Introduksjon

IN2010 – Algoritmer og datastrukturer

Lars Tveito

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

Velkommen

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
    - terminere etter et endelig antall steg
    - hvert steg må være helt presist definert

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
    - terminere etter et endelig antall steg
    - hvert steg må være helt presist definert
    - ta null eller flere input

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
    - terminere etter et endelig antall steg
    - hvert steg må være helt presist definert
    - ta null eller flere input
    - produsere et output som står i forhold til input

Litt om algoritmer

- Hvis du gir en *presis* beskrivelse for hvordan man skal løse et *problem*, så har du beskrevet det vi kaller en algoritme
- Algoritmer er en serie med små og entydige steg
- En algoritme må
  - terminere etter et endelig antall steg
  - hvert steg må være helt presist definert
  - ta null eller flere input
  - produsere et output som står i forhold til input
- En algoritme *bør* være effektiv

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Strukturen gir oss informasjon om relasjonen mellom elementene

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Strukturen gir oss informasjon om relasjonen mellom elementene
- Algoritmer er avhengig av gode datastrukturer for å bli enkle og raske

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Strukturen gir oss informasjon om relasjonen mellom elementene
- Algoritmer er avhengig av gode datastrukturer for å bli enkle og raske
- Datastrukturer er avhengig av gode algoritmer for å opprettholde strukturen

Litt om datastrukturer

- En samling av verdier som følger en fast struktur utgjør en datastruktur
- Strukturen gir oss informasjon om relasjonen mellom elementene
- Algoritmer er avhengig av gode datastrukturer for å bli enkle og raske
- Datastrukturer er avhengig av gode algoritmer for å opprettholde strukturen
- Et godt valg av datastruktur bør gjøre data vi trenger ofte lett tilgjengelig

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
    - da må dere finne en algoritme som løser problemet

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
    - da må dere finne en algoritme som løser problemet
- Ofte vil dere ikke finne algoritmen dere leter etter, og dere må lage den selv

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
    - da må dere finne en algoritme som løser problemet
- Ofte vil dere ikke finne algoritmen dere leter etter, og dere må lage den selv
    - da gjelder det å studere *lignende* problemer

Perspektiv

- Vi vil fokusere på å forstå, analysere, anvende og lage algoritmer
- Vi vil bygge forståelse ved å studere, og helst implementere, mange eksempler
- Vi vil analysere dem i form av å forutsi hvor effektive de er
- Dere vil anvende algoritmer når dere løser oppgaver
  - da må dere finne en algoritme som løser problemet
- Ofte vil dere ikke finne algoritmen dere leter etter, og dere må lage den selv
  - da gjelder det å studere *lignende* problemer
  - eller løse et enklere problem først

Noen råd

- Vær en god medstudent

Noen råd

- Vær en god medstudent
- Dra på gruppetimer

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
  - `https://open.kattis.com/`

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
    - `https://open.kattis.com/`
    - `https://leetcode.com/`

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
    - `https://open.kattis.com/`
    - `https://leetcode.com/`
    - `https://projecteuler.net/`

Noen råd

- Vær en god medstudent
- Dra på gruppetimer
- Skriv *litt* kode hver dag
- Finn på egne oppgaver
- Gjør en av disse om til din nye hobby
    - `https://open.kattis.com/`
    - `https://leetcode.com/`
    - `https://projecteuler.net/`
    - `https://cses.fi/problemset/`

Undervisningstilbud

- Forelesninger tirsdager kl. 12:15

Undervisningstilbud

- Forelesninger tirsdager kl. 12:15
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser

Undervisningstilbud

- Forelesninger tirsdager kl. 12:15
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
  - Opptak vil publiseres

Undervisningstilbud

- Forelesninger tirsdager kl. 12:15
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
  - Opptak vil publiseres
- Det er ukentlige gruppetimer

Undervisningstilbud

- Forelesninger tirsdager kl. 12:15
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
  - Opptak vil publiseres
- Det er ukentlige gruppetimer
  - Vi har en gjeng med 12 enestående gruppelærere og rettere

Undervisningstilbud

- Forelesninger tirsdager kl. 12:15
  - Videoene fra høsten 2020 kan brukes som ekstra læringsressurser
  - Opptak vil publiseres
- Det er ukentlige gruppetimer
  - Vi har en gjeng med 12 enestående gruppelærere og rettere
- Det er to lab-timer der dere kan jobbe med oppgaver

Obligatoriske innleveringsoppgaver

- Det vil være tre obligatoriske innleveringsoppgaver

Obligatoriske innleveringsoppgaver

- Det vil være tre obligatoriske innleveringsoppgaver
- Dere skal *levere i grupper* på inntil tre personer

Obligatoriske innleveringsoppgaver

- Det vil være tre obligatoriske innleveringsoppgaver
- Dere skal *levere i grupper* på inntil tre personer
- Dere må selv finne noen å levere sammen med

Obligatoriske innleveringsoppgaver

- Det vil være tre obligatoriske innleveringsoppgaver
- Dere skal *levere i grupper* på inntil tre personer
- Dere må selv finne noen å levere sammen med
- Det er mulig å få fritak fra gruppelevering ved behov

Obligatoriske innleveringsoppgaver

- Det vil være tre obligatoriske innleveringsoppgaver
- Dere skal *levere i grupper* på inntil tre personer
- Dere må selv finne noen å levere sammen med
- Det er mulig å få fritak fra gruppelevering ved behov

Obligatoriske innleveringsoppgaver

- Det vil være tre obligatoriske innleveringsoppgaver
- Dere skal *levere i grupper* på inntil tre personer
- Dere må selv finne noen å levere sammen med
- Det er mulig å få fritak fra gruppelevering ved behov

Innlevering	Publiseres	Frist
Innlevering 1	25. august	12. september
Innlevering 2	15. september	3. oktober
Innlevering 3	6. oktober	24. oktober

Kjøreplan for semesteret

En semesterkalender for er publisert på semestersiden

Abstrakte datatyper

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner
- En abstrakt datatype realiseres som en konkret *datastruktur*

Abstrakte datatyper

- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner
- En abstrakt datatype realiseres som en konkret *datastruktur*
- Grensesnitt (`interface`) er Java sin måte å snakke om abstrakte datatyper

Abstrakte datatyper

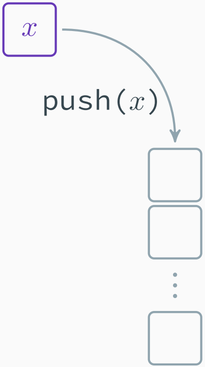
- En abstrakt datatype (ADT) sier om *oppførsel*, men ingenting om implementasjon
- En abstrakt datatype kan ha flere ulike konkrete implementasjoner
- En abstrakt datatype realiseres som en konkret *datastruktur*
- Grensesnitt (`interface`) er Java sin måte å snakke om abstrakte datatyper
- En stor del av IN2010 handler om å finne *effektive* implementasjoner for noen sentrale abstrakte datatyper

Stack

En *stack* er en liste hvor alle innsettinger og slettinger forekommer på *samme* ende av listen

