Sorteringsmetoder

Ola Natvig <olanatv@stud.ntnu.no>

Først litt praktisk info

- Gruppeøvinger har startet
- http://selje.idi.ntnu.no:1234/tdt4120/gruppeoving.php
- De som ikke har fått gruppe må velge en av de 4 gruppende og sende mail til algdat@idi.ntnu.no

Nordisk mesterskap i programmering (NCPC)

- Lørdag 6.10 11:00 - 16:00
- Åpent for alle
- Gratis pizza og brus
- Vinn tur til nordvest-europeisk mesterskap
- Java / C / C++
- (frivillig) Treningsrunde 29.9 11:00 - 16:00
- http://ncpc.idi.ntnu.no/ncpc2007/ntnu.html

Agenda

- Sortert?
- Boblesortering
- Innsettingssortering
- Heapsort
- Mergesort
- Quicksort
- Stabilitet
- Sortering i den virkelige verden
- Hvorfor ikke boblesortering?
- Hva med Moores lov?
- Gjennomgang av øvinger

Sortert?

$$X_1 \leq X_2 \leq X_3 \leq ..... \leq X_{n-1} \leq X_n$$

6	2	4	1	1	3	7	2

1	1	2	2	3	4	6	7

Boblesortering

- ”Sjekker” definisjonen av sortert

$$X_1 \leq X_2 \leq X_3 \leq ..... \leq X_{n-1} \leq X_n$$

- Hvis et usortert par oppdages: $X_i > X_{i+1}$
  - Bytt om på de to verdiene
- Gjentar helt til ingen usorterte par oppdages

Boblesortering

```
def sjekkOgBytt(A):
    konflikt = False
    for i in range(0, len(A) - 1):
        if A[i] > A[i + 1]:
            A[i], A[i + 1] = A[i + 1], A[i] # bytt
            konflikt = True
    return konflikt

def boble(A):
    while sjekkOgBytt(A): pass
```

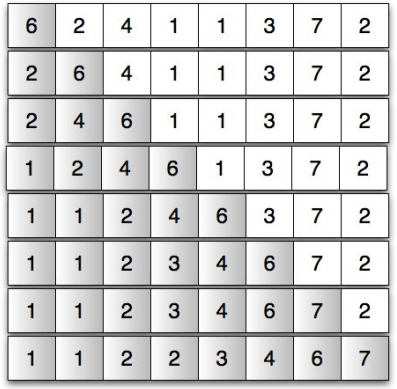
Boblesortering

- Kjøretid
  - I beste tilfelle er input allerede sortert, da oppdages det ingen konflikter og algoritmen tar O(n) tid.
  - I verste tilfelle er input sortert i motsatt rekkefølge
    - Element 1 må da flyttes n plasser, element 2 n - 2 plasser, osv... $O(n^2)$

Innsettingssortering

- Deler input i to deler:
  - En sortert del
  - En usortert
- Tar første element i den usorterte delen og setter denne inn i den sorterte på riktig plass.
- Gjentar inntil hele input er sortert

Innsettingssortering



6	2	4	1	1	3	7	2
2	6	4	1	1	3	7	2
2	4	6	1	1	3	7	2
1	2	4	6	1	3	7	2
1	1	2	4	6	3	7	2
1	1	2	3	4	6	7	2
1	1	2	3	4	6	7	2
1	1	2	2	3	4	6	7

Innsettingssortering

```
def innsetting(A):
    for i in range(1, len(A)):
        settInn = A[i]
        j = i - 1
        while j >= 0 and settInn < A[j]:
            A[j + 1] = A[j] # gi plass til settInn
            j = j - 1
        A[j + 1] = settInn
```

Innsettingssortering

- Kjøretid
  - I beste tilfelle er listen allerede sortert
    - Da går den indre løkken (while løkken) ingen ”runder” og kjøretiden blir O(n)
  - I verste tilfelle er input sortert i motsatt rekkefølge
    - Da går den indre løkken fra i og ned til 0 for hver verdi som skal settes inn og kjøretiden blir $O(n^2)$

