Sortering: Heap, Quick, Bucket og Radix

IN2010 – Algoritmer og Datastrukturer

Lars Tveito

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

Høsten 2023

Litt mer om sortering

Litt mer om sortering

- Vi har sett på fire sorteringsalgoritmer tidligere
- 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 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
- I konteksten av sorteringsalgoritmer betyr det at algortimen bruker $\mathcal{O}(1)$ minne
- Av algoritmene vi har sett til nå er Bubble, Selection og Insertion in-place
  - Men Merge er ikke in-place.

Heapsort

Heapsort – Idé

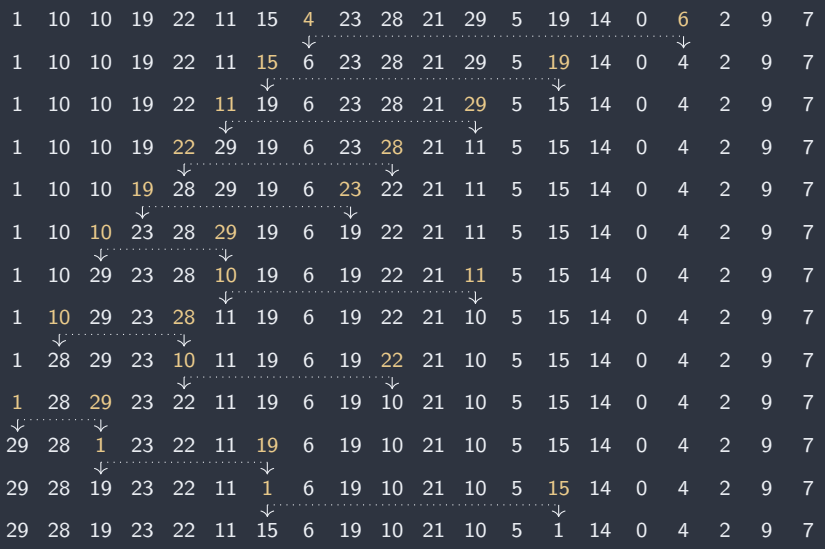


- Idéen bak heapsort er å bygge en heap og poppe elementer av heapen
- Fordi en heap kan implementeres med et array, gjør vi arrayet om til en heap
- Litt mer presist skal vi
  1. gjør arrayet om til en max-heap
  2. la i være $n - 1$ der n er størrelsen på arrayet
  3. pop fra max-heapen og plasser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

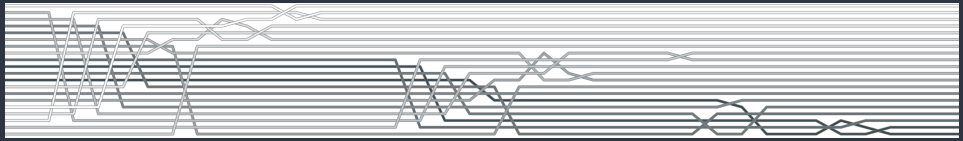
- Vi vet at `BuildMaxHeap` er i $\mathcal{O}(n)$
- Etter det gjør vi n iterasjoner
- For hver iterasjon kaller vi på `BubbleDown`
  - som er i $\mathcal{O}(\log(n))$
  - Her vil vi alltid kalle `BubbleDown` fra rotnoden
- Dette gir $\mathcal{O}(n \log(n))$ i kjøretidskompleksitet

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)
```

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 – 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 ideelle pivot-elementet er medianverdien av arrayet
  - Å finne medianverdien krever sortering, så vinninga går opp i spinninga
- Vanlige strategier er
  - Velg tilfeldig
  - Velg medianverdien av `A[0]`, `A[`$\lfloor \frac{n}{2} \rfloor$`]` 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: 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         while right ≥ left and A[right] ≥ pivot do
11             right ← right − 1
12         if left < right then
13             A[left], A[right] ← A[right], A[left]
14     A[left], A[high] ← A[high], A[left]
15     return left
```

Quicksort – Implementasjon

Algorithm: 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 $\mathcal{O}(high - low)$
  - lineær tid med hensyn til 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
- Verste tilfelle er $\mathcal{O}(n^2)$ fordi vi får $\mathcal{O}(n)$ rekursive kall
  - og hvert rekursive kall har lineær tid
- I beste tilfelle er pivot-elementet den «midterste» verdien i arrayet
  - da halverer vi arbeidet for hvert rekursive kall, som gir $\mathcal{O}(n \log(n))$
- Quicksort har $\mathcal{O}(n^2)$ i verste tilfellet
  - Men dette skjer sjeldent, så den er *som regel* svært effektiv

Algorithm: 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 $\mathcal{O}(n \log(n))$!
- 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, der 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 og 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 – 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♡ 2♡ A♣ J♣ 9♠ 6♢ 7♢ 7♣ K♡ 8♡

[A♡ A♣] [2♡] [] [] [] [6♢] [7♢ 7♣] [8♡] [9♠] [] [J♣] [] [K♡]

A♡ A♣ 2♡ 6♢ 7♢ 7♣ 8♡ 9♠ J♣ K♡

[A♣ 7♣ J♣] [6♢ 7♢] [9♠] [A♡ 2♡ 8♡ K♡]

A♣ 7♣ J♣ 6♢ 7♢ 9♠ A♡ 2♡ 8♡ K♡

Bucket sort – Implementasjon

ALGORITHM: 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      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

Bucket sort – Kjøretidsanalyse

- Først må vi gå gjennom alle elementene
  - Dette er $\mathcal{O}(n)$
- Vi går gjennom alle bøttene
  - Dette er $\mathcal{O}(N)$
- For hver bøtte, går vi gjennom elementene i bøtta
- Men det er bare n elementer i bøttene!
  - Kun n elementer i *alle bøttene tilsammen*
- Som gir $\mathcal{O}(N + n)$
- Når N er konstant kan det forenkles til $\mathcal{O}(n)$

Algorithm: 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      j ← 0
7      for k ← 0 to N − 1 do
8          for hver x i listen B[k] do
9              A[j] ← x
10             j ← j + 1
11     return A
```

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: 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 − 1 down to 0 do
4         A ← BucketSort(A) etter det ite sifferet
5     return A
```

Radix sort – Kjøretidsanalyse

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

Algorithm: Radix sort

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