Midtsemesterforelesning

Algdat 2020

Datastrukturer :3

HVa gjør en datastruktur?

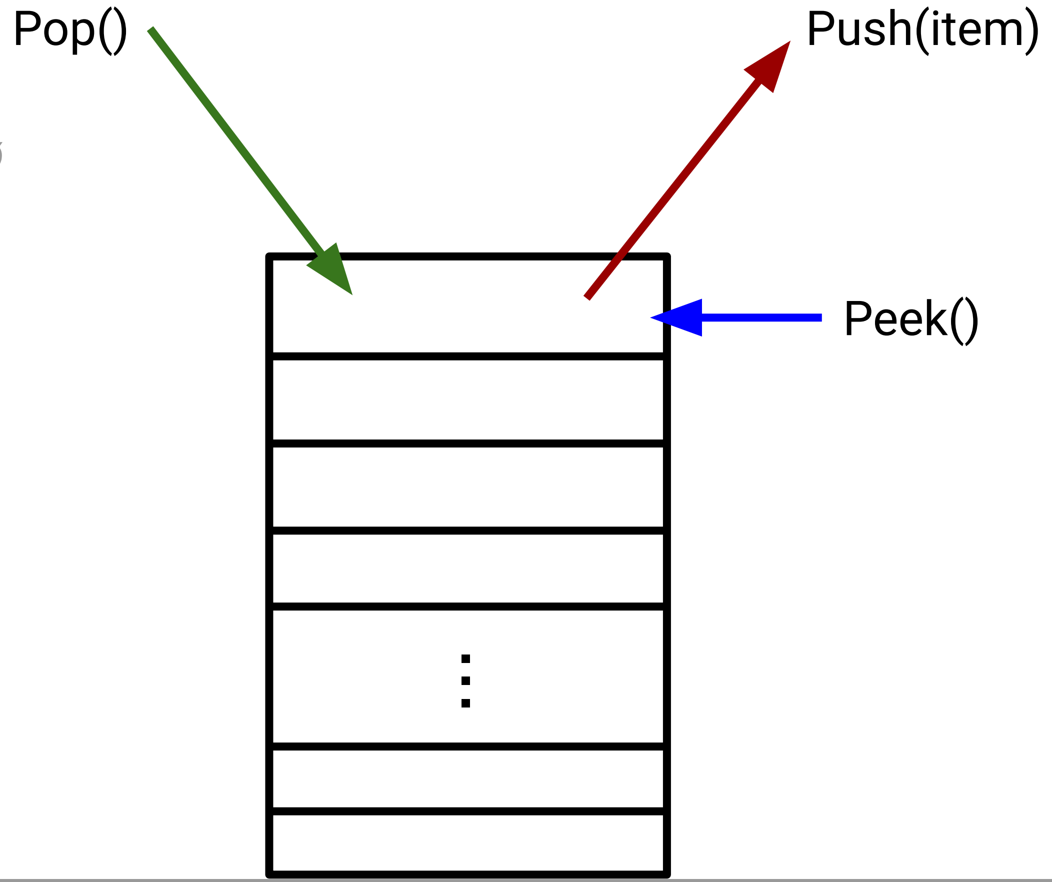
- Representere data

HVORDAN!?
I feks:

- Prioritetskøer, Trær, Lenkede lister, hash-maps… osv.

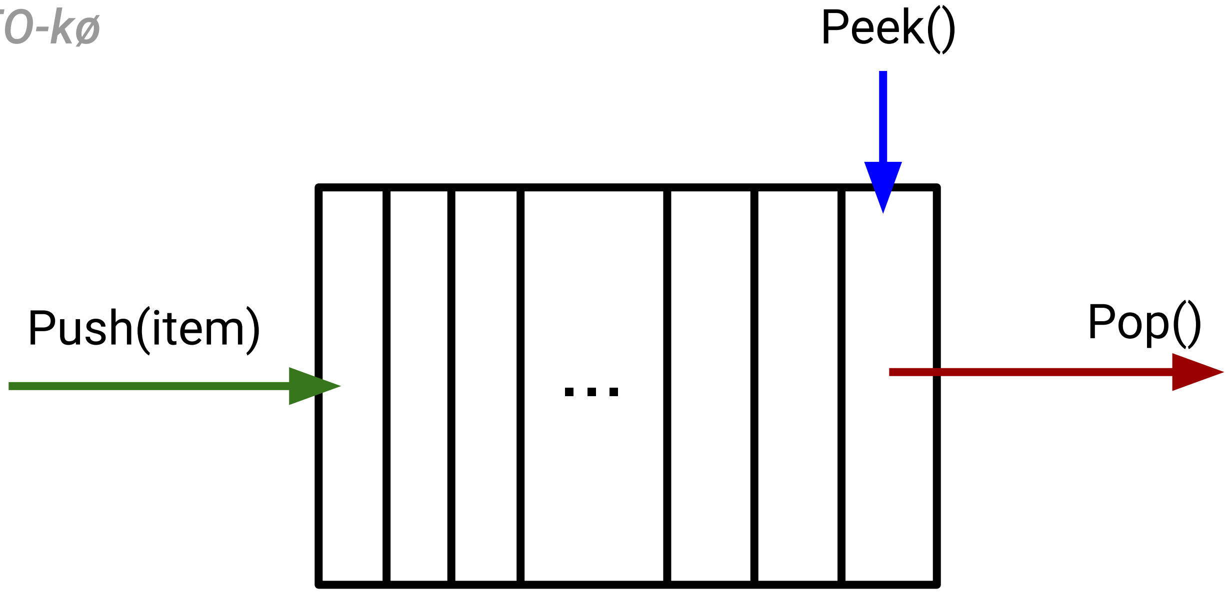
Stack

eller LIFO-kø

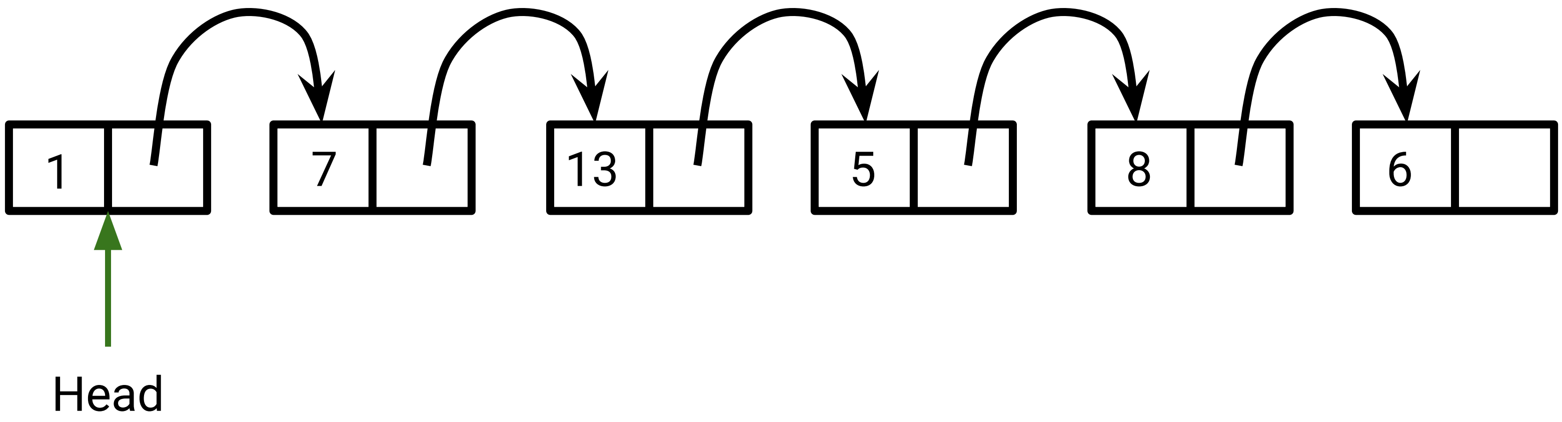


Queue

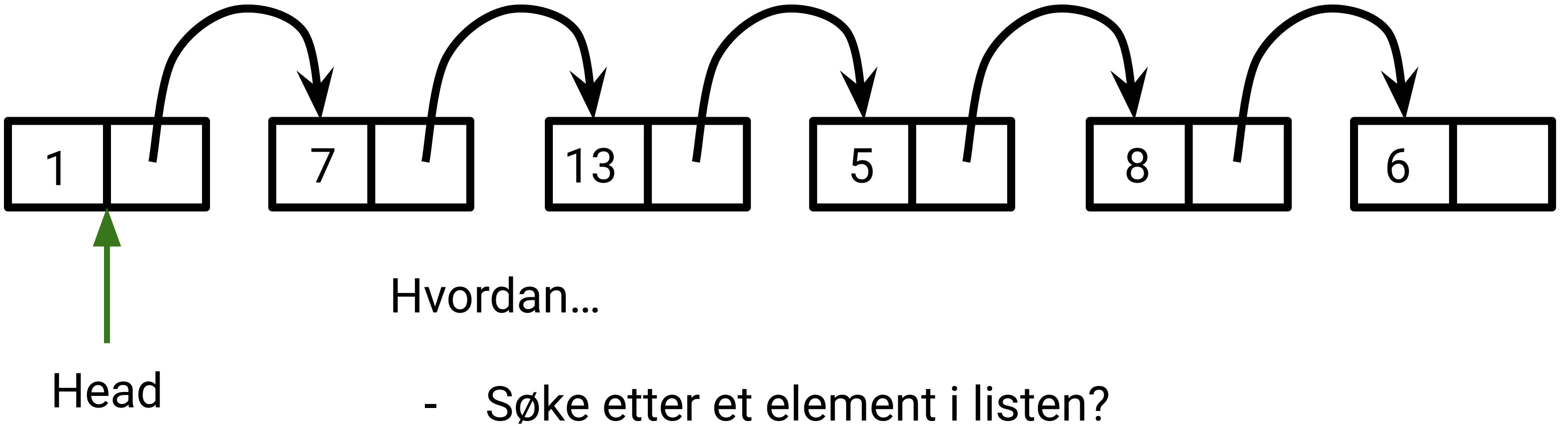
eller FIFO-kø



Linked list



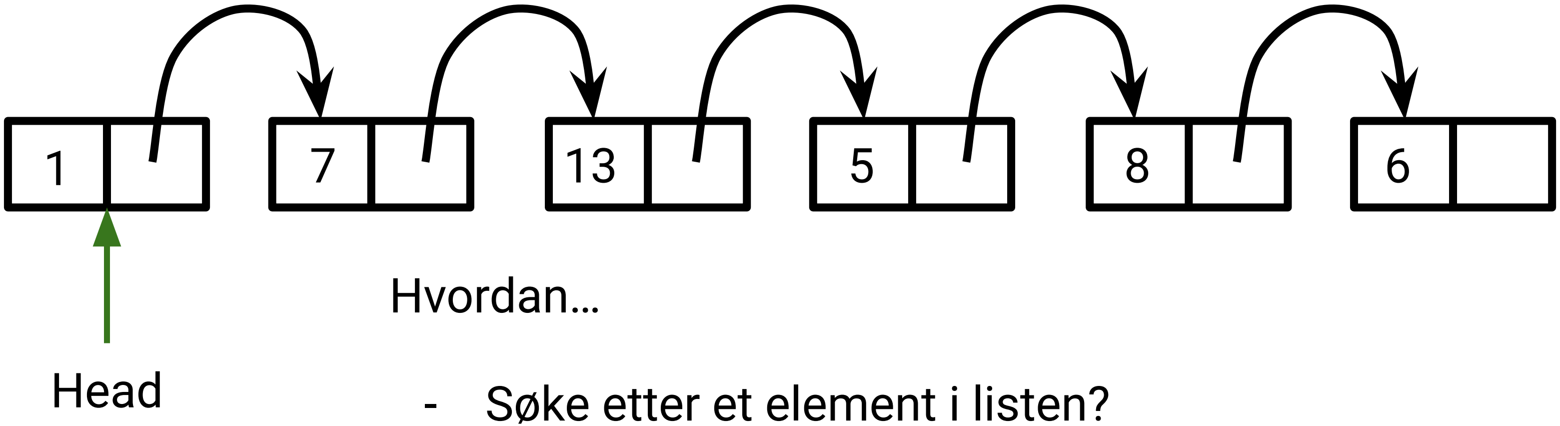
Linked list



Hvordan…

- Søke etter et element i listen?

Linked list

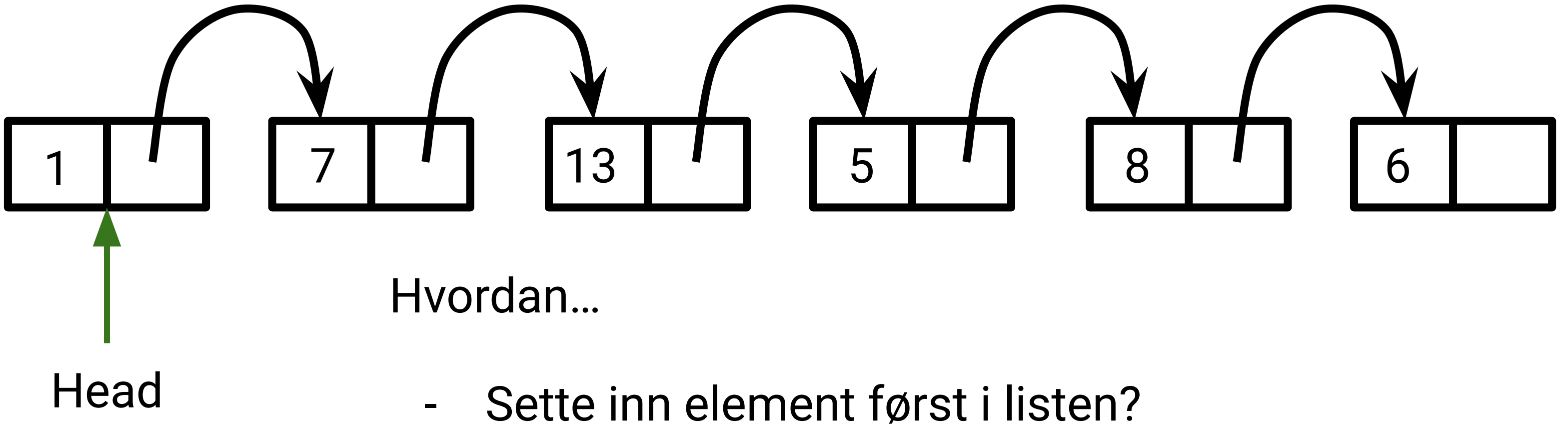


Hvordan…

- Søke etter et element i listen?

Begynne fra starten → lineært søk
Kjøretid: O(n)

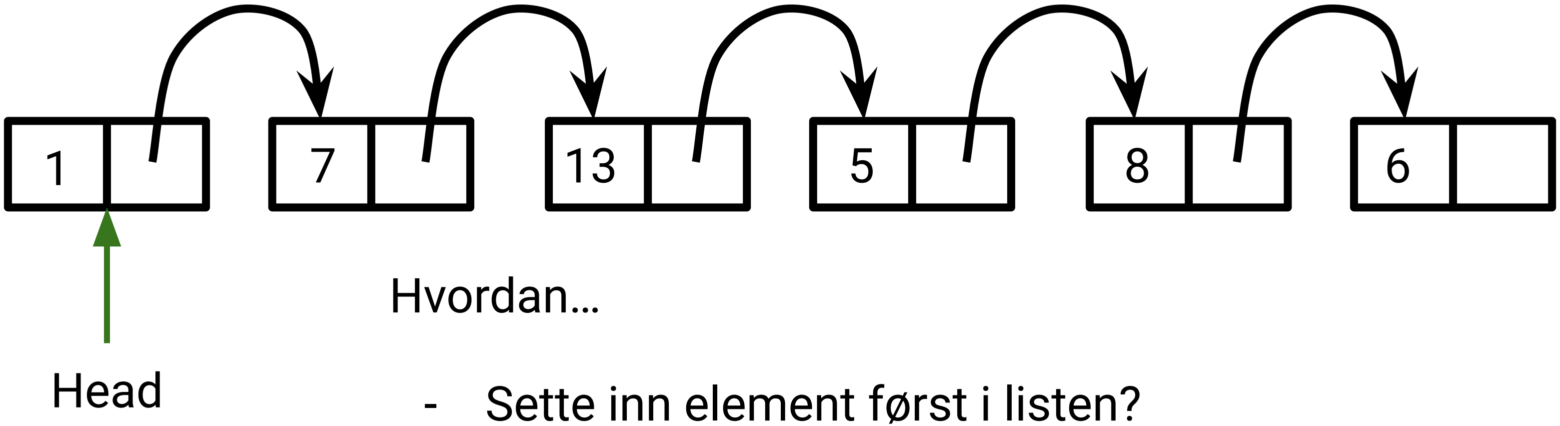
Linked list



Hvordan…

- Sette inn element først i listen?

