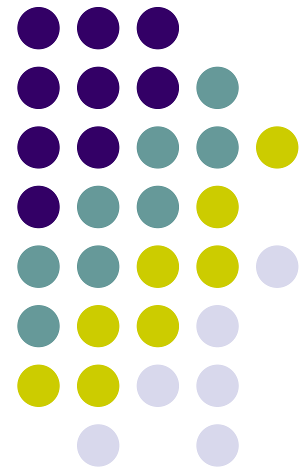
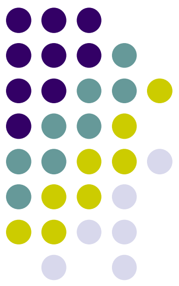
Øvingsforelesning 14

Parallellitet og oppsummering



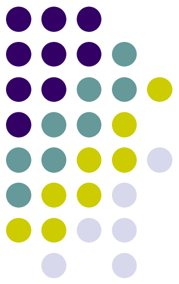
Kristian Veøy og Torbjørn Morland
algdat@idi.ntnu.no

Øvingsforelesning med Åsmund Eldhuset



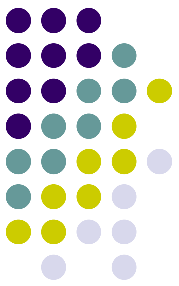
- Tidliger øvingsansvarlig Åsmund Eldhuset holder eksamenskurs lørdag og søndag
- Sted: R1
- Lørdag: 14:15
- Søndag: 12:15
- Mer informasjon kan finnes via lenke fra hjemmesiden til faget.

Dagens tema



- Parallellitet
- Oppsummering av hele pensum

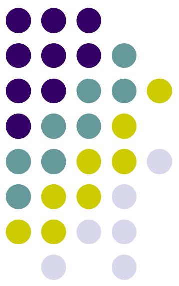
Parallellitet



- Når vi har flere rekursive kall, må vi vente på at et skal bli ferdig før neste kan utføres
- `Fib(n):`

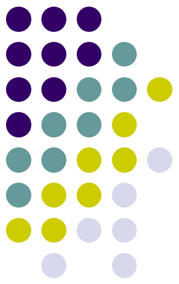
```
Fib(n):
    if n <= 1:
        return n
    else:
        x = Fib(n - 1)
        y = Fib(n - 2)
        return x + y
```

Parallellitet



- Dette kan vi unngå ved å gjøre ting parallelt
- P-Fib(n):

```
    if n <= 1:
        return n
    else:
        x = spawn Fib(n - 1)
        y = Fib(n - 2)
        sync
        return x + y
```



Parallellitet

P-FIB(4)

P-FIB(3)

P-FIB(2)

P-FIB(2)

P-FIB(1)

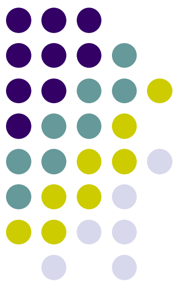
P-FIB(1)

P-FIB(0)

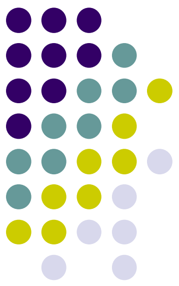
P-FIB(1)

P-FIB(0)

Parallellitet



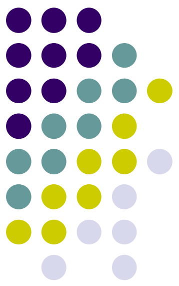
- Vanligvis vil vi finne T(n) når vi analyserer en algoritme
- Her er T_p mer interessant
  - Hva som skjer når vi har p prosessorer
- T_1 er tiden det tar med én prosessor, altså totalt arbeid
- T_∞ er tiden det tar med uendelig mange prosessorer, altså den kritiske stien



Parallellitet

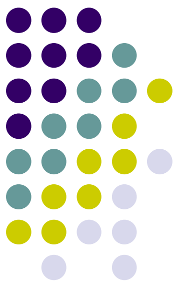
- Vi vil utføre to oppgaver, A og B?
  - $T_1(A \cup B) = T_1(A) + T_1(B)$
  - $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$
- Hva om de kan utføres parallelt?
  - $T_1(A \cup B) = T_1(A) + T_1(B)$
  - $T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$

Eksempel - fibonacci



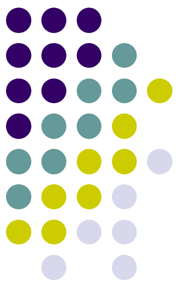
- $T_1(n) = T(n) = T(n-1) + T(n-2) + \Theta(1)$
  $= \Theta(\phi^n)$
- $T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$
  $= T_\infty(n-1) + \Theta(1)$
  $= \Theta(n)$
- Eksponensielt arbeid, men lineær kritisk sti

Parallellitet

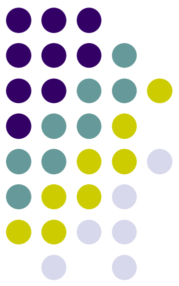


- The work law:
  - $T_p \geq T_1/P$
- The span law:
  - $T_p \geq T_\infty$
- Speedup: T_1 / T_p
  - Fra work law får vi at speedup ≤ p
  - Lineær speedup: $\frac{T_1}{T_p} \in \Theta(p)$
  - Perfekt speedup: $T_1/T_p = p$

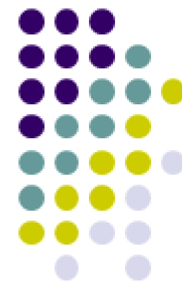
Parallellitet



- Parallellitetsgrad: T_1/T_∞
  - Gjennomsnittlig arbeid per trinn i kritisk sti
  - Maksimal speedup
- Eks: fibonacci $\frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{\phi^n}{n}\right)$
- Vokser svært fort som funksjon av n, altså skal det ikke noen stor n til før vi får nærmest perfekt speedup



Bellman-Ford	BFS/DFS	Binære søketrær	Binærsøk	Bubblesort
Bucket sort	Counting sort	DAG-shortest-path	Dijkstra	Edmonds-Karp
Floyd-Warshall	Hashing	Heapsort	Huffman-koding	Insertion sort
Kruskal	Merge sort	Prim	Quicksort	Radix sort
Randomized Select	Select	Selection sort	Sterke komponenter	Topologisk sortering



Bellman-Ford	BFS/DFS	Binære søketrær	LP	ubblesort
Bucket sort	Counting sort	DAG-shortest-path	Dijkstra	Edmonds-Kar
Floyd-Warshall	Hash	rt	Huffman-koding	Insertion
Kruskal	Merge sort	Prim	Quicksort	Radix sort
Randomized Select		Selection sort	Sterke komponenter	Topologisk sortering

D&C

+Par!

Flyt+

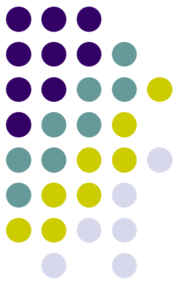
Gr.

NPC

DP

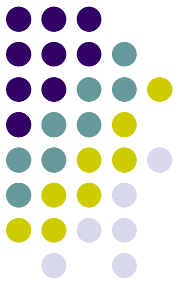
Θ etc.

Oppsummering av pensum



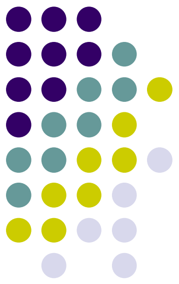
- Del 1:
  - Datastrukturer
  - Søking
  - Randomized-Select, Select
  - Sterke komponenter
  - Asymptotisk kjøretid, masterteoremet
  - Sorteringsalgoritmer
- Del 2:
  - Minimale spenntrær
  - Korteste vei
  - Designmetoder
  - Flytnettverk
  - P, NP, NPC

Del 1



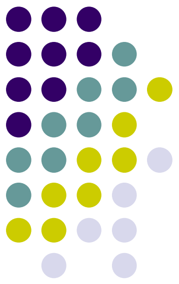
- Datastrukturer
- Søking
- Randomized-Select, Select
- Sterke komponenter, SCC
- Asymptotisk kjøretid, masterteoremet
- Sorteringsalgoritmer

Abstrakte datatyper (ADT)



- Datatyper der operasjonene defineres, men ikke implementasjonen
- Eksempler: stakk, kø, trær

Trestrukturer: Terminologi



Rot

B og C's
Foreldre-node

Kanter

Noder

R

Subtre

A

D

I

Løvnoder

B

C

F

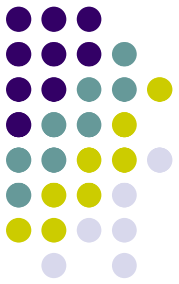
G

H

A's venstre barn

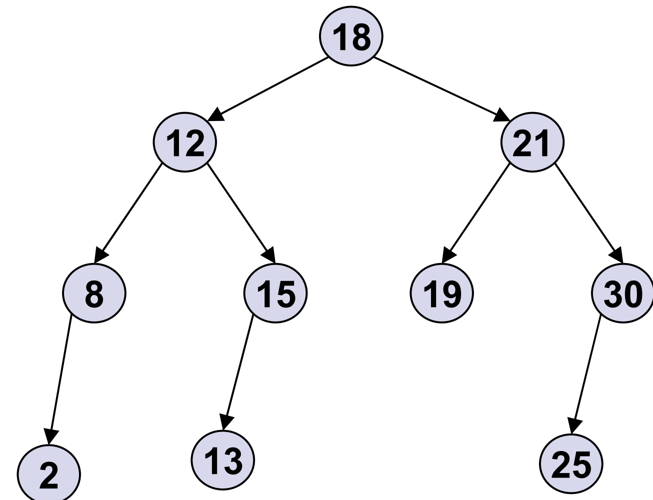
A's høyre barn

Søskennoder



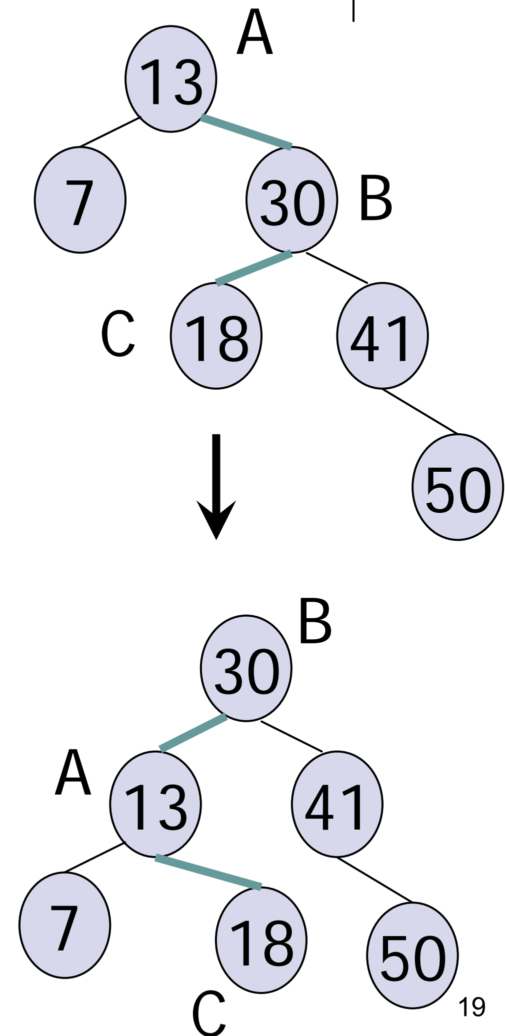
Binære søketrær

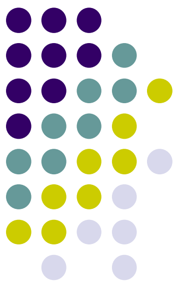
- Verdien i en node er større enn alle verdiene i venstre subtre, og mindre enn alle verdiene i høyre subtre.



Rotasjon

- Sentralt i balansering av trær.
- Venstre-rotering:
  - La node A bli B sitt venstre barn.
  - La node C bli A sitt høyre barn.
- Høyre-rotering:
  - Symmetrisk av venstre-rotering.
- Brukes av f.eks AVL-trær for å holde høydeforskjellen mellom subtrær til maks 1.

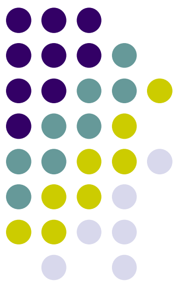




Red-Black-trær

- Selvbalanserende binært søketre
  - En node er enten rød eller sort
  - Rotnoden er sort
  - Alle løvnoder er sorte
  - Begge barn til en rød node er sorte
  - Alle stier fra en node til enhver av løvnodene som er dens etterkommer, inneholder samme antall sorte noder

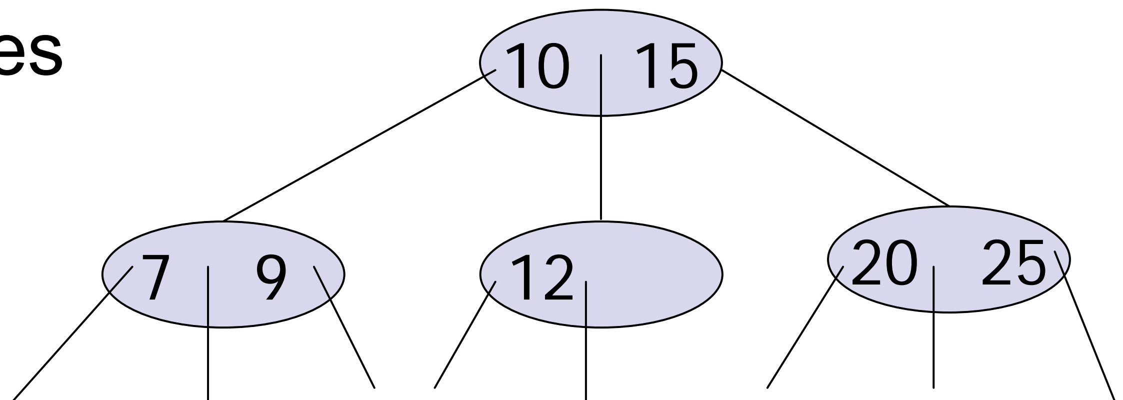
B-trær

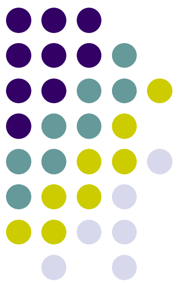


- Et søketre, men hver node har flere verdier og barn.
- Holder seg balansert ved innsetting og sletting.
- Når man legger til en verdi i en node som ikke har plass, splittes den, og

forandringen dyttes

oppover i treet.

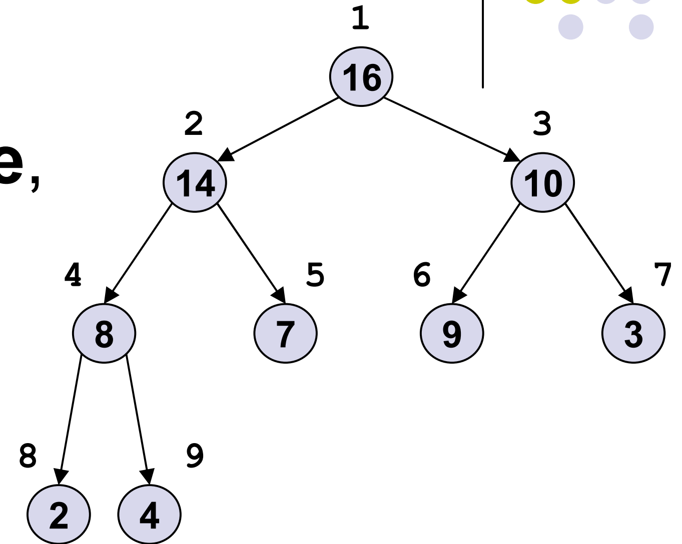




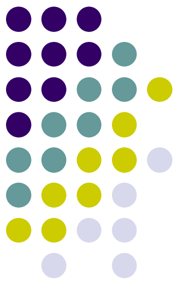
Kø og stack

- Kø:
  - Førstemann inn er førstemann ut
    - Kassakø
- Stack:
  - Sistemann inn er førstemann ut
    - Stabel med ark
- Begge to har operasjonene pop og push
  - Pop henter neste element
  - Push legger på et nytt element

Heap

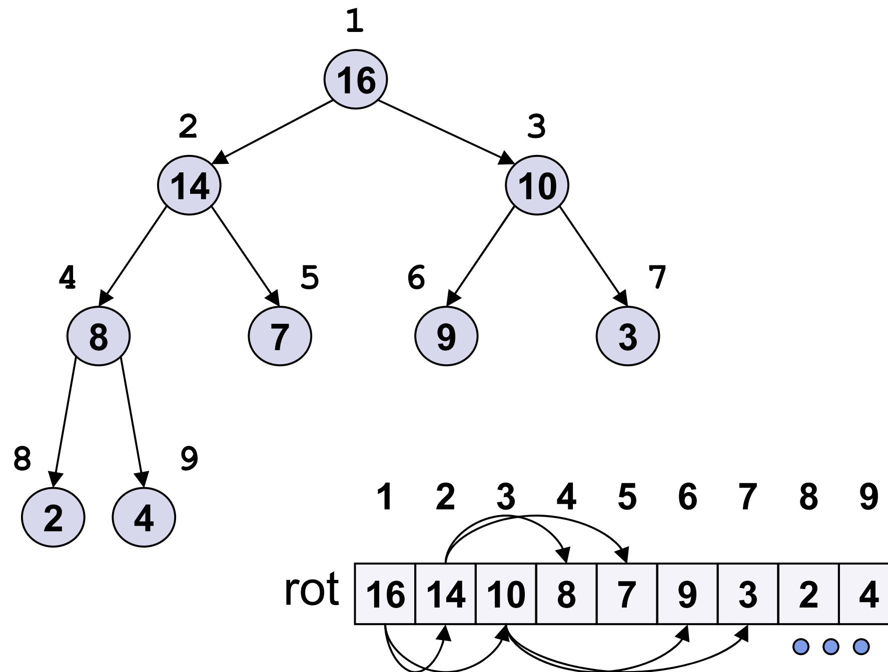


- En heap er et **komplett binærtre**,
  - der alle nivå er fylt opp,
  - unntatt eventuelt det siste,
  - som er fylt opp fra venstre til høyre
- For alle noder i en heap gjelder:
  - Barna har lavere eller lik verdi(for max-heap)
  - Barna har større eller lik verdi(for min-heap)
- En heap **brukes til/av** blant annet
  - Prioritetskøer
  - Heapsort

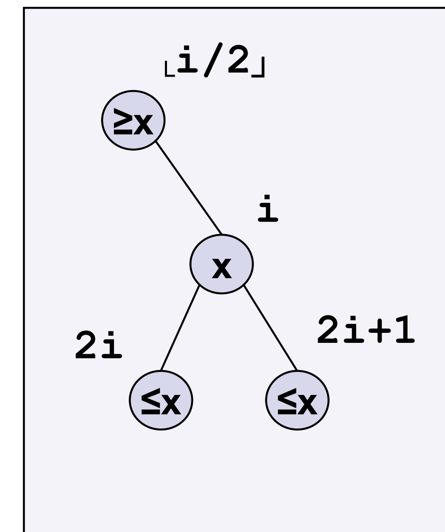


Heap

- Representasjon av binærtre i et array:

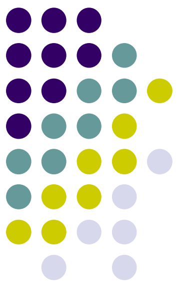


Generelle far/barn relasjoner

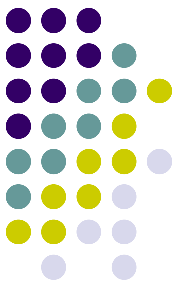


(Heap-egenskapen)

Hashing – søking i konstant tid (average case)



- Hashing er en måte å konvertere verdier fra et stort utfallsrom til et som er mye mindre.
- Hashing gir en form for fingeravtrykk av en verdi.
- Hasher kan for eksempel brukes til å angi eksakt posisjon i en tabell.

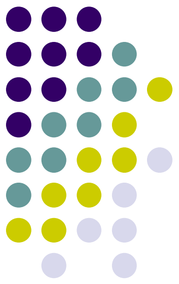


Hashtabeller: Fordelene

- Oppslag i O(1) tid
- Innsetting i O(1) tid
- Sletting i O(1) tid

- NB! Dette er average-case, ikke worst case

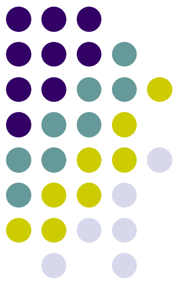
Hashing



- Hashfunksjon:
  - h(k) = x

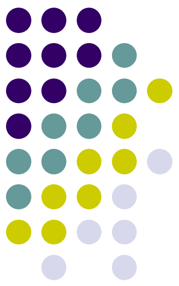
  - h er *hashfunksjonen* vi har valgt oss
  - k er *hashnøkkelen*, hele eller deler av dataene
  - x er *hashen* av nøkkelen, dvs. posisjonen der vi plasserer dataene i en *hashtabell*

Hashtabeller - krasjhåndtering



- Hva om, $h(k) = h(m)$, $k \neq m$?
  Vi har mange alternative løsninger:
  - Bruke små lenkede lister til å lagre k og m på samme plass.
  - Ha flere alternative hasher.

Hashtabeller - eksempel

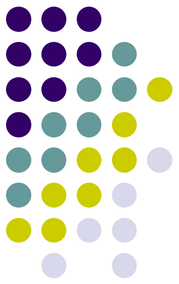


Hvis vi velger f.eks
H(x) = x mod 32
som hashfunksjon
(dvs. betrakt siste 5 bit),
får vi 32 mulige bøtter.

Asymptotisk søketid blir kongruent med forventet antall poster per bøtte (konstant).

Vi må justere antall bøtter etter inputstørrelsen for å holde forventet bøttestørrelse konstant

X:	247	57	188	72	83
	176	242	196	188	132
	4	231	168	221	21
	18	14	236	245	167
H(**X**):	10111	11001	11100	01000	10011
	10000	10010	00100	11100	00100
	00100	00111	01000	11101	10101
	10010	01110	01100	10101	00111
H(X):	23	25	28	8	19
	16	18	4	28	4
	4	7	8	29	21
	18	14	12	21	7



Hashtabeller - eksempel

Våre hasher:

23	25	28	8	19
16	18	4	28	4
4	7	8	29	21
18	14	12	21	7

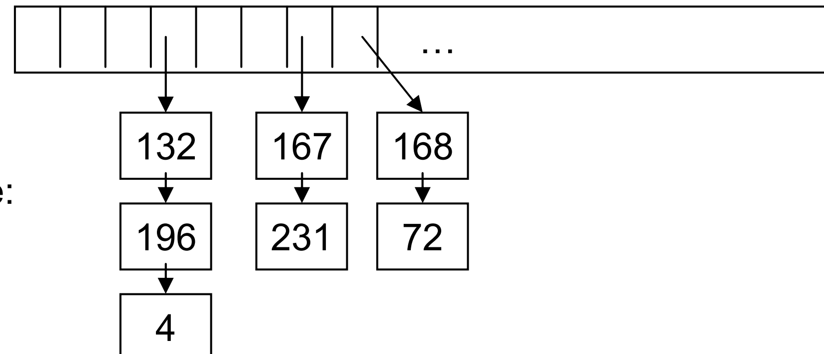
Sortert

4, 4, 4,
7, 7,
8, 8,
12,
14,
16,
18, 18,
19,
21, 21,
23,
25,
28,
28,
29

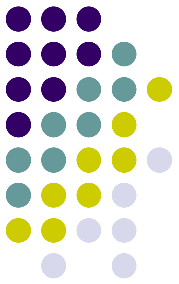
Søk i denne tabellen vil kreve 1,35 sammenligninger i snitt, hvis vi implementerer bøttene som lenkede lister.

14 av verdiene krever 1 sammenligninger
5 av verdiene krever 2 sammenligninger
1 av verdiene krever 3 sammenligninger

Tabellen vil se ut noe á la dette:

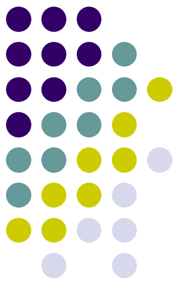


Valg av hashfunksjon



- Mål: transformere potensielt store data til en indeks i en tabell
- Påkrevd egenskap: Deterministisk
- Ønsket egenskap: Uniform fordeling
- Ønsket egenskap: Kjapp å utføre

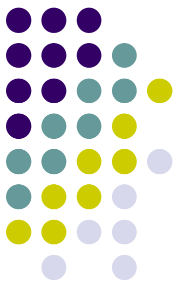
Søkealgoritmer



- Binærsøk
- Dybde-først-søk
- Bredde-først-søk
- Randomized-Select, Select

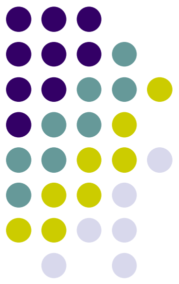
- Anvendelse av DFS: Finne sterke komponenter

Binærsøk



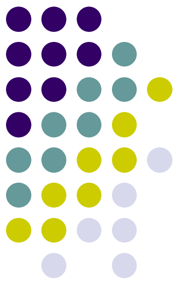
- Krever sortert tabell/binært søketre
- Prinsipp: Del hele tiden søkerommet i to, til du enten har funnet det du leter etter, eller sitter igjen med ett element
- Hvordan? Sjekk det midterste elementet, eliminér den siden som ikke kan være med

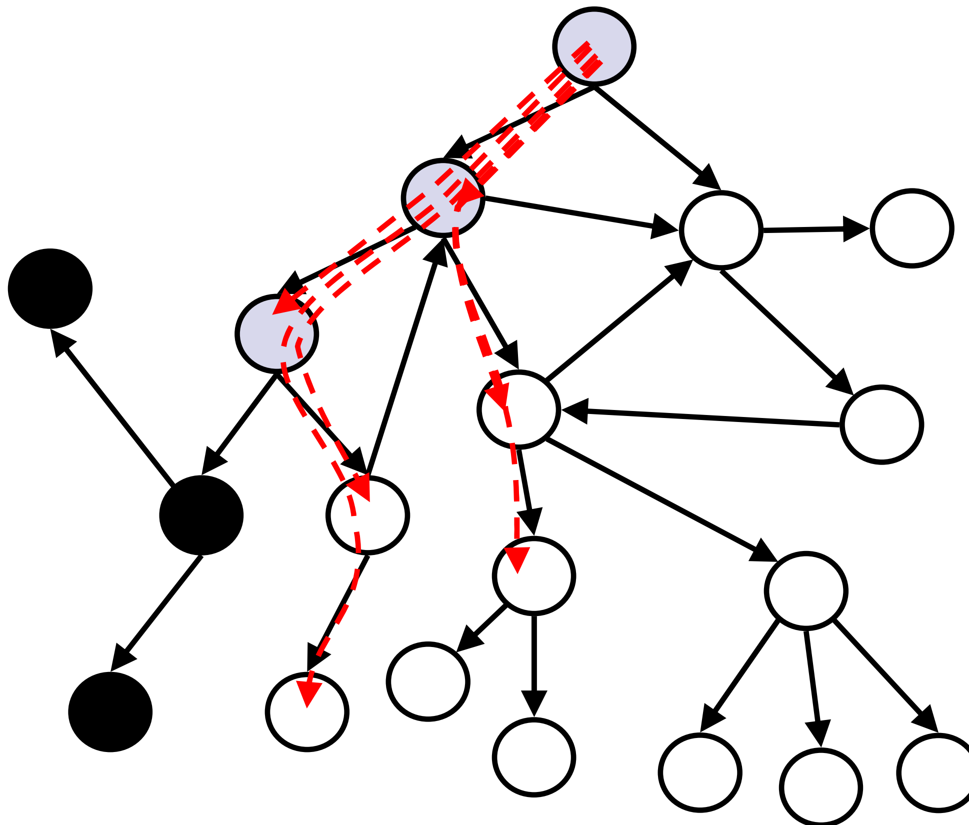
Kort om Dybde Først Søk (DFS)



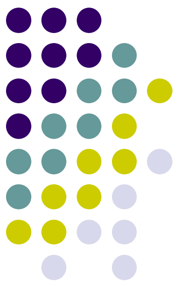
- Et søk som traverserer alle noder i en graf.
- Algoritmen fullfører hver eneste node den oppdager når den oppdager dem. (bruker LIFO-stakk)
- Går kun innom noder som ikke er oppdaget ennå.
- Viktig komponent i mange andre algoritmer. (SCC og topologisk sortering bl.a.)
- Vi bruker fargene hvit/grå/svart for å merke at nodene er uoppdaget/påbegynt/ferdig

DFS i aksjon



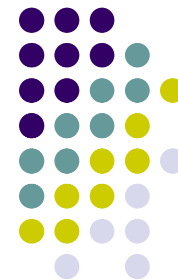


Merk!
Finner ikke nødvendigvis alle noder i en rettet graf, men noder som kan nås fra noden man starter i.

