Algoritmer og datastrukturer

Introduktion til kurset

Rolf Fagerberg

Forår 2025

Hvem er vi?

Underviser:

- Rolf Fagerberg, Institut for Matematik og Datalogi (IMADA)
  Forskningsområde: algoritmer og datastrukturer

Deltagere:

- BA i Datalogi (2. sem.)
- BA i Kunstig Intelligens (2. sem.)
- BA i Software Engineering (4. sem.)
- BA i Matematik-Økonomi (4. sem.)
- BA i Anvendt Matematik (4. sem.)
- BA sidefag i Datalogi (6. sem.)
- KA profil i Data Science (2./8. sem.)

Bemærk stor diversitet: forskellige semestre i uddannelsen, forskellige mængder af programmering og af matematiske fag på uddannelsen.

Tre (fire) kurser i ét

DM578:
- Algoritmer og datastrukturer (7.5 ETCS)
- Skriftlig MC eksamen (3 timer)

DM507/DS814:
- Algoritmer og datastrukturer (7.5 ETCS)
- Programmeringsprojekt i tre dele (2.5 ETCS)
- Skriftlig MC eksamen (3 timer)

SE4-DMAD:
- Algoritmer og datastrukturer (7.5 ETCS)
- Diskret matematik (2.5 ETCS)
- Skriftlig MC eksamen (3 timer og 3 kvarter)

Diskret matematik har separate forelæsninger (Lene Monrad Favrholdt) og øvelsestimer. Tirsdage 10-12 i ti uger.

Der er et ITS-kursusrum for DM507/DM578/DS814 og et andet ITS-kursusrum for SE4-DMAD.

Kursets format (DM578)

Forudsætninger:

Programmering i Python eller Java, lidt matematisk modenhed

Format:

Forelæsninger (f-timer) ved Rolf Fagerberg.
Opgaveregning (e-timer) ved instruktor.
Arbejde selv og i studiegrupper.

Eksamenform:

Skriftlig eksamen (juni), 7.5 ECTS:

Multiple-choice (med bøger og noter). Karakter efter 7-skala. Mål: check af kendskab til stoffet. [NB: reeksamen er *mundtlig*.]

Kursets format (DM507/DS814)

Forudsætninger:

Programmering i Python, lidt matematisk modenhed

Format:

Forelæsninger (f-timer) ved Rolf Fagerberg.
Opgaveregning (e-timer) ved Instruktor.
Arbejde selv og i studiegrupper.

Eksamenform:

Skriftlig eksamen (juni), 7.5 ECTS:

Multiple-choice (med bøger, noter). Karakter efter 7-skala. Mål: check af kendskab til stoffet. [NB: reeksamen er *mundtlig*.]

Projekt undervejs, 2.5 ECTS, Python:

I tre dele. Karakter B/IB. Skal ikke bestås for at gå til skriftlig eksamen. Mål: træne overførsel af stoffet til praksis (programmering).

Kursets format (SE4-DMAD)

Forudsætninger:

Programmering i Java eller Python, lidt matematisk modenhed

Format:

Forelæsninger (f-timer) ved Rolf Fagerberg og Lene Monrad Favrholdt.
Opgaveregning (e-timer) ved instruktor.
Arbejde selv og i studiegrupper.

Eksamenform:

Skriftlig eksamen (juni), 10 ECTS:

Multiple-choice (med bøger, noter). Karakter efter 7-skala.
Mål: check af kendskab til stoffet. [NB: reeksamen er *mundtlig*.]

Materialer i algoritmer og datastrukturer

Lærebog:

Cormen, Leiserson, Rivest, Stein:
Introduction to Algorithms, 4th edition, 2022.

(Jeg vil også angive læsestof og opgaver ud fra 3rd edition.)

Andet læremateriale på kursets webside:

Slides fra forelæsninger
Opgaver til øvelsestimer
Tidligere eksamenssæt
Projektet

Forventet arbejdsindsats (DM578)

- Skim stof før forelæsning: 0,5 timer $\Leftarrow$ mindst vigtig
- Forelæsning: 2 timer
- Læs stof efter forelæsning: 1,5 timer
- Opgaveregning (hjemme): 3 timer $\Leftarrow$ mest vigtig
- Opgaveregning (klasse): 2 timer $\Leftarrow$ mest vigtig

Ovenstående i gennemsnit 1.5 gang per uge over 14 uger. Dertil følgende én gang:

- Eksamenslæsning: 40 timer
- Spørgetime og eksamen: 6 timer

I alt: $14 \cdot 1.5 \cdot 9 + 40 + 6 = 235$ timer

7.5 ECTS $= 1/8$ årsværk $= 1650/8$ timer $= 206$ timer

Forventet arbejdsindsats (DM507/DS814)

- Skim stof før forelæsning: 0,5 timer $\Leftarrow$ mindst vigtig
- Forelæsning: 2 timer
- Læs stof efter forelæsning: 1,5 timer
- Opgaveregning (hjemme): 3 timer $\Leftarrow$ mest vigtig
- Opgaveregning (klasse): 2 timer $\Leftarrow$ mest vigtig

Ovenstående i gennemsnit 1.5 gang per uge over 14 uger. Dertil følgende én gang:

- Projektet: 15+15+25 timer
- Eksamenslæsning: 40 timer
- Spørgetime og eksamen: 6 timer

I alt: $14 \cdot 1.5 \cdot 9 + 55 + 40 + 6 = 290$ timer

10 ECTS = 1/6 årsværk = 1650/6 timer = 275 timer

Forventet arbejdsindsats (SE4-DMAD)

- Skim stof før forelæsning: 0,5 timer $\Leftarrow$ mindst vigtig
- Forelæsning: 2 timer
- Læs stof efter forelæsning: 1,5 timer
- Opgaveregning (hjemme): 3 timer $\Leftarrow$ mest vigtig
- Opgaveregning (klasse): 2 timer $\Leftarrow$ mest vigtig

Ovenstående i gennemsnit 1.5 per uge over 14 uger (Rolf) og 0.5 gang per uge over 10 uger (Lene). Dertil følgende én gang:

- Eksamenslæsning: 40 timer
- Spørgetime og eksamen: 6 timer

I alt: $(14 \cdot 1.5 + 10 \cdot 0.5) \cdot 9 + 40 + 6 = 280$ timer

10 ECTS $= 1/6$ årsværk $= 1650/6$ timer $= 275$ timer

Kursets formål og plads i det store billede

Generelt mål i IT: Få en computer til at udføre en opgave.