Heapsort

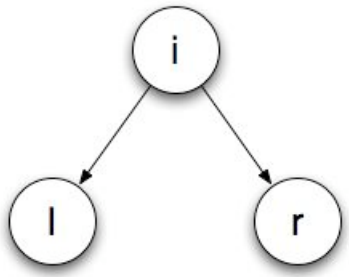
- Bruker en (binær) heap (haug)
- En heap er en trestruktur der følgende utsagn gjelder: (heap property)
  - En foreldre node er alltid mindre(/større) eller lik barn nodene.
- En heap gjør det mulig for oss å hente ut (og fjærne) det minste (eller største) elementet i O(lg n) tid.

Binær heap

- En binær heap vedlikeholdes av en heapify rutine.
  - Heapify sørger for at heap property holdes ved like.
  - Heapify brukes også ved initiering av en heap, da som en subrutine i en build-heap rutine.

Binær heap: Heapify

- Heapify kalles på en node ”i” i en min-heap.
  - Heapify finner ut hvilke noder av ”i” og ”i”s to barn ”l” og ”r” som er minst.
  - Hvis ”i” ikke er den minste noden
    - Bytt ”i” med den minste og kall heapify på ”i”s nye plass.



Binær heap: Bygg en heap

- For å bygge en heap lager man først en trestruktur av input, så starter man på det nest laveste nivået i treet og kaller heapify på hver av nodene i trestrukturen i omvendt bredde først rekkefølge.

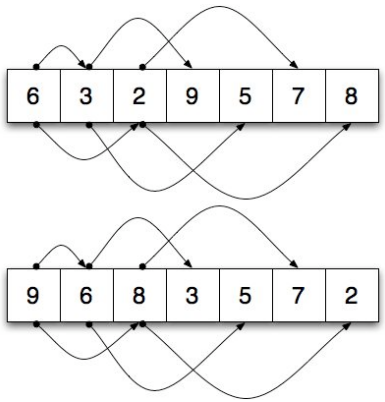
Binær heap som tabell/array

- Det er ikke vanlig å representere binære heaper som grafer.
- Det vanlige er å bruke tabeller
- Strukturen er da som følgende (gitt 1 indekserte tabeller)
  - Foreldre(): floor(indeks / 2)
  - Venstre(): 2 * index
  - Høyre(): 2 * index + 1

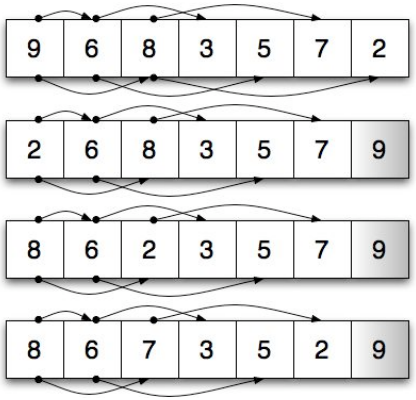
Heapsort

- Heapsort bygger først en max-heap av input
- For hver posisjon i fra slutten av tabellen til den nest første i tabellen
  - Bytt i med toppen av heapen
  - Krymp heapen slik at posisjon i ikke er med i heapen
  - Kall heapify på toppen av heapen