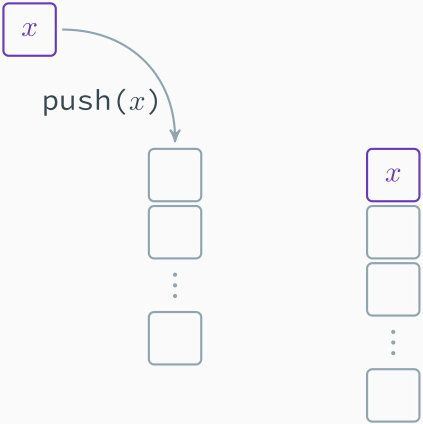
Stack

En *stack* er en liste hvor alle innsettinger og slettinger forekommer på *samme* ende av listen



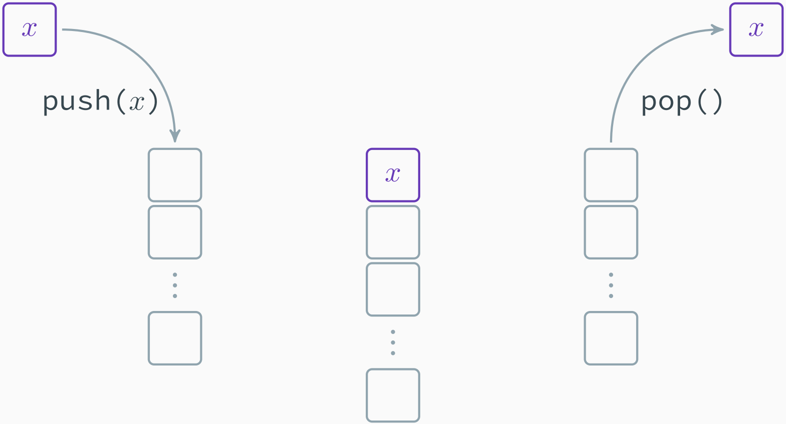
Stack

En *stack* er en liste hvor alle innsettinger og slettinger forekommer på *samme* ende av listen



Stack

En *stack* er en liste hvor alle innsettinger og slettinger forekommer på *samme* ende av listen

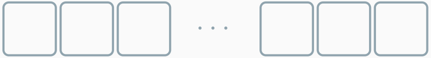


Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen

Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen



Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen

Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen

Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen

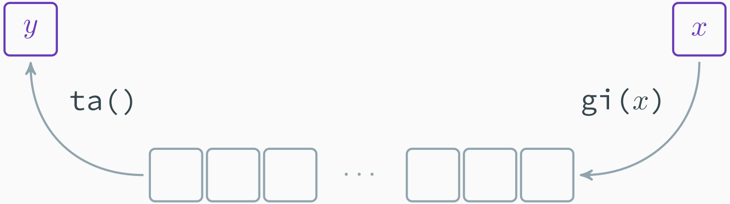
Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen

- I Java kalles operasjonene `offer()`/`poll()`

Kø

En *kø* er en liste hvor alle innsettinger forekommer på den ene enden av listen, og alle slettinger forekommer på den andre enden av listen



- I Java kalles operasjonene `offer()`/`poll()`
  - Kalles også ofte `enqueue()`/`dequeue()`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`

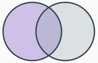
Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
    - Sjekke om x forekommer i mengden: $x \in A$
    - Sett inn i mengden: `Insert(A, x)`
    - Fjern et element fra mengden: `Remove(A, x)`
    - Operasjonene union, snitt og differanse:

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



`Union(A,B)`

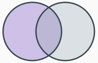
Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

Set

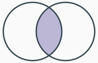
- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
    - Sjekke om x forekommer i mengden: $x \in A$
    - Sett inn i mengden: `Insert(A, x)`
    - Fjern et element fra mengden: `Remove(A, x)`
    - Operasjonene union, snitt og differanse:

$A \cup B$



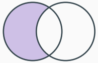
`Union(A,B)`

$A \cap B$



`Intersection(A,B)`

$A \setminus B$



`Difference(A,B)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$ $\quad$ $A \cap B$ $\quad$ $A \setminus B$ $\quad$ $B \setminus A$

`Union(A,B)` `Intersection(A,B)` `Difference(A,B)` `Difference(B,A)`

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$ $A \cap B$ $A \setminus B$ $B \setminus A$

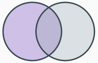
`Union(A,B)` `Intersection(A,B)` `Difference(A,B)` `Difference(B,A)`

- Hverken rekkefølge eller antall forekomster spiller noen rolle i mengder

Set

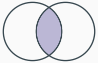
- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
  - Sjekke om x forekommer i mengden: $x \in A$
  - Sett inn i mengden: `Insert(A, x)`
  - Fjern et element fra mengden: `Remove(A, x)`
  - Operasjonene union, snitt og differanse:

$A \cup B$



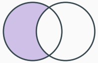
`Union(A,B)`

$A \cap B$



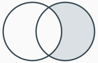
`Intersection(A,B)`

$A \setminus B$



`Difference(A,B)`

$B \setminus A$



`Difference(B,A)`

- Hverken rekkefølge eller antall forekomster spiller noen rolle i mengder
- Implementeres oftest som ordnede trær eller ved hashing

Set

- Den abstrakte datatypen for mengder kalles `Set`, hvor vi forventer å kunne
    - Sjekke om x forekommer i mengden: $x \in A$
    - Sett inn i mengden: `Insert(A, x)`
    - Fjern et element fra mengden: `Remove(A, x)`
    - Operasjonene union, snitt og differanse:

$A \cup B$ $\quad$ $A \cap B$ $\quad$ $A \setminus B$ $\quad$ $B \setminus A$

`Union(A,B)` `Intersection(A,B)` `Difference(A,B)` `Difference(B,A)`

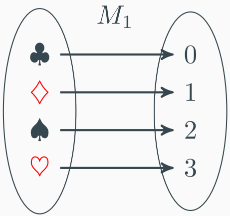
- Hverken rekkefølge eller antall forekomster spiller noen rolle i mengder
- Implementeres oftest som ordnede trær eller ved hashing
    - Java implementerer begge i henholdsvis `TreeSet` og `HashSet`

Ordbok

- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi

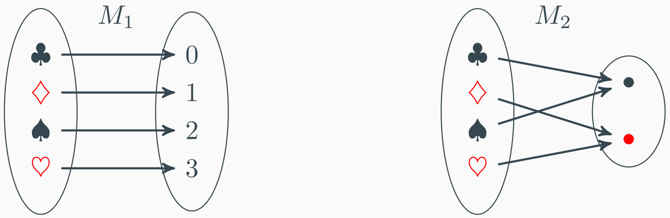
Ordbok

- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi



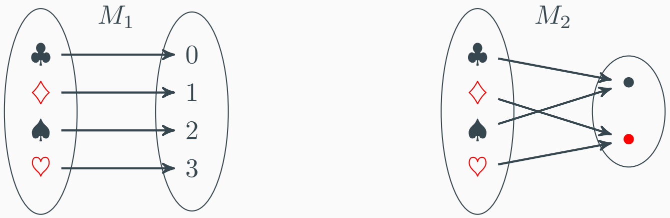
Ordbok

- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi



Ordbok

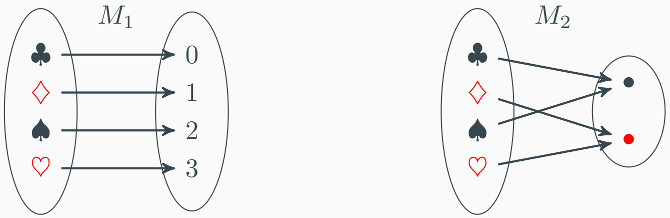
- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M