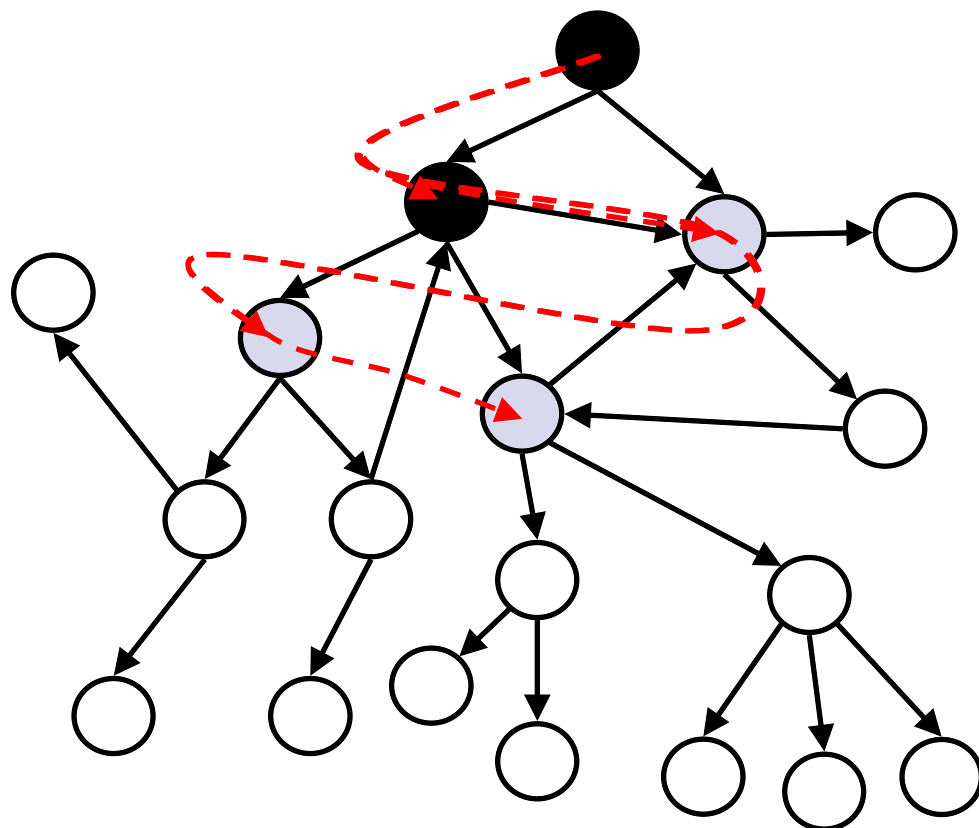


Kort om Bredde Først Søk (BFS)

- Et søk som traverserer alle noder i en graf
- Algoritmen bruker en kø hvor den setter inn alle nyoppdagede noder hver gang den besøker en node.
- Nodene blir dermed fullført etter økende avstand fra start

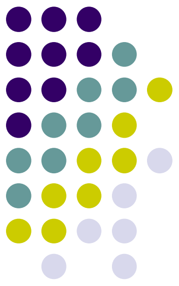


BFS i aksjon

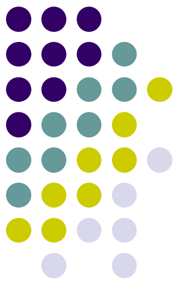


Merk!
Finner ikke nødvendigvis alle noder i en rettet graf, men noder som kan nås fra noden man starter i.

Strongly Connected Components

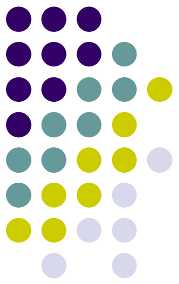


Strongly Connected Components



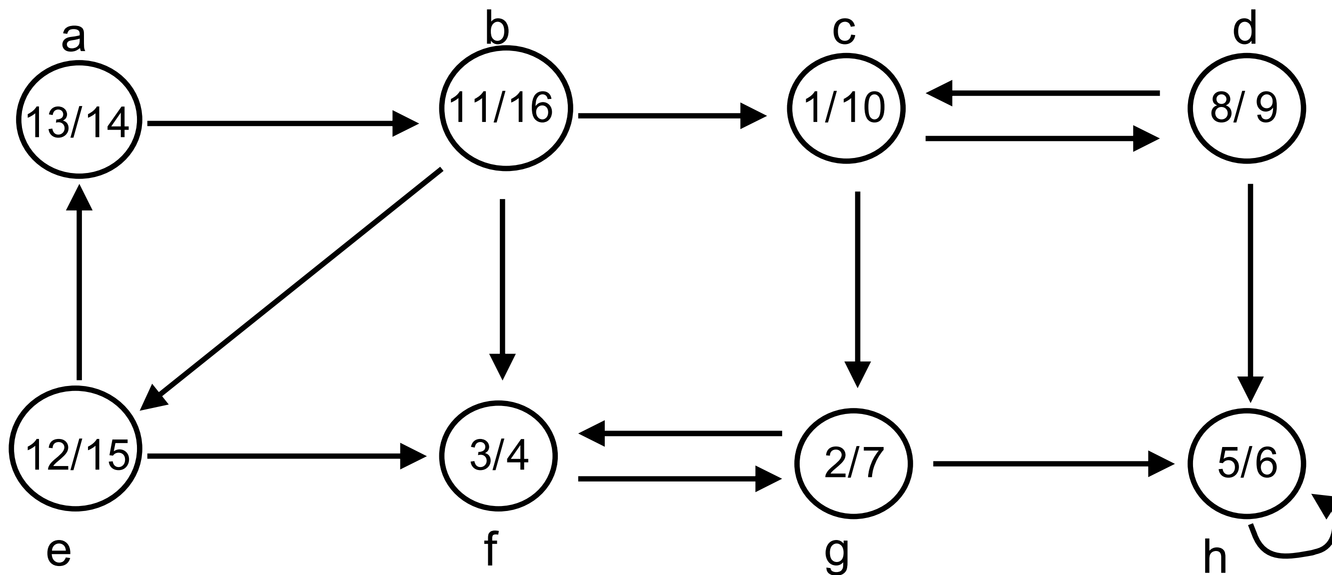
- Problemstilling:
  - Finne alle SCC'er i en graf
- Algoritme:
  - Kjør DFS på G for å finne alle finish-tider
  - Kjør DFS på Gt men velg startnoder med synkende finish-tid
  - Resulterende dybde først skog består av ett tre for hver SCC

Strongly Connected Components



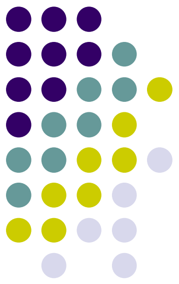
Algoritme:

1: Kjør DFS på G for å finne alle endetider.
2: Kjør DFS på G^T men velg utgangsnoder med synkende endetid.
3: Resulterende dybde først skog består av et tre for hver SCC.



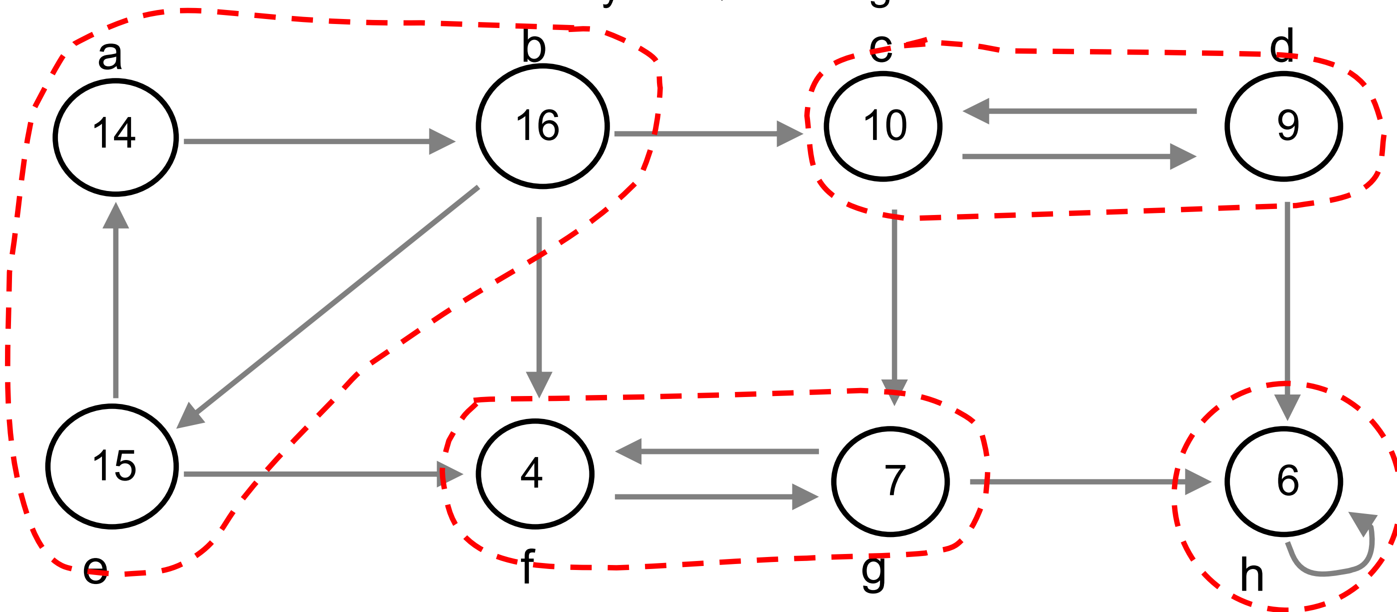
(s. 553 i Cormen)

Strongly Connected Components

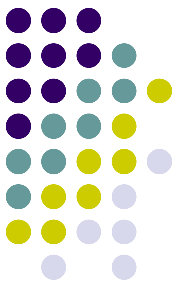


Algoritme:

1: Kjør DFS på G for å finne alle endetider.
2: Kjør DFS på G^T men velg utgangsnoder med synkende endetid.
3: Resulterende dybde først skog består av et tre for hver SCC.

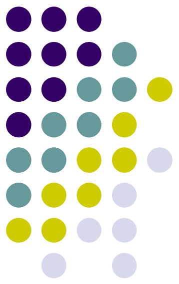


Seleksjon i en liste



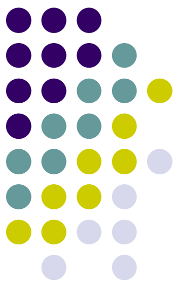
- Select:
  - Del i grupper på 5
  - Finn medianen i hver (med Insertion Sort)
  - Finn medianen til medianene med Select
  - Bruk denne i Partition
  - Kjør Select rekursivt på riktig halvdel
- Randomized select:
  - Plukker tilfeldig pivot til partition
  - Dårligere worst-case enn median-av-median

Asymptotisk kjøretid



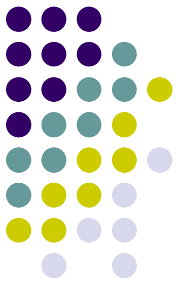
- Hva er asymptotisk kjøretid
  - Omega, Theta, Big-O
- Rekurensligninger
- Masterteoremet

Asymptotisk kjøretid



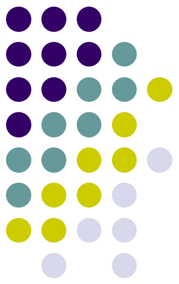
- $\Theta(g(n)) = \{f(n)$: det finnes positive konstanter c_1, c_2 og n_0 slik at $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for alle $n \geq n_0\}$
- Altså: $\Theta(g(n))$ består av alle funksjoner som $g(n)$ kan bruke som både øvre og nedre grense

Asymptotisk kjøretid



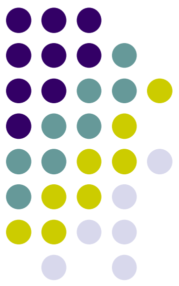
- Hvis vi bare vil si at g er en *øvre* grense for f, bruker vi *O*-notasjon: $f(n) = O(g(n))$
- Hvis vi bare vil si at g er en *nedre* grense for f, bruker vi Ω-notasjon: $f(n) = \Omega(g(n))$

Kjøretidsanalyse av rekursive funksjoner



1. Prøv å sette inn uttrykk for de rekursive stegene i hverandre, for å kunne gjette et uttrykk for T(N)
2. Bevis at hvis det stemmer for N, så vil det også stemme for N+1

Foroversubstitusjon



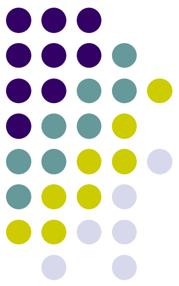
T(N) = 2T(N-1) + 1, T(1) = 1

T(2) = 2T(1) + 1
T(3) = 2(2T(1) + 1) +1
T(4) = 2(2(2T(1) + 1) +1) +1

- På denne måten får man et mønster som hjelper oss til å gjette hvordan kjøretiden vokser.
- Man kan for eksempel gjette at T(4) er $2^3+2^2+2^1+2^0$

Dette er

$$T(N) = \sum_{i=0}^{N-1} 2^i = 2^N - 1$$



Tilbakesubstitusjon

- I utgangspunktet det samme som foroversubstitusjon, men man går bakover, I stedet for forover.

  $T(N) = 2T(N-1) + 1$

  $T(N) = 2(2T(N-2) + 1) +1$

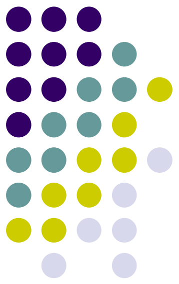
  $T(N) = 2(2(2T(N-3) + 1) +1) +1$

  $T(N) = 2T(N-1) + 1$

  $T(N) = 2^2T(N-2) + 2 + 1$

  $T(N) = 2^3T(N-3) + 2^2 + 2 +1$

Induktiv test/bevis

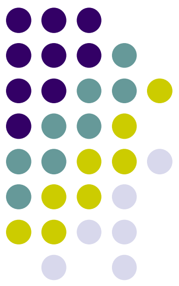


Vi har gjettet at $T(N) = 2^N - 1$

Induktivt steg:

$$
\begin{aligned}
T(N+1) &= 2(T(N)) + 1 \\
&= 2(2^N - 1) + 1 \\
&= 2^{N+1} - 2 + 1 = 2^{N+1} - 1
\end{aligned}
$$

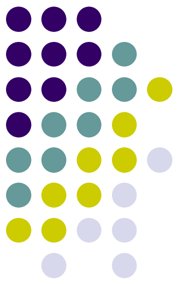
Tremetoden, eksempel 1



$T(0) = O(1)$

$T(n) = T(n-1) + T(n-1) + 1$

- For hver dekrementering av T()'s argument får vi dobbelt så mange kall
- I hvert kall må vi gjøre O(1) arbeid.



Tremetoden

T(N)

T(N-1)

T(N-2)

T(N-3)

4

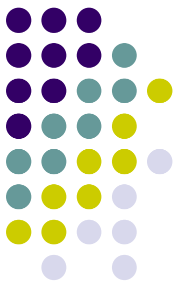
2^4-1

2^4

Vi ser at T(4) får kjøretid $2^4 + 2^4 - 1 = 2^5 – 1$

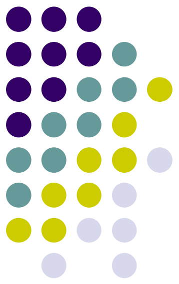
Vi ser at T(N) blir $2^{N+1}-1 \in O(2^N)$

Rekursiv funksjon, eksempel 2

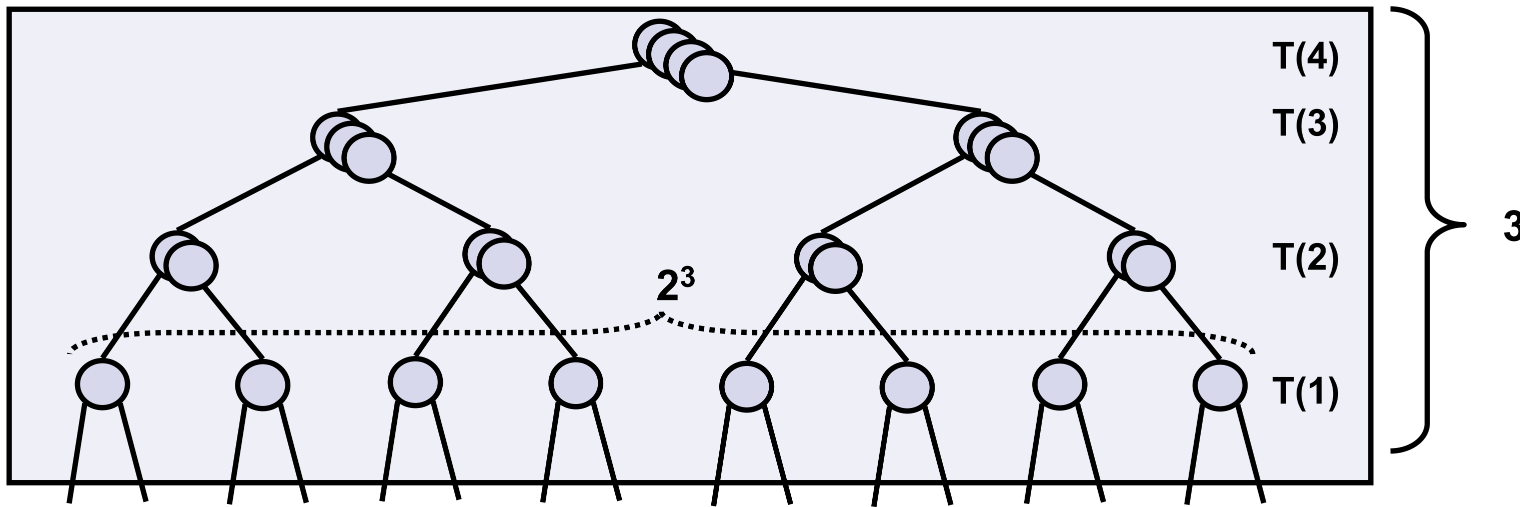


T(0) = 0
T(N) = T(N-1) + T(N-1) + O(N)

- For hver dekrementering av T()’s argument får vi dobbelt så mange kall.
- Totalt arbeid er $\sum_{i=0}^{n} 2^i (n-i)$
- For hver dekrementering må vi gjøre maksimalt 2^{N-1} arbeid.
  - hint: siste ledd er $2^{N-1}(1)$, og det er det garantert største.
- Vi får altså $T(N) \in O(N*2^N)$



Tremetoden



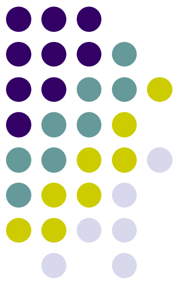
Her er det markert hvilke noder som representerer mest arbeid.

For eksempel koster T(4) 4 arbeid.

Vi ser at hvert nivå har maksimum 2^{N-1} arbeid og vi har N-1 nivå.

Altså er $T(N) \leq (N-1)2^{N-1} \in O(N2^N)$

Masterteoremet

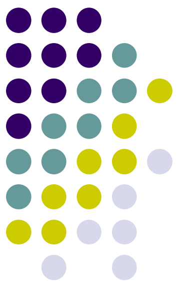


- Masterteoremet kan brukes I alle tilfeller hvor tidskompleksiteten kan uttrykkes som:

$$T(n) = aT(n/b) + f(n)$$

- Det angir 3 løsninger for 3 ulike tilfeller av $f(n)$

Masterteoremet



$$T(n) = aT(n/b) + f(n)$$

$$f(n) = O(n^{\log_b a - \varepsilon}), \quad \varepsilon > 0 \quad \rightarrow \quad T(n) \in \Theta(n^{\log_b a})$$

$$f(n) = \Theta(n^{\log_b a}), \quad \rightarrow \quad T(n) \in \Theta(n^{\log_b a} \log n)$$

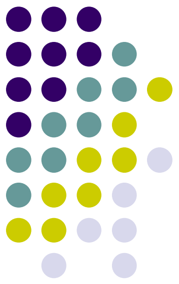
$$f(n) \in \Omega(n^{\log_b a + \varepsilon}), \quad \varepsilon > 0, \quad af(n/b) \leq cf(n), \quad c < 1 \quad \rightarrow \quad T(n) \in \Theta(f(n))$$

Eksempel:
T(n) = 2T(n/2) + 1

Vi ser at
$f(n) = 1 \in O(n^{\log_2 a - \varepsilon})$

hvis ε = ½.
da får vi T(n) ∈ Θ(n)

Sorteringsalgoritmer:



Sammenligning:

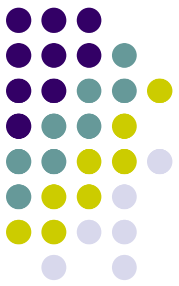
- $O(n^2)$
  - Bubble sort
  - Insertion sort
  - Selection sort
- $O(n \lg n)$
  - Heapsort
  - Merge sort
  - Quicksort

Uten sammenligning:

(må anta litt om input)

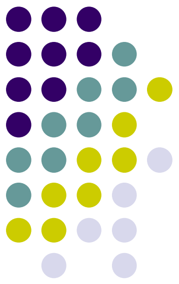
- $O(n)$
  - Counting sort
  - Radix sort
  - Bucket sort

Nedre grense for sortering ved sammenligning



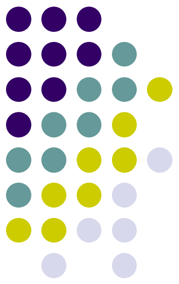
- Hver sammenligning halverer antall muligheter.
- Vi starter med n! mulige rekkefølger.
- Beste «worst-case» blir lg(n!)
  - ”Vi ser lett at” lg(n!) er $\Omega(n \lg n)$

Bubblesort



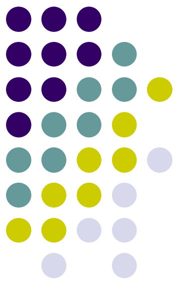
- ”Sjekker” definisjonen av sortert:
- X1 <= X2 <= X3 <= ... <= Xn-1 <= Xn
- Hvis et usortert par oppdages: Xi > Xi+1
  - Bytt om på de to verdiene.
- Gjenta helt til ingen usorterte par oppdages.

Insertion sort:

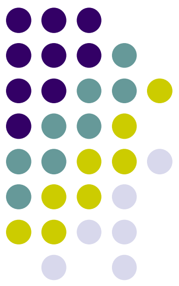


- Del input i to deler:
  - En sortert del. (Først tom)
  - En usortert. (Alle elementene ligger her i starten)
- Så lenge det finnes usorterte elementer:
  - Sett et usortert element på riktig plass i den sorterte delen.

Selection sort:



- Del input i to deler:
  - En sortert del
  - En usortert
- Så lenge det finnes usorterte elementer:
  - Finn største element i den usorterte delen, og flytt dette foran i den sorterte delen.

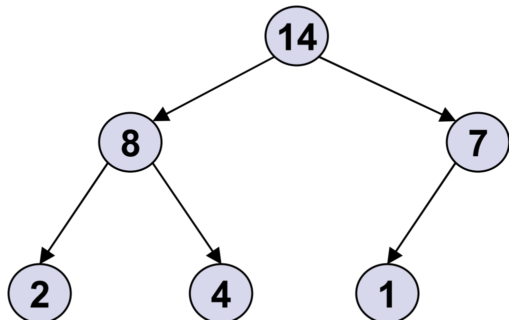


Heapsort-Algoritmen:

- Lag heap av tallene (i arrayet)
- For siste posisjon til 2.:
  - Bytt med rota, og la heap-størrelsen minke med 1
  - Kall Max-heapify på rotnoden for å gjenopprette heapegenskapen.
- Selection sort, men med heap.