Linked list

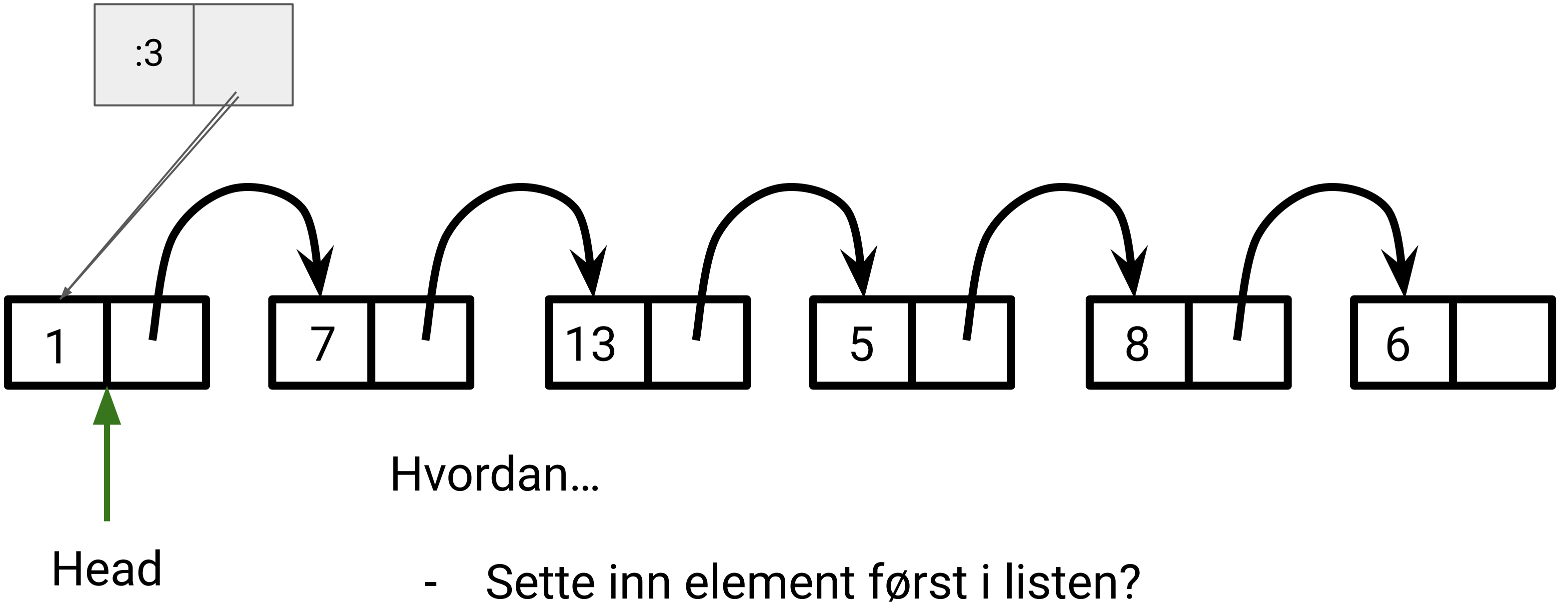


Hvordan…

- Sette inn element først i listen?

Sett inn element → Sett peker til neste element i listen
Kjøretid: O(1)

Linked list

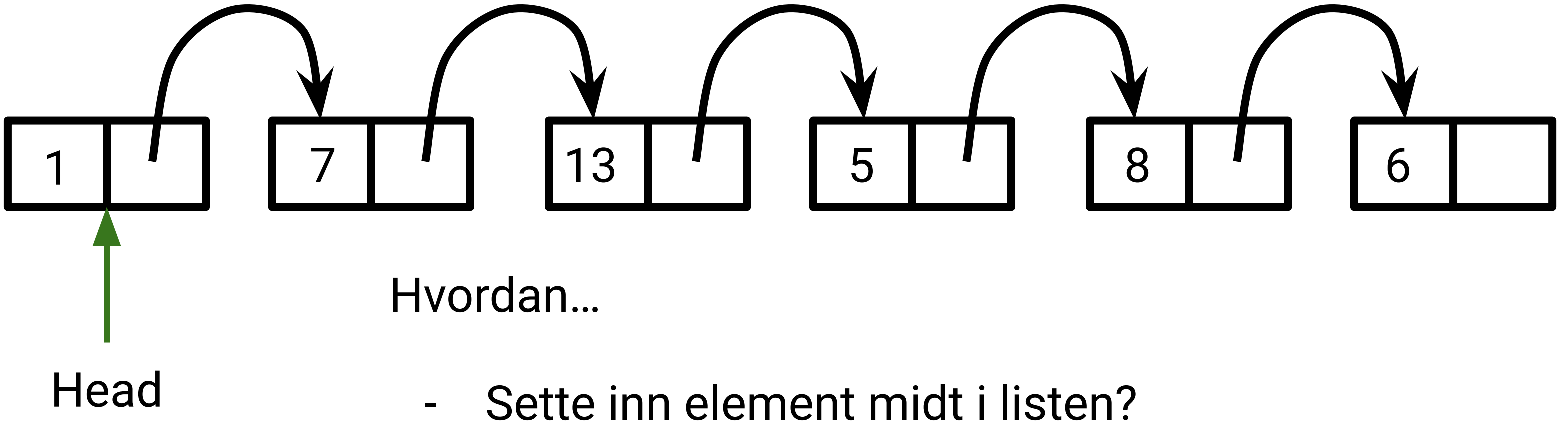


Hvordan…

- Sette inn element først i listen?

Sett inn element → Sett peker til neste element i listen
Kjøretid: O(1)

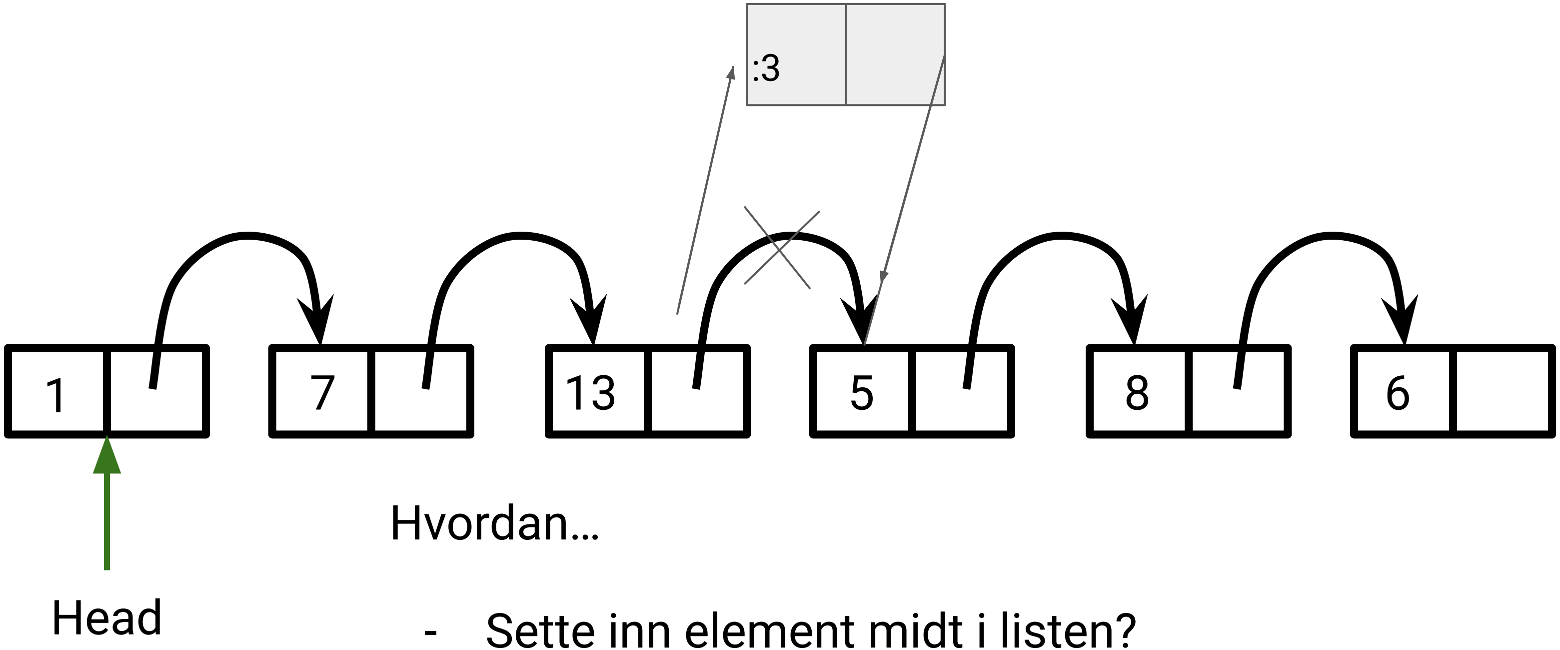
Linked list



Hvordan…

- Sette inn element midt i listen?

Linked list

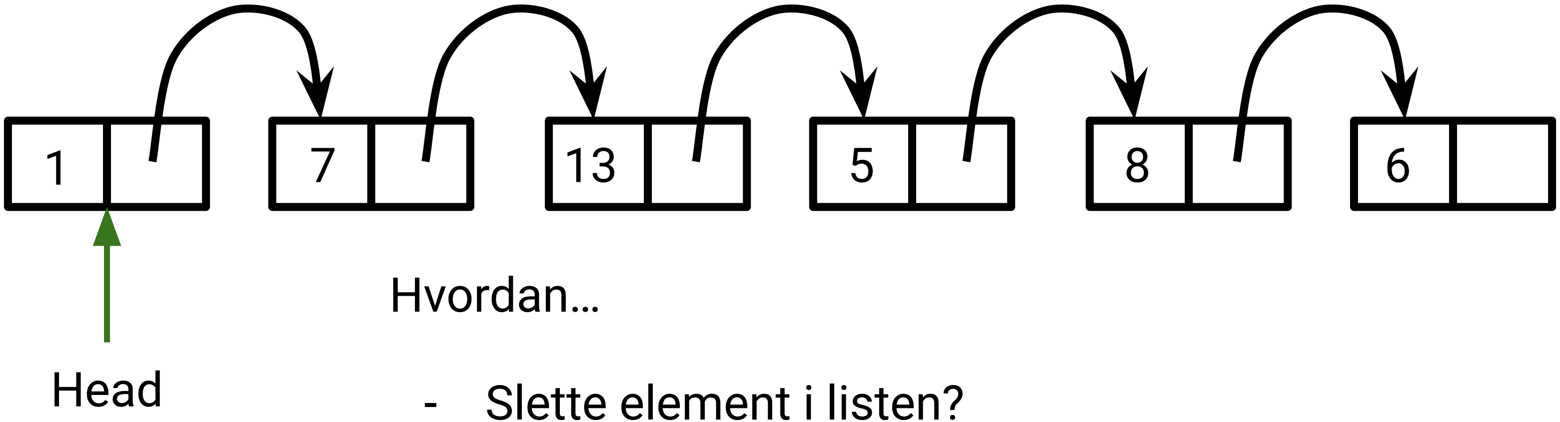


Hvordan...

- Sette inn element midt i listen?

Søke seg frem → oppdatere lenker → sette inn element
Kjøretid: O(n)

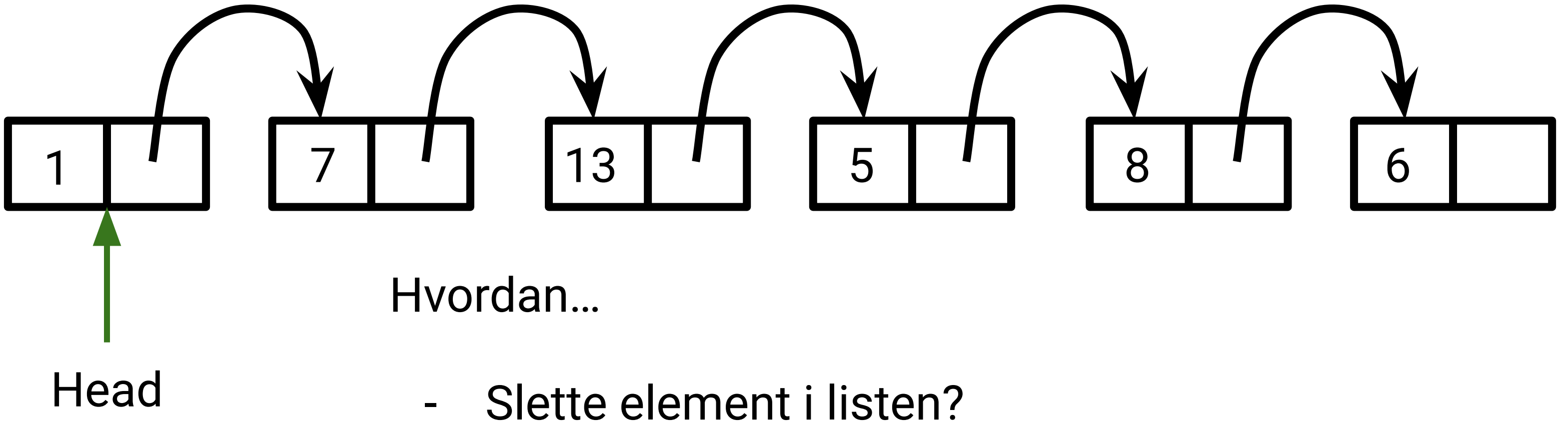
Linked list



Hvordan…

- Slette element i listen?

Linked list

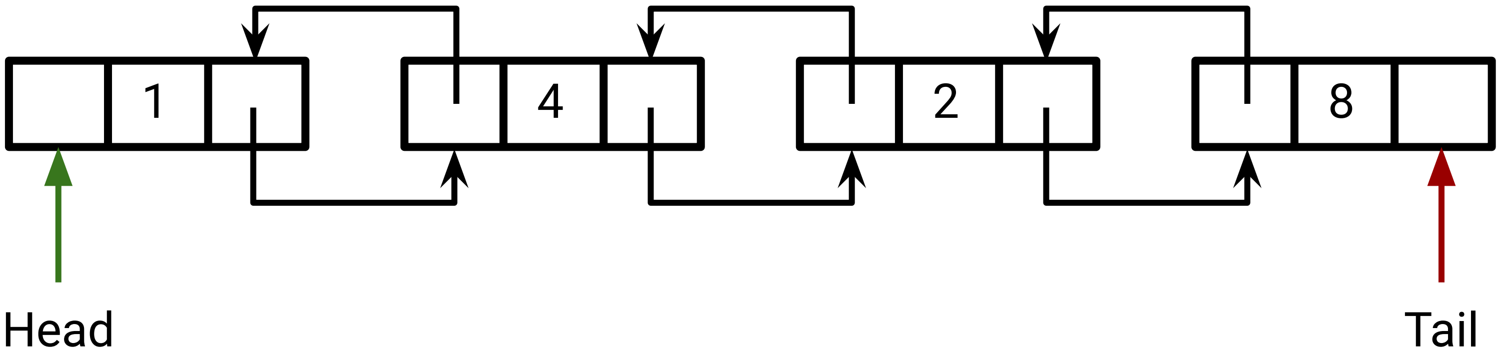


Hvordan…

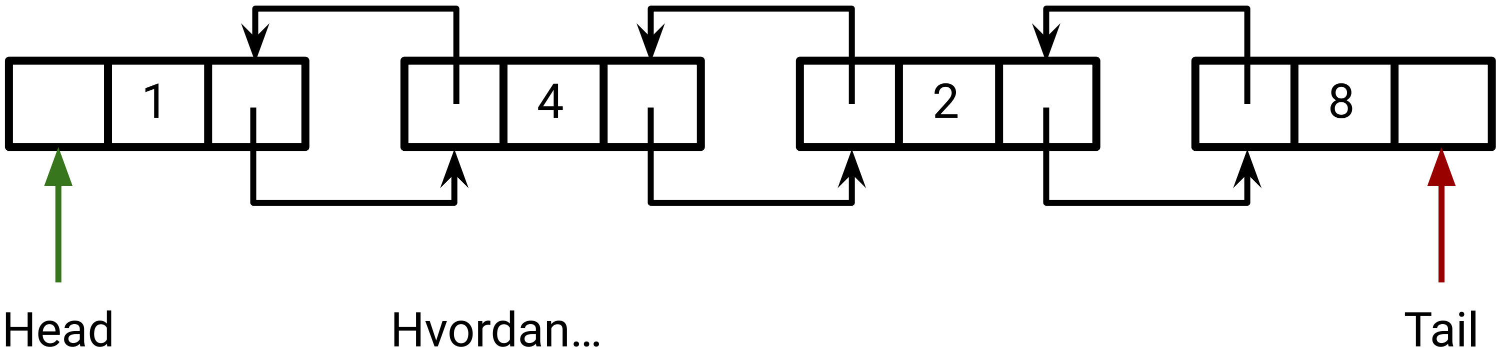
- Slette element i listen?

Søke seg frem → oppdatere lenke ved å hoppe over et element
Kjøretid: O(n)

Doubly linked list



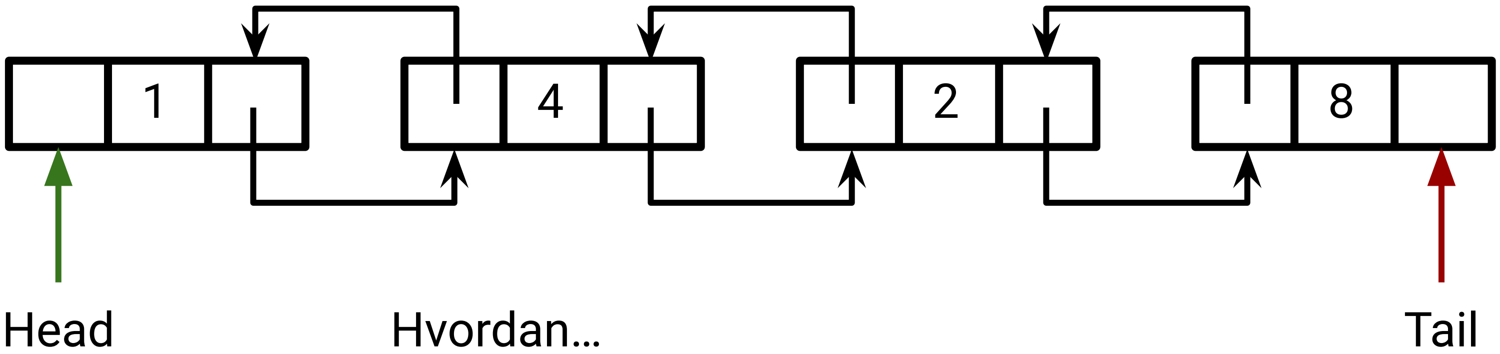
Doubly linked list



Hvordan…

- Slette element i listen?