Ordbok

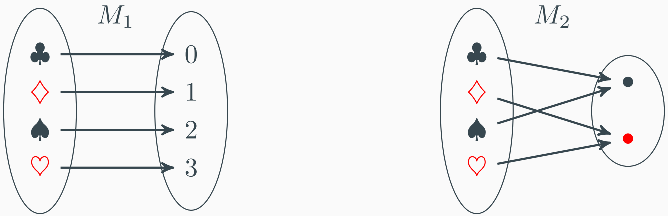
- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M
  - $M[k] \leftarrow v$ eller $\texttt{put}(M, k, v)$ Assosiere nøkkelen k med verdien v

Ordbok

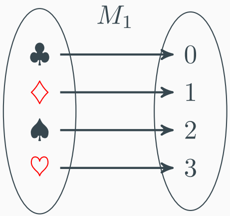
- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi

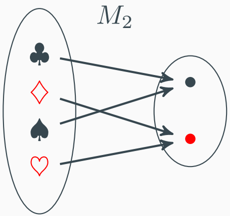


- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M
  - $M[k] \leftarrow v$ eller `put`(M, k, v) — Assosiere nøkkelen k med verdien v
  - $M[k]$ eller `get`(M, k) — Hente ut verdien som nøkkelen k er assosiert med

Ordbok

- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi

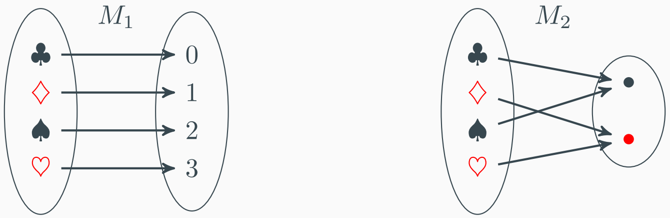




- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M
  - $M[k] \leftarrow v$ eller $\texttt{put}(M, k, v)$ — Assosiere nøkkelen k med verdien v
  - $M[k]$ eller $\texttt{get}(M, k)$ — Hente ut verdien som nøkkelen k er assosiert med
  - $M \setminus k$ eller $\texttt{Remove}(M, k)$ — Fjerne assosiasjonen forbundet med nøkkelen k

Ordbok

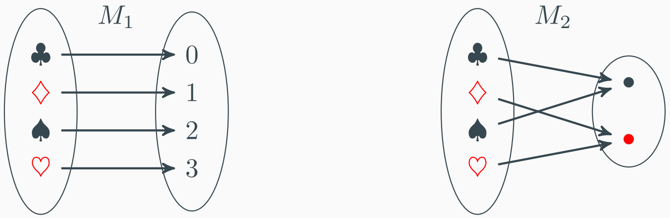
- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M
  - $M[k] \leftarrow v$ eller $\texttt{put}(M, k, v)$ — Assosiere nøkkelen k med verdien v
  - $M[k]$ eller $\texttt{get}(M, k)$ — Hente ut verdien som nøkkelen k er assosiert med
  - $M \setminus k$ eller $\texttt{Remove}(M, k)$ — Fjerne assosiasjonen forbundet med nøkkelen k
- Implementeres oftest som ordnede trær eller ved hashing

Ordbok

- En *ordbok*, eller et *map*, assosierer en nøkkel med nøyaktig én verdi



- Den abstrakte datatypen for ordbøker krever at vi kan for en gitt ordbok M
  - $M[k] \leftarrow v$ eller `put`(M, k, v) — Assosiere nøkkelen k med verdien v
  - $M[k]$ eller `get`(M, k) — Hente ut verdien som nøkkelen k er assosiert med
  - $M \setminus k$ eller `Remove`(M, k) — Fjerne assosiasjonen forbundet med nøkkelen k
- Implementeres oftest som ordnede trær eller ved hashing
  - Java implementerer begge i henholdsvis `TreeMap` og `HashMap`

Litt om pseudokode

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer
- Pseudokode bruker mekanismer og konvensjoner som man finner i de fleste programmeringsspråk

Litt om pseudokode

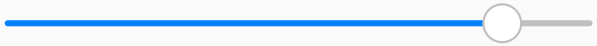
- Vi bruker pseudokode for å formidle algoritmer
- Pseudokode bruker mekanismer og konvensjoner som man finner i de fleste programmeringsspråk
- Pseudokoden skal være presis nok til at en kan enkelt oversette det til andre programmeringsspråk

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer
- Pseudokode bruker mekanismer og konvensjoner som man finner i de fleste programmeringsspråk
- Pseudokoden skal være presis nok til at en kan enkelt oversette det til andre programmeringsspråk
- Pseudokode er skrevet for en menneskelig leser, og ikke for en datamaskin

Litt om pseudokode

- Vi bruker pseudokode for å formidle algoritmer
- Pseudokode bruker mekanismer og konvensjoner som man finner i de fleste programmeringsspråk
- Pseudokoden skal være presis nok til at en kan enkelt oversette det til andre programmeringsspråk
- Pseudokode er skrevet for en menneskelig leser, og ikke for en datamaskin
- Slideren er posisjonert *veldig omtrentlig* der vi ønsker (ta det med en klype salt)

Naturlig språk  Python/Java

Notasjon

- Notasjonen vi bruker inkluderer:

Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$

Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$

Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$
  - Tilordninger $i \leftarrow 0$

Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$
  - Tilordninger $i \leftarrow 0$
  - Antall elementer i en datastruktur $|A|$

Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$
  - Tilordninger $i \leftarrow 0$
  - Antall elementer i en datastruktur $|A|$
  - While-løkker **`while`** `test` **`do`** body

Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$
  - Tilordninger $i \leftarrow 0$
  - Antall elementer i en datastruktur $|A|$
  - While-løkker **while** test **do** body
  - For-løkker **for** $i \leftarrow 0$ **to** $n - 1$ **do** body

Notasjon

- Notasjonen vi bruker inkluderer:
  - Aritmetiske uttrykk $a + b - \frac{c}{2}$
  - Sammenligninger $a \leq b$
  - Tilordninger $i \leftarrow 0$
  - Antall elementer i en datastruktur $|A|$
  - While-løkker **`while`** `test` **`do`** body
  - For-løkker **`for`** $i \leftarrow 0$ **`to`** $n - 1$ **`do`** body
- Pseudokoden vi skriver skal være lett å oversette til et programmeringsspråk

Søk – spesifikasjon

ALGORITHM: SØK (SPESIFIKASJON)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner **`true`** ellers **`false`**

Søk – spesifikasjon

ALGORITHM: SØK (SPESIFIKASJON)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon av et problem

Søk – spesifikasjon

ALGORITHM: SØK (SPESIFIKASJON)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon av et problem
- Problemet er å avgjøre om et element forekommer i et array eller ikke

Søk – spesifikasjon

Algorithm: Søk (spesifikasjon)

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon av et problem
- Problemet er å avgjøre om et element forekommer i et array eller ikke
- Vi har to input, et array A og et element x

Søk – spesifikasjon

ALGORITHM: SØK (SPESIFIKASJON)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner **`true`** ellers **`false`**

- Dette er en spesifikasjon av et problem
- Problemet er å avgjøre om et element forekommer i et array eller ikke
- Vi har to input, et array A og et element x
- Output er **`true`** eller **`false`**, avhengig av om x er med i A eller ikke