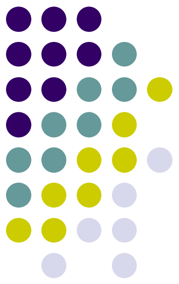
Eksempel

14	8	7	2	4	1




```
Heapsort(A)
  for i = [length[a]] downto 2
    do exchange A[1]↔ A[i]
      heap-size[A] = heap-size[A]-1
      Max-Heapify(A,1)
```

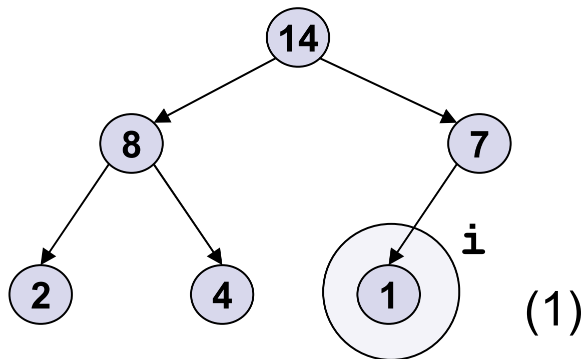
Heapsort : O(n*lg n)



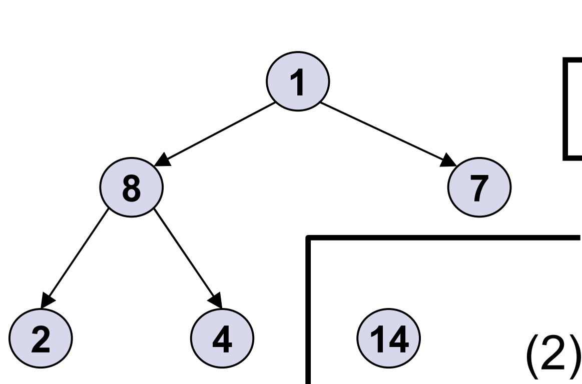
Eksempel

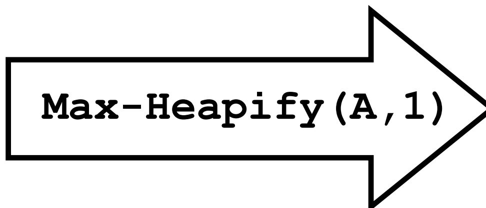
14	8	7	2	4	1

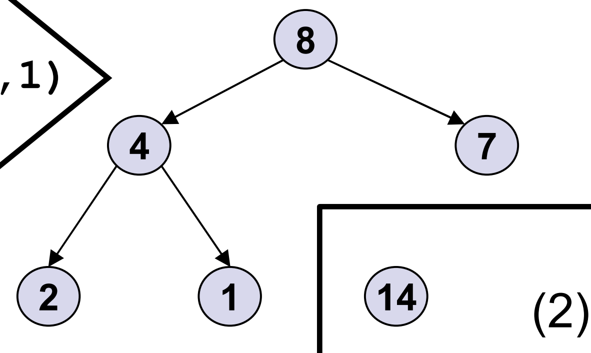
Usortert



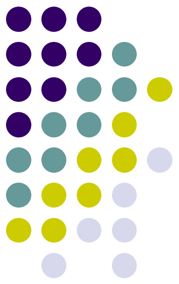

```
Heapsort(A)
  for i = [length[a]] downto 2
    do exchange A[1]↔ A[i]
      heap-size[A] = heap-size[A]-1
      Max-Heapify(A,1)
```



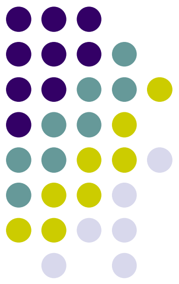




Merge sort:

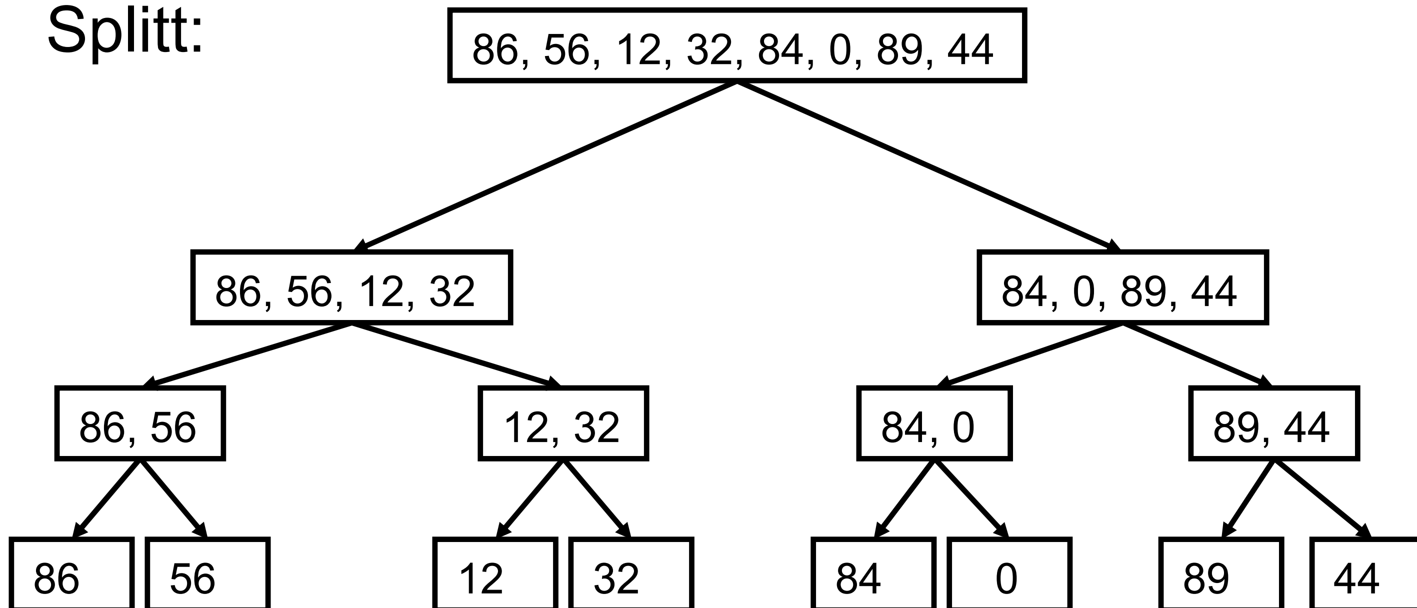


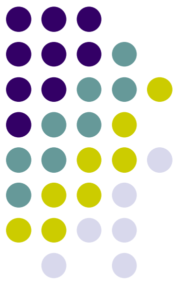
- Splitt og hersk-sortering.
- Del input i to like store biter.
- Kjør mergesort rekursivt på hver av de to bitene.
- Flett (merge) de to bitene sammen:
  - Velg det minste av bitenes første elementer helt til begge er tomme.
- Returner den flettede listen.



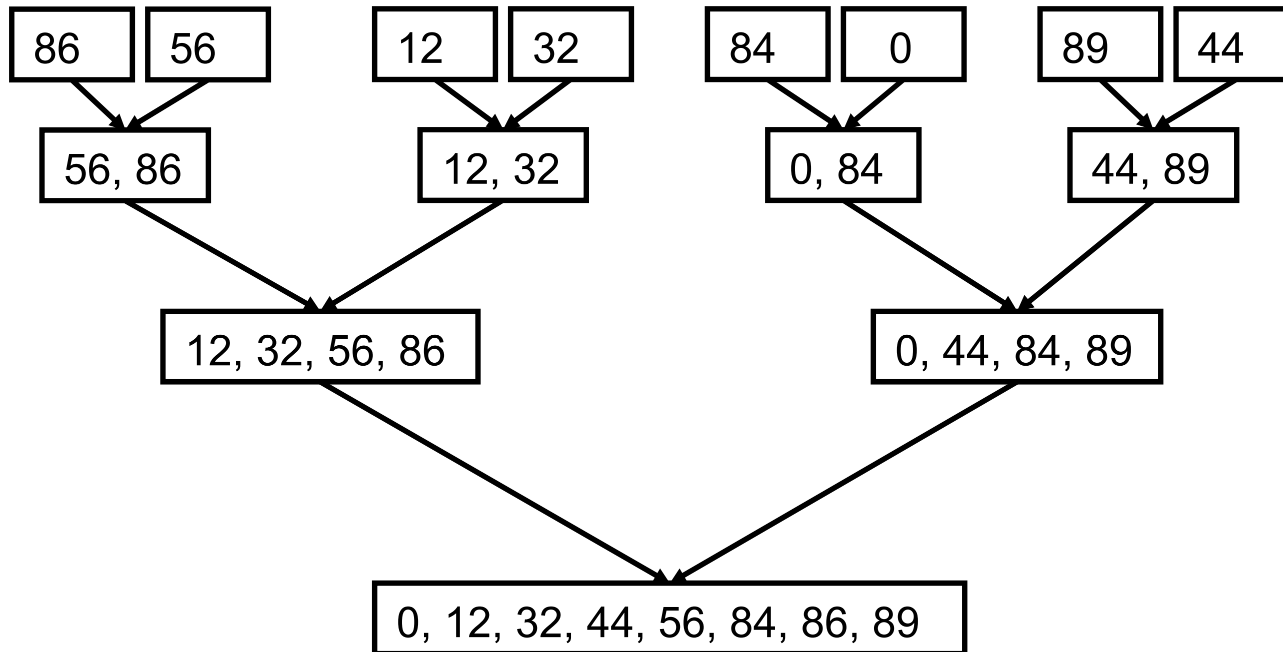
Eksempel: merge sort

Splitt:

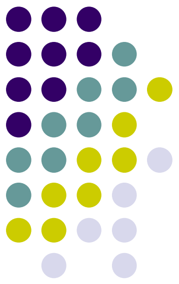




Hersk(Merge):

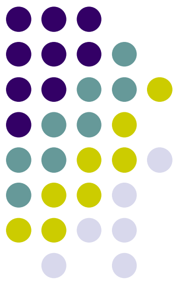


Quicksort:



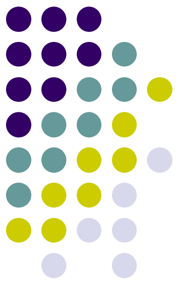
- Til forveksling lik mergesort, men:
  - ”Hersker før splitt”
  - Bruker mindre minne
- Istedet for fletting (merge) brukes partisjonering.

Quicksort: Partisjonering



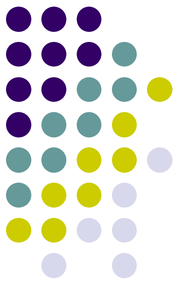
- Velger et element i input som kalles ”pivot element”
- Deler resten av input i to deler:
  - En del der alle tallene er mindre eller lik pivot elementet
  - En annen del der alle tallene er større enn pivot elementet.

Quicksort: fremgangsmåte

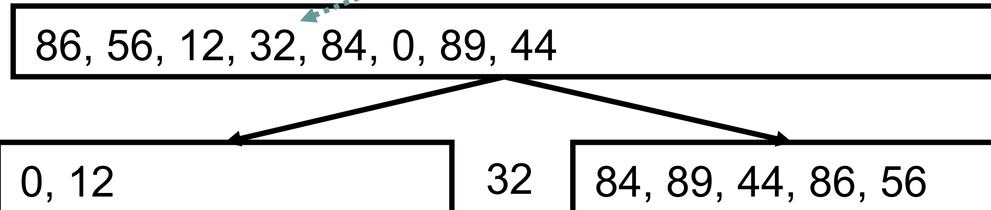


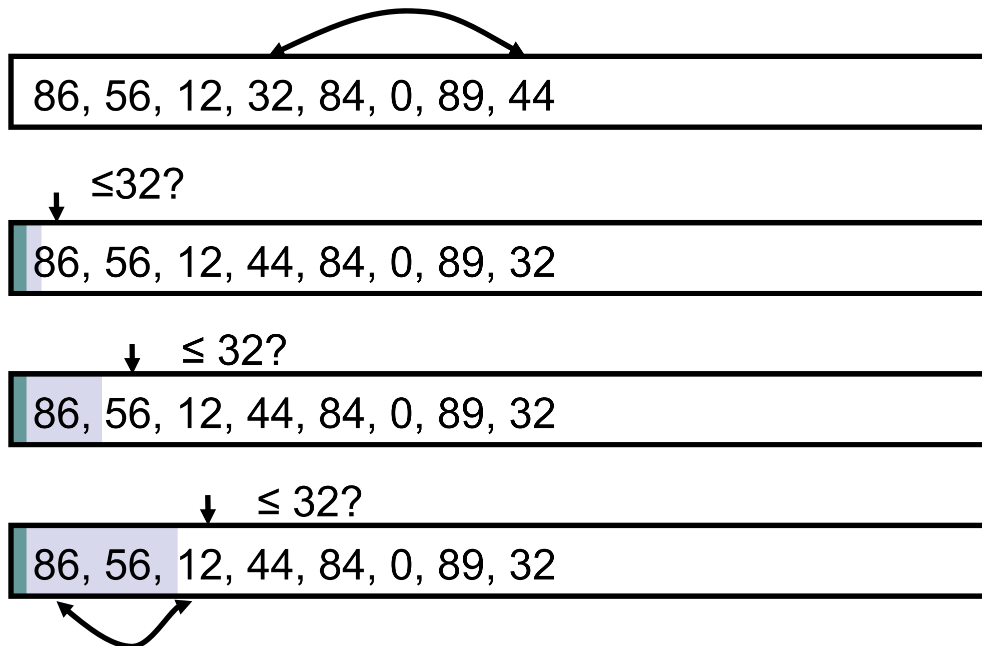
- Quicksort partisjonerer input og kaller seg selv på hver av de to partisjonene.
- Quicksort avslutter når det bare er ett element igjen.

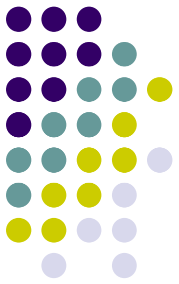
Quicksort

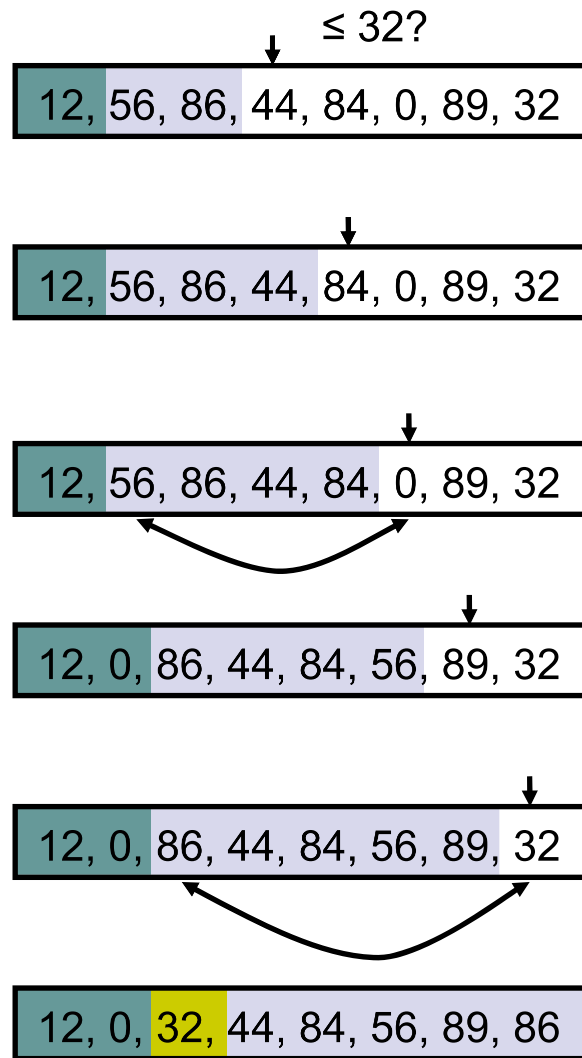


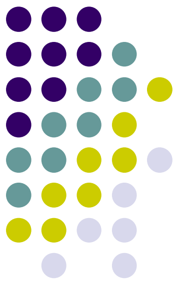
Hva skjer her?








≤ 32?
12, 56, 86, 44, 84, 0, 89, 32
12, 56, 86, 44, 84, 0, 89, 32
12, 56, 86, 44, 84, 0, 89, 32
12, 0, 86, 44, 84, 56, 89, 32
12, 0, 86, 44, 84, 56, 89, 32
12, 0, 32, 44, 84, 56, 89, 86



Pivot element

86, 56, 12, 32, 84, 0, 89, 44

0, 12

32

84, 89, 44, 86, 56

0

12

44

56

84, 89, 86

84

86

89

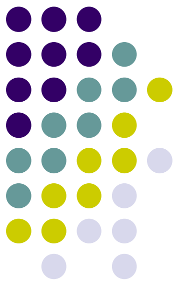
84, 86, 89

0, 12

44, 56, 84, 86, 89

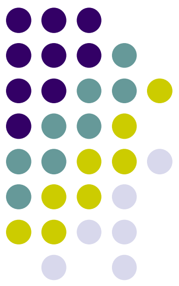
0, 12, 32, 44, 56, 84, 86, 89

Telle-Sortering (Counting-Sort)



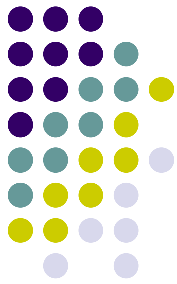
- Sorteringsrutinen antar at hvert av de n-elementene, som skal sorteres, er **heltall mellom 1 og k**, der k er et heltall
- **Idé**: For hvert element x skal vi finne antall elementer **mindre enn eller lik x**.
  - Informasjonen brukes til å plassere x direkte i det sorterte arrayet

Plassforbruk Tellesortering



- Sorteringsrutinen **krever stor plass** da 3 array brukes
  - Array A[1..n] som skal sorteres
  - Array B[1..n] er sortert resultat
  - Array C[1..k] er midlertidig arbeidslager
- Algoritmen **sorterer i lineær tid**
  - $\Theta(n)$, dersom $k = O(n)$
  - Generelt: $\Theta(n + k)$

Eksempel



	1	2	3	4	5	6	7	8	j
A	3	6	4	1	3	4	1	4	

Array som skal sorteres

	1	2	3	4	5	6
C	2	0	2	3	0	1

Antall 4'ere i A

	1	2	3	4	5	6
C	2	2	4	7	7	8

Antall elementer ≤ 3

j = 8

	1	2	3	4	5	6	7	8
B							4	

	1	2	3	4	5	6
C	2	2	4	6	7	8

j = 7

	1	2	3	4	5	6	7	8
B		1					4	

	1	2	3	4	5	6
B	1	2	4	6	7	8

j = 6

	1	2	3	4	5	6	7	8
B		1				4	4	

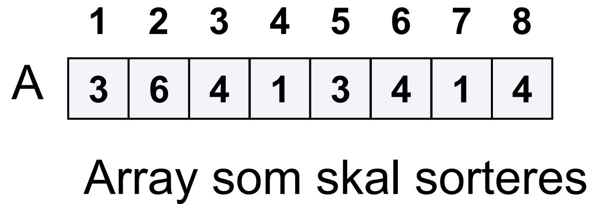
	1	2	3	4	5	6
C	1	2	4	5	7	8

j = 5

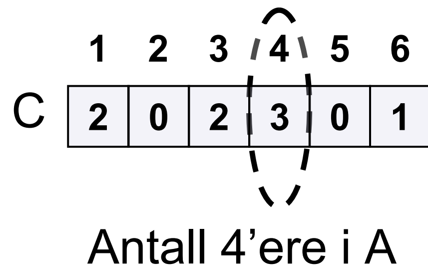
	1	2	3	4	5	6	7	8
B		1		3		4	4	

	1	2	3	4	5	6
C	1	2	3	5	7	8

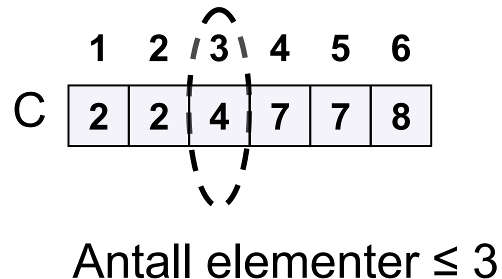
Eksempel



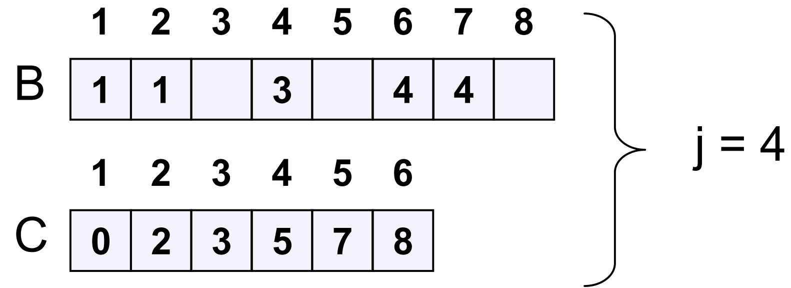
Array som skal sorteres

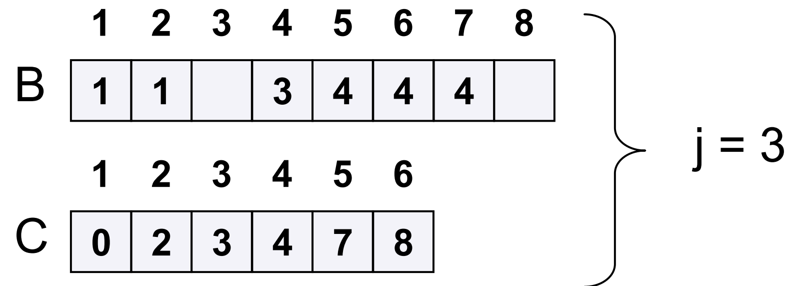


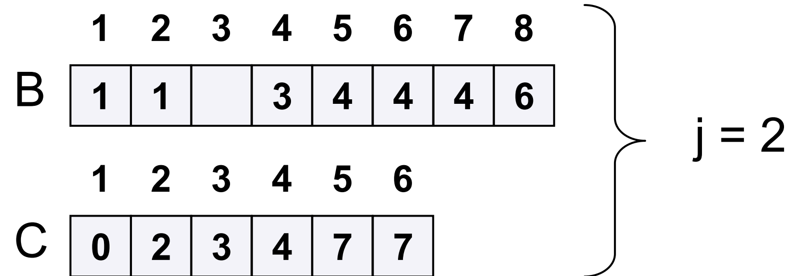
Antall 4'ere i A

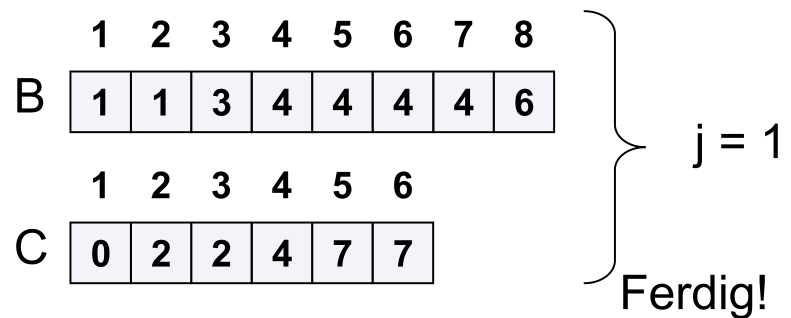


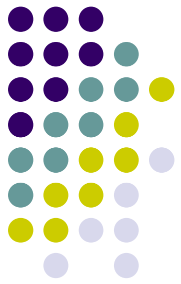
Antall elementer ≤ 3

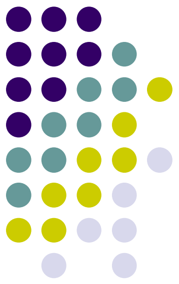






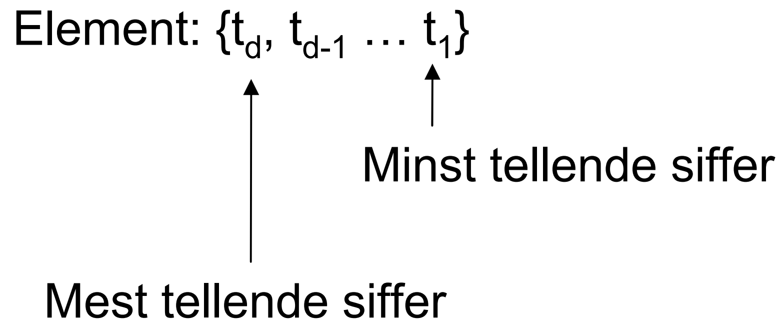




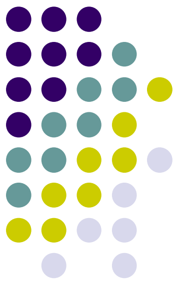


Radix-sort

- Radix-sort sorterer input i flere omganger.
- Først på minst tellende siffer, så nest minst tellende, osv til den til slutt sorterer på mest tellende siffer.



Radix-sort



- Sorterer bits fra høyre til venstre

```
Radix-Sort(A,d)
  for i=1 to d
     do use a stable sort to sort
    array A on digit i
```

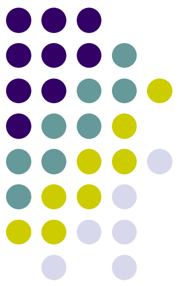
0	1	0
0	0	0
1	0	1
0	0	1
1	1	1
0	1	1
1	0	0
1	1	0

0	1	0
0	0	0
1	0	0
1	1	0
1	0	1
0	0	1
1	1	1
0	1	1

0	0	0
1	0	0
1	0	1
0	0	1
0	1	0
1	1	0
1	1	1
0	1	1

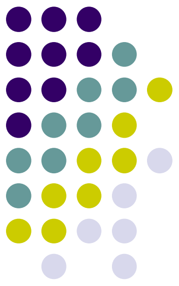
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Bucket sort:



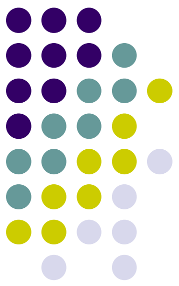
- Antar at input er jevnt fordelt over et intervall
- Idé:
  - Vi deler opp intervallet i $k = \Theta(n)$ like store bøtter, for eksempel $k = n$.
  - Elementene som skal sorteres puttes i sine respektive bøtter, de minste elementene i den minste bøtta osv.
  - Hver bøtte sorteres med insertion sort.
  - Bøttene settes sammen til det ferdige resultatet.

Kjøretid



- Innsetting i bøtter tar O(n) worst case.
- Sorteringen i hver bøtte er avhengig av hvor mange elementer som finnes i den enkelte bøtte.

Oppsummering kvaliteter



- Stabil

  En stabil sorteringsalgoritme tar vare på den eksisterende rekkefølgen til elementer som har samme verdi

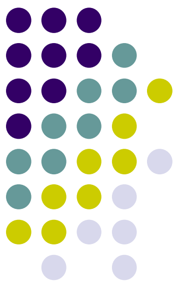
- Parallelliserbar

  Sorteringen kan splittes opp i deler som kan utføres uavhengig av hverandre

- In place

  Elementene kan befinne seg i inputlista under hele sorteringen

Topologisk sortering



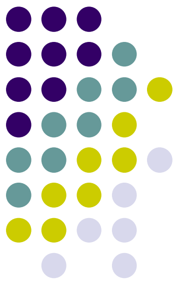
Problemstilling:

Finn en ordning av alle noder i en DAG slik at alle noder som er foreldre-noder listes opp før sine etterkommere.