Relaterede spørgsmål:

- Hvordan skrives programmer?
  Programmering, programmeringssprog, software engineering.
- Hvordan skal programmet løse opgaven? $\Leftarrow$ DM507/DM578/...
  Algoritmer og datastrukturer, databasesystemer, lineær algebra med anvendelser, data mining og machine learning.
- (Hvor godt) er det overhovedet muligt at løse opgaven?
  Nedre grænser, kompleksitet, beregnelighed.
- Hvordan fungerer maskinen der udfører opgaven?
  Baggrundsviden om computerarkitektur og operativsystemer.

Hvordan skal programmet løse opgaven?

Algoritme = løsningsmetode.

Tilpas præcist skrevet ned: tegning, tekst, pseudo-kode,...

Datastruktur = data + effektive operationer herpå.

Forskellige niveauer af løsning:

1. **Opfind** én algoritme som løser opgaven.
2. **Sammenlign** flere algoritmer som løser opgaven.
3. Hvad er den **bedst mulige** algoritme som løser opgaven?

Udvikling og vurdering af algoritmer

1. **Opfind** en algoritme som løser opgaven: Kræver ideer, erfaring, og en værktøjskasse med både eksisterende algoritmer og metoder til at udvikle nye.

2. **Sammenlign** flere algoritmer som løser opgaven: Kræver definition af hvad kvalitet er (ofte: kvalitet = lavt tidsforbrug).

3. Hvad er den **bedst mulige** algoritme som løser opgaven?

Analyse (tænkearbejde, argumenter, beviser): godt værktøj til punkt 1, 2 og 3. Giver høj sikkerhed for korrekthed af metoden/idéen. Sparer implementationsarbejde. Sammenligning upåvirket af: maskine, sprog, programmør, og valg af testdata.

Afprøvning (implementation, test): godt værktøj til punkt 1 og 2. Kan udforske ideer, fange implementationsfejl, belyse ting som ikke fanges af analysen.

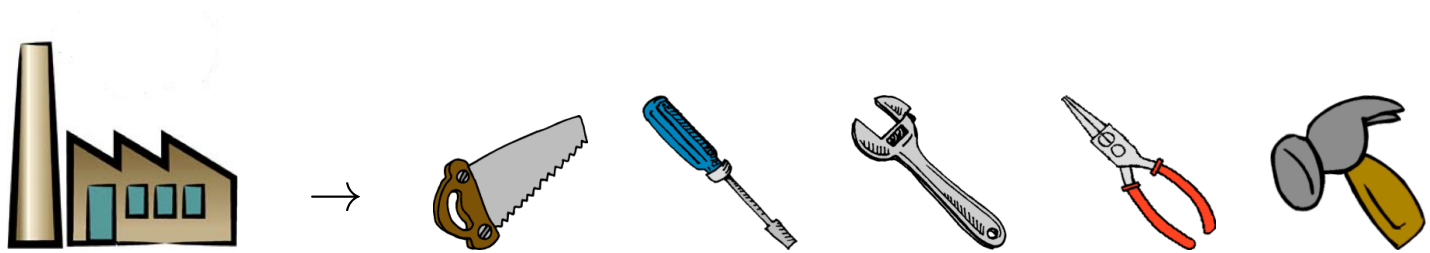
Udvikling og vurdering af algoritmer

DM507/DM578/DS814/SE4-DMAD vil have mest fokus på analyse, lidt mindre på implementation og afprøvning.

I alle byggefag analyserer og planlægger man før man bygger (tænk f.eks. storebæltsbro).

Målsætning for kurset

DM507 giver dig en værktøjskasse af algoritmer for fundamentale opgaver, samt metoder til at udvikle og analysere nye algoritmer og varianter af eksisterende.



Målsætning for kurset

Øvelser og programmeringsprojekter øger din forståelse for værktøjerne og træner dig i brugen af dem.

Undervejs begejstres du måske også over smarte og elegante ideer i algoritmer og analyser.

Konkret indhold af kurset

Algoritmer:

- Analyse af algoritmer: korrekthed og køretid (værktøj)
- Del og hersk algoritmer (algoritmemetode)
- Grådige algoritmer (algoritmemetode)
- Dynamisk programmering (algoritmemetode)
- Konkrete algoritmer for sortering, graf-problemer (BFS, DFS, korteste veje, udspændende træer,...), fil-komprimering, multiplikation af matricer,...

Datastrukturer:

- Ordbøger (søgetræer og hashing)
- Prioritetskøer (heaps)
- Disjunkte mængder

Analyse af algoritmer for ombytningspuslespil

Algoritmer for ombytningspuslespil

Prøv ombytningspuslespillet på kurset webside. Hvilken score opnåede du i tredie forsøg?