Søk – spesifikasjon

ALGORITHM: SØK (SPESIFIKASJON)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon av et problem
- Problemet er å avgjøre om et element forekommer i et array eller ikke
- Vi har to input, et array A og et element x
- Output er `true` eller `false`, avhengig av om x er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må *løse problemet*, altså

Søk – spesifikasjon

ALGORITHM: SØK (SPESIFIKASJON)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner **`true`** ellers **`false`**

- Dette er en spesifikasjon av et problem
- Problemet er å avgjøre om et element forekommer i et array eller ikke
- Vi har to input, et array A og et element x
- Output er **`true`** eller **`false`**, avhengig av om x er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må *løse problemet*, altså
    - terminere på et endelig antall steg

Søk – spesifikasjon

ALGORITHM: SØK (SPESIFIKASJON)

`Input:` Et array A og et element x
`Output:` Hvis x er i arrayet A, returner `true` ellers `false`

- Dette er en spesifikasjon av et problem
- Problemet er å avgjøre om et element forekommer i et array eller ikke
- Vi har to input, et array A og et element x
- Output er `true` eller `false`, avhengig av om x er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må *løse problemet*, altså
  - terminere på et endelig antall steg
  - gi riktig svar uansett hva A og x er

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

```
Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner true ellers false
Procedure Search(A, x)
```

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

```
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
```

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
```

Rett-frem søk – implementasjon

ALGORITHM: RETT FREM SØK

```
Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner true ellers false
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    return false
```

Rett-frem søk – implementasjon

ALGORITHM: RETT FREM SØK

Input: Et array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

Procedure Search(A, x)
    for $i \leftarrow 0$ **to** $|A| - 1$ **do**
        if $A[i] = x$ **then**
            return true
    return false

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide

Rett-frem søk – implementasjon

ALGORITHM: RETT FREM SØK

```
Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner true ellers false
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

```
Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner true ellers false
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
  - Det vil si vi må gjøre $|A|$ sammenligninger

Rett-frem søk – implementasjon

ALGORITHM: RETT FREM SØK

```
Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner true ellers false
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
    - Det vil si vi må gjøre $|A|$ sammenligninger
    - Her angir $|A|$ størrelsen på A

Rett-frem søk – implementasjon

ALGORITHM: RETT FREM SØK

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
  - Det vil si vi må gjøre $|A|$ sammenligninger
  - Her angir $|A|$ størrelsen på A
  - Vi bruker 0-indekserte arrayer

Rett-frem søk – implementasjon

Algorithm: Rett frem søk

Input: Et array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
    - Det vil si vi må gjøre $|A|$ sammenligninger
    - Her angir $|A|$ størrelsen på A
    - Vi bruker 0-indekserte arrayer
    - (Merk at boken bruker 1-indekserte arrayer)

Binærsøk – spesifikasjon

Algorithm: Binærsøk (spesifikasjon)

`Input:` Et ordnet array A og et element x
`Output:` Hvis x er i arrayet A, returner **`true`** ellers **`false`**

Binærsøk – spesifikasjon

ALGORITHM: BINÆRSØK (SPESIFIKASJON)

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner `true` ellers `false`

- Merk ordet *ordnet*

Binærsøk – spesifikasjon

ALGORITHM: BINÆRSØK (SPESIFIKASJON)

`Input:` Et ordnet array A og et element x
`Output:` Hvis x er i arrayet A, returner **`true`** ellers **`false`**

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |A|$, så $A[i] \leq A[j]$

Binærsøk – spesifikasjon

ALGORITHM: BINÆRSØK (SPESIFIKASJON)

`Input:` Et ordnet array A og et element x
`Output:` Hvis x er i arrayet A, returner **`true`** ellers **`false`**

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |A|$, så $A[i] \leq A[j]$
- Et rett-frem søk (som på forrige slide) vil fungere fint!

Binærsøk – spesifikasjon

ALGORITHM: BINÆRSØK (SPESIFIKASJON)

`Input:` Et ordnet array A og et element x
`Output:` Hvis x er i arrayet A, returner **`true`** ellers **`false`**

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |A|$, så $A[i] \leq A[j]$
- Et rett-frem søk (som på forrige slide) vil fungere fint!
- Ved å anta at arrayet er ordnet, kan vi finne på noe mye lurere

Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en ordliste

Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en ordliste
- Utfordringen er å formulere dette som en presis algoritme

Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en ordliste
- Utfordringen er å formulere dette som en presis algoritme
  - Altså oversette fremgangsmåten din til entydige steg

Binærsøk – eksempel 1

$$x = 2$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑ low						↑ high

Binærsøk – eksempel 1

$$x = 2$$

`1`	`2`	`4`	`6`	`7`	`9`	`10`
`0`	`1`	`2`	`3`	`4`	`5`	`6`
↑			↑			↑
`low`			`i`			`high`

Binærsøk – eksempel 1

$x = 2$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑ low		↑ high				

Binærsøk – eksempel 1

$$x = 2$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑ low	↑ i	↑ high				

Binærsøk – eksempel 2

$$x = 9$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑						↑
low						high

Binærsøk – eksempel 2

$$x = 9$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑			↑			↑
low			i			high

Binærsøk – eksempel 2

$$x = 9$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
				↑		↑
				low		high

Binærsøk – eksempel 2

$$x = 9$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
				↑	↑	↑
				low	i	high

Binærsøk – implementasjon

ALGORITHM: BINÆRSØK

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
```

Binærsøk – implementasjon

ALGORITHM: BINÆRSØK

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
    low ← 0
    high ← |A| − 1
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
    low ← 0
    high ← |A| − 1
    while low ≤ high do
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
    low ← 0
    high ← |A| − 1
    while low ≤ high do
        i ← ⌊(low+high)/2⌋
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
    low ← 0
    high ← |A| − 1
    while low ≤ high do
        i ← ⌊(low+high)/2⌋
        if A[i] = x then
            return true
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
    low ← 0
    high ← |A| − 1
    while low ≤ high do
        i ← ⌊(low+high)/2⌋
        if A[i] = x then
            return true
        else if A[i] < x then
            low ← i + 1
```

Binærsøk – implementasjon

Algorithm: Binærsøk

Input: Et ordnet array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
    low ← 0
    high ← |A| − 1
    while low ≤ high do
        i ← ⌊(low+high)/2⌋
        if A[i] = x then
            return true
        else if A[i] < x then
            low ← i + 1
        else if A[i] > x then
            high ← i − 1
```

Binærsøk – implementasjon

ALGORITHM: BINÆRSØK

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure BinarySearch(A, x)
    low ← 0
    high ← |A| − 1
    while low ≤ high do
        i ← ⌊(low+high)/2⌋
        if A[i] = x then
            return true
        else if A[i] < x then
            low ← i + 1
        else if A[i] > x then
            high ← i − 1
    return false
```

Binærsøk – eksempel 3

$$x = 5$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑ low						↑ high

Binærsøk – eksempel 3

$$x = 5$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑ low			↑ i			↑ high

Binærsøk – eksempel 3

$$x = 5$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑ low		↑ high				

Binærsøk – eksempel 3

$$x = 5$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
↑ low	↑ i	↑ high				

Binærsøk – eksempel 3

$$x = 5$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
		↑				
		low				
		high				

Binærsøk – eksempel 3

$$x = 5$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
		↑				
		low				
		i				
		high				

Binærsøk – eksempel 3

$$x = 5$$

1	2	4	6	7	9	10
0	1	2	3	4	5	6
		high	low			

Introduksjon til kjøretidskompleksitet

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
- Hvordan skiller vi mellom hastigheten til programmet og til maskinen det kjører på?