Doubly linked list

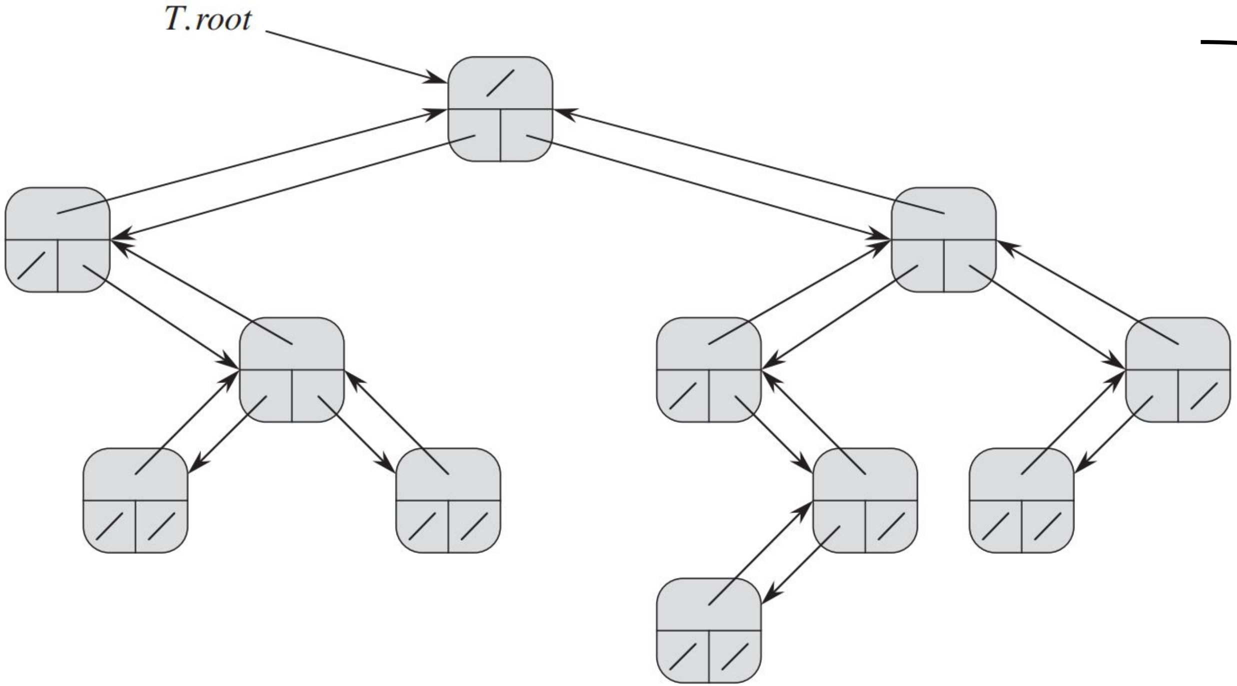


Hvordan...

- Slette element i listen?

Oppdatere naboene ved å lenke dem til hverandre
Kjøretid: O(1)

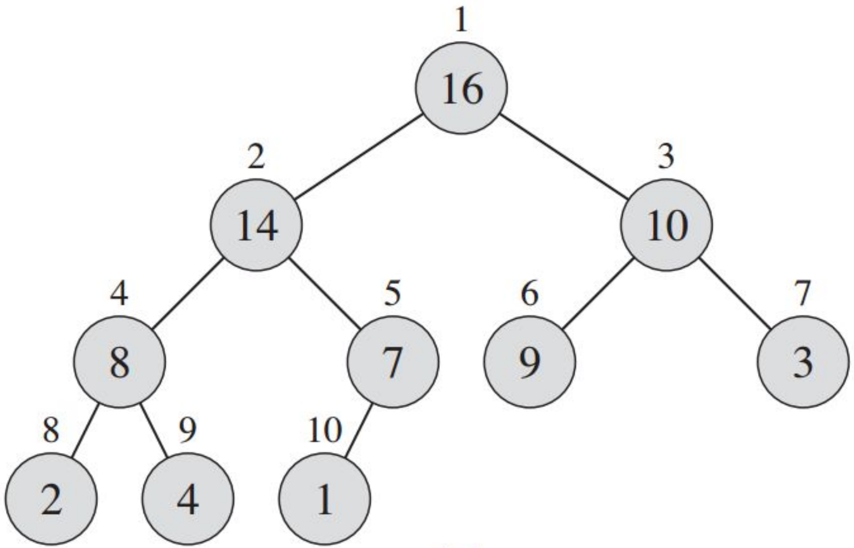
Trær



log(n) høyde

https://www.youtube.com/watch?v=t0Cq6tVNRBA&t=365s

Heap



(a)

Insert = O(log n), O(h)

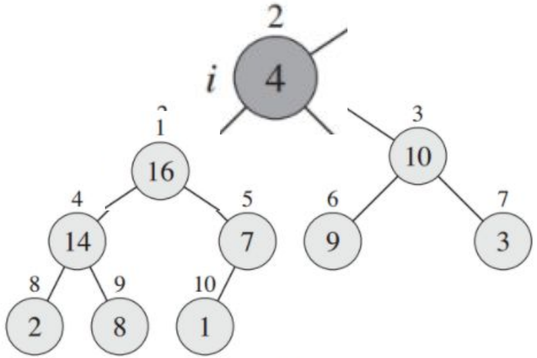
Delete = O(log n), O(h)

Build = O(n)

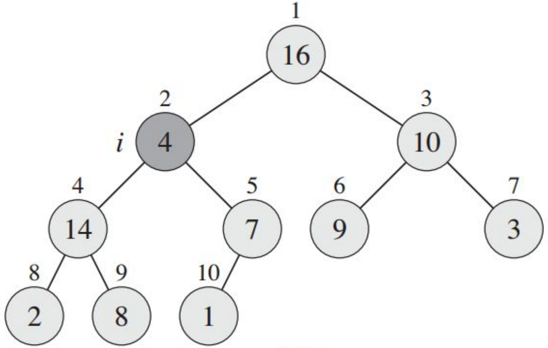
MAX-HEAPIFY

MAX-HEAPIFY = O(log

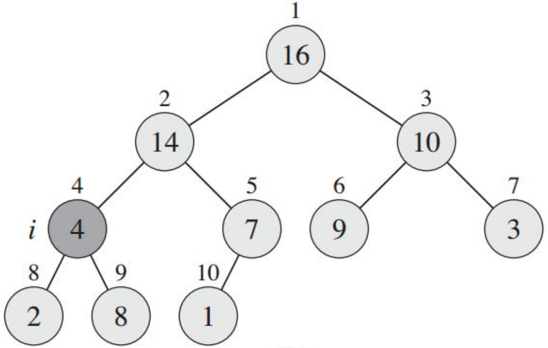
HEAP-EXTRACT-MAX = O(log



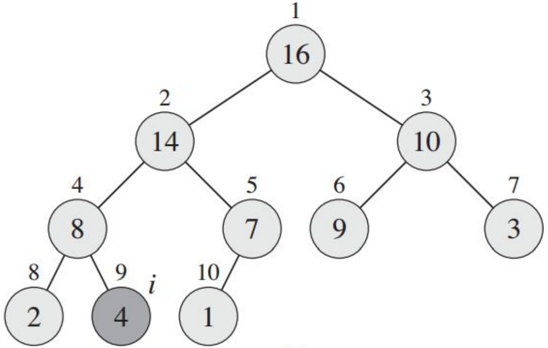
(a)



(a)



(b)

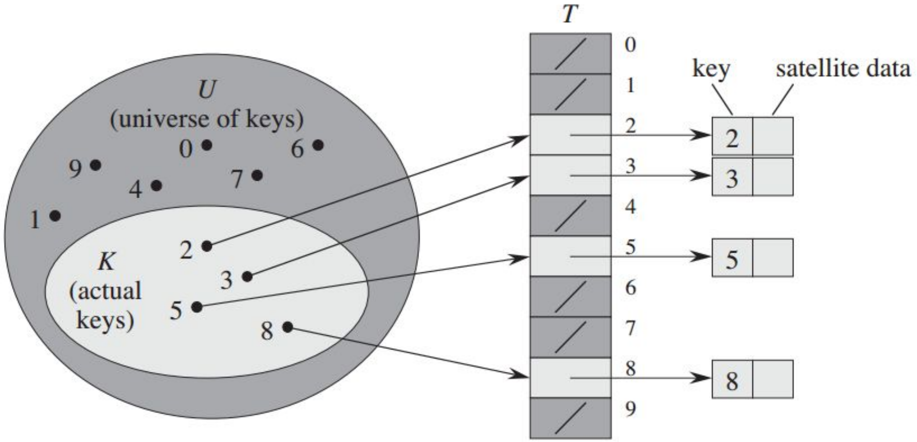


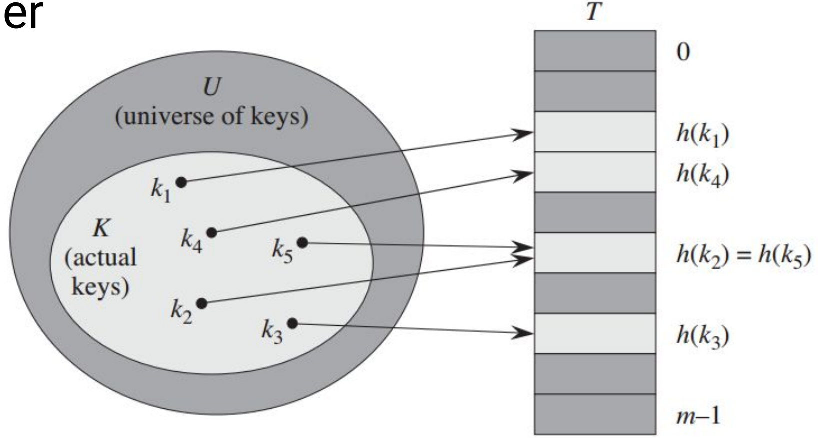
(c)

Hashing

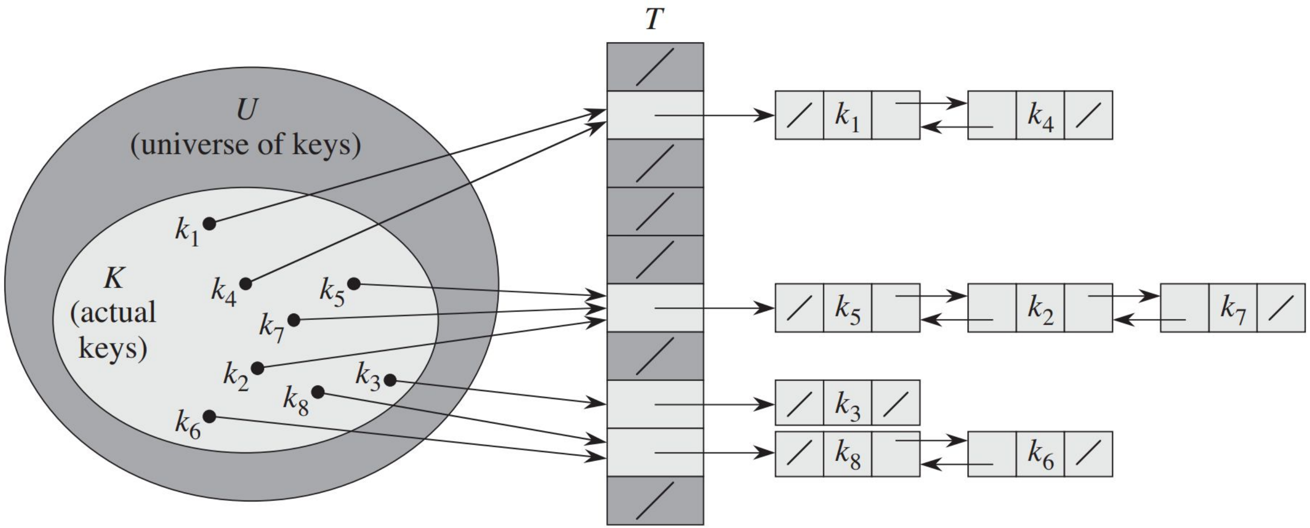
h(k) = index

- Mapping fra nøkler til indekser
- Må være deterministisk
- **Bør** fordele jevnt





Hash chaining



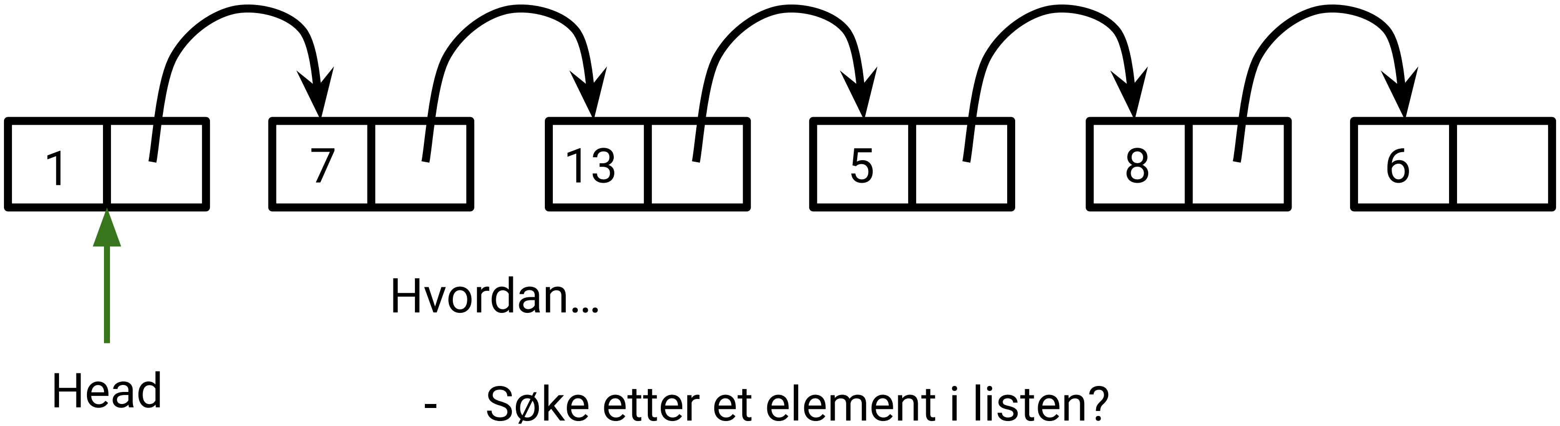
Hva er greia? Hvorfor vil vi ha datastrukturer?

Hva er greia? Hvorfor vil vi ha datastrukturer?

Fordi forskjellige Datastrukturer har forskjellige egenskaper vi kan benytte oss av!

Du har et lager med godteri og du vil ha et program som gjør at du raskt kan sjekke dataen om enkelte typer godteri. Hvilken datastrukturer kan passe her?

Linked list



Hvordan…

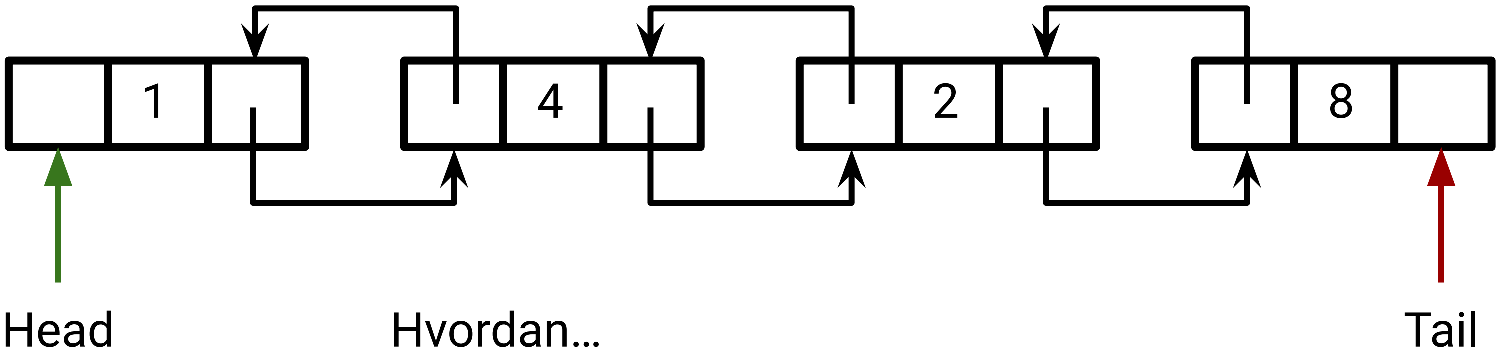
- Søke etter et element i listen?

Begynne fra starten → lineært søk
Kjøretid: O(n)

https://www.youtube.com/watch?v=pKO9UjSeLew

Eller hva om du vil raskt kunne sjekke hvilket godteri som veier mest men også sette inn nye typer godteri raskt?

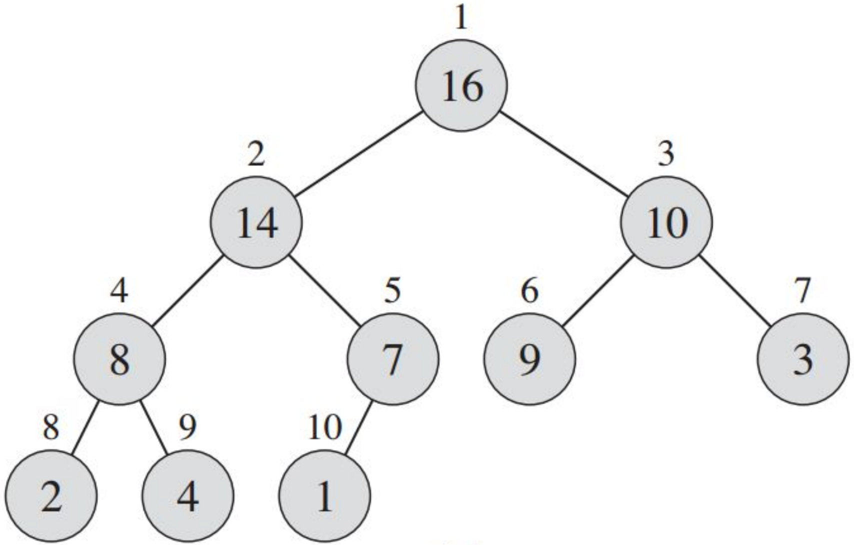
Doubly linked list



Hvordan…

- Slette element i listen?

Eller hva om du vil raskt kunne sjekke hvilket godteri som veier mest? KONSTANT TID!



(a)

Sorteringer

:3

Hvilke egenskaper kan en sorteringsalgoritme ha?