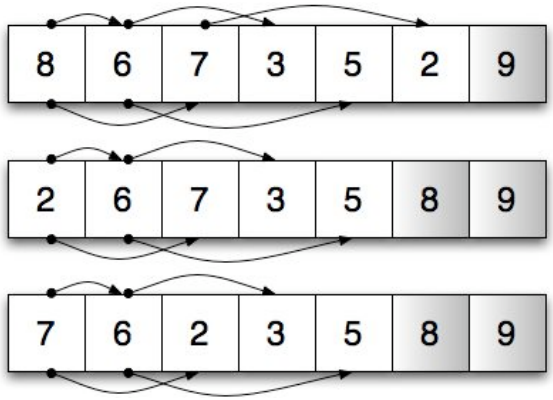
Heapsort: Bygg heap



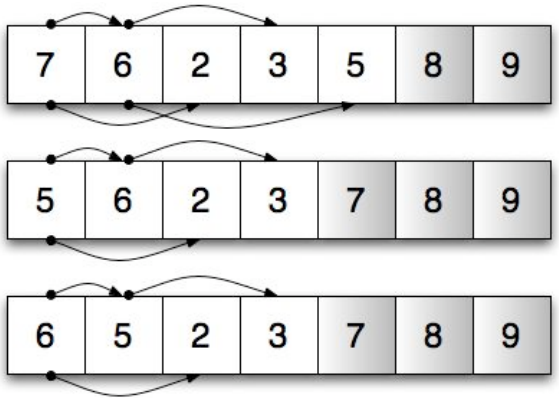
Heapsort: Kjøring



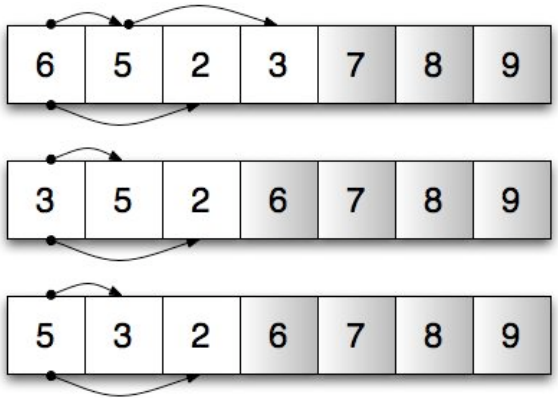
Heapsort: Kjøring



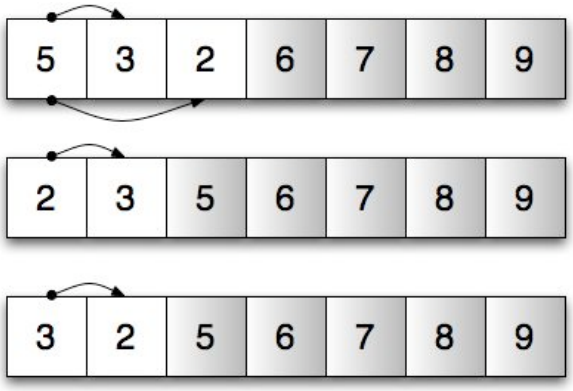
Heapsort: Kjøring



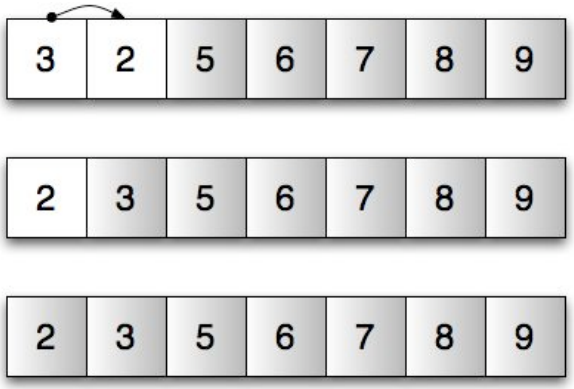
Heapsort: Kjøring



Heapsort: Kjøring



Heapsort: Kjøring



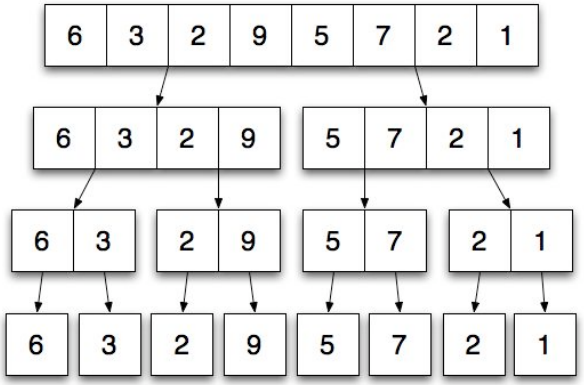
Heapsort: kjøretid

- En binær heap kan bygges i O(n) tid.
- For hvert tall som tas ut av heapen kjører vi heapify, heapify tar O(lg n) tid.
- Kjøretiden blir da O(n + n lg n) = O(n lg n)