- Hvilken algoritme bruger du?
- Kan du sige noget om bedste og værste køretid for din algoritme for puslespil med n brikker?
- Er køretiden relateret til antal brikker, som står rigtigt til at starte med?
- Er den "grådige algoritme" (= sæt én brik på plads i hvert skridt) bedst mulig, eller kan man få flere skridt hvor to sættes på plads ved nogle gange at undlade at sætte én på plads?
- Mere generelt, *kan vi præcist beskrive alle bedst mulige algoritmer?*

Model af puslespil

Vi modellerer brikkerne i et puslespil som tallene $1, 2, 3, \ldots, n$, nummereret efter den plads, brikken skal stå på:

5	10	14	3
1	11	9	15
8	7	2	12
4	13	6	16

$\rightarrow$

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

NB: pladen kan også modelleres som et array/en liste (grå tal er indekser, her startende med 1):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
5	10	14	3	1	11	9	15	8	7	2	12	4	13	6	16

Dette gør vi til opgavetimerne (i programmeringsopgaverne)

En opstilling af tallene $1, 2, 3, \ldots, n$ i et array af længde n kaldes også en *permutation*.

Den grådige algoritme og dens analyse

```
WHILE ikke alle brikker på plads:
  Vælg en brik ikke på plads
  Byt den med brikken på dens plads
```

For et puslespil med n brikker, hvad er det maksimale antal ombytninger?

> For hver ombytning: mindst én brik kommer på plads, og ingen brik forlader dens plads. Derfor maksimalt n ombytninger.

For et puslespil med n brikker, hvoraf t allerede står på plads, hvad er det maksimale antal ombytninger?

> For hver ombytning: mindst én brik kommer på plads, og ingen brik forlader dens plads. Derfor maksimalt $n - t$ ombytninger.

For et puslespil med n brikker, hvoraf t allerede står på plads, hvad er det minimale antal ombytninger?

> For hver ombytning: højst to brik kommer på plads. Derfor mindst $(n - t)/2$ ombytninger.

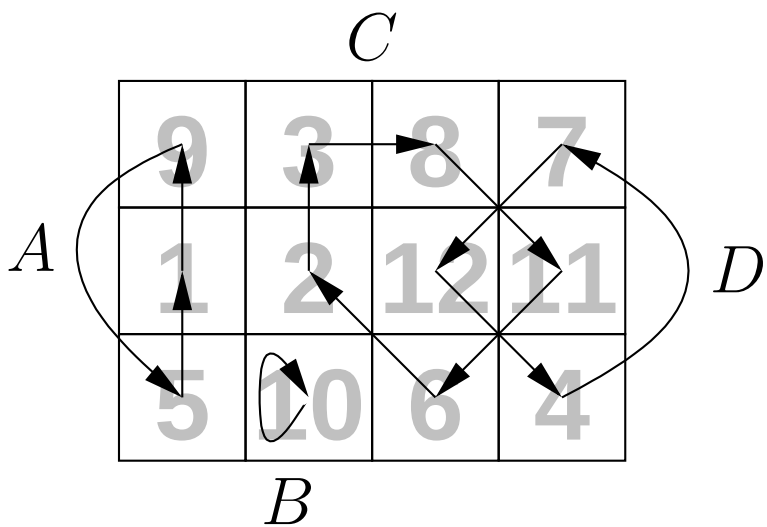
Kredse

Denne analyse på mellem $(n-t)/2$ og $n-t$ ombytninger er allerede ganske præcis (øvre og ned grænse er en faktor to fra hinanden).

Men vi kan lave en *endnu bedre* (mere præcis) analyse.

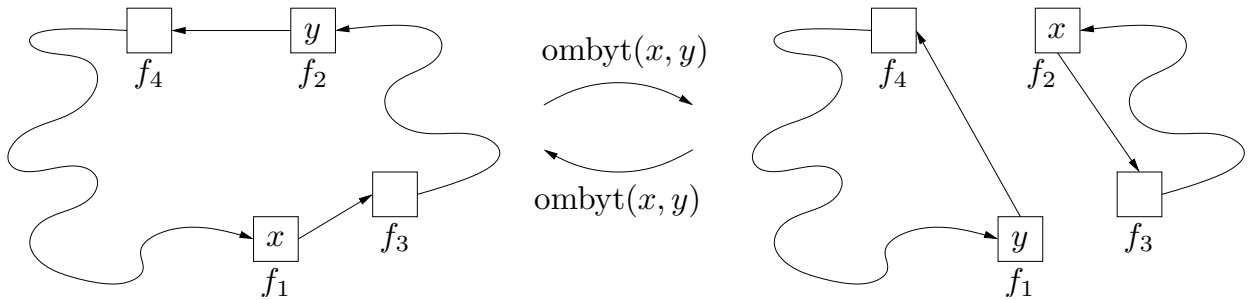
Observation: en permutation giver på naturlig måde anledning til en samling *kredse*:

Lad en brik (tal) t pege på den plads, hvor den skal stå, nemlig pladsen med index t.



Kredse og ombytninger

Observation: En ombytning af to brikker i samme kreds øger antal kredse med præcis én. En ombytning af to brikker i forskellige kredse mindsker antal kredse med præcis én:



Observation: Brik er på plads $\Leftrightarrow$ brik er i en kreds af længde én.

Derfor: puslespil løst $\Leftrightarrow$ der er n kredse.

Konklusion: Et puslespil med n brikker og k *kredse* i startopstillingen kræver mindst $n - k$ ombytninger, og man kan altid gøre det med $n - k$ ombytninger (bla. via den grådige algoritme, som i hvert skridt deler en kreds af længde t i to kredse af længde $t - 1$ og 1).