- Sammenligning / ikke sammenligning
- Best / Average / Worst case kjøretid
- Minnebruk - in-place?
- Stabil
- Parallelliserbarhet

Sorteringsalgoritmer

Sammenligningsbasert

Dårlig aka $O(n^2)$

- Bubble
- Insertion
- Selection

Optimalt aka $O(n \lg n)$

- Merge sort
- Quicksort
- Heapsort

Ikke-sammenligningsbasert

$O(n)$

- Counting sort
- Radix sort
- Bucket sort

Bubble sort

Sammenligning	Ja
Best	O(n)
Average	$O(n^2)$
Worst	$O(n^2)$
Minne	O(1)
In-place	Ja
Stabil	Ja
Parallelliserbar	Nei

6 5 3 1 8 7 2 4

Insertion sort

Sammenligning	Ja
Best	$\Theta(n)$
Average	$\Theta(n^2)$
Worst	$\Theta(n^2)$
Minne	$O(1)$
In-place	Ja
Stabil	Ja
Parallelliserbar	Nei

6 5 3 1 8 7 2 4

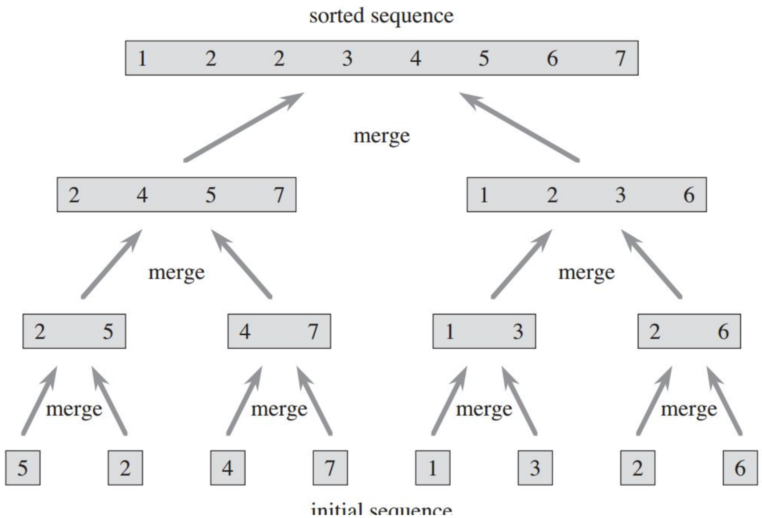
Selection sort

Sammenligning	Ja
Best	$O(n^2)$
Average	$O(n^2)$
Worst	$O(n^2)$
Minne	O(1)
In-place	Ja
Stabil	Nei
Parallelliserbar	Nei

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

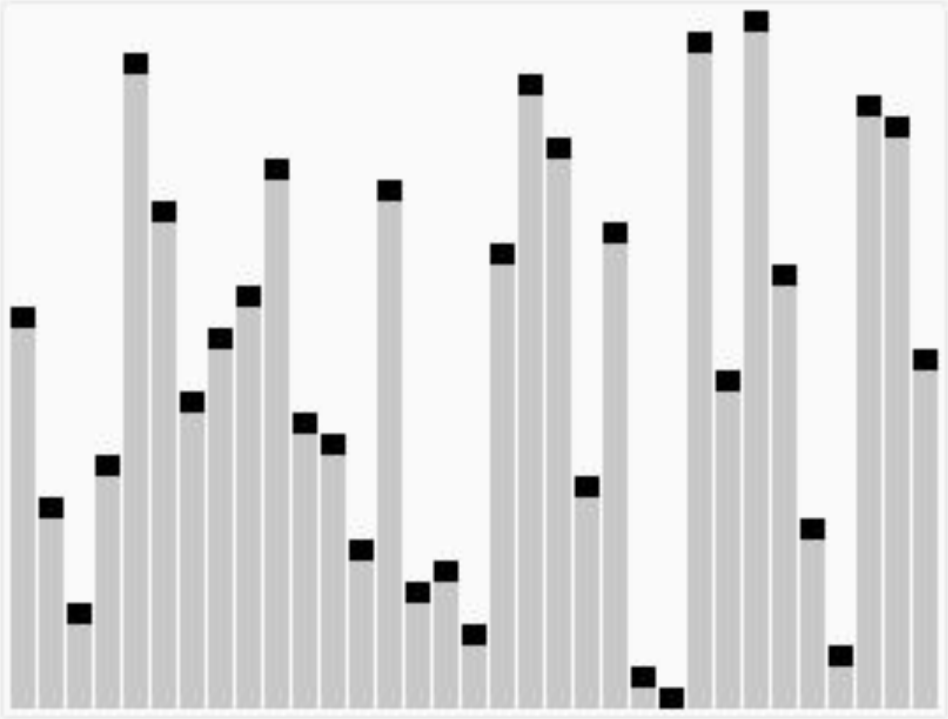
Merge sort

Sammenligning	Ja
Best	Θ(n lg n)
Average	Θ(n lg n)
Worst	Θ(n lg n)
Minne	O(n)
In-place	Nei
Stabil	Ja
Parallelliserbar	Ja



Quicksort

Sammenligning	Ja
Best	Θ(n lg n)
Average	Θ(n lg n)
Worst	$\Theta(n^2)$
Minne	O(lg n)
In-place	Ja
Stabil	Nei
Parallelliserbar	Ja

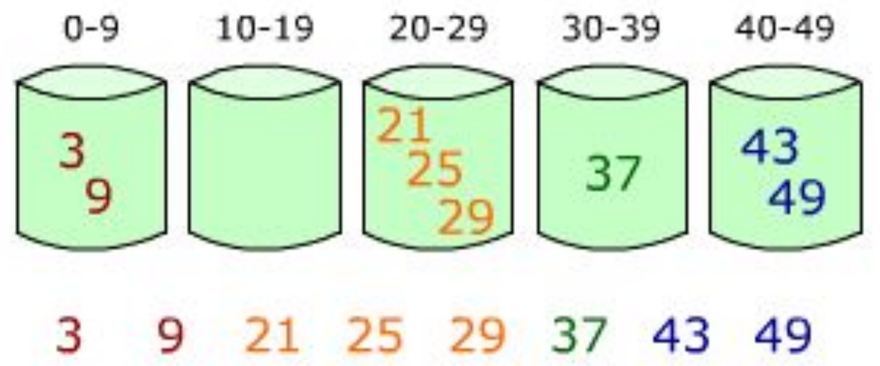


Heapsort

Sammenligning	Ja
Best	Θ(n lg n)
Average	-
Worst	Θ(n lg n)
Minne	O(1)
In-place	Ja
Stabil	Nei
Parallelliserbar	Nei

Bucket sort

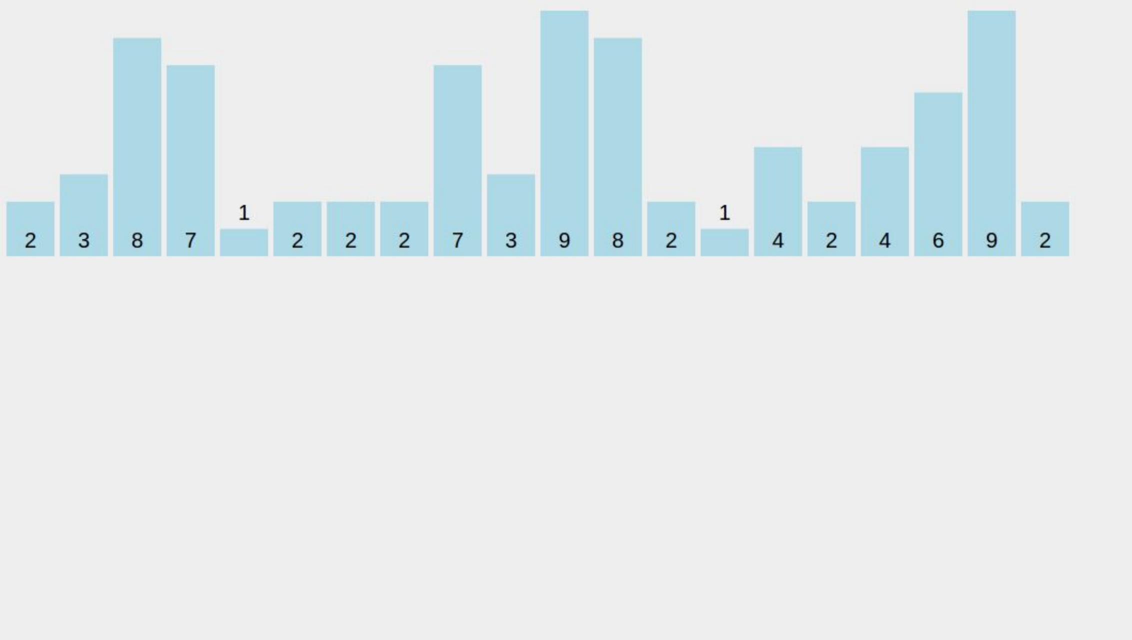
Sammenligning	Nei
Best	$\Theta(n)$
Average	$\Theta(n)$
Worst	$\Theta(n^2)$
Minne	$O(n)$
In-place	Nei
Stabil	Ja
Parallelliserbar	Nei



NB: krever uniform sannsynlighetsfordeling!

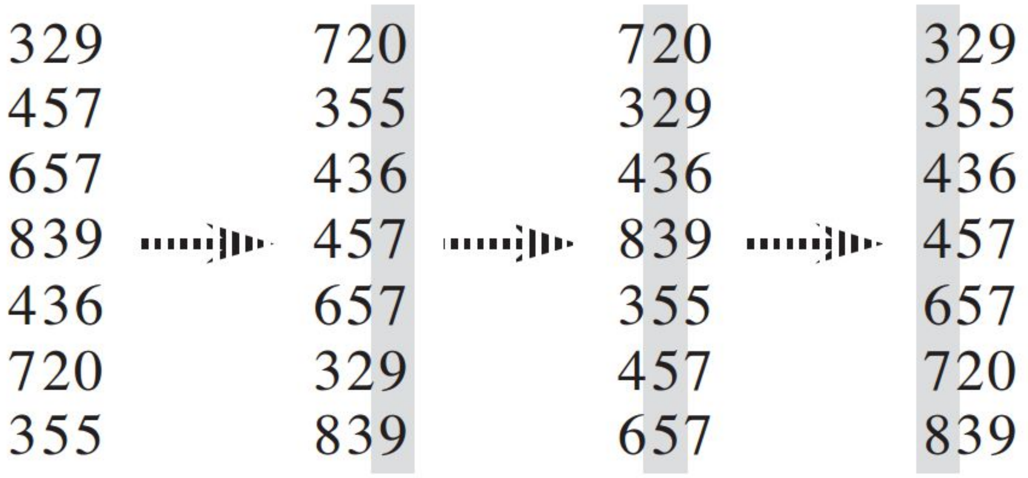
Counting sort

Sammenligning	Nei
Best	Θ(n + k)
Average	Θ(n + k)
Worst	Θ(n + k)
Minne	O(n + k)
In-place	Nei
Stabil	Ja
Parallelliserbar	Nei

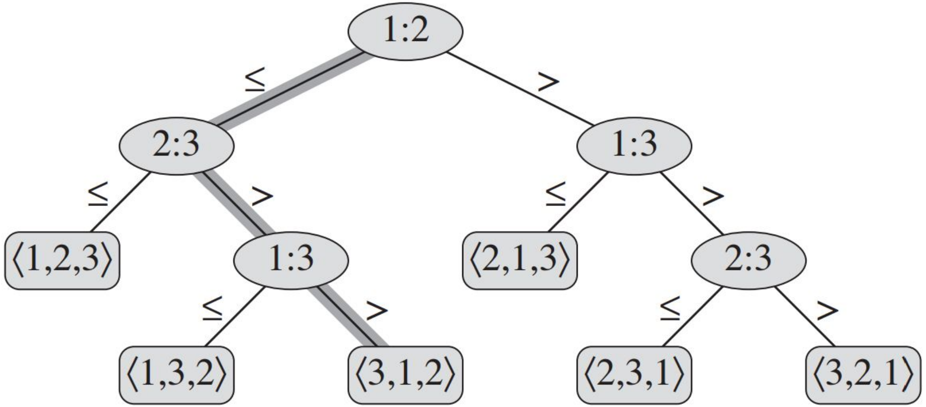


Radix sort

Sammenligning	Nei
Best	$\Theta(d(n + k))$
Average	$\Theta(d(n + k))$
Worst	$\Theta(d(n + k))$
Minne	$O(n + k)$
In-place	Nei
Stabil	Ja
Parallelliserbar	Nei



Hvor fort kan vi gå?



Theorem 8.1
Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof From the preceding discussion, it suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf. Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have

$$n! \leq l \leq 2^h ,$$

which, by taking logarithms, implies

$$\begin{aligned} h &\geq \lg(n!) && \text{(since the lg function is monotonically increasing)} \\ &= \Omega(n \lg n) && \text{(by equation (3.19))} . \end{aligned}$$

■

H2011, oppgave 2c)

c) En venn av deg påstår han har utviklet en generell prioritetskø der operasjonene for å legge til et element, å finne maksimum og å ta ut maksimum alle har kjøretid $O(1)$ i verste tilfelle. Forklar hvorfor dette ikke kan stemme.

Svar (7%):

H2010, oppgave 1g)

g) (6 %) HEAPSORT er optimal, men RADIX-SORT har bedre asymptotisk kjøretid. Forklar svært kort hvordan dette henger sammen.

H2014, oppgave 3a)

Du ønsker å sortere sekvensen $A = (a_1, a_2, \ldots, a_n)$. Det er velkjent at for sammenligningsbasert sortering er $\Omega(n \log n)$ den bese kjøretiden vi kan få, i forventet (*average-case*) og verste tilfelle.

a) Anta at elementene er reelle tall, distribuert etter en gitt sannsynlighetsfordeling, som kan beregnes i konstant tid for ethvert element. Hva er den beste forventede kjøretiden du kan få, og hvordan kan du oppnå den? Forklar kort.

Svar (5 p):

H2014, oppgave 3a)

1 Du har en tabell (*array*) med n heltall og skal finne de k største tallene. Noen løsninger vil ha lavere asymptotisk kjøretid mens andre vil være enklere å implementere, og ha lavere overhead på grunn av enklere datastrukturer, for eksempel. Diskuter mulige løsninger med ulike kjøretider, for eksempel $\Theta(kn)$, $\Theta(n \lg n)$, $\Theta(n \lg k)$ og $\Theta(n)$. Si litt om hvordan de fungerer, og hvilke fordeler og ulemper de har. Hvilke av løsningene dine vil fungere om du vil unngå ekstra minnebruk, og utføre operasjonene *in-place*, ved å bytte om på posisjonene til elementene? Hvilken av disse *in-place*-løsningene ville du ha brukt i en faktisk implementasjon?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

Dynamisk programmering

Dynamisk programmering

Krav til DP:

- Optimal substruktur
- Overlappende subproblemer

Hvordan gjøre det i praksis?

- Memoisering
- Bottom-up problemløsing

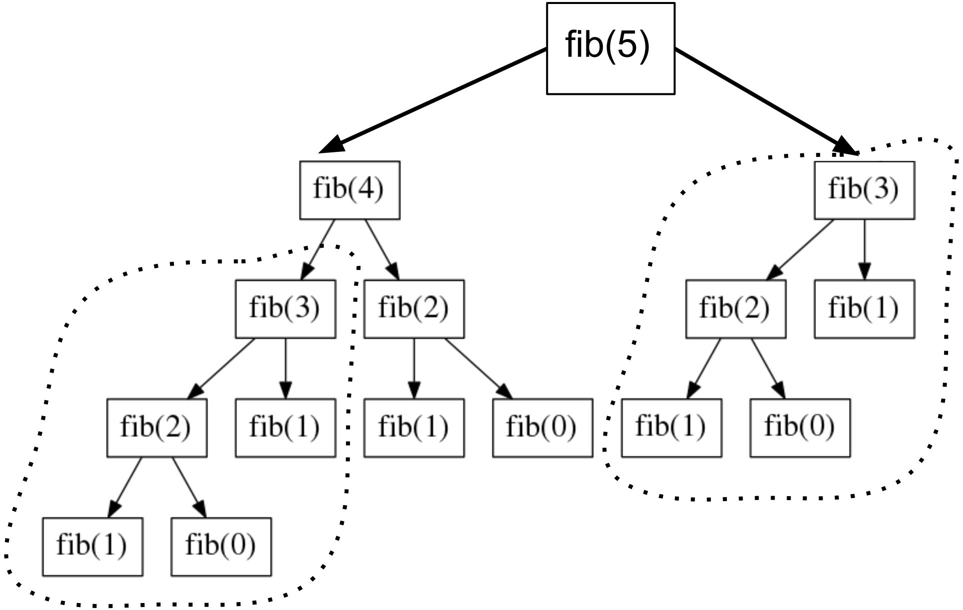
Dynamisk programmering - motivasjon

Fibonacci-tall

$$f_n = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ f_{n-1} + f_{n-2} & \text{hvis } n > 1 \end{cases}$$

Fibonacci

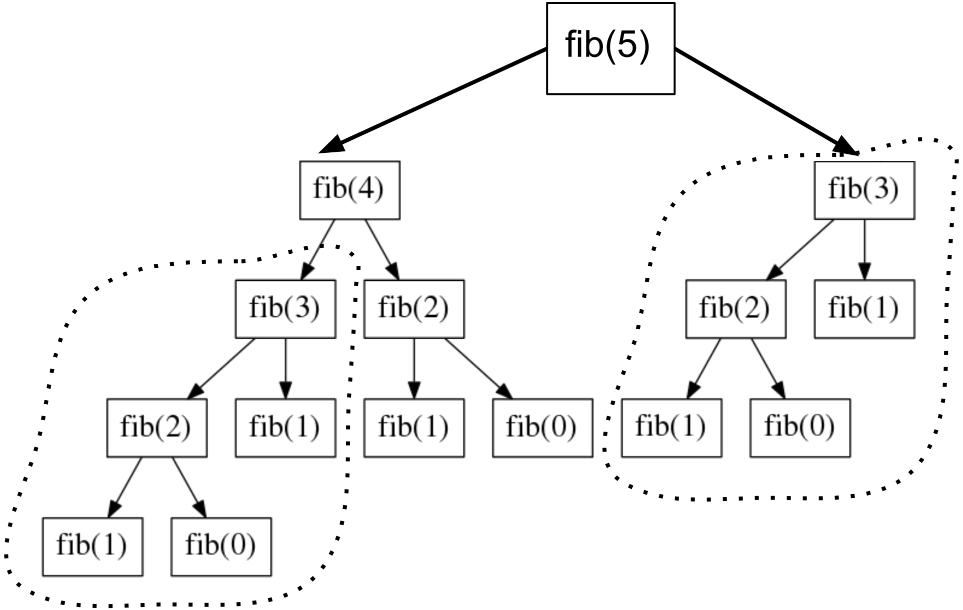
$$f_n = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ f_{n-1} + f_{n-2} & \text{hvis } n > 1 \end{cases}$$



Fibonacci

$$f_n = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ f_{n-1} + f_{n-2} & \text{hvis } n > 1 \end{cases}$$

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

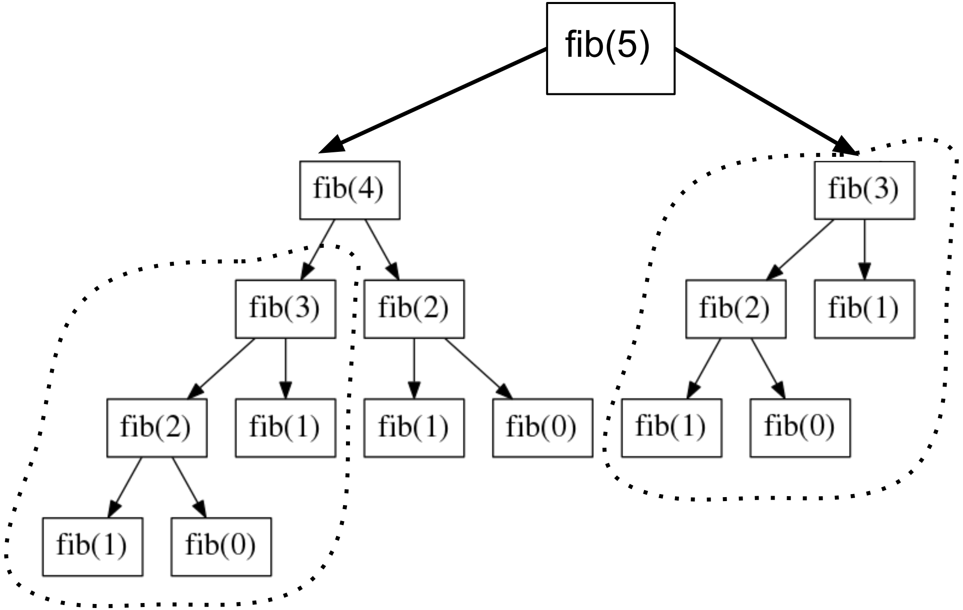


Fibonacci

$$f_n = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ f_{n-1} + f_{n-2} & \text{hvis } n > 1 \end{cases}$$

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

La oss prøve å kjøre den på noen n'er!

