Algoritme elementer

Mindstekrav: Ingen uendelige løkker, Korrekt output.
Kvaliteter for disse: Hastighed, Pladsforbrug, Kompleksitet (implementering), Andre egenskaber (eg. stabilitet)
For analyse kræves: Beskrivelse af problemer og maskiner (modeller), definition af kvalitet, Værktøjskasse af analyseredskaber.
Udviklingsfaser: Ord/Billeder -> Pseudo kode -> Implementering

RAM-modellen

CPU -> Hukommelse -> Basale operationer (*add, sub, mult, shift, compare, flyt, dataelemern, jump*).
Tid: Antal basale operationer udført. **Plads:** Maks antal hukommelsesceller.

Køretider

Rækkefølge: 1, log n, $\sqrt{n}$, n, n log n, $n\sqrt{n}$, n^2, n^3, n^{10}, 2^n

$f(n) = o(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \quad$ (jvf. $x < y \Rightarrow x \leq y$)

$f(n) = \Theta(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \quad$ (jvf. $x = y \Rightarrow x \leq y$)

$f(n) = O(g(n)) \;\Leftrightarrow\; g(n) = \Omega(f(n)) \quad$ (jvf. $x \leq y \Leftrightarrow y \geq x$)

$f(n) = o(g(n)) \;\Leftrightarrow\; g(n) = \omega(f(n)) \quad$ (jvf. $x < y \Leftrightarrow y > x$)

$f(n) = O(g(n))$ og $f(n) = \Omega(g(n)) \;\Rightarrow\; f(n) = \Theta(g(n))$
(jvf. $x \leq y$ og $x \geq y \Rightarrow x = y$)

Asymptotiske notationer

$f(n) = notation\big(g(n)\big) \;|\;\; O(f \leq g), \quad \Omega(f \geq g), \quad \Theta(f = g), \quad o(f < g), \quad \omega(f > g)$

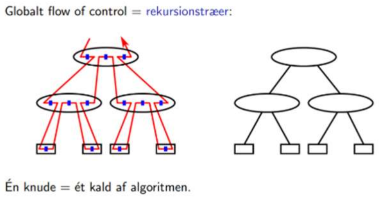
Analyse: $Hvis\ \frac{f(n)}{g(n)} \to k > 0\ for\ n \to \infty \ \ \Rightarrow f(n) = \Theta\big(g(n)\big) \;\;|\;\; Hvis\ \frac{f(n)}{g(n)} \to 0\ for\ n \to \infty \ \ \Rightarrow f(n) = \mathrm{o}\big(g(n)\big)$

$For\ alla\ a, d > 0\ og\ c > 1\ gælder \frac{(\log_c n)^a}{n^d} \to 0\ \ for\ n \to \infty$

Ethvert polynomium er $o()$ af enhver exponentialfunktion
Enhver logaritme (selv opløftet i enhver potens) er $o()$ af ethvert polynomium.

$$\frac{f(n)}{g(n)} = \frac{600n^2 + 500n + 400}{6n^3 + 5n^2 + 4n + 3} = \frac{600/n + 500/n^2 + 400/n^3}{6 + 5/n + 4/n^2 + 3/n^3} = \frac{0 + 0 + 0}{6 + 0 + 0 + 0} = 0 \quad for \quad n \to \infty \ \ \Rightarrow f(n) = o(g(n))$$

Divide-and-Conquer



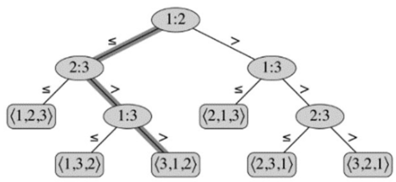
Generel algoritme-udviklingsmetode, som opdeler problemer i mindre delproblemer af samme type, løser dem via rekursive kald og til sidst kombinere alle delproblemerne ind til en løsning på problemet.

Køretider: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Beskrevet nærmere: Find højden, sum hvert lag i træet sammen for sig, herefter sum de resulterende værdier for lag.

Decision trees

Labels: ID'er (indeks for input der sammenlignes)
Label blade: Svaret når algoritmen stopper, som er hvilken opstilling som skal laves for at få sorteret orden, angivet med liste af oprindelige ID'er.
Køretid: Længste rod-blad sti = træest højde
Algoritmer der kan ses sådan:
insertionsort, selectionsort, mergesort, quicksort, heapsort



Figur 1: Sammenligningsbaserede sorteringsalgoritmer

Nedre grænse: Der er $n!$ Inputs / rækkefølger af sortering. Derved skal der min. Være det antal blade.
Et træ med højde h, har højst 2^h blade. Derved $2^h \geq antal\ blade \geq n!\ \Rightarrow h \geq \log(n!) = \frac{n}{2}(\log(n) - 1)$
Wors-case køretid = træets højde: $\Omega(n \log n)$

Invarianter

1) Invariant overholdt i starten 2) Invariant overholdt før et skridt ⇒ overholdt efter	⇒	Invariant altid overholdt

Et forhold, som vedligeholdes af algoritmen gennem (dele af) dens udførelse. Udgør ofte kernen af ideen bag algoritmen.

Sorterings algoritmer

Sortere n tal, så de er i en form for ordnet rækkefølge. Eg. **Stigende**. Central og fundamental opgave.
Altid liggende i liste/array.

	Worstcase	Inplace
QuickSort		√
MergeSort	√	
HeapSort	√	√

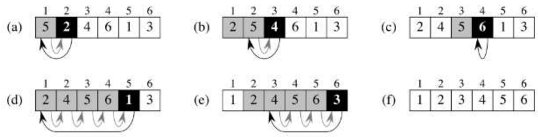
Figur 2: n log n køretids algoritmer. heap er langsomst (random access) Introsort: (kombi quick & heap)

Insertionsort

Sortere fra venstre mod højre, og indsætter det i det allerede sorteret del af elementet

Som pseudo-kode:

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```



```
INSERTION-SORT(A, n)                                   cost  times
for j = 2 to n                                          c1    n
    key = A[j]                                          c2    n − 1
    // Insert A[j] into the sorted sequence A[1 .. j − 1].  0  n − 1
    i = j − 1                                           c4    n − 1
    while i > 0 and A[i] > key                          c5    Σ_{j=2}^{n} t_j
        A[i + 1] = A[i]                                 c6    Σ_{j=2}^{n} (t_j − 1)
        i = i − 1                                       c7    Σ_{j=2}^{n} (t_j − 1)
    A[i + 1] = key                                      c8    n − 1
```

Køretid: $$\sum_{j=1}^{n} j = (1+2+3+\cdots+n) = \frac{(n+1)n}{2} = \frac{n^2+n}{2} \leq \frac{2n^2}{2} = n^2.$$

Selectionsort

Lineær søgning. *Indliste* og *udliste*, fra udlisten tages den mindste altid og proppes i indliste sidste plads. Køretid: n^2

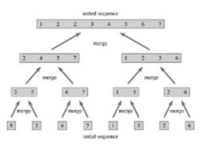
$\text{tid} \leq c \cdot (n + (n-1) + (n-2) + \cdots + 1) \leq c \cdot n^2.$

```
IndListe = input
UdListe = tom liste
While IndListe ikke tom:
    find mindste element x i IndListe
    flyt x fra IndListe til enden af UdListe
```

Mergesort

Tager udgangspunkt i, at det er hurtigere at flette to lister, som allerede er sorteret sammen. Divide-and-conquer.
Minder om selectionsort, men splitter ud, for at få mindre lister.