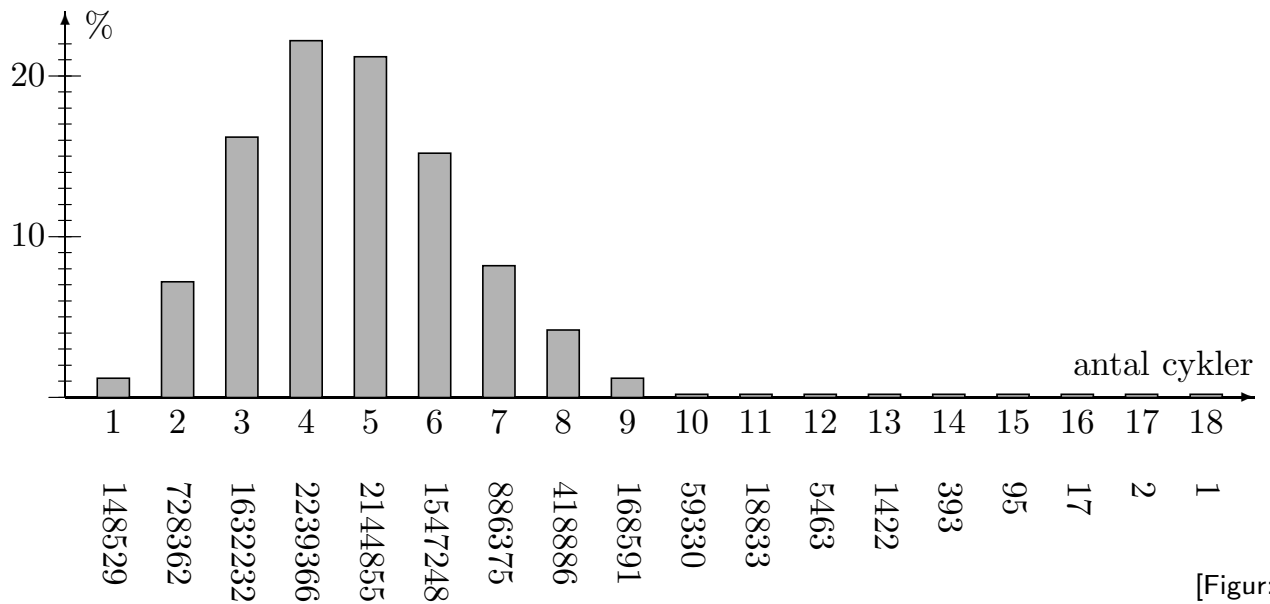
Konklusion: En algoritme bruger det optimale antal ombytninger $(n - k)$ hvis og kun hvis hver ombytning er med to brikker som er i samme kreds.

Forventet antal kredse

Sætning (uden bevis her): Det forventede antal kredse i en tilfældig permutation er

$$H_n = \sum_{i=1}^{n} 1/i$$

Ved simulering (afprøvning, 10.000.000 tilfældige permutationer) for $n = 64$ ses følgende fordeling af antallet af permutationer:



[Figur: Gerth Brodal]

Analyse af algoritmer

Fokus i kurset

Recap fra sidst: i DM507/DM578/DS814/SE4-DMAD vil vi

> Beskrive eksisterende algoritmer, udvikle nye algoritmer og vurdere/analysere algoritmer.

Spørgsmål: hvad skal vi fokusere på, når vi analyserer en algoritme?

Analyse af algoritmer

Mindstekrav til algoritmer er **korrekthed:**

- Stopper for alle input (aldrig uendelig løkke).
- Giver korrekt output, når den stopper (dvs. giver et svar på vores problem).

Korrekte algoritmer kan have forskellig **kvalitet:**

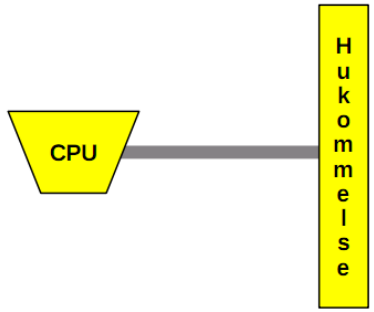
- Hastighed
- Pladsforbrug
- Kompleksitet af implementation
- Ekstra egenskaber (problemspecifikke)

Ingredienser i algoritmeanalyse

Vi har brug for:

- **Model af problem.** Individuelt for hvert problem (men ofte ret ligetil).
- **Model af maskine.** Vi bruger RAM-modellen (se næste side).
- **Mål for kvalitet.** Vi fokuserer på tidsforbrug.
- **Matematiske værktøjer.** Vi møder i dette kursus bl.a. asymptotisk notation, invarianter, induktion, rekursionsligninger.

RAM-modellen



- En hukommelse: En stor række af celler, hver med plads til et tal eller et tegn. (NB: liste/array = mange naboceller)
- En CPU med et antal basale operationer:
  - *plus*, *minus*, *gange*, *sammenlign*, ... to cellers indhold
  - *flyt* indhold mellem to celler
  - *jump i program* ($\Rightarrow$ *løkke*, *forgrening*, *funktions/metode-kald*).

Alle basale operationer antages at tage den samme tid.

- **Tid** for en algoritme: antal basale operationer udført.
- **Plads** for en algoritme: maks antal optagne hukommelsesceller.

Måle tidsforbrug: hvilket input?

