten av elementene
  - vi skal rekursivt kalle `qs` på `ys` og `zs`, og plassere `x` i midten
- Tredje linje sier at
  - `xs` splittes opp i to lister `ys` og `zs`
  - der alt i `ys` er mindre enn `x`
  - og alt i `zs` er ikke mindre enn `x`

Quicksort – Velge pivot

- Valget av pivot er avgjørende for at quicksort skal være effektiv
- Man har ingen *effektiv* måte å finne det *beste* pivot-elementet
  - Det beste pivot-elementet er den medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga
- Vanlige strategier er
  - Velg tilfeldig
  - Velg den medianverdien av A[0], A[$n//2$] og A[n-1]
- Å velge første eller siste som pivot
  - gir verste tilfelle for arrayer som allerede er sortert
- Vi antar en funksjon `ChoosePivot`
  - som velger pivot etter en rimelig strategi

Quicksort – Partition

Algorithm 5: Partition

Input: Et array A med n elementer, *low* og *high* er indekser

Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
1  Procedure Partition(A, low, high)
2      p ← ChoosePivot(A, low, high)
3      A[p], A[high] ← A[high], A[p]
4      pivot ← A[high]
5      left ← low
6      right ← high − 1
7      while left ≤ right do
8          while left ≤ right and A[left] ≤ pivot do
9              left ← left + 1
10         end
11         while right ≥ left and A[right] ≥ pivot do
12             right ← right − 1
13         end
14         if left < right then
15             A[left], A[right] ← A[right], A[left]
16     end
17     A[left], A[high] ← A[high], A[left]
18     return left
```

Quicksort – Implementasjon

Algorithm 6: Quicksort

Input: Et array A med n elementer, *low* og *high* er indekser

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

```
1  Procedure Quicksort(A, low, high)
2      if low ≥ high then
3          return A
4      p ← Partition(A, low, high)
5      Quicksort(A, low, p - 1)
6      Quicksort(A, p + 1, high)
7      return A
```

- Vi kaller på `Quicksort`(A, 0, $n - 1$) for å sortere hele arrayet

Quicksort – Kjøretidsanalyse

- `Partition` er i $O(high - low)$
  - lineær tid med hensyn på den delen av arrayet vi ser på
- Vi gjør rekursive kall der $high - low$ stadig blir mindre
- I *verste tilfelle* er $p = low$ eller $p = high$ i hvert kall
  - Dette skjer hvis vi velger første element som pivot på et array som allerede er sortert
- I verste tilfelle får vi $O(n^2)$ fordi vi får $O(n)$ rekursive kall
  - og hvert rekursive kall har lineær tid
- I beste tilfelle får er p midt mellom *low* og *high*
  - da halverer vi arbeidet for hvert rekursive kall, som gir $O(n \log(n))$
- Quicksort har $O(n^2)$ i verste tilfellet
  - Men dette skjer sjeldent, så den er *som regel* svært effektiv

Algorithm 7: Quicksort

```
1 Procedure Quicksort(A, low, high)
2   if low ≥ high then
3     return A
4   p ← Partition(A, low, high)
5   Quicksort(A, low, p - 1)
6   Quicksort(A, p + 1, high)
7   return A
```

Bucket sort

Bucket sort – Introduksjon

- Alle sorteringsalgoritmene vi har sett til nå er basert på sammenligning
  - slike sorteringsalgoritmer kan ikke bli bedre en $O(n \log(n))$!
- Hvis vi vet mer om verdiene som skal sorteres, kan vi få til noe bedre
- Bucket sort går ut på å lage N bøtter
  - hvor hver bøtte svarer til en kategori eller sort
  - og kategoriene er ordnet
- Elementene vi skal sortere har en kategori, som vi kaller nøkkelen
- I bucket sort plasserer vi hvert element i riktig bøtte
  - basert på nøkkelen
- Så løper vi gjennom hver bøtte
  - plasserer elementene tilbake i arrayet
- Merk at man noen ganger ønsker å sortere bøttene hver for seg
  - De lærde virker til å strides på dette punktet

Bucket sort – Implementasjon

Algorithm 8: Bucket sort

Input: Et array A med n elementer

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

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      end
7      j ← 0
8      for k ← 0 to N − 1 do
9          for hver x i listen B[k] do
10             A[j] ← x
11             j ← j + 1
12         end
13     end
14     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $O(n)$
- Vi går gjennom alle bøttene
  - Dette er $O(N)$
- For hver bøtte, går vi gjennom elementene i bøtta
- Men det er bare n elementer i bøttene!
- Som gir $O(N + n)$
- Hvis N er liten er dette strålende!

Algorithm 9: Bucket sort