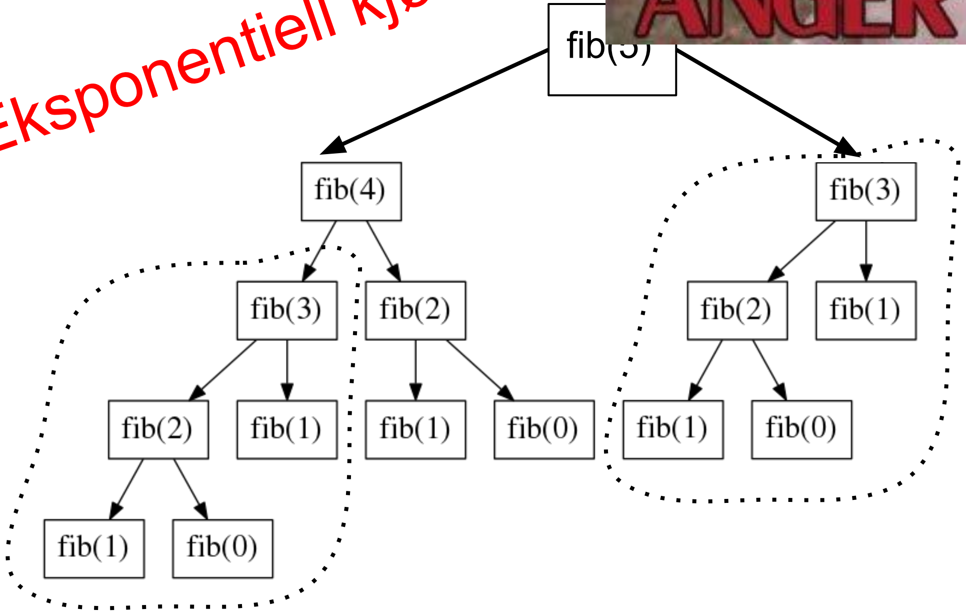


Fibonacci

$$f_n = \begin{cases} 0 & \text{hvis } n = 0 \\ 1 & \text{hvis } n = 1 \\ f_{n-1} + f_{n-2} & \text{hvis } n > 1 \end{cases}$$


```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Eksponentiell kjøretid !



La oss prøve å kjøre den på noen n’er!

Det er jo kjempetregt !

Forbedre Fibonacci

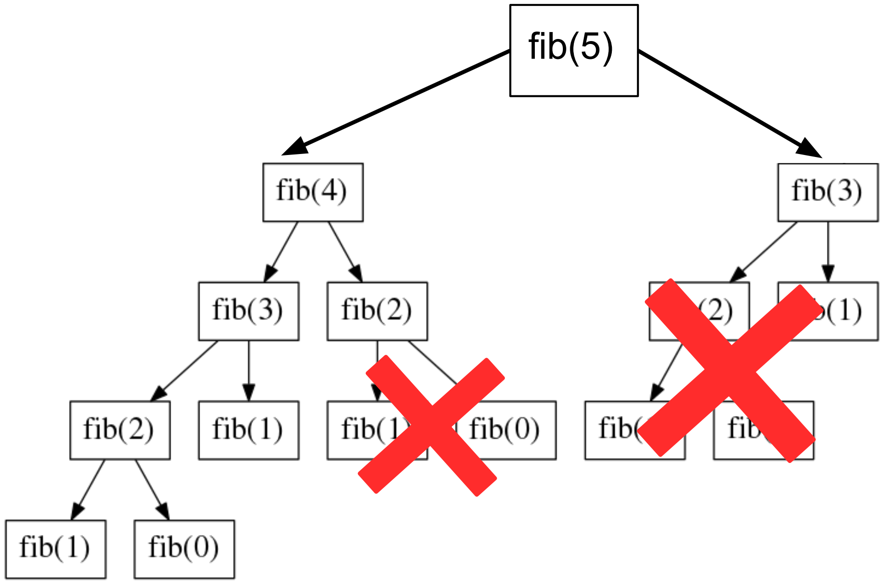
Vi må utnytte de overlappende delproblemene

Forbedre Fibonacci

Vi må utnytte de overlappende delproblemene

```python
memo = {}

def fib(n):
    if n in memo:
        return memo[n]

    f = 0

    if n == 0:
        f = 0
    elif n == 1:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)

    memo[n] = f
    return f
```

La oss prøvekjøre vidunderet!

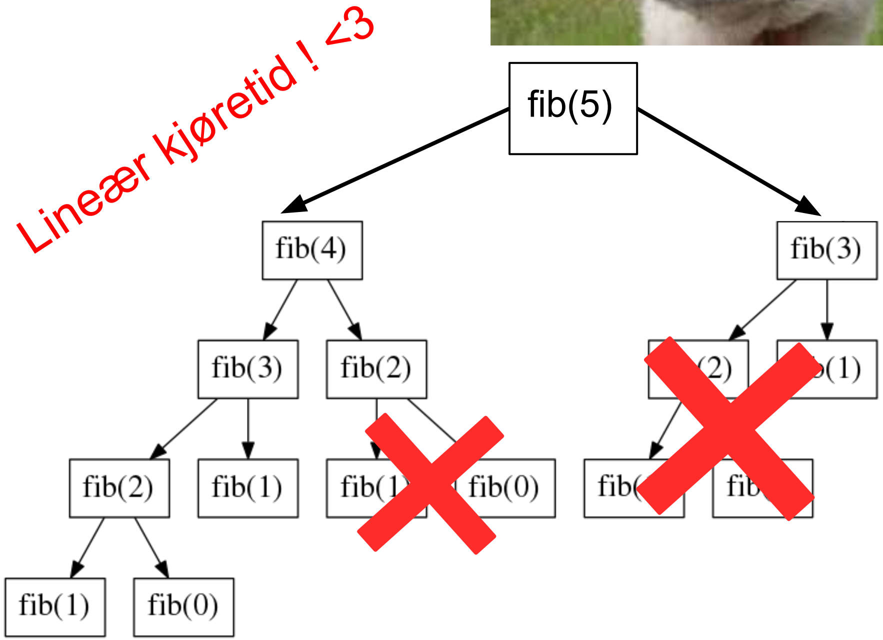


Forbedre Fibonacci

Vi må utnytte de overlappende delproblemene

```python
memo = {}

def fib(n):
    if n in memo:
        return memo[n]

    f = 0

    if n == 0:
        f = 0
    elif n == 1:
        f = 1
    else:
        f = fib(n-1) + fib(n-2)

    memo[n] = f
    return f
```

La oss prøvekjøre vidunderet!



Enda bedre?

0	1	1	2	3	5	8	13

```
def fib(n):
    f = [None] * max(2,n+1)
    f[0] = 0
    f[1] = 1

    for i in xrange(2, n+1):
        f[i] = f[i-1] + f[i-2]

    return f[n]
```

La oss prøvekjøre denne

Enda bedre?

0	1	1	2	3	5	8	13

```python
def fib(n):
    f = [None] * max(2,n+1)
    f[0] = 0
    f[1] = 1

    for i in xrange(2, n+1):
        f[i] = f[i-1] + f[i-2]

    return f[n]
```

Også lineær kjøretid <3

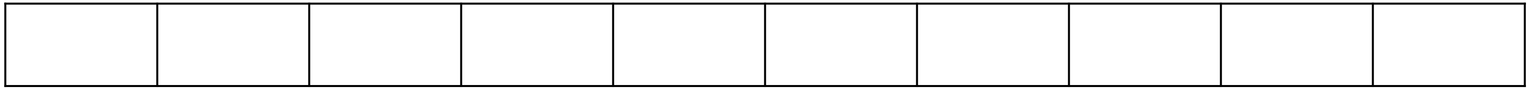
La oss prøvekjøre denne

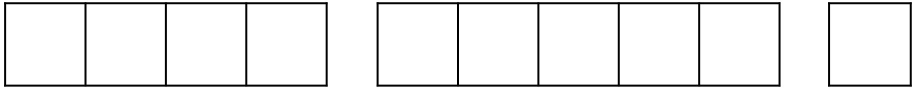
Dynamisk programmering

- Naiv (dum) løsning:
  - Lett, men eksponentiell tid
- Memoisering
  - Lineær tid
- Bottom up
  - Lineær tid

- Problemer som består av delproblemer
- Delproblemene overlapper (brukes flere ganger)
- Som regel er vi interessert i optimaliseringsproblemer
  - La oss se på et

Rod Cutting



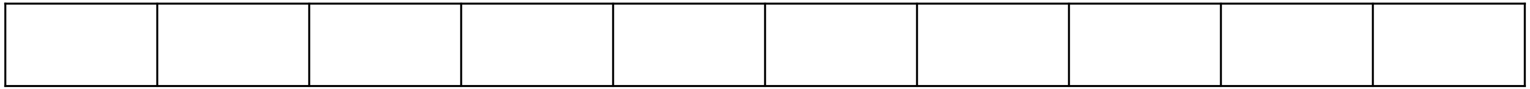


9 + 10 + 1 = 20

Optimal substruktur

1	$1
2	$5
3	$8
4	$9
5	$10
6	$17
7	$17
8	$20
9	$24
10	$30

Rod Cutting



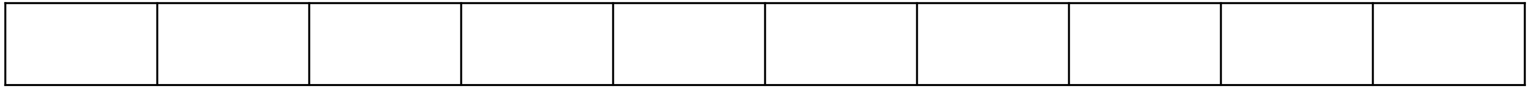
r_n: maks avkastning for stang av lengde n
p_n: prisen for en stang av lengde n

$$r_n = \max(p_n, r_1+r_{n-1}, r_2+r_{n-2}, \dots , r_{n-2}+r_2, r_{n-1}+r_1)$$

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

1	\$1
2	\$5
3	\$8
4	\$9
5	\$10
6	\$17
7	\$17
8	\$20
9	\$24
10	\$30

Rod Cutting



r_n: maks avkastning for stang av lengde n
p_n: prisen for en stang av lengde n

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

```
CUT-ROD(p,n)
        if n == 0
                return 0
        q = -∞
        for i = 1 to n
                q = max(q, p[i] + CUT-ROD(p, n-i))
        return q
```

1	$1
2	$5
3	$8
4	$9
5	$10
6	$17
7	$17
8	$20
9	$24
10	$30

Rod Cutting

r_n: maks avkastning for stang av lengde n
p_n: prisen for en stang av lengde n

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

```
CUT-ROD(p,n)
    if n == 0
        return 0
    q = -∞
    for i = 1 to n
        q = max(q, p[i] + CUT-ROD(p, n-i))
    return q
```

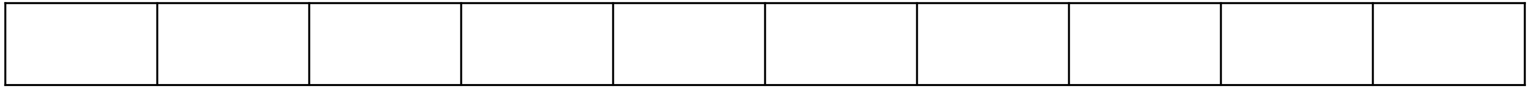
Eksponentiell kjøretid :-(

Men vi kan memoisere :-)

Hva blir kjøretiden da?

1	\$1
2	\$5
3	\$8
4	\$9
5	\$10
6	\$17
7	\$17
8	\$20
9	\$24
10	\$30

Rod Cutting



r_n: maks avkastning for stang av lengde n
p_n: prisen for en stang av lengde n

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Hva hvis vi skal bygge løsningen nedenfra og opp?
La oss prøve å gjøre det

1	\$1
2	\$5
3	\$8
4	\$9
5	\$10
6	\$17
7	\$17
8	\$20
9	\$24
10	\$30

Rod Cutting

1	$1
2	$5
3	$8
4	$9
5	$10
6	$17
7	$17
8	$20
9	$24
10	$30

r_n: maks avkastning for stang av lengde n
p_n: prisen for en stang av lengde n

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Hva hvis vi skal bygge løsningen nedenfra og opp?
La oss prøve å gjøre det

i	0	1	2	3	4	5	6	7	8	9	10
r[i]	0	1	5	8	10	13	17	18	22	25	30

Rod Cutting

1	$1
2	$5
3	$8
4	$9
5	$10
6	$17
7	$17
8	$20
9	$24
10	$30

r_n: maks avkastning for stang av lengde n
p_n: prisen for en stang av lengde n

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

```
BOTTOM-UP-CUT-ROD(p, n)
    let r[0..n] be a new array
    r[0] = 0
    for j = 1 to n
        q = -∞
        for i = 1 to j
            q = max(q, p[i] + r[j - i])
        r[j] = q
    return r[n]
```

Hva blir kjøretiden?

Rod Cutting

r_n: maks avkastning for stang av lengde n
p_n: prisen for en stang av lengde n

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