Køretid: $\leq c \cdot n$ per lag, $c \cdot n \cdot \log_2 n$ for det hele. $\log_2 n$ splits



k	Antal lister
⋮	⋮
k	$n/2^k$
⋮	⋮
3	$n/2^3$
2	$n/2^2$
1	$n/2$
0	n

```
MERGE-SORT(A, p, r)
if p ≥ r                              // zero or one element?
    return
q = ⌊(p + r)/2⌋                       // midpoint of A[p : r]
MERGE-SORT(A, p, q)                   // recursively sort A[p : q]
MERGE-SORT(A, q + 1, r)               // recursively sort A[q + 1 : r]
// Merge A[p : q] and A[q + 1 : r] into A[p : r].
MERGE(A, p, q, r)
```

Som pseudo-kode, med to rækker $A[p \ldots q]$ and $A[q+1 \ldots r]$:

```
MERGE(A, p, q, r)
n1 = q − p + 1
n2 = r − q
let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = ∞
R[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else A[k] = R[j]
        j = j + 1
```

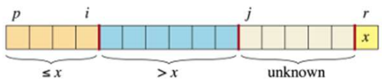
Quicksort

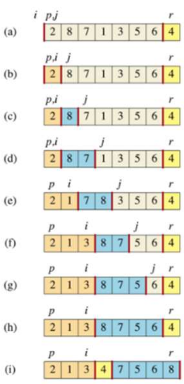
Bygger på partition og minder om merge sort. - Den op i to dele (X og Y, $X \leq Y$), sorter hver for sig rekursiv, returner X efterfulgt af Y.Partition opdeler således:
$A[q] = x \mid A[p \ldots q-1] \leq x \mid A[q+1 \ldots r] > x$

```
QUICKSORT(A, p, r)
if p < r
    // Partition the subarray around the pivot, which ends up in A[q].
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)   // recursively sort the low side
    QUICKSORT(A, q + 1, r)   // recursively sort the high side
```

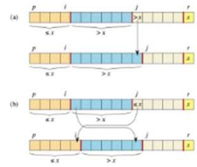
Partition

Opel et array i større eller mindre dele, an på et element x valgt.
Tag ukendt område og flyt hen på i, hvis under eller efter i hvis større. Flyt første element i større end listen til enden af større end listen, hvis det undersøgte element er mindre end x.





```
PARTITION(A, p, r)
x = A[r]                          // the pivot
i = p − 1                         // highest index into the low side
for j = p to r − 1                // process each element other than the pivot
    if A[j] ≤ x                   // does this element belong on the low side?
        i = i + 1                 // index of a new slot in the low side
        exchange A[i] with A[j]   // put this element there
exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
return i + 1                      // new index of the pivot
```



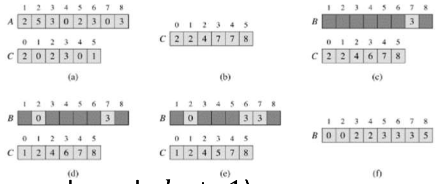
Køretid: $O(n \cdot \log n)$ (balanceret) $O(n^2)$ (ubalanceret)
Kald hendholdsvis: balanceret $\frac{n-1}{2}$ u. balanceret: $0, n-1$
Plads: Inplace (bruge ikke mere plads)

Heapsort

Heap: binært træ, heap-orden, heap-facon, i array. *Det er ikke heap som i hukommelse*.
En form for selectionsort hvor der bruges en heap til hele tiden at udtage det største tilbageværende element. Tid: $O(n \cdot \log n)$

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

Counting sort



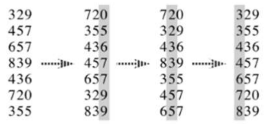
Elementer bruges som index (derfor de skal være heltal)
Sortere n heltal af størrelse mellem 0 og k (inkl.)
Tre arrays. A (In, længde n), B (Out, længde n), C (tæller array, længde $k + 1$)
C arrayet indeholder den sidste plads i B arrayet, det tal skal have. Når tallet (array index i C) bemærkes i A, så indsættes den, på pladsen i B, som svare til C. Herefter dekrementes værdien i C.
Tid: $O(n + k)$ - Denne er stabil, dvs. at elementer med ens værdier beholder deres indbyrdes plads.

```
COUNTING-SORT(A,B,k)
  for i = 0 to k
    C[i] = 0
  for j = 1 to A.length
    C[A[j]] + +
  for i = 1 to k
    C[i] = C[i] + C[i − 1]
  for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] − −
```

Radix sort

Sortere n heltal med d cifre i base (radix) k.
Aka. Cifre i heltal i: $\{0,1,2, \dots, k-1\}$.
Tid: $O(d(n+k))$ (ved brug af counting sort i løkken)
Efter hver iteration, så er alle cifre på højre side sorteret (korrektheden)
Eksempler: $2\ cifrede\ tal,\ base\ 10^6,\ Tid: O\big(2(n+10^6)\big),\ O(n)\ hvis\ n \geq 10^6$
$4\ cifrede\ tal,\ base\ 2^8,\ Tid: O\big(4(n+2^8)\big),\ O(n)\ hvis\ n \geq 256$

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

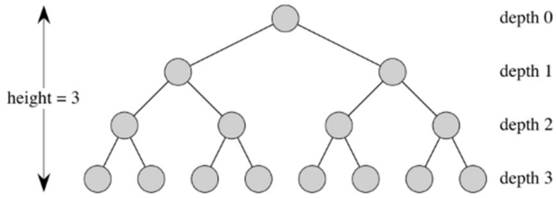


Datastruktur

Datastruktur: data + operationer | $Data = (Id\ (nøgle),\ extra\ data)$ - Der refereres ofte kun til ID
Datastrukturens egenskaber udgøres af operationerne og køretiden. Målet er fleksibilitet og effektivitet (modstridende)
Fungere som et API (niveau 1: operationer | niveau 2: implementering)
Trivielle operationer for alle: $CreateNewEmpty,\ RemoveEmpty,\ isEmpty$

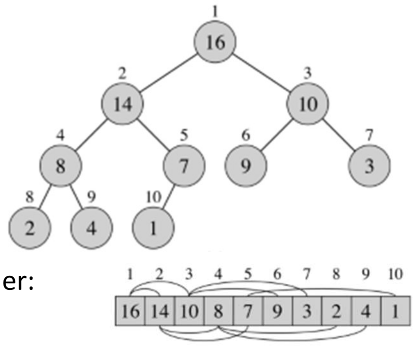
Binært træ

Er enten det tomme træ, eller en knude v, med to undertræer (højre, venstre). V med som har undertræer kaldes **rod**, mens en der ikke har, er et **blad**. Forbindelse mellem knuder er **kanter**. Dybde: Antal kanter til rod. Højde af knude: max antal kanter til blad. Højde af træ: højde af dets rod. Fuldt (complete) binært træ: Træ med alle blade i samme dybde.



Knude tal, for fuldt træ i højde h: $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ (Formal A.5) - Heraf 2^h blade.

Heap



Heap-orden

Max-heaporden: $v \geq u$ (rod størst), Min-heaporden: $v \leq u$ (rod mindst), hvor u er børn til v. Dubletter er tilladt

Heap-facon

Alle lag i træet er fyldte, undtagen det sidste lag, hvor alle blade er fyldt ud fra venstre side. (Fuldt træ har heapfacon). Heapfacon træ, af højde h med n knuder:

$$n > antal\ knuder\ i\ fuldt\ træ\ af\ højden\ h - 1 = 2^h - 1$$

$$n > 2^h - 1 \Leftrightarrow n + 1 > 2^h \Leftrightarrow \log_2(n+1) > h$$

Heap array form

Top down, venstre til højre. Navigering kan dernæst gøres med: Forældre: $\left\lfloor \frac{i}{2} \right\rfloor$.
Børn på plads $2i, 2i + 1$. i = knude position i arrayet.

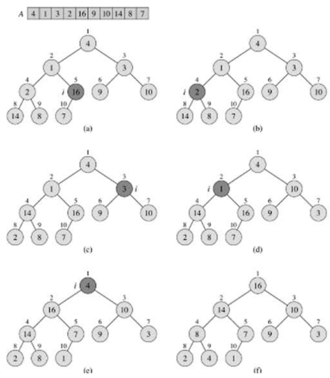
```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

```
BUILD-MAX-HEAP(A, n)
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i, n)
```

Heap operationer

Max-Heapify

Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden. Aka. Knudens nøgle kan være minde end en af sine børns. Løses ved at bytte nøgle med barned med den største nøgle, hvorefter der køres *Max-Heapify* på den. Tid: $O(højde\ af\ knude)$



Build-Max-Heap

Lav n input elementer (uordnede) til en heap. For at lave træet i heap-facon, for efterfølgende at sortere det i heap-facon. Køretid: $O(n \cdot \log_2 n)$, kan også være $O(n)$ ved bedre analyse.

```
HEAPSORT(A, n)
  BUILD-MAX-HEAP(A, n)
  for i = n downto 2
      exchange A[1] with A[i]
      MAX-HEAPIFY(A, 1, i − 1)
```

Prioritetskøer

	Heap	Usorteret liste	Sorteret liste
EXTRACT-MAX	$O(\log n)$	$O(n)$	$O(1)$
BUILD	$O(n)$	$O(1)$	$O(n \log n)$
INCREASE-KEY	$O(\log n)$	$O(1)$	$O(n)$
INSERT	$O(\log n)$	$O(1)$	$O(n)$

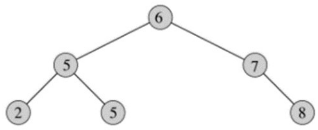
Data: Element = Nøgle (id)
Centrale operationer (max-version):
Q.Extract-Max: Returner største nøgle i Q og fjern det fra listen
Q.Insert(e: element): Tilføj element e til Q | Ud fra disse to, kan man lave sortering.
Q.Increase-Key(r: reference til element i Q, k: nøgle): Ændre nøglen til max{k, gamle nøgle} for elementer refereret til af r
Q.Build(L: liste af elementer): Bygger en prioritetskø indeholdende elementer i listen L
Heap implementering (Extract-Max & Build findes).
Increase key: Ændre nøgle for element, genopret heaporden, hvis elementet er større end forældre skift plads (Tid: Højde af træ $O(\log n)$).
Insert: Indsæt det nye element sidst, således heap-facon er i orden. Derefter genopret heaporden, som ovenfor (Tid: Højde af træ $O(\log n)$).

Dictionaries

Search(key): returnere elementer med nøglen key, Insert(key), Delete(key)
Kræver ordnet keys: Predecessor(key): Find elementet med højeste nøgle < key, Sucessor(key): Find elementet med laveste nøgle > key, Orderedtraversal(): Udskriv elementer i sorteret orden.
Uordet nøgler: Unordered dictionary, ordret nøgler: Ordered dictionary | Java: Map interface, Python: dict
Implementeringer vi møder: Balancerede binære søgetræer (TreeMap) (understøtter endnu flere operationer) i $O(\log n)$ tid. Hashing (HashMap), som understøtter Unordered dictionaries i $O(1)$

Ubalanceret binære søgetræer

Binært træ, med knuder i inorder (venstre træ er altid lavere end højre træ, mens roden er imellem de to - Inorder gennemløb i denne rækkefølge).
$nøgler\ venstre\ undertræ \leq nøgle\ i\ rod \leq nøgler\ i\ højre\ undertræ$
Normalt har et knude reference til forældre, venstre og højre undetræ.
Inorder gennemløb vil resultere i sorteret orden: $O(n)$ køretid, ved $O(1)$ arbejde pr. knude.



```
INORDER-TREE-WALK(x)
 if x ≠ NIL
     INORDER-TREE-WALK(x.left)
     print key[x]
     INORDER-TREE-WALK(x.right)
```

Søgning

Tree-Search Princip: Hvis søgte element findes, er det i det undertræ, vi er kommet til
Andre typer: Tree-Maximum, Tree-Minimum & Tree-Successor
Tree-Sucessor princtip: Se på Se på stien fra x til rod. Ingen side-træer på den kan indeholde det søgte element (pga. in-order).

```
TREE-SEARCH(x, k)
 if x == NIL or k == key[x]
     return x
 if k < x.key
     return TREE-SEARCH(x.left, k)
 else return TREE-SEARCH(x.right, k)
```

```
TREE-SUCCESSOR(x)
 if x.right ≠ NIL
     return TREE-MINIMUM(x.right)
 y = x.p
 while y ≠ NIL and x == y.right
     x = y
     y = y.p
 return y
```

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

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

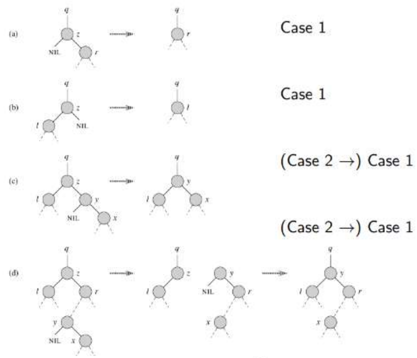
Indsætning

Søg nedad fra rod og indsæt de nye elementer efter inorder-krav. Ved blad så indsæt ny knude.

Sletning

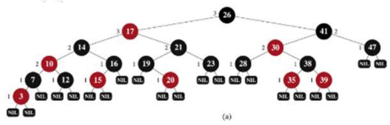
Slet z | Case 1: Mindst et barn er et NIL, fjern knude z, og lad barnet tage pladsen. |Case 2: Ingen NIL børn. Successor-knuden y, er den mindste knude i z højre under-træ. Erstat z med denne.

Tider: $O(højde) = O(\log_2 n\,(\pm 1))$



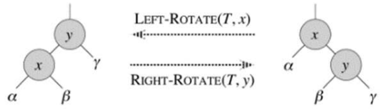
Balanceret binære søgetræer (Rød-sorte træer)

Dette er for at opretholde $O(\log n)$ tider. Balanceinformation i 1 bit (rød-sort)
VIGTIGT: ikke to røde i træk (rod-blad), samme antal sorte på alle stier, rod og blade er sorte.
Min højde: $2^k - 1$, Maks højde: $2^{2k} - 1$

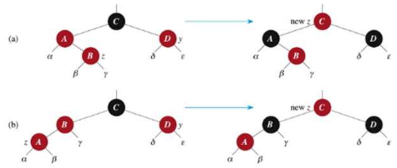


Indsætning

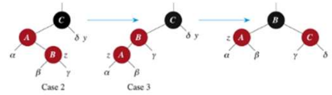
Indsætning kræver rebalancering. Ved rebalancering skubbes rød-rød problemet længere op i træet.
Rotation: Left og right. Se illustration.
Efter rotation, skal der måske rebalanceres - Se billede ude til højre.



Case 1: Rød onkel (onkel = forælders søskend).



Case 2: Sort onkel (onkel = forælders søskend).

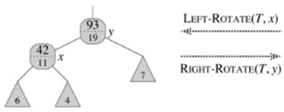


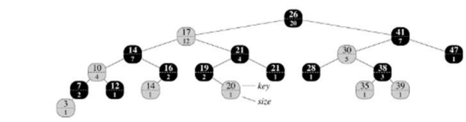
Figur 3: Indsættelses rebalancering

Sletning

Steps: Slet knude og opret ubalance
Fjernet rød: Rød-sort krav overholdt | Fjernet sort: Ikke længere samme antal sorte på stier og skal rebalanceres
Se evt. script

Ekstra data i knude





```
OS-RANK(T, x)
 r = x.left.size + 1
 y = x
 while y ≠ T.root
     if y == y.p.right
         r = r + y.p.left.size + 1
     y = y.p
 return r
```

```
OS-SELECT(x, i)
 r = x.left.size + 1
 if i == r
     return x
 elseif i < r
     return OS-SELECT(x.left, i)
 else return OS-SELECT(x.right, i - r)
```

Hashing

Keys af heltal op til max-grænse k. Fungere på mange måder som countingsort. Men vi skrumper universet af keys, for at spare plads. Vi tager en hashing funktion, som skrumper tallet. $h : U \to \{0,1,\dots,m-1\}, m = O(n)$

Eksempel: $h(x) = x \ mod \ m$ | Det skal noteres at der er kolisioner. Eksempler: Chaining, Open adressing (Linear hashing, double hashing), Universal hashing. Chaining giver $\Theta(n)$, men i praksis $O(1)$, men kan gå op til $O(\log n)$ via balancerede søgetræer. Sletning i open adressing er besværlig, og derved slettes der ikke, men markeres blot "til sletning" (rebuild en gang imellem). Dertil i open adressing, så øges i, indtil element eller ledig plads findes.

Linear hashing:
$h(k,i) = (h'(k) + i) \mod m$

Double hashing
$h(k,i) = (h'(k) + i \cdot h''(k)) \mod m$

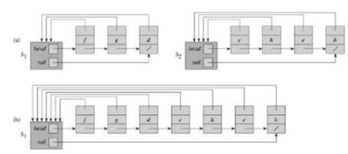
Insert: $i = 0,1,2,\dots$ forsøges til en empty slot findes.
Search: $i = 0,1,2,\dots$ forsøges til element eller empty slot findes.

Universal hashing: $h(k) = \big((a \cdot k + b) \ mod \ p\big) \ mod \ m, \ p \ (primtal) > |U|, 1 \le a \le p-1, 0 \le b \le p-1$

Disjoint sets

En Partition (disjunkt opdeling) af en mængde S er en samling ikke-tomme delmængder $A_i, i = 1,\dots,k$ som er disjunkte og tilsammen udgør S: $A_i \neq \emptyset \ for \ alle \ i, \quad A_i \cap A_j = \emptyset \ \ for \ i \neq j, \quad a_1 \cup a_2 \cup \dots \cup A_k = S$

Eksempel: $\{a,b,e\},\{f\},\{c,d,g,h\}$ er en partition af $\{a,b,c,d,e,f,g,h\}$



Operationer

$Make\_Set(x)$: opret $\{x\}$ som en mængde.
$Union(x,y)$: Slå $\{a,b,\dots,x\}$ og $\{h,i,\dots,y\}$ sammen til $\{a,b,\dots,x,h,i,\dots y\}$
$Find\_Set(x)$: Returner en ID for mængden indeholdende x.

- FIND-SET(x): returner pointer til header: $O(1)$.
- MAKE-SET(x): opret ny liste: $O(1)$.
- UNION(x,y): slå lister sammen, behold én header, ændre alle header-pointere i den anden liste: $O(n)$.

Så bedre analyse: n MAKE-SET, op til $n-1$ UNION, og m FIND-SET koster $O(m + n \log n)$.

Pseudokode for implementering via træer:

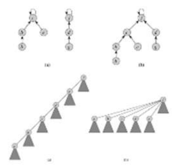
Pseudokode (med union by rank og path compression) er simpel:

```
MAKE-SET(x)                UNION(x, y)
  x.p = x                    LINK(FIND-SET(x), FIND-SET(y))
  x.rank = 0

FIND-SET(x)
  if x ≠ x.p
      x.p = FIND-SET(x.p)
  return x.p

LINK(x, y)
  if x.rank > y.rank
      y.p = x
  else x.p = y
      // If equal ranks, choose y as parent and increment its rank.
      if x.rank == y.rank
          y.rank = y.rank + 1
```

- FIND-SET(x): gå til rod.
- MAKE-SET(x): opret nyt træ.
- UNION(x,y): gør rod af ét træ til barn af andet træ.



Master theorem

$\leq$	$\geq$	$=$	$<$	$>$
O	Ω	Θ	o	ω

Rekursionsligning: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ Følgende løsninger ($\alpha = \log_b a$):

1) $hvis\ f(n) = O(n^{\alpha-\epsilon})\ for\ \epsilon > 0\ \ så\ \ T(n) = \Theta(n^{\alpha})$

2) $hvis\ f(n) = \Theta\big(n^{\alpha}(\log n)^{k+1}\big)\ for\ k \geq 0\ \ så\ \ T(n) = \Theta\big(n^{\alpha}(\log n)^{k+1}\big)$

3) $hvis\ f(n) = \Omega(n^{\alpha-\epsilon})\ for\ \epsilon > 0\ \ så\ \ T(n) = \Theta(f(n))$ (Kræver $c < 1$ og n_0 som opfylder $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ når $n \geq n_0$)

Følgende ville ikke kunne løses: $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$ (Der er for mange led)

Som eksempel på en rekrusionsligning er mergesort. Ved ulige n, ville det med mergesort derved faktisk være:

$T(n) = 2T\left(\frac{n}{2}\right) + n \iff T(n) = T\left(\left\lceil\frac{n}{2}\right\rceil\right) + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + n$ DETTE HAR INGEN EFFEKT (floor/ceiling)

Dynamisk programmering

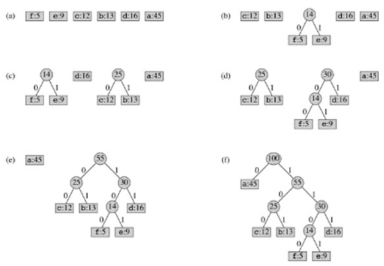
En metode til at udvikle algoritmer til kombinatoriske optimeringsproblemer. F.eks.: rute, pakning, undervisningsskema. Kan forklares som et specialtilfælde af Divide-and-Conquer. Ved brug af lagring (table, caching), går vi fra eksponentiel til polynomiel tid.

Grådige algoritmer

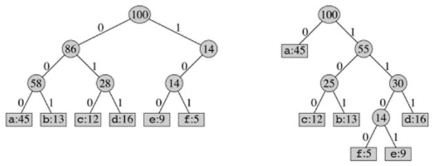
Tager det bedste valg nu og her. Håber at lokal optimering giver global optimering.

Huffmans algoritme

Variable længde encoding. Vigtigt element.
Der bygges nedefra, med de mindste elementer.
Tid: $O(n \log n)$



	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

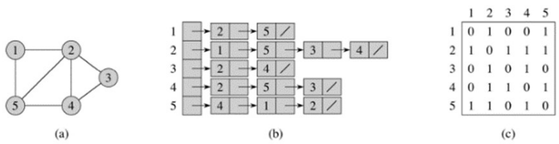


Grafer

Orienterede grafer: Kander er ordnede par | Uorienterede grafer: Kanter er uordnede par | Vægtede grafer: Hver kant har en tal/vægt tilknyttet | Vertices = knude| Mængde V af knuder | Edge = kant | $E \in V \times V$ par af knuder

Eksempler: Ledningsnet, vejnet, venner, WWW

Adjanceny lists / matrix



Listen (b) for *u* (grafen) indeholder *v* for alle kanter $(u, v) \in E$

Knuder er heltal mellem 1 og n (0, n-1) | Plads: $O(n+m)$

Der bruges primært lists, medmindre andet er specificeret. Plads for ***matrix*** $O(n^2)$

Uorienterede graf er specieltilfælde af orienterede grafer

Grafgennemløb

Hvid: Ikke besøgt | Grå: I gang | Sort: Done |Generel tid: $O(n+m)$ ved $O(1)$ valg af grå knude.

BFS: $O(n+m)$ (optimal) $v.d = distance$ | DFS: $O(n+m)$ $v.d = discovery\ time, u.f = finish\ timestamp,$ $v.d \rightarrow v.f = tid\ på\ stak$ | Hvid-sti lemma: $u.d < w.d < w.f < u.f$

1. *tree-kanter:* *v* hvid.
2. *back-kanter:* *v* er grå (er på stak).
3. *forward-kanter:* *v* er sort (den er ikke længere på stak, men har været det sammen med *u*).
4. *cross-kanter:* *v* er sort (den er ikke længere på stak, og har ikke været det sammen med *u*).

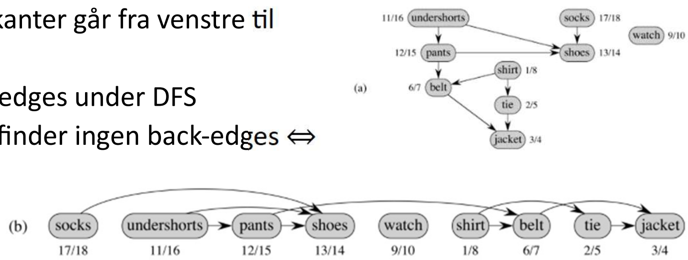
DAG (Directed Acyclic Graph) & topologisk sortering

Orienteret graf uden kredse (cycles) - Afhængigheds modellering

Topologisk sortering -> Lineær ordning af knuderne så alle kanter går fra venstre til højre.

Lemma: En orienteret graf har en kreds $\Leftrightarrow$ Der findes back-edges under DFS

Korollar til to foregående lemmaer: Graf er en DAG $\Leftrightarrow$ DFS finder ingen back-edges $\Leftrightarrow$ ordning af knuder efter faldende finish-tider giver en topologisk sortering.



Topologisk sortering køretid: $O(n+m)$

```
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
```

Ækvivalensrelationer

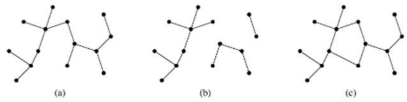
- $x \sim x$.
- $x \sim y \Rightarrow y \sim x$.
- $x \sim y \wedge y \sim z \Rightarrow x \sim z$.

Relation: R | x står i relation til y | relation: $\sim$ | $v \sim u$ → Er en uorienteret sti mellem u og v
Sammenhængskomponenter (CC) er en partition af knuder (uorienteret grafer)
Kan findes via generel traversal - Tid: $O(n+m)$
Stærke sammenhængskomponenter (SSC) (orienteret grafer)
Kan findes via SCC - Tid: $O(n+m)$

```
SCC(G)
  call DFS(G) to compute finishing times u.f for all u
  compute G^T
  call DFS(G^T), but in the main loop, consider vertices in order of decreasing u.f
      (as computed in first DFS)
  output the vertices in each tree of the depth-first forest formed in second DFS
      as a separate SCC
```

Træer

Frit/u-rodet træ er en uorienteret graf, som er sammenhængende og acyklisk



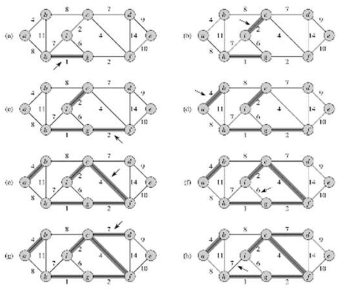
Træ Skov Graf med kreds (ikke træ)

Minimum Spanning Tree (MST)

Udspændende træ for uorienteret, sammenhængende graf, er der en delgraf som er et træ (E'). Hvis det er vægtet, er sum af kantvægte mindst lige så meget for udspændt træ. MST indeholder n-1 kanter.
Algoritme: Prim-Jarnik MST (Prim 1957, Jarnik 1930), hvor r (startknude) udvides i form af sammenhængskomponent. Køretid: $O(m \log n)$
Algoritme: Kruskal MST (1956), hvor kanter bliver tilføjet i global letteste-første-roden
Køretid: $O(m \log m)$

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

PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)      // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```



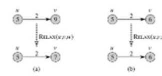
```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

Korteste veje i vægtede grafer

Længden af sti: Sum af vægten på kanterne stien består af | Højest n! for stier uden kredse
Relaxation: Brug kanter til at udbrede information fra knude til knude om længder af stier

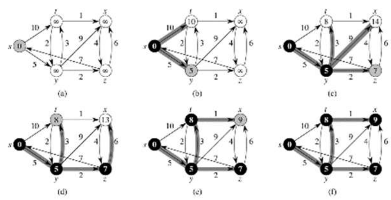
```
INIT-SINGLE-SOURCE(G, s)
  for each v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0

RELAX(u, v, w)
  if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```



Dijkstras (1959)

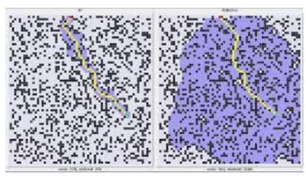
Grådig algoritme, som trinvis bygger bedre $v.d$ og $v.\pi$ | Prioritetskø bruges | $vægt \geq 0$
Tid: $O(m \cdot \log n)$



```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V        // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

A* (1968)

Heuristik. $h(v)$ - Admissible $h(v) \leq faktiske\ vej$
Consistent $\Rightarrow$ Admissible $\Rightarrow$ korrekt (med genindsættelse i PQ).
Consistent $\Rightarrow$ korrekt (uden genindsættelse i PQ).



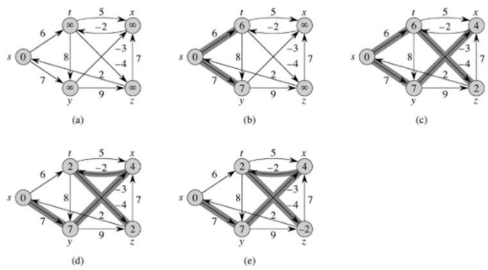
DAG shortest paths

Topologisk search | Tid: $O(n+m)$
Kan klare negative vægtninger (ikke negative kredse)
Tid: $O(n+m)$

```
DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices
  INIT-SINGLE-SOURCE(G, s)
  for each vertex u, taken in topologically sorted order
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Bellman-Ford-Moore [1956-57-58]

Tid: $O(nm)$



```
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| - 1
      for each edge (u, v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
          return FALSE
  return TRUE
```

Korteste veje mellem alle par af knuder

Floyd-Warshalls algoritme (1962)

Burger dynamisk programmering igennem adjacency-matrix

Tid: $O(n^3)$ Plads: $O(n^2)$

FLOYD-WARSHALL(W, n)

$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Figur 4: Kun konstruktion af D-matricen vises

Johnson algoritme (1977)

Kører Bellman-Ford-Moore en gang på led udvidet graf, hvorefter den kustere kantvægte så alle bliver positive uden essentielt at ændre korteste veje. Til sidste køres Dijkstra.

Tid: $O(nm \log n)$

Revægtning: For alle knuder tildeles en nyt tal $Ø(v)$. Herfra kan vi lave vægte: $\widetilde{w}(u, v) = w\big(u, v + Ø(v) - Ø(u)\big)$

Se på en sti $v_1, v_2, \ldots, v_k$. Da gælder

$$\begin{aligned}\sum_{i=1}^{k-1} \tilde{w}(v_i, v_{i+1}) &= \sum_{i=1}^{k-1}(w(v_i, v_{i+1}) + \phi(v_{i+1}) - \phi(v_i)) \\ &= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + (\phi(v_k) - \phi(v_1))\end{aligned}$$

Matematiske elementer

$\frac{n^2}{k} = 1 \Leftrightarrow n = 2^k \Leftrightarrow \log_2 n = k$ $\quad (\log n)^a \Leftrightarrow \log^a n$ $\quad \frac{k}{n^a} = 0$ $\quad \frac{n}{b^a} = 1 \Leftrightarrow b^a = n \Leftrightarrow a = \log_b n$

$a^{\log_a(n)} = n$ $\quad (a^b)^c = a^{bc} = (a^c)^b$ $\quad a^{\log_b n} = n^{\log_b a}$ $\quad \log_b a = \frac{\log_c a}{\log_c b}$

For $a > 0$ og $b > 1$ gælder $\frac{n^a}{b^n} \to 0\ for\ n \to \infty$ Dvs. ethvert polynomium er $o()$ af enhver exponentialfunktion

Følgende gælder for $c > 1$: $1 + c + c^2 + \cdots + c^k = \frac{c^{k+1}-1}{c-1} = c^k \cdot \frac{c-\frac{1}{c^k}}{c-1} = \Theta(c^k)$
Aka: Hvis elementerne i en sum ændrer sig eksponentielt, så er hele summen domineret af det største led.

Potens regel: $2^k \cdot 2^n = 2^{k+n}$

Diskret matematik

(Precedens hieraki) Rækkefølge: *(kvantore)* ¬ ∧ ∨ ⇒⇔ ⊕
Kvantore kommer før den normale rækkefølge

1+1=2 ∧ 3>5	F	$\wedge\ = og$	p $\wedge q$: Begge er sande
1+1=2 ∨ 3>5	S	$\vee\ = eller$	p $\vee q$: Mindst en er sand

p q	$p \wedge q$	$p \vee q$
S S	S	S
S F	F	S
F S	F	S
F F	F	F

Negering: ı ! -
Og: & ∧ (konjunktion)
Eller: | | ∨ (disjunktion)

Implikation / betinget udsagn

P medfører q | Hvis P, så q
Hvis man ikke kan fanges i løgn, så er det sandt

P	Hypotesen / antagesen
Q	Konklusionen / konsekvensen

p q	p ⇒ q
S S	S
S F	F
F S	S
F F	S

Kan også skrives som (ækvivalens):
$p \Rightarrow q \equiv \neg p \vee q$
$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$

Bi implikation:
p er ensbetydende med q
$p \Leftrightarrow q$

p q	p ⇔ q
S S	S
S F	F
F S	F
F F	S

Kan også skrives som (ækvivalens):
$p \Leftrightarrow q \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$
Andre:
$p \Leftrightarrow q \equiv (p \Rightarrow q) \vee (q \Rightarrow p)$
$p \Leftrightarrow q \equiv (p \Rightarrow q) \vee (\neg p \Rightarrow \neg q)$

XOR:
$p \oplus q$

p q	p ⊕ q
S S	F
S F	S
F S	S
F F	F

Tautologi (altid sandt)
Sammensat udsagn, som er sandt,
uanset sandhedsværdien af de udsagnsvariable, som indgår
Eksempelvis: $p \vee \neg p$

Modstrid (altid falsk)
Sammensat udsagn, som er falsk,
uanset sandhedsværdien af de udsagnvariable, som indgår
Eksempelvis: $p \wedge \neg p$

Kontingens
Hverken tautologi eller modstrid.
D.v.s. alt andet end Tautologi og Modstrid
Eksempelvis: $p \vee q$

$\mathbb{Z}$	$\dots, -2, -1, 0, 1, 2, \dots$	Heltal
$\mathbb{Z}^+$	1,2,3,…	Positive heltal
$\mathbb{Z}^-$	0,1,2,…	Naturlige tal
$\mathbb{N}$	0,1,2,… *(Nogle starter fra 1)*	Naturlige tal
$\mathbb{Q}$	$\frac{m}{n} \mid m \in \mathbb{Z},\ n \in \mathbb{Z}^+$	Rationale tal
$\mathbb{R}$		Reelle tal

Ækvivalenser

$p \Rightarrow q \equiv \neg p \vee q$
$p \Rightarrow q \equiv \neg q \Rightarrow \neg p$
$p \Leftrightarrow q \equiv (p \wedge q) \vee (\neg p \wedge \neg q)$
$p \Leftrightarrow q \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$
$p \Leftrightarrow q \equiv (p \Rightarrow q) \vee (\neg p \Rightarrow \neg q)$

De Morgan (love)
$\neg(p \wedge q) \equiv \neg p \vee \neg q$
$\neg(p \vee q) \equiv \neg p \wedge \neg q$
$\neg\forall x : p(x) \equiv \exists x : \neg p(x)$
$\neg\exists x : p(x) \equiv \forall x : \neg p(x)$

De Morgan (love) for kvantorer

Åbent udsagn (propositional function)

$P(x): 2x > x$ $(x\ er\ fri\ variabel)$
$P(2): 2 * 2 > 2 \qquad S$
$P(1): 2 * 1 > 1 \qquad S$
$P(0): 2 * 0 > 0 \qquad F$
$P(-1): 2 * -1 > -1\ F$

$P(x, y):\ x + y = 0$
$P(2,\ -2) \equiv S$
$P(2,0) \qquad \equiv F$

TABLE 1 Quantifications of Two Variables.		
Statement	*When True?*	*When False?*
$\forall x \forall y P(x, y)$ $\forall y \forall x P(x, y)$	$P(x, y)$ is true for every pair x, y.	There is a pair x, y for which $P(x, y)$ is false.
$\forall x \exists y P(x, y)$	For every x there is a y for which $P(x, y)$ is true.	There is an x such that $P(x, y)$ is false for every y.
$\exists x \forall y P(x, y)$	There is an x for which $P(x, y)$ is true for every y.	For every x there is a y for which $P(x, y)$ is false.
$\exists x \exists y P(x, y)$ $\exists y \exists x P(x, y)$	There is a pair x, y for which $P(x, y)$ is true.	$P(x, y)$ is false for every pair x, y.

Kvantore

Alkvantor - For alle: $\forall x \in \mathbb{Z}^+ : P(x)$
$\forall\ = alkvantor = for\ alle \qquad \mathbb{Z}^+ = universet \qquad : \ = \ $ gælder
$x = Bunde\ variabel \qquad$ Til sammen er det et udsagn

„For alle x i mængden af positive heltal gælder 2x>x"
„For alle positive heltal x gælder 2x>x"
„For ethvert positivt heltal x gælder 2x>x"

$Q(x): 2x > x + 4 \quad => \quad Q(5) \wedge Q(6) \wedge Q(7) \wedge \ldots$

Eksistenskvantor - $\exists x \in \mathbb{Z}^+ : Q(x)$
$\exists = Der\ eksistere = Eksistenskvator \qquad := Sådan\ at$
$\exists x \in \mathbb{Z} : 2x > x + 4$

Negering:

$\neg\forall x \in \mathbb{Z} : 2x > 4 \quad \equiv \quad \exists x \in \mathbb{Z} : \neg(2x > 4) \quad \equiv \quad \exists x \in \mathbb{Z} : 2x \leq 4$
$\neg\exists x \in \mathbb{Z} : 2x > 4 \quad \equiv \quad \forall x \in \mathbb{Z} : \neg(2x > 4) \quad \equiv \quad \forall x \in \mathbb{Z} : 2x \leq 4$
$\neg\exists x \in \mathbb{Z} : \forall y \in \mathbb{Z} \ : x > y \ \equiv \forall x \in \mathbb{Z} \ : \exists y \in \mathbb{Z} \ : \neg(x > y)$

$\exists!\, x P(x) =$ "There exists a unique x such that P(x) is true"

Indlejrede kvantorer: $\forall x \in \mathbb{Z} : \exists y \in \mathbb{Z} : x + y = 0$ (sandt)
(For hvert x, find et y som passer (her $y = -x$))
For hvert $x \in \mathbb{Z}$: Find et $y \in \mathbb{Z}$, så: $x + y = 0$
Orden har stor betydning: $\exists y \in \mathbb{Z} : \forall x \in \mathbb{Z} : x + y = 0$ (falsk)
(Vælg et y, som passer med alle x)

$\forall s \in S : \exists h \in H : P(s, h)$ = Alle studerende i dette lokale har en hobby
$\exists h \in H : \forall s \in S : P(s, h)$ = Der er en hobby, som alle stud. i dette lokale har
$\forall x \in \mathbb{Z} : \forall y \in \mathbb{Z} : (x > y \Rightarrow x \geq y + 1) \equiv \ \forall x, y \in \mathbb{Z} : (x > y \Rightarrow x \geq y + 1)$
$\exists x \in \mathbb{Z} : \exists y \in \mathbb{Z} : (x > y \Rightarrow x \geq y + 1) \equiv \exists x, y \in \mathbb{Z} : (x > y \Rightarrow x \geq y + 1)$
$\neg\forall s \in S : \exists h \in H : P(s, h) =$ Der er en studerende i dette lokale, som ikke har nogen hobby
$\neg\exists h \in H : \forall s \in S : P(s, h) =$ Der er ingen hobby, som deles af alle studerende i lokalet

Beviser

Direkte bevis

Benytter, at $(p \Rightarrow p_1 \Rightarrow p_2 \ldots \Rightarrow p_n \Rightarrow q) \Rightarrow (p \Rightarrow q)$

Direkte bevis
n ulige $\quad$ n^2 ulige
Benytter, at $(p \Rightarrow p_1 \Rightarrow p_2 \ldots \Rightarrow p_n \Rightarrow q) \Rightarrow (p \Rightarrow q)$
n ulige $\quad$ $\exists k \in \mathbb{Z}: n^2 = 4k^2 + 4k + 1$ $\quad$ n^2 ulige
$\exists k \in \mathbb{Z}: n = 2k + 1$ $\quad$ $\exists k \in \mathbb{Z}: n^2 = 2(2k^2 + 2k) + 1$

Def. 1.7.1

Lad $n \in \mathbb{Z}$. Da gælder
n er lige $\Leftrightarrow$ $\exists k \in \mathbb{Z}: n = 2k$
n er ulige $\Leftrightarrow$ $\exists k \in \mathbb{Z}: n = 2k + 1$
To heltal har samme paritet, hvis begge er lige, eller begge er ulige

Skal bevise Lad $n \in \mathbb{Z}$. Da gælder $n\ ulige \Rightarrow n^2\ ulige$

Direkte bevis:
$n\ er\ ulige$
Definitionen på ulige: $\exists k \in \mathbb{Z} : n = 2k + 1$
Vi vil her, omskrive n
$\qquad \exists k \in \mathbb{Z} : n^2 = (2k + 1)^2 = 4k^2 + 1 + 4k$
$\qquad\qquad = 2 * (2k^2 + 2k) + 1$
Her er $(2k^2 + 2k) \in \mathbb{Z}$
$\qquad \exists l \in \mathbb{Z} : n^2 = 2l + 1$
$\qquad l = 2l^2 + 2k$
Derved: n^2 er ulige

Kontrapositions bevis

Benytter, at $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$

Kontrapositions bevis

Benytter, at $p \Rightarrow q \Leftrightarrow \neg q \Rightarrow \neg p$

n^2 ulige, n ulige, n lige, n^2 lige

Modstridsbevis

Benytter, at $(\neg p \Rightarrow f) \Rightarrow p$

Modstridsbevis

Benytter, at $(\neg p \Rightarrow F) \Rightarrow p$

$\neg\exists$ måned med ≥ 2 fødselsdage; ≤ 12 personer; $\exists$ måned med ≥ 2 fødselsdage

$\exists \triangle : c \geq a+b$; $c^2 > a^2+b^2$; $\forall \triangle : c < a+b$

Pythagoras eksempel:

For enhver retvinklet trekant gælder $c < a + b$ (når $a, b > 0$)

Vi skal derved sige (og tilstrækkeligt), at der findes en trekant, som overholder: $c \geq a + b$

Bevis:

Antag til modstrid, at der eksisterer en retvinklet trekant med $c \geq a + b$

Derved:

$c \geq a + b \Leftrightarrow c^2 \geq (a+b)^2 \Leftrightarrow c^2 \geq a^2 + b^2 + 2ab$

Pythagoras siger at dette ikke er sandt, fordi: $c^2 = a^2 + b^2$

Så det er derved i **modstrid** til Pythagoras

Kontrapositions bevis:

Benytter, at $(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)$

Derved: $n\ lige \equiv n^2\ lige$

Det her kan vi bedre, da vi går fra n til n^2 i stedet for den anden vej, som er meget svær

Bevis:

$$n\ lige$$
$$\exists k \in \mathbb{Z} : n = 2k$$
$$\exists k \in \mathbb{Z} : n^2 = 4k^2 = 2 * 2k^2$$
$$l = 2k^2$$
$$\exists l \in \mathbb{Z} : n^2 = 2 * l$$

Derved: $n^2 er\ lige$

Modstridsbevis:

Der er to deltagere i denne forelæsning, som har fødselsdage i samme måned.

Her går vi fra, at det så ikke er sandt, derved kan det siges: I januar er der højst 1 som har fødselsdag, etc.

- Dette medfører, at der højst kan være 12 deltagere

Bevis:

*Antag til **modstrid**, at der <u>ikke</u> er nogen måned med to fødselarer. Da er der højst 12 deltagere*

Induktionsbevis

Bruges til at bevise parametriserede (iterative / rekursive) udsagn - F.eks.: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

Generelt: (induktionsskridt)

Hvis vi ved, at $2^0+2^1+\ldots+2^{k-1} = 2^k-1$, hvor $k \in \mathbb{Z}^+$, kan vi udlede, at $2^0+2^1+\ldots+2^k = 2^{k+1}-1$:

$$2^0+2^1+\ldots+2^{k-1} = 2^k-1$$
$$\Updownarrow$$
$$2^0+2^1+\ldots+2^{k-1}+2^k = 2^k-1+2^k = 2\cdot 2^k - 1 = 2^{k+1}-1$$
$$\Updownarrow$$
$$2^0+2^1+\ldots+2^{k-1}+2^k = 2\cdot 2^k - 1$$
$$\Updownarrow$$
$$2^0+2^1+\ldots+2^{k-1}+2^k = 2^{k+1}-1$$

Ovenstående beviser, at $P(k-1) \Rightarrow P(k)$, for $k \geq 1$.

D.v.s. $(P(0) \Rightarrow P(1)) \wedge (P(1) \Rightarrow P(2)) \wedge (P(2) \Rightarrow P(3)) \wedge \ldots$

Vi har også verificeret $P(0)$ (og $P(1)$, $P(2)$ og $P(3)$).

Dermed har vi bevist $P(n)$ for alle $n \in \mathbb{N}$.

D.v.s. vi har udført et <u>Induktionsbevis</u>:

Basis: Bevis $P(0)$

Induktionsskridt: Bevis $P(k-1) \Rightarrow P(k)$ for $k \geq 1$

$P(0)$: $\underbrace{2^0}_{1} = \underbrace{2^1-1}_{2-1=1}$ ✓ (Basis)

$P(1)$: $\underbrace{2^0}+2^1 = 2^2-1$

$2^1-1+2^1 = 2\cdot 2^1-1 = 2^2-1$ ✓

$P(2)$: $\underbrace{2^0+2^1}+2^2 = 2^3-1$

$2^2-1+2^2 = 2\cdot 2^2-1 = 2^3-1$ ✓

$P(3)$: $\underbrace{2^0+2^1+2^2}+2^3 = 2^4-1$

$2^3-1+2^3 = 2\cdot 2^3-1 = 2^4-1$ ✓

Nu kan vi se et mønster...

„Domino-effekt":

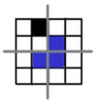
$P(0) \Rightarrow P(1) \Rightarrow P(2) \Rightarrow P(3) \Rightarrow \ldots$

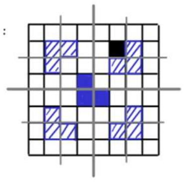
Basis, $k=1$, $k=2$, $k=3$

Eks 14:

Et $2^n \times 2^n$ skakbræt ($n \geq 1$) med et felt fjernet kan dækkes af ⊞-brikker (uden at brikker overlapper eller stikker ud over kanten)

$n=1$:  ✓

$n=2$:  ✓

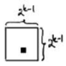
$n=3$:  ✓

Eks 5.1.14:

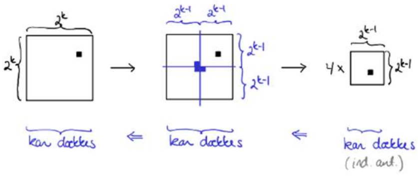
Et $2^n \times 2^n$ skakbræt ($n \geq 1$) med et felt fjernet kan dækkes af ⊞-brikker

Bevis ved induktion over n

Basis ($n=1$): 

Ind. ant:  2^{k-1}, hvor $k \geq 2$, kan dækkes

Ind. skridt: ($k \geq 2$)



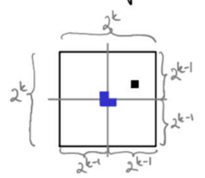
Et $2^n \times 2^n$ skakbræt ($n \geq 1$) med et felt fjernet kan dækkes af ⊞-brikker

Bevis ved induktion over n

Basis ($n=1$): 

Ind. ant. ($k \geq 2$):
Et $2^{k-1} \times 2^{k-1}$ skakbræt med et felt fjernet kan dækkes af ⊞-brikker

Ind. skridt. ($k \geq 2$):
Del brættet op i fire lige store kvadrater.
Hvert af dem har størrelse $2^{k-1} \times 2^{k-1}$:



Placer en brik, så den dækker et felt i hvert af de tre kvadrater, som ikke har et hul.

Nu mangler hvert af de fire kvadrater præcis et felt.
Dermed kan de hver især dækkes, ifølge ind. ant..

Relationer

Relationer beskriver sammenhænge ml. elementer i en eller flere mængder. Eks. I databaser.

(studie-nr, kursus)
<
=
er barn af

} binære relationer

Def. 9.1.2
A: mængde
En relation på A er en relation fra A til A, d.v.s. en delmængde af $A \times A$

Def. 9.1.1
A, B: mængder
En relation fra A til B er en delmængde af $A \times B$
kartesiske produkt

Eks ovenfor:
A: mgd. af studie-nr, B: mgd. af kurser
Eller $A = B = \mathbb{Z}$
Eller...

$$\{1,2\} \times \{3,4\} = \{(1,3),(1,4),(2,3),(2,4)\}.$$

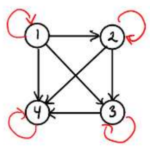
Figur 5: Kartesisk produkt

Notation:

Eks: $A = \{1,2,3,4\}$

$$R_{\leq} = \{(a,b) \in A \times A \mid a \leq b\}$$
$$= \{(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)\}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

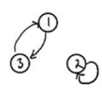


Eks: Relationer på $\{1,2,3\}$

$R_1 = \{(1,3),(2,2),(3,1)\}$

1 er relateret til 3 gennem R_1
2 ——"—— 2 ——"——
3 ——"—— 1 ——"——

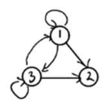
$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$



Symmetrisk

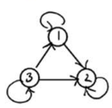
$R_2 = \{(1,1),(1,2),(1,3),(3,1),(3,2),(3,3)\}$

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$



$R_3 = \{(1,1),(1,2),(2,2),(3,1),(3,2),(3,3)\}$

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$



Refleksiv

Egenskaber

Refleksiv

Def. 9.1.3
R: relation på A
R er refleksiv, hvis
$(a,a) \in R$, for alle $a \in A$

Symmetrisk

Def. 9.1.4
R: relation på A
R er symmetrisk, hvis
$(a,b) \in R \Rightarrow (b,a) \in R$, for alle $a,b \in A$

✓: Samme paritet, $=$, $|\cdot|$, $\neq$

Antisymmetrisk

Def. 9.1.4
R: relation på A
R er antisymmetrisk, hvis
$(a,b) \in R \wedge (b,a) \in R \Rightarrow a = b$, for alle $a,b \in A$

Eks:
Anti-symmetrisk: $<, \leq, =, |$
Symmetrisk: $=$, samme paritet
Ingen af delene: $\{(1,1),(1,2),(2,1),(1,3)\}$

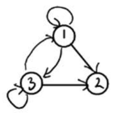
Transitiv

Def. 9.1.5

R: relation på A
R er transitiv, hvis
$(a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R$, for alle $a,b,c \in A$

$R_2 = \{(1,1), (1,2), (1,3), (3,1), (3,2), (3,3)\}$

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$



Transitiv

Lukninger

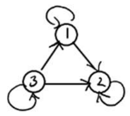
Def. 9.4.1

R: relation, P: egenskab
Lukningen af R mht. P (hvis den eksisterer)
er den mængde C, som opfylder
(1) $R \subseteq C$
(2) C har egenskaben P
(3) For enhver relation S, som opfylder (1) og (2), gælder $C \subseteq S$.

D.v.s. lukningen af R mht. P er den minimale relation, som indeholder R og har egenskaben P.

$R_3 = \{(1,1), (1,2), (2,2), (3,1), (3,2), (3,3)\}$

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$



Refleksiv
Antisymmetrisk
Transitiv

Refleksive lukning

R: relation på A
Den refleksive lukning af R er
$r(R) = R \cup \{(a,a) \mid a \in A\}$

Eks: Relation på $\{1,2,3,4\}$:
$R = \{(1,3), (2,2), (2,3), (3,1), (3,3)\}$

$r(R) = R \cup \{(1,1), (2,2), (3,3), (4,4)\}$
$= \{(1,1), (1,3), (2,2), (2,3), (3,1), (3,3), (4,4)\}$

Den refleksive lukning skabes ved at tilføje præcis de elementer, der mangler for, at R er refleksiv.

Symmetriske lukning

Den symmetriske lukning af en relation R er
$s(R) = R \cup \{(b,a) \mid (a,b) \in R\}$

$s(\{(1,3), (2,2), (2,3), (3,1)\})$
$= \{(1,3), (2,2), (2,3), (3,1), (3,2)\}$

$s(R_<) = R_< \cup R_> = R_{\neq}$

$s(R_\leq) = R_\leq \cup R_> = A \times A$

$s(\text{„samme paritet"}) = \text{„samme paritet"}$

Antisymmetrisk lukning

Eksistere ikke *(Tilføje ting, hjælper ikke)*

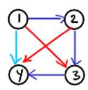
Transitiv lukning

Den transitive lukning af en relation R er
$t(R) = \{(a,b) \mid \exists x_1, x_2, \dots, x_n : (a = x_1 \wedge x_n = b \wedge \forall i \in \{1,2,\dots,n-1\} : (x_i, x_{i+1}) \in R)\}$

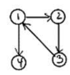
D.v.s.:
Hvis der er en sti fra a til b i grafen for R, er der en kant fra a til b i grafen for t(R).

Eks: $R = \{(1,2), (2,3), (3,4)\}$

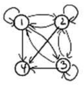
$t(R) = \{(1,2), (2,3), (3,4)\} \cup \{(1,3), (2,4)\} \cup \{(1,4)\}$
$= \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}$



R:



t(R):



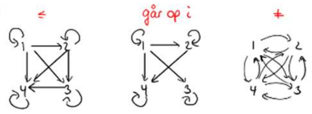
Ækvivalensrelationer

Def 9.5.1

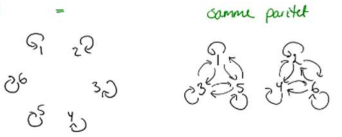
En relation, som er refleksiv, symmetrisk og transitiv, kaldes en ækvivalensrelation

Eks:

✗: ≤ går op i ≠



✓: = samme paritet

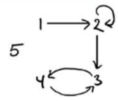


Ækv. klasser:
$[1] = \{1\}$
$[2] = \{2\}$
$\vdots$
$[6] = \{6\}$

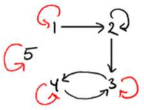
Ækv. klasser:
$[1] = \{1,3,5\} = [3] = [5]$
$[2] = \{2,4,6\} = [4] = [6]$

Eks

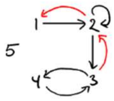
$R = \{(1,2), (2,2), (2,3), (3,4), (4,3)\}$
på $\{1,2,3,4,5\}$



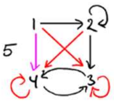
Refleksiv lukning af R:
$r(R) = \{(1,1), (1,2), (2,2), (2,3), (3,3), (3,4), (4,3), (4,4), (5,5)\}$



Symmetrisk lukning af R:
$s(R) = \{(1,2), (2,1), (2,2), (2,3), (3,2), (3,4), (4,3)\}$



Transitiv lukning af R:
$t(R) = \{(1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,3), (4,4)\}$



Partielle ordninger

Def 9.6.1

Hvis en relation R på en mængde A er refleksiv, antisymmetrisk og transitiv, kaldes den en partiel ordning.
(A,R) kaldes en partielt ordnet mængde (poset)

Eks:

✓: $\leq$ $=$ $|$ $\subseteq$

✗: $<$ $\neq$ $\subset$ „samme paritet"

✗ refl. ✗ trans. ✗ antisym.

Def 9.6.2

Lad $\preccurlyeq$ være en partiel ordning.
Hvis $a \preccurlyeq b$ eller $b \preccurlyeq a$, kaldes a og b sammenlignelige.

Def. 9.6.3

Lad $(A, \preccurlyeq)$ være en partielt ordnet mængde.
Hvis alle par $a, b \in A$ er sammenlignelige, kaldes $\preccurlyeq$ en total ordning

Eks:

✓: $\leq$

✗: $=$, $|$, $\subseteq$, $<$

Leksikografisk ordning

Lad $\preccurlyeq_1, \preccurlyeq_2$ være partielle ordninger

$a \prec_1 b$ betyder $a \preccurlyeq_1 b \land a \neq b$

$a \prec_2 b$ betyder $a \preccurlyeq_2 b \land a \neq b$

$(a_1, a_2) \prec (b_1, b_2)$, hvis

$a_1 \prec_1 b_1$

eller

$a_1 = b_1$ og $a_2 \prec_2 b_2$

Eks: Koordinat-sæt

$(1,4) \prec (2,3)$

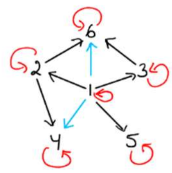
$(1,2) \prec (1,3)$

Eks: Ordbog

abe ≺ bil

abe ≺ absolut

Eks: $R = \{(a,b) \mid a|b\}$ på $\{1,2,3,4,5,6\}$



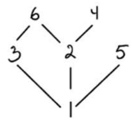
følger af egenskaben refleksiv

→ følger af egenskaben transitiv

Hasse-diagram

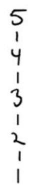
Som graf, men:

- Udelad kanter, som kan udledes af egenskaberne refleksiv og transitiv.
- Tegn a under b, hvis aRb



Eks: $R = \{(a,b) \mid a \leq b\}$ på $\{1,2,3,4,5\}$ er en total ordning.

Hasse-diagram:



Hurtig opslag

$\leq$ $\geq$ $=$ $<$ $>$

O Ω Θ o ω

Køretider