```
1  Procedure BucketSort(A)
2      La B være et array med N tomme lister
3      for i ← 0 to n − 1 do
4          La k være nøkkelen assosiert med A[i]
5          Legg til A[i] på slutten av listen B[k]
6      end
7      j ← 0
8      for k ← 0 to N − 1 do
9          for hver x i listen B[k] do
10             A[j] ← x
11             j ← j + 1
12         end
13     end
14     return A
```

Bucket sort – Kortstokk

- Vi kan definere en ordning på kort, slik at

$$A < 2 < 3 < \cdots < 10 < J < Q < K$$

- Videre har vi at

$$\clubsuit < \diamondsuit < \spadesuit < \heartsuit$$

- Vi ønsker å sortere kort vi får på hånden, slik at
  - alle med samme sort kommer sammen
  - hver sort er innbyrdes sortert
- Det kan vi oppnå ved å gjøre to bucket sort
  - sorter først på verdi
  - så sorter på sort

Bucket sort – Kortstokk (eksempel)

$A\heartsuit$ $2\heartsuit$ $A\clubsuit$ $J\clubsuit$ $9\spadesuit$ $6\diamondsuit$ $7\diamondsuit$ $7\clubsuit$ $K\heartsuit$ $8\heartsuit$

$[A\heartsuit\ A\clubsuit]\ [2\heartsuit]\ [\,]\ [\,]\ [\,]\ [6\diamondsuit]\ [7\diamondsuit\ 7\clubsuit]\ [8\heartsuit]\ [9\spadesuit]\ [\,]\ [J\clubsuit]\ [\,]\ [K\heartsuit]$

$A\heartsuit$ $A\clubsuit$ $2\heartsuit$ $6\diamondsuit$ $7\diamondsuit$ $7\clubsuit$ $8\heartsuit$ $9\spadesuit$ $J\clubsuit$ $K\heartsuit$

$[A\clubsuit\ 7\clubsuit\ J\clubsuit]\ [6\diamondsuit\ 7\diamondsuit]\ [9\spadesuit]\ [A\heartsuit\ 2\heartsuit\ 8\heartsuit\ K\heartsuit]$

$A\clubsuit$ $7\clubsuit$ $J\clubsuit$ $6\diamondsuit$ $7\diamondsuit$ $9\spadesuit$ $A\heartsuit$ $2\heartsuit$ $8\heartsuit$ $K\heartsuit$

Radix sort

Radix sort – Idé

- Radix sort er nært beslektet bucket sort
  - (noen vil ikke engang skille mellom disse algoritmene)
- Eksempelet vi så med å sortere kort, er faktisk en type radix sort
- En svært god intuisjon for radix sort er at det er
  - *suksessiv anvendelse av bucket sort*
- Radix sort kan brukes for data som kan ordnes *leksikografisk*

Radix sort – Leksikografiske ordninger

- Leksikografiske ordninger er en generalisering av alfabetisk rekkefølge
  - Vi kan ordne ord etter bokstavene (som i seg selv er ordnet)
    - der første bokstav prioriteres over andre, som prioriteres over tredje, osv...
  - Vi kan ordne tall på samme måte
    - der første siffer prioriteres over andre, som prioriteres over tredje, osv...
- Mer generelt kan vi tenke oss tupler med symboler $(a_1, a_2, \ldots, a_d)$
- Vi sier at $(a_1, a_2, \ldots, a_d) < (b_1, b_2, \ldots, b_d)$ hvis
  - $a_1 < b_1$ eller
  - $a_1 = b_1$ og $(a_2, \ldots, a_d) < (b_2, \ldots, b_d)$
- Fra korteksempelet har vi for eksempel at $7\clubsuit < J\clubsuit$
  - fordi $\clubsuit = \clubsuit$, men $7 < J$

Radix sort – Eksempel (heltall)

1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632

1814, 0232, 2888, 0031, 1455, 2242, 4345, 1470, 0515, 3632

[1470] [0031] [0232, 2242, 3632] [ ] [1814] [1455, 4345, 0515] [ ] [ ] [2888] [ ]

1470, 0031, 0232, 2242, 3632, 1814, 1455, 4345, 0515, 2888

[ ] [1814, 0515] [ ] [0031, 0232, 3632] [2242, 4345] [1455] [ ] [1470] [2888] [ ]

1814, 0515, 0031, 0232, 3632, 2242, 4345, 1455, 1470, 2888

[0031] [ ] [0232, 2242] [4345] [1455, 1470] [0515] [3632] [ ] [1814, 2888] [ ]

0031, 0232, 2242, 4345, 1455, 1470, 0515, 3632, 1814, 2888

[0031, 0232, 0515] [1455, 1470, 1814] [2242, 2888] [3632] [4345] [ ] [ ] [ ] [ ] [ ]

0031, 0232, 0515, 1455, 1470, 1814, 2242, 2888, 3632, 4345

31, 232, 515, 1455, 1470, 1814, 2242, 2888, 3632, 4345

Radix sort – Implementasjon for positive heltall

Algorithm 10: Radix sort for positive heltall

Input: Et array A med n positive heltall

Output: Et *sortert* array med de samme n positive heltallene

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     end
6     return A
```

Radix sort – Kjøretidsanalyse

- Vi vet allerede at `BucketSort` er i $O(N + n)$
- Radix sort må gjøre d antall bucket sort
  - I tillegg må man beregne d (som tar $O(n)$ tid)
- Generelt er radix sort i $O(d(N + n))$
- For heltall kan vi sette $N = 10$
  - (dersom vi bruker titallsystemet)
- Da får vi $O(d(10 + n)) = O(d \cdot n)$
- Hvis vi vet at tallene ikke blir større enn 10^{10}
  - Et naturlig valg, siden $2^{32} < 10^{10}$
- Så får vi $O(10n) = O(n)$

Algorithm 11: Radix sort

```
1 Procedure RadixSort(A)
2     d ← antall siffer i det største tallet
3     for i ← d down to 0 do
4         A ← BucketSort(A) etter det
              ite sifferet
5     end
6     return A
```