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
- Hvordan skiller vi mellom hastigheten til programmet og til maskinen det kjører på?
    - Det *samme programmet* kjører mye tregere på en gammel datamaskin enn en ny

Noe mer presist enn raskt

- Det er vanskelig å si om et program er *raskt* eller ikke
- Hvordan skiller vi mellom hastigheten til programmet og til maskinen det kjører på?
    - Det *samme programmet* kjører mye tregere på en gammel datamaskin enn en ny
- I stedet for å snakke om *hastighet* snakker vi om *effektivitet*

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
    - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
    - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
    - En algoritme kan ikke ha god ytelse i seg selv
    - Den kan gi opphav til god ytelse under ulike omstendigheter
- I dette kurset bryr vi oss primært om effektivitet, og lite om ytelse

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter
- I dette kurset bryr vi oss primært om effektivitet, og lite om ytelse
  - Det er fordi en effektiv løsning er effektiv for alltid

Effektivitet og ytelse

- Vi definerer effektivitet som å *ikke gjøre unødvendig arbeid*
  - En effektiv algoritme er en algoritme som ikke gjør mange unødvendige steg
- Vi definerer ytelse som *å utføre arbeidet raskt*
  - En algoritme kan ikke ha god ytelse i seg selv
  - Den kan gi opphav til god ytelse under ulike omstendigheter
- I dette kurset bryr vi oss primært om effektivitet, og lite om ytelse
  - Det er fordi en effektiv løsning er effektiv for alltid
  - For å snakke om ytelse trenger vi vite alt om maskinvaren, kompilatoren, temperaturen i rommet, og så videre. . .

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er *tid* og *minne*

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er *tid* og *minne*
- Kompleksitet defineres *relativt til størrelsen på input*

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er *tid* og *minne*
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er *tid* og *minne*
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er
    - Vi kaller dette *kjøretidskompleksiteten* til algoritmen

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er *tid* og *minne*
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er
  - Vi kaller dette *kjøretidskompleksiteten* til algoritmen
- Minne måles i hvor mange elementer som lagres i samlinger

Kompleksitet

- Kompleksiteten til en algoritme defineres av hvor mye ressurser den bruker
- De viktigste ressursene er *tid* og *minne*
- Kompleksitet defineres *relativt til størrelsen på input*
- Tid måles i antall steg algoritmen bruker i forhold til hvor stort input er
  - Vi kaller dette *kjøretidskompleksiteten* til algoritmen
- Minne måles i hvor mange elementer som lagres i samlinger
- I dette kurset fokuserer vi på kjøretidskompleksitet

Størrelsen av input

- Et input kan være stort eller lite

Størrelsen av input

- Et input kan være stort eller lite
    - Vi kan jobbe med en kort eller lang binærstreng

Størrelsen av input

- Et input kan være stort eller lite
    - Vi kan jobbe med en kort eller lang binærstreng
    - En liste med få eller mange noder

Størrelsen av input

- Et input kan være stort eller lite
    - Vi kan jobbe med en kort eller lang binærstreng
    - En liste med få eller mange noder
    - Et array med få eller mange elementer

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi
  - $\vdots$

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi
  - $\vdots$
- Størrelsen på input angis konvensjonelt ved en variabel n

Størrelsen av input

- Et input kan være stort eller lite
  - Vi kan jobbe med en kort eller lang binærstreng
  - En liste med få eller mange noder
  - Et array med få eller mange elementer
  - En mengde med få eller mange elementer
  - Et tall med lav eller høy verdi
  - $\vdots$
- Størrelsen på input angis konvensjonelt ved en variabel n
  - Merk at det ikke er noe spesielt med n

Telle steg

- Vi anser følgende som *primitive steg*

Telle steg

- Vi anser følgende som *primitive steg*
    - Tilordninger $x \leftarrow 3$

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **`return`** a

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **`return`** a
- Listen over er ikke nødvendigvis fullstendig

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **`return`** a
- Listen over er ikke nødvendigvis fullstendig
- De koster alle 1 i tidsbruk

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **`return`** a
- Listen over er ikke nødvendigvis fullstendig
- De koster alle 1 i tidsbruk
- En **`while`**-løkke arver kostnaden fra testen og kroppen **`while`** `test` **`do`** body

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **`return`** a
- Listen over er ikke nødvendigvis fullstendig
- De koster alle 1 i tidsbruk
- En **`while`**-løkke arver kostnaden fra testen og kroppen **`while`** `test` **`do`** body
  - Der kostnaden ganges med antall iterasjoner

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **`return`** a
- Listen over er ikke nødvendigvis fullstendig
- De koster alle 1 i tidsbruk
- En **`while`**-løkke arver kostnaden fra testen og kroppen **`while`** `test` **`do`** body
  - Der kostnaden ganges med antall iterasjoner
- Tilsvarende for **`for`**-løkker

Telle steg

- Vi anser følgende som *primitive steg*
  - Tilordninger $x \leftarrow 3$
  - Aritmetiske operasjoner $a + b$
  - Sammenligninger $a < b$
  - Aksessering på index i arrayer $A[i]$
  - Returnering **`return`** a
- Listen over er ikke nødvendigvis fullstendig
- De koster alle 1 i tidsbruk
- En **`while`**-løkke arver kostnaden fra testen og kroppen **`while`** `test` **`do`** body
  - Der kostnaden ganges med antall iterasjoner
- Tilsvarende for **`for`**-løkker
- Kostnaden arves også fra prosedyre- og metodekall

En enkel omskrivning

- Den samme koden som tidligere, der **for** byttes med **while**

```
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    return false
```

$\rightsquigarrow$

```
Procedure Search(A, x)
    i ← 0
    while i < |A| do
        if A[i] = x then
            return true
        i ← i + 1
    return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet

```
Procedure Search(A, x)
    i ← 0
    while i < |A| do
        if A[i] = x then
            return true
        i ← i + 1
    return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n

```
Procedure Search(A, x)
    i ← 0
    while i < |A| do
        if A[i] = x then
            return true
        i ← i + 1
    return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg

```
Procedure Search(A, x)
  i ← 0
  while i < |A| do
    if A[i] = x then
      return true
    i ← i + 1
  return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **`while`**-løkken kjøres $n + 1$ ganger

```
Procedure Search(A, x)
  i ← 0
  while i < |A| do
    if A[i] = x then
      return true
    i ← i + 1
  return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **`while`**-løkken kjøres $n + 1$ ganger
    - Hver gang gjøres det én sammenligning, som er ett primitivt steg

```
Procedure Search(A, x)
  i ← 0
  while i < |A| do
    if A[i] = x then
      return true
    i ← i + 1
  return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **`while`**-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, som er ett primitivt steg
- **`while`**-løkken vil ha n iterasjoner:

```
Procedure Search(A, x)
  i ← 0
  while i < |A| do
    if A[i] = x then
      return true
    i ← i + 1
  return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **`while`**-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, som er ett primitivt steg
- **`while`**-løkken vil ha n iterasjoner:
  - I **`if`**-testen sammenlignes $A[i]$ med x