```
BOTTOM-UP-CUT-ROD(p, n)
    let r[0..n] be a new array
    r[0] = 0
    for j = 1 to n
        q = -∞
        for i = 1 to j
            q = max(q, p[i] + r[j - i])
        r[j] = q
    return r[n]
```

Hva blir kjøretiden?

$\Theta(n^2)$
Akkurat som memoisering

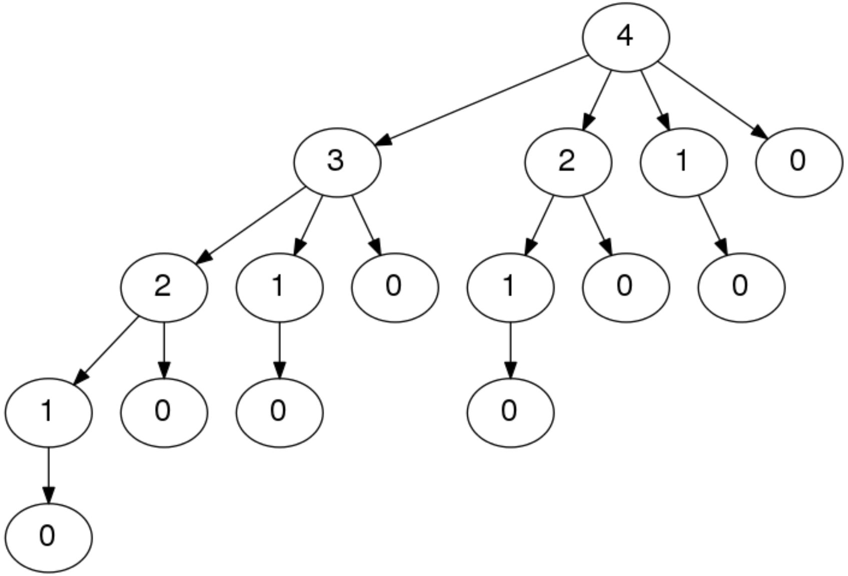
1	$1
2	$5
3	$8
4	$9
5	$10
6	$17
7	$17
8	$20
9	$24
10	$30

Hva skjedde nå?

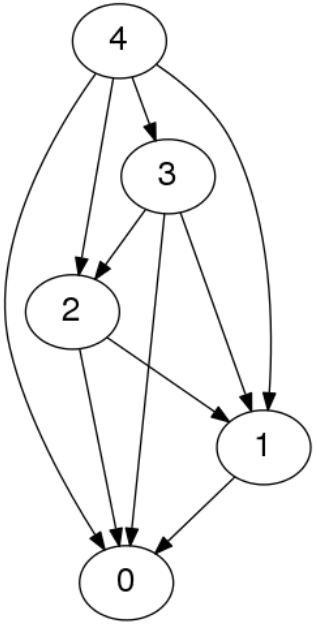
- Problemet vårt hadde optimal substruktur
- Problemet hadde overlappende delproblemer
- Vi sørget for å løse et delproblem kun én gang, og effektivt byttet lagringsplass mot bedre kjøretid

Hva skjedde nå?

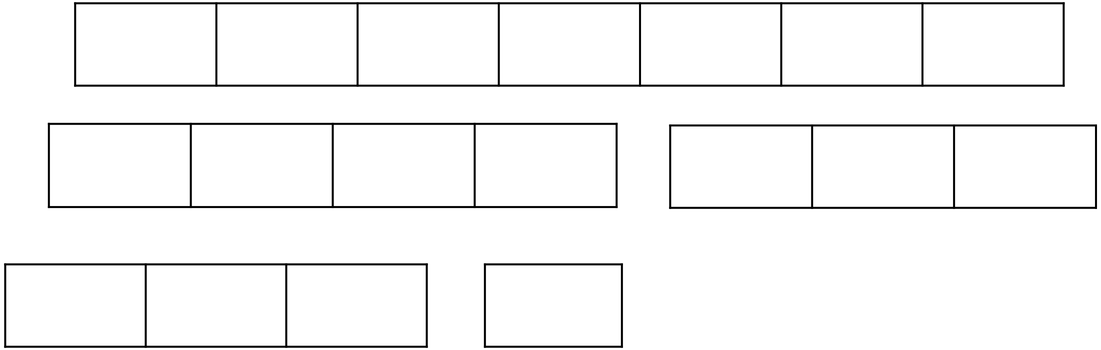
- Problemet vårt hadde optimal substruktur
- Problemet hadde overlappende delproblemer
- Vi sørget for å løse et delproblem kun én gang, og effektivt byttet lagringsplass mot bedre kjøretid



Delproblem-grafen

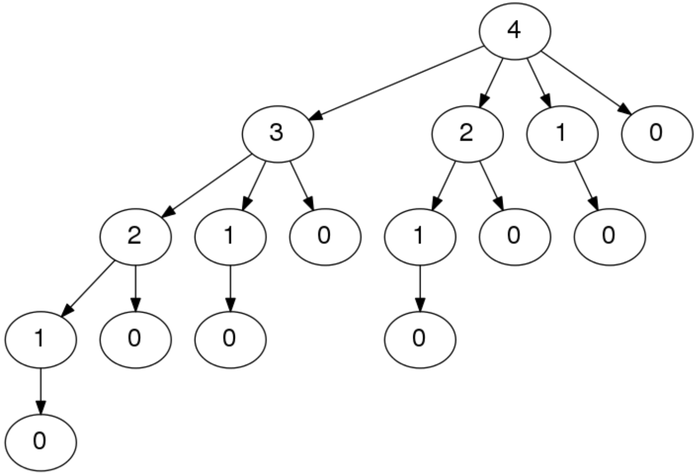


Abstraksjon

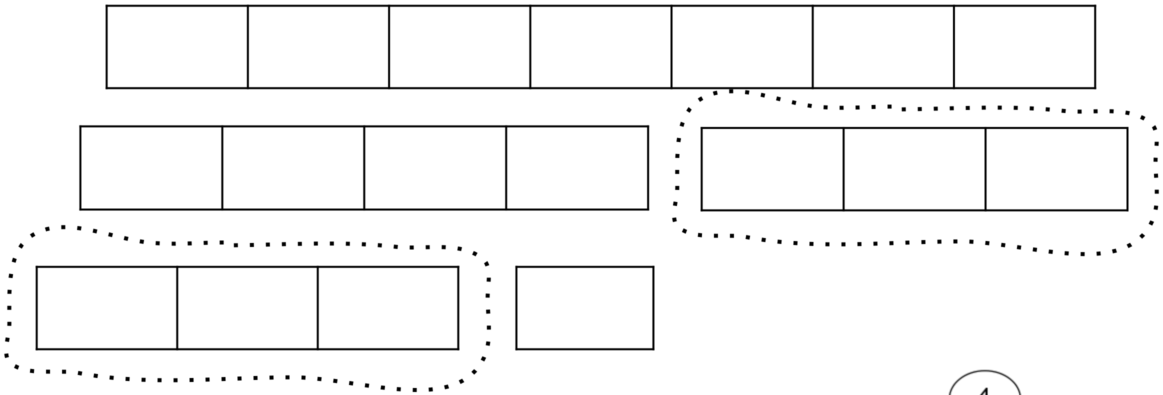


Vi må ha:

- Optimal substruktur
- Overlappende delproblemer

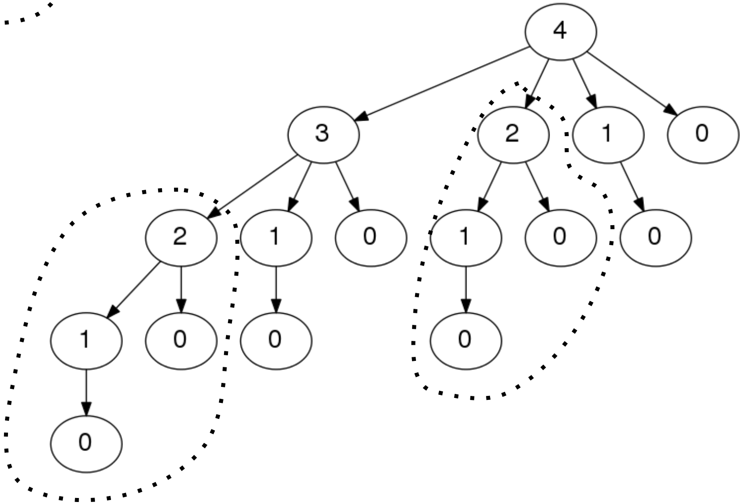


Abstraksjon



Vi må ha:

- Optimal substruktur
- Overlappende delproblemer

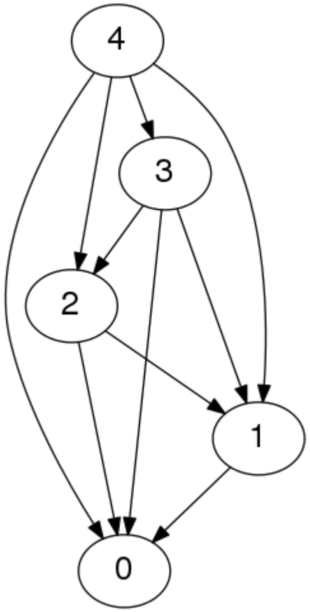


H2010, oppgave 1h

h) (6 %) Hvorfor er det ikke alltid nyttig å bruke memoisering i rekursive algoritmer?

Så hvordan går vi fram?

1. Beskriv/karakteriser strukturen til en optimal løsning (definer problem-parametere og finn delproblem-grafen) - tenk på hvilke **valg** som må gjøres
2. Definer rekursivt verdien til en optimal løsning
3. Regn ut verdien til en optimal løsning (og husk valgene du gjør)
4. Bygg opp en optimal løsning basert på beregnet informasjon



H2012, oppgave 1f

f) I ryggsekkproblemet (0-1 *knapsack*), la $c[i, w]$ være optimal verdi for de i første objektene, med en kapasitet på w. La v_i og w_i være henholsvis verdien og vekten til objekt i. Fyll ut rekurrensen for $c[i, w]$, hvis vi antar at $i > 0$ og $w_i \leq w$.

Svar (6%): $c[i, w] = \max\{$, $\}$.

3 Når en algoritme utføres, gjentas gjerne et «trinn» mange ganger, til vi har funnet resultatet for en gitt instans. Hvordan relaterer dette seg til dekomponering av instansen i delinstanser (*subproblems*)? Hvordan arter dette seg forskjellig for ulike designmetoder? Hvilken rolle spiller matematisk induksjon oppi det hele? Hvilket slektskap har dekomponering til reduksjoner og hardhetsbevis, og hvordan stemmer forklaringen din overens med dette?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

5 % **3** Vurder følgende utsagn om splitt-og-hersk (*divide-and-conquer*):

> Metoden bør unngås når vi har overlappende delproblemer.

Stemmer dette? Svar ja eller nei og forklar kort.

Rekkefølge :3

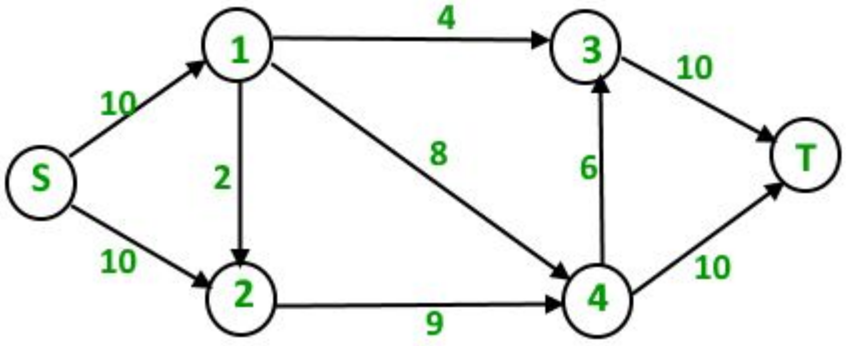
1. Flyt
2. NP
3. Datastrukturer + sortering + dynamisk prog

Flyt

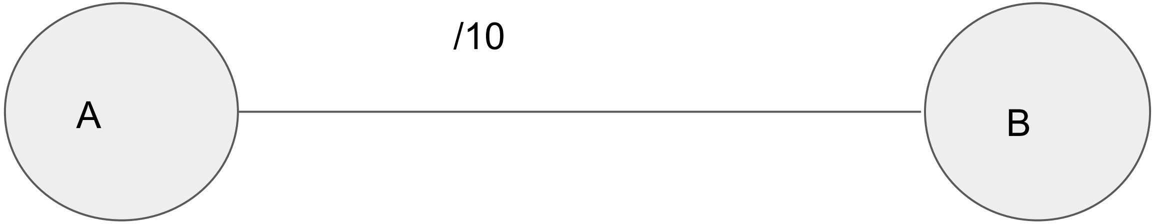
:3

Hva er flyt-problemet?

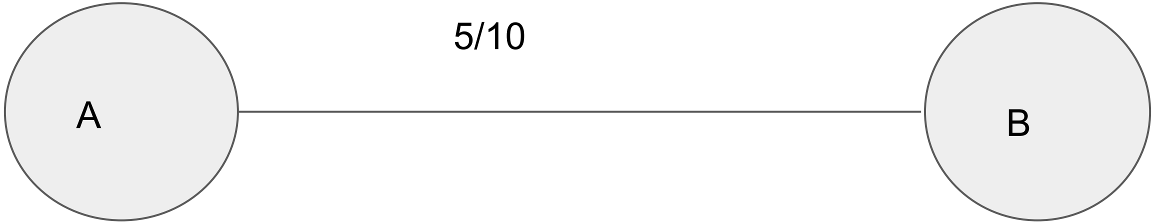
Hvor mye kan vi sende fra S til T?



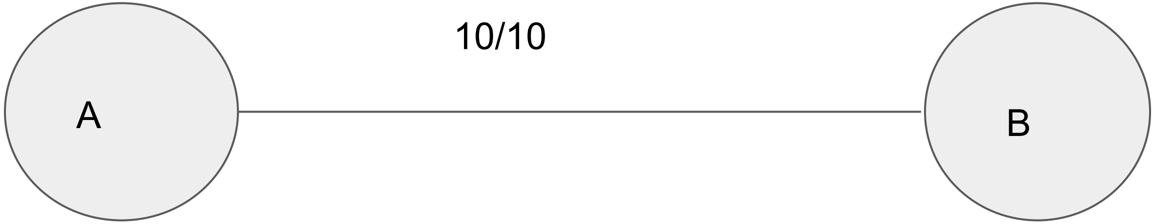
Hvordan kan vi sende flyt?



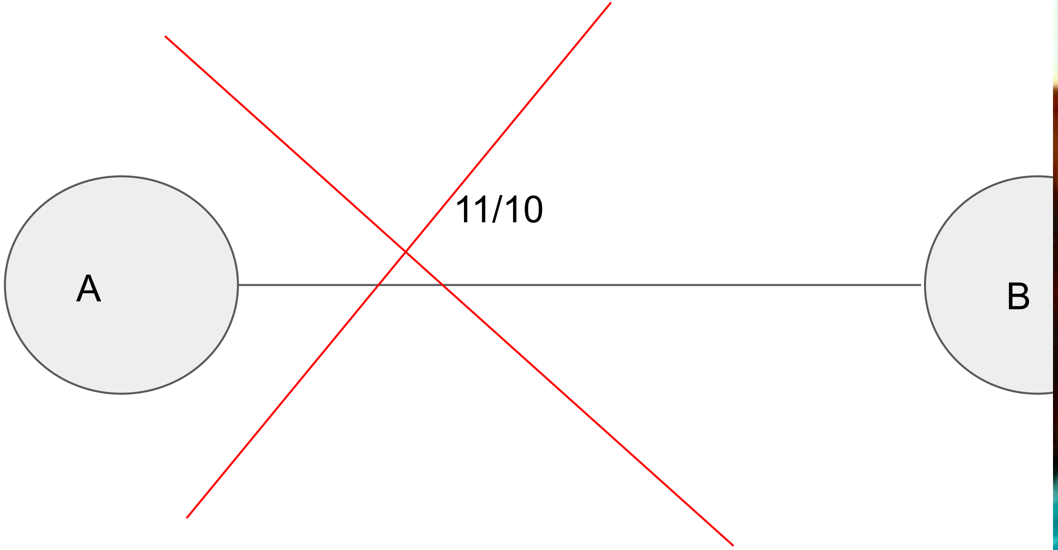
Hvordan kan vi sende flyt?



Hvordan kan vi sende flyt?



Hvordan kan vi sende flyt?



Gloser & representasjon av flyt:

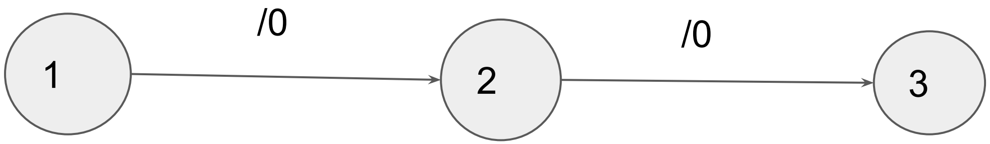
- Kapasitetsmatrise
- Flytmatrise
- Residualmatrise

Representasjon av Kapasitet

Kapasitet:

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

===

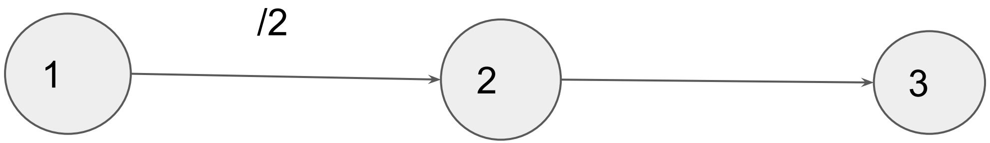


Representasjon av kapasitet

Kapasitet:

$$\begin{array}{cc} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$

===

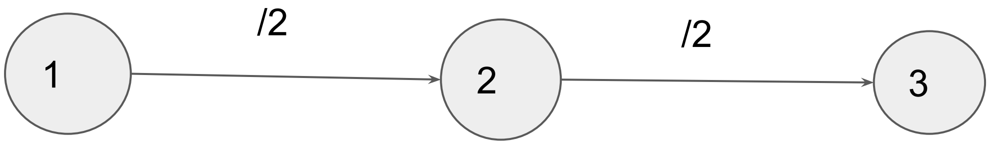


Representasjon av kapasitet

Kapasitet:

$$\begin{array}{cc} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left[\begin{array}{ccc} 0 & 2 & 0 \\ 0 & 0 & 2 \\ 0 & 0 & 0 \end{array}\right] \end{array}$$

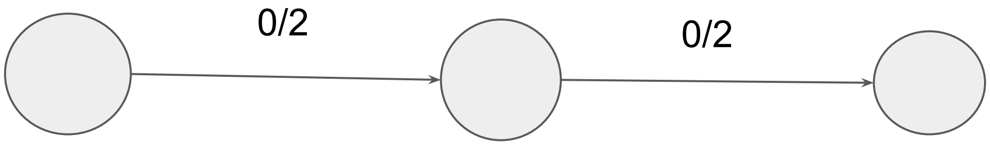
===



Representasjon av flyt

Flytmatrise:

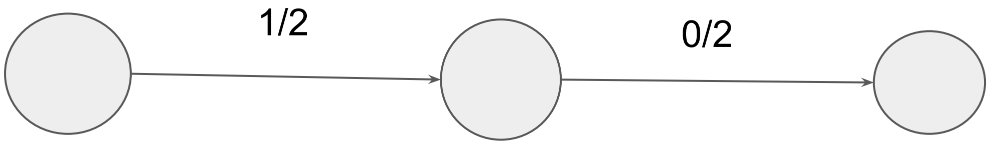
$$
\begin{array}{c c}
 & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\end{array}
$$



Representasjon av flyt

Flytmatrise:

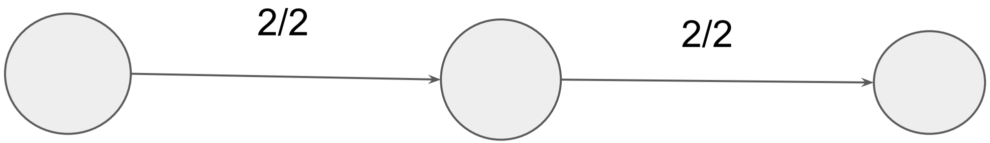
$$
\begin{array}{c|ccc|}
 & 1 & 2 & 3 \\
1 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 \\
3 & 0 & 0 & 0
\end{array}
$$



Representasjon av flyt

Flytmatrise:

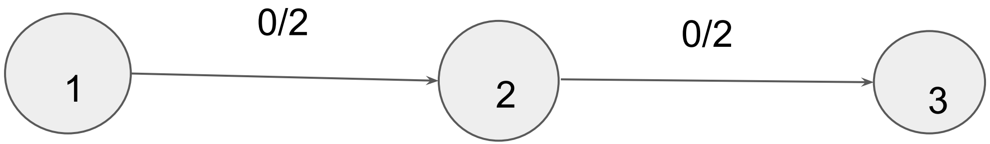
$$
\begin{array}{c|ccc}
 & 1 & 2 & 3 \\
1 & [\,0 & 2 & 0\,] \\
2 & [\,0 & 0 & 2\,] \\
3 & [\,0 & 0 & 0\,]
\end{array}
$$



Residualgraf

Residualmatrise:

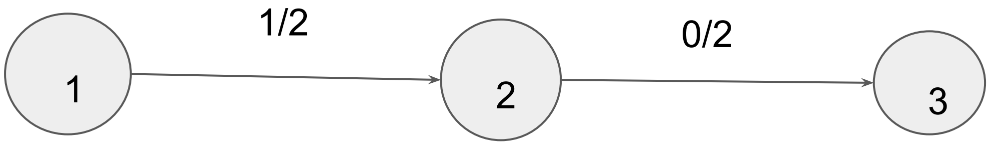
$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ 1 & [\,0 & 2 & 0\,] \\ 2 & [\,0 & 0 & 2\,] \\ 3 & [\,0 & 0 & 0\,] \end{array}$$



Residualgraf

Residualmatrise:

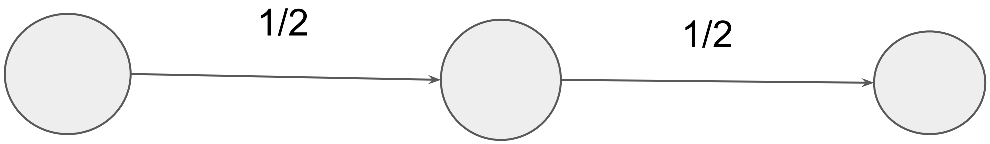
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array}\begin{array}{c} \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \end{array}$$



Residualgraf

Residualmatrise:

$$\begin{array}{c c} & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \end{array} & \left[\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right] \end{array}$$



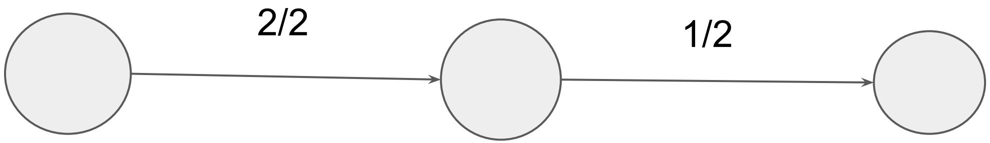
Residualgraf

Residualmatrise:

$$
\begin{array}{c c}
 & \begin{matrix} 1 & 2 & 3 \end{matrix} \\
\begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}
\end{array}
$$

Fra u til v = C(u,v) - F(u,v)

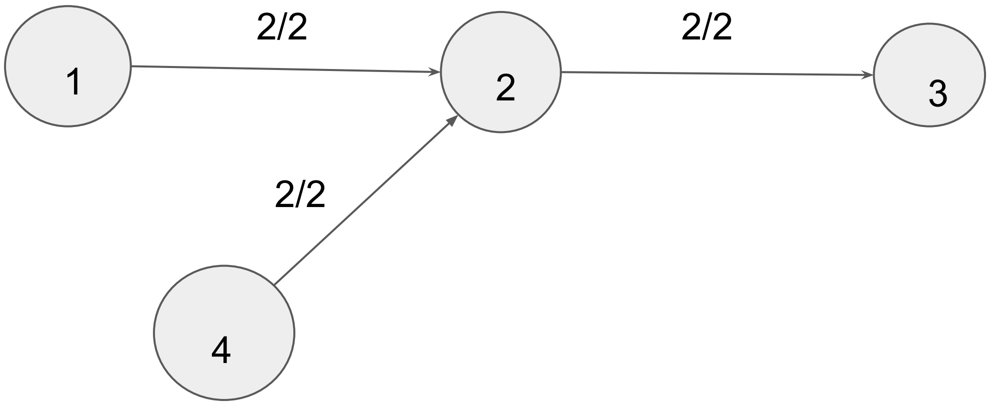
Fra v til u = F(u,v)



Residualgraf

Residualmatrise:

$$\begin{array}{c|c} & \begin{matrix} 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 2 & 0 \end{bmatrix} \end{array}$$

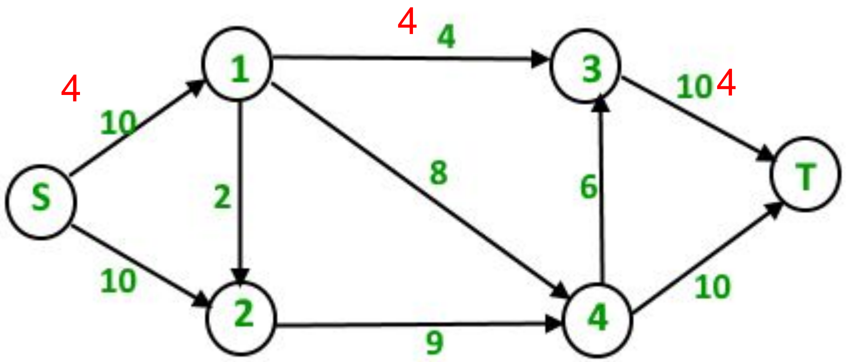


Ford fulkerson

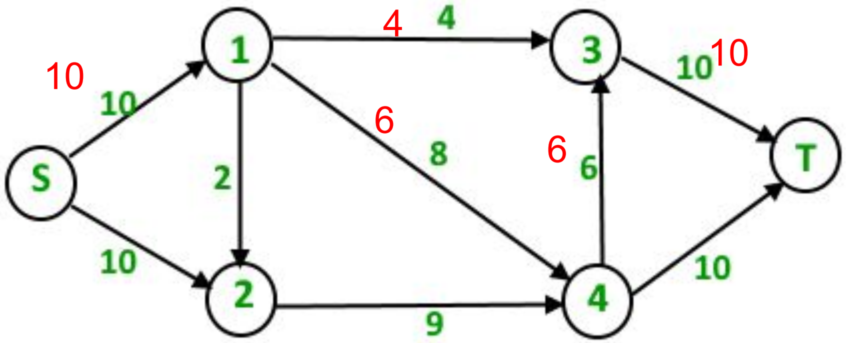
Så lenge vi finner en sti som kan øke flyten:

Endre flyten

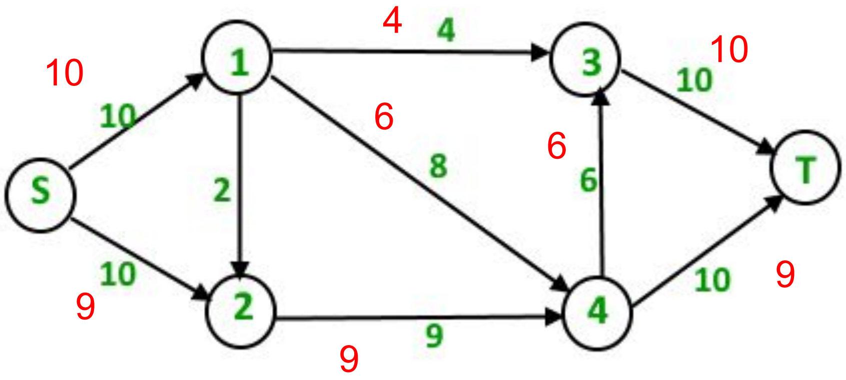
Sti: s => 1 => 3 => T



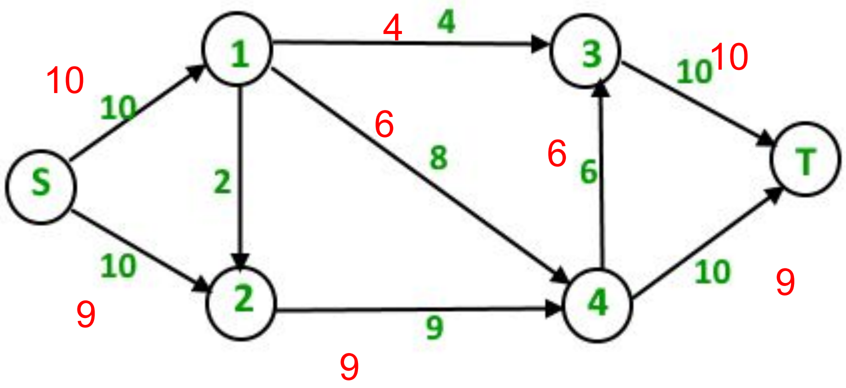
Sti: s => 1 => 4 => 3 => T



Sti: s => 2 => 4 => T

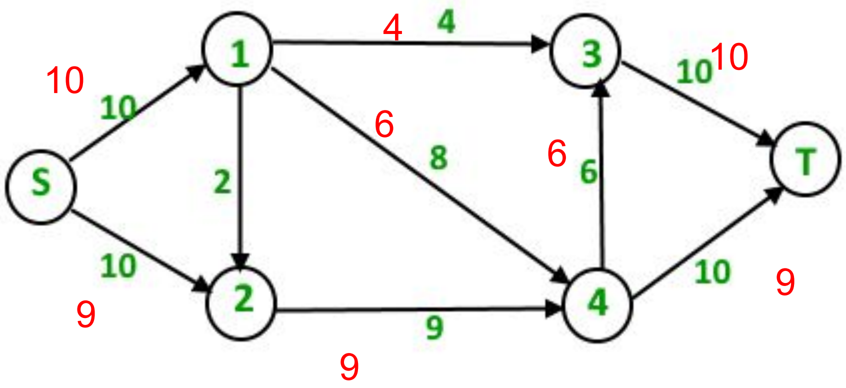


Ingen fler stier: maks flyt = 19



Minimalt kutt = maks flyt

Minimalt kutt er delmengdene s = {}, t = {} slik at flyt-kapasiteten fra nodene i S til T minimal.

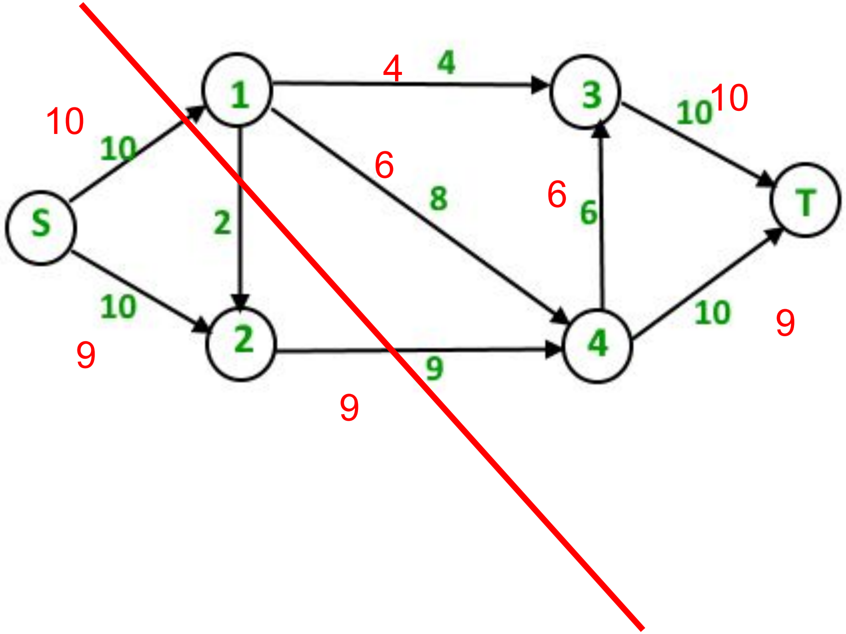


Minimalt kutt = maks flyt

Minimalt kutt er delmengdene s = {}, t = {} slik at flyt-kapasiteten fra nodene i S til T minimal.

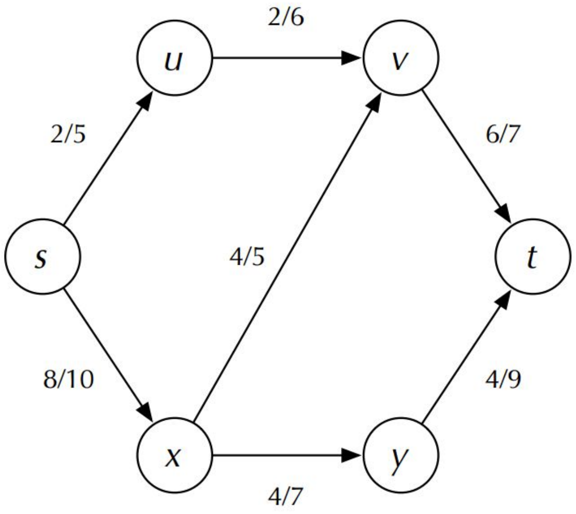
T = {1, 3, 4, T}

S = {S, 2}



H2011, oppgave 2d

Betrakt følgende flytnettverk over nodene {s, t, u, v, x, y}, med kilde s og sluk t:



Flyt og kapasitet er angitt på kantene (for eksempel er flyten fra x til y på 4, med kapasitet på 7).

d) Angi den flytforøkende stien (*augmenting path*) som gir størst økning i flyten. Svaret oppgis som en sekvens av noder.

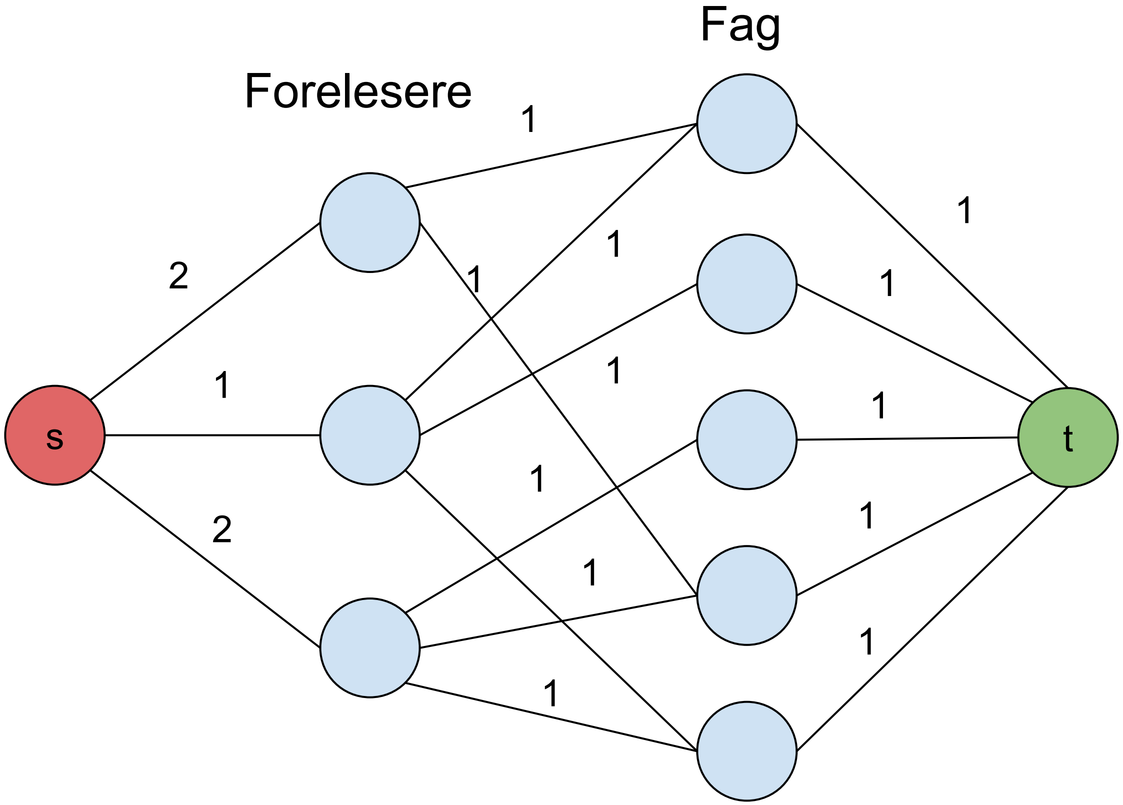
Svar (7%):

H2010, oppgave 6a

Oppgave 6 Du skal fordele fag på forelesere for neste semester. Du har en liste med fag som skal foreleses, og et sett med forelesere å ta av. For hver foreleser har du en liste med fag som denne er kompetent til å forelese. Foreleserne har også ulik arbeidskapasitet, og du har oppgitt hvor mange fag hver foreleser orker å forelese i løpet av et semester. Målet er å sørge for at alle fagene foreleses uten at noen må forelese noe de ikke kan, og uten at noen må forelese flere fag enn de orker.

a) (5 %) Beskriv hvordan problemet kan løses som et flytproblem.

H2010, oppgave 6a

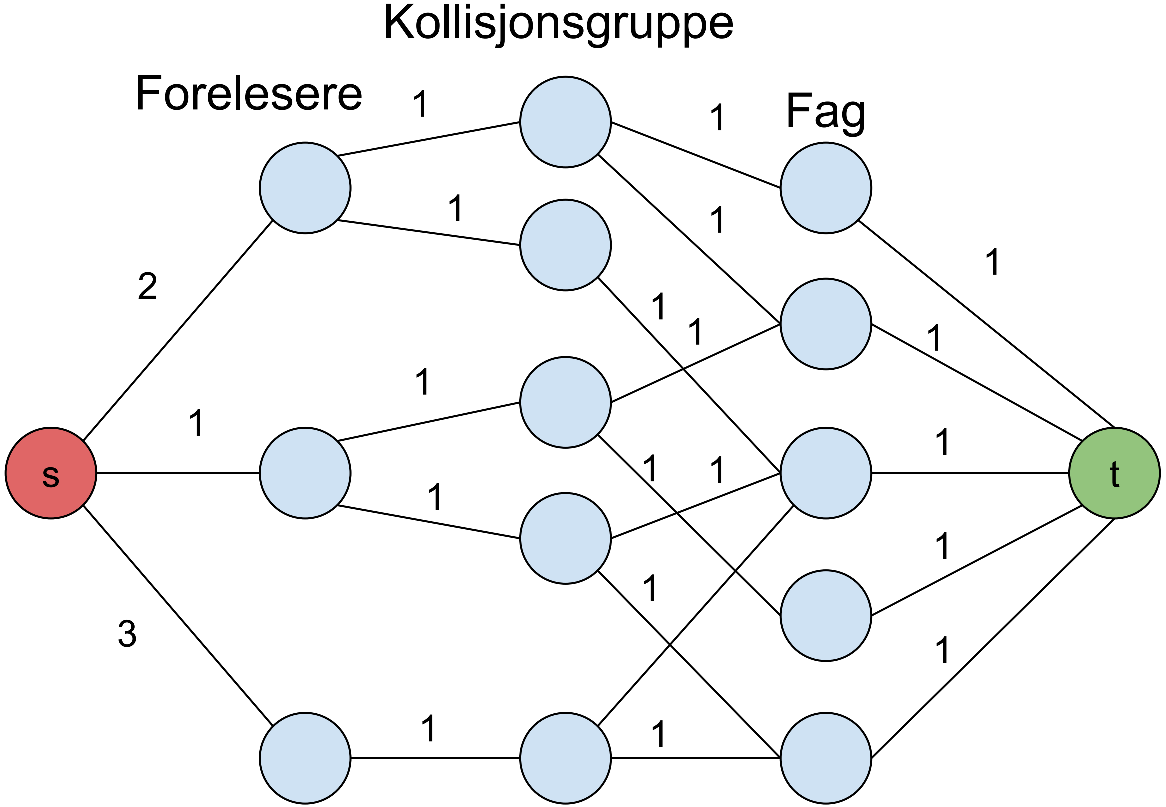


H2010, oppgave 6b

Du oppdager nå at det er kollisjoner mellom enkelte av fagene (det vil si, de skal foreleses på samme tidspunkt). Du vil unngå å gi en faglærer to (eller flere) fag som kolliderer. (Dersom to fag kolliderer har de eksakt samme forelesningstidspunkt og varighet. Du kan dermed anta at om fag A kolliderer med fag B og B kolliderer med fag C så kolliderer også A med C.)

b) (5 %) Beskriv hvordan du kan håndtere kollisjonene og fortsatt løse problemet som et flytproblem.

H2010, oppgave 6b



(5%) (c) Et team med forskere skal gjennomføre et sett med prosjekter.

- Hvert prosjekt består av flere oppgaver, og et visst antall av prosjektets oppgaver (f.eks. 7 av 12) må gjøres. Ulike prosjekt kan ha ulike antall.
- Hver oppgave skal utføres av én forsker.
- Hver forsker har en viss kapasitet, dvs., et antall oppgaver hun rekker å gjøre.