Algoritme:

Kjør DFS på alle noder og legg dem på en stack etter hvert som de fullføres.

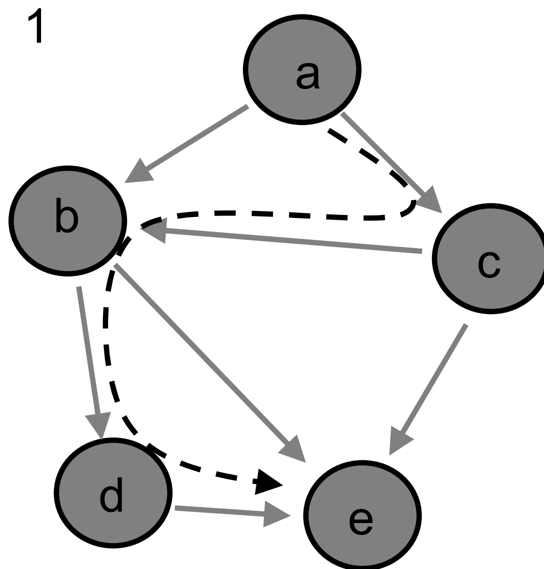
PS! DFS på en vilkårlig graf uten sykler starter på en vilkårlig node som ikke er besøkt. Dette gjøres om igjen helt til alle noder er besøkt.



Topologisk sortering

Algoritme:

Kjør DFS på alle noder og legg dem
først i en liste etter hvert som de fullføres.



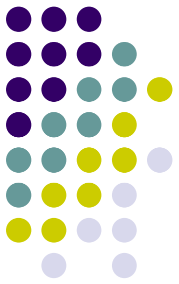
DFS på graf 1:

1:	[e]
2:	[d, e]
3:	[b, d, e]
4:	[c, b, d, e]
5:	[a, c, b, d, e]

Asymptotisk kjøretid:

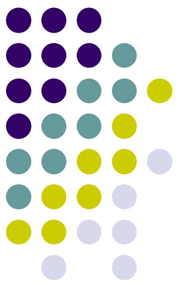
For hver node, sjekk alle
kantene fra noden.
$\Theta(V + E)$

Del 2



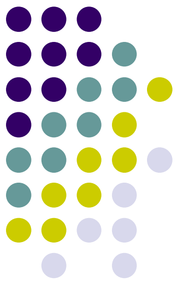
- Minimale spenntrær
- Korteste vei
- Designmetoder
- P, NP, NPC
- Flytnettverk

Minimale spenntrær



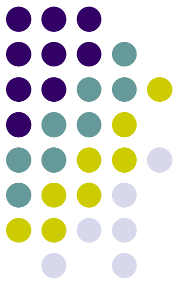
- Spenntre:
  - Subsett av kanter slik at alle noder kan nå hverandre
  - Trenger minst n – 1 kanter
- Minimalt spenntre:
  - Spenntre der kantsummen er minimal
- Algoritmer:
  - Prims
  - Kruskals

Minimale spenntrær



- Prims algoritme:
  - Velg en vilkårlig startnode (rotnode).
  - Opprett en liste over naboer.
  - Så lenge det finnes ubesøkte naboer:
    - Velg den billigste og fjern denne fra lista.
    - Legg til nodens hittil ubesøkte naboer.

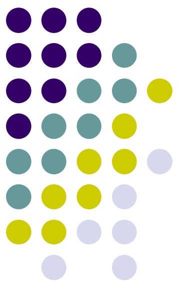
Minimale spenntrær



Kruskals algoritme:

```
Så lenge du ikke har et spenntre:
    vurder den billigste kanten
    om den ikke danner en sykel:
        legg den til
```

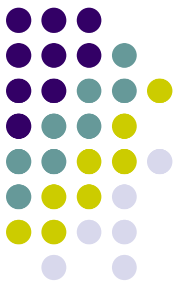
Minimale spenntrær



Hver kant som vi velger setter sammen to trær, dette gjør vi N-1 ganger; Derfor blir N trær til 1.

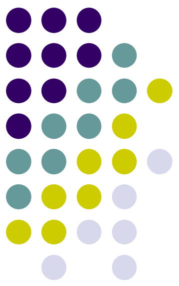
Dette gjør vi ved å velge de absolutt billigste kantene vi kan (greedy).

Korteste vei

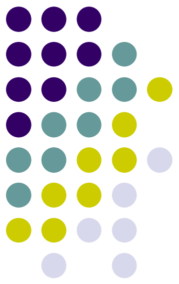


- Én-til-alle
  - DAG-Shortest-Path
  - Dijkstra (grådig/DP)
  - Bellman-Ford
- Alle-til-alle
  - Floyd-Warshall (DP)

DAG-Shortest-Path

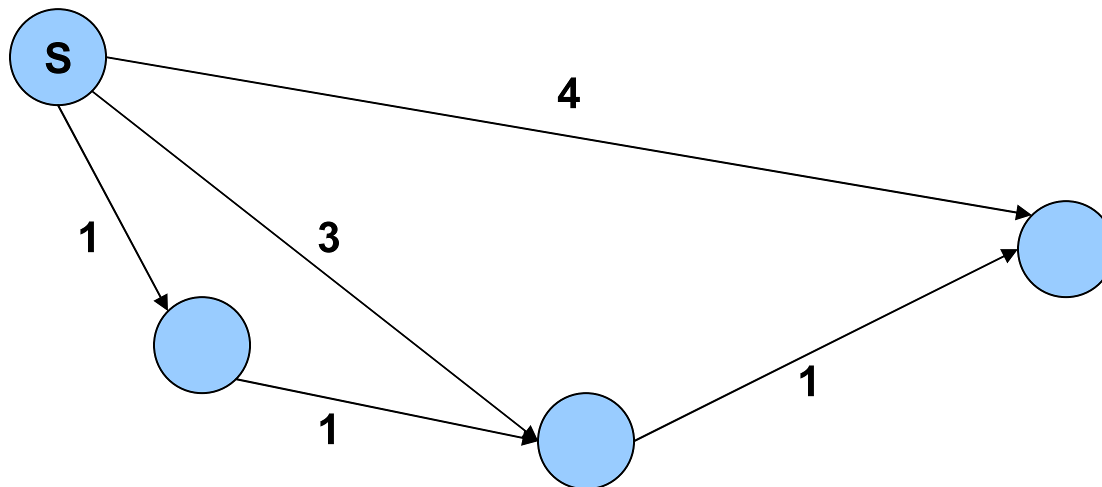


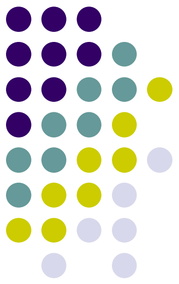
- Om vi har en DAG kan vi finne en topologisk sortering
- Vi besøker nodene i denne rekkefølgen:
  - Når vi velger en node er alle noder som kan nå den ferdig
  - Dvs den har allerede fått sin korteste vei



DAG-shortest-paths

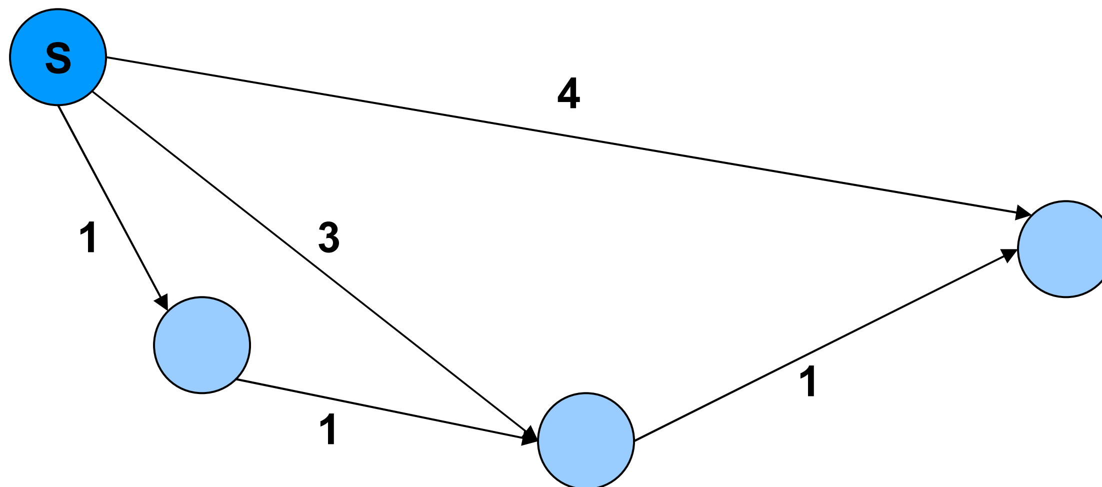
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

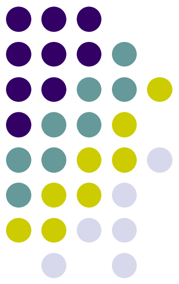




DAG-shortest-paths

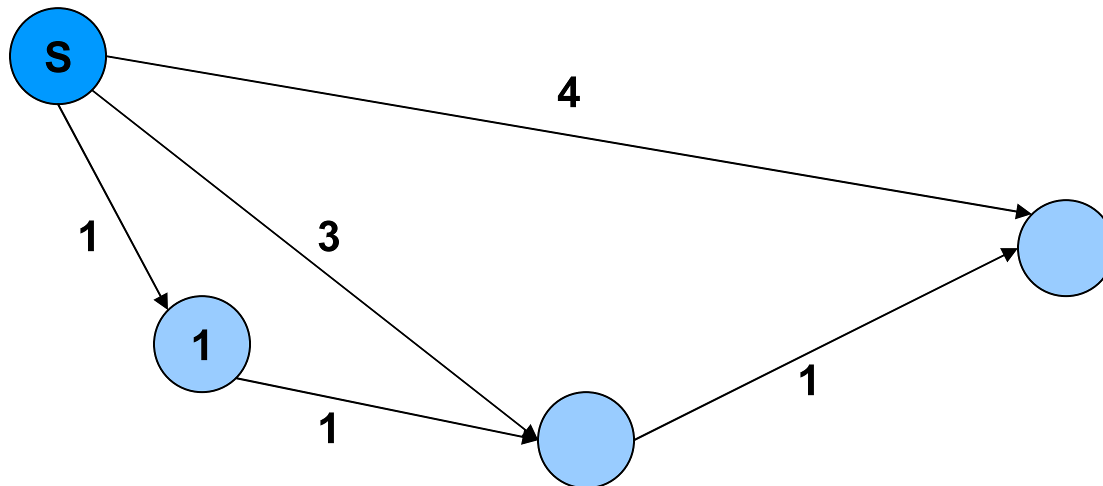
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

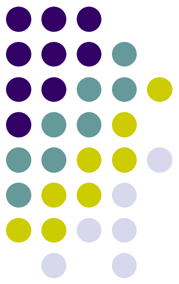




DAG-shortest-paths

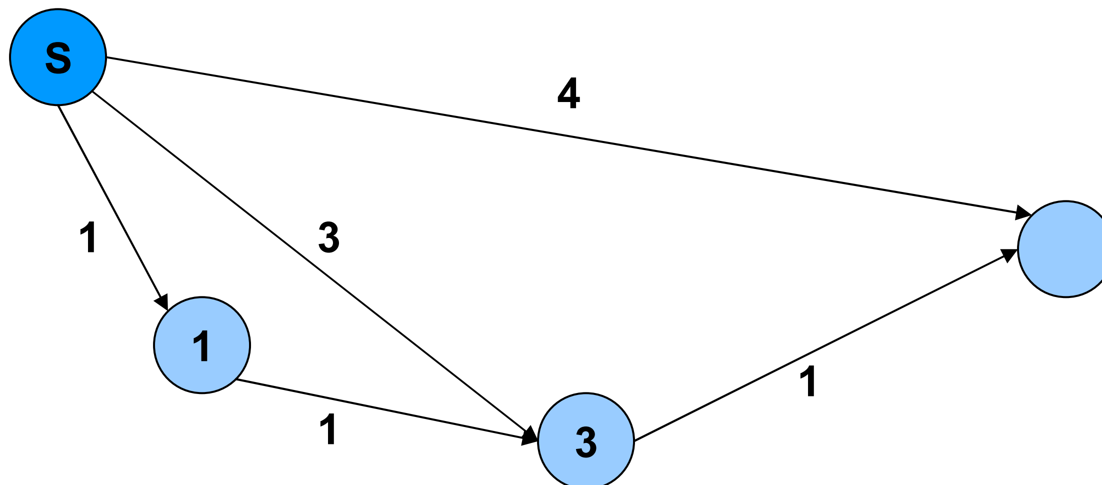
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

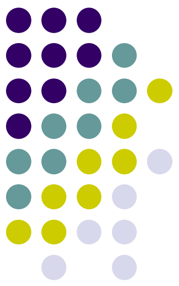




DAG-shortest-paths

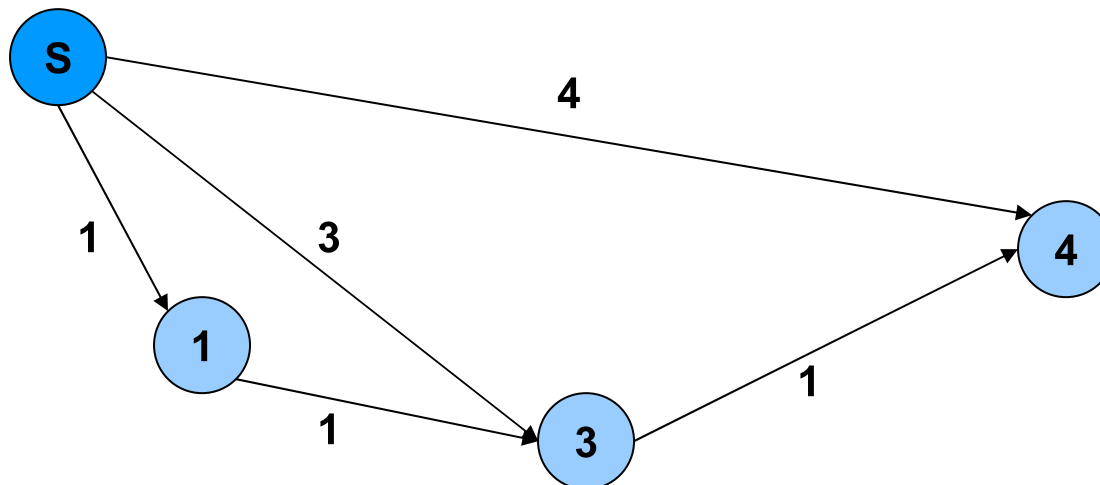
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

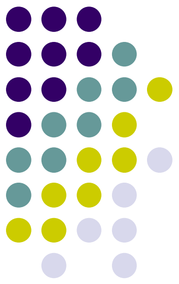




DAG-shortest-paths

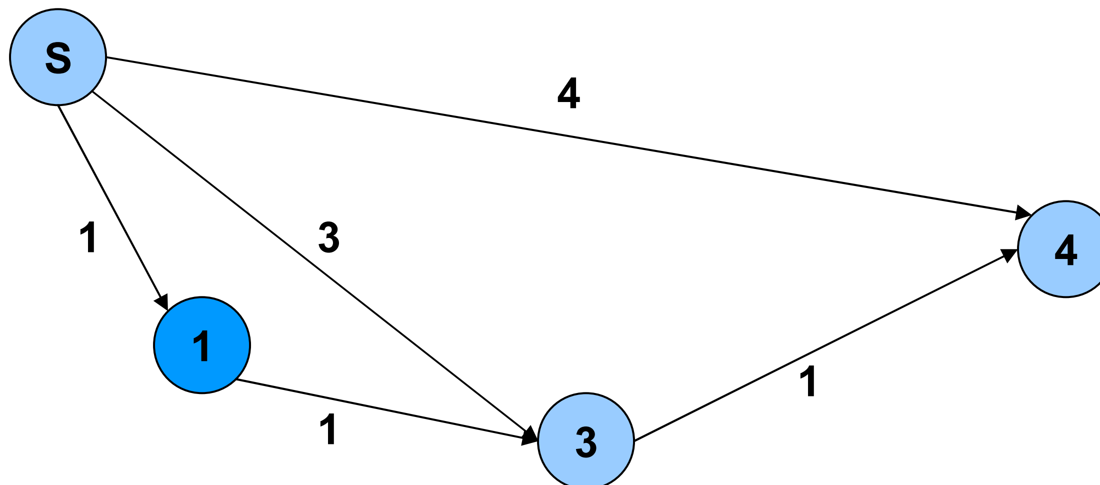
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

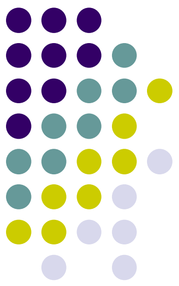




DAG-shortest-paths

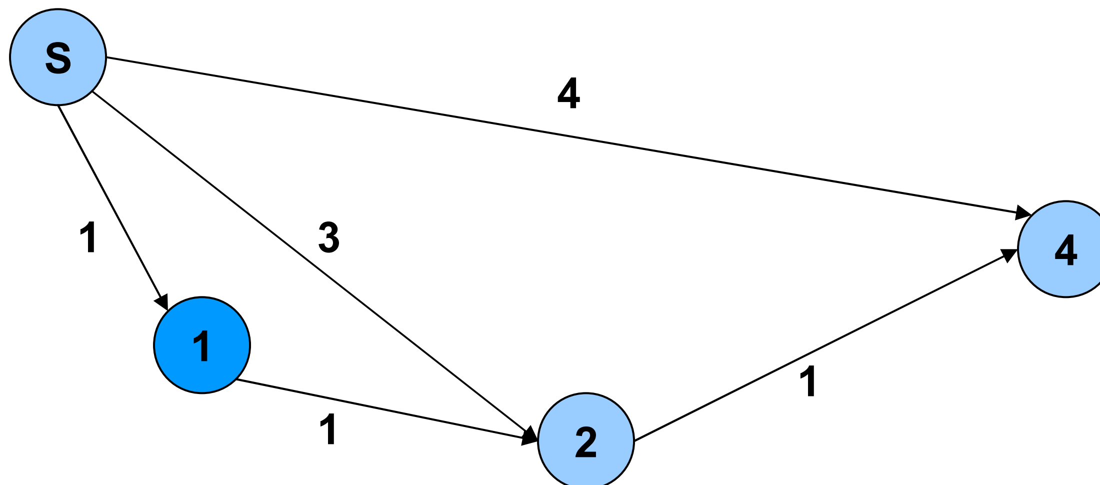
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

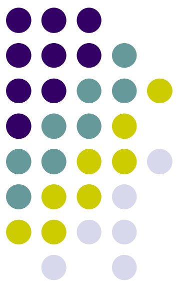




DAG-shortest-paths

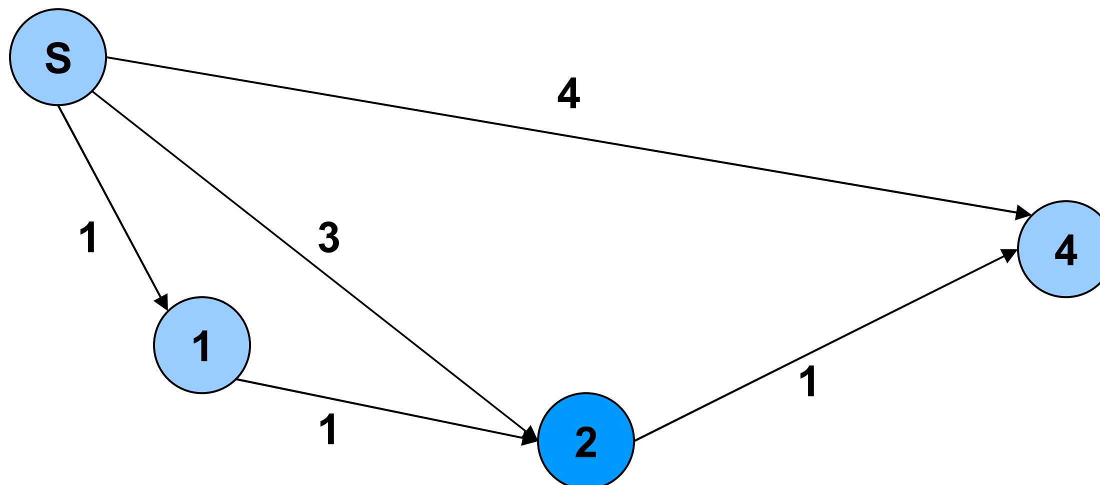
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

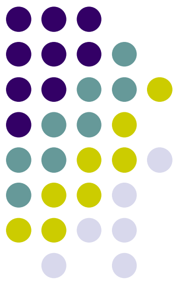




DAG-shortest-paths

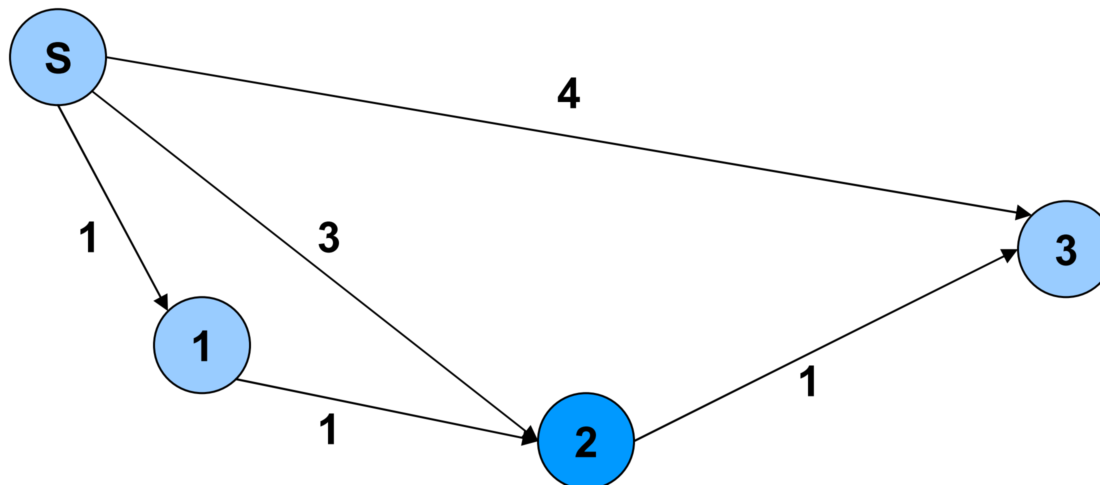
- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:



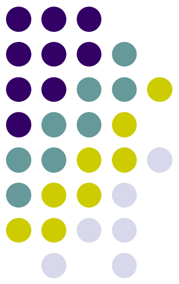


DAG-shortest-paths

- Topologisk sortering:
  - Lar oss besøke i en slik rekkefølge at estimatene ikke kan oppdateres etter at en node har blitt besøkt:

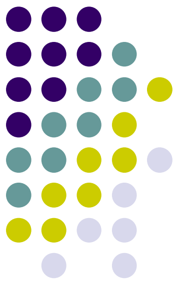


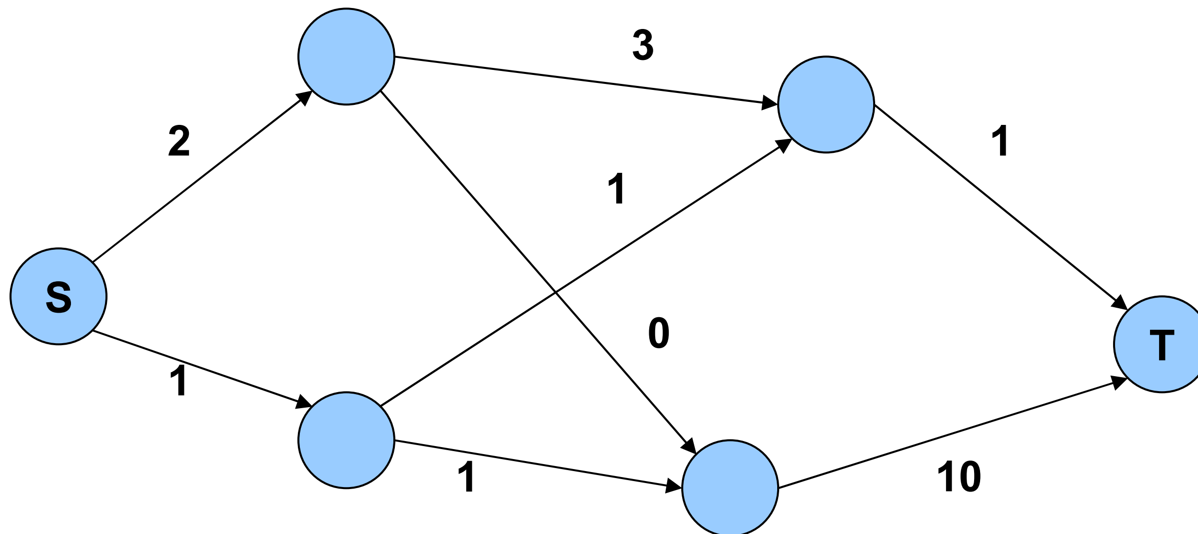
Dijkstras algoritme i en setning



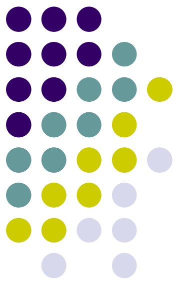
Relax kanter fra nærmeste node helt til målet velges