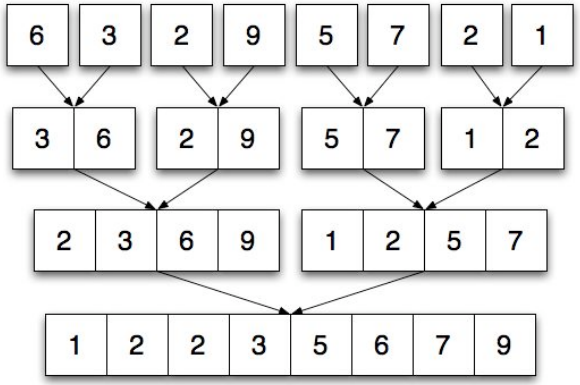
Mergesort

- Splitt og hersk sortering
- Del input i to like store biter
- Kjøre mergesort på hver av de to bitene
- Flett (merge) de to bitene sammen
  - Velg det minste av bitenes første element helt til begge er tomme.
- Returner den flettede listen

Mergesort: Splitt



Mergesort: Hersk



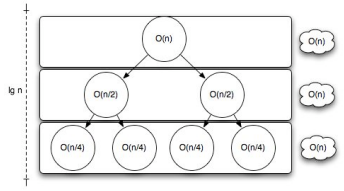
Mergesort: merge

```
def merge(A, B):
    i, j = 0, 0
    R = [0] * (len(A)+ len(B))
    while len(A) > i and len(B) > j:
        if A[i] <= B[j]:
            R[i + j] = A[i]
            i = i + 1
        else:
            R[i + j] = B[j]
            j = j + 1
    while len(A) > i:
        R[i + j] = A[i]
        i = i + 1
    while len(B) > j:
        R[i + j] = B[j]
        j = j + 1
    return R
```

Mergesort: kode

```
def mergesort(A):
    if (len(A) > 1):
        split = len(A) // 2
        return merge(mergesort(A[:split]), mergesort(A[split:]))
    else: return A
```

Mergesort: kjøretid



- Hvert nivå i rekursjonstreet har samlet kjøretid O(n)
- Et ballansert binærtre har høyde lg n
- Kjøretid blir da O(n lg n)

Quicksort

- Til forveksling lik mergesort, men:
  - ”Hersker før splitt”
  - Bruker mindre minne
- Isteden for fletting (merge) brukes partisjonering.

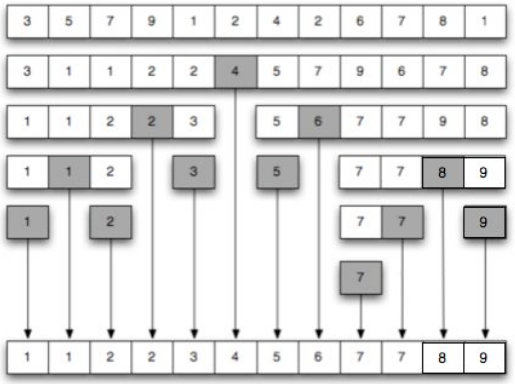
Quicksort: Partisjonering

- Velger et element i input som kalles ”pivot element”
- Deler resten av input i to deler, en del der alle tallene er mindre eller lik pivot elementet, en annen del der alle tallene er større enn pivot elementet.

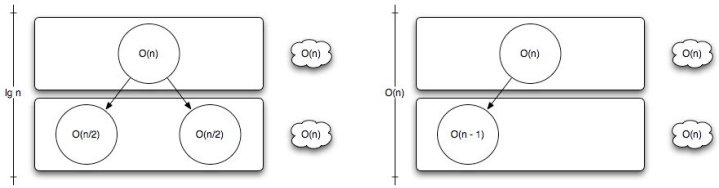
Quicksort: fremgangsmåte

- Quicksort partisjonerer input og kaller seg selv hver av de to partisjonene.
- Quicksort avslutter når det bare er et element igjen.