- Hver forsker er kompetent til å gjøre noen av oppgavene, men ikke nødvendigvis alle.

Du skal avgjøre hvem som skal gjøre hvilke oppgaver, eller finne ut at det ikke går.

Beskriv hvordan du kan løse dette som et maks-flyt-problem. Tegn gjerne et flytnettverk.

4 Din venn Lurvik mener han har funnet på en ny algoritme for å finne korteste vei i vektede, rettede grafer, der vektene er positive heltall. Ideen hans er å transformere kanter (u, v) med vekt $w(u, v) = k > 1$ til stier $\langle u, x_1, x_3, \ldots, x_{k-1}, v \rangle$ med lengde k, og så bruke BFS til å finne korteste vei. Hvilke fordeler og ulemper har denne metoden? Diskuter slektskap med algoritmer i pensum. Kunne du ha gjort noe lignende for å finne maksimal flyt med heltallskapasiteter, hvis du hadde en algoritme som kunne finne maksimal flyt? Hva slags algoritme måtte du i så fall ha hatt for å ta BFS sin plass?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

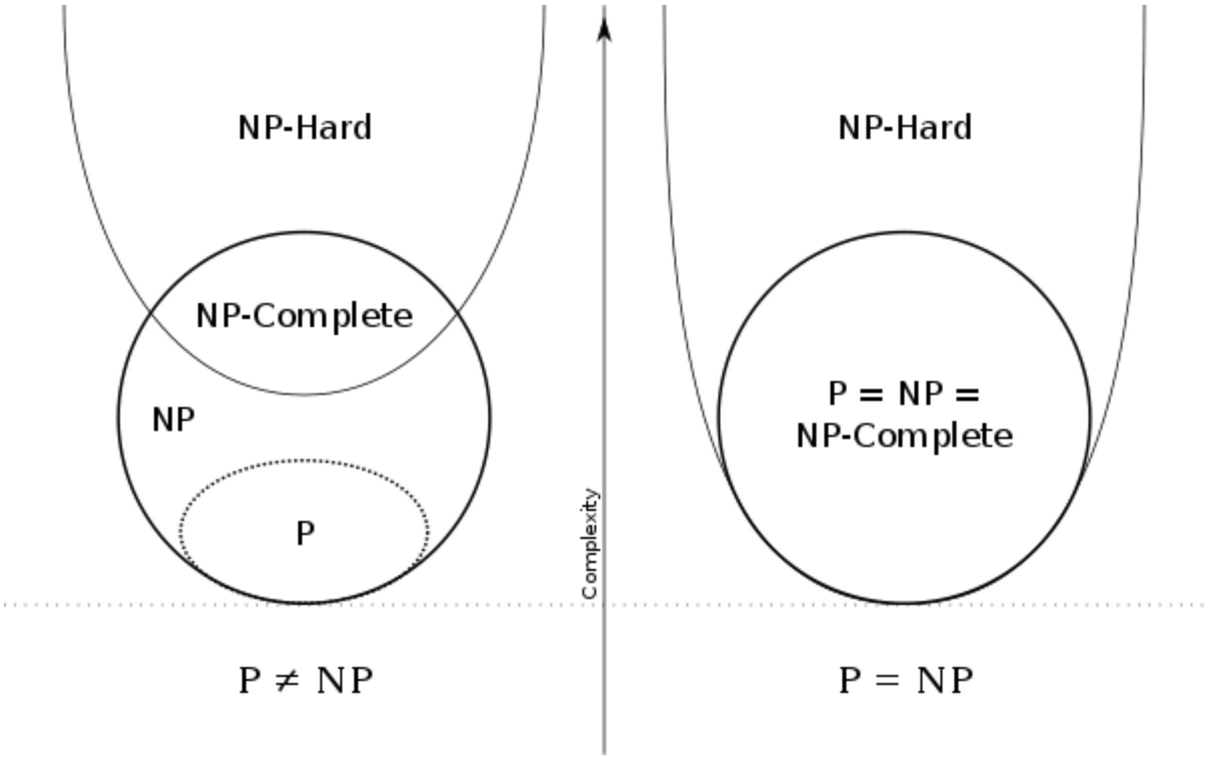
Kompleksitet/NP

P

NP

$$P \subseteq NP$$

P=NP


NP-Hard
NP-Complete
NP
P
Complexity
NP-Hard
P = NP =
NP-Complete
P ≠ NP
P = NP

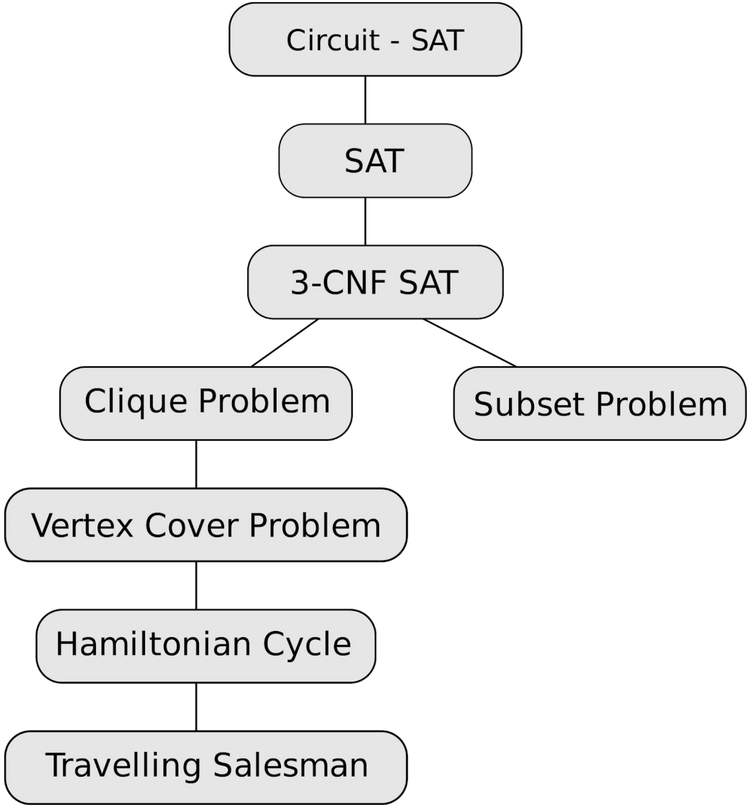
NPC

NP-Complete

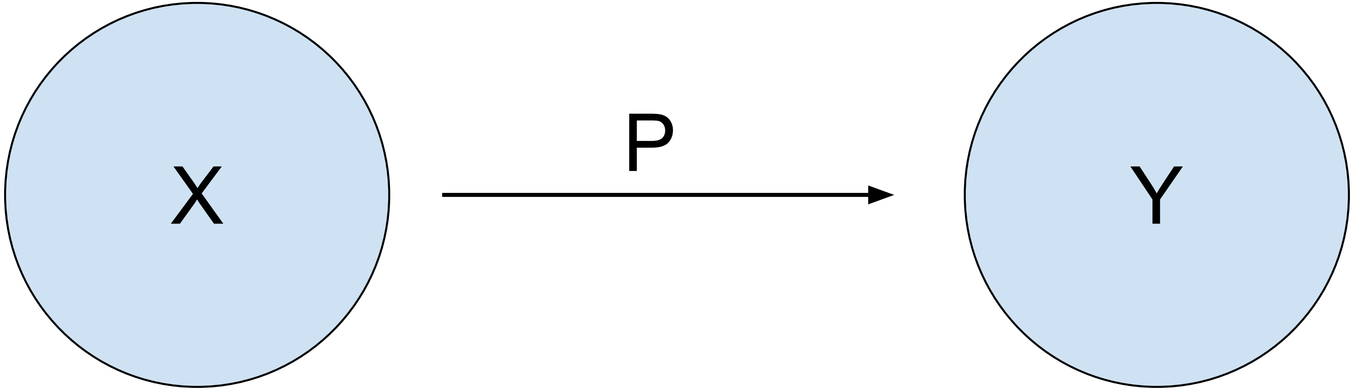
1. Må være i NP
2. Må være “minst like vanskelig som alle andre problemer i NP”

(NP-Hard hvis 2. er oppfylt, men ikke nødvendigvis 1)

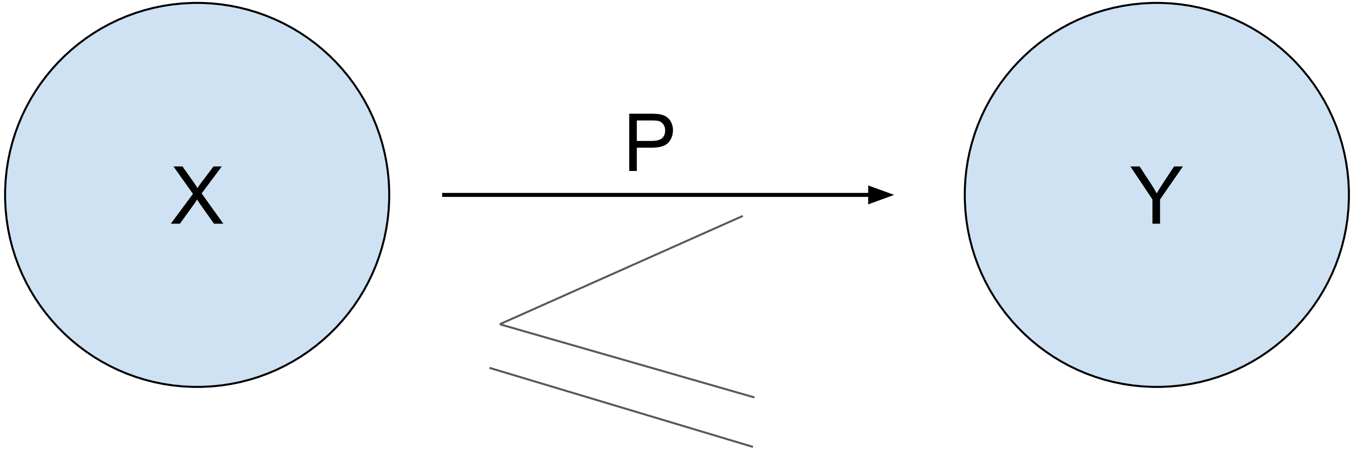
NPC


Circuit - SAT
SAT
3-CNF SAT
Clique Problem
Subset Problem
Vertex Cover Problem
Hamiltonian Cycle
Travelling Salesman

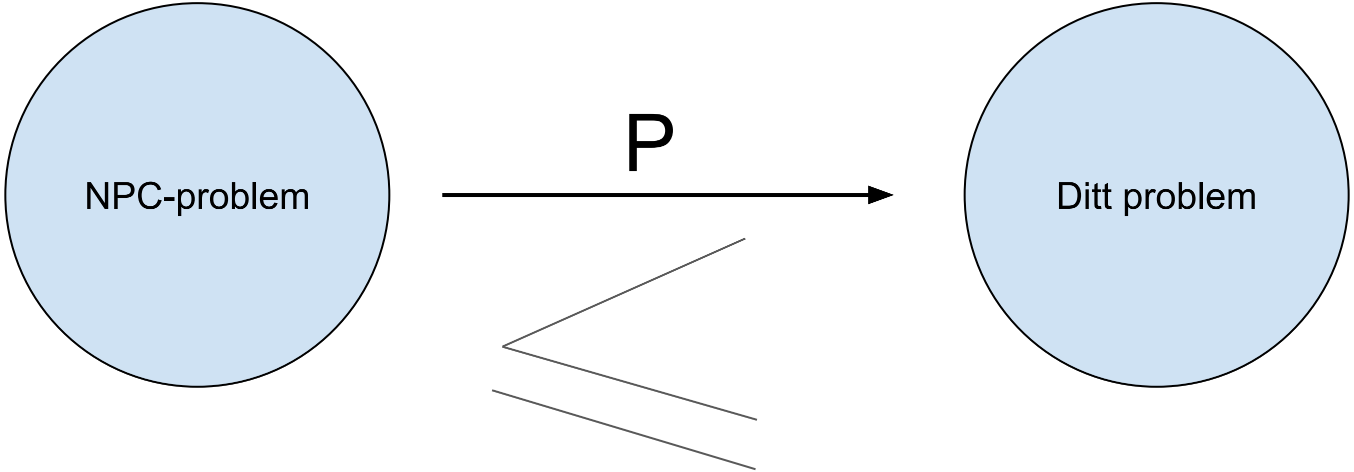
Reduksjon



Reduksjon



Hva kan vi bruke det til?



Oppgave

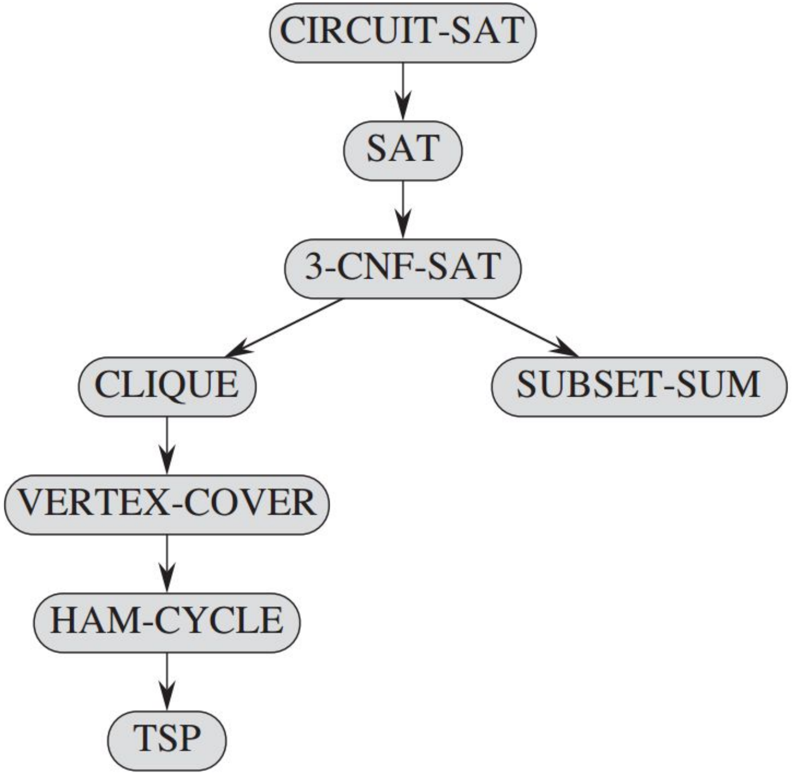
Vi har et ukjent problem A og et kjent problem B som vi vet er i mengden NP.
For å vise at A er i NP hva må vi redusere fra og til?

Oppgave

Problem A er i P og problem B er i NP.

For å vise at P = NP hva må vi redusere fra og til?

Hva vi har å ta av



Du vil invitere til en stor blind date med flere av vennene dine! KULT!
Du vil altså at ingen av vennene dine skal kjenne hverandre fra før, så du tenker hva om jeg bare ikke invitere K av vennene mine, slikat de resterende ikke kjenner hverandre. Dessverre synes du det er veldig vanskelig å finne disse K personene :(
Vis at dette problemet er np-komplett!

August 2012 h)

h) Du kan redusere fra A til B i polynomisk tid. Svarene til både A og B kan verifiseres i polynomisk tid. Fyll ut følgende scenarier (under antagelsen P ≠ NP).

Svar (6%): Kryss av i ruten (□) ved den mest spesifikke klassen som gjelder.

A er i …	**B er i …**
P	□ P / □ NPC / □ NP
NPC	□ P / □ NPC / □ NP
□ P / □ NPC / □ NP	**P**
□ P / □ NPC / □ NP	**NPC**

H2013, oppgave 9

9. Your friend claims to have invented an algorithm that solves the traveling salesman problem in $O(n \log(\log(n)))$ time, where n is the number of vertices in the graph. If your friend's claim were correct, would there be any important consequences of his or her discovery? Explain briefly.

3 Når en algoritme utføres, gjentas gjerne et «trinn» mange ganger, til vi har funnet resultatet for en gitt instans. Hvordan relaterer dette seg til dekomponering av instansen i delinstanser (*subproblems*)? Hvordan arter dette seg forskjellig for ulike designmetoder? Hvilken rolle spiller matematisk induksjon oppi det hele? Hvilket slektskap har dekomponering til reduksjoner og hardhetsbevis, og hvordan stemmer forklaringen din overens med dette?

Forklar og utdyp. Knytt til relevant teori, gjerne i ulike deler av pensum.

10. You have n friends. You know that some of your friends hate each other. Hatred is always mutual. You know exactly which pairs of friends that hate each other. You wish to unfriend some of your friends, so that none of your remaining friends hate each other. Your task is to determine if you can solve this problem by unfriending k friends, where k is a an input parameter. You can assume that this decision problem is in NP. Show that it is NP-complete. Explain your reasoning clearly, and make sure you include all required parts of such an argument.