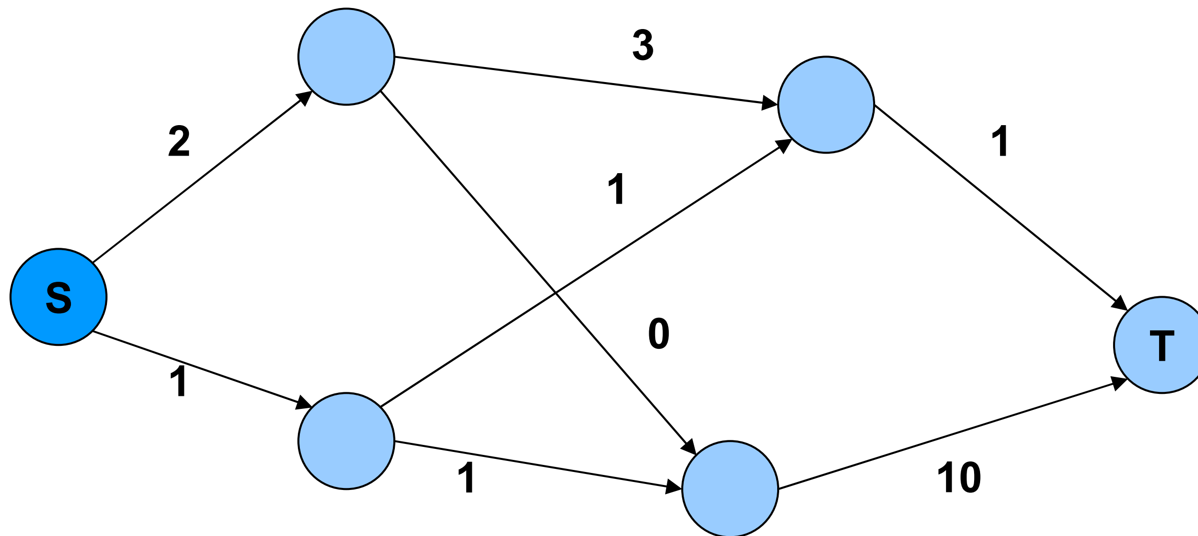
Dijkstras algoritme – eksempel



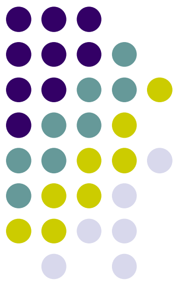


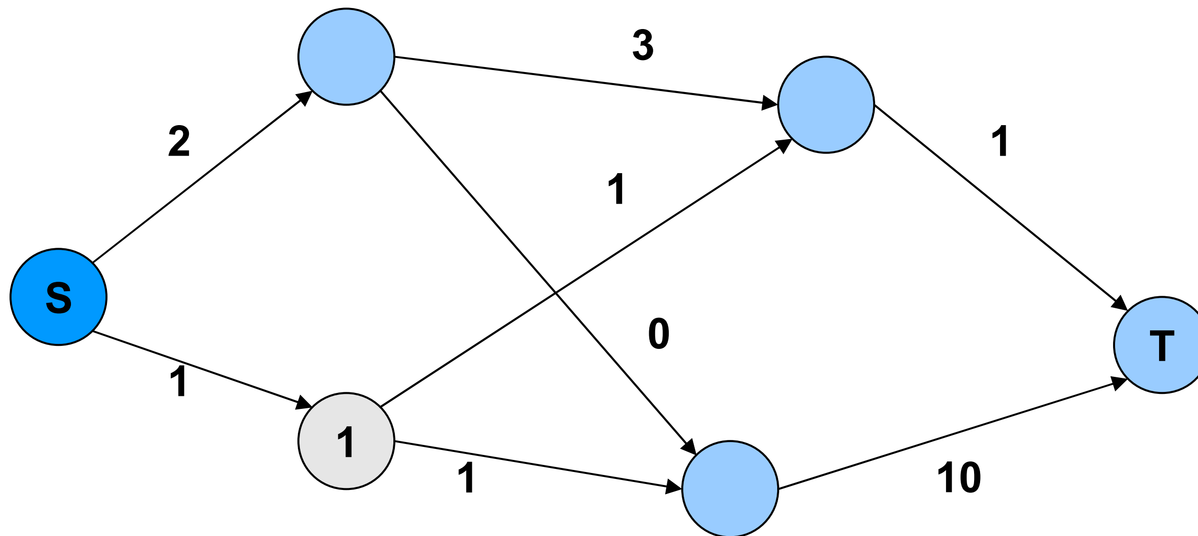
Dijkstras algoritme – eksempel



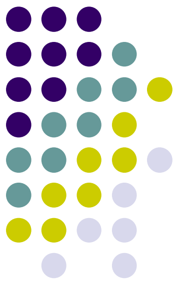


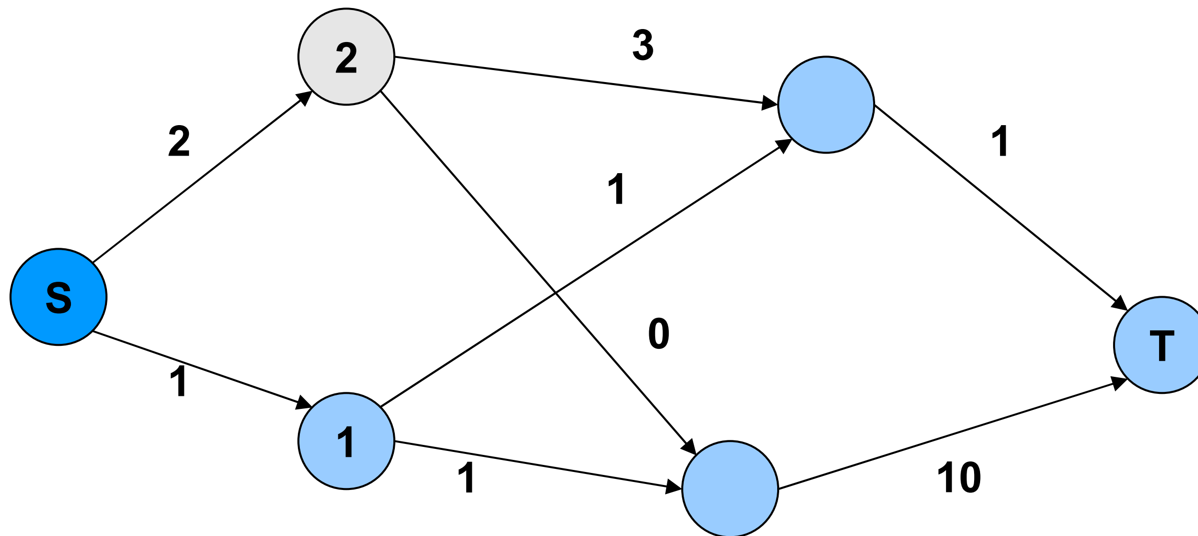
Dijkstras algoritme – eksempel



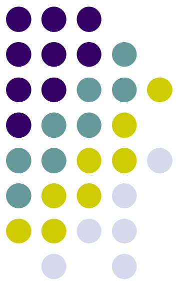


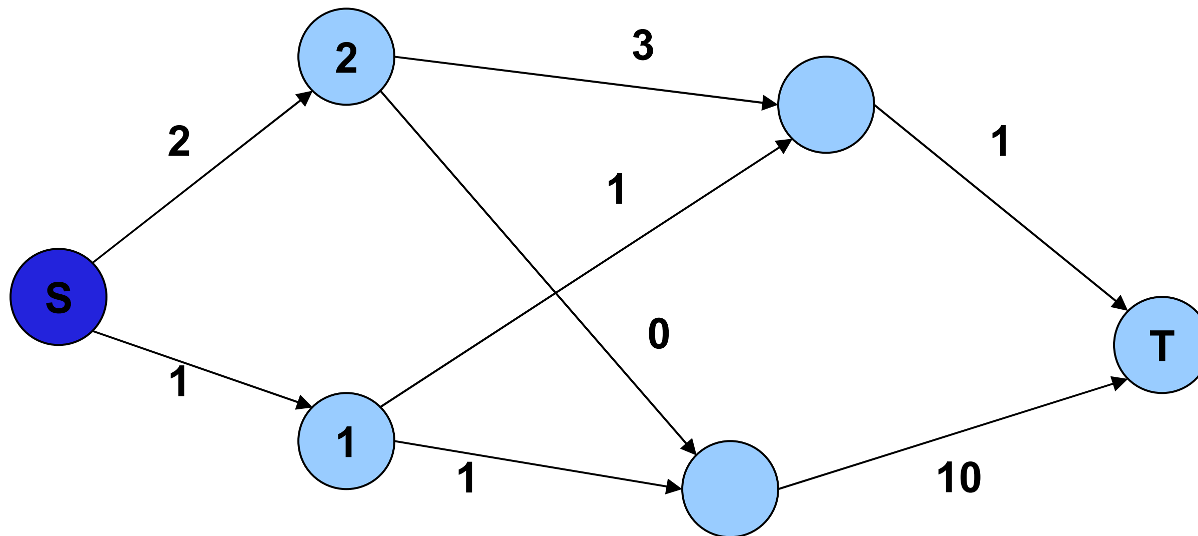
Dijkstras algoritme – eksempel



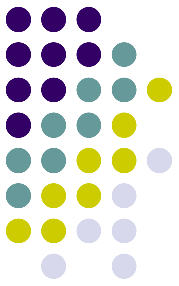


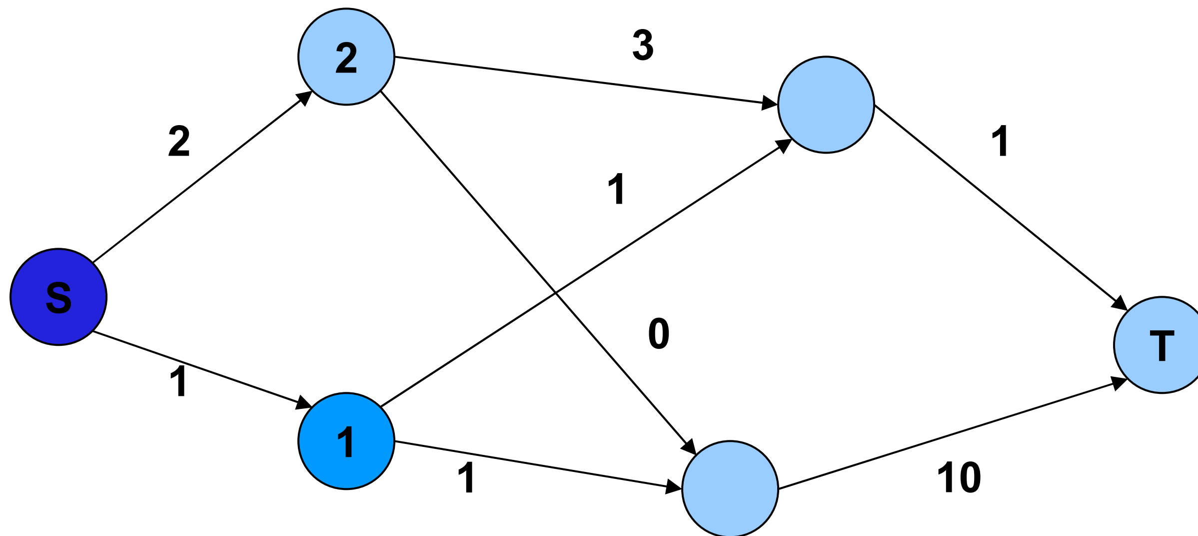
Dijkstras algoritme – eksempel



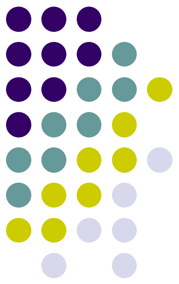


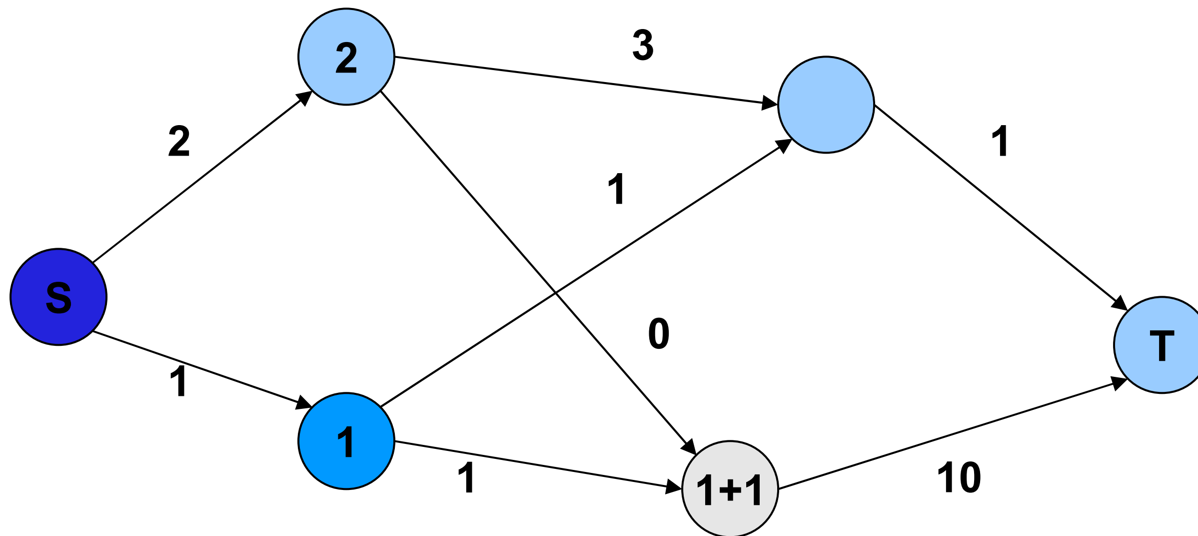
Dijkstras algoritme – eksempel



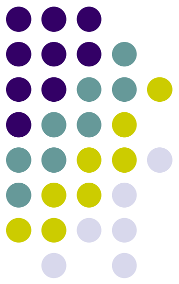


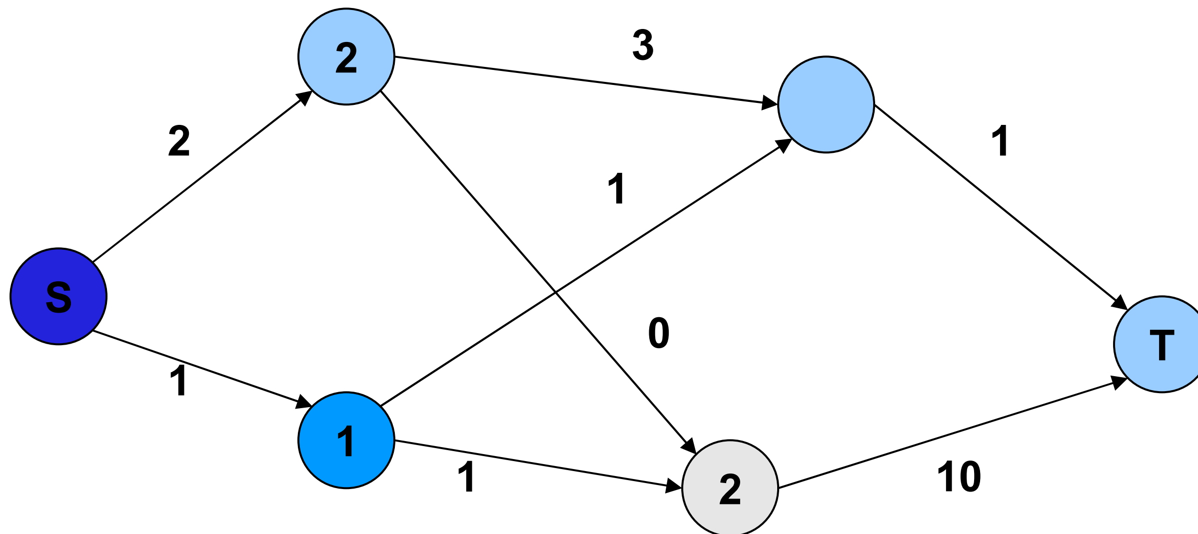
Dijkstras algoritme – eksempel



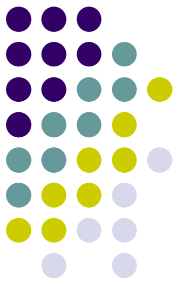


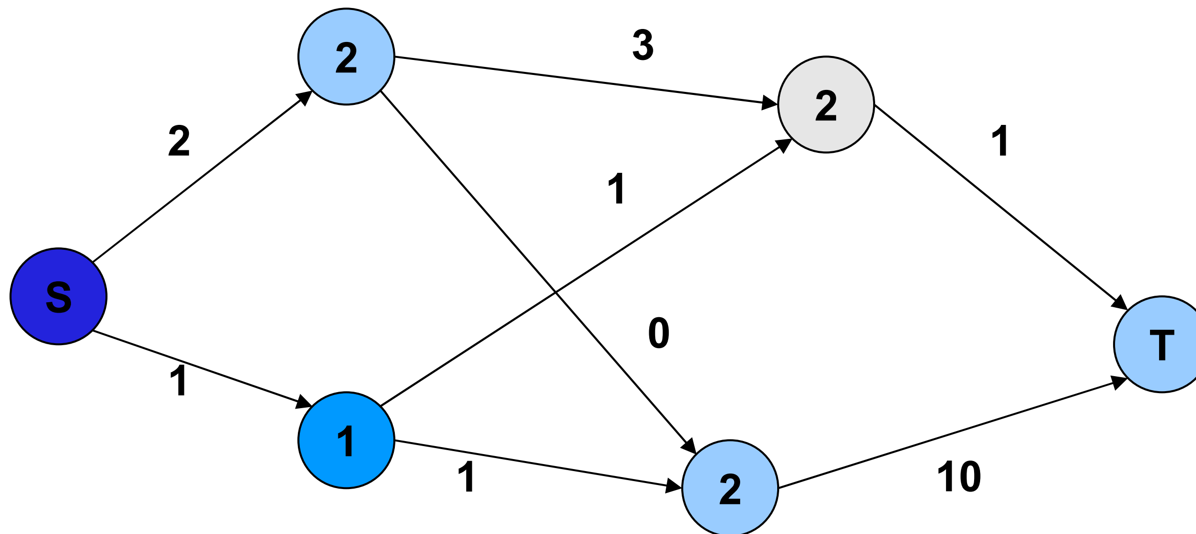
Dijkstras algoritme – eksempel



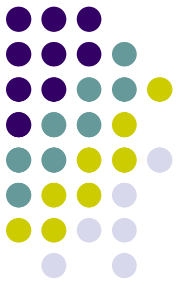


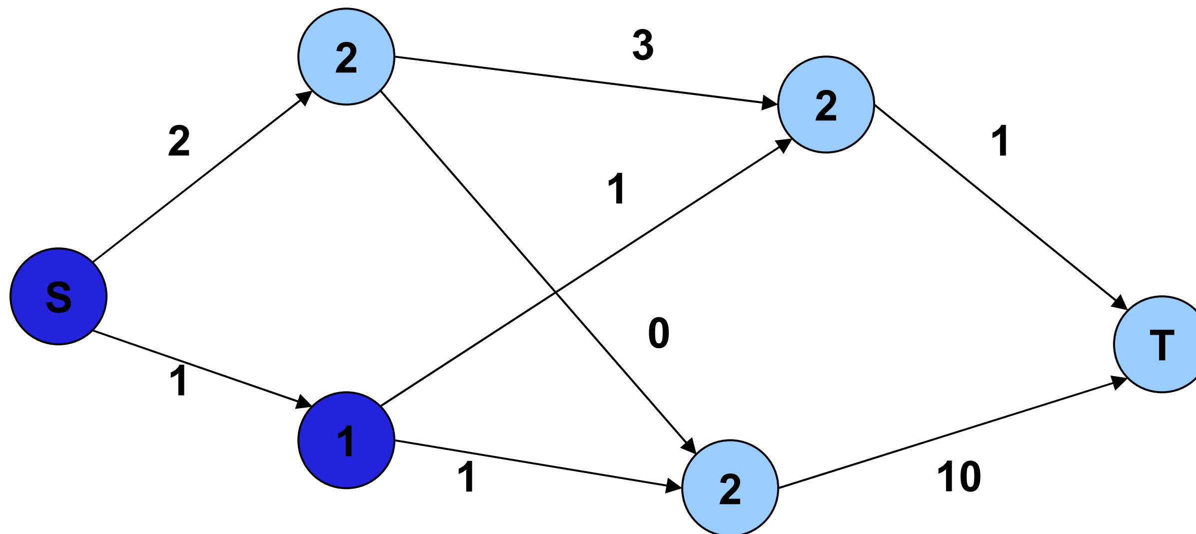
Dijkstras algoritme – eksempel



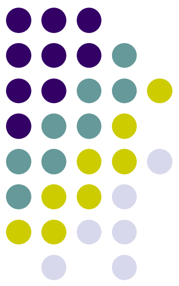


Dijkstras algoritme – eksempel





Dijkstras algoritme – valg av prioritetskø



Vi skal gjøre følgende:

1: Konstuere prioritetskø

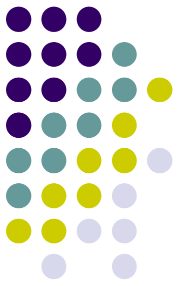
2: $|V|$ antall extract-min fra køen

3: $|E|$ anntall oppdateringer i køen (til lavere verdi)

	Array	**(Min-)Heap**
konstruer	O(n)	O(n)
extract-min	O(n)	O(log n)
decreasekey	O(1)	O(log n)

totalt	$\|V\|+\|V\|^2+\|E\|$	$\|V\|+ \|V\| log \|V\|+ \|E\| log\|V\|$
forenklet	$\|V\|^2$	$\|E\| log\|V\|$

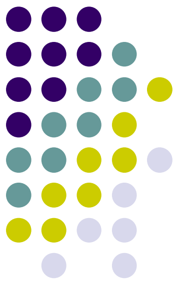
Bellman-Fords algoritme i en setning + en finesse

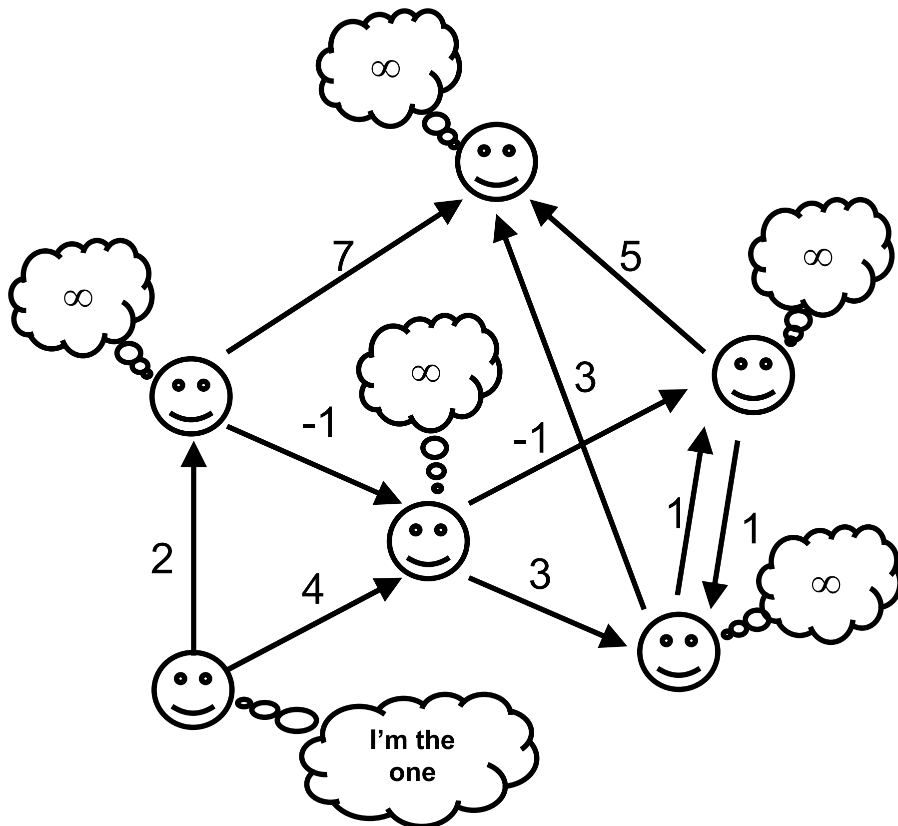


Kjør relax over alle kanter i grafen, og repeter til det er gjort n-1 ganger.

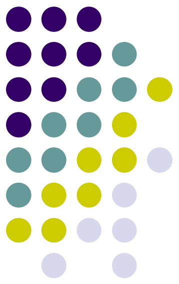
Hvis man repeterer n'te gang og relax finner en forbedring, har man negative sykler.

