TDT 4120 - Algoritmer og datastrukturer

- Repo: https://github.com/Hoyby/NTNU/tree/master/TDT4120-AlgDat/
- PDF download: https://github.com/Hoyby/NTNU/raw/master/TDT4120-AlgDat/README.pdf

Teori

1. *Problem og algoritmer*
2. *Datastrukturer*
3. *Splitt og hersk*
4. *Rangering i lineær tid*
5. *Rotfaste trestrukturer*
6. *Dynamisk programmering*
7. *Grådige algoritmer*
8. *Traversering av grafer*
9. *Minimale spenntrær*
10. *Korteste vei fra én til alle*
11. *Korteste vei fra alle til alle*
12. *Maksimal flyt*
13. *NP-kompletthet*
14. *NP-komplette problemer*

- Her ligger de fleste av algoritmene som er pensum skrevet i *Python*
- Her ligger kjøretiden på de fleste av algoritmene nevnt i dette dokumentet.

Liste over øvinger:

- ☑ Øving 1
- ☑ Øving 2
- ☑ Øving 3
- ☑ Øving 4
- ☑ Øving 5
- ☑ Øving 6
- ☑ Øving 7
- ☑ Øving 8
- ☑ Øving 9

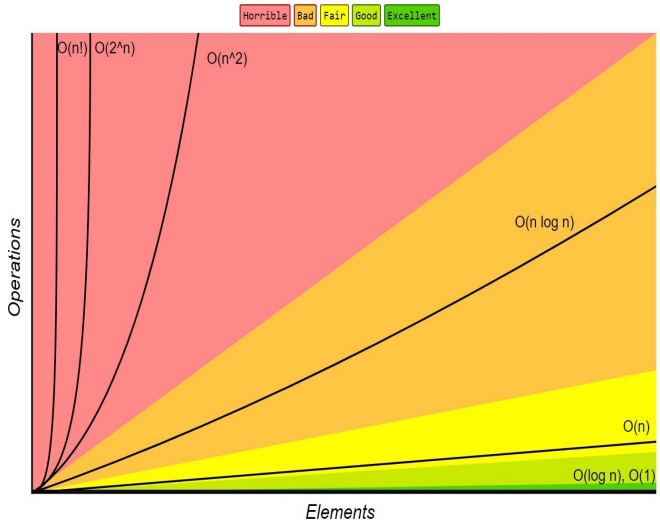
Andre gode kilder:

- https://www.wikipendium.no/TDT4120_Algoritmer_og_datastrukturer/nb/
- https://www.programiz.com/dsa
- https://github.com/henrhoi/Algdat-TDT4120

- https://kjaer.io/algorithms/#why-always-n-log-n

Problem og algoritmer

Asymptotic Complexity Chart | **Asymptotic Notation**



Big-O Notation	Comparison Notation	Limit Definition
$f \in o(g)$	$f \lessdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
$f \in O(g)$	$f \leqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$
$f \in \Theta(g)$	$f = g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} \in \mathbb{R}_{>0}$
$f \in \Omega(g)$	$f \geqslant g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} > 0$
$f \in \omega(g)$	$f \gtrdot g$	$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \infty$

O: Øvre grense **Ω**: Nedre grense **Θ**: Øvre og nedre grense

Kjøretid:

- Best-case:
- Worst-case:
- Average-case:

Amortisert analyse:

I en amortisert analyse, regner vi ut den gjennomsnittlige tiden for å utføre en sekvens av operasjoner over alle operasjonene som ble utført. Amortisert analyse vurderer både kostbare og rimeligere operasjoner sammen over hele serien av operasjoner av algoritmen og viser at gjennomsnittskostnaden for en operasjon kan være liten, selv om en enkelt operasjon i sekvensen kan være dyr. Amortisert analyse skiller seg fra gjennomsnittsanalyse ved at sannsynligheten ikke er involvert – En amortisert analyse garanterer gjennomsnittlig ytelse for hver operasjon i verste tilfelle.

Eksempel:

Vi vil definere en *load-factor* α til en ikke-tomt array *T* til å være `α = elements/A.length`

- **Tabell-ekspansjon**:
  - Et array er full når enten alle plassene i arrayet er i bruk eller når *load-factoren* α = 1.
  - Dersom vi skal innsette et element i et fullt array, må vi ekspandere arrayet, ved å lage et nytt array med fler plasser enn den gamle og kopiere over alle de gamle elementene.

- Så en gang i blant dersom α = 1 vil innsetting av et element bruke mye lenger tid enn *O(1)*, og dette tar vi med i beregningen med **amortisert analyse**.

```
TABLE-INSERT(T,x):
1    if T.size == 0:
2        allocate T.table with 1 slot
3        T.size = 1
4
5    if T.num == T.size:
6        allocate new-table with 2 * T.size slots
7        insert all items in T.table into new-table
8        free T.table
9        T.table = new-table
10       T.size = 2 * T.size
11
12   insert x into T.table
13   T.num = T.num + 1
```

Dynamisk array: Tabell som automatisk blir utvidet dersom alle plassene er tatt eller *load-factor* α = 1.

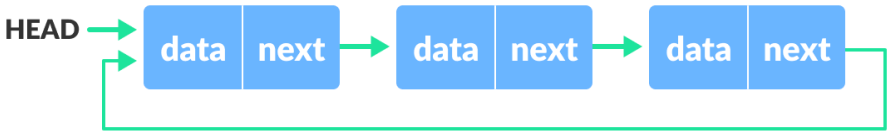
Datastrukturer

Lenket liste

- *Enkle* lenkede lister

- *Doble* lenkede lister

- *Sykliske* lenkede lister



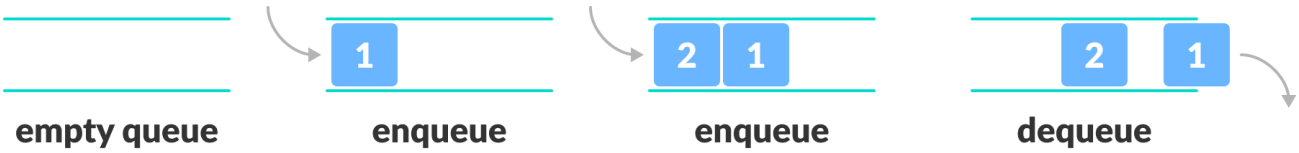
Kjøretider *(antar enkel lenket liste)*:

```
*  Innsetting i starten: *O(1)*
*  Innsetting i slutten: *O(n)*
*  Oppslag: *O(n)*
*  Slette element: *Oppslagstid* + *O(1)* = *O(n)*
```

> Ved dobbel lenke liste blir det lett med innsetting, trenger kun å endre *.prev* og *.next* til de nye naboene. Dette gjøres i O(1)

Queue

FIFO-struktur (First in First Out)



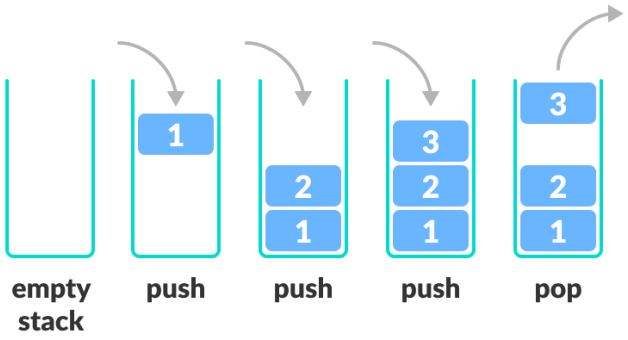

```python
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()
```

> Operasjonene bruker *O(1)* tid

> En *prioritetskø* fjerner elementene i køen som har høyest prioritet fremfor å følge den vanlige FIFO-strukturen.

Stack

LIFO-struktur (Last in First Out)

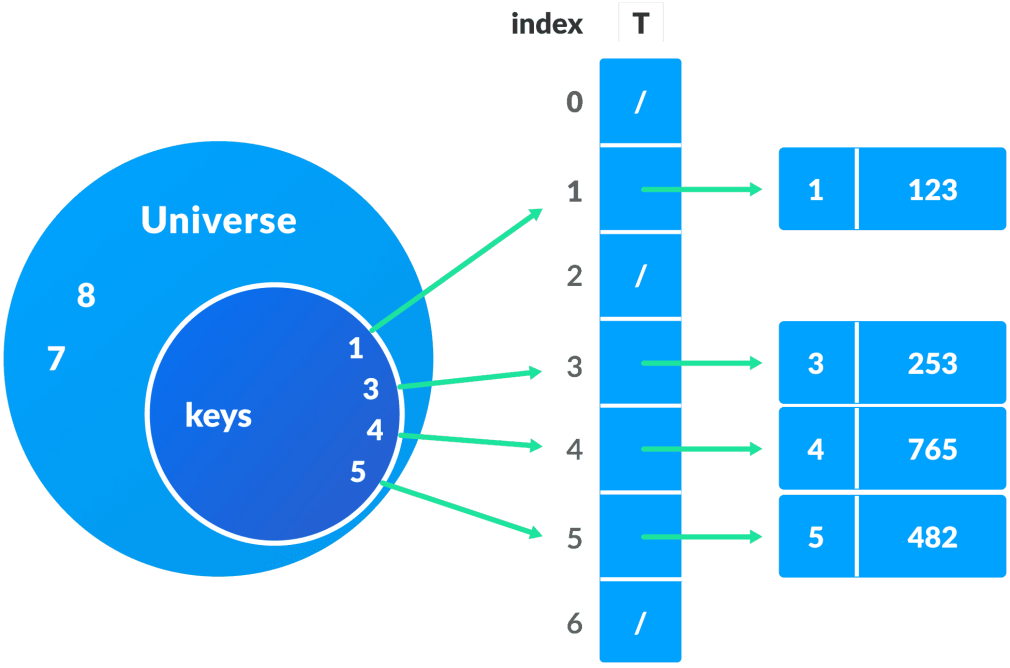



```
class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()
```

> Operasjonene bruker *O(1)* tid

Hash-tabeller

En fornuftig måte å adressere og komprimere.



Bruker hash-funksjoner og nøkler slik at et element *e* med nøkkel *k* blir lagret på plass h(*k*).

- Vi kan løse kollisjon-problemet ved chaining med lenkede lister.
- Idéen med hash-tabeller er å lage *h* slik at den virker "random" for å forhindre kollisjon eller i det minste; minske antallet.
- Ved hash-tabell med lenket liste har vi metodene:
  - Chained-Hash-Insert (T, x) `- insert x at the head of list T[h(x.key)`
  - Chained-Hash-Search (T, k) `- search for an element with key k in list T[h(k)]`
  - Chained-Hash-Delete (T, x) `- delete x from list T[h(x.key)]`
- WC for insertion er O(1)
- WC for søk er O(n), ønsker å ha O(1), så kan være dårlig hashemetode som forårsaker dårlig søketid.
- Vi kan slette et element med O(1) dersom hash-tabellen bruker *doble* lenkede lister.
- *Hva karakteriserer en god hashefunksjon:* Den unngår kollisjoner, og like sannsynlig for hver mulige nøkkel å bli plassert et sted.

Splitt og hersk

Designmetoden i splitt og hersk:

- **Splitt** problemet inn i subproblemer som er mindre instanser av det samme problemet.

- **Hersk** subproblemene ved å løse dem rekursivt. Hvis et subproblems størrelse er lite nok, løs problemet direkte.
- **Kombiner** løsningene på subproblemene inn i løsningen på problemet i utgangspunktet

Vi deler opp problemet helt til vi kommer til minste mulige instans av problemet, så sier vi at rekursjonen har "bottoms out" og vi har kommet til base case. Vi får resultatet når vi kombinerer løsningene.

Binærsøk:

Input: En sortert liste A, pivot-element *p*, slutt-element *r* og elementet *v* som vi søker etter

Output: Indeks *i* slik at `A[i] = v`

Rekursiv løsning:

```
def Recursive_binary_search(A, p, r, v):
    i = p
    if p < r:
        mid = (p+r)//2
        if v <= A[mid]:
            i = Recursive_binary_search(A,p,mid,v)

        else:
            i = Recursive_binary_search(A,mid+1,r,v)
    return i
```

Iterativ løsning:

```
def Iterative_binary_search(A, p, r, v):
    while p < r:
        mid = (p+r)//2

        if v <= A[mid]:
            r = mid

        else:
            p = mid + 1

    return p
```

> Dersom det finnes flere forekomster av *v* i A vil Bisect returnere indeksen til forekomsten lengst til venstre, altså den **laveste** indeksen

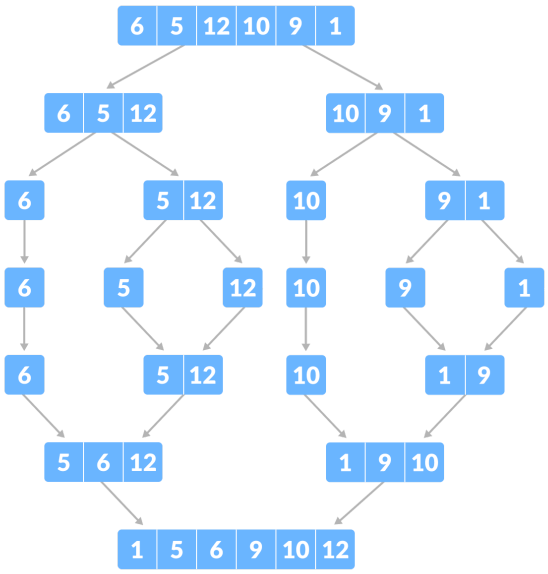
Kjøretid:

- $\Theta(lg\ n)$

Merge sort

- Sammenligningsbasert
- Stabil
- Splitt og hersk rekursiv

1. **Splitt:** Del-opp steget regner kun ut midten av listen, som tar konstant tid. Da blir `D(n) = Θ(1)`.
2. **Hersk:** Vi løser rekursivt to delproblemer, hver på størrelse n/2, som bidrar med `2*T(n/2)` kjøretid på algoritmen.
3. **Kombiner:** Merge-prosedyren bruker Θ(n) tid på en n-element liste, så derfor blir `C(n) = Θ(n)`



Når vi adderer funksjonene D(n) og C(n) for merge-sort analysen, vil summen av (n) og (1), bli (n). Når vi summerer det igjen sammen med 2T(n/2)-delen fra ”hersk”-seget gir rekurrensen for verste kjøretiden T(n) for merge-sort:

`T(n) = 2T(n/2) + Θ(n) if n > 1, else O(1)`

> Dersom vi bruker master-teoremet (**Kap. 4**) så kan vi vise at `T(n) = (n lg n)`.

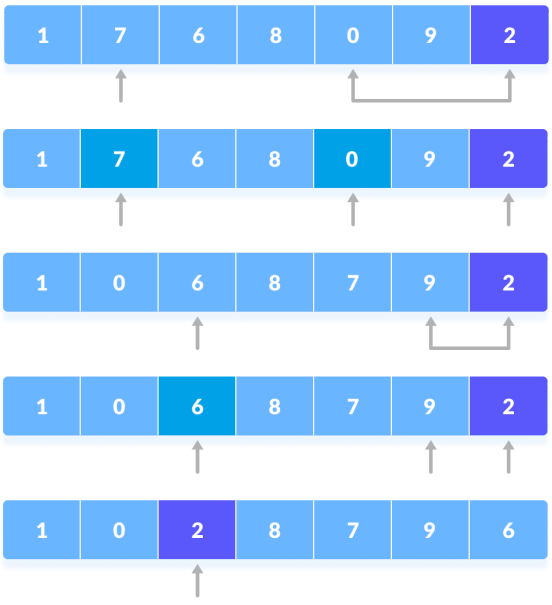
```
def merge_sort(A):
    if len(A)>1:
        q = len(A)//2
        lh = merge_sort(A[:q])
        rh = merge_sort(A[q:])
        return merge(lh,rh)

    return A


def merge(lh,rh):
    res = []
```

```
    i = 0
    j = 0

    while i<len(lh) and j<len(rh):
        if lh[i] < rh[j]:
            res.append(lh[i])
            i+=1

        else:
            res.append(rh[j])
            j+=1

    if i<len(lh): res.extend(lh[i:])
    if j<len(rh): res.extend(rh[j:])

    return res
```

Quicksort

- Sammenligningsbasert
- Ikke Stabil
- Splitt og hersk rekursiv

1. **Splitt:** Del opp listen `A[p..r]` i to sublister ved å benytte en pivot `q`-, slik at hvert element i `A[p..q-1]` er mindre eller lik pivoten, som igjen er mindre eller lik hvert element i `A[q+1..r]`.
2. **Hersk:** Sorter de to listene `A[p..q-1]` og `A[q+1..r]` igjen med rekursive kall til quicksort.
3. **Kombiner:** Fordi sublistene allerede er sortert, trengs det ikke å gjøres noe for å kombinere dem: hele listen `A[p..r] er nå sortert.



```
def Quicksort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        Quicksort(A, p, q-1)
        Quicksort(A, q+1, r)


def Partition(A, p, r):
    # Partition jobber slik
    # ≤x | ≥x | x

    x = A[r]
    i = p-1
    for j in range(p, r):
        if A[j] <= x:
            i += 1
            A[i], A[j] = A[j], A[i]

    A[i+1], A[r] = A[r], A[i+1]
    # Listen blir slik slik:
    # ≤x | x | ≥x
    return i+1
```

> *Quicksort* er **ikke stabil**, da den ikke beholder den relative rekkefølgen til like elementer under sorteringen av listen.

Partition:

Listeelementet `A[r]` blir pivot-elementet *x*. Listen itererer fra venstre til høyre og sammenlikner veriden med pivoten. Er den høyere blir den swappet med det første elementet i listen, dette gjør den for alle elementer i listen, og resulterer i en liste som ser slik ut: ≤x | x | ≥x

> De to siste linjene i Partition avslutter prosedyren ved å bytte pivot elementet A[r] med A[i+1]

Kjøretid:

- *Worst-case:* $\Theta(n^2)$
- *Forventet kjøretid:*
  - Rekursjonstre med dybde $\Theta(lg\ n)$ med $O(n)$ arbeid på hvert nivå
  - `T(n) = 2T(n/2) + Θ(n) = Θ(n lg n)`

Randomized-Quicksort

Samme algoritme som *quicksort*, bortsett fra at pivot-elementet byttes ut med et tilfeldig element fra listen. Vil gi færre tilfeller av worst-case-kjøretid.

```
def Quicksort(A, p, r):
    if p < r:
```

```
        q = Randomized_Partition(A, p, r)
        Quicksort(A, p, q-1)
        Quicksort(A, q+1, r)


def Randomized_Partition(A, p, r):
    i = random.randint(p,r)
    A[i], A[r] = A[r], A[i]

    return Partition(A,p,r)


def Partition(A, p, r):
    x = A[r]
    i = p-1
    for j in range(p, r):
        if A[j] <= x:
            i += 1
            A[i], A[j] = A[j], A[i]

    A[i+1], A[r] = A[r], A[i+1]
    return i+1
```

Rangering i lineær tid

Sammenligningsbasert sortering:

Disse algoritmene benytter seg kun av sammenlikning av input-elementene. Slike sorteringsalgoritmer har følgende kompleksiteter:

Worst	Average	Best
$T_W(n) = O(∞)$	$T_A(n) = O(∞)$	$T_B(n) = O(∞)$
$T_W(n) = Θ(?)$	$T_A(n) = Θ(?)$	$T_B(n) = Θ(?)$
$T_W(n) = Ω(n lg(n))$	$T_A(n) = Ω(n lg(n))$	$T_B(n) = Ω(n)$

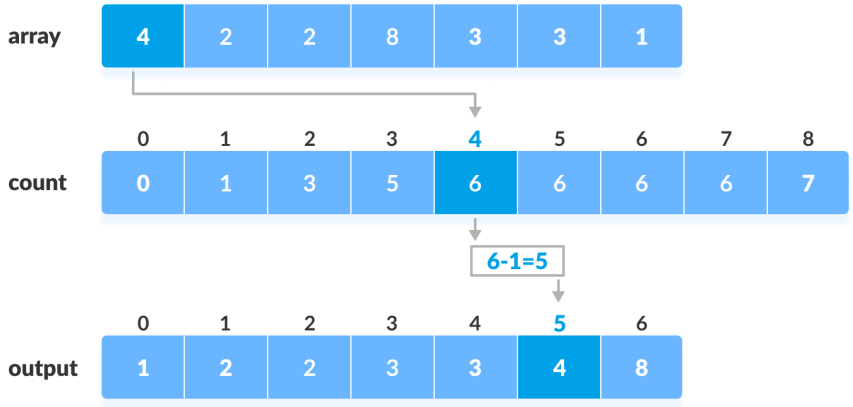
> Enhver sammenligningsbasert sorteringsalgoritme krever (n lg n) sammenlikninger i worst case.

Counting Sort

- Ikke sammenligningsbasert
- Stabil

Counting sort antar at hvert av de n elementene er et tall mellom 0 og *k*. Når *k* er *O(n)*, sorterer algoritmen på Θ(*n*).

Algoritmen teller antall forekomster av verdiene og legger den kumulative summen på indexene til den nye listen, `count[0..k]`.




```
def counting_sort(A,k):
    res = [0]*len(A)
    count = [0 for _ in range(k+1)]

    for j in range(0,len(A)):
        count[A[j]] += 1

    # C[i] inneholder nå antall forekomster av element i

    for i in range(1,k+1):
        count[i] += count[i-1]
    # Count er nå kumulativ sum
    # C[i] inneholder nå antall elementer mindre eller lik i

    # Itererer baklengs gjennom A, for at Counting blir stabil. Trekker fra
en på count når vi plasserer et element
    for j in range(len(A)-1,-1,-1):
        element = A[j]

        res[count[element]-1] = element
        count[element] -= 1

    return res
```

Radix sort

*Radix sort *er algoritmen som brukes i kort-sortering maskiner. Radix sort løser problemet ved å sortere på det *least significant digit* først.

```
RADIX-SORT(A, d)
    for i = 1 to d
        use a stable sort to sort array A on digit i
```

Input: En liste *A* med *n* elementer bestående av *d* siffer

Output: Sortert liste bestående av elementene i *A*

 radix

```
def radix_sort(A, d):
    for i in range(d-1,-1,-1):
        # Bruker vlagfri stabil sorterings algoritme
        A = counting_sort(A,9,i)

    return A


# Sorterer større tall ved å kun se på et siffer.
# k = støste tall (9), i = sifferindeks

def counting_sort(A,k,d):
    res = [0]*len(A)
    count = [0 for _ in range(k+1)]

    for j in range(0,len(A)):
        element = int(str(A[j])[d])
        count[element] += 1

    # C[i] inneholder nå antall forekomster av element i

    for i in range(1,k+1):
        count[i] += count[i-1]
    # Count er nå kumulativ sum
    # C[i] inneholder nå antall elementer mindre eller lik i

    # Itererer baklengs gjennom A, for at Counting blir stabil. Trekker fra
en på count når vi plasserer et element
    for j in range(len(A)-1,-1,-1):
        element = A[j]

         #Plasserer hele elementet i listen selvom jeg sorterer på hensyn
på ett siffer
        res[count[int(str(element)[d])]-1] = element
        count[int(str(element)[d])] -= 1

    return res
```

Gitt *n* *d*-siffrede tall kan hvert siffer være en av *k* mulige verdier, vil Radix sort sortere disse tallene i `Θ(d (n + k))` tid, hvis den stabile sorteringsalgoritmen bruker `Θ(n + k)` tid.

Viktig at sorteringsalgoritmen vi velger er **stabil** fordi at elementene med likt tall på siffer *d* ikke mister sin relative rekkefølge og ødelegger for sorteringen på de tidligere sorteringskallene.

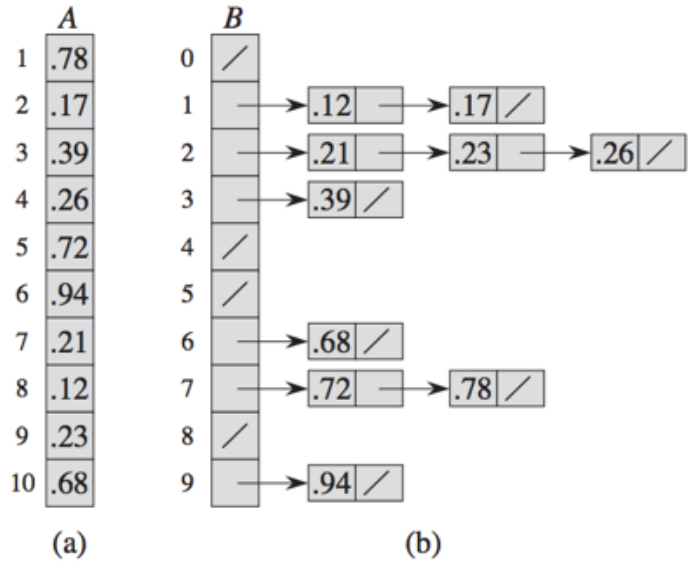
Bucket sort

Bucket sort antar at instansen er tatt fra en uniform fordeling og har en average-case kjøretid på `O(n)`, og worst-case `O(n^2)`.

Som *Counting sort* er Bucket sort rask fordi den gjør antagelser på instansen. Bucket sort deler opp intervallet `[0, 1)` inn i n like store intervaller, eller **buckets**.

```
def bucket_sort(A):
    n = len(A)
    B = [[] for _ in range(n)]

    for i in range(n):
        B[int(n*A[i])].insert(-1, A[i])

    for j in range(n):
        insertion_sort(B[j])

    res = []
    for i in range(len(B)):
        res += (B[i])

    return res
```

> Tallet `int(n*A[ i ])` gir hvilken bucket som elementet skal legges i, `n*A[ i ]` rundes ned og blir en verdi i intervallet `[0, 1)` som har *n* buckets



Kjøretid:

- Average-case: *O(n)*
- Worst-case: $O(n^2)$

Minimum og maksimum

```
MINIMUM(A):
1    min = A[0]
2    for i in range(1,len(A))
3        if min > A[i]:
4            min = A[i]
5    return min
```

> Finner maksimum ved å ende `min > A[i]` til `min ≤ A[i]`

Randomized-Select

Randomized-Select jobber kun på **én** side av partisjoneringen, og har derfor forventet kjøretid på `O(n)`, og worst-case `O(n^2)`. Algoritmen skal returnere det *i*'te minste elementet i listen `A[p .. r]`. Bruker litt samme fremgangsmåte som quicksort, men pivot er alltid på siden av tallet vi skal finne. Når sorteringen har kommet frem til indexen vi leter etter vet vi verdien.

Input: En liste *A* med pivot-element *p*, sluttelement *r* og ønske om å finne *i* ´te minste element i *A*

Output: Indeks i A til *i* ´te minste element

```
RANDOMIZED-SELECT(A,p,r,i):
def randomized_select(A,p,r,i):
    if p == r:
        return A[p]
    q = randomized_partition(A,p,r)

    # k er antall tall til venstre for q, dvs. at det finnes nøyaktig k
tall mindre enn A[q]
    k = q - p + 1
    if k == i:
        return A[q]
    elif i < k:
        return randomized_select(A,p,q-1,i)
    else:
        return randomized_select(A,q+1,r,i-k)
```

- Trykk for video for bedre forklaring!

Kjøretid:

- Expected-case: $\Theta(n)$
- Worst-case: $\Theta(n^2)$

Select

Som *Randomized-Select*, finner *Select* et ønsket element gjennom rekursiv partisjonering av input. I motsetning til Randomized-Select, kan vi *garantere* en god split under partisjoneringen. *Select* bruker den deterministiske part. algoritmen *Partition*, med modifisert til å ta inn hvilket element som partisjoneringen skal skje rundt.

Select-algortimen returnerer det *i* 'te minste elementet i input med n > 1 distinkte elementer ved å gjennomføre følgende steg. Dersom n = 1, returnerer den bare input.

1. Del opp de *n* elementene i input til $\lfloor n/5 \rfloor$ grupper med 5 elementer hver, og på det meste en gruppe bestående av de gjenstående *n* mod 5 elementene.

2. Finn medianen til hver av de $\lceil n/5 \rceil$ gruppene ved å sortere elementene (≤ 5) med Insertion-sort, og velg deretter median.

3. Bruk *Select* rekursivt for å finne medianen *x* av de $\lceil n/5 \rceil$ medianene i steg 2. Hvis det er partalls medianer blir *x* den mindre medianen.

4. Partisjoner input rundt median av medianer *x* ved å bruke den modifirserte versjonen av *Partition*. La *k* være en større en antall elementer på venstre side av partisjoneringen, slik at *x* er det *k* 'te minste elementet og det der *n* - *k* elementer på høyre side.

5. Dersom `i == k`, returner *x*. Hvi sikke bruk *Select* rekursivt for å finne det *i* 'te minste elementet på venstre side `if i < k` eller det (*i* - *k*)'te minste elementet på høyre side `if i > k`.

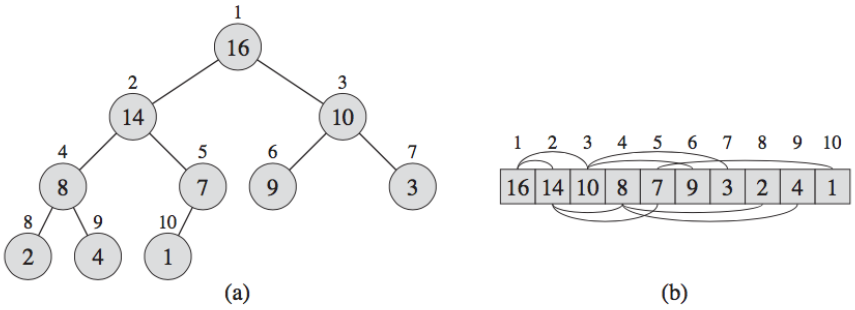
```
SELECT(A,i)
    if A.length = 1
        return A[0]
    if A.length ≤ 5
        INSERTION-SORT(A)
        return A[i]
    Partition L into the subsets S[i] with five elements each
        # There will be n/5 ± 1 subsets total.
    for i = 1 to n/5
        x[i] = select(S[i],3)
    M = select({x[i]}, n/10)
    Partition A into L[..] < A[M] and R[..] > A[M]

    if k <= length(L)
        return select(L,k)
    elif k > length(L)
        return select(R,i-len(L))
    else return A[M]
```

- *Select* kodet i Python ligger **her**

Rotfaste trestrukturer

Den **binære heap** datastrukturen er en liste som vi kan se på som et nesten komplett binærtre. Hver node i treet korresponederer til et element til listen. Treet er helt fylt i alle nivående med unntak av mulig det laveste, som er fylt fra venstre mot høyre.

Roten til treet er A[0] og gitt en index i til en node, kan vi lett finne indeksen til dets forgjenger, venstre barne eller høyre barn



```
def parent(i):
    return ⌊i/2⌋

def left(i):
    return 2*i

def right(i):
    return 2*i + 1
```

Det finnes to typer binære heaps. I begge typene tilfredsstiller verdiene i nodene en heap-egenskap, som avhenger av typen heap:

- **Max-heap egenskapen**:
  - For hver node $i \neq 0$ er `A[parent(i)] ≥ A[i]`
  - En nodes verdi er på det meste sin forgjengers verdi - dvs største element ligger i roten.
- **Min-heaps egenskapen**:
  - For hver node $i \neq 0$ er `A[parent(i)] ≤ A[i]`
  - En nodes verdi er på det minste sin forgjengers verdi - dvs. minste element ligger i roten.

Dersom vi ser på en heap som et tree, definerer vi *høyden* til en node i treet til å den lengste enkle veien fra noden til en løvnode, og vi definer *høyden* til treet til å være høyden til roten.

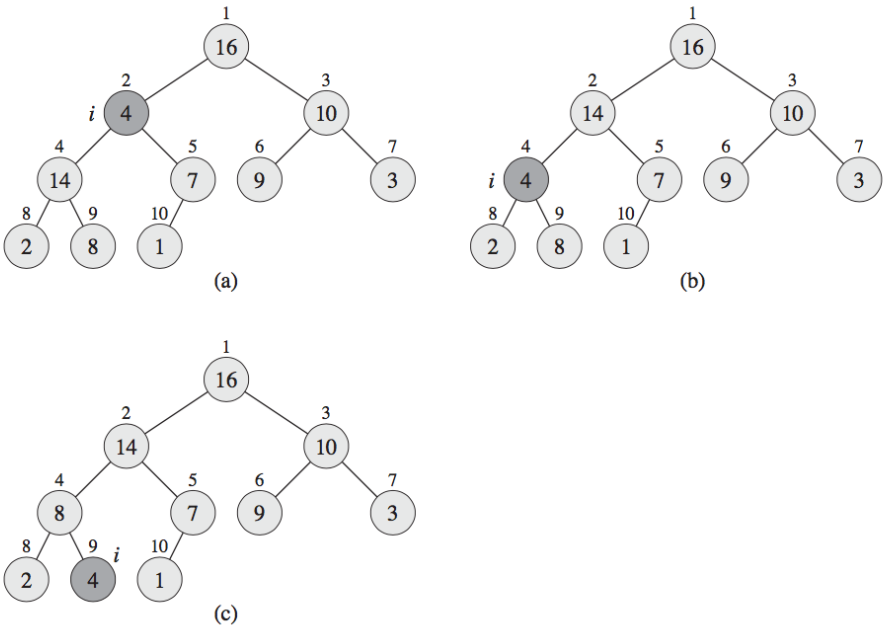
> Siden en heap av n elementer er basert på et komplett binært tre, er dens høyde $\theta(\lg n)$, so vi ser igjen på tradisjonelle heap-prosedyrer.

Max-Heapify

For å bygge en heap med *max-heap egenskapen*, kaller vi på prosedyren *Max-Heapify*. Når den kalles antar algoritmen at binærtreet med røtter i `left(i)` og `right(i)`, er max_heaps, men at A[i] kanskje er mindre enn sine barn, som bryter med heap-egenskapen. *Max-Heapify* lar verdien til A[i] "*flyte ned*" i max-heapen slik at subtreet med rot på index i holder heap-egenskapen.

Problem: Gjøre at input holder *heap-egenskapen*

```
MAX-HEAPIFY(A, i)
1   l = left(i)
2   r = right(i)
3   if l ≤ A.heap-size and A[l] >A[i]
4       largest = l
5   else largest = i
6   if r ≤ A.heap-size and A[r] > A[largest]
7       largest = r
8   if largest ≠ i
9       exchange A[i] with A[largest]
10      MAX-HEAPIFY(A,largest)
```



Kjøring av *Max-Heapify* :

- På hvert steg velges det største elementet av A[i], A[left(i)] og A[right(i)], og dets indeks blir lagret som *largest*. Dersom A[*i*] er størst vil subtreet på node *i* allerede være en max-heap og prosedyren terminerer.
- Hvis ikke er en av de to barna det største elementet, og bytter vi plass på A[*i*] og A[*largest*], som gjør at node *i* og dets barn tilfredstiller max-heap egenskapen.
- Noden med indeks *largest* har nå den orginale verdien til A[*i*], og derfor kan det hende at subtreet med rot *largest* muligens bryter med max-heap egenskapen. Derfor kaller vi *Max-Heapify* rekursivt på subtreet.

Kjøretid:

- `T(n) ≤ T(2n/3) + θ(1)`, som med *master teoremet* gir `T(n) = O(lg n)`
- Alternativt kan vi karakterisere kjøretiden på en node med høyde *h* som `O(h)`

Bygging av heaps

Vi kan bruke *Max-Heapify* på en bottom-up måte for å convertere en liste A[0..n-1], hvor `n = A.length`, til en max-heap. Elementene i listen `A[(⌊n/2⌋+1)..n]` er alle blader i treet, og alle er til å begynne med en 1-element heap.

Prosedyren *Build-Max-Heap* går igjennom de **resterende** nodene av treet og kjører *Max-Heapify* på hver node.

```
BUILD-MAX-HEAP(A)
1   A.heap-size = A.length
2   for i = ⌊A.heapsize/2⌋ downto 1
3       MAX-HEAPIFY(A, i)
```

Vi kan regne ut en øvre grense for kjøretiden til *Build-Max-Heap* som følgende:

- Hvert kall på *Max-Heapify* koster `O(lg n)`, og *Build-Max-Heap* gjør `O(n)` slike kall.
- Derfor blir **kjøretiden** `O(n lg n)`, Det er en øvre grense, men ikke asymptotisk rett.
- Vi kan sette en grense på **kjøretiden** til *Build-Max-Heap* som `O(n)` da vi ser på høyden til nodene kaller *Max-Heapify* på ikke gir O(lg n) på alle kallene.

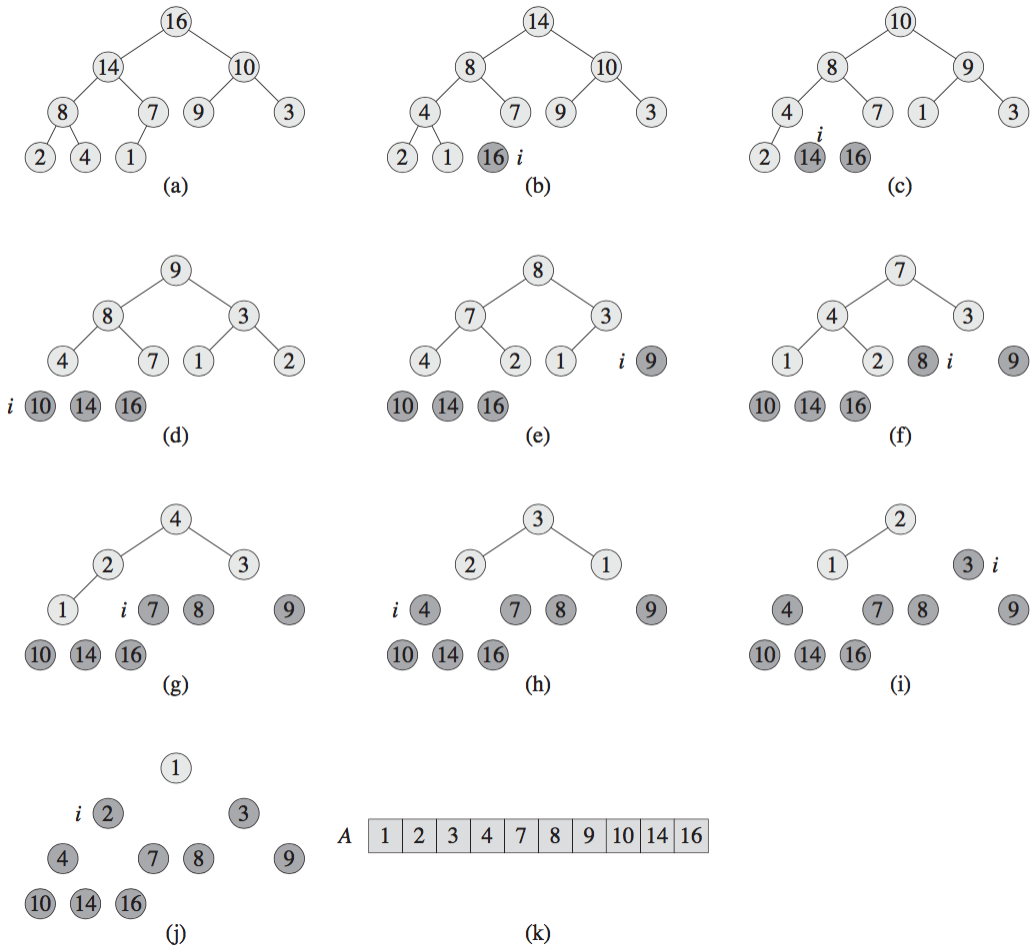
Heapsort

Heapsort-algoritmen starter med å bygge en max-heap av input `A[1..n]`. Siden det største elementet nå ligger som roten *A[1]*, kan v putte den i sin endelige posisjon ved å bytte den med *A[n]*. Hvis vi nå ser bort fra node *n* i heapen, så kan vi enkelt deinkrementere A.heap-size.

```
HEAPSORT(A)
1   BUILD-MAX-HEAP(A)
2   for i = A.length - 1 downto 1
3       exchange A[0] with A[i]
4       A.heapsize -= 1
5       MAX-HEAPIFY(A,0)
```

Kjøretid:

- **Heapsort** prosedyren bruker `O(n lg n)` tid siden kallet på *Build-Max-Heap* tar O(*n*) tid og hvert av de *n - 1* allee til *Max-Heapify* tar O(lg *n*) tid.



> Bruker mindre lagringsplass enn Merge-Sort

Prioritetskø

Tar utgangspunkt i en max-heap for å implementere max-prioritetskøer. For å lage min-prioritetskøer er det bare å endre litt på prosedyrene.

Prioritetskø: En prioritetskø er en datastruktur å opprettholde et sett S med elementer, hver assosiert med en verdi kalt *key*. En *max-prioritetskø* støtter følgende operasjoner

- `INSERT(S, x)` setter inn et element *x* inn i settet *S* som er operasjonen `S = S ∪ {x}`
- `MAXIMUM(S)` returnerer elementet i *S* med størst *key*
- `EXTRACT-MAX(S)` fjerner og returnerer elementet i *S* med størst *key*
- `INCREASE-KEY(S,x,k)` øker verdien tl elementet *x* ´s *key* til den nye verdien *k*, som antas å være større enn *x* 's nåværende *nøkkelverdi*

> Alternativt støtter en min-prioritetskø operasjonene: `INSERT(S, x)`, `MINIMUM(S)`, `EXTRACT-MIN(S)` og `DECREASE-KEY(S,x,k)`.

```
HEAP-MAXIMUM(A)
1   return A[0]
```

- *Kjøretid:* `θ(1)`

```
HEAP-EXTRACT-MAX(A)
1   if A.heap-size < 1
2       error "heap underflow"
3   max = A[0]
4   A[0] = A[A.heapsize]
5   A.heapsize -= 1
6   MAX-HEAPIFY(A,0)
7   return max
```

- *Kjøretid:* `O*(lg n)` siden den gjør konstant arbeid på toppen av $O(\lg n)$ tiden for *Max-Heapify*

```
HEAP-INCREASE-KEY(A,i,key)
1   if key < A[i]
2       error "new key is smaller than current key"
3   A[i] = key
4   while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

- *Kjøretid:* `O(lg n)` siden veien fra noden oppdatert i linje 3 til roten har lengde $O(\lg n)$.

```
MAX-HEAP-INSERT(A, key)
1   A.heap-size += 1
2   A[A.heap-size] = -∞
3   HEAP-INCREASE-KEY(A, A.heap-size, key)
```

- *Kjøretid:* `O(lg n)` siden den kun gjør O(1) arbeid over *Heap-Increase-Key*.

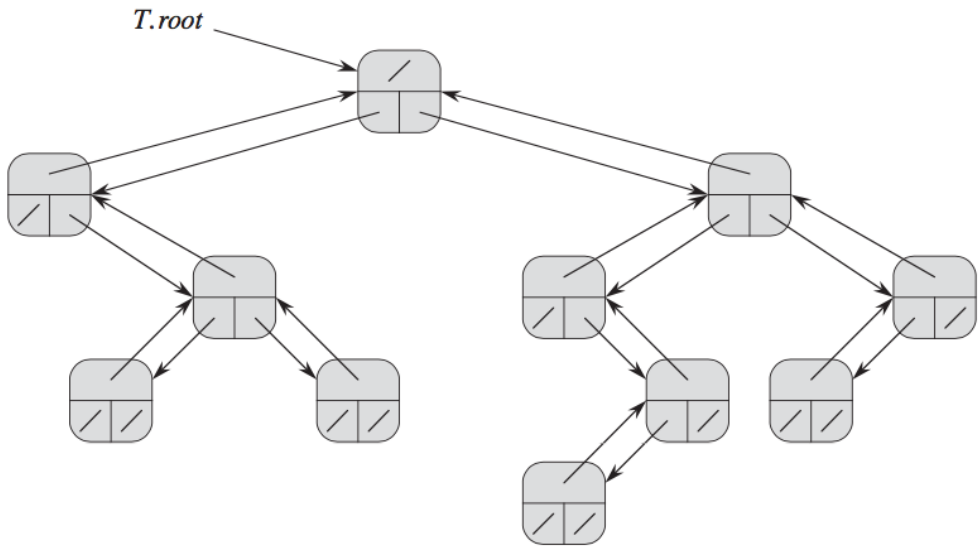
Oppsummering: En heap støtter enhver prioritetskø-operasjon på et sett av størrelse n på `O(lg n)` tid!

Rotfestede trær

Problem: Hvordan represetere rotfestede trær ved hjelp av lenket datastruktur.

Binære trær: Figuren under viser hvordan vi kan bruke attributtene *p*, *left* og *right* til å lagre pekere til forelder, venstre barn og høyre barn til hver node i binærtreet T.

- Dersom $x.p$ = NIL, da er x roten.
- Dersom x ikke har noen venstre barn, da er $x.left$ = NIL, og likt for høyre barn.
- Roten til treet T peker til å være attributten T.*root*. Dersom T.*root* = NIL, da er treet tomt.



Rotfestede trær med ubundet forgrening: Vi kan utvide representasjonen av et binært tre til en klasse av trær der antall barn til hver node er på det meste en konstant k - vi bytter *left* og *right* attributtene til $child_1$, $child_2$,.., $child_k$.

Vi kan bruke $O(n)$ minne for en vilkårlig n 'te rotfestet tre

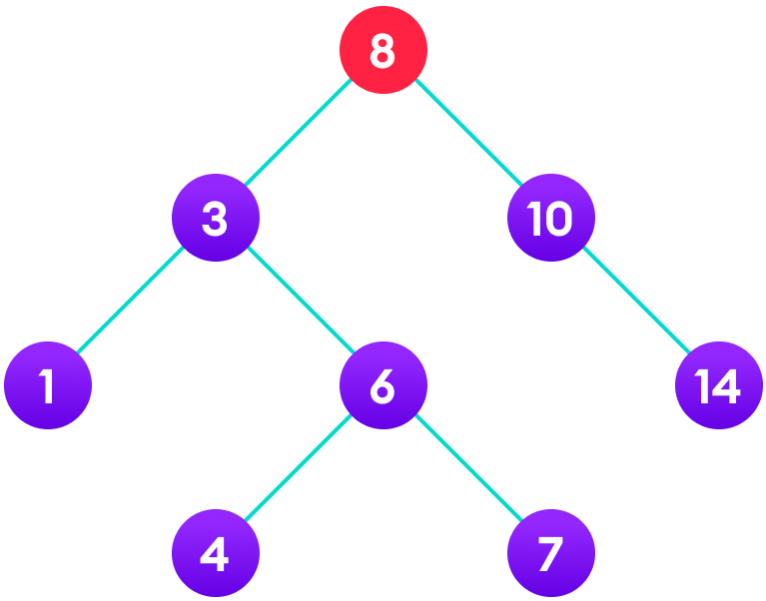
- For å finne oss frem i treet har hver node x kun to pekere:
  - $x.left\text{-}child$ peker til det barnet mest til venstre for x
  - $x.right\text{-}sibling$ peker til den søskenen rett til høyre for x

Binære søketrær

Denne søketre datastrukturen støtter mange dynamisk-sett operasjoner inkludert

Operasjoner	**Kjøretid**
Inorder-Tree-Walk	O(n)
Tree-Search	O(h)
Tree-Minimum	O(h)
Tree-Successor	O(h)
Tree-Insert	O(h)
Tree-Delete	O(h)

Binært søketre: Et binært søketre er organisert i et binærttre som vist under. Vi kan representere et slikt tre som en lenket datastruktur der hver node er et objekt. I tilegg til en *key* og et sett med data, har hver node attributtene *left*, *right*, *p* som peker til nodene korrespondere til sitt venstre barn, høyre barn og forelder, respektivt. Dersom et barn eller forelder mangler er den gjeldende attributtens verdi NIL. Rotnoden er den eneste noden i treet som har forelder lik NIL.




```
class Node:
    def __init__(self, key):
        self.key = key
        self.p = None
        self.left = None
        self.right = None
```

- *Binærsøketre-egenskapen*:
  - La *x* være en node i ett binært søketre:
    - Hvis *y* er en node i det venstre subtreet til *x*, da er `y.key ≤ x.key`.
    - Hvis *y* er en node i det høyre subtreet til *x*, da er `y.key ≥ x.key`.

Inorder tree walk: Simpel rekursiv algoritme som printer ut alle nøklene i treet i rekkefølge.

```
INORDER-TREE-WALK(x)
1   INORDER-TREE-WALK(x.left)
2   print x.key
3   INORDER-TREE-WALK(x.right)
```

> Det tar θ(*n*) tid å gå igjennom et *n*-node binært søketre.

Søking: Vi bruker følgende prosedyre for å søke etter en node med en gitt nøkkel i et binært søketre. Gitt en peker til roten og en nøkkel *k*, returnerer *Tree-Search* en peker til noden med key *k*, hvis den eksisterer, hvis ikke returnerer den NIL.

```
TREE-SEARCH(x, k)
1   if x == NIL or k == x.key
2       return x
3   if k < x.key
4       return TREE-SEARCH(x.left, k)
```

```
5   else
6       return TREE-SEARCH(x.right, k)
```

Kjøretid: $O(h) = O(\lg n)$

> Starter ved å søke ved roten, og traversere seg nedover, enten i venstre eller høyre subtre, til den finner den noden som den leter etter.

Vi kan også skrive om denne algoritmen til å være *iterativ* ved å bytte ut rekursjonen til en **while**-løkke.

```
ITERATIVE-TREE-SEARCH(x, k)
1   while x ≠ NIL and k ≠ x.key
2       if k < x.key
3           x = x.left
4       else x = x.right
5   return x
```

> På de fleste PC-er er en iterativ versjon mer effektiv

Minimum og maximum:

- *Binærsøketre-egenskapen* garanterer oss at *Tree-Minimum* og *Tree-Maximum* er korrekte.

```
TREE-MINIMUM(x)
1   while x.left ≠ NIL
2       x = x.left
3   return x
```

> For å finne minimum traverserer man seg bare nedover mot venstre i treet til det ikke lenger går.

```
TREE-MAXIMUM(x)
1   while x.right ≠ NIL
2       x = x.right
3   return x
```

> For å finne maksimum traverserer man seg bare nedover mot høyre i treet til det ikke lenger går.

Kjøretid: Begge prosedyrene kjører på $O(h) = O(\lg n)$ tid

Etterkommer og forgjenger

Etterkommer (*eng. Successor*): Etterkommeren til en gitt node *x* er den noden med minst nøkkelverdi, større enn *x.key*

Forgjenger: (*eng. Predecessor*): Forgjengeren til en gitt node *x* er den noden med størst nøkkelverdi, mindre enn *x.key*

Gitt en node i et binært søketre, trenger vi noenganger å finne etterkommeren dens i sortert rekkefølge bestemt av en *inorder tree walk*. Dersom alle nøkler er distinkte er etterkommeren til en node *x* den noden med minst.

```
TREE-SUCCESSOR(x)
1   if x.right ≠ NIL
2       return TREE-MINIMUM(x.right)
3   y = x.p
4   while y ≠ NIL and x == y.right
5       x = y
6       y = y.p
7   return y
```

1. Dersom høyre subtre til node *x* er ikke-tomt, da er etterkommeren til *x* noden helt til venstre i *x* 's høyre subtre. Etterkommeren finner vi med *Tree-Minimum* på linje 2.
2. Dersom høyre subtre til node *x* er tomt, og x har en forgjenger *y*, da er **etterkommeren** det første elementet som er større enn *x* som algoritmen finner.

Kjøretid: `O(h) = O(lg n)`

```
TREE-PREDECESSOR(x)
1   if x.left ≠ NIL
2       return TREE-MAXIMUM(x.left)
3   y = x.p
4   while y ≠ NIL and x == y.left
5       x = y
6       y = y.p
7   return y
```

Kjøretid: `O(h) = O(lg n)`

Innsetting og sletting

Sletting

For å sette inn en ny verdi *v* inn i et binært søketre *T*, bruker vi prosedyren *Tree-Insert*. Prosedyren tar en node *z* der `z.key = v`, `z.left = NIL` og `z.right = NIL`. Den modifiserer *T* og noen av attributtene til *z* slik blir satt inn i treet på en passende posisjon.

```
TREE-INSERT(T,z)
1    y = NIL
2    x = T.root
3    while x ≠ NIL
```

```
4         y = x
5         if z.key < x.key
6             x = x.left
7         else x = x.right
8     z.p = y
9     if y == NIL
10        T.root = z     //Tree was empty
11    elif z.key < y.key
12        y.left = z
13    else y.right = z
```

- **Kjøretid:** Som alle andre primitive operasjoner på søketrær bruker prosedyren `O(h) = O(lg n)` tid på en tre med høyde *h*.

Sletting:

Strategien som brukes for å slette en node *z* har tre generelle tilfeller, men som kan være litt kompliserte.

- Dersom *z* ikke har noen barn, kan vi simpelten fjerne noden ved å modifisere forelderen ved å erstatte *z* med NIL som dens barn: `z.p.child = NIL`
- Dersom *z* kun har ett barn kan vi bare la barnet overta *z* 's posisjon i treet, ved å modifisere *z* 's forelder til å erstatte *z* med *z* 's barn, og endre *z* 's barn forelder-attributt.
- Dersom *z* har to barn, da finner vi *z* 's etterkommer *y* - som må være i *z* 's høyre subtre. Resten av *z* 's høyre subtre blir *y* 's nye høyre subtre, og *z* 's venstre subtre blir *y* 's nye venstre subtre.
    - Dette tilfellet er litt mer komplekst enn de andre, og det avhenger av om *y* er *z* 's høyre barn.

For å kunne bevege på subtrær rundt in i et binært søketre, definerer vi en subrutine *Transplant*, som erstatter et subtre som et barn til sin forelder med et annet subtre. Når *Transplant* ersstatter subtreet med rot *u* med subtreet med rot *v*, bytter de foreldre.

```
def Transplant(T, u, v):
    if u.p == None:
        T.root = v
    elif u == u.p.left
        u.p.left = v
    else:
        u.p.right = v
    if v ≠ None:
        v.p = u.p
```

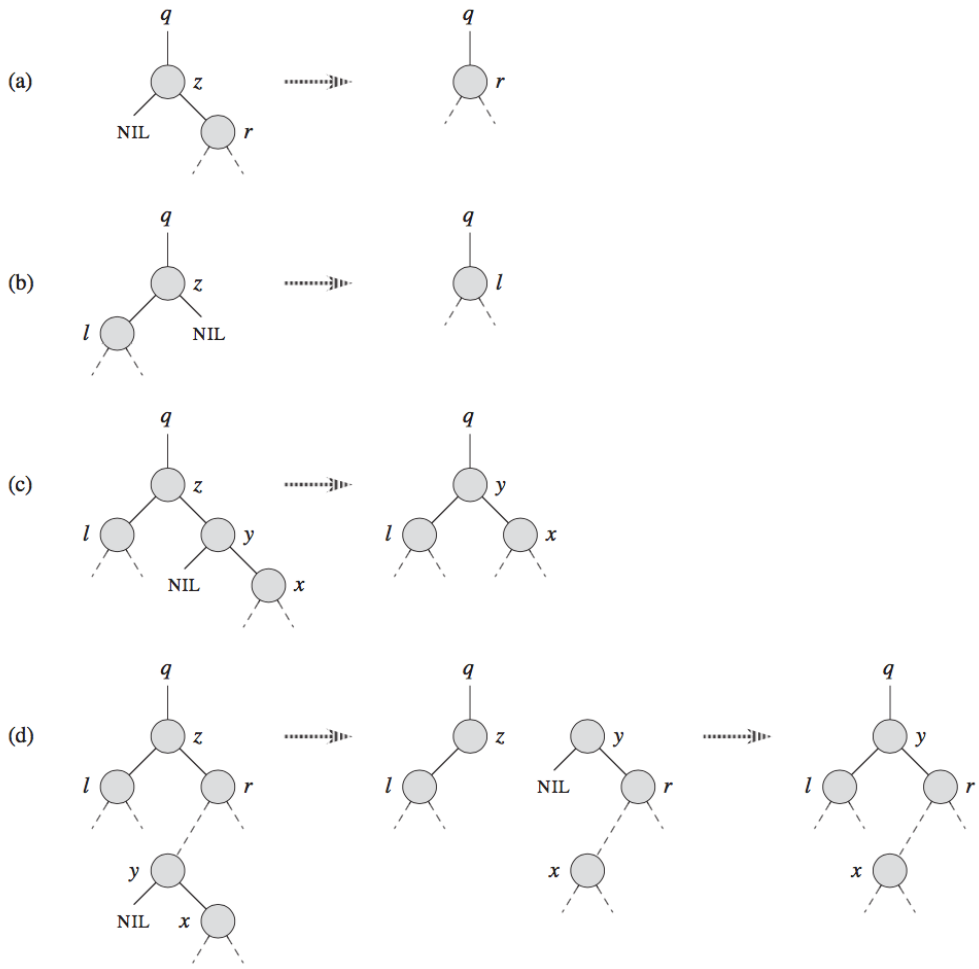
> *Transplant* oppdaterer ikke *v.left* og *v.right*, om det blir gjort eller ikke er opp til den som kaller på prosedyren

```
def Tree-Delete(T, z):
    if z.left == None:
```

```
        Transplant(T, z, z.right)

    elif z.right == None:
        Transplant(T, z, z.left)

    else:
        y = Tree-Minimum(z.right)
        if y.p ≠ z:
            Transplat(T, y, y.right)
            y.right = z.right

        Transplant(T, z, y)
        y.left = z.left
        y.left.p = y
```

Prosedyren for å slette en gitt node *z* tar inn en pekere til *T* og *z*.

- Dersom *z* ikke har noen venstre barn (*del (a) av figuren under*) da erstatter vi *z* med dets høyre barn som kan være NIL. Når *z* 's høyre barn er NIL løser vi dette problemet som situasjonen der *z* ikke har noen barn. Når *z* 's høyre barn er ikke-NIL, har vi en situasjon er *z* kun har ett barn, nemlig dens høyre.
- Dersom *z* kun har ett barn, som er dens venste barn (*del (b)*), da erstatter vi *z* med sitt venste barn.
- Hvis ikke har *z* både ett høyre og en venstre barn. Da finner vi *z* 's etterkommer *y*, som ligger i *z* 's subtre, og har ingen venstre barn. Vi ønsker å klippe *y* ut av sin nåværende posisjon og erstatte *z* i treet.
    - Dersom *y* er *z* 's høyre barn (*del (c)*), da erstatter vi *z* med *y*, og lar *y* 's høyre barn være i fred.
    - Hvis ikke ligger *y* i *z* 's høyre subtre, men er ikke dets høyre barn (*del (d)*). Dersom dette er tilfellet erstatter vi *y* med sitt høyre barn, og erstatter *z* med *y*.



Kjøretid: Hver linje i *Tree-Delete*, inkludert kallet på *Transplant*, tar konstant tid, untatt kallet på *Tree-Minimum*. Dermed har *Tree-Delete* en kjøretid på `O(h)`, på et tree med høyde *h*

Forventet høyde på binomisk søketre

Ved hjelp av et bevis i Cormen på side 300, kan man se at forventet høyde *h* på et tilfeldig bygd binomisk søketre med n distiskte elementer er *O*(lg *n*). Dvs.

```
O(h) = O(lg n)
```

> Det finnes søketrær som har garantert høyde h = θ(lg *n*) - et eksempel på et slikt tre er red-black tree.

Dynamisk programmering

Dynamisk programmering, som splitt og hersk, løser problemer ved å kombinerer løsninger på delproblemer. Vi bruker dynamisk programmering når delproblemene *overlapper*, og det er når delproblemer deler SUB-delproblemer. I denne konteksten gjør splitt og hersk mer arbeid enn nødvendig, og løser samme delproblem flere ganger. En dynamisk programmerings algoritme løser hvert deldelproblem kun en gang, og **lagrer** resultatet for at den skal slippe å regne gjennom samme problem flere ganger.

Vi bruker gjerne dynamisk programmering ved *optimaliseringsproblemer*. Slike problemer kan ha mange mulige løsninger, og hver løsning har en verdi og vi ønsker å finner den løsningen med optimal verdi. Det kaller vi en optimal løsning på problemet.

- Det er to måter å gjøre dynamisk programmering på
    - Top Down: Memoizering med en rekursiv tilnærming
    - Bottom Up: finn optimal løsning å fyll inn løsnings-tabell

Når vi skriver en dynamisk programmerings algoritme følger vi følgende steg:

1. Finn ut om problemet kan løses med DP (ofte det vanskeligste).
2. Identifiser variabler.
3. Identifiser relasjoner mellom variablene.
4. Finn en optimal løsning, typisk på en *bottom-up* måte.
5. Bruk memoisering for å optimalisere løsningen.

Optimal delstruktur:

"Det første steget i å løse et optimaliserings problem med dynamisk programmering er å karakterisere strukturen til en optimal løsning".

Optimal delstruktur: En optimal løsning for problemet finnes i den optimale løsningen for del-problemene.

Finne optimale delstrukturer:

1. Vise at en løsning til et problem består av et valg, slik som å velge et start-kutt i en stav. Å ta dette valget gir en eller flere delproblemer å løse.
2. Gitt et problem, får man gitt et valg som leder til en optimal løsning. Ikke tenk på hvordan man kan ta dette valget, bare anta det man har fått.
3. Gitt et valg, må man velge hvilke delproblemer som følger og hvordan man best karakteriserer "rommet" av delproblemene.
4. Viser at løsningen til delproblemene brukt i en optimal løsning til en problem også selv må være optimale.

Overlappende delproblemer

Det andre som må være til stede for å kunne bruke dynamisk progammering er at "rommet" til delprolemene må være "lite" på den måten at en rekursiv algoritme av problemet løser de samme delproblemene igjen og igjen - i steden for å alltid lage nye delproblemer.

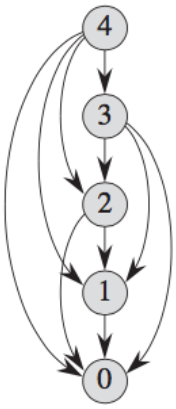
Når en rekursiv algoritme møter på samme problem gjentatte gngaer, sier vi at optimaliseringsproblemet har **overlappende problemer**.

> Typisk er antallet av distinkte delproblemer er polinomisk i input størrelsen.

Delproblemgraf

Når vi tenker på dynamisk programmerings problem, bør vi forstå settet med delproblemer som involvert, og hvordan de avhenger av hverandre.

Delproblemgrafen for et problem gjengir nettopp dene informasjonen. Det er en rettet graf, med en node for hvert distinkt delproblem. Delproblemgrafen har en rettet kant fra noden for delproblemet *x* til noden for delproblemet *y*, dersom en optimal løsning for *x* avhenger av en optimal løsning av delproblemet *y*.



Størrelsen på en delproblemgraf `G = (V, E)` kan hjelpe oss til å forstå **kjøretiden** til en algoritme med dynamisk programmering. Siden vi må løse hvert delproblem kun en gang, er kjøretiden summen av antall ganger vi må løse et delproblem.

- Typisk er kjøretiden for å finne en løsning på et delproblem proposjonal med antall *utgående* kanter i delproblemgrafen

Stavkutting

Problem: Gitt en stav med lengde n tommer og en liste med priser p_i for $i = 1,2,...,n$ for å finne maximum inntekt r_n ved å kutte staven opp i deler og selge de.

Vi kan kutte en stav på lengde n på 2^{n-1} forskjellige måter.

Dersom en optimal løsning kutter opp staven i k deler, for en $1 \leq k \leq n$, da er en optimal dekomposisjon $n = i_1 + i_2 + ... + i_k$ og gir maximum avkastning på $r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}$.

Rekursiv *top-down* implementasjon (Ikke dynamisk programmering):

Input: En liste $p[1...n]$ av priser og et tall n.

Output: Maksimum avkastning

```
def cutRod(price, n):
    if(n <= 0): # base case, hvis staven har lengde 0
        return 0 # så er prisen 0
    max_val = -1
```

```
    for i in range(0, n): # iterer gjennom lengen av staven
        # finn max mellom max_val og et rekursivt kall som vil returnere
alle kombinasjoner av staver.
        max_val = max(max_val, price[i] + cutRod(price, n - i - 1))
    return max_val
```

> Kjøretiden blir her $O(2^n)$, og er derfor en **ekstremt dårlig algoritme**.

***Top-down* implementasjon med memoisering (!):**

Memoisering: Lagre en verdi som vi kan se på igjen senere

```
def topDownRodCutRod(price, n, result = None):
    if result == None: # initialiserer løsnings-tabellen
        result = dict()
        result[0] = 0

    if(n in result and result[n] >= 0): # her brukes del-løsningen i
løsnings-tabellen
        return result[n]
    max_val = -1

    for i in range(0, n):
        max_val = max(max_val, price[i] + topDownRodCutting(price, n-i-1,
result)) # rekursivt kall for å finne optimal løsning

    result[n] = max_val # legg til optimal løsning for hvert rekursivt kall
    return result[n] # Det siste kallet i call-stacken vil returnere
optimal løsning for den endlige n
```

Hovedprosedyren i *topDownRodCutting* er å initialisere en hjelpeliste result, som skjekker om vi allerede vet verdien vi ser etter. Hvis ikke regner den ut den ønskede verdien q på den vanlige måten, lagrer den i *result[n]* og returnerer den.

Iterasjoner: Algoritmen kjører **for**-løkken *n* ganger og gir en aritmetisk rekke med $\theta(n^2)$ iterasjoner

***Bottom-up* implementasjon med memoisering:**

> *Enda enklere enn top-down implementasjonen*

Bottoms-up løsning:

```
def bottomsUpCutRod(price, n):
    result = [0 for x in range(n+1)] # Løsnings-tabell, 0..n+1
    result[0] = 0 # Verdien til en stav med lenge 0 = 0

    for i in range(1, n+1): # For hver lenge av staven
        max_val = -1
        for j in range(i): # Finn den optimale løsningen
             max_val = max(max_val, price[j] + result[i-j-1]) # her brukes
del-løsningen i løsnings-tabellen
        result[i] = max_val # Lagre del-løsning til løsnings-tabellen

    return result[n] # Returner optimal løsning for en stav av lenge n.
```

Iterasjoner: Algoritmen kjører dobbel **for**-løkke og gir $\theta(n^2)$ kjøretid.

Rekonstruere en løsning fra lagrede beslutninger

Ser igjen på *stavkutting-problemet*. De tidligere løsningene av stavkuttings-problemet har kun returnert verdien av de optimale løsningen, men ikke den faktiske løsningen: en liste med stykker av staven. Vi kan utvide den dynamiske programmeringen til å lagre den **optimale verdien** for hvert subproblem men også et **valg** som ledet den til den optimale verdien.

```
def extendedBottomsUpCutRod(price, n):
    result = [0 for x in range(n+1)] # Løsnings-tabell, 0..n+1
    result[0] = 0 # Verdien til en stav med lenge 0 = 0
    s = dict()

    for i in range(1, n+1): # For hver lenge av staven
        max_val = -1
        for j in range(i): # Finn den optimale løsningen
            if max_val < price[j] + result[i-j-1]:
                max_val = price[j] + result[i-j-1] # her brukes del-
løsningen i løsnings-tabellen
                s[i] = j+1
        result[i] = max_val # Lagre del-løsning til løsnings-tabellen

    return result, s
```

> s returnerer en dictionary med det høyeste kuttet brukt for alle kutt `0..n`

LCS - Longest Common SubSequence

Gitt to sekvenser X og Y, sier vi at sekvensen Z er en felles subsekvens til X og Y dersom Z er en subsekvens i både X og Y.

F.eks. dersom $X = \langle A, B, C, B, D, A, B \rangle$ og $Y = \langle B, D, C, A, B, A \rangle$, er sekvens $\langle B, C, A \rangle$ en felles subsekvens til X og Y. Sekvensen $\langle B, C, A \rangle$ er derimot ikke den *lengste* felles subsekvensen (*LCS*) til X og Y. Da det finnes en

lengre subsekvens som f.eks. ⟨B, D, A, B⟩.

Lengste subsekvens-problemet: Vi blir gitt to sekvenser, X og Y. Vi ønsker å finne den aller lengste felles subsekvensen til *X* og *Y*. Dette kan løses med *dynamisk programmering*:

En rekursiv løsning: Vi må undersøke enten en eller to delproblemer når vi skal finne *LCS* til $X = \langle x_1, x_2,...,x_m \rangle$ og $Y = \langle y_1, y_2*,...,y_n \rangle$.

- Dersom $x_m = y_n$ må vi finne LCS til X_{m-1} og Y_{n-1}.
- Den optimale substrukturen til LCS-problemet gir da den rekursive funksjonen:

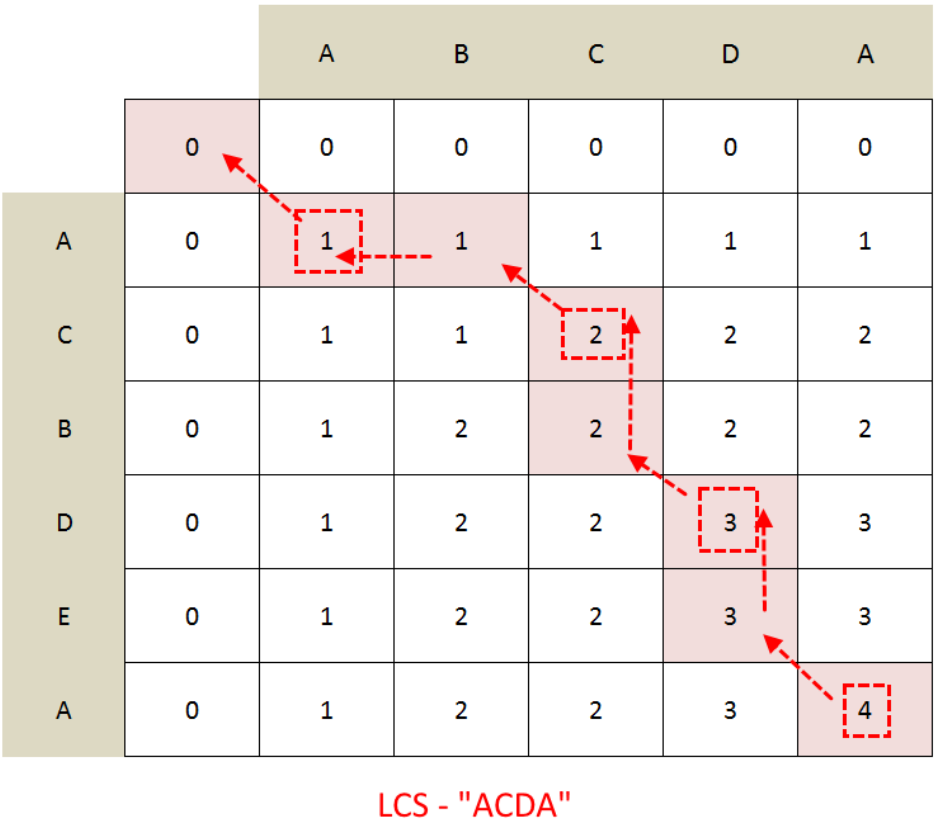
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

2. **Regne ut lengden på en LCS:**
   - Basert på ligning over kan vi lett skrive en eksponentiell rekursiv algoritme for å regne ut lengden til en LCS til to sekvenser. Til tross for dette kan vi bruke *dynamisk programmering* til å løse problemet.
   - Prosedyren *LCS-Length* tar inn to sekvenser *X* og *Y* som input, og lagrer verdiene i en matrise *c* [0..*m*, 0..*n*], og fyller ut plassene i orden (dvs. fylle rad 1, så rad 2 osv).
   - Prosedyren lager også matrise *b* [1..*m*,1..*n*] for å hjelpe oss med å konstruere en optimal løsning. Intuitivt peker *b*[*i*][*j*] til en element korresponderende til en optimal delproblem-løsning av *c*[*i*, *j*].
   - Prosedyren returnerer matrisene *b* og *c* og **c[*m*, *n*]** inneholder lengden til en LCS til *X* og *Y*

```
def LCS-Length(X, Y):
    m = len(X)
    n = len(Y)
    b = [[0]*n for row in range(m)]
    c = [[0]*(n+1) for row in range(m+1)]

     for i in range(1,m+1):
         for j in range(1,n+1)
             if X[i-1] == Y[j-1]:
                 c[i][j] = c[i-1][j-1] + 1
                 b[i-1][j-1] = '↖'

             elif c[i-1][j] ≥ c[i][j-1]:
                 c[i][j] = c[i-1][j]
                 b[i-1][j-1] = '↑'

             else:
                 c[i][j] = c[i][j-1]
                 b[i-1][j-1] = '←'

     return c, b
```

> Kjøretiden på denne prosedyren er **θ(*mn*)**, siden hvert matriselement tar θ(1) å regne ut.

Dette kan også visualiseres i en matrise som vist under:



LCS - "ACDA"

1. **Konstruere en LCS:**
    - Tabellen *b* returnert av *LCS-Length* lar oss lett konstruere en LCS til sekvensene *X* og *Y*. VI begynner simpelten på *b* [*m*][*n*] og følger pilene. For hver gang vi støter på en '↖' betyr det at $x_i = y_j$ er et element av LCS-en som *LCS-Length* har funnet. Med denne metoden finner vi elementene i LCS i baklengs rekkefølge. Følgende prosedyre printer ut LCS til *X* og *Y* i riktig rekkefølge:

```
def Print-LCS(b, X, i, j):
    if i == -1 or j == -1:
        return

    if b[i][j] == '↖':
        Print-LCS(b, X, i-1, j-1)
        print(X[i])

    elif b[i][j] == '↑':
        Print-LCS(b, X, i-1, j)

    else:
        Print-LCS(b, X, i, j-1)
```

> Denne prosdyren bruker ***O(m + n)*** tid, siden den dekrementerer minst en av *i* og *j* for hvert rekursive kall.

Kjøretid

Kjøretiden til en algoritme i *dynamisk programmering* avhenger av et produkt av to faktorer: **Antall delproblemer** og hvor mange **valg** vi har i hvert delproblem.

- I stavkuttingen hadde vi $\theta(n)$ delproblemer, og max n valg i hvert delproblem, altså fikk vi kjøretid $O(n^2)$

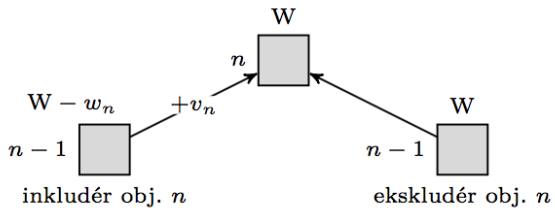
0-1 Knapsack

Det såkalte *ryggsekkproblemet* kommer i flere varianter. Den *fraksjonelle* varianten er letter å løse: Man tar bare med seg så mye som mulig av den dyreste gjenstanden, og fortsetter nedover på lista, sortert etter kilopris. I 0-1-varianten, derimot, blir ting litt vanskeligere - her må man ta med en hel gjenstand eller la den ligge.

Akkurat som i f.eks. Floyd-Warshall baserer dekomponeringen seg på et *ja-nei-spørsmål*, i dette tilfellet; «Skal vi ta med gjenstand *i* ?». For hver av de to mulighetene sitter vi igjen med et delproblem som vi løser rekursivt. Som vanlig tenker vi oss at dette er siste trinn og antar at vi har gjentstander 1,...,*i* tilgjengelige. Da har vi to muligheter:

1. **Ja**, vi tar med gjenstand *i*. Vi løser så problemet for gjenstander 1,...,*i-1* men der kapasiteten er redusert med w_i. Vi legger så til v_i til slutt.
2. **Nei**, vi tar ikke med gjenstand *i*. Vi løser så problemet for gjenstander 1,..,*i-1*, men kan fortsatt bruke hele kapasiteten. Til gjengjeld får vi ikke legge til v_i til slutt.

Situasjonen er illustrert i figuren under, der hver rute representerer en delløsning (en celle i løsningstabellen, f.eks) og pilene er avhengigheter, som vanlig. Vi kan sette opp en rekursiv løsning slik:




```
KNAPSACK(n, W)
1   if n == 0
2       return 0
3   x = KNAPSACK(n-1, W)
4   if W < w_n
5       return x
6   else y = KNAPSACK(n-1, W - w_n) + v_n
7       return max(x, y)
```

> Denne prosedyren vil naturligvis ha eksponentiell kjøretid.

Dette er ikke polynomisk!

0-1-knapsack er et såkalt NPC problem, og det derfor ingen som har funnet noen polynomsk løsning på det.

Kjøretiden til *Knapsack* er *θ(nW)*, siden det er *nW* delproblemer og vi utfører en konstant mengde arbeid per delproblem. I forbindelse med NP-kompletthet holder vi oss til *antall bits* i input, i en rimelig encoding. Størrelsen blir da *θ(n + lg W)*, siden vi bare trenger θ(lg *W*) bits for å lagre parameteren W.

Poenget er altså at *W* vokser eksponentielt som funksjon av lg *W*, og kjøretiden er, teknisk sett, eksponentiell. Vi lar *m* være antall bits i *W*, og kan skrive kjøretiden som: $T(n, m) = θ(n2^m)$

Da er det tydelig at dette ikke er en polynomsik kjøretid. Kjøretider som er polynomisk hvis vi lar et tall fra input være med som parameter til kjøretiden (slik som θ(*nW*), der W er et tall fra input, og ikke direkte en del av problemstørrelsen) kaller vi *pseudopolynomiske*. (ofte lureoppgave på eksamen)

Grådige algoritmer

En **grådig algoritme** tar alltid et valg som ser best ut der og da. Som betyr, den tar en *lokalt optimalt valg* i håp om at det vil lede til den *globale optimale løsningen*.

For å kunne benytte en grådig algoritme må problemet ha:

- Optimal sub-struktur
- Grådighetsegenskapen

Husk: Et problem har **optimal delstruktur** dersom en optimal løsning for problemet finnes i den optimale løsningen for del-problemene. Vi kan bevise optimal substruktur ved å bruke induksjon på delproblemene til å vise at det å ta det grådige valget i hvert steg produserer en optimal løsning.

Grådighetsegenskapen:

Et problem har denne egenskapen dersom man kan ende opp med en optimal løsning ved å velge ut det som ser best ut i øyeblikket.

Eksempel: Hvis du skal betale for en vare med mynter, vet du at du alltid vil ende opp med å betale minst antall mynter dersom du heletiden legger til den mynten med høyest verdi i den endlige løsningen. Man må altså ikke regne ut alle kombinasjoner av mynter.

Det er her forskjellen mellom grådige algoritmer og *dynamisk programmering* ligger. I dynamisk programmering tar vi valg på hvert steg, men som vanligvis avhenger av løsningen på delproblemene. Og i motsetning til dynamisk programmering tar grådige algoritmer sitt første valg, før den løser noen av delproblemene.

Elementer ved den grådige strategien

Vi designer gråde algoritmer i henhold til følgende punkter:

1. Gitt et optimaliseringsproblem skal vi ta et valg og står igjen med ett subproblem å løse.
2. Vis at det alltid er en optimal løsning på det originale problemet som tar grådige valget, slik at det grådige valget alltid er trygt.
3. Demonstrer den optimale substrukturen, ved å vise at dersom vi tar det grådige valget, gjenstår det et delproblem som har den egenskapen at hvis vi kombinerer en optimal løsning på subproblemet og det grådige valget vi tok, kommer vi frem til en optimal løsning på det originale problemet.

Aktivitetutvelgelse

La oss anta et set $S = \{a_1, a_2,...,a_n\}$ av n foreslåtte aktiviteter som ønsker å - for eksempel bruke en gymhall, som kun kan brukes til en aktivitet av gangen. Hver aktivitet a_i har en **start-tid** s_i og en **slutt-tid** f_i, hvor $0 \leq s_i \leq f_i < \infty$. Dersom en aktivitet a_i er valgt i intervallet $[s_i, f_i)$, er aktivetetene a_i og a_j **kompatible** dersom intervallene $[s_i, f_i)$ og $[s_j, f_j)$ ikke overlapper.

I aktivitetutvalg-problemet ønsker vi å velge max-størrelse subset av kompatible aktiviteter. Vi antar at aktivitetene er sortert i stigende rekkefølge etter *slutt-tid*:

Det grådige valget:

Problemet har grådig valg egenskapen fordi man kan velge en aktivitet og legge det til i den optimale løsningen uten å først måtte løse alle delproblemene. På denne måten sparer vi oss for mye arbeid

Vi må velge den aktiviteten i S med tidligst *slutt-tid*, siden det lar det være mer tid igjen til de andre aktivitetene.

Dersom vi tar det grådige valget, har vi kun ett delproblem å løse: Finne en aktivitet som starter etter a_1 slutter. Vi må finne en aktivitet som slutter etter aktivitet a_1

En rekursiv grådig algortime

Prosedyren *Recursive-Activity-Selector* tar aktivitetene $A[a_1,..a_n]$ og start- og slutt-tiden til aktivitetene representert som listene s og f, indeksen k som definerer subproblemet S_k og størrelsen n til det originale problemet. Antar A som global variabel med aktiviteter og henter derfra.

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1    m = k + 1
2    while m ≤ n and s[m] < f[k]
3        m = m + 1
4    if m ≤ n
5        return {A[m] ∪ RECURSIVE-ACTIVITY-SELECTOR(s,f,m,n)}
6    else return ∅
```

Vi kan også konvertere den rekursive prosedyren til en iterativ en. Prosedyren *Greedy-Activity-Selector* er en iterativ versjon av prosedyren over. Den antar forøvrig at input-aktivitetene er sortert i stigende rekkefølge etter slutt-tid. Antar fortsatt A som global variabel med aktiviteter og henter derfra.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1    n = s.length
2    res = [A[1]]
3    k = 1
4    for m = 2 to n
5        if s[m] ≥ f[k]
6            res += A[m]
7            k = m
8    return res
```

Kjøretid: Begge algoritmen planlegger n aktiviteter på $\theta(n)$ tid.

Fractional knapsack problem

Samme oppsett som i *0-1-knapsack*, men man kan ta med seg deler (*fractions*) av elementer (*items*), istedet for å måtte ta et binært (0-1) valg for hvert element. Begge ryggsekkproblemene utviser optimal substruktur. Vi kal løse det fraksjonelle ryggsekkproblemet med en grådig strategi.

For å løse det fraksjonelle problemet, må vi regne ut kiloprisen v_i / w_i for hvert element. Ved å følge den gråde strategien tar vi så mye som mulig av det elementet med høyest kilopris, og deretter så mye som mulig av det nest dyreste elementet, til ryggsekken når sin vektgrense W.

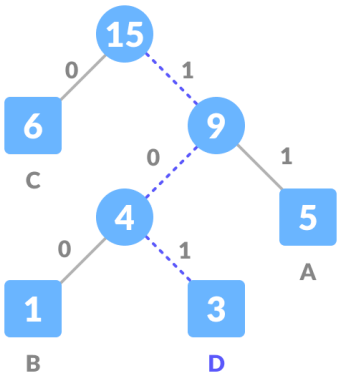
Kjøretiden: Siden algorimetn må sortere elementene med tanke på kilopris, kjører den grådige algoritmen på $O(n \lg n)$ tid.

Huffmann-koder

Huffmann-koder komprimerer data veldig effektivt, og gir besparelser på 20-90%. Vi ser her på "prefix-frie koder". Når vi skal encode for binær kode, skiller vi bare mellom kodeordene som representerer karakterene i fien.

Når en skal skrive et binært tre som decoder/encoder en tekst, lager man et binærtre der bladene er gitte tegn, og kantene er nummerert med 0 eller 1. Der venstre kant er 0 og høyre kant er 1. Så når man leser fra en krypert kode, så betyr 0: Gå til venstre barn, og 1: Gå til høyre barn.

eksempel: vi skrive *abc* fra grafen under som 11|100|0 = 111000



Størrelsen for eksemepelet over ville uten huffman-enconding gitt en størrelse på: `15tegn * 8bit = 120bits`

Beregning av størrele etter utført huffman-enconding kan man se i grafen under:

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32bits	15 bits		28 bits

Dette gir en total størrelse på: 32 + 15 + 28 = 75

> En optimal kode for en fil er alltid representert som en fullt binærtre.

Antall bit for å encode en fil er

$\displaystyle B(T) = \sum_{c \in C} c.\text{freq} \cdot d_T(c)$

> der *c.freq* er frekvensen til ett tegn og $d_T(c)$ er lengden på kodeordet for *c*.

Konstruere Huffmann-koder

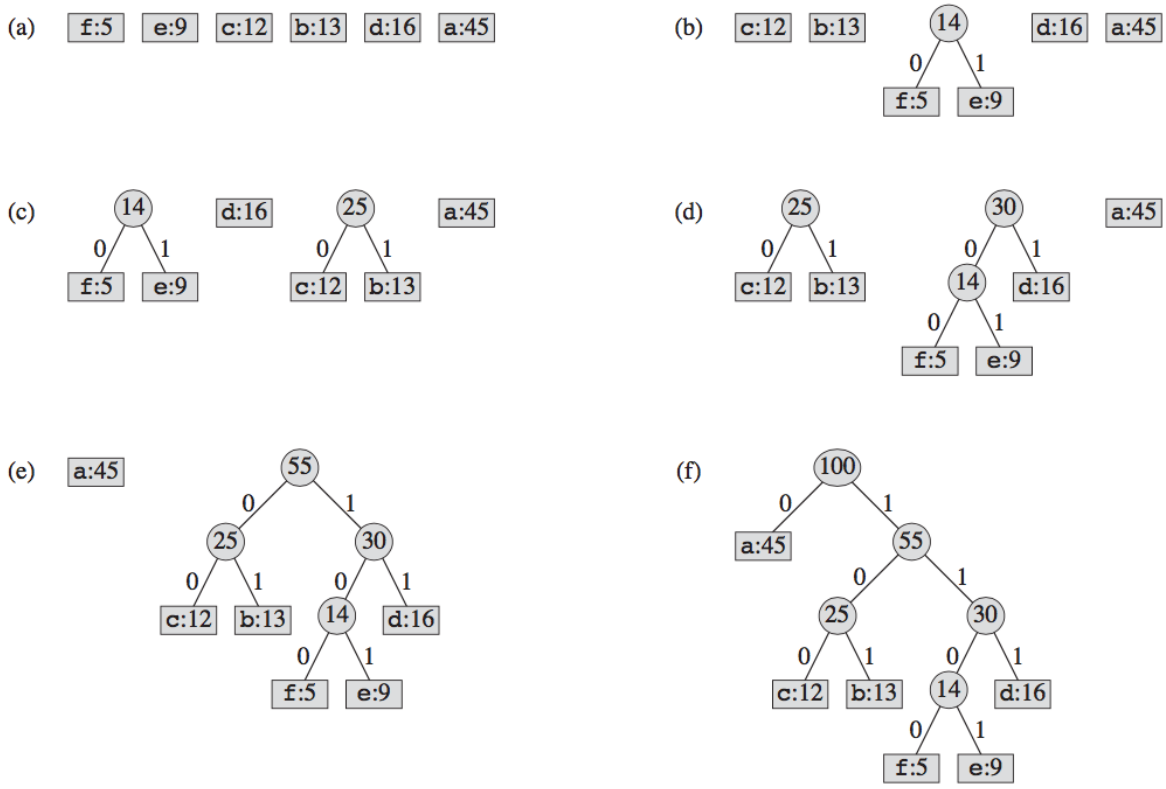
Man starter med et sett *C* med *n* tegn, og at hvert tegn c ∈ C har en attributt *c.freq* som betegner dens frekvens. Algoritmen *Huffman* bygger et tree *T* korresponderende til den optimale koden på en *bottom-up* måte.

1. Algoritmen legger alle tegnene i en kø
2. Deretter fjerner den de to nodene/tegnene *x* og *y* med minst frekvens fra køen, og lager en ny node *z* med *x* og *y* som barn, og *z.freq* = *x.freq* + *y.freq*, og legger *z* til køen.
3. Til slutt er det kun en rot igjen i køen, og dette er roten til Huffmann-treet, som returneres.

> På denne måten vil elementene med lavest frekvens få lengst vei i treet.

```
HUFFMAN(C)
1    n = |C|
2    Q = C
3    for i = 1 to n - 1
4        allocate a new node z
5        z.left = x = EXTRACT-MIN(Q)
6        z.right = y = EXTRACT-MIN (Q)
7        z.freq = x.freq + y.freq
8        INSERT(Q, z)
9    return EXTRACT-MIN(Q)      // returnerer roten i treet
```

Kjøretid: `O(n lg n)` med binær-heap



Bevise korrektheten til Huffmans algoritme: For å vise at den er korrekt må vi vise at den utviser *grådighetsegenskapen* og har en *optimal substruktur*.

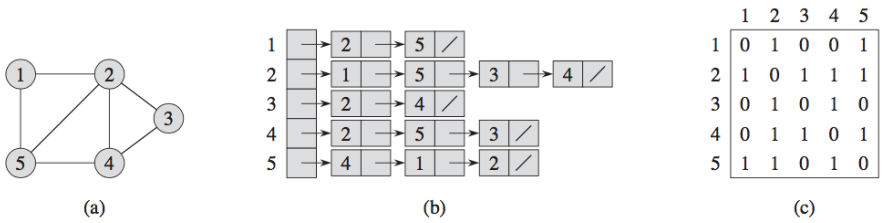
Traversering av grafer

Representasjon av grafer

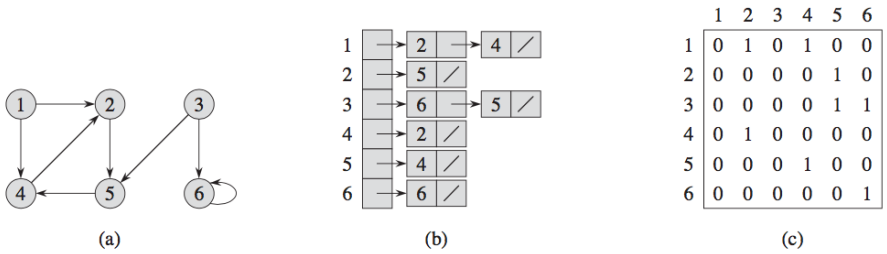
Vi kan velge mellom to standard måter å representere en graf `G = (V, E)`; som ett sett nabolister - eller som en nabomatrise. Begge måtene kan brukes til rettede og urettede grafer. Naboliste representasjonen

gir en mer kompakt måte å representere en **spredt** (*eng. sparse*) graf - der $|E|$ **er mye mindre enn** $| V |^2$. I de fleste algoritmene i boken antar vi at input-grafen er representert på en nabo-liste form. Vi kan også bruke en nabomatrise når vi har en **tett** (*eng. dense*) graf - der $|E|$ **er nær** $|V|^2$, eller når vi kjapt trenger å finne ut om det er en kant som binder to gitte noder.

Urettede grafer: a) graf b) naboliste c) matrise



Rettede grafer: a) graf b) naboliste c) matrise



Naboliste: En liste med $|V|$ elementer består aven liste med $deg(V_i)$ elementer

> Krever θ(V + E) lagringsplass.

Nabomatrise: En $|V| \times |V|$ matrise $A = a_{ij}$.

> Krever θ(V^2) lagringsplass.

Bredde-først søk - BFS

Bredde-først søk er en av de enkleste algoritmene for å søke i en graf. Gitt en graf `G = (V, E)` og en gitt **kilde** *s*, kan bredde-først-søk systematsik utforske kantene i G, for å finne hver node som kan nås fra *s*. Den regner ut avstanden (minste antall kanter) fra *s* til hver node node man kan nå. Den produserer også ett *bredde-først tre*, med roten *s* som inneholder alle noder som kan nås.

For hver node *v* som kan nås fra *s*, den enkle stien i bredde-først treet fra *s* til *v* korresponderer til den "korteste veien" fra *s* til *v* i G. Algoritmen fungerer på både rettede og urettede grafer. Algoritmen finner alle noder med avstand *k* fra *s*, før den finner noen noder med avstand *k* + 1.

Algoritmen konstruerer et bredde-først tre, først med bare sin rot *s*. Den kan først utforske en ny node *v* etter at den har scannet nabolisten til en allerede funnet node *u*. Deretter vil noden *v* og kanten (*u*, *v*) bli lagt til i treet. Vi sier at *u* er **forgjengeren** eller **forelderen** til *v* i treet. Siden hver node kun kan bli funnet *en* gang, har nodene kun en forelder.

Implementasjonen av BFS prosedyren under antar at input-grafen `G = (V, E)` er representert i en naboliste. Vi lagrer tilstanden til hver node $u \in V$ i attributten *u.color*, hvor hvit ikke er oppdaget, grå venter på å bli traversert, og sort er ferdig. Forgjengerer til u ligger i attributten $u.\pi$. Dersom noden mangler noen av disse attributtene vil de være satt til å være NIL. Algoritmen bruker også en FIFO kø Q, for å håndtere settet med gråfargede noder.

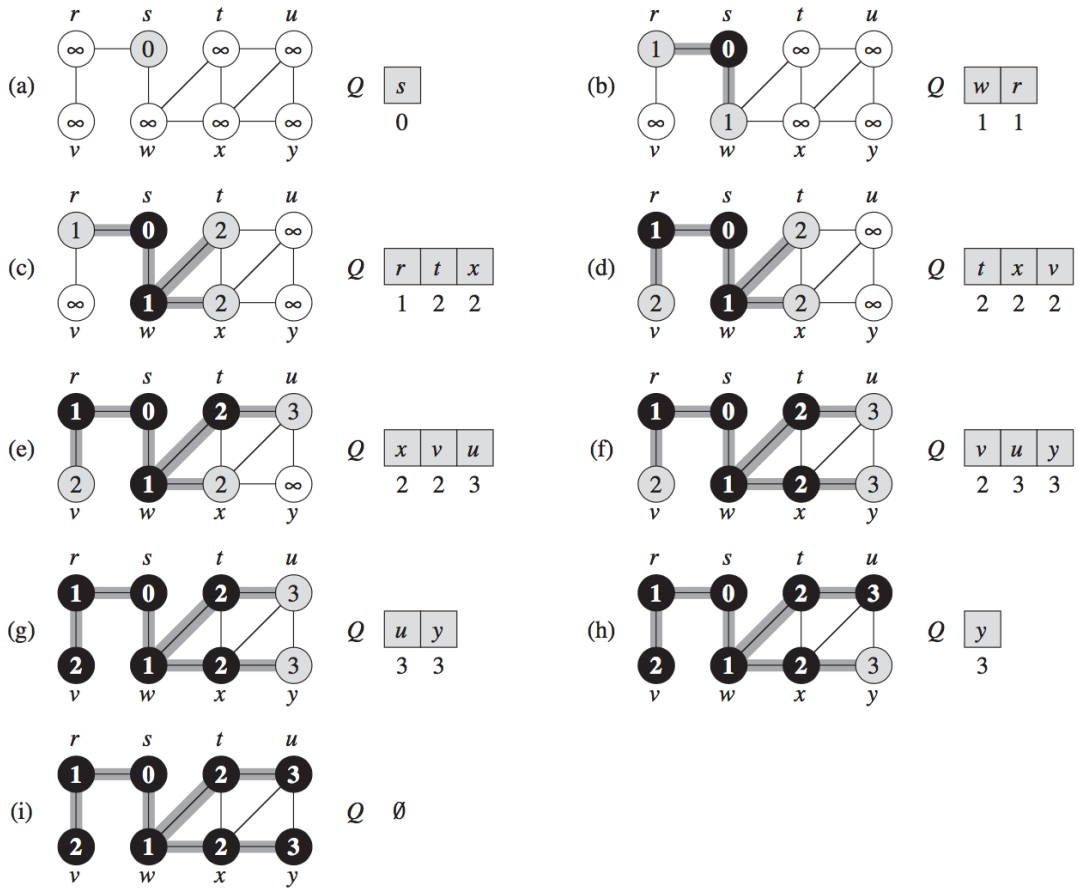
- Det at vi bruker en FIFO-kø er det som lar BFS finne de korteste stiene til alle noder, siden vi utforsker grafen "lagvis" utover.

```
BFS(G, s)
1       for each vertex u ∈ G.V - {s}      //setter farge, avstand og nabo
for hver node - O(V)
2           u.color = WHITE
3           u.d = ∞
4           u.π = NIL
5       s.color = GRAY
6       s.d = 0
7       s.π = NIL
8       Q = ∅
9       ENQUEUE(Q,s)                // O(1)
10      while Q ≠ ∅
11          u = DEQUEUE(Q)          // O(1)
12          for each v ∈ G.Adj[u]   // Summen av lengden til alle
nabolistene er θ(E), og tid brukt tid på å scanne disse blir - O(E)
13              if v.color = WHITE
14                  v.color = GRAY
15                  v.d = u.d + 1
16                  v.π = u
17                  ENQUEUE(Q, v)       // O(1)
18          u.color = BLACK
```

Kjøretiden:

- Operasjonene for *Enqueueing* og *Dequeueing* tar *O(1)* tid, og da blir total tid brukt på kø-operasjoner *O(V)*.
- Siden prosedyren skanner igjennom nabolisten til hver node kun når noden blir *dequeuet*, går den igjennom hver naboliste på det meste én gang. Siden summen av lengden på alle nabolistene er θ(*E*).
- Initialiseringen på starten er O(*V*).
- Den totale kjøretiden for BFS er derfor `O(V + E)`.

Kjøring av prosedyren BFS:



Bredde-først trær:

Prosedyren BFS bygger et bredde-først tre når den søker i grafen. Treet korresponderer til π attributten. For en graf $G = (V, E)$ med en kilde s, definerer vi forgjenger subgrafen til G som $G_\pi = (V_\pi, E_\pi)$. Vi kaller kantene i E_π for **tre-kanter**. I dette kapittelet antas det at alle kanter har en enhet vekt, dvs. lik, siden de egentlig ikke har noen vekt.

Print-Path

Følgende prosedyre printer ut nodene til den korteste veien fra s til v, der en antar at BFS allerede har konstruert et bredde-først tre.

```
PRINT-PATH(G, s, v)
1    if v == s
2        print s
3    elif v.π == NIL
4        print "no path from " s " to " v " exists"
5   else
6        PRINT-PATH(G,s,v.π)
7        print v
```

> Denne prosedyren kjører i linær tid i antall noder i veien som printes, siden hvert rekursive kall er for en vei en node kortere.

Dybde-først søk

Strategien med dybde-først søk er som navnet impliserer - søke dypere i grafen når det er mulig. Algoritmen utforsker kantene ut fra den nyligste oppdagede noden *v*, som fortsatt har ikke-utforskede kanter. Når alle av *v* 's kanter har blitt utforsket, går prosedyren tilbake til noden *v* kom fra for å se etter ikke-utforskede kanter.

Som i bredde-først søk, vil dybde-først søk når den oppdager en node *v* i en naboliste til en allerede oppdaget node *u*, notere dette ved å sette `v.π = u`. I motsetning til bredde-først søk, der forgjengerne former et tre, vil **forgjenger delgrafen** til DFS være litt annerledes. Vi lar $G_\pi = (V, E_\pi)$, der $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq NIL\}$.

Forgjenger subgrafen til DFS danner derfor en **dybde-først skog** med flere **dybde-først trær**. Kantene i E_π er *tre-kanter*.

- Som i BFS, farger dybde-først søk nodene som den finner underfveis i prosedyren for å markere deres status: Hver node farges initiellt `WHITE` , og blir `GRAY` når de blir oppdaget i søket, og blir farget `BLACK` når de er ferdige, og det er når nabolisten har blitt utforsket fullstendig.

I tillegg til å lage en *dybde-først skog*, **tidsstemlpler** DFS også hver node. Hver node *v* har to tidsstempler:

- Første tidsstempel - *v.d* har lagret når *v* først ble funnet og farger *v* `GRAY`.
- Andre tidsstempel - *v.f* har lagret når søket slutter å se på *v* 's naboliste, og farger *v* `BLACK`.

Disse tidsstempelene gir viktig informasjon om strukturen til grafen og generelt hjelpende når man skal resonnere over oppførselen til dybde-først søket.

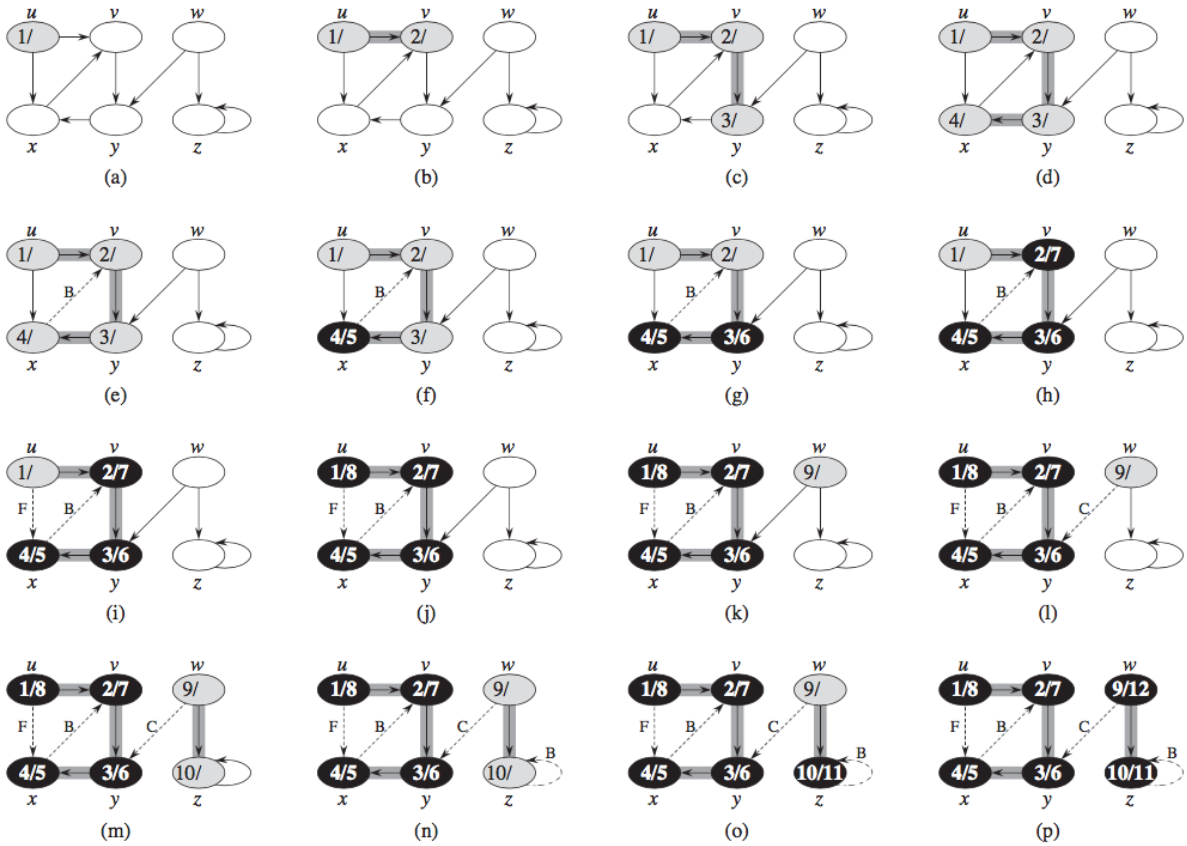
Prosedyren DFS under lagrer når den oppdager noden *u* i attributten *u.d* og når den blir ferdig med noden *u* i *u.f*. Disse tidsstempelene er tall mellom 1 og 2 | V |, siden det er to tidsstempeler for hver node (|V| noder).

Input er en graf G som kan være rettet eller urettet, og variabelen time er en global variabel som brukes for *tidsstempling*.

```
DFS(G)
1    for each vertex u ∈ G.V
2        u.color = WHITE
3        u.π = NIL
4    time = 0
5    for each vertex u ∈ G.V
6        if u.color == WHITE
7                DFS-VISIT(G, u)


DFS-VISIT(G, u)
1    time = time + 1            // white vertex u has just been discovered
2    u.d = time
3    u.color = GRAY
4    for each v ∈ G.Adj[u]      // explore (u, v)
```

```
5       if v.color == WHITE
6           v.π = u
7           DFS-VISIT(G, v)
8   u.color = BLACK                 // blacken u; it is finished
9   time = time + 1
10  u.f = time
```

Kjøring av algoritmen:



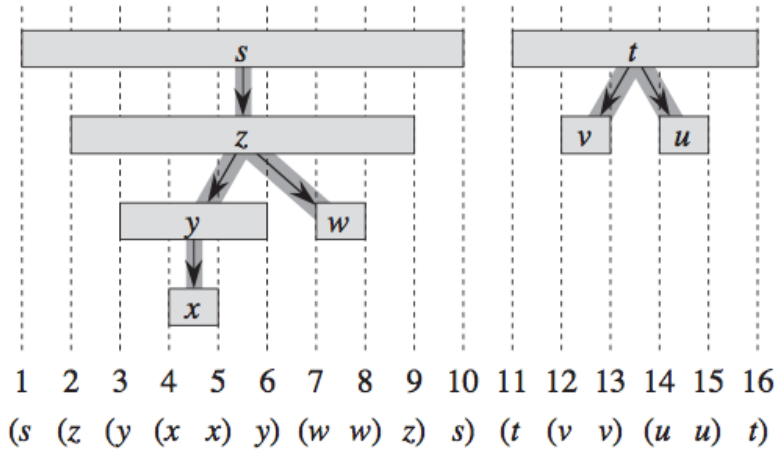
Kjøretid:

- Løkkene på linje 1-3 og linje 5-7 i DFS tar **θ(*V*)**, ekslusiv tiden det tar å kjøre kallet på *DFS-Visit*.
- Prosedyren *DFS-Visit* blir kalt på nøyaktiv én gang per node $v \in V$, siden noden u som *DFS-Visit* blir kalt med må være `WHITE`og det første *DFS-Visit* gjør er å farge den `GRAY`.
    - Under utføringen av *DFS-Visit(G, v)* kjøres løkken på linje 4-7 $|Adj[v]|$ ganger. Siden $\sum|Adj[v| = \theta(E)$, blir den totale kostnader for linje 4-7 i *DFS-Visit* **θ(*E*)**
- Den total kjøretiden til DFS blir derfor **θ(*V* + *E*)**

Egenskaper til dybde-først søk

Den mest essentielle egenskapen til DFS er at forgjenger subgrafen G_π former en skog av trær, siden strukturen til dybde-først trærne speiler strukturen til de rekursive kallene på *DFS-Visit*.

En annen viktig egenskap til DFS er oppdagelse og slutt tiden har **parantes struktur**. Dersom vi representerer funnet av noden *u* med en venstre parantes "(*u*" og representerer slutten til noden med høyre parantes "u)", da former historien av "discoveries" og "finishes" et vellformet uttrykk:



Parantesteoremet

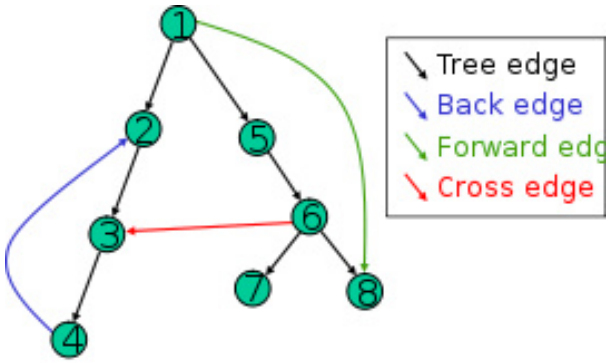
I ethvert dybde-først søk av en (rettet eller urettet) graf $G = (V, E)$, hvor for ethvert par noder u og v, holder akkurat ett av disse tre forholdene:

- Intervallene $[u.d, u.f]$ og $[v.d, v.f]$ er helt disjunkte, og hverken u eller v er en etterkommer den andre i dybde-først skogen.
- Hele intervallet $[u.d, u.f]$ er i intervallet $[v.d, v.f]$, og u er en etterkommer av v i ett dybde-først-tre.
- Hele intervallet $[v.d, v.f]$ er i intervallet $[u.d, u.f]$, og v er en etterkommer av u i ett dybde-først-tre.

Klassifisering av kanter

Vi definerer fire typer kanter i dybde-først skogen G_π produsert av et dybdeførst søk på G:

1. **Tree edges** er kanter i dybde-først skogen G_π. Kanten (u, v) er en *tree edge* dersom v først ble funnet ved utforskning av kanten (u, v).
2. **Back egdes** er kantene (u, v) som forbinder en node u til en forgjenger v i et dybde-først tre. Vi ser på selv-løkker, som kan forekomme i rettede grafer til å være *back egdes*.
3. **Forward edges** er de *non-tree edges* (u, v) som forbinder en node u til en etterkommer v i ett dybde-først tre.
4. **Cross edges** er alle de andre kantene. De kan gå mellom noder i samme dybde-først tre, så lenge en av nodene ikke er en forgjenger til den andre, eller så kan de gå mellom noder i forskjellige dybde-først trær.



I DFS har vi klassifisert kantene slik:

- `WHITE` har indikert en *tree edge*
- `GRAY` har indikert en *back egde*
- `BLACK` har indikert en *forward* eller *cross edge*

Implementere DFS med en Stack

Prosedyren BFS, som skrevet om over, kan tilpases til å oppføre seg nesten helt likt som DFS. Dette kan en gjøre ved å bytte ut *FIFO-køen* ***Q*** med en *LIFO-kø*, eller *stakk* (eng. *stack*). Vi mister da tidsstemplene (*v.d* og *v.f*), men rekkefølgen noder farges grå og svarte på vil bli den samme.

Slik DFS er implementert over har den *ingen startnode*, men starter bare fra hver node etter tur, til den har nådd hele grafen. Derfor kan man si at *BFS*s slekter mer på *DFS-Visit*.

Grunnen til at en LIFO-kø (*stack*) gir oss samme atferd som en rekursiv traversering (altså DFS) er at vi egentlig bare simulerer hvordan rekursjon er implementert:

- Internt bruker maskinen en *kallstakk*, der informasjon om hvert kall legges øverst og hentes frem når rekursive kall er ferdige.

```
STACK-DFS(G, s)
1       for each vertex u ∈ G.V - {s}
2           u.color = WHITE
3           u.d = ∞
4           u.π = NIL
5       s.color = GRAY
6       s.d = 0
7       s.π = NIL
8       S = ∅
9       PUSH(S, v)
10      while Q ≠ ∅
11          u = POP(S)
12          for each v ∈ G.Adj[u]
13              if v.color = WHITE
14                  v.color = GRAY
15                  v.d = u.d + 1
16                  v.π = u
17                  PUSH(S, v)
18          u.color = BLACK
```

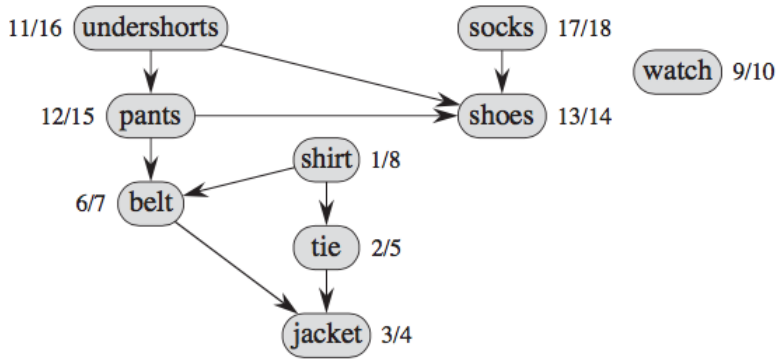
Topologisk sortering

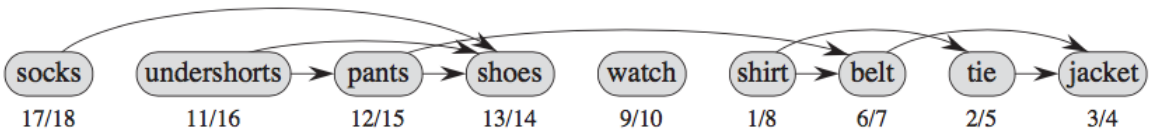
Vi kan bruke **dybde-først** søk til å **topologisk sortere** en rettet asyklisk graf eller en *DAG* (*eng. directed acyclic graph*). En *topologisk sortering* av en DAG `G = (V, E)` er en lineær ordning av alle nodene slik at dersom G inneholder en node *v* med egde til *u*, da kommer *u* før *v* i ordningen.

Vi kan se på topologisk sortering av en graf som en ordning av nodene langs en horisontal linje slik at alle de *rettede kantene* går fra venstre mot høyre. Man begynner med noden som ikke har noenkanter inn til seg.

```
TOPOLOGICAL-SORT(G)
1    call DFS(G) to compute finishing times v.f for each vertex v
2    as each vertex is finished, insert it onto the front of a linked list
3    return the linked list of vertices
```

Prosedyren vil returnere en lenket liste med topologisk sorterte noder i synkende rekkefølge med hensyn på *v.f (finish-time)*, som du ser i figuren under:





Kjøretid: Vi kan utføre topologisk sortering på $\Theta(V + E)$ tid, siden dybde-først søk bruker $\Theta(V + E)$ tid og det tar O(1) tid å innsette hver av de $|V|$ nodene foran i den lenkede listen.

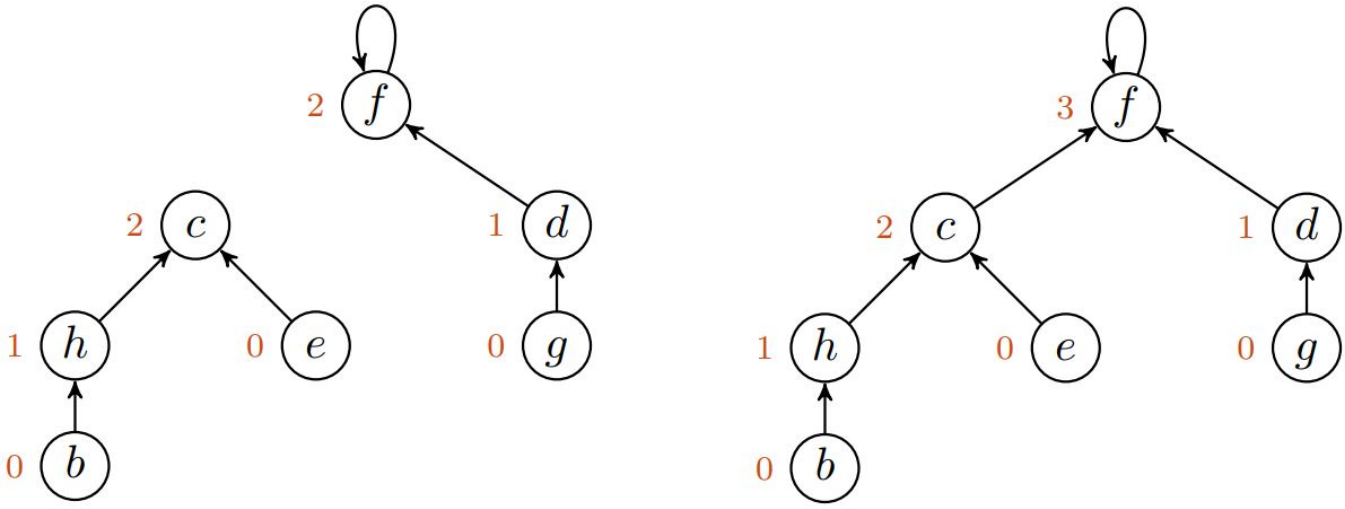
Minimale spenntrær

Disjunkte mengder

En disjunkt-sett datastruktur vedlikeholder en samling $S = \{S_1, S_2,...,S_k\}$ av disjunkte dynamiske sett. Vi identifiserer hvert sett med en representativ, som er et medlem av settet.

Som i de andre dynamisk-sett implementasjonene vi har sett på, representerer vi hvert element i ett sett med et objekt. La *x* være et objekt, ønsker vi å støtte følgende funksjoner:

- `MAKE-SET(x)` lager et nytt sett med dens eneste medlem, og dens representativ, som *x*. Siden settene er disjunkte krever vi at *x* ikke allerede ikke er i et annet sett.
- `UNION(x, y)` forener de dynamiske settene som inneholder *x* og *y*, la oss si S_x og S_y, inn i ett nytt sett som er unionen av disse to settene.
    - Representativen til det resulterende settet kan være et vilkårlig element i $S_x \cup S_y$, selvom mange implementasjoner av *Union* velger en av representantene til *S_x og S_y, som den nye representanten.
    - Siden vi krever at settene i *S* er disjunkte må vi nå fjerne S_x og S_y fra samlingen *S*.
- `FINDSET(x)` returnerer en peker til representanten til det (unike) settet som inneholder *x*



Illusterer hvordan to disjunkte menger kobles sammen ved å først finne roten ved hjelp av `findset(x)`, for så å og kalle `union(x,y)` på dem.

En av de mange bruksområdene til disjunkte-sett datastrukturen er å kunne definere de koblede komponenetene i en urettet graf. Prosedyren *Connected-Components* bruker de disjukte-sett operasjonene til å regne ut de koblede komponentene i grafen. Når *Connected-Components* har prosessert grafen, kan prosedyren *Same-Component* svare på om to noder er i den samme koblede komponenten.

```
CONNECTED-COMPONENTS(G)
1    for each vertex v ∈ G.V
2        MAKE-SET(v)
3    for each edge (u, v) ∈ G.E
4        if FIND-SET(u) ≠ FIND-SET(v)
5            UNION(u, v)
```

```
SAME-COMPONENT(u, v)
1    if FIND-SET(u) == FIND-SET(v)
```

```
2        return True
3    return False
```

Disjunkte-sett skoger

En raskere implementasjon av disjunkte sett er at vi representerer settene med rotfestede trær, der hver node inneholder ett medlem og hvert tre representerer ett sett. I en *disjunkt-sett skog* peker hvert element kun til sin forelder. Roten i hvert tre innholder representativen og sin egen forelder.

Vi utfører de tre disjunkt-sett operasjonene følgende. Operasjonen *Make-Set* lager helt enkelt et tre med kun en node. Vi bruker *Find-Set* ved å følge forelder-pekerne elt til vi finner roten av treet. *Union* operasjonen får roten til det ene treet til å peke til roten til det andre.

Pseudokode for disjunkte-sett skoger

For å implementere en disjunkt-sett skog med union-av-rang hierarki må vi holde styr på rangene, dvs at hver node x får attributten $x.rank$, som er en øvre grense på høyden til x.

```
MAKE-SET(x)
1   x.p = x
2   x.rank = 0


UNION(x, y)
1   LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1   if x.rank > y.rank
2       y.p = x
3   else
4       x.p = y
5       if x.rank == y.rank
6           y.rank = y.rank + 1

FIND-SET(x)
1   if x ≠ x.p
2       x.p = FIND-SET(x.p)
3   return x.p
```

Kjøretiden: Når vi skal regne på samlet kjøretid for disse algoritmene får vi O(m lg n) der n er antall `MAKE-SET` operasjoner, og m er total antall `MAKE-SET`, `UNION` og `FIND-SET` operasjoner. Vi antar at de n *Make-Set*-operasjonene er de første n operasjonene som blir gjort.

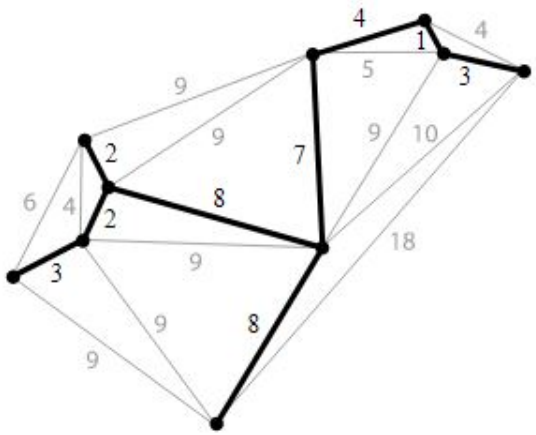
Minimale spenntrær - MST

Vi lar $G = (V, E)$ være en urettet graf. Vi ønsker å finne et asyklisk subset $T \subseteq E$, som kobler alle nodene sammen og der den totale vekten

$\displaystyle w(T)= \sum_{(u,v)\in T} w(u,v)$

er minimert. Siden *T* er asyklisk og kobler sammen alle nodene må den forme et tre, som vi kaller ett **spenntre**, da den "spanner" grafen *G*. Vi kaller problemet av å definere treet *T det minimale spenntre problemet*.

Vi skal se på to algoritmer for å løse MST-problemet: Kruskal's algoritme og Prim's algoritme. Begge algoritmene er grådige algoritmer og på hvert steg må algoritmene ta ett av flere mulige valg. Vi skal også se på en generisk MST metode, som lager et minimalt spenntre ved å legge til en kant av gangen. Deretter skal vi se på *Krusals*, som likner på *Connected-Components* algoritmen. Vi skal også se på *Prims* algoritme, som minner om *Djikstra's* korteste vei algoritme.



Et minimalt spenntre

Bygge et minimalt spenntre

Antat at vi har en sammenhengende, urettet graf $G = (V, E)$ med en vektfunksjon $w : E \rightarrow \mathbb{R}$ og vi ønsker å finne et MST for *G*. De to algoritmene vi skal se på bruker den grådige tilnærmingen på problemet. Denne grådige strategien er vist i den følgende generiske metoden:

I hvert steg ønsker vi å finne en kant (u, v) som vi kan legge til i A slik at $A \cup \{(u, v)\}$ også er et subset t av et minimalt spenntre. Vi kaller en slik kant for en **trygg kant** (*eng. safe edge*) for *A*, siden vi trygt kan legge den til i *A* og fortsatt vedlikeholde invarianten.

```
GENERIC-MST(G, w)
1   A = ∅
2   while A does not form a spanning tree
3       find an edge (u,v) that is safe for A
4       A = A ∪ {(u,v)}
5   return A
```

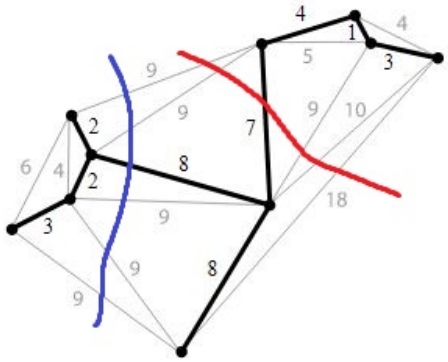
Definerer et **snitt** $(S, V - S)$ til en urettet graf $G = (V, E)$ som en partisjon av *V*. Vi sier at en kant $(u,v) \in E$ **krysser** snittet $(S, V - S)$ dersom en av dens endepunkter er i *S* og den andre er i $V - S$. Vi sier at et snitt

respekterer et sett *A* av kanter dersom ingen kanter i A krysser snittet. En kant er en **lett kant** som krysser et snitt dersom dens vekt er minimumet av enhver kant i snittet.

Hvordan identifisere en trygg kant:

La $G = (V, E)$ være en sammenhengende, urettet graf med vekter definert på E. La A være et subsett av E som er inkludert et minimalt spenntre for G, la $(S, V - S)$ være et snitt i G som respekterer A, og la (u,v) være en lett kant som krysser $(S, V - S)$. Da er kanten (u,v) en trygg kant for A. **Derfor er lette kanter, trygge kanter.**

Med andre ord. For et gitt snitt, så vil edgen med den laveste vekten være en del av det minimale spenntreet som vist under med et rødt og et blått snitt:

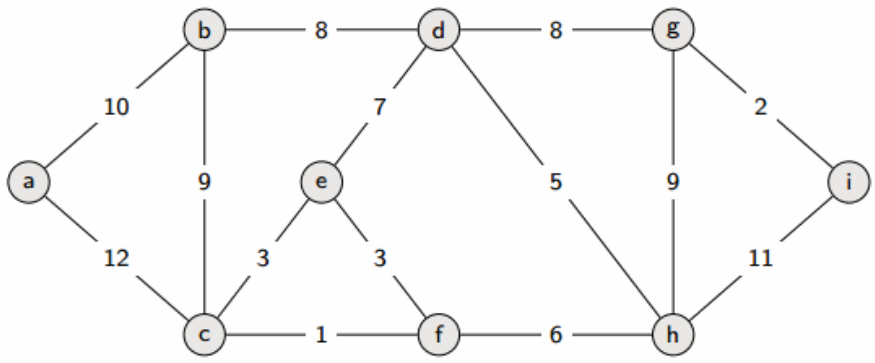


Kruskal's algoritme

Helt enkelt: Velg til enhver tid den billigste kanten i treet som kobler sammen nye noder men som ikke skaper en sykel.

Kruskal's algoritme finner en trygg kant for å legge til i den voksende skogen, ved å finne alle kantene med lavest vekt som kobler sammen to trær i skogen, men som samtidig ikke lager en sykel

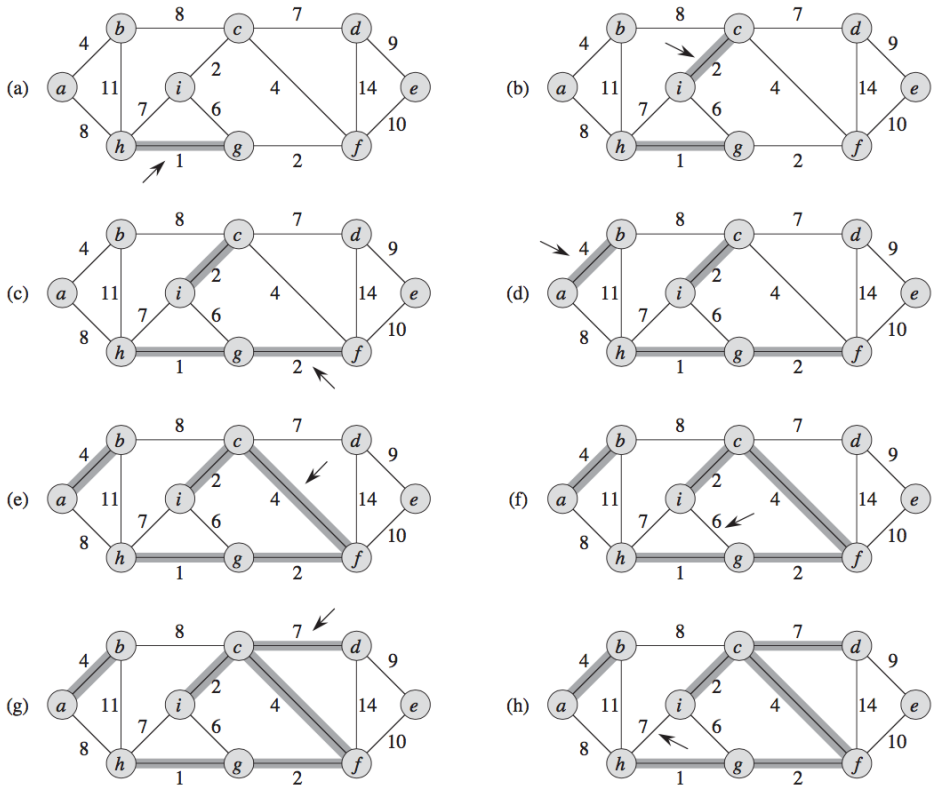
---



Kruskal's algoritme kvalifiseres som en grådig algoritme fordi ved hvert steg legger den til en kant med minst mulig vekt i skogen. Implementeringen av *Kruskal* 's algoritme likner algoritmen for å finne sammenhengende komponenter fra *traversering av grafer*:

- Den bruker disjunkt-sett datastruktur for å vedlikeholde flere disjunkte sett med elementer. Hvert sett inneholder nodene til et tre i den gjeldene skogen.
- Operasjonen *Find-Set(u)* returnerer et element fra settet som representerer inneholdet *u*. Derfor kan vi bestemme om to noder *u* og *v* kommer fra det samme treet, ved å skjekke `FIND-SET(u) == FIND-SET(v)`.
- For å kombinere trær, bruker *Kruskal* 's algoritmen *Union* prosedyren.

```
MST-KRUSKAL(G, w)
1    A = ∅
2    for each vertex v ∈ G.V
3        MAKE-SET(v)
4    sort the edges of G.E into nondecreasing order by weight w
5    for each edge (u,v) ∈ G.E, taken in nondecreasing order by weigth
6        if FIND-SET(u) ≠ FIND-SET(v)
7            A = A ∪ {(u,v)}
8            UNION(u,v)
9    return A
```



Figuren over viser hvordan *Kruskal's* fungerer:

- Linje 1-3 initialiserer settet *A* til et tomt sett og lager | *V* | trær, hvert tre med en node.
- For-løkken på linje 5-8 ser på kanter etter vekt, fra lav til høy. For-løkken skjekker, for hver kant (*u,v*), om endepunktene *u* og *v* er i samme tre.
    - Dersom de er det, kan ikke kanten (*u,v*) bli lagt til i skogen uten å lage en sykel, og kanten blir derfor forkastet.

- Dersom de tilhører forskjellige trær, i dette tilfellet så legges kanten (u,v) til A, og i linje 8 merges nodene i de to trærne.

Kjøretid:

Kjøretiden til Kruskal's algoritmen for en graf $G = (V, E)$, avhenger av hvordan vi har implementert den disjunkte datastrukturen. Dersom vi antar at vi har brukt den *disjunkte-sett-skog* implementasjonen med *union-av-rang* og *sti-kompresjon* hierarki, siden det er den raskeste implementasjonen vi vet om.

Operasjon	**Antall**	**Kjøretid**
Make-Set	V	O(1)
Sortering	1	O(E lg E)
Find-Set	O(E)	O(α(V))
Union	O(E)	O(α(V))

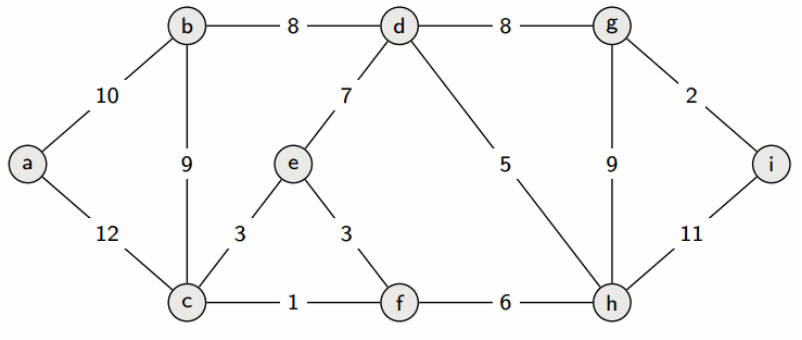
Det gir at kjøretiden totalt er: **O(E lg V)**

Prim's algoritme

Helt enkelt: Begynn i tilfeldig node. Velg den billigste kanten ut fra den noden som kobler inn en ny node.

Prim's algoritme opererer ganske så likt som Dijkstra's algoritme for å finne korteste vei i en graf. Prim's algoritme har den egenskapen at kantene i settet A alltid former et enkelt tre.

Treet starter med en vilkårlig rot node r og vokser til treet spenner alle nodene i V. Hvert steg legger til en lett kant til treet A, som kobler A til en isolert node - en som ingen andre kanter i A går til. Denne regelen gjør at kun trygge kanter legges til i A, og derfor når algoritmen terminerer vil kantene i A forme et minimalt spenntre.



Man kan også tenke på det som at det blå området former et snitt rundt det allerede eksisterende spenntreet, og at algoritmen velger den kanten med lavest vekt som krysser snittet, og legger denne til i spenntreet som nevnt i - hvordan identifisere en trygg kant.

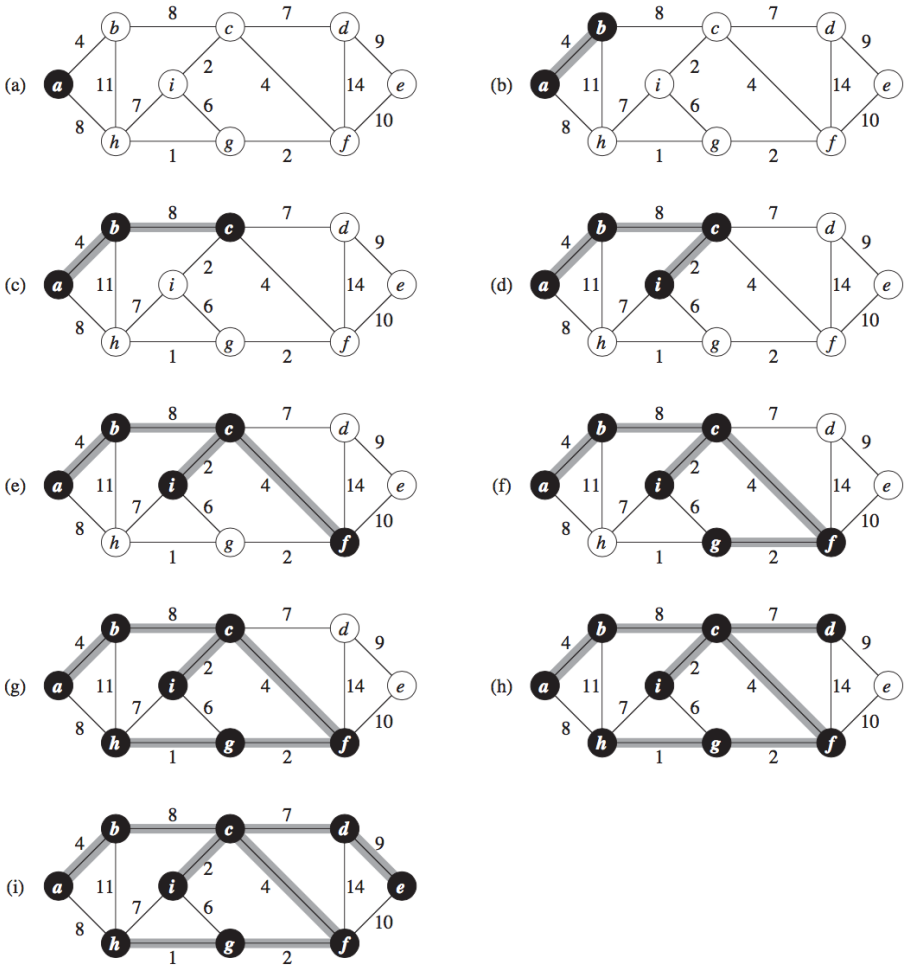
Denne strategien kvalifiseres som grådig siden det til treet legges til en kant, som bidrar minst mulig til den totale vekten til treet.

Under kjøringen av algoritmen vil alle nodene som ikke er i treet enda, ligge i en min-prioritets kø *Q* basert på *key* attributten. For hver node *v*, er *v.key* den minste vekten for enhver kant som kobler *v* til en node i treet.

```
MST-PRIM(G, w, r)
1    for each u □ G.V
2        u.key = ∞
3        u.π = NIL
4    r.key = 0
5    Q = G.V
6    while Q ≠ ∅
7        u = EXTRACT-MIN(Q)
8        for each v ∈ G.Adj[u]
9            if v ∈ Q and w(u,v) < v.key
10               v.π = u
11               v.key = w(u,v)
```

- Linje 1-5 setter *key* til hver node til ∞, unntat roten, samt hver forelder til å være NIL. Den initialiserer også min-prioritetskøen *Q* som inneholder *nodene*.

Illustrasjon av algortimen:



Kjøretiden

Kjøretiden til *Prim's algoritme* avhenger av hvordan vi har implementert min-prioritetskøen Q. Dersom vi implementerer Q som en binær min-heap, kan vi bruke *Build-Min-Heap* prosedyren for å gjøre linje 1-5 i $O(V)$ tid.

Kroppen til **while**-løkken kjøres $|V|$ ganger, og siden hver *Extract-Min* operasjon tar $O(\lg V)$ tid, blir den totale tiden for alle kall av *Extract-Min* $O(V \lg V)$.

For-løkken på linje 8-11 kjøres $O(E)$ ganger til sammen, og summen av lengden på alle nabolistene blir $2|E|$.

Endring av attributt på linje 11 involverer implisitt *Decrease-Key* operasjonen på min-heapen, som en binær min-heap bruker $O(\lg V)$ tid på.

Til sammen blir derfor den **totale kjøretiden** for *Prim's algoritme*:

- $O(V \lg V + E \lg V) = \mathbf{O(E \lg V)}$

> Dersom vi hadde brukt en Fibonacci heap, ville vi kunne forbedret *Prim's algoritme* til å kjøre på $\mathbf{O(E + V \lg V)}$ tid.

Korteste vei fra én til alle

I et **korteste vei problem** blir vi gitt en vektet, rettet graf $G = (V, E)$, med en vektfunskjon $w : E \rightarrow \mathbb{R}$ som mapper vektene til et sett kanter. Vekten $\boldsymbol{w(p)}$ av veien $p = \langle v_0, v_1, \ldots, v_k \rangle$ er summen av $w(p) = \sum w(v_{i-1},v_i)$ vektene til kantene på veien:

Vi definerer den korteste-vei vekten $\delta(u,v)$ fra u til v med:

$$w(p) = \sum^k_{i=1} w(v_{i-1}, v_i)$$

Varianter:

I denne forelesningen er fokuset på **single-source shortest-paths problem**: Gitt en graf $G = (V, E)$, ønsker vi å finne en korteste vei fra en gitt **kilde** (*eng. source*) node $s \in V$ for hver node $v \in V$. Algoritmen for single-source problemet kan løse mange andre problemer, som f.eks. disse variantene:

Single-destination shortest-paths problem: Finn en korteste vei til en gitt *destinasjon* node t fra alle andre noder. Ved å reversere retingen til hver kant i grafen, kan vi redusere dette problemet til et *singe-source problem*

Single-pair shortest-path problem: Finn en korteste vei fra u til v for gitte noder u og v. Dersom vi løser *single-source problemet* med kilde-node u, løser vi dette problemet også. Alle kjente algoritmer for dette problemet har samme *worst-case* kjøretid som den beste single-source algoritmen.

All-paris shortest-paths problem: *(Alle til alle)* Finn en korteste vei fra u til v for hvert eneste par av noder u og v. Vi kan løse dette problemet ved å kjøre en *single-source* algoritme en gang fra hver node, men vi kan i mange tilfeller løse den raskere. Mer om dette i neste seksjon.

Optimal substruktur til en korteste vei

Korteste-vei algoritmer avhenger typisk av egenskapen om at en korteste vei mellom to noder inneholder andre korteste veier innad. **Merk** at optimal substruktur er en av **nøkkelindikatorene** på at dynamisk programmering og den grådige metoden muligens tar sted. *Djikstra's algoritme*, som vi snart kommer til, er en grådig algoritme

Negative kanter

Noen instanser til single-source shortest-path problemet kan inkludere negative kanter. Dersom grafen G inneholder en negativ sykel, som kan nås fra s, er ikke lenger den korteste veien $\delta(u,v)$ definert (kan være definert som -∞). Ingen vei fra s til en node i sykelen kan være korteste vei, og ingen sti bli *kortest*.

Sykler

Som vi har sett kan den ikke innholde en negativ sykel. Den korteste veien kan heller ikke inneholde en positiv sykel, da om man hadde fjernet sykelen ville man fått en **enda kortere vei** med samme kilde s og destinasjon t. Dersom vi har en sykel med vekt 0, vil det fortsatt finnes en korteste-vei uten denne sykelen. Derfor sier vi at når vi finner korteste vei, har de ingen sykler, de er **enkle** veier.

> Siden enhver asyklisk graf $G = (V, E)$ har maks $|V|$ distinkte noder, har den også på det meste $|V| - 1$ kanter. Derfor kan vi kun se på korteste veier med maksimalt $|V| - 1$ kanter.

Representere korteste veier

Vi representerer korteste veier noe likt som vi representerte bredde-først trær. Gitt en graf $G = (V, E)$ har vi for hver node $v \in V$ en forgjenger $v.\pi$ som enten er en annen node eller NIL. Korteste-vei algoritmene i dette kapittelet *(Kap. 25)* setter π attributten slik at kjeden av forgjengere fra en node v løper tilbake langs en korteste vei fra s til v. Gitt en node v, der $v.\pi \neq$ NIL, vil prosedyren *Print-Path(G,s,v)*, skrive ut korteste vei fra node s til v.

Et korteste-vei tre er som et bredde-først tre, men den inneholder korteste veier fra kilden s definert på kant-vekter, isteden for antall kanter. Et korteste-vei tre med rot s er en rettet subgraf $G' = (V', E')$, hvor $V' \subseteq V$ og $E' \subseteq E$ slik at:

1. V' er settet med alle noder nåbare fra s i G
2. G' former et rotfestet tre med rot s, og
3. for alle noder $v \in V'$, er den unike veien fra s til v i G' den korteste veien fra s til v i G.

Slakking

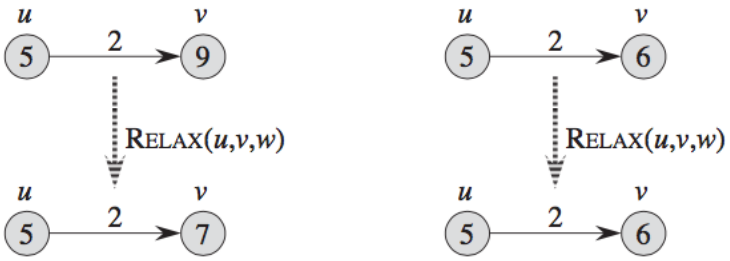
Algoritmene vi skal se på bruker teknikken **slakking** *(eng. relaxation)*. For hver node $v \in V$, vedlikeholder vi attributten *v.d*, som er en øvre grense på vekten til den korteste veien fra en kilde *s* til *v*. Vi kaller *v.d* **korteste vei estimatet**. Vi initialiserer korteste vei estimatet og forgjengerne med følgende **θ(V)** prosedyre:

```
INITIALIZE-SINGLE-SOURCE(G, s)
1   for each vertex v ∈ G.V
2       v.d = ∞
3       v.π = NIL
4   s.d = 0
```

Etter initialisering har vi v.π = NIL for alle noder $v \in V$, og s.d = 0 og v.d = ∞ for alle $v \in V$ - { *s* }

Prosessen av å slakke en kant (*u,v*) består av teste om vi kan forbedre den korteste veien til *v* som vi har, og dersom det går oppdatere *v.d* og *v.π*. Følgende kode utfører et slakke-steg på en kant (*u,v*) i **O(1)** tid:

```
1   if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```



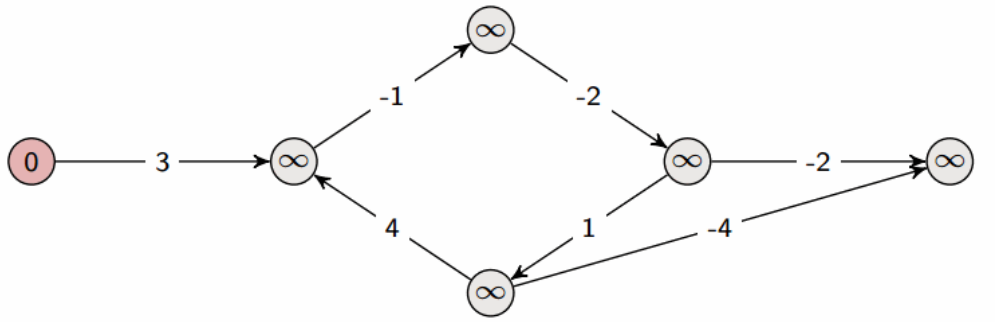
Figuren viser ett eksempel hvor en kortere sti ble funnet, og en hvor det ikke ble det.

Algoritmene kaller først *Initialize-Singe-Source* og slakker kantene gjentatte ganger:

- *Djikstra's algoritme* og *DAG-Shortest-Path* slakker hver kant nøyaktiv én gang.
- *Bellman-Ford* algoritmen slakker hver kant | *V* | - 1 ganger.

Bellman-Ford

Bellman-Ford algoritmen løser single-source korteste vei problemet på generelt basis der kantvektene **kan være negative**. Gitt en vektet, rettet graf $G = (V, E)$ med kilde *s* og vektfunksjon $w: E \rightarrow \mathbb{R}$, returnerer *Bellman-Ford* algoritmen en boolean verdi som indikerer om det finnes en negativ sykel som kan nås fra *s*. Dersom det er finnes en slik negativ sykel, betyr det at det ikke finnes noen løsning. Dersom det ikke er en negativ sykel, produserer algoritmen en korteste vei og dens vekter.



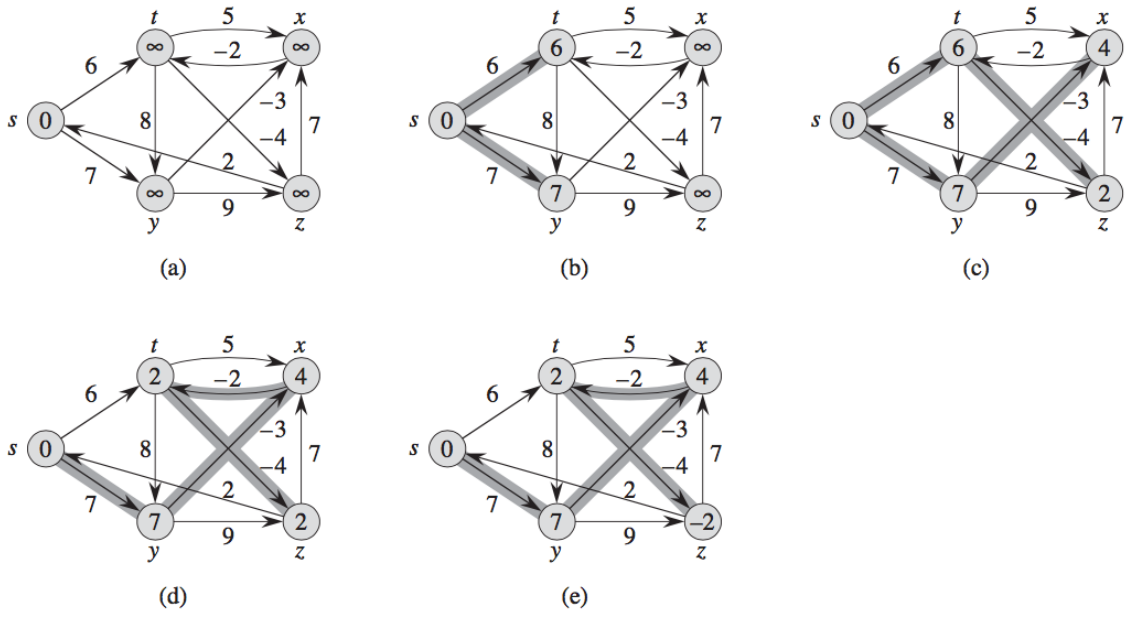
Algoritmen slakker kanter, ved å minimere *v.d* på kantene til en korteste vei fra *s* til hver node $v \in V$, til den finner den faktiske korteste-vei vekten $\delta(s,v)$. Algoritmen retureren True, hvis og bare hvis grafen ikke inneholder noen negative sykler.

```
BELLMAN-FORD(G, w, s)
1    INITIALIZE-SINGLE-SOURCE(G, s)
2    for i = 1 to |G.V| - 1
3        for each edge (u,v) ∈ G.E
4            RELAX(u,v,w)
5    for each edge (u,v) ∈ G.E
6        if v.d > u.d + w(u,v)
7            return False
8    return True
```

- Algoritmen slakker hver kant **| V | - 1** ganger.

Kjøretid: Bellman-Ford algoritmen kjører på **O(*VE*)** tid

> Siden initialiseringen tar θ(V) tid, og alle | *E* | kantene slakkes | *V* | - 1 ganger, og for-løkken på linje 5-7 tar O(*E*) tid, blir den totale kjøretiden derfor som sagt: **O(*VE*)**.



DAG-Shortest-Path

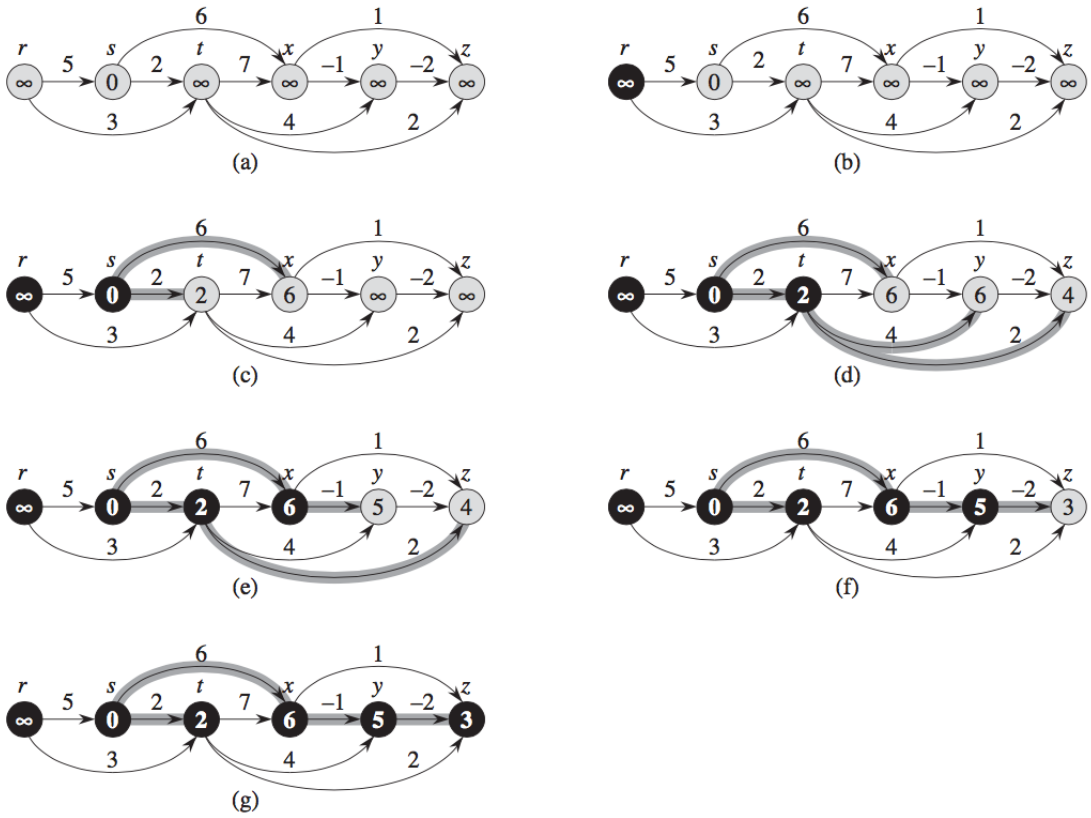
Ved å slakke kantene til en vektet DAG $G = (V, E)$ ifølge en topologisk sortering av nodene, kan vi regne ut den korteste veien fra en enkel kilde i $\boldsymbol{\theta(V + E)}$ tid. Korteste vei er godt definert i en DAG, siden det verken finnes **negative kanter eller sykler**.

Algoritmen starter med å topologisk sortere DAG-en til en lineær ordning på nodene. Dersom DAG-en inneholder en vei fra node u til node v, da kommer u før v i den topologiske sorteringen. Vi skal bare gå over nodene en gang i den topologiske sorterte rekkefølgen. Når vi prosesserer hver node, slakker vi hver kant som forlater noden. Slakker utkantene til nodene fra venstre mot høyre.

```
DAG-SHORTEST-PATH(G, w, s)
1   topological sort the vertices of G
2   INITIALIZE-SINGE-SOURCE(G,s)
3   for each vertex u taken in topological sorted order
4       for each vertex v ∈ G.Adj[u]
5           RELAX(u,v,w)
```

- Algoritmen slakker hver kant nøyaktig èn gang.

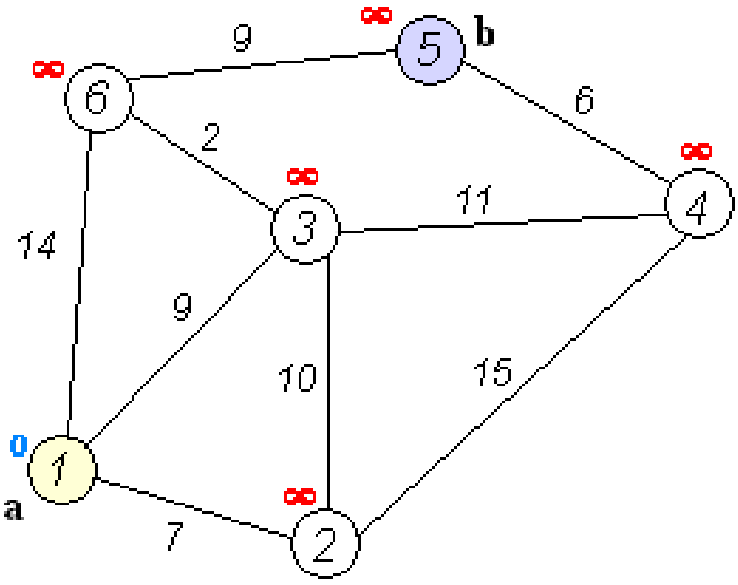
Kjøretiden: Kjøretiden til algoritmen er ganske enkel å analysere. Den topologiske sorteringen i linje 1 tar $\theta(V + E)$ tid. Kallet til *Initialize-Single-Source* på linje 2 tar θ(*V *) tid. *For*-løkken på linjene 4-5 slakker hver kant nøyaktig en gang, og hver iterasjon av for-løkken tar O(1) tid. Derfor blir den totale kjøretiden derfor $\boldsymbol{\theta(V + E)}$



> Korteste-vei problemet har optimal delstruktur. Delproblemene er avstanden fra kildenoden til *i* naboer, velg den som gir best resultat.

Dijkstra's algoritme

Dijkstra's algoritme løser single-source korteste vei problemet på en vektet, rettet graf $G = (V, E)$ der alle kantene har positiv vekt. Det betyr at Dijkstra's algoritme **ikke** kan brukes på grafer med **negative kanter**. Derfor antar vi videre at $w(u,v) \geq 0$ for hver kant $(u,v) \in E$. Som vi skal se er kjøretiden til Dijkstra's lavere enn *Bellman-Ford*.



Dijkstra's algoritme velger gjentatte ganger den noden $u \in V - S$ med minst korteste-vei-estimat, legger til u i S, og slakker alle kanter **ut** fra u.

I følgende implementasjon, bruker vi en min-prioritetskø Q av noder, basert på deres d (*distance*) verdi.

```
DIJKSTRA(G, w, s)
1   INITIALIZE-SINGLE-SOURCE(G,s)
2   S = ø33
3   Q = G.V
4   while Q ≠ ø
5       u = EXTRACT-MIN(Q)
6       S = S ∪ {u}
7       for each vertex v ∈ G.Adj[u]
8           RELAX(u,v,w)
```

> Slakker hver node én gang.

- Dijkstra slakker alle utkantene til den noden v med minst $v.d$.

Løkkeinvariant: $Q = V - S$, har også at $v.d = \delta(s,v)$

Kjøretid:

Operasjon	**Antall**	**Kjøretid**
Initialisering	1	O(V)
Build-Heap	1	O(V)
Extract-Min	V	O(lg V)
Decrease-Key	E	O(lg V)

Som gir den totale kjøretiden på `O(E lg V + V lg V)`

> Dersom vi hadde benyttet oss av en Fibonacci heap, vil *Extract-Min være O(1) og den totale kjøretiden blir da `O(V lg V + E)`

Korteste vei fra alle til alle

Nå skal vi se på problemet om å finne en korteste vei fra alle par av noder i en graf *(korteste vei fra alle til alle)*. Som i korteste vei fra en til alle problemet blir vi gitt en vektet, rettet graf $G = (V, E)$, og en vektfunksjon w. Vi ønsker å finne korteste vei mellom alle par $u,v \in V$.

Vi kan løse alle korteste vei fra alle til alle problemer ved å kjøre en single-source korteste vei algoritme $|V|$ ganger, en gang for hver node som kilden.

- Dersom alle kantvektene er positive, kan vi bruke Dijkstra's algoritme:
    - Med linær-liste som min-prioritetskø blir kjøretiden: $O(V^3)$
    - Med binær heap som min-prioritetskø blir kjøretiden: $O(VE \lg V)$
    - Med Fibonacci heap som min-prioritetskø blir kjøretiden: $O(V^2 \lg V + VE)$
- Dersom grafen har negative kanter kan vi bruke den tregere algoritmen, Bellman-Ford:
    - Den resulterende kjøretiden blir $O(V^2 E)$
    - På en tett graf der $E \approx V^2$ vil kjøretiden bli på hele $O(V^4)$

På dette problemet ser vi på nabomatriser, i stedet for nabolister som vi tidligere har jobbet med. Vi antar at nodene er nummerert 1,2,...,$|V|$, slik at input er en n x n matrise W som representerer kantvektene til en rettet graf G med n noder.

- Vi tillater negative kanter, men vi antar at input-grafen ikke har noen negative sykler.

For å løse kortestevei fra alle til alle problemet på en nabomatrise, må vi ikke bare regne ut korteste vei vektene men også en forgjenger matrise $\Pi = (\pi_{ij})$, hvor π_{ij} = NIL dersom $i = j$, eller dersom det ikke er en vei fra i til j, ellers er π_{ij} forgjengeren til j på en koreste vei fra i.

For å printe ut den korteste veien fra en node i til j, kan vi brue følgende prosedyre:

```
PRINT-ALL-PAIRS-SHORTEST-PATH(Π, i, j)
1       if i == j
2               print i
3       elif Π(i,j) == NIL
4               print "no path from " i " to " j "exists"
5       else PRINT-ALL-PAIRS-PATH(Π, i, π_ij)
6            print j
```

Floyd-Warshall

Nå skal vi se på en dynamisk programmerings algoritme for korteste vei fra alle til alle problemetet på en rettet graf *G*.

Algoritmen ser på mellomliggende noder av en korteste vei, hvor mellomliggende *p* er enhver node i *G* unntatt v_1 og v_2.

Vi skriver Floyd-Warhall algoritmen som en rekursiv bottom up algoritme. Vi definerer $d_{ij}^{(k)}$ rekursivt som:

$d_{ij}^{(k)} = \begin{cases} & w_{ij} & \text{if } k = 0 \ & min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k\geq 1 \ \end{cases}$

, og matrisen $D^{(n)} = d_{ij}^{(n)}$ gir det siste svaret $d_{ij}^{(n)} = \delta(i,j)$ for alle $i,j \in V$.

```
FLOYD-WARSHALL(W)
1 n = W.rows
2 D^(0) = W
3 for k = 1 to n
4   let D^(k) = (d_ij^(k)) be a new n x n matrix
5   for i = 1 to n
6       for j = 1 to n
7           d_ij^(k) = min(d_ij^(k - 1), d_ik^(k - 1) + d_kj^(k - 1))
8 return D^(n)
```

> Kjøretiden er bestemt av de tre nestede *for*-løkkene på linje 3-7. Siden hver utførerelse av linje 7 tar O(1) tid, kjører algoritmen på $\Theta(n^3) = \Theta(V^3)$.

- Det er $|V|$ noder vi skal gå igjennom, og for hver node kan man variere startnoden med $|V-1|$ muligheter, og sluttnoden $|V-2|$ muligheter.
- *Floyd-Warshall* kjører på $\theta(V^3)$ tid.
- Dijkstra bruker også $O(V^3)$ på alle-til-alle, men operasjonene per ledd i Floyd-Warshall er så mye mindre at denne vil lønne seg.
    - Dersom det er relativt få kanter i forhold til noder, vil derimot Dijkstra med en heap fremdels være bedre.

Illustrasjon av Floyd-Warshall:

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

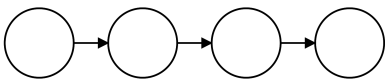
$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

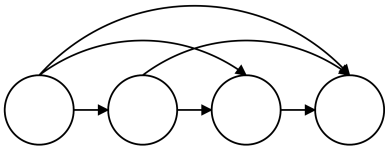
Transitive Closure

Gitt en rettet graf $G = (V, E)$ med et sett noder $V = \{1,2,..,n\}$ vil vi kanskje finne ut om G inneholder en vei fra i til j for alle par $i, j \in V$. Derfor definerer vi **transitiv closure** til G som grafen $G^* = (V, E^*)$, hvor $E^* = \{(i, j) :$ there is a path from i to j in $G\}$

Input



Output



Illustrerasjon av hva transitive closure gjør

Vi kan kjøre denne type algoritme på $\theta(n^3)$, og kan endre alle kantvektene til 1 og bruke *Floyd-Warshall*. , eller bruke operasjoner $\vee$, $\wedge$ for å regne ut om det finnes en vei.

```
1   n = |G.V|
2   let T (0) = (t_ij (0)) be a new n x n matrix
3   for i = 1 to n
4       for j = 1 to n
5           if i == j or (i, j) ∈ G.E
```

```
6                t_ij (0) = 1
7            else
8                t_ij (0) = 0
9    for k = 1 to n
10       let T (k)> = (t_ij (k)) be a new n x n matrix
11      for i = 1 to n
12          for j = 1 to n
13              t_ij (k) = t_ij (k - 1) v ( t_ik (k - 1) ∧ t_kj (k - 1))
14   return T (n)
```

Maksimal flyt

Vi kan se på en rettet graf som et "flytnettverk" og bruke det til å svare på spørsmål om materiell flyt. Se for deg en materie (f.eks. sjokolade) som flyter igjennom et system, fra en kilde *s*, hvor materien blir produsert, til et sluk *t*, hvor det konsumeres. Vi kan se på hver kant i flytnettverket som et rør med en viss kapasitet, og vi ønsker å oppnå maksimal flyt til sluket.

I maksimal flyt problemet ønsker vi å finne ut den største mengden vi kan frakte fra kilden til sluket uten å bryte noen av kapasitetene i flytnettverket.

Flytnettverk

- Et flytnettverk `G = (V, E)` er en rettet graf.
- Hver kant har en **kapasitet** $c(u,v) \geq 0$.
- Vi krever også at dersom det finnes en kant (u,v), finnes det ikke noen kant (v,u) i den motsatte retningen.
- Dersom $(u,v) \notin E$, da definerer vi $c(u,v) = 0$.
- Grafen er sammenhengende og har ikke selv-løkker
- Vi har en **kilde** s og et **sluk** $t \in V$.
- Vi antar at hver node ligger på en vei fra kilden til sluket.
    - Dvs. at for hver node $v \in V$, inneholder flytnettverket en vei $s \rightsquigarrow v \rightsquigarrow t$.

Flyt

En flyt i et flytnettverk G er en funksjon $f: V \times V \rightarrow \mathbb{R}$, som har følgende egenskaper:

- **Kapasitetsbegrensning:** For alle $u,v \in V$, krever vi at $0 \leq f(u,v) \leq c(u,v)$
- **Flytbeholdning:** For alle $u \in V - \{s,t\}$, krever vi at `∑ f(u,v) = ∑ f(v,u)`.
    - Flyt inn = Flyt ut

Vi kaller mengden $f(u,v)$ for flyten fra node u til v.

Flytverdien er definert ved `|f| = ∑ f(s,v) - ∑ f(v,s)`, som den totale flyten ut av kilden, minus flyten inn i kilden.

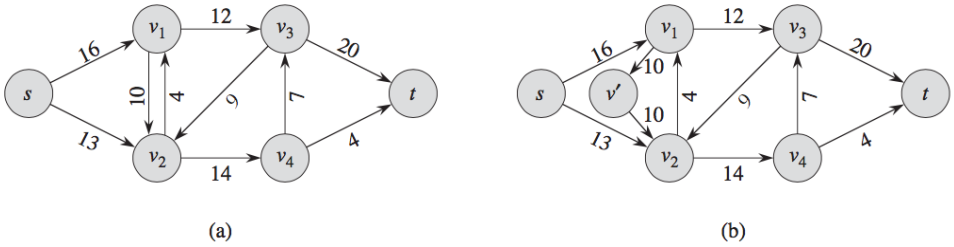
Antiparallelle kanter

La oss anta at man allerede i flytnetterverket har en kant $(v_1,v_2) \in E$, også får man et tilbud om en til kant (v_2, v_1). Da strider dette imot det vi antok over, det at dersom $(u,v) \in E$, så $(v,u) \notin E$.

Vi kaller to kanter (v_1, v_2) og (v_2, v_1) **antiparallelle kanter**. Dette løser vi ved å:

1. Velge en av de to antiparalelle kantene, f.eks. (v_1, v_2).
2. Splitter den, ved å legge til en ny node v'
3. Erstatte (v_1, v_2) med et par av kanter (v_1, v') og (v', v_2).
4. Begge kantene med kapasitet som den originale kanten.

Illustrasjon:

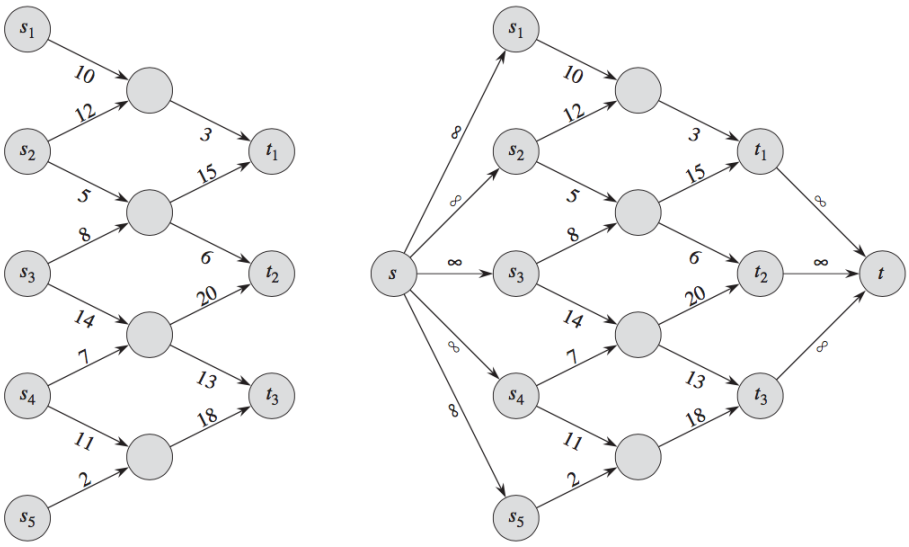


Nettverk med flere kilder og sluk

Et maksimal flyt problem kan har flere kilder og sluk, istedet for en av hver. Dersom man har et sett med *kilder* $\{ s_1, s_2,..., s_m \}$ og et sett *sluker* $\{ t_1, t_2,..., t_n \}$.

Vi kan redusere dette problemet til et vanlig maksimal flyt problem. Vi legger da til en **superkilde** s og legger til en rettet kant (s, s_i) med kapasitet $c(s, s_i) = \infty$ for hver $i = 1,2,..,m$. Vi legger også til et **supersluk** t og legger til en rettet kant (t_i, t) med kapasitet $c(t_i, t) = \infty$ for hver i = 1,2,..,n

Illustrasjon:



Ford-Fulkerson-metoden

Vi skal nå se på Ford-Fulkerson metoden, og kaller det metode og ikke for en algortime da det finnes mange implementasjoner med forskjellige kjøretider. Metoden avhenger av tre viktige ideer:

- *Restnettverk*
- *Forøkende stier*
- *Kutt*

Ford-Fulkerson metoden øker flytverdien iterativt. Vi starter med $f(u, v) = 0$ for alle $u, v \in V$, gitt en initiell flyt av verdi 0. Ved hver iterasjon øker vi flytverdien i G ved å finne en *forøkende sti* i et *restnettverk* G_f. Når vi vet kantene til en forøkende sti, kan vi lett øke flyten slik at vi øker flytverdien. Vi øker flytverdien med kapasiteten til den laveste kanten i den forøkende stien vi fant.

Vi øker flyten helt til restnettverket ikke har flere forøkende stier.

```
FORD-FULKERSON-METHOD(G, s, t)
1       initialize flow f to 0
2       while there exists an augmenting path p in the residual network Gf
3               augment flow f along p
4       return f
```

Restnettverk

Intuitivt, gitt et flytnettverk G og en flyt f, består restnettverket G_f av kanter med kapasiteter som representerer hvor mye vi kan endre flyten på kantene i G.

En kant i flytnettverken kan ta imot enda større flyt, lik kantens kapasitet minus flyten i kanten. Dersom denne verdien er positiv kan vi putte kanten i G_f med en restkapasitet på $c_f(u, v) = c(u, v) - f(u,v)$. De

eneste kantene i G som er i G_f er de som kan ta imot mer flyt.

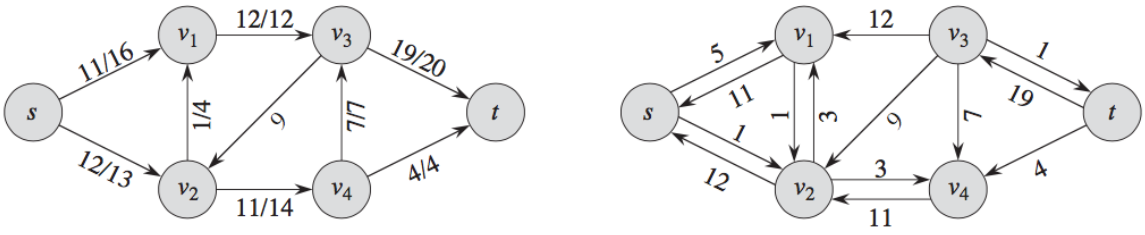
> De kantene som (u, v) som har like stor flyt som kapasitet har restkapasitet $c_f(u, v) = 0$, er ikke i G_f.

Restnettverket innholder kanskje kanter som ikke er i G. For å representere en mulig minskning av positiv flyt $f(u, v)$ på en kant i G, putter vi inn en kant (v, u) i restnettverket med restkapasitet $c_{f</sub}$ - det betyr at man kan sende flyt tilbake i kanten, dvs å **oppheve** (*eng. cancel*) flyten i kanten (u, v).

- Disse reverserte kantene i restnettverket lar algoritmen sende tilbake flyt som den allere har sent langs kanten. Det er ekvivalent med å senke flyten på kanten.
- Å sende flyt langs en kant, der det allerede går flyt, i et restnettverk er også kjent som **oppheving**. Det er dette bakoverkantene i restnettverket representerer.
- Dersom vi har en kant (u,v) med $c(u,v) = 16$ og $f(u,v) = 11$, kan vi øke $f(u,v)$ med $c_f(u,v) = 5$. Men algoritmen kan også sende tilbake 11 enheter av flyten fra v til u og dermed $c_f(v,u) = 11$.

En flyt i et restnettverk gir et kart for å legge til flyt i det originale flytnettverket.

Illustrasjon av restnettverk ut fra et flytnettverk:



Forøkende stier

Gitt et flytnettverk $G = (V, E)$, og en flyt f, en en enkel sti fra s til t i et restnettverk G_f en **forøkende sti** (*eng. augmenting path*).

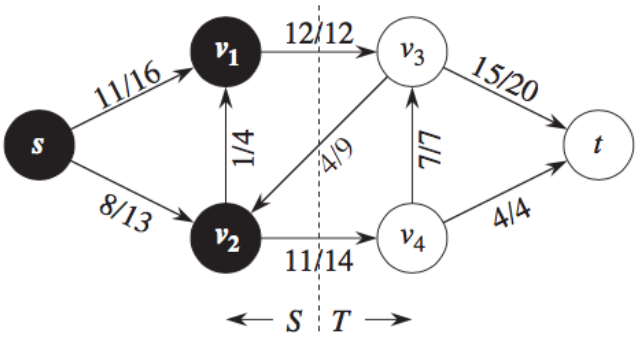
- Langs fremoverkanter: *Flyten kan økes*
- Langs bakoverkanter: *Flyten kan omdirigeres*
  - Altså: En sti der den totale flyten kan økes med opptil $c_f(u, v)$ uten å bryte med noen av kapasitetene i G.

Vi har at vi kan øke flyten på en kant i en forøkende sti p med restkapasiteten til p.

Snitt i flytnettverk

Ett **snitt** (S, T) av et flytnettverk $G = (V, E)$ er en partisjon av V inn i S, og $T = V - S$ slik at $s \in S$ og $t \in T$.

Her er et snitt (S, T):



- Nettoflyten lags kuttet blir: **f(*S, T*)** = $f(v_1, v_3) + f(v_2, v_4) - f(v_3, v_2)$ = 12 + 11 - 4 = **19**
- Kapasiteten blir da: **c(*S, T*)** = $c(v_1, v_3) + c(v_2, v_4)$ = 12 + 14 = **26**

Ford Fulkerson

Normal implementasjon:

- Finn økende sti først
- Finn så flaskehalsen i stien
- Oppdater flyt langs stien med denne verdien

I hver iterasjon av *Ford-Fulkerson-metoden*, finner vi *en eller annen* forøkende sti p og bruker p til å modifisere flyten f. Da erstatter f med $f \uparrow f_p$. Der f_p er flaskehalsen ($c_f(p)$) til p. Dermed får man den nye flytverdien $|f| + |f_p|$.

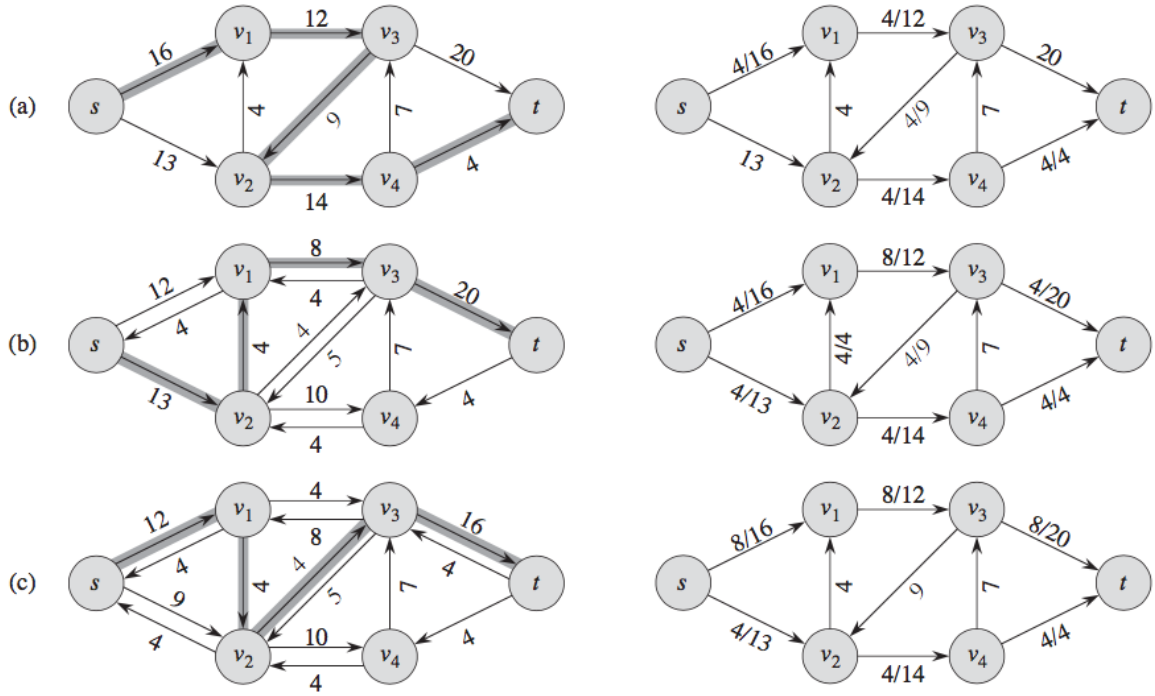
```
FORD-FULKERSON(G, s, t)
1       for each edge (u,v) ∈ G.E
2               (u,v).f = 0
3       while there exists a path p from s to t in the residual network G_f
4               c_f(p) = min{c_f(u,v) : (u,v) is in p}
5               for each edge (u,v) in p
6                       if (u,v) ∈ E
7                               (u,v).f = (u,v).f + c_f(p)
8                       else
9                               (v,u).f = (v,u).f - c_f(p)
```

- Linje 1-2 initialiserer flyten f til 0.
- Linje 3-9 kjører en **while**-løkke som gjentatte ganger finner en forøkende sti p i G_f og øker flyten f langs p med restkapasiteten $c_f(p)$. Hver restkant i stien p er enten en kant i det orignale nettverket eller en motsatt kant.
- Linje 6-9 oppdaterer flyten til hvert tilfelle:
  - Legge til flyt når restkanten er en original kant eller trekke fra dersom ikke.

Kjøretid:

- Dersom vi sier at $f*$ gir oss den maksimale flyten som vi kan oppnå.
- Da vil vi på det meste kjøre **while**-løkken for å finne en forøkende sta, $|f*|$ ganger, da flyten f må øke med minst en enhet av gangen.
- Hver iterasjon av **while**-løllen tar O(E) tid, samme gjør initialiseringen på linje 1-2.
- Dermed blir den totale kjøretiden på *Ford-Fulkerson-algoritmen* `O(E |f*|)`.

Illustrasjon av algoritmen:



Edmonds-Karp

Vi kan forbedre grensen på *Ford-Fulkerson* ved å finne en forøkende sti p i linje 3 med **bredde-først søk**. Det vil si at vi velger en forøkende sti som den korteste veien fra s til t, hvor hver kant har en enhet-vekt.

Denne algoritmen kaller vi for **Edmonds-Karp algoritmen**. Algoritmen kjører på O(VE^2)tid, som vi skal se på under.

Korteste-vei algoritmer avhenger typisk av egenskapen om at en korteste vei mellom to noder inneholder andre korteste veier innad. Det gjør også Edmonds Karp.

Mulig økning(*augmentation*): `v.a`

```
EDMONDS-KARP(G,s,t)
1       for each edge (u, v) ∈ G.E
2               (u, v).f = 0
3       repeat > until t.a == 0
4               for each vertex u ∈ G.V
```

```
5                       u.a = 0    //Reaching u in G_f
6                       u.π = NIL
7               s.a = ∞
8               Q = ∅
9               ENQUEUE(Q, s)
10              while t.a == 0 and Q ≠ ∅
11                      u = DEQUEUE(Q)
12                      for all edges (u, v), (v, u) ∈ G.E
13                              if (u, v) ∈  G.E
14                                      c_f(u, v) = c(u, v) - (u, v).f
15                              else c_f(u, v) = (v, u).f
16                              if c_f(u, v) > 0 and v.a == 0
17                                      v.a = min(u.a, c_f(u, v))
18                                      v.π = u
19                                      ENQUEUE(Q, v)
20              u, v = t.π, t    // Nå er t.f = c_f(p)
21              while u ≠ NIL
22                      if (u, v) ∈ G.E
23                              (u, v).f = (u, v).f + t.a
24                      else
25                              (v, u).f = (v, u).f - t.a
26                      u, v = u.π, u
```

Kjøretid:

- Operasjon: *Finn forøkende sti*
  - Antall: O(*VE*)
  - Kjøretid på operasjon: O(*E*)
- **Totalt:** O(VE^2) med bredde-først-søk i restnettverk

Hvorfor har vi O(VE) iterasjoner?

- Avstander *synker ikke* i residualnettverket
- En kant (*u, v*) kan være flaskehals maks annenhver iterasjon
- Vi velger korteste økende stier
  - Dermed må *v* først være 1 kant lenger unna enn *u*
  - Så, idet (*u, v*) dukker opp igjen, må *u* være 1 lenger unna enn *v*
  - Når (*u, v*) så er kritisk igjen, har altså avstanden til *u* økt med minst *2*
- Dermed kan vi maks ha *O(VE)* iterasjoner

Maksimum bipartitt matching

Gitt en urettet graf $G = (V, E)$, er en **matching** et subsett av kanter $M \subseteq E$ slik at for hver node $v \in V$, har er på det meste i én kant i M. Det vil si at ingen kantener i M deler noder.

Vi sier at en node $v \in V$ er **matchet** av matchingen M dersom en node i M har v i seg, hvis ikke er v **umatchet**. En **maksimum matching** er en matching med maksimum kardinalitet, det vil si flest mulig kanter, dvs. der $| M |$ er maksimal.

Forklaring av problemet:

Vi kan se på problemet som at vi har n antall nyredonorer, også har vi m pasienter som venter på en nyre. Det vi skal finne ut, er det maksimale antall med matcher, det vil si maksimale antall personer som kan få en nyre.

Vi lar da nodene i R representere donorene, og L representere pasientene, og kantene mellom dem representerer om nyrene er kompatibel med pasienten.

Bipartite grafer

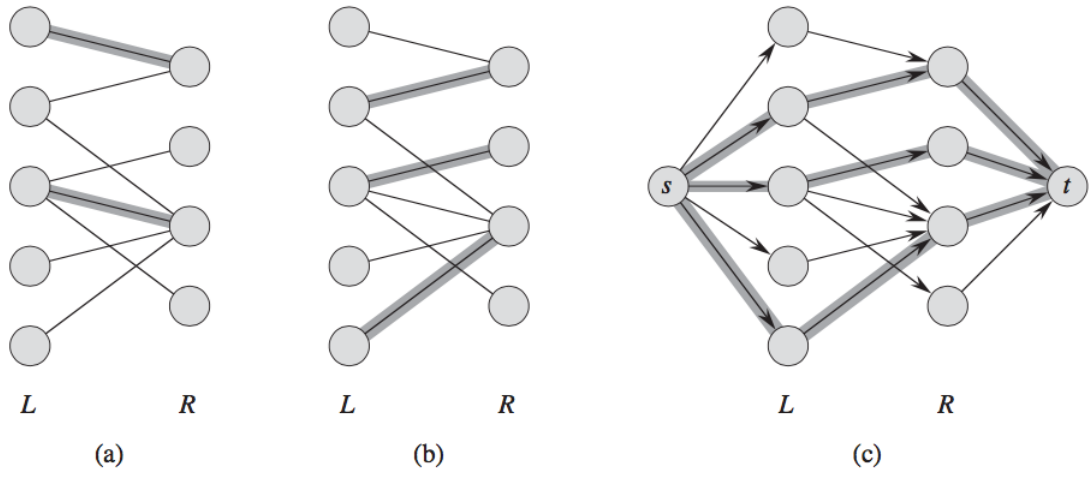
En graf der nodesettet kan partisjoneres til $V = L \cup R$, hvor L og R er disjunkte, og alle kanter i E går mellom L og R.

Finne en maksimum bipartitt matching

Vi kan bruke Ford-Fulkerson-metoden for å finne en maksimu matching på en urettet bipartitt graf $G = (V, E)$ i tid polynomisk med $| V |$ og $| E |$. Trikset er å konstruere et flytnettverk der flyt korresponderer med matcher, som vist i figuren under.

Vi definerer det korresponderende flytnettverket $G' = (V', E')$ for den bipartitte grafen som følgende:

- Vi lar kilden s og sluket t være nye noder, ikke i V, og vi lar $V' = V \cup \{ s, t \}$.
- De rettede kantene i G' er kantene i E, rettet fra L til R, sammen med $| V |$ nye kanter fra kilden til L og R til t.



Maksimal matching i en bipartitt graf G korresponderer til en maksimal flyt i det korresponderende flytnettverket G', og at vi dermed kan finne maksimum matching ved å kjøre en maksimal flyt-algoritme på G'.

Problemet er at maksimal flyt-algoritmen kan returnere desimaler, selvom flyt-verdien $|f|$ må være et heltall. Følgende teorem viser at vi kan bruke Ford-Fulkerson for å løse dette problemet.

Heltallsteoremet

Dersom kapasitetsfunksjonen *c* kun tar på seg heltallsverdier, da vil maksimumflyten *f* produsert av *Ford-Fulkerson-metoden* ha den egenskapen at $|f|$ er en heltall.

Generelt, vil flyten mellom to noder *f(u,v)* være et heltall for alle noder *u* og *v*.

NP-kompletthet

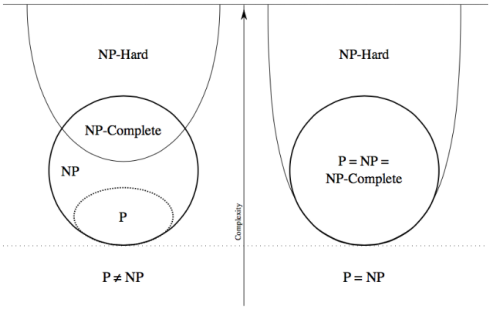
Nesten alle algoritmene vi har sett på hittil har vært **polynomisk-tid algoritmer**: med input på størrelse *n*, og som har worst-case kjøretid på $O(n^k)$. Slik er det nemlig ikke med alle problemer. Vi ser gjerne på problemersom kan løses i polynomisk-tid algoritmer som lette, og problemer som krever *superpolynomisk-tid* som vanskelige.

Vi skal nå se på en klasse problemer kalt de *NP-komplette* problemer. Ingen polynomisk-tid algoritme er funnet for å løse NP-komplette problemer, ingen har heller klart å bevise at det heller ikke finnes noen. Dette er det såkalte **N ≠ NP** spørsmålet, som er et av de store spørsmålene i datateknikk.

Flere NP-komplette problemer ligner gjerne på overflaten på problemer som vi vet vi kan løse i polynomisk tid. I hvert av de følgende parene av problemer, er det ene løsbart i polynomisk tid, og det andre er NP-komplett:

- **Shortest vs. longest simple path:** Vi kan finne single-source shortest path i en rettet graf $G = (V, E)$ i $O(VE)$ tid. For å finne koreste enkle vei ellom to noder er vanskelig. Men det å bestemme om en graf inneholder en enkel vei med minst et gitt antall kanter er NP-komplett.
- **Euler sti vs. Hamilton sykel:** En *Euler sti* til en sammenhengende rettet graf *G*, er en sykel som traverserer gjennom hver kant i G minst en gang, men vi kan besøke en node mer enn en gang. En *Hamilton sykel* til en rettet graf *G* er en enkel sykel som inneholder hver node i *V*. Å avgjøre om en rette graf innholder en hamilton sykel er NP-komplett.

NP-kompletthet og klassene P og NP



Gjennom det siste av pensum skal vi referere til tre klasser av problemer: **P**, **NP** og **NPC** (NP-komplett).

- Klassen **P** inneholder problemene som kan løses i polynomisk tid, altså i $O(n^k)$ for en konstant k, og inputstørrelse n
- Klassen **NP** består av problemer som kan *verifiseres* i polynomisk tid. Hva mener vi med at den kan verifiseres? Dersom vi hadde blitt gitt et **vitne** på en løsning, da kan vi bekrefte at *vitne* er korrekt i polynomisk tid. Klassen **co-NP** består av problemer som kan falsifiseres i polynomisk tid.
    - For eksempel i Hamilton sykel problemet, gitt en rettet graf G, ville *vitnet* vært en sekvens $\langle v_1, v_2,...,v_{|V|} \rangle$ av $|V|$ noder. Vi kan da lett skjekke i polynomisk tid at $(v_i, v_{i+1}) \in E$ for i = 1,2,.., $|V| - 1$, og at $(v_{|V|}, v_1) \in E$ også.

Ethvert problem i P er også i NP, siden dersom et problem er i P kan vi løse det i polynomisk tid, selv uten å bli gitt et vitne. Derfor tror vi for nå at **P** $\subseteq$ **NP**.

- Klassen **NPC** består av problemer som referer til som NP-komplette - det vil si at de er i NP og at er så "vanskelige" som ethvert problem i NP.
    - Dersom *et* eneste NP-komplett problem kan bli løst i polynomisk tid, har *alle* problemer i NP polynomisk-tid algoritme.

Hvordan vise at et problem er NP-komplett:

Når vi skal vise at et problem er NP-komplett, gjør vi en uttalelse om hvor vanskelig det er (eller i det minste hvor vanskelig vi tenker det er), istedet for å si hvor lett det er. Vi prøver ikke å vise eksistensen av en effektiv algoritme, men istedet vise at det er lite sannsynelig at en slik effektiv algoritme eksisterer.

Vi er **avhengig** av tre nøkkelkomponenter for å vise at et problem er NP-komplett:

Beslutnings problemer vs. optimaliserings problemer:

Mange interessante problemer er *optimaliseringsproblemer*, hvor hver mulige løsning har en tilknyttet verdi, og vi ønsker å finne en mulig løsning med den beste verdien. For eksempel korteste-vei problemet, der vi ønsker å finne en optimal løsning - den korteste veien.

NP-kompletthet gjelder ikke direkte for optimaliseringsproblemer, men beslutningsproblemer, der svaret kun er "*ja*" eller "*nei*" (eller mer formelt "1" eller "0").

Selv om NP-komplette problemer er begrenset til et rike beslutningsproblemer, kan vi dra nytte av det praktiske *forholdet* mellom optimaliseringsproblemer og beslutningsproblemer. Vi kan vanligvis *caste* et gitt optimaliseringsproblem som et relatert beslutningsproblem ved å legge inn en bundet verdi for å bli optimalisert. For eksempel er et avgjørelsesproblem relatert til *Kortest-vei is Sti*: Gitt en rettet graf G, noder u og, og et heltall k, eksisterer en sti fra u til bestående av maksimalt k kanter?

Vi kunne her løse *Sti* ved å løse *Korteste-vei*, og så sammenligne antall kanter i korteste vei med verdien til beslutningsproblemet k. Beslutningsproblemet er *lettere*, eller ikke vanskeligere, enn optimaliseringsproblemet.

Angitt på en måte som er mer relevant for NP-fullstendighet, hvis vi kan bevise at et *beslutningsproblem* er vanskelig, gir vi også bevis for at det relaterte *optimaliseringsproblemet* er vanskelig.

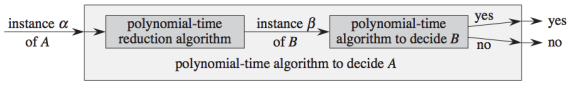
Reduksjoner:

Det at vi over viser at et problem ikke er vanskeligere eller lettere enn andre, gjelder selv når begge problemene er beslutningsproblemer. Vi tar fordel av denne ideen i nesten hvert eneste bevis av NP-kompletthet.

La oss se på et beslutningsproblem *A*, som vi ønsker å løse i polynomisk tid. Vi kaller inputen til en problem for instansen. La det være slik at vi allerede vetr hvordan vi kan løse et annet beslutningsproblem *B* i polynomisk tid. Til sist, la det være slik at vi har en prosedyre som *transformerer* enhver instans α av *A* til en instans β i *B*, med følgende egenskaper:

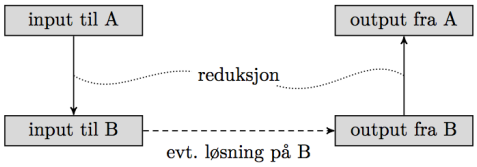
- Transformasjonen tar polynomisk tid.
- Svarene er det samme. Det vil si at svaret for α er *"ja"* hvis og bare hvis svaret for β også er *"ja"*.

Vi kaller en slik prosedyre i polynomisk tid en **reduksjonsalgoritme** og det gir oss en måte å løse problem *A* i polynomisk tid:



1. Git en instans α av problem *A*, buruker vi en polynomisk reduksjonsalgoritme som transformerer den til en instans β av problem *B*.
2. Kjør beslutningsalgoritmen for B, i polynomisk tid, på instansen β.
3. Bruk svaret for β som svar for α

> Vi transformerer input fra ett problem til et annet.



Vi kan utifra dette trekke to logiske konklusjoner og et par betraktninger:

1. Hvis vi kan løse *B*, så kan vi løse *A*
2. Hvis vi *ikke* kan løse *A*, så kan vi *ikke* løse *B*
3. Hvis vi *ikke* kan løse *B*, så sier det *ingenting* om *A*
4. Hvis vi *kan* løse *A*, sier det *ingenting* om *A*

La oss tenke oss at vi allerede er kjent med et problem *X*, og så støter på et nytt og ukjent problem *Y*, så har vi to scenarier der vi kan gjøre noe fornuften. Vi må gi *Y* rollen som *A* eller *B*:

- Hvis vi vil vise at *Y ikke er vanskeligere enn X*, så kan vi la *Y* innta rollen som *A*, og prøve å finne en reduksjon fra *Y* til *X*. Dette gjør vi ofte når vi prøver å bruke eksisterende algoritmer for et problem *X* til å løse et nytt problem *Y* → Vi reduserer *Y* til *X*, og løser så *X*.

- Men av og til mistenker vi at et problem vi støter på er vanskelig. Kanskje vi kjenner til et problem X, som vi *vet* er vanskelig, og vi vil vise at Y er *minst like vanskelig*. Da må vi i stedet la Y innta rollen som B, og redusere fra det vanskelige problemet. Vi skriver $A \leq B$ for å uttrykke at problemet A kan *løses ved hjelp av* B.
    - Det betyr at A ikke er vanskeligere B, siden vi skal redusere til B.

Abstrakte problemer

Vi definerer et **abstrakt problem** Q til å være en binær relasjon på et sett I av probleminstanser, og et sett S av problemløsninger.

Vi kan se på en abstrakt beslutningsproblem som en funksjon som mapper et sett av instanser I til et løsningssett {0,1}.

Dersom settet I skulle blitt kodet til binære strenger hadde vi kalt det et *konkret problem*, som vi skal se mer på under.

Koding av en instans

En **koding** (*eng. encoding*) av et sett S av abstrakte objekter er en mapping e fra S til et sett med binære strenger.

For at et dataprogram skal klare å løse et abstrakt problem, må vi representere probleminstansene på en måte som programmet skjønner. For eksempel er vi kjente med de naturlige tallene $\mathbb{N}$ = {0,1,2,3,...} som strengene {0,1,10,11,100,...}. Ved å bruke denne kodingen e(17) = 10001.

Vi kaller et problem der settet S med instanser er et set av binære strenger for et **konkret problem**. Vi sier at en algoritme som *løser* et konkret problem i O($T(n)$) tid, dersom den gitt en probleminstans i med lengde $n = |i|$ produserer en løsning i O($T(n)$) tid.

- Et konkret problem er **polynomisk-tid løsbar** dersom det finnes en algoritme som kan løse den på O(n^k) tid, for en konstant k.
- Vi definerer den *komplekse klassen P* som et sett av konkrete beslutningsproblemer som er polynomisk-tid løsbar.

Vi kan bruke *koding* for å mappe abstrakte problemer til konkrete problemer:

- Gitt et abstrakt beslutningsproblem Q, vil mapping av et sett instanser I til {0,1}, en koding $e : I \rightarrow$ {0,1}* kan lage et relatert konkret beslutningsproblem, so vi kaller $e(Q)$.

> Vi noterer {0,1}* for settet av alle strenger bestående av symboler fra settet {0,1}.

For et sett av instanser I sier vi at to enkodinger e_1 og e_2 er **polynomiske relaterte** dersom det finnes to polynomisk-tid funksjoner f_{12} og f_{21} slik at for hver $i \in I$, har vi at $f_{12}(e_1(i)) = e_2(i)$, og $f_{21}(e_2(i)) = e_1(i)$.

Representasjon av beslutningsproblemer som formelle språk

Alfabet **∑** er et avgrenset sett av symboler. *Språket* **L** over ∑ er et sett av strenger dannet av symboler fra ∑. Et beslutningsproblem Q er settet ∑* (språket av alle strenger over ∑), der ∑ = {0,1}.

Siden Q er kjennetegnet av de probleminstansene som produsere 1 ("*ja *"), kan vi se på Q som språket L over $\sum = \{0,1\}$, der

$$L = \{x \in \Sigma^* : Q(x) = 1\} .$$

- Betegner den tomme strengen med ε, og det tommespråket med ∅.
- Definerer komplementet til L med $\overline{L} = \sum - L$
- Vi definerer sammensetningen av to språk L_1L_2 av to språk L_1 og L_2 til språket $L = \{ x_1x_2 : x_1 \in L_1$ and $x_2 \in L_2$

Språkrammeverket gir oss muligheten til å konsistent utrykke relasjonen mellom beslutningsproblemer og algoritmer som løser de. En algoritme A **aksepterer** en streng s i {0,1}* dersom gitt input x gir A(*x *) = 1. Språlet som er **akseptert** av en algoritme A er settet av strenger:

$$L = \{x \in \{0,1\}^* : A(x) = 1\}$$

- En algoritme **avviser** en streng dersom A(x) = 0.
- Et språk er **bestemt** av en algoritme A dersom hver binærstreng i L er akseptert av A og hver binærstreng ikke i L er avvist av A.
- Et språk L er **akseptert i polynomisk tid** av en algoritme A hvis det er akseptert av A, og hvis det finnes en konstant k slik at for alle strenger med lengde n i L, aksepterer A input x på O(n^k) tid.
- Et språk L er **bestemt i polynomisk tid** av en algoritme A, hvis det eksisterer en k slik at for alle strenger x i {0,1}* av lengde n, algoritmen bestemmer at x er i L på O(n^k) tid:

$$\mathrm{P} = \{L \subseteq \{0,1\}^* : \text{ there exists an algorithm } A \text{ that decides } L \text{ in polynomial time}\} .$$

In fact, P is also the class of languages that can be accepted in polynomial time.

Verifikasjonsalgoritme

En **verifikasjonsalgoritme** skjekker om en løsning stemmer (*ja/nei*). Bruker et **vitne/sertifikat** for å skjekke problemet, for eksempel en Hamilton-sykel.

Vitne (*eng. certificate*): Gjelder for gitt input → Skal kunne gi *"Ja"* svar hvis svaret er *"Nei"*. Hvis svaret er *"Nei"*, skal det ikke eksistere Finnes ikke vitne hvis svaret er *Nei*.

Kompleksitetsklassen NP

Klassen av språk som kan *verifiseres* av en polynomisk-tid algoritme. Et språk hører til i NP hvis og bare hvis det eksisterer en to-input polynomisk algoritme A og en konstant c slik at:

$$L = \{x \in \{0,1\}^* : \text{ there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\} .$$

Vi sier at A verifiserer språket L i polynomisk tid.

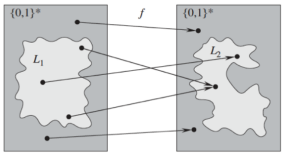
Co-NP er settet av språk slik at $\overline{L} \in$ NP. Vi har at P ⊆ NP ∩ co-NP.

Redusibilitets-relasjonen $\leq_p$

Et språk L_1 er **polynomisk-tid reduserbar** til språk L_2, betegnes med $L_1 \leq_p L_2$, hvis det eksisterer en polynomisk-tid kalkulerbar funksjon $f : \{0,1\}^* \rightarrow \{0,1\}^*$ slik at vi for $\forall x \in \{0,1\}^*$ har at $x \in L_1$ hvis og bare hvis $f(x) \in L_2$.

Vi kaller funksjonen f for **reduksjonsfunksjon**, og en polynomisk-tid algoritme F somm kalkurerer f for **reduksjonsalgoritme**. Reduksjonsfunksjonen sørger for en polynomisk-tid mapping slik at hvis $x \in L_1$, så er $f(x) \in L_2$.

Eksempel: Løser et lineært uttrykk $ax + b = 0$ med formelen for et andregradsuttrykk. Da har vi redusert det lineære uttrykket til en form hvor vi kan løse det enkelt.



NP-kompletthet og NP-hardhet

Polynomisk-tid-reduksjon hjelper oss å vise at et problem er minst like hardt som et annet. Det vil si hvis $L_1 \leq_p L_2$, så er L_1 ikke mer enn en polynomisk faktor hardere enn L_2. Vi bruker dette til å definere NP-komplette problemer.

Et språk $L \subseteq \{0,1\}^*$ er **NP-komplett** hvis

1. $L \in$ NP, og
2. $L' \leq_p L$ for every $L' \in$ NP.

Dersom et språk L tilfredsstiller krav **2**, men **ikke** 1, sier vi at L er **NP-hardt**.

Den konvensjonele hypotesen om forholdet mellom P, NP og NPC

Dersom **et** NP-komplett problem er polynomisk-tid løsbar, da er P = NP. Ekvivalent, dersom **et** problem i NP ikke er polynomisk-tid løsbar, da er **ingen** NP-komplette problem polynomisk tid løselige.

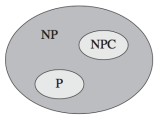


Figure 34.6 How most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained within NP, and $P \cap NPC = \emptyset$.

Hvorfor DP-løsningen til boken av 0-1 knapsack ikke er polynomisk

Den dynamisk programmerte algoritmen for 0-1 knapsack problemet har en kjøretid på $O(nW)$, hvor n er antall elementer og W er den maksimale vekten som knapsack-en kan holde. Dette er **ikke** en polynomisk-tid algoritme for noen fornuftig representasjon av input. I en fornuftig representasjon er alle numeriske verdier (vektene og verdiene, etc.) gitt i binærtall. For å representere verdien W, trenger vi $\boldsymbol{lg\ W}$ bits. Dermed blir kjøretiden $O(nW)$ eksponentiel i størrelsen til input.

NP-komplette problemer