Quicksort: Kjøring



Quicksort: kjøretid



- Quicksorts kjøretid er veldig avhenging av at pivot elementet er nærest mulig medianen av input.
- Om pivot elementet er nærme medianen av input er kjøretiden lik mergesorts O(n lg n)
- Men dersom pivot er det største eller minste elementet i input blir kjøretiden kvadratisk

Quicksort: Hvordan velge pivot

- Siden quicksort kan få kvadratisk kjøretid hvis man velger dårlige pivot elementer, er det viktig å velge ”riktig” pivot.
- Naivt:
  - Første eller siste element i input. (problematisk ?)
- Median av første, siste og midterste element?
- Tilfeldig element?

Quicksort: popularitet

- Quicksort er en av de mest populære sorteringsalgoritmene fordi:
  - O(n lg n) kjøretid i de fleste tilfeller, worst-case kjøretid på $O(n^2)$ kan som regel unngås eller gjøres svært lite sannsynlig
  - Partisjonering kan gjøres veldig effektivt på vanlige datamaskinarkitekturer
  - Bruker mindre minne enn f.eks. Mergesort

Quicksort: kode

```
def partition(A, fra, til):
    pivot = A[til - 1] # siste element som pivot
    skille = fra - 1
    for i in range(fra, til - 1):
        if A[i] <= pivot:
            skille = skille + 1
            A[i], A[skille] = A[skille], A[i]
    A[til - 1], A[skille + 1] = A[skille + 1], A[til - 1]
    return skille + 1

def quicksort(A, fra, til):
    if til > fra:
        pivot = partition(A, fra, til)
        quicksort(A, fra, pivot)
        quicksort(A, pivot + 1, til)
```

Stabilitet

- En sorteringsalgoritme sies å være stabil dersom:
  - To objekter med samme sorteringsnøkkel har samme rekkefølge i resultatet som i input.

Sortering i den virkelige verden: Hvor

- Databaser
- Søkemotorer
- Postsortering
- Nesten alle brukerprogrammer
- ++++++

Sortering i den virkelige verden: Hvordan

- Java:
  - mergesort med noen optimaliseringer for stabil sortering (sortering av objekter)
  - Quicksort med optimaliseringer for ustabilsortering
    - Sjekk ut: Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function”
- C++
  - Mange implementasjoner av STL bruker en hybrid av Quicksort og Heapsort som kalles Introsort for ustabil sortering.

Sortering i den virkelige verden: Hvordan

- Generellt
  - Veldig vanlig med quicksort som sortere input delvis (ikke kalle quicksort på nytt dersom det er færre enn f.eks. 10 elementer mellom ”fra” og ”til” pekerene) etterfulgt av innsettingssortering på hele listen når quicksort er ferdig.
- Postsortering o.l.:
  - Radix sort

Hvorfor ikke boblesortering?

- Lite eksperiment: Sorteringskonkurranse:
  - Deltagere:
    - Apple MacBook Pro: 2.33 GHz Intel Core 2 Duo, 2GB Ram, Python 2.5.1 og Boblesortering
    - Nokia N73: 220 MHz TI Prosessor, 20 MB Ram, Python 2.2.2 og Quicksort + Insertionsort hybrid
  - Data:
    - 5000 tilfeldige heltall
    - 10000 tilfeldige heltall

Hva med Moores lov

- Datamaskiner blir raskere
  - Dobling per 18. Måned
- Til nå har denne doblingen kommet av raskere prosessorkjerner
- Dette endrer seg:
  - Nå: flere kjerner
  - Om ti år: prosessorer med 100+ kjerner.
- Hvordan utnytte dette?
  - Parallellitet (IKKE PENSUM)

Parallell quicksort

- Etter første partisjonering:
  - Start hver av de to rekursjonene i egne tråder.
  - Quicksort kan nå utnytte opptil to kjerner.
  - Det finnes mer fancy måter å sortere på mange kjerner, men denne er grei å holde styr på.

Gjennomgang av øving