For en givet størrelse n af input er der ofte mange forskellige konkrete input. Eksempel:

> **Sortering** = stille n elementer i stigende orden

For $n = 8$ er følgende lister fire ud af mange mulige input:

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

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

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

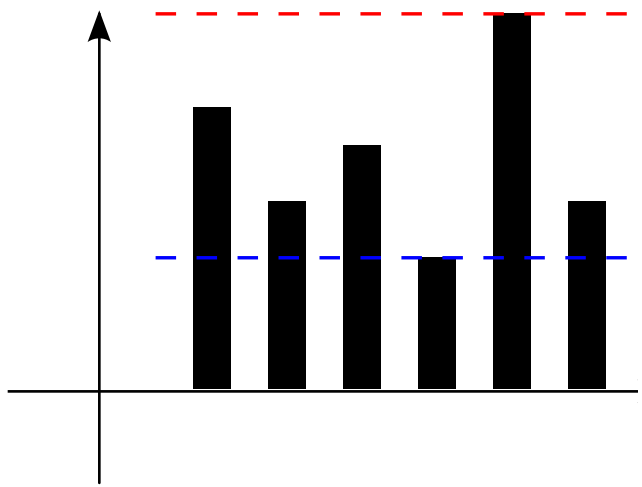
8,7,6,5,4,3,2,1

En algoritme har som regel *forskelligt* tidsforbrug på hver af disse.

Hvilke input skal vi bruge til at vurdere tidsforbruget?

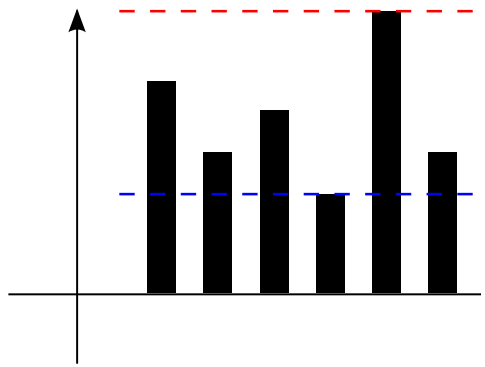
Måle tidsforbrug: hvilket input?

- Worst case (max over alle input af størrelse n)
- Average case (gennemsnit over en fordeling af input af størrelse n)
- Best case (min over alle input af størrelse n)



Køretid for de forskellige input af (samme) størrelse n

Worst case tidsforbrug



Worst case giver **garanti**. Er desuden ofte repræsentativ for average case (men kan også være mere pessimistisk).

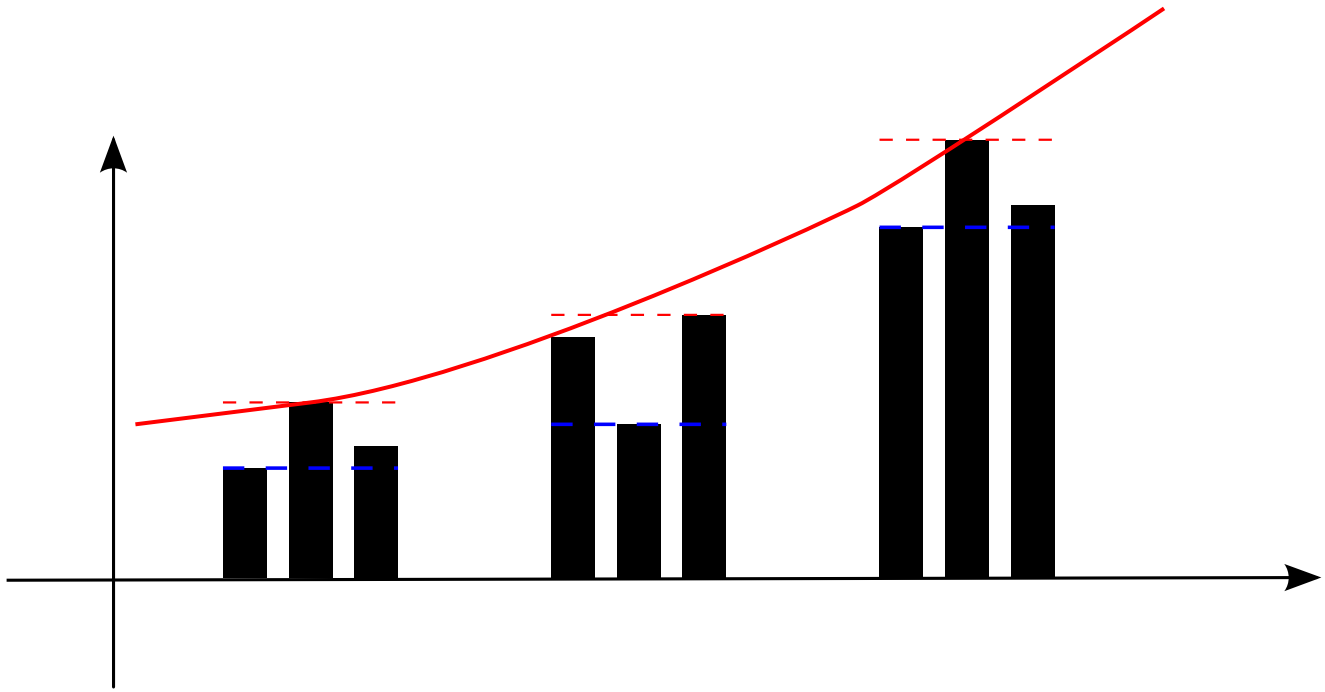
Average case: Hvilken fordeling af input? Hvorfor er denne fordeling realistisk/relevant? Analyse er ofte (matematisk) svær at gennemføre.

Best case: Giver ofte ikke så megen relevant information.

Næsten alle analyser i dette kursus er worst case.

Forskellige inputstørrelser

Worstcase køretid er normalt en voksende funktion af inputstørrelsen n:

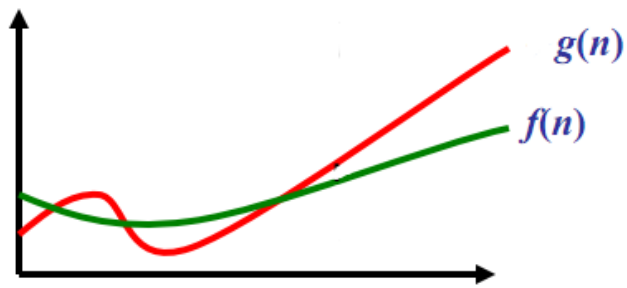


Køretid for de forskellige input af stigende størrelse n

Voksehastighed

En analyse må derfor give en funktion $f(n)$ af inputstørrelsen n.