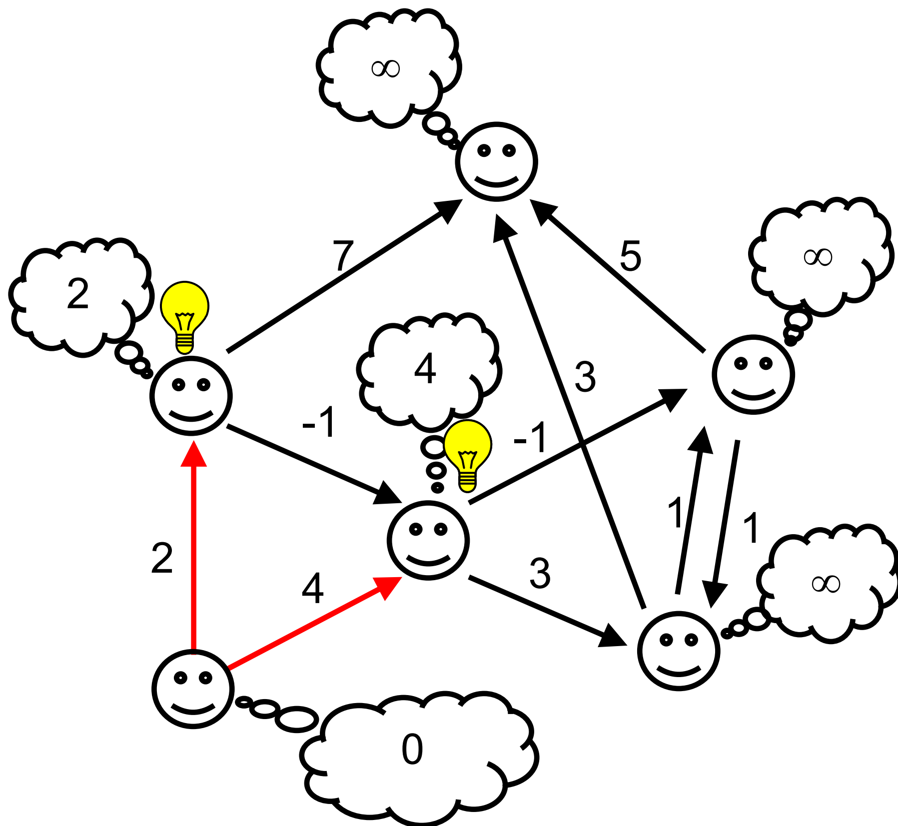
Bellman Fords algoritme



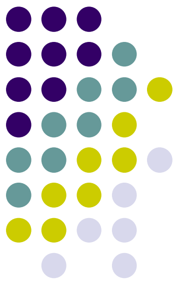


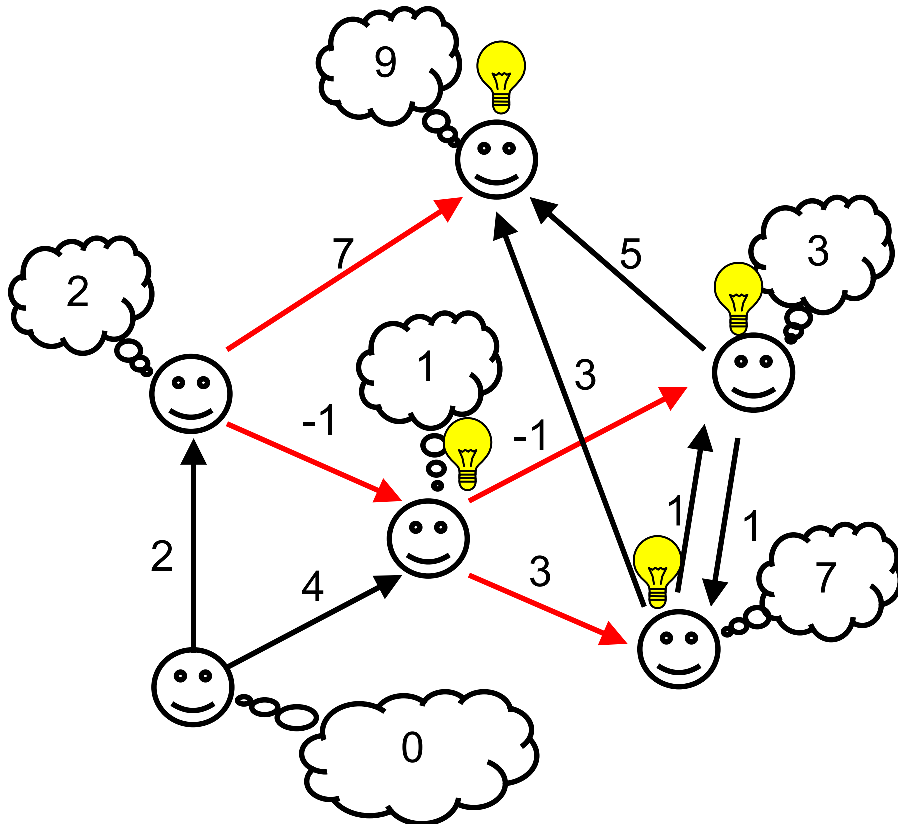
Bellman Fords algoritme



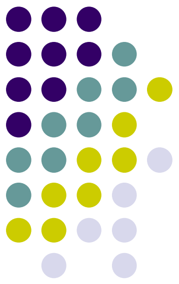


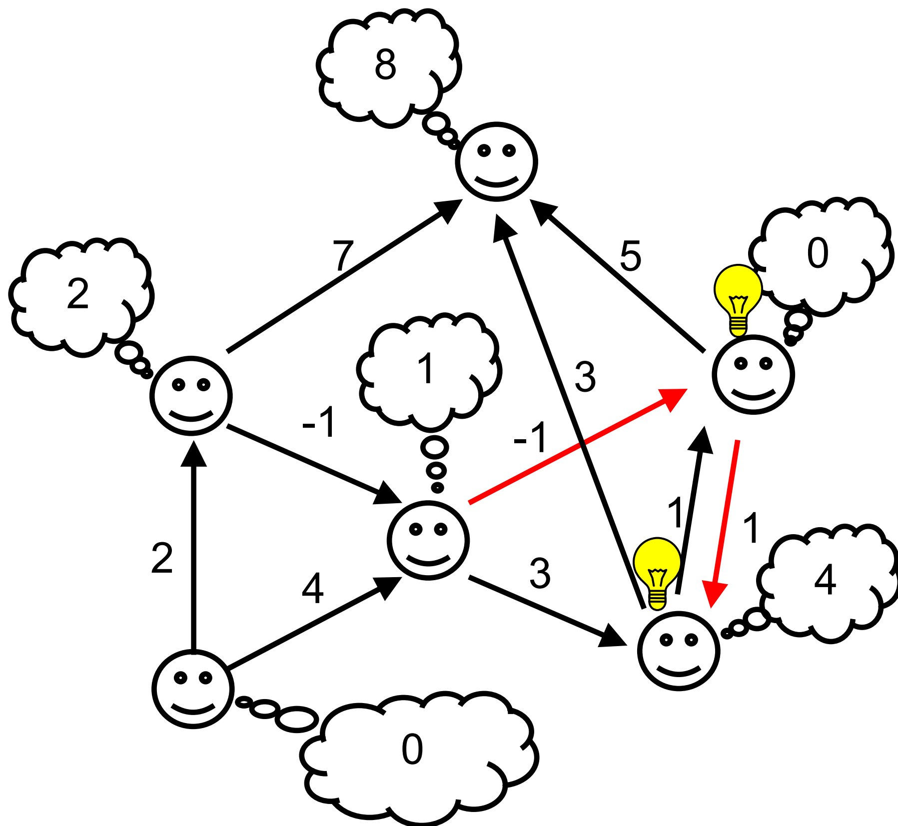
Bellman Fords algoritme



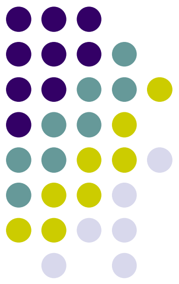


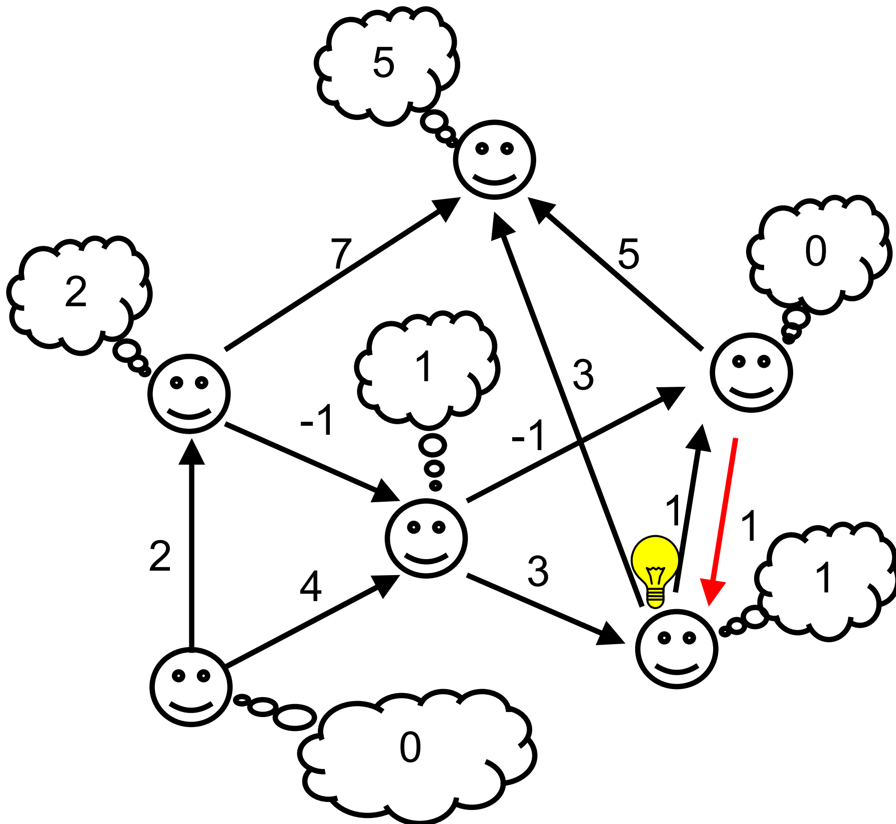
Bellman Fords algoritme



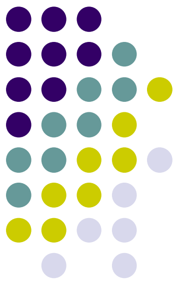


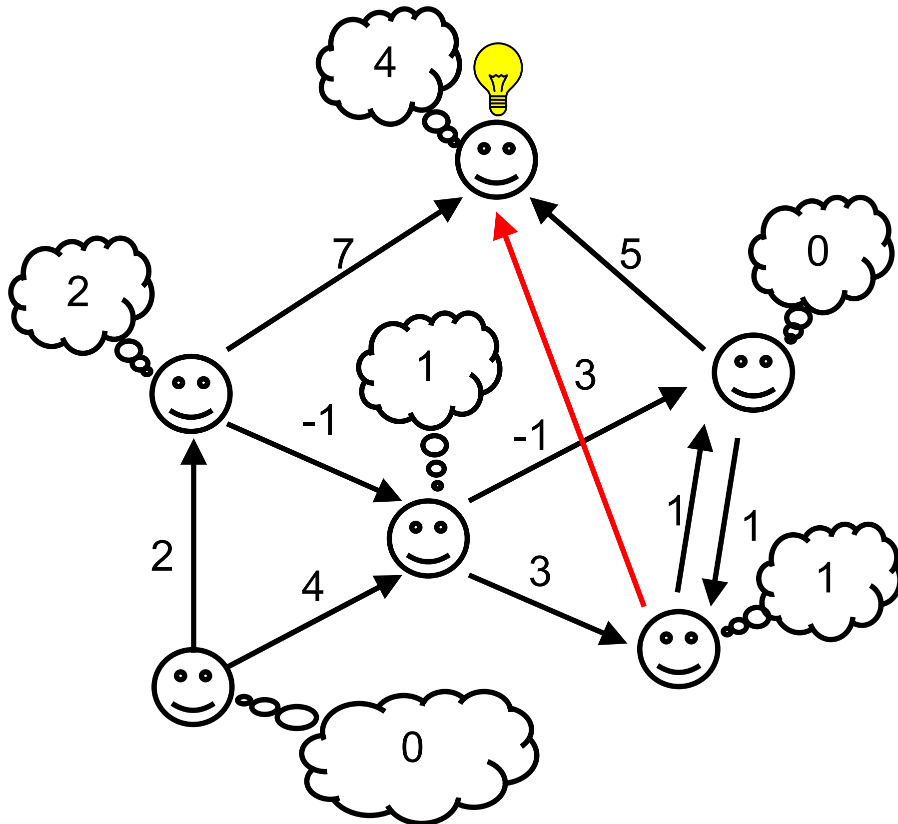
Bellman Fords algoritme



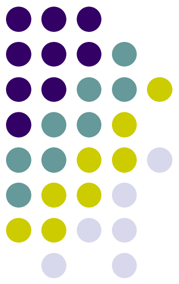


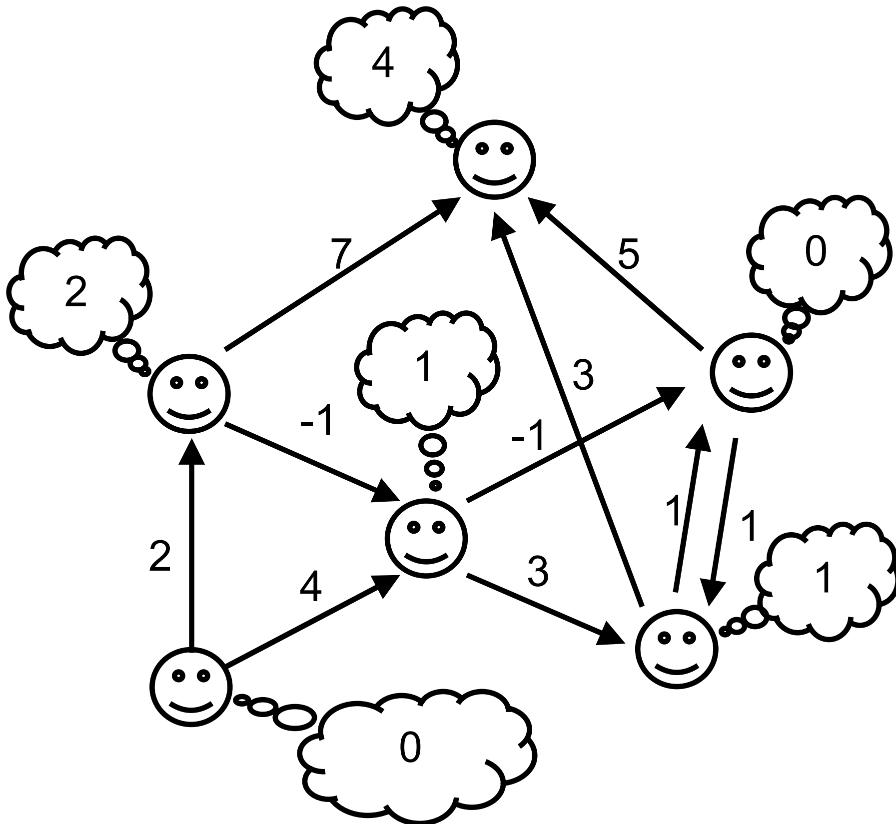
Bellman Fords algoritme

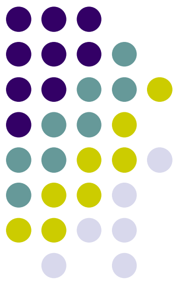




Bellman Fords algoritme



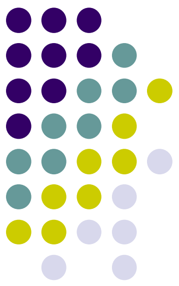




Bellman Fords algoritme

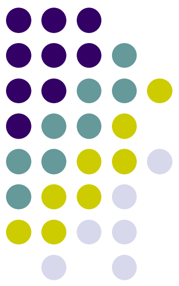
- Relax blir kjørt på alle kantene.
- Dette gjøres |V| ganger.

- Kjøretid: O(|V|*|E|)



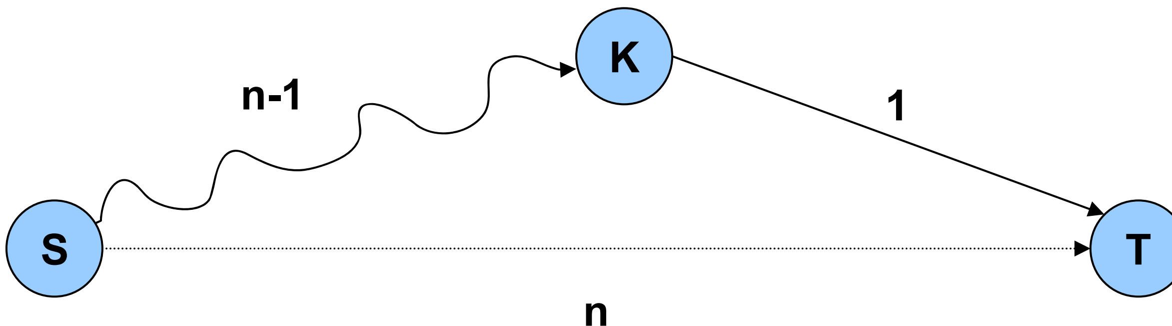
Korteste vei - alle-til-alle

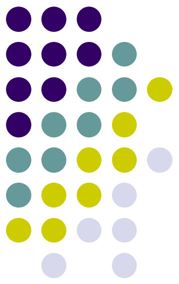
- To måter å dekomponere veier på
- Det er forventet at disse kan forklares (godt)



Korteste vei - alle-til-alle Metode 1

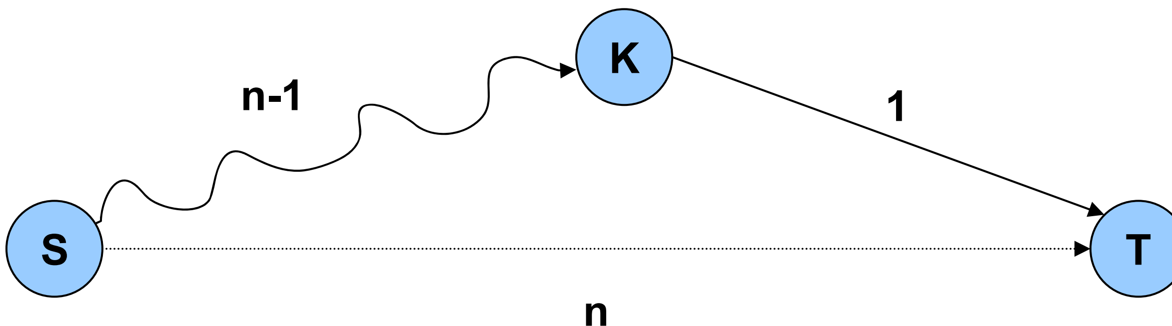
- Den korteste veien fra S til T kan bygges opp med den korteste fra S til K, og kanten mellom K og T
- K representerer her den siste noden før T



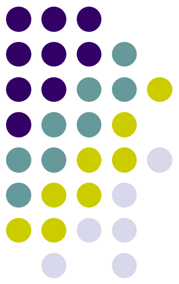


Korteste vei - alle-til-alle Metode 1

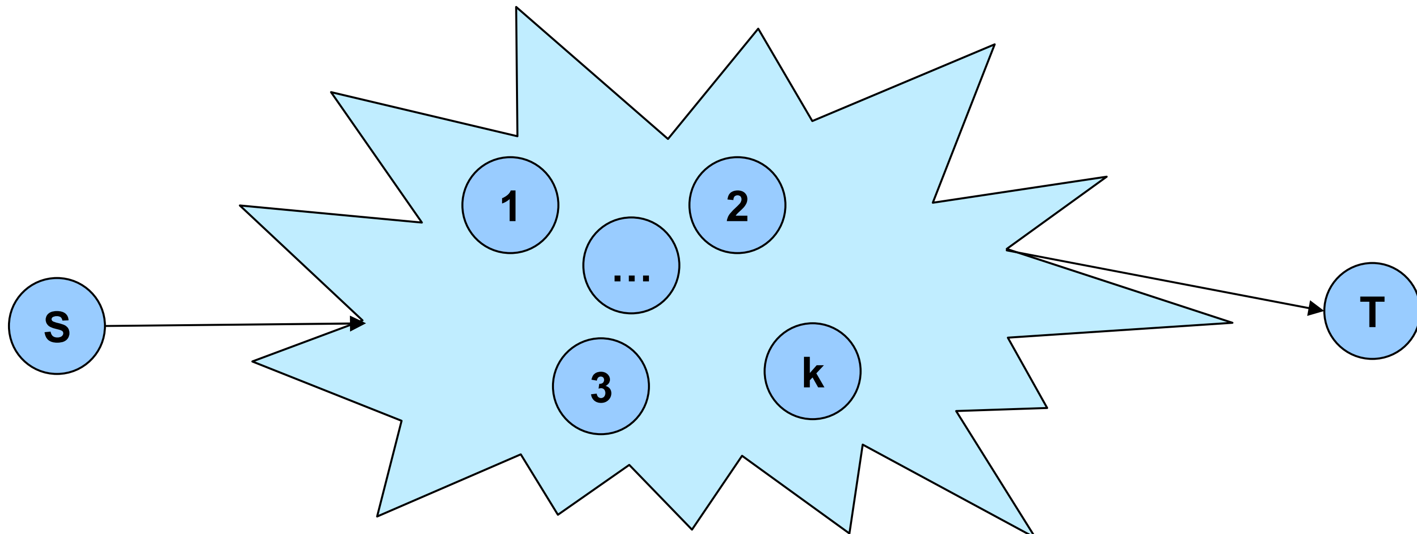
- Gir opphav til en $O(V^4)$ algoritme for korteste vei alle til alle

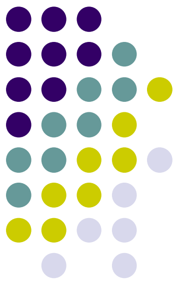


Korteste vei - alle-til-alle Metode 2



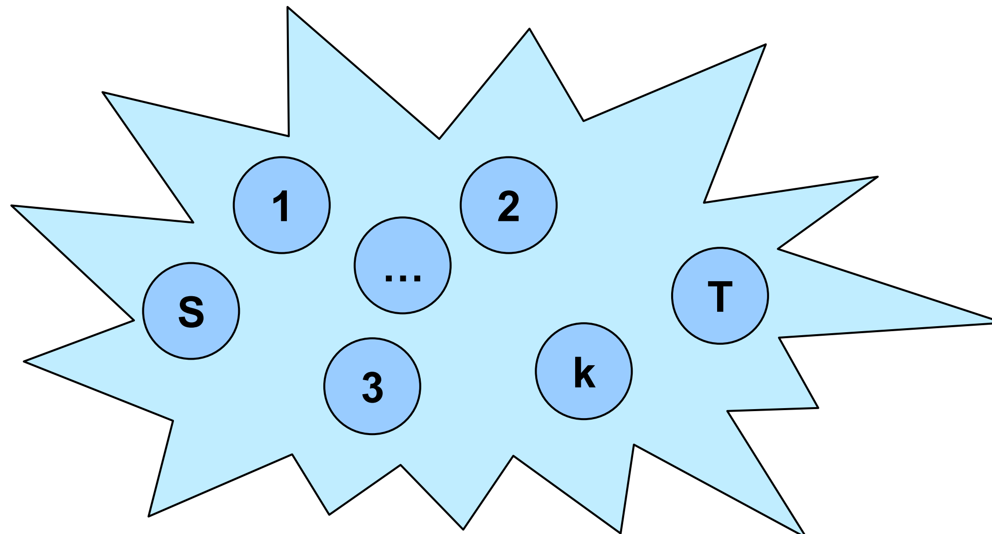
- La oss for øyeblikket kun se på korteste veier som bruker node (1, 2, 3, …, k) foruten endenodene:



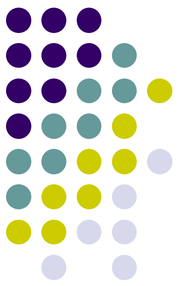


Korteste vei - alle-til-alle Metode 2

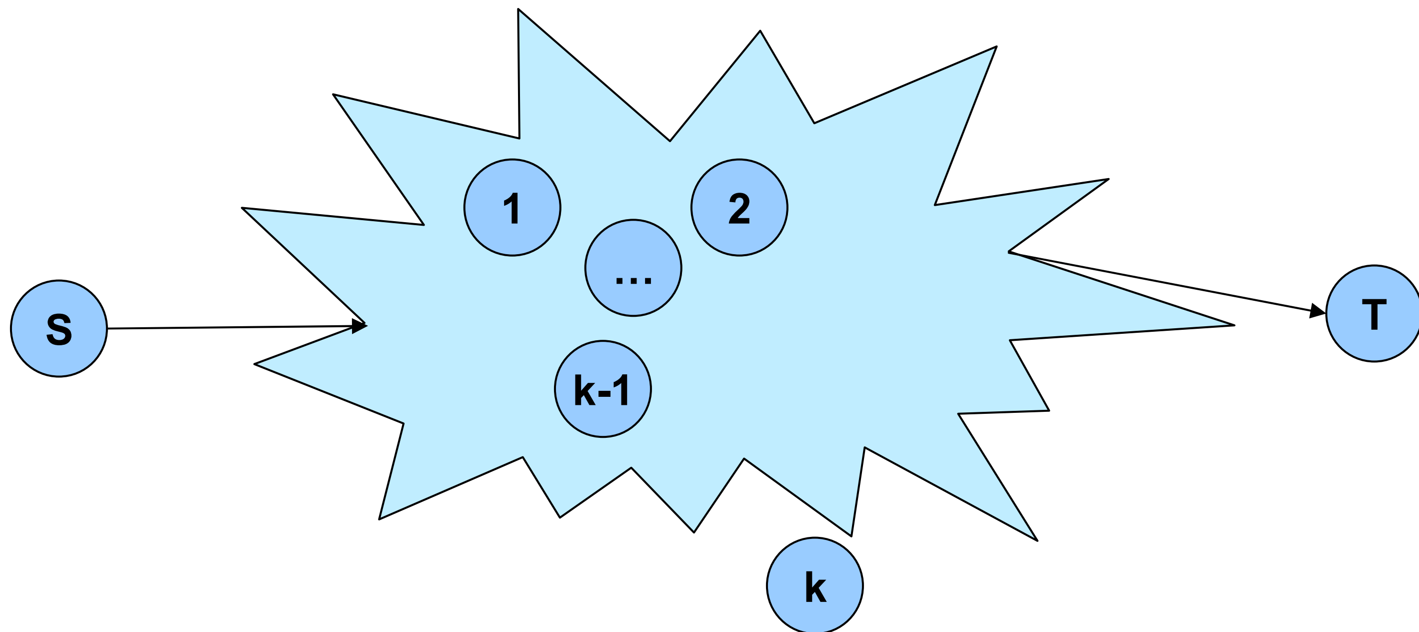
- La oss for øyeblikket kun se på korteste veier som bruker node (1, 2, 3, …, k) foruten endenodene:

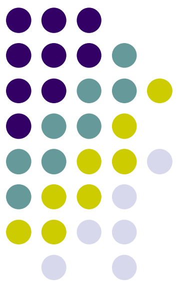


Korteste vei - alle-til-alle Metode 2



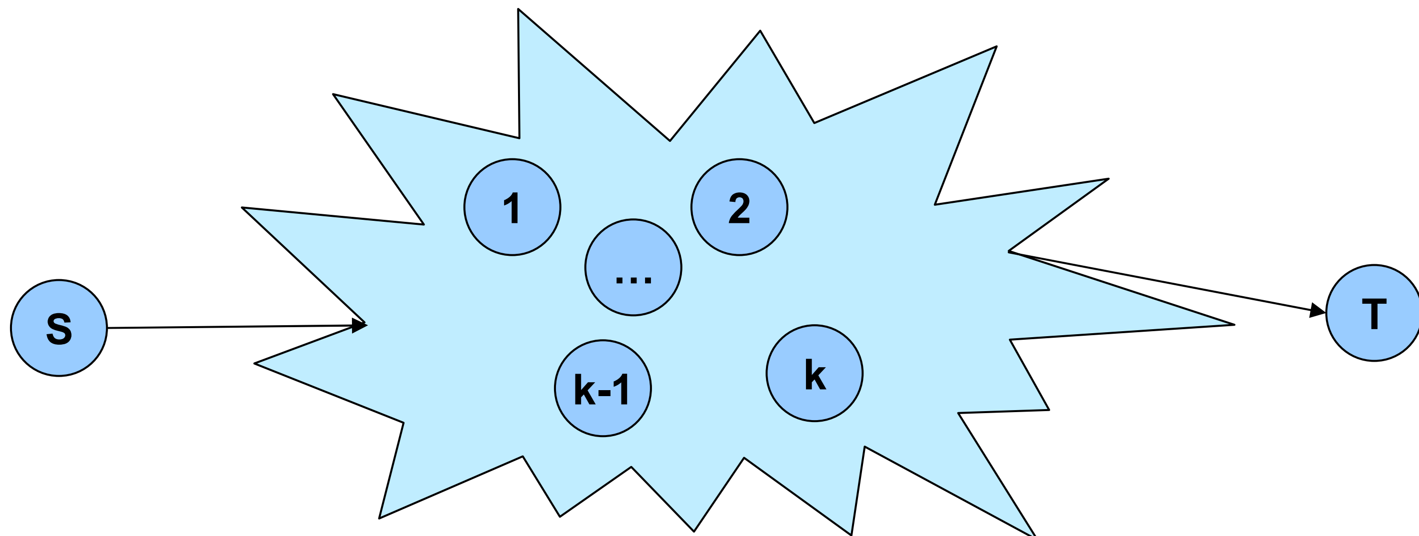
- Hver korteste vei vil da enten bestå av noder til-og-med k-1,