```
Procedure Search(A, x)
    i ← 0
    while i < |A| do
        if A[i] = x then
            return true
        i ← i + 1
    return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **`while`**-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, som er ett primitivt steg
- **`while`**-løkken vil ha n iterasjoner:
  - I **`if`**-testen sammenlignes $A[i]$ med x
    - En aksessering og en sammenligning utgjør 2 primitive steg

```
Procedure Search(A, x)
  i ← 0
  while i < |A| do
    if A[i] = x then
      return true
    i ← i + 1
  return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **while**-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, som er ett primitivt steg
- **while**-løkken vil ha n iterasjoner:
  - I **if**-testen sammenlignes $A[i]$ med x
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes i med én

Procedure Search(A, x)
  $i \leftarrow 0$
  while $i < |A|$ **do**
    if $A[i] = x$ **then**
      return **true**
    $i \leftarrow i + 1$
  return **false**

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **while**-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, som er ett primitivt steg
- **while**-løkken vil ha n iterasjoner:
  - I **if**-testen sammenlignes $A[i]$ med x
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes i med én
    - En tilordning og en addisjon utgjør 2 primitive steg

```
Procedure Search(A, x)
    i ← 0
    while i < |A| do
        if A[i] = x then
            return true
        i ← i + 1
    return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **while**-løkken kjøres $n + 1$ ganger
    - Hver gang gjøres det én sammenligning, som er ett primitivt steg
- **while**-løkken vil ha n iterasjoner:
    - I **if**-testen sammenlignes $A[i]$ med x
        - En aksessering og en sammenligning utgjør 2 primitive steg
    - I hver iterasjon økes i med én
        - En tilordning og en addisjon utgjør 2 primitive steg
- Til slutt returnerer vi, som utgjør 1 primitivt steg

```
Procedure Search(A, x)
    i ← 0
    while i < |A| do
        if A[i] = x then
            return true
        i ← i + 1
    return false
```

Telle steg (eksempel)

- La $n = |A|$ og anta at x *ikke* er i arrayet
- Vi ønsker å finne antall steg algoritmen bruker som et uttrykk av n
- Variabelen i tilordnes en verdi; det utgjør 1 primitivt steg
- Testen til **while**-løkken kjøres $n + 1$ ganger
  - Hver gang gjøres det én sammenligning, som er ett primitivt steg
- **while**-løkken vil ha n iterasjoner:
  - I **if**-testen sammenlignes $A[i]$ med x
    - En aksessering og en sammenligning utgjør 2 primitive steg
  - I hver iterasjon økes i med én
    - En tilordning og en addisjon utgjør 2 primitive steg
- Til slutt returnerer vi, som utgjør 1 primitivt steg
- Samlet antall steg blir $5 \cdot n + 3$, fordi $1 + (n + 1) + n \cdot (2 + 2) + 1 = 3 + n + n \cdot 4 = 5 \cdot n + 3$

```
Procedure Search(A, x)
  i ← 0
  while i < |A| do
    if A[i] = x then
      return true
    i ← i + 1
  return false
```

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke for input av en gitt størrelse

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
    - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke for input av en gitt størrelse
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke for input av en gitt størrelse
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre
  - Vi ville trengt sannsynlighetsfordelingen for alle mulig input

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke for input av en gitt størrelse
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre
  - Vi ville trengt sannsynlighetsfordelingen for alle mulig input
- Med verste tilfelle kan vi finne trygget i at det ikke kan bli verre enn verst!

Verste tilfelle

- Som regel ønsker vi en *verste tilfelle* analyse
  - Det vil si at vi finner det høyeste antall steg algoritmen *kan* bruke for input av en gitt størrelse
- En analyse av gjennomsnittet er mye vanskeligere å gjennomføre
  - Vi ville trengt sannsynlighetsfordelingen for alle mulig input
- Med verste tilfelle kan vi finne trygget i at det ikke kan bli verre enn verst!
- Det viktigste spørsmålet er: Hvordan utvikler kjøretiden seg når input blir stort?

Stor $\mathcal{O}$

- Vi bruker et verktøy, kalt stor $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet

Stor $\mathcal{O}$

- Vi bruker et verktøy, kalt stor $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for stor $\mathcal{O}$ er å se for seg at input blir *veldig* stort

Stor $\mathcal{O}$

- Vi bruker et verktøy, kalt stor $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for stor $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanterfaktorer om til 1:

$$\mathcal{O}(4 \cdot n^2 + 5 \cdot n + 3) = \mathcal{O}(1 \cdot n^2 + 1 \cdot n + 1) = \mathcal{O}(n^2 + n + 1)$$

Stor $\mathcal{O}$

- Vi bruker et verktøy, kalt stor $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for stor $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanterfaktorer om til 1:

$$\mathcal{O}(4 \cdot n^2 + 5 \cdot n + 3) = \mathcal{O}(1 \cdot n^2 + 1 \cdot n + 1) = \mathcal{O}(n^2 + n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n^2 + n + 1) = \mathcal{O}(n^2)$$

Stor $\mathcal{O}$

- Vi bruker et verktøy, kalt stor $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for stor $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanterfaktorer om til 1:

$$\mathcal{O}(4 \cdot n^2 + 5 \cdot n + 3) = \mathcal{O}(1 \cdot n^2 + 1 \cdot n + 1) = \mathcal{O}(n^2 + n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n^2 + n + 1) = \mathcal{O}(n^2)$$

- Det lar oss regne *mye* grovere

Stor $\mathcal{O}$

- Vi bruker et verktøy, kalt stor $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for stor $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanterfaktorer om til 1:

$$\mathcal{O}(4 \cdot n^2 + 5 \cdot n + 3) = \mathcal{O}(1 \cdot n^2 + 1 \cdot n + 1) = \mathcal{O}(n^2 + n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n^2 + n + 1) = \mathcal{O}(n^2)$$

- Det lar oss regne *mye* grovere
- Og bevare hovedtrenden i hvordan kjøretiden vokser når input blir stor

Stor $\mathcal{O}$

- Vi bruker et verktøy, kalt stor $\mathcal{O}$-notasjon, for å uttrykke kjøretidskompleksitet
- Intuisjonen for stor $\mathcal{O}$ er å se for seg at input blir *veldig* stort
- Da tillater vi oss å gjøre alle konstanterfaktorer om til 1:

$$\mathcal{O}(4 \cdot n^2 + 5 \cdot n + 3) = \mathcal{O}(1 \cdot n^2 + 1 \cdot n + 1) = \mathcal{O}(n^2 + n + 1)$$

- Og å stryke alt bortsett fra det *største leddet*:

$$\mathcal{O}(n^2 + n + 1) = \mathcal{O}(n^2)$$

- Det lar oss regne *mye* grovere
- Og bevare hovedtrenden i hvordan kjøretiden vokser når input blir stor
- Vi vil med *få* unntak foretrekke en algoritme med lavere kjøretidskompleksitet

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

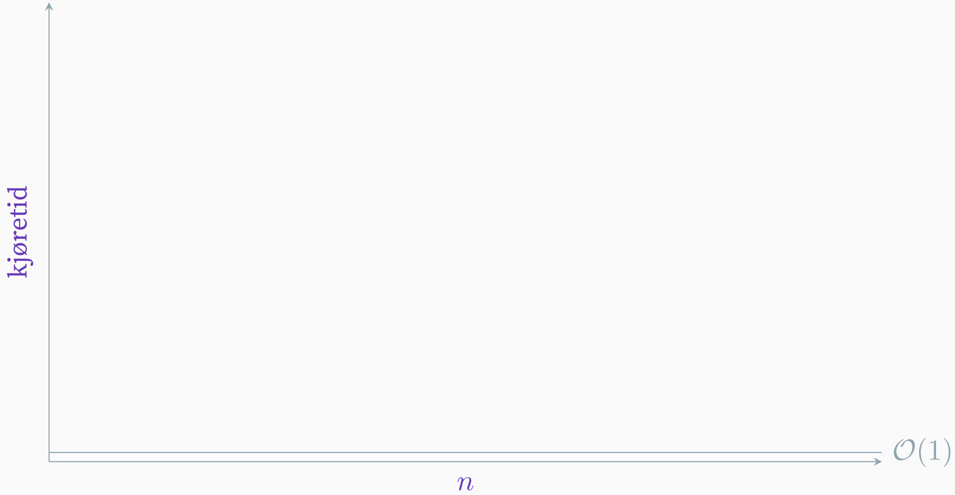
Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid

Vanlige uttrykk for kjøretidskompleksitet

- De fleste algoritmene vi ser på i dette kurset faller inn under en av disse kategoriene

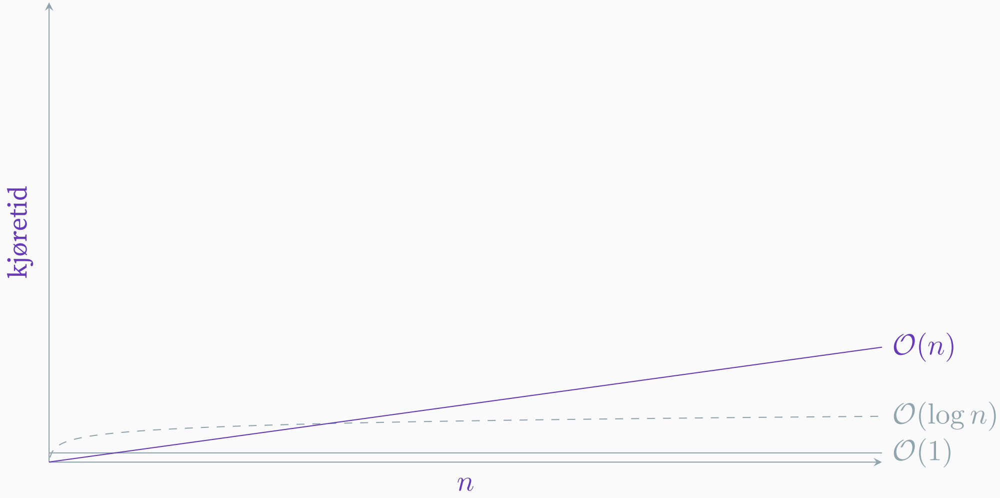
Notasjon	Uttrykk
$\mathcal{O}(1)$	Konstant tid
$\mathcal{O}(\log(n))$	Logaritmisk tid
$\mathcal{O}(n)$	Lineær tid
$\mathcal{O}(n \cdot \log(n))$	Lineæritmisk tid
$\mathcal{O}(n^2)$	kvadratisk tid
$\vdots$	
$\mathcal{O}(n^k)$	polynomiell tid
$\mathcal{O}(2^n)$	eksponensiell tid

Vanlige uttrykk for kjøretidskompleksitet (graf)

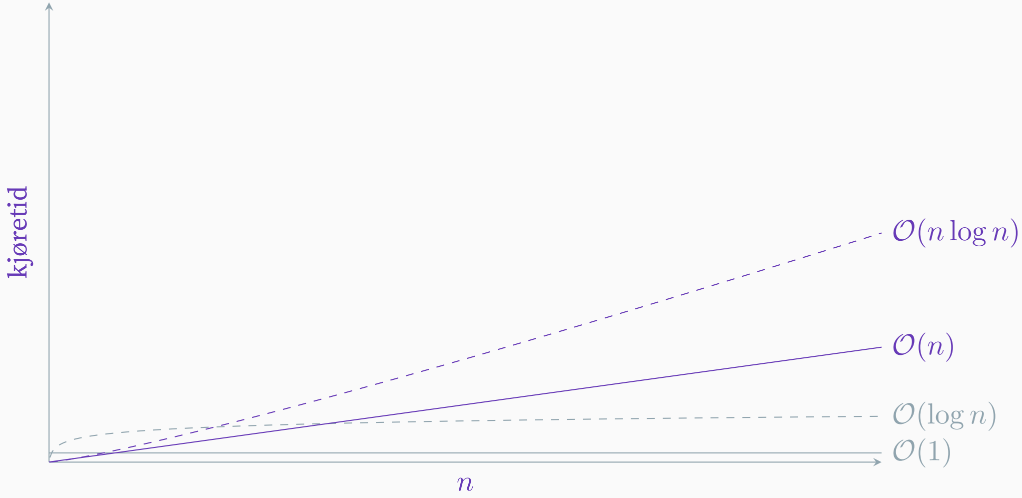


Vanlige uttrykk for kjøretidskompleksitet (graf)

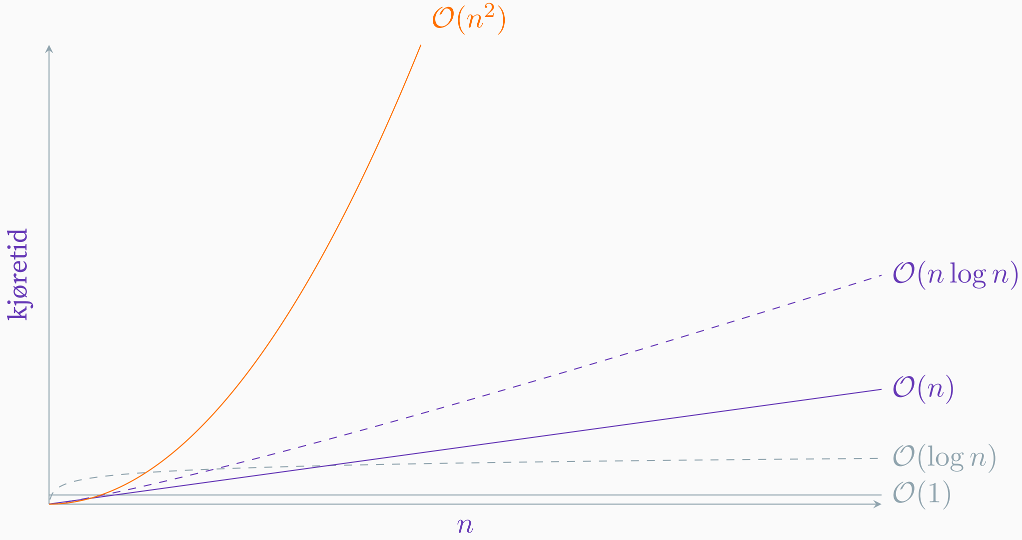
Vanlige uttrykk for kjøretidskompleksitet (graf)



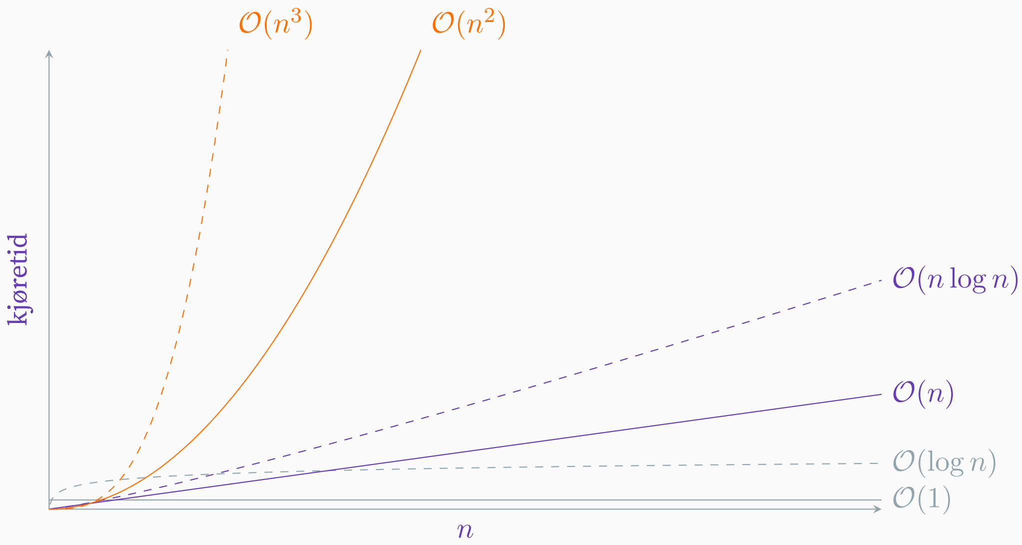
Vanlige uttrykk for kjøretidskompleksitet (graf)



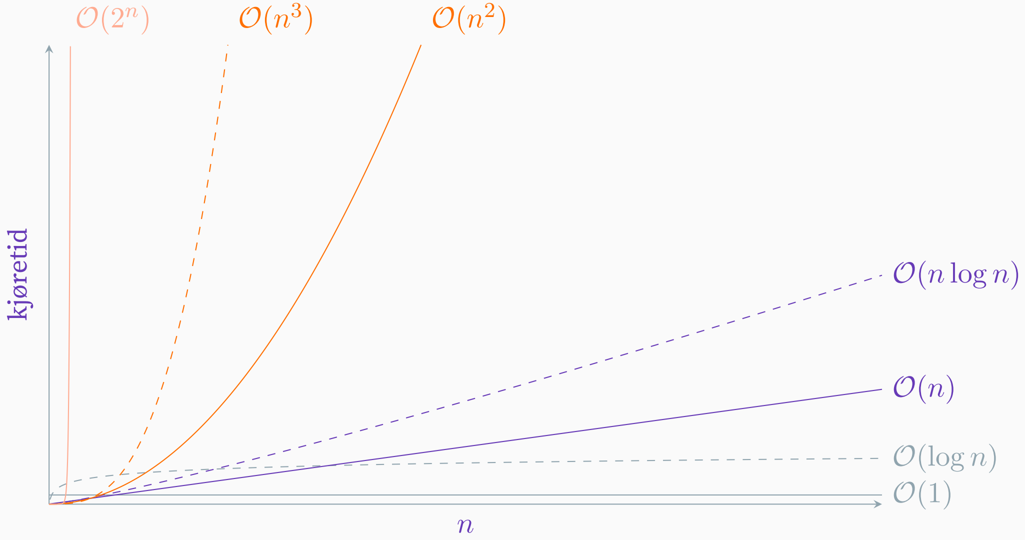
Vanlige uttrykk for kjøretidskompleksitet (graf)



Vanlige uttrykk for kjøretidskompleksitet (graf)



Vanlige uttrykk for kjøretidskompleksitet (graf)



Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
Procedure Constant(n)
    return n · 3                                  // O(1)
```

Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
Procedure Constant(n)
    return n · 3                                // O(1)
```

```
Procedure Log(n)
    i ← n
    while i > 0 do
        Constant(i)
        i ← ⌊i/2⌋                               // O(log(n))
```

Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
Procedure Constant(n)
  | return n · 3                                   // O(1)
```

```
Procedure Log(n)
  | i ← n
  | while i > 0 do
  |   | Constant(i)
  |   | i ← ⌊i/2⌋                                  // O(log(n))
```

```
Procedure Linear(n)
  | for i ← 0 to n − 1 do
  |   | Constant(i)                                // O(n)
```

Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
Procedure Constant(n)
    return n · 3                              // O(1)
```

```
Procedure Log(n)
    i ← n
    while i > 0 do
        Constant(i)
        i ← ⌊i/2⌋                             // O(log(n))
```

```
Procedure Linear(n)
    for i ← 0 to n − 1 do
        Constant(i)                           // O(n)
```

```
Procedure Linearithmic(n)
    for i ← 0 to n − 1 do
        Log(n)                                // O(n · log(n))
```

Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
Procedure Constant(n)
    return n · 3                                // O(1)
```

```
Procedure Log(n)
    i ← n
    while i > 0 do
        Constant(i)
        i ← ⌊i/2⌋                               // O(log(n))
```

```
Procedure Linear(n)
    for i ← 0 to n − 1 do
        Constant(i)                             // O(n)
```

```
Procedure Linearithmic(n)
    for i ← 0 to n − 1 do
        Log(n)                                  // O(n · log(n))
```

```
Procedure Quadratic(n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            Constant(i)                         // O(n^2)
```

Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
Procedure Constant(n)
    return n · 3                                  // O(1)
```

```
Procedure Log(n)
    i ← n
    while i > 0 do
        Constant(i)
        i ← ⌊i/2⌋                                 // O(log(n))
```

```
Procedure Linear(n)
    for i ← 0 to n − 1 do
        Constant(i)                               // O(n)
```

```
Procedure Linearithmic(n)
    for i ← 0 to n − 1 do
        Log(n)                                    // O(n · log(n))
```

```
Procedure Quadratic(n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            Constant(i)                           // O(n^2)
```

```
Procedure Polynomial(n)
    for i_1 ← 0 to n − 1 do
        for i_2 ← 0 to n − 1 do
            ⋮
            for i_k ← 0 to n − 1 do
                Constant(i)                       // O(n^k)
```

Vanlige uttrykk for kjøretidskompleksitet (kodeeksempler)

```
Procedure Constant(n)
    return n · 3                                  // O(1)
```

```
Procedure Log(n)
    i ← n
    while i > 0 do
        Constant(i)
        i ← ⌊i/2⌋                                 // O(log(n))
```

```
Procedure Linear(n)
    for i ← 0 to n − 1 do
        Constant(i)                               // O(n)
```

```
Procedure Linearithmic(n)
    for i ← 0 to n − 1 do
        Log(n)                                    // O(n · log(n))
```

```
Procedure Quadratic(n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            Constant(i)                           // O(n^2)
```

```
Procedure Polynomial(n)
    for i_1 ← 0 to n − 1 do
        for i_2 ← 0 to n − 1 do
            ⋮
            for i_k ← 0 to n − 1 do
                Constant(i)                       // O(n^k)
```

```
Procedure Exponential(n)
    if n = 0 then
        return 1
    a ← Exponential(n − 1)
    b ← Exponential(n − 1)
    return a + b                                  // O(2^n)
```