For at sammenligne algoritmers køretid har vi derfor brug for at sammenligne funktioner. Mere præcist vil vi gerne undersøge, hvordan to funktioner $f(n)$ og $g(n)$ vokser, når n stiger.



Specielt vil vi gerne kunne genkende, hvis f.eks. $g(n)$ altid vil overstige $f(n)$, når n bliver stor nok. Vi siger så, at $g(n)$ har større voksehastighed end $f(n)$, og vi vil normalt foretrække algoritmen med køretid $f(n)$.

For små n er (næsten) alle algoritmer hurtige, så det er køretiden for store n, som er interessant.

Voksehastighed

Eksempler på funktioner listet efter stigende voksehastighed:

$$1, \quad \log n, \quad \sqrt{n}, \quad n, \quad n \log n,$$

$$n\sqrt{n}, \quad n^2, \quad n^3, \quad n^{10}, \quad 2^n$$

Disse har ret forskellig effektivitet i praksis:

	n	$n \log n$	n^2	n^3	n^{10}	2^n
1 minut	$6,0 \cdot 10^{10}$	$1,9 \cdot 10^9$	245.000	3.910	12	36
1 måned	$2,6 \cdot 10^{15}$	$5,7 \cdot 10^{13}$	50.900.000	137.000	35	51

Tabellen viser, hvilke inputstørrelser n man kan klare, hvis algoritmen skal udføre $f(n)$ CPU-operationer, og den skal være færdig efter henholdsvis et minut og en måned. Det er antaget, at en CPU kan lave 10^9 operationer per sekund. Se udleveret program for beregningerne.

Asymptotisk voksehastighed

Vi laver senere en formel definition af begrebet

asymptotisk voksehastighed

for funktioner. Dette bliver vores værktøj til at sammenligne køretider af algoritmer.

Sortering

Sortering

Input: n tal
Output: De n tal i sorteret orden

Eksempel:

$$6, 2, 9, 4, 5, 1, 4, 3 \rightarrow 1, 2, 3, 4, 4, 5, 6, 9$$

Mange opgaver er hurtigere i sorterede data (tænk på ordbøger, adresselister i telefoner,...). Dette gælder både for mennesker og for computere. Sortering er ofte en byggesten i algoritmer for andre problemer.

Sortering er en fundamental og central opgave.

Mange algoritmer er udviklet: Insertionsort, Selectionsort, Bubblesort, Mergesort, Quicksort, Heapsort, Radixsort, Countingsort, ...

Vi skal møde alle ovenstående i dette kursus.

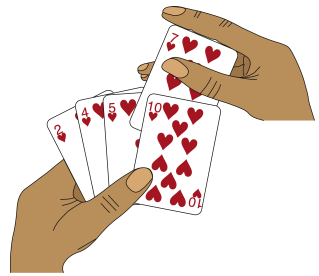
Sortering

Kommentarer:

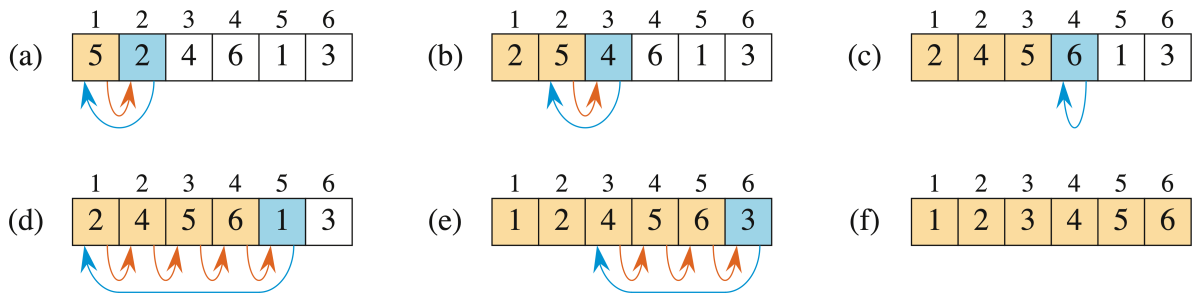
- Sorteret orden kan være stigende eller faldende. Vi vil i dette kursus altid bruge stigende (mere præcist: ikke-faldende). Skal man sortere faldende, skal alle sammenligninger bare vendes.
- Vi vil antage, at input ligger i en liste (Python) / et array (Java).
- Man sorterer ofte elementer sammensat af en sorteringsnøgle samt yderligere information. Sorteringsnøglen kan være et tal, kommatal, eller andet der kan sammenlignes (f.eks. strenge/ord). Vi viser i dette kursus blot elementer som heltal.

Insertionsort

Bruges af mange, når man sorterer en hånd i kort:



Samme idé udført på tal i en liste / et array:

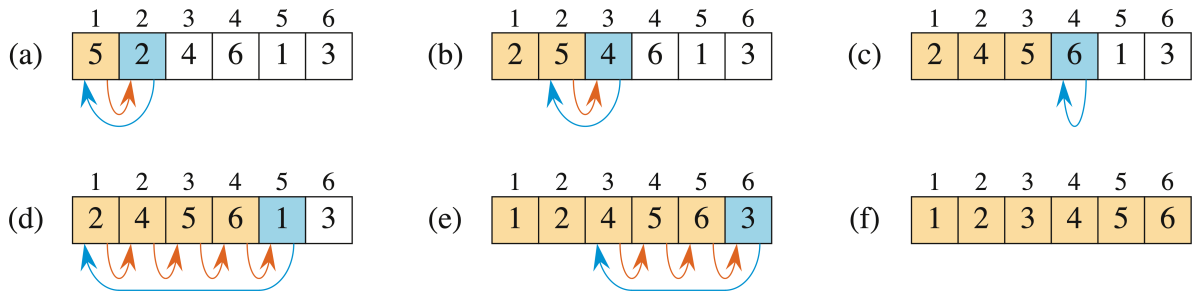


Argument for korrekthed: Den gule del af array er altid sorteret. Denne del udvides med én hele tiden ($\Rightarrow$ algoritmen stopper, og når den stopper er alle elementer sorteret).