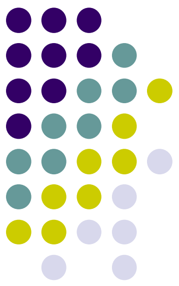




Korteste vei - alle-til-alle Metode 2

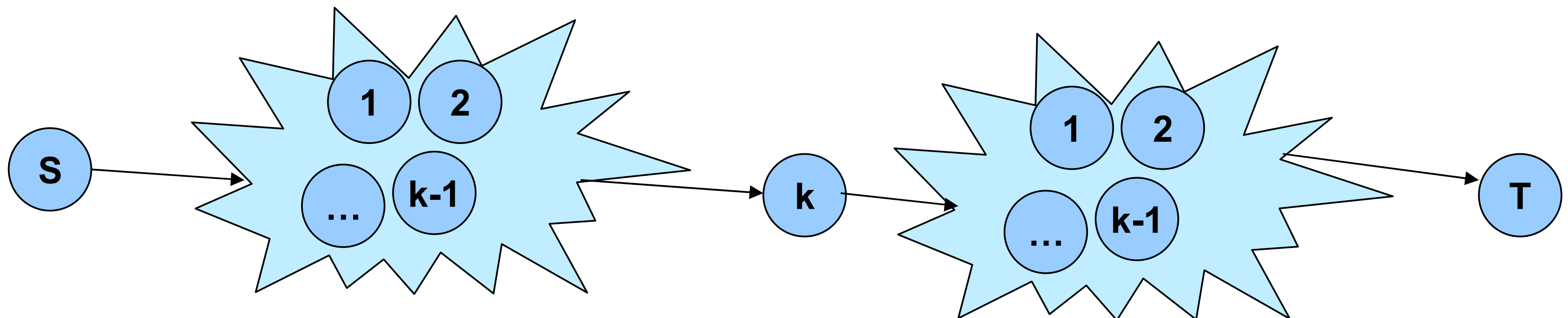
- Eller en korteste vei med k
  - Denne kan dekomponeres:



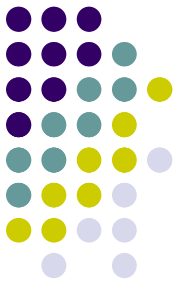


Korteste vei - alle-til-alle Metode 2

- Eller en korteste vei med k
  - Denne kan dekomponeres:

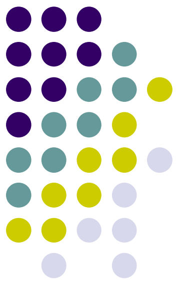


Korteste vei - alle-til-alle Metode 2

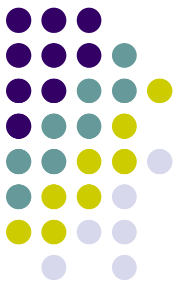


- Vi får da at korteste veier som bruker noder til og med node k, kan bygges opp av korteste veier som bruker til og med node k-1
- Gir opphav til Floyd-Warshall som har kjøretid $O(V^3)$

Designmetoder



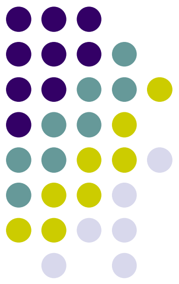
- Generelle måter å designe algoritmer på
  - Splitt og hersk
  - Grådighet
  - Dynamisk programmering
  - Lineær programmering



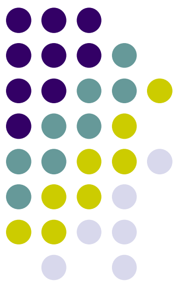
Splitt og hersk

- Noen problemer kan rekursivt deles opp i flere uavhengige delproblemer
- Dette kalles splitt og hersk:
  - Splitt problemet opp i delproblemer
  - ”Hersk” delproblemene rekursivt hver for seg
  - Kombiner resultatet
- Eksempel: mergesort, quicksort

Grådighet



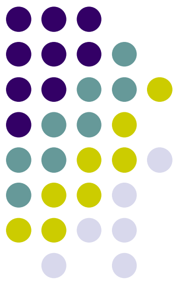
- Greedy-choice property:
  - At det ikke skader å ta det valget som er best akkurat nå.
- Noen problemer som kan løses dynamisk, kan også løses vha. grådighet.
- Eksempel: Dijkstras algoritme velger alltid neste node grådig.



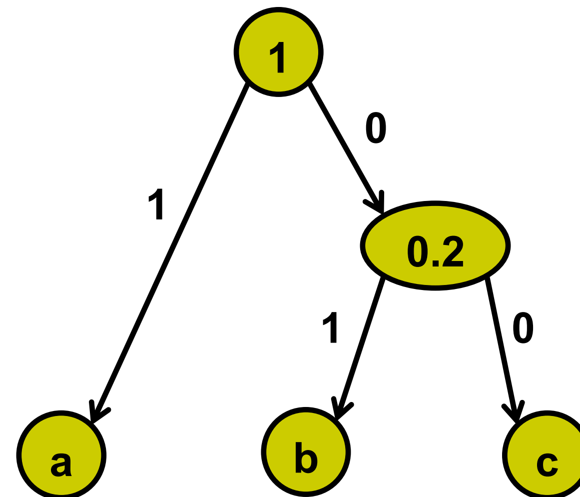
Huffman-koding
-et grådig problem

- Finn et optimalt alfabet til en gitt ”tekst”.
- Vi har noen tegn som vi skal lage et alfabet til, og vi vet frekvensen til tegnene.
- Legg alle tegn i en liste
- Så lenge det er flere enn et element:
  - Slå sammen de to billigste trærne, og lag en ny node med frekvens lik summen som legges tilbake.

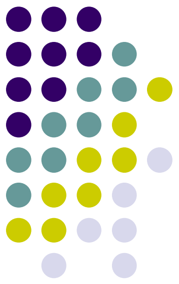
Huffman-koding
-et grådig problem



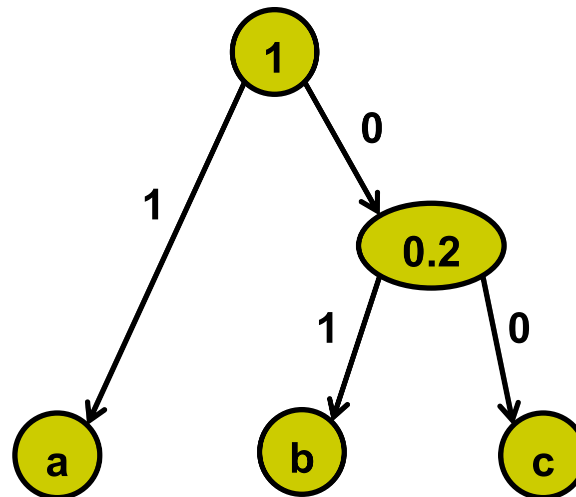
Tegn	Frekvens
a	0.8
b	0.1
c	0.1



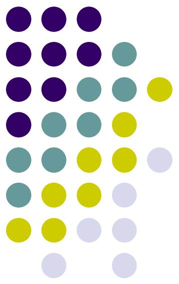
Huffman-koding
-et grådig problem



Komprimer ”abaac”:
1011100

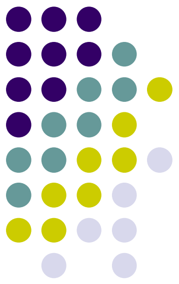


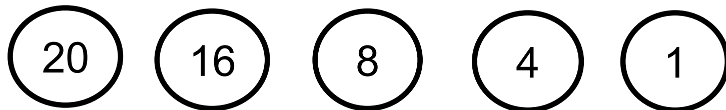
Dynamisk programmering



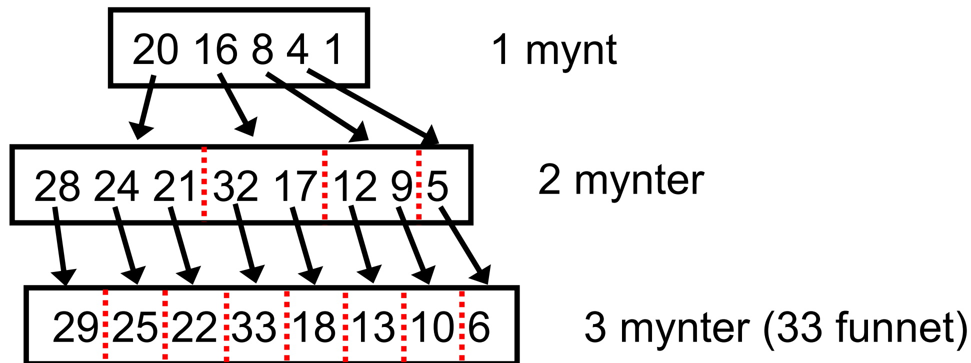
- ”Bygg løsningen fra bunnen av”
- Vi skal ta 3 eksempler

Dynamisk programmering - eksempel 1, pengeveksling

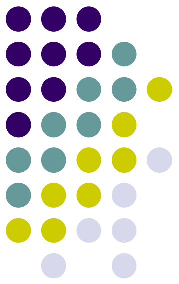




Vi bygger steg for steg summer som kan inneholde maks *i* mynter og øker *i* til vi har det rette svaret. Vi prøver 33.



Dynamisk programmering - eksempel 2, fibonaccitallene



Fibonaccitallene er definert som følger:
Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2), n > 1

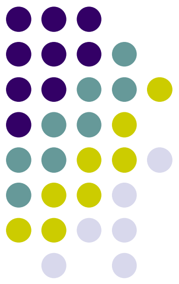
Her kan man programmere definisjonen direkte, men da vil man få en kjøretid som vokser som 2^n.

```
def fibo(n):
   if n < 2:
      return n
   else:
      return fibo(n - 1) + fibo(n - 2)
```

Hvis man bygger fra bunnen av og holder styr på 2 tall hele veien, får man lineær kjøretid.

```
def fibonacci(n):
   a, b = 0, 1
   for i in xrange(n-1):
      a, b = b, a + b
   return b
```

Memoisering - fibonaccitallene



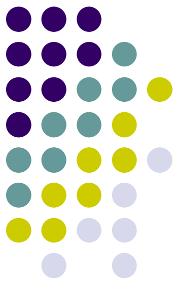
Med noen små triks kan vi bruke den intuitive implementasjonen av fibo() og likevel få lineær kjøretid.

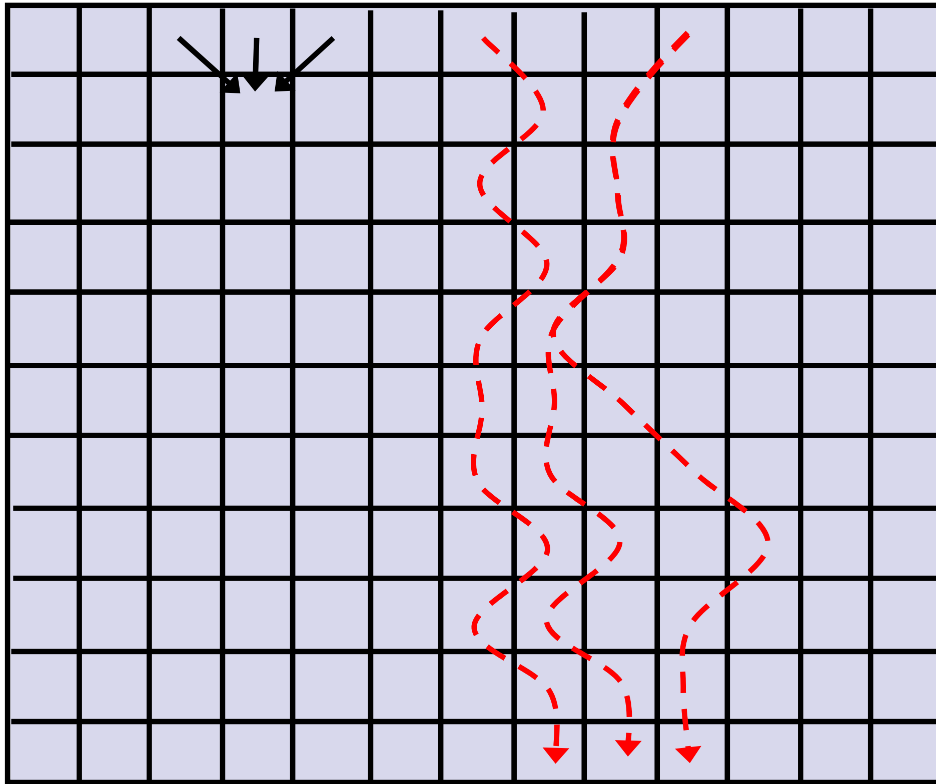
```
def fibo(n):
  if n < 2:
    return n
  else:
    return fibo(n - 1) + fibo(n - 2)
```

Vi tar vare på alle svar som blir regnet ut, og sparer oss for jobben å gjøre akkurat samme utregning to ganger.

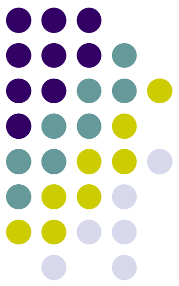
```
global  done
done = {}
def fiboM(n):
  if n in done:
    return done[n]
  if n < 2:
    result = n
  else:
    result = fiboM(n-1) + fiboM(n-2)
  done[n] = result
  return result
```

Dynamisk programmering - eksempel 3, korteste vei over bilde



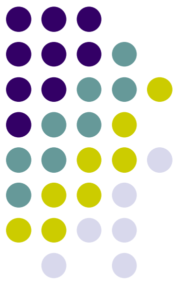


Lineært program



- Et lineært program er en matematisk struktur som består av:
  - Et uttrykk som enten skal maksimeres eller minimeres.
  - Et sett med ulikheter eller likheter som skal overholdes. Disse er lineære funksjoner.

Et lineært program



- Enkelt eksempel med 2 variabler:

  Maksimer $x_1 + x_2$

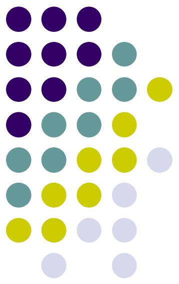
  med beskrankninger

  $4x_1 - x_2 \leq 8$

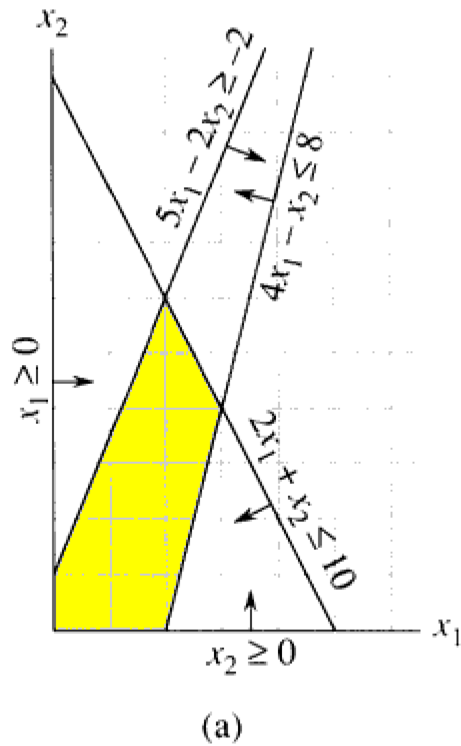
  $2x_1 + x_2 \leq 10$

  $5x_1 - 2x_2 \geq -2$

  $x_1, x_2 \geq 0$

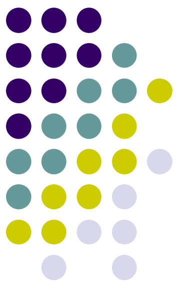


2 dimensjoner i koordinatsystem



(a)

Kjente problemer kan uttrykkes som LP



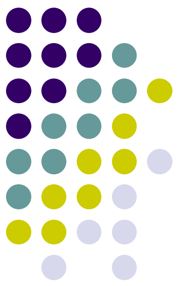
- F.eks: Korteste vei:
- G = (V, E), med kantvekter $w(u,v)\ \forall (u,v) \in E$
- Maksimer:

$$d[t]$$

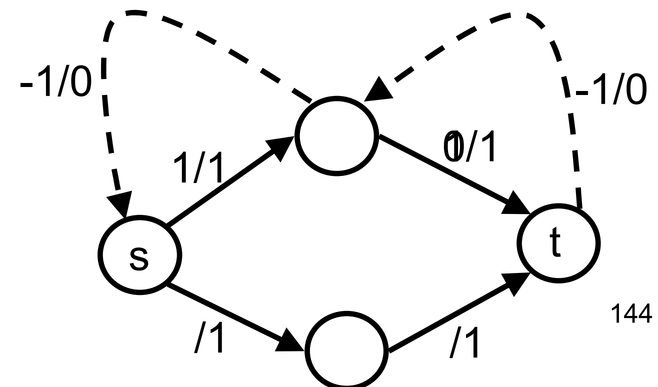
- Med beskrankninger:

$$d[v] \leq d[u] + w(u,v)\ \forall (u,v) \in E$$
$$d[s] = 0$$

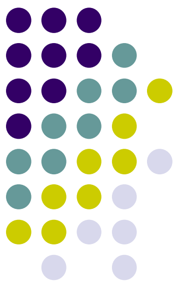
Flyt – hva er flyt?



- En strøm som går fra node til node gjennom kanter.
- For hver node er flyten inn lik flyten ut. Ingen flyt kan oppbevares i en node.
- Kanter har ofte en maksgrense for mengde flyt som kan gå igjennom.
- Skew-symmetry

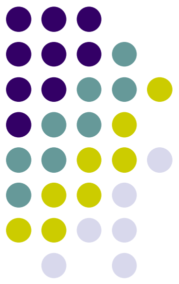


Max-flow min-cut teoremet



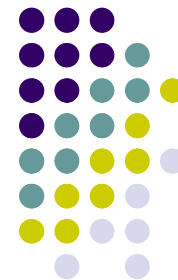
- Anta flytnettverk G = (V,E) med kilde *s* og sluk *t*. Da er følgende utsagn ekvivalente:
  - f er maksimal flyt i G
  - Residualnettverket G_f har ingen flytforøkende sti
  - | f | = c(S,T) for et snitt (S,T) av G
    - Et slikt snitt er et min-cut av G

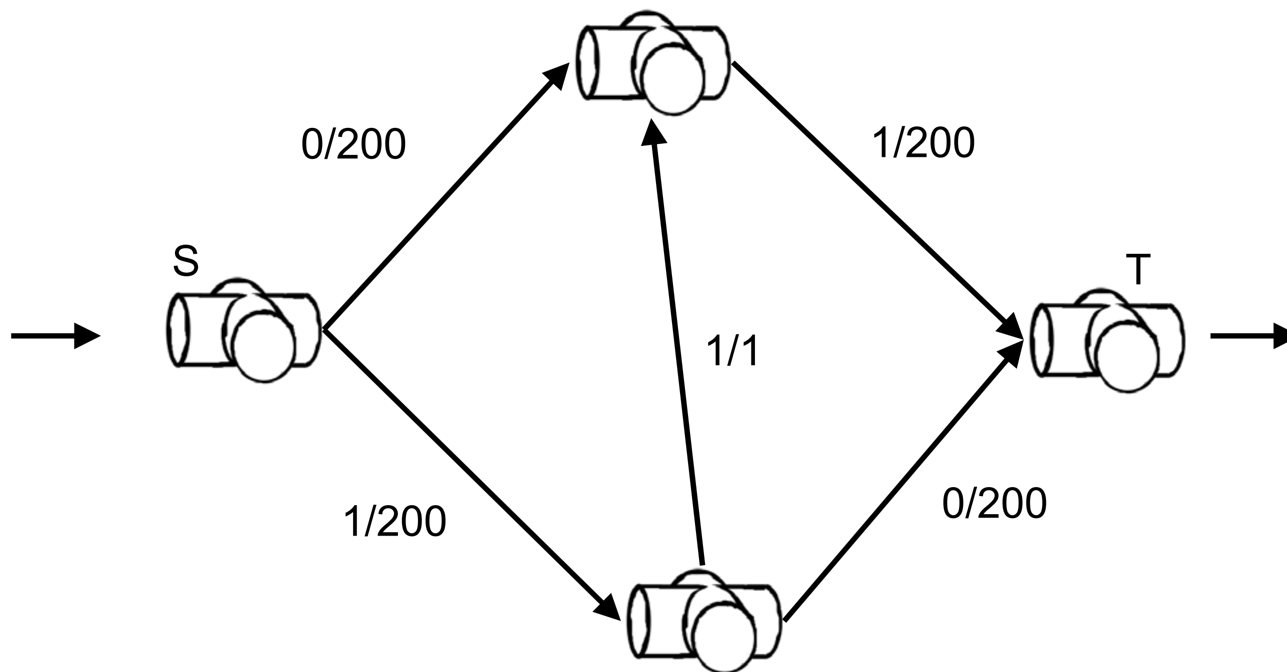
Løsning på maks-flytproblemet - Ford-fulkerson metoden



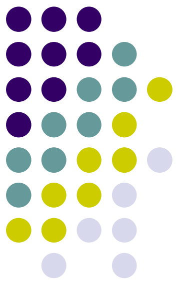
Så lenge det finnes minst en flytforøkende sti fra S til T, øk flyten fra S til T med så mye som stien tillater.

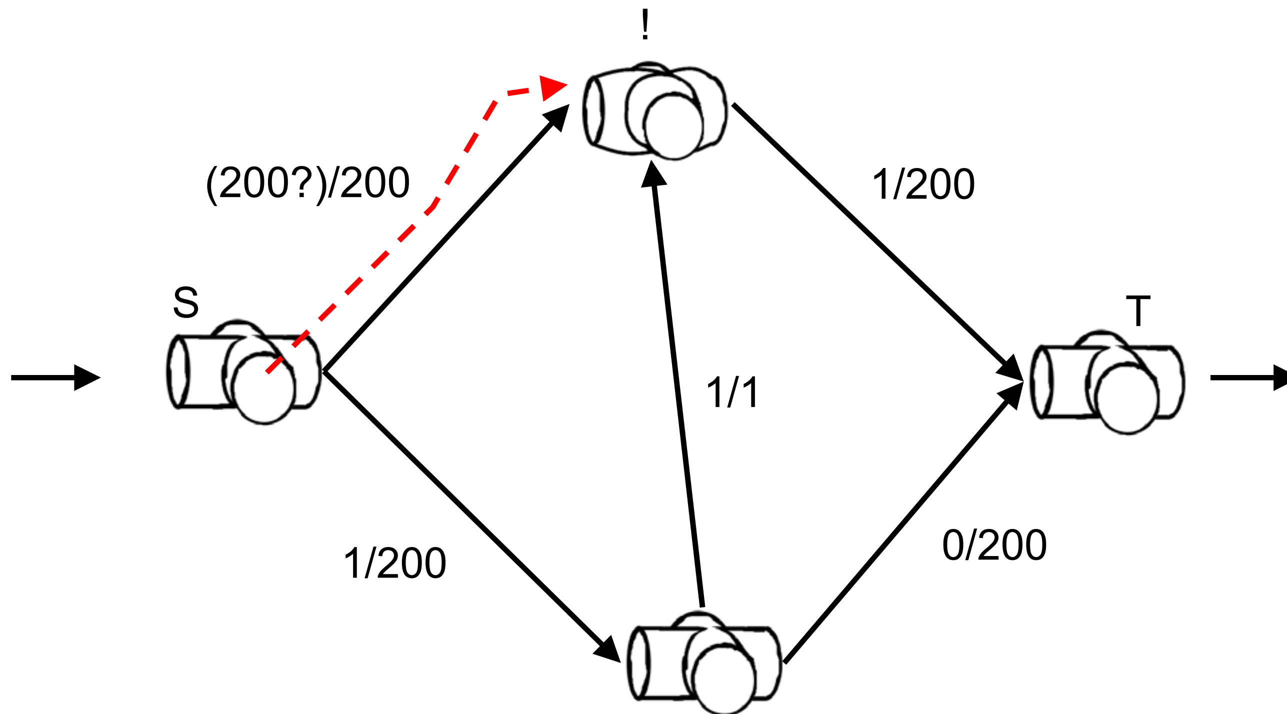
Maksimal flyt med Ford-fulkersons metode



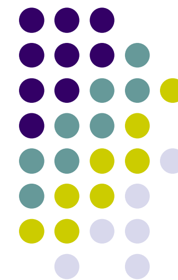


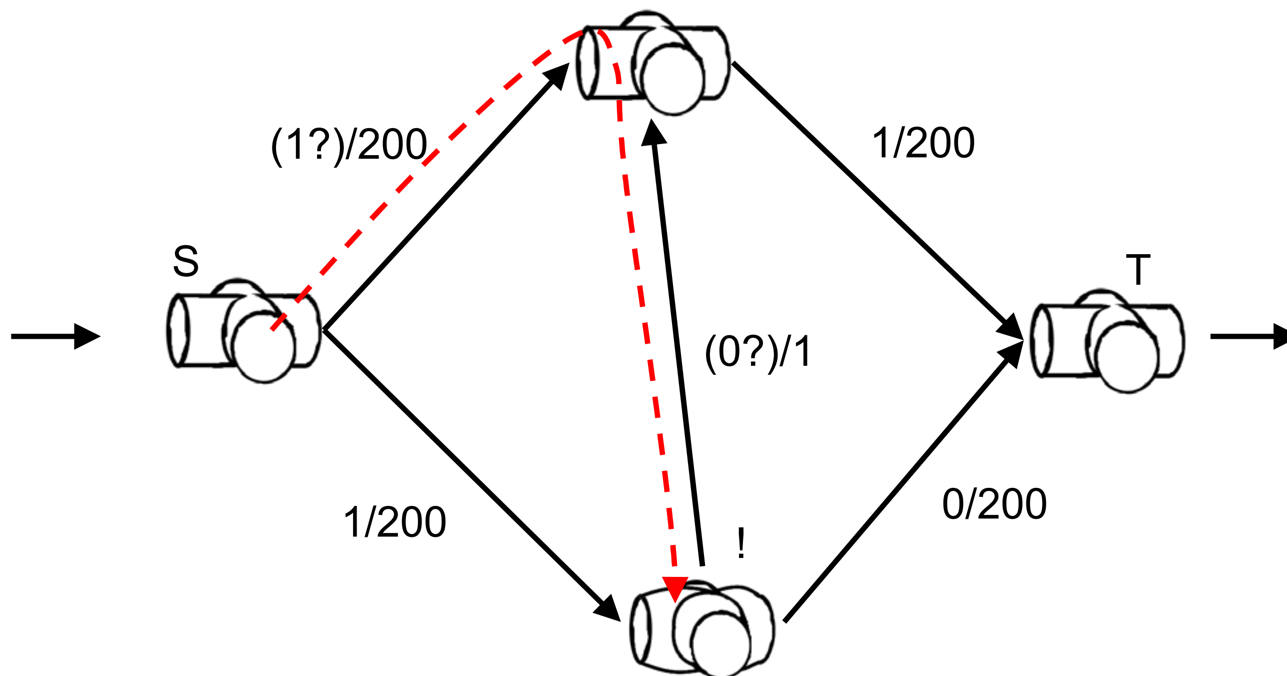
Maksimal flyt med Ford-fulkersons metode



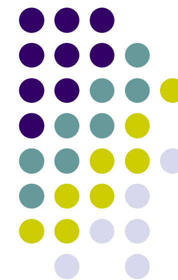


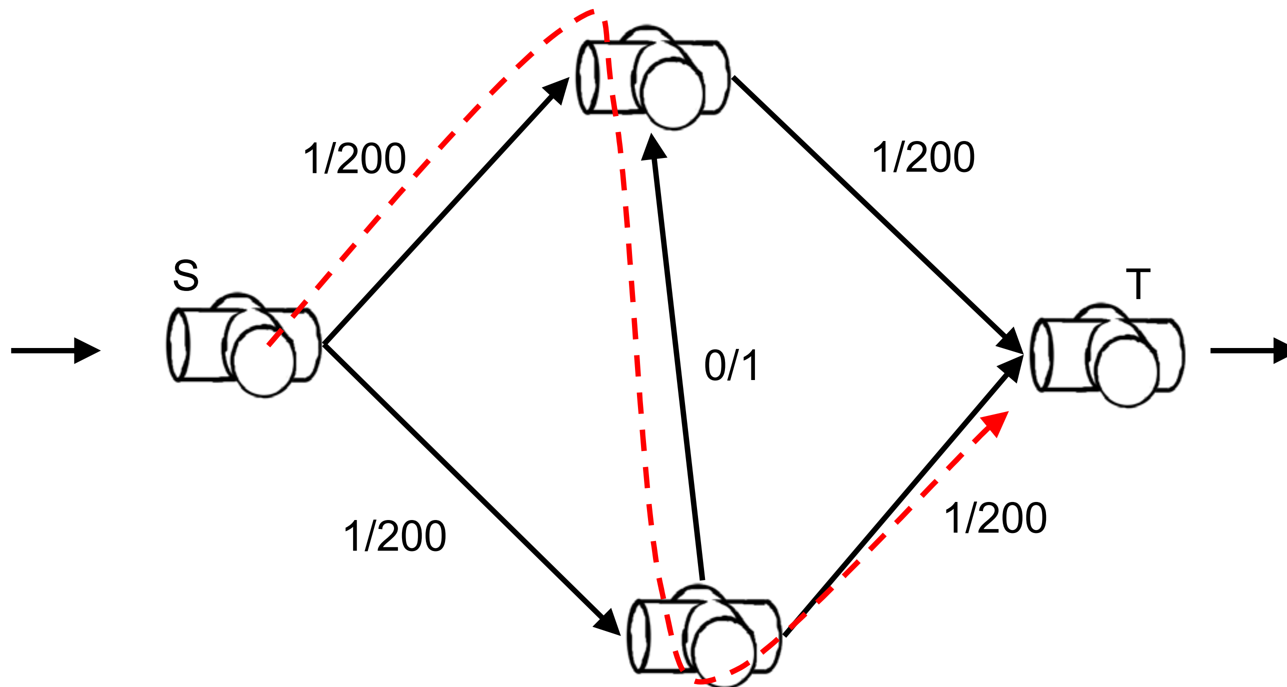
Maksimal flyt med Ford-fulkersons metode

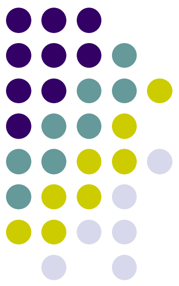




Maksimal flyt med Ford-fulkersons metode



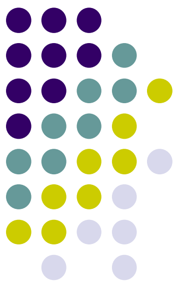




Edmonds-Karps algoritme

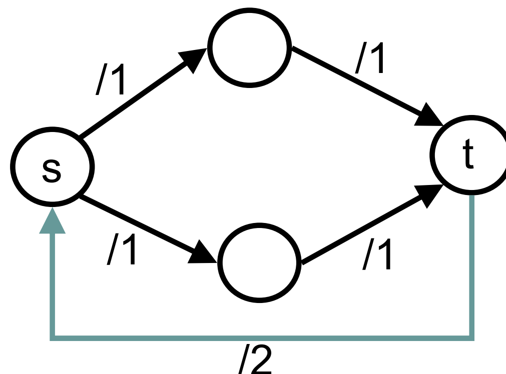
- Ford-fulkerson med garantert worst-case kjøretid E^2V

Finn flytforøkende sti med BFS

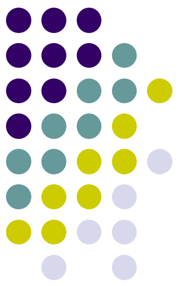


Sirkulasjonsproblemet

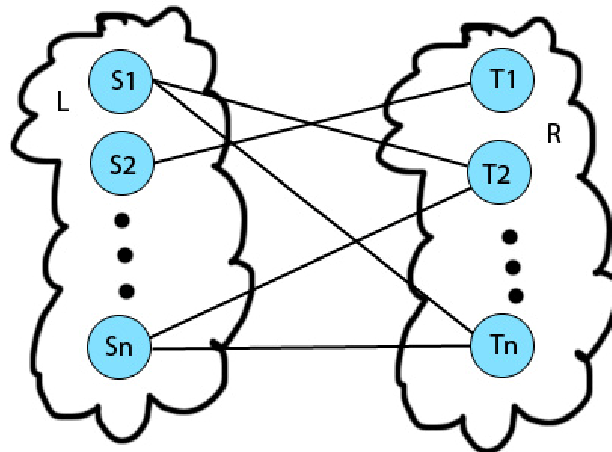
- Det vi kjenner som maks-flyt nå kan generaliseres til et sirkulasjonsproblem.
- I et sirkulasjonsproblem har vi ikke start- og sluttnoder for flyten. Den må gå i en sykel.
- Vi kan konvertere vanlige maks-flyt problemer til sirkulasjon ved å legge til en kant fra terminal til source.



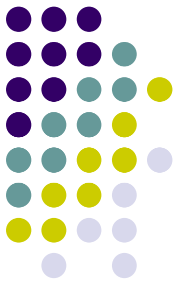
Maksimal bipartitt matching



- Hva er en bipartitt graf?
  - En graf der nodene kan deles opp i to mengder L og R, slik at:
    - Nodene i R bare har kanter til noder i L
    - Nodene i L bare har kanter til noder i R

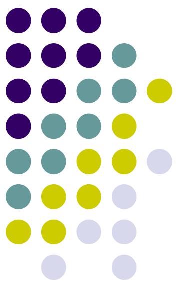


Maksimal bipartitt matching



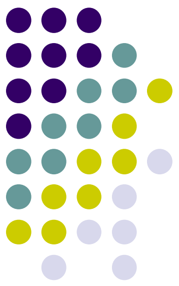
- Eksempel
  - Jenter som skal danse med gutter, noen vil danse med mange, mens noen vil danse med bare én annen. Ikke lov til å danse med samme kjønn. Hvordan få flest mulig personer ut på dansegulvet?

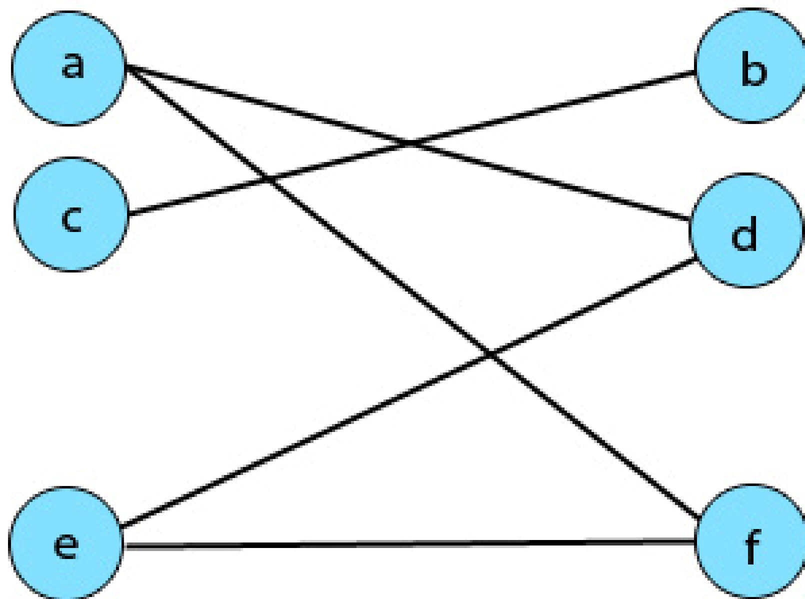
Maksimal bipartitt matching



- Hva er bipartitt matching?
  - Anta G=(V,E) er en bipartitt graf, og M er en undermengde av E, slik at for grafen G' = (V,M) holder følgende egenskap:
    - For alle noder v i V, deg(v) ≤ 1
  - Så hver node kan ha maks 1 nabo
  - Ønsker å maksimere |M|
  - *Maksimal* bipartitt matching er når |M| er størst mulig

Maksimal bipartitt matching





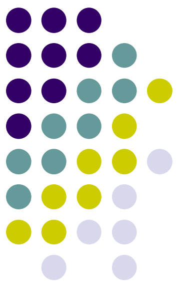
$G = (V,E)$

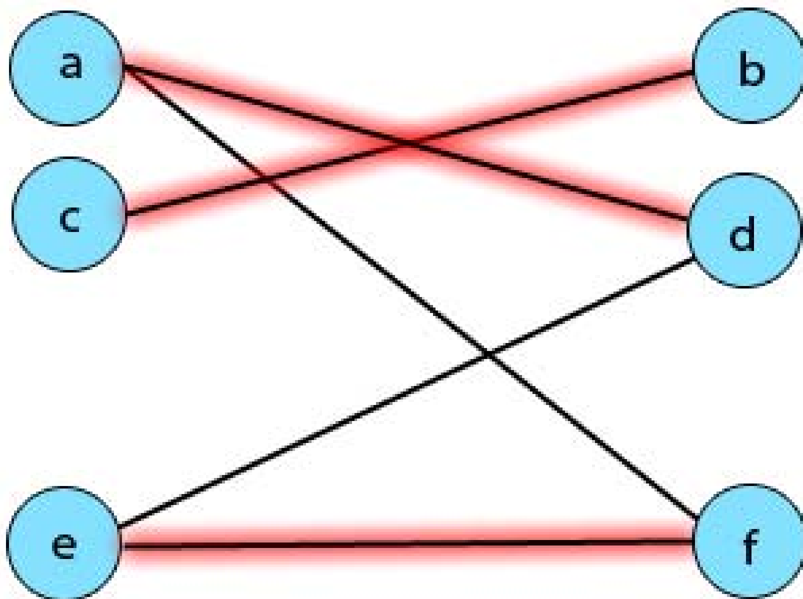
$V = \{a,b,c,d,e,f\}$

$M = \{\}$

$|M| = 0$

Maksimal bipartitt matching





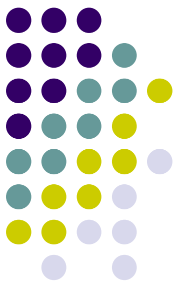
G = (V,E)

V = {a,b,c,d,e,f}

M = {(a,d), (c,b)
(e,f)}

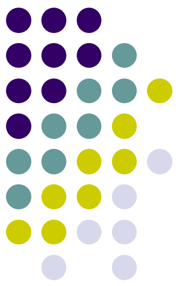
|M| = 3

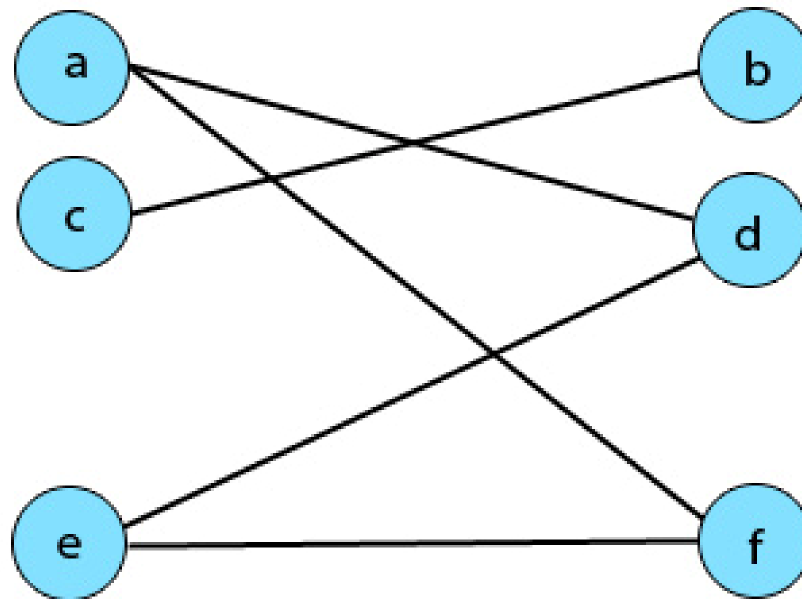
Maksimal bipartitt matching



- Hvordan får vi til maskimal bipartitt matching?
  - Dvs. hvordan maksimerer vi |M| ?
  - Bygger på grafen litt slik at vi får ett flytnettverk
  - Legger til en kilde s, sluk t, retninger på kantene fra L til R, og makskapasitet på hver kant til 1
  - Kilden har en kant til hver node i L, og hver node i R har en kant til sluken

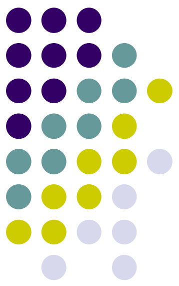
Maksimal bipartitt matching

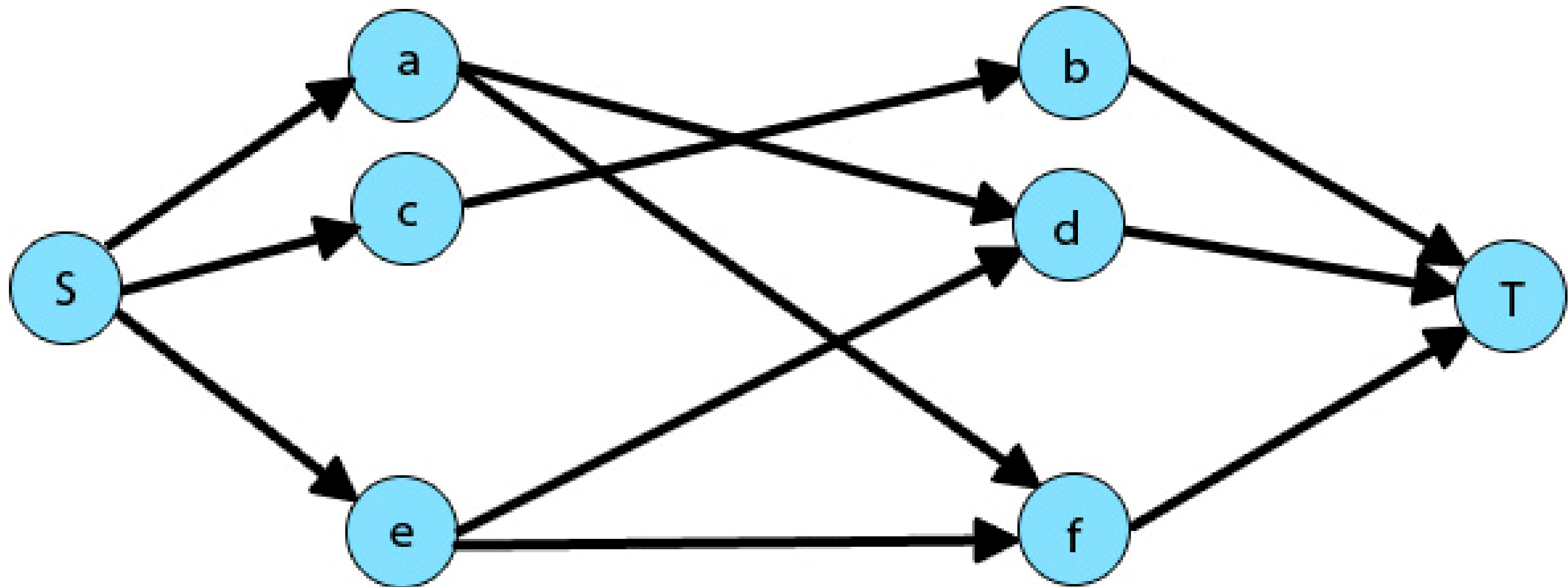




Har en bipartitt graf

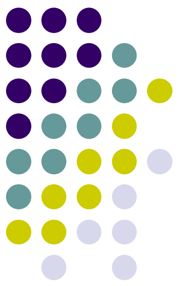
Maksimal bipartitt matching

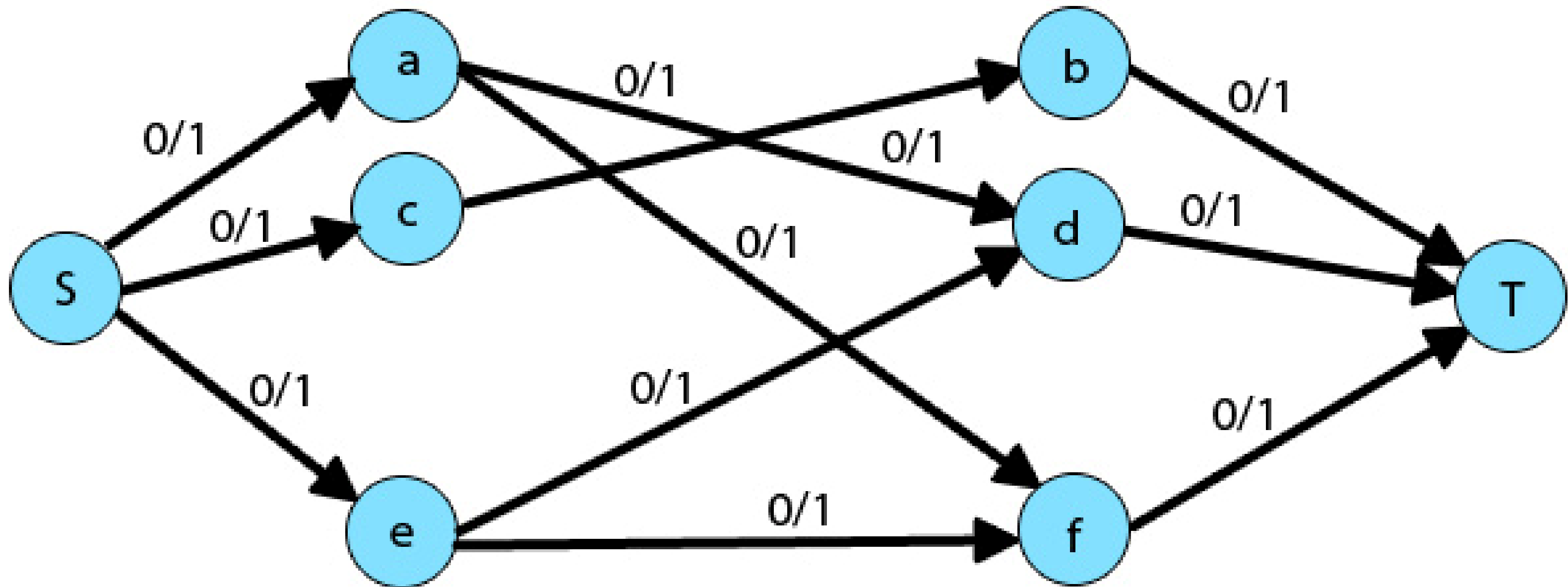




Legger til kilde s og sluk t, og rettede kanter fra s til t

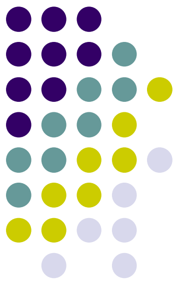
Maksimal bipartitt matching





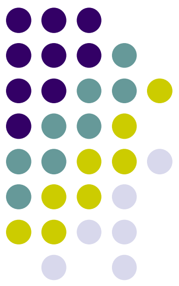
Legger på kapasitet 1 på kantene

Maksimal bipartitt matching



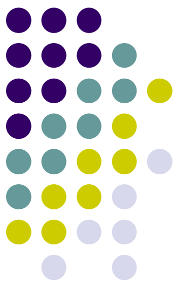
- Etter man har gjort disse stegene, kan man kjøre en flytalgoritme på flytnettverket
- Da vil maksflyten |f| = |M|, og vi har løst problemet med maskimal bipartitt matching
- Brukes Ford-Fulkersens metode blir kjøretiden O(V*E)

Problemklasser



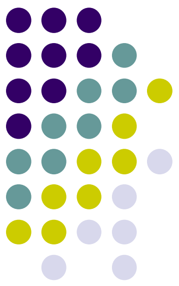
- P: Problemer der vi kan finne en løsning i polynomisk tid
- NP: Problemer der en korrekt løsning lar seg verifisere i polynomisk tid
- NP-Hard: Problemer som alle problemene i NP lar seg redusere til i polynomisk tid. Vanskeligere enn alt annet i NP
- NPC: Problemer som er både NP og NP-Hard

Problemklasser – Eksempler

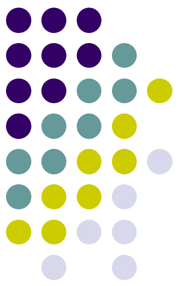


- P: Stort sett alle algoritmer i kurset:
  - Sortering, søking osv…
- NP-Hard:
  - Travelling Salesman Problem, halting-problem
- NPC:
  - Subset-sum, vertex-cover
  - Lurt å ha lest litt på de Hetland har gått igjennom

Avgjørelsesproblemer



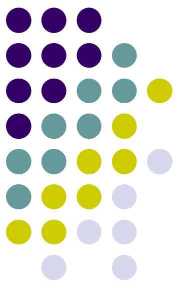
- Vi skal ta for oss avgjørelsesproblemer, som er nært beslektet med optimaliseringsproblemer
- Avgjørelsesproblemer: Problemer der svaret er enten ja eller nei
- Optimaliseringsproblem: Finne det optimale svaret av flere mulige svar, finne max eller min



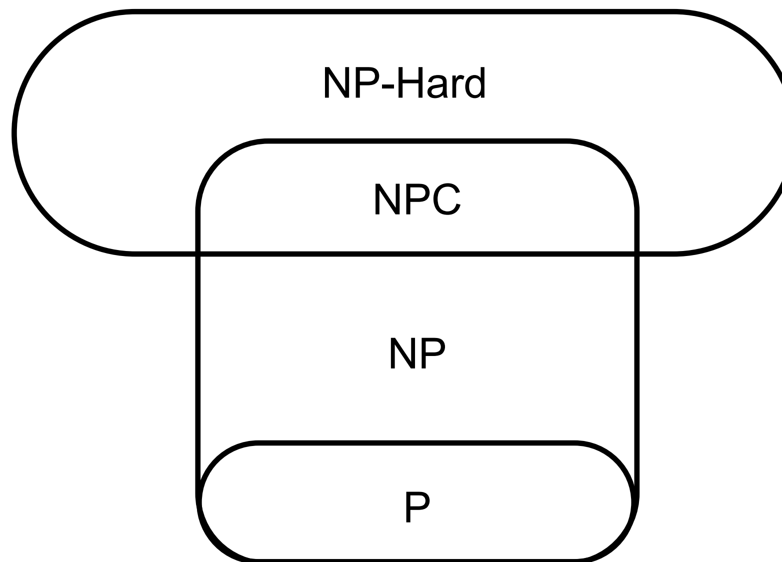
Avgjørelsesproblemer

- SHORTEST-PATH (Optimaliseringsproblem)
  - Finne korteste vei fra *u* til *v* i en urettet, uvektet graf
- PATH (Relatert avgjørelsesproblem)
  - Finne ut om det finnes vei fra *u* til *v* med max. *k* kanter

Problemklassene P, NP, NPC og NP-hard

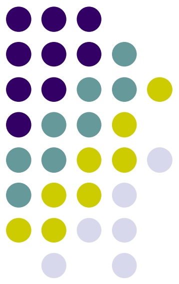


P: Problemer som har en løsning i polynomisk tid
NP: Problemer, hvor en korrekt løsning lar seg verifisere i polynomisk tid.
NP-Hard: Problemer som alle problemene i NP lar seg redusere til i polynomisk tid.
NPC: Problemer som er både NP og NP-Hard



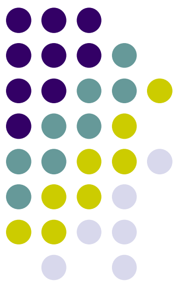
Merk: Det kan hende at P=NPC, eller at alle problemer i NP er enten i P eller NPC.

Hva betyr å redusere til?



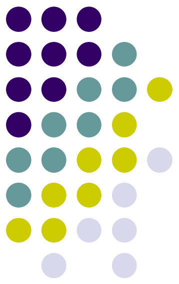
- ”Å løse ved hjelp av”

Hvordan bevise at et problem er NP-komplett?



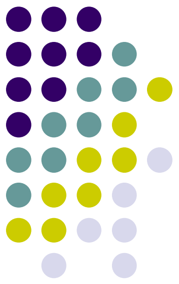
- Finn en måte å transformere et NP-komplett problem til ditt, i polynomisk tid.
- Det vil trolig ikke finnes noen polynomisk løsning på ditt problem, da dette samtidig ville vært en polynomisk løsning på det NP-komplette problemet.

Eksamensforelesning

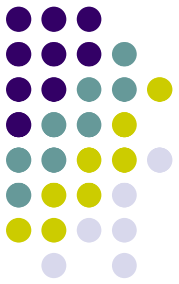


- Lørdag 21. november, 14:15, R1
- Søndag 22. november, 12:15, R1

- Åsmund Eldhuset holder eksamenskurs i Algdat, vi anbefaler dere å møte opp!



Lykke til med eksamen!



Hilsen undass-gruppa

- O Store leder Jon Marius Venstad, og hans disipler

  Magnus Botnan
  Geir-Arne Fuglstad
  Torbjørn Morland
  Børge Rødsjø
  Kristian Veøy