Insertionsort

Som pseudo-kode:

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```



Analyse af køretid for Insertionsort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n-1$
3 // Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4 $j = i-1$	c_4	$n-1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n}(t_i-1)$
7 $j = j-1$	c_7	$\sum_{i=2}^{n}(t_i-1)$
8 $A[j+1] = key$	c_8	$n-1$

Her er t_i hvor mange gange testen i den indre **while**-løkke udføres. Dvs. $t_i - 1$ er hvor mange gange denne løkke kører (hvilket er hvor mange elementer det i'te element skal forbi under indsættelsen). Bemærk, at $1 \leq t_i \leq i$. Sæt $c = c_1 + c_2 + \cdots + c_8$.

Best case: $t_i = 1$ for alle i. Samlet tid $\leq c \cdot n$.

Worst case: $t_i = i$ for alle i. Samlet tid $\leq c \cdot n^2$, eftersom

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

Selectionsort

En anden simpel og naturlig sorteringsalgoritme:

```
indListe = input
udListe = tom liste
while indListe ikke tom:
    find mindste element x i indListe
    flyt x fra indListe til enden af udListe
```

Klart **korrekt**, dvs. giver sorteret output (hvert element, som udtages, er mindre eller lig alle efterfølgende, der udtages).

Køretid?

I alt n gange findes mindste element i `IndListe`.

En simpel metode til er finde mindste element er lineær søgning, som kigger én gang på hvert tilbageværende element.

Derved bliver tiden $\leq c \cdot (n + (n-1) + (n-2) + \cdots + 1) \leq c \cdot n^2$.

Merge

Input: To **sorterede** rækker A og B
Output: De samme elementer i én sorteret række C

Eksempel:

$$\begin{array}{l} A = 2,4,5,7,8 \\ B = 1,2,3,6 \end{array} \quad \Rightarrow \quad C = 1,2,2,3,4,5,6,7,8$$

Vi kan naturligvis sortere $A \cup B$.

Men det er hurtigere at flette (**merge**):

Repeat:
Flyt det mindste af de to forreste elementer i A og B til enden af C

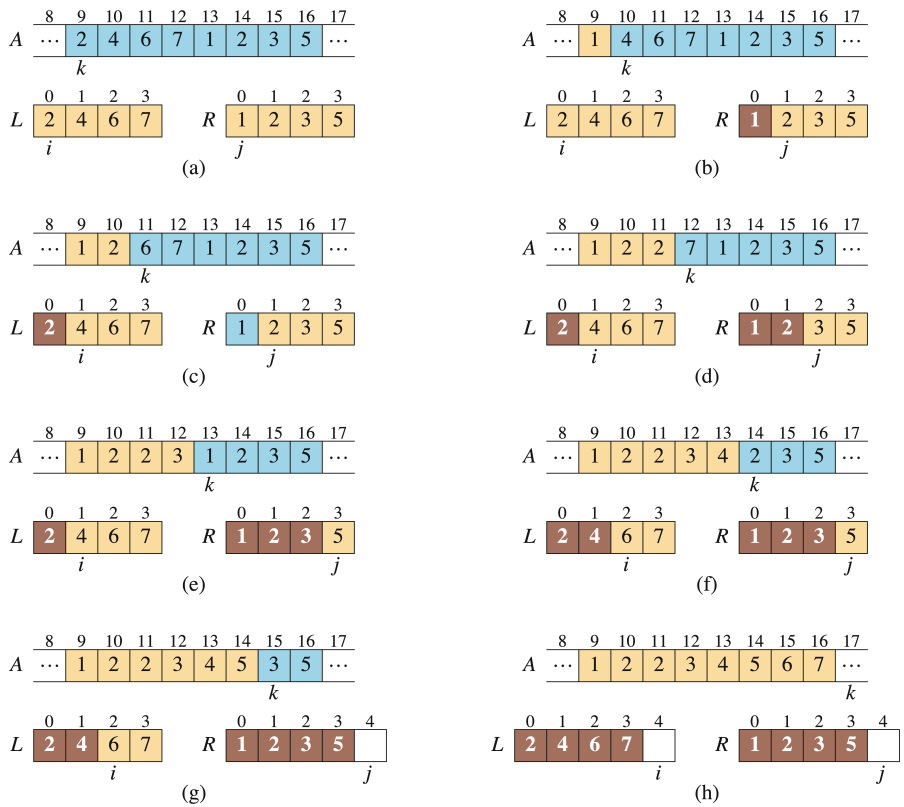
Køretid: $\leq c \cdot n$, hvor $n =$ antal elementer i alt i A og B.

Korrekthed: Merge kan ses som en udgave af Selectionsort, der udnytter at A og B er sorterede, så den mindste i (resten af) $A \cup B$ kan findes ved kun at se på de to forreste i A og B, hvilket tager konstant tid.

Eksempel på merge

Her antages, at de to input-lister er nabodele af samme liste/array A, nemlig $A[p \ldots q]$ og $A[q+1 \ldots r]$. De flyttes først over i L og R.

(Bemærk, at i bogen er $A[p \ldots q]$ lig elementerne $A[p]$, $A[p+1]$,. . . , $A[q]$, hvilket er ét element mere end næsten samme notation i Python.)

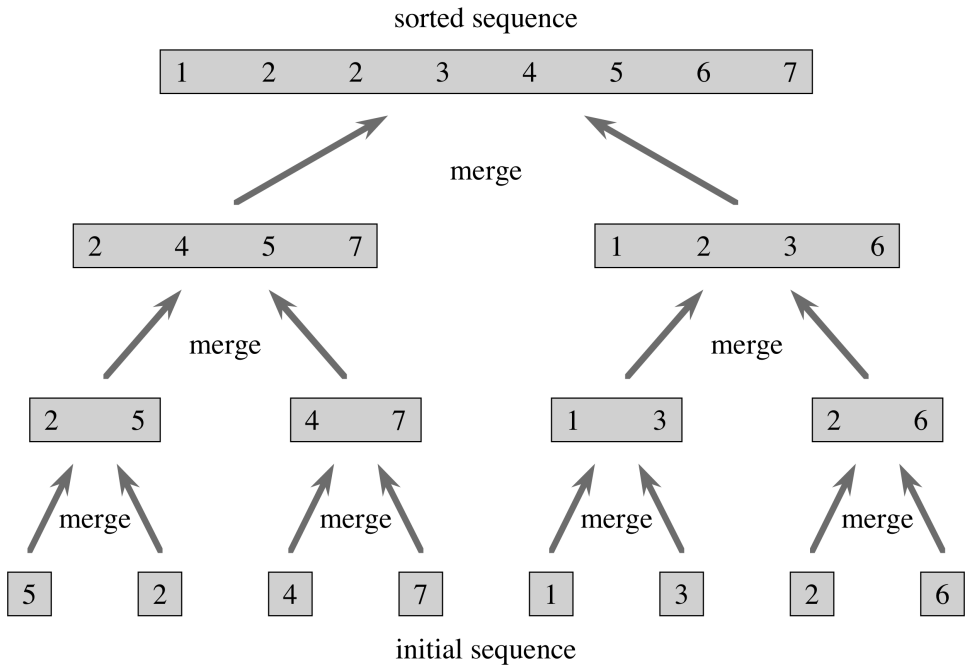


Pseudo-kode for Merge

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Mergesort

Mergesort: opbyg længere og længere sorterede dele af input ved gentagen brug af merge:

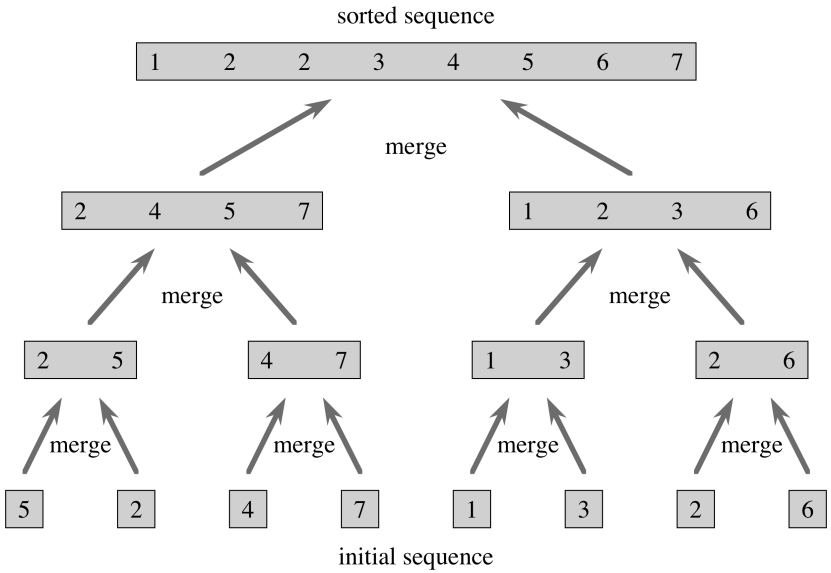


Tid: Hver merge bruger højst $c \cdot n_1$ tid, hvor n_1 er antal elementer der merges. Så alle merge-operationer i ét lag bruger tilsammen højst $c \cdot (n_1 + n_2 + \dots) = c \cdot n$. Dette gælder alle lagene. Der er i alt $\log_2 n$ lag, så den samlede tid er højst $c \cdot n \cdot \log_2 n$.

Mergesort

Hvorfor er der $\log_2 n$ merge-lag?

Antal sorterede lister efter k merge-lag (antag at n er en potens af 2):



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

Algoritmen stopper, når der er én sorteret liste:

$$n/2^k = 1 \quad \Leftrightarrow \quad n = 2^k \quad \Leftrightarrow \quad \log_2 n = k$$

Mergesort hvis n ikke er en potens af 2?

Algoritmen merger i hvert lag så mange par, den kan, og der bliver evt. én liste, som ikke merges (denne er med som liste på næste lag).

F.eks. bliver 12 lister til $12/2 = 6$ lister mens 13 $(= 12 + 1)$ lister bliver til $12/2 + 1 = 6 + 1 = 7$ lister.

Generelt: Hvis der er x lister før et merge-lag, er der $\lceil x/2 \rceil$ lister efter.

Se på to input størrelse n og n', med $n \leq n'$. Da $\lceil x/2 \rceil$ er en voksende funktion af x, kan antallet af lister i hvert lag ikke være mindre for n' end for n. Derfor kan antallet af lag (før algoritmen når ned på én liste) ikke være mindre for n' end for n.

Sæt n' til mindste potens af to, som er større end eller lig n. Der er præcis $\log_2 n'$ lag for n', og dermed højst så mange lag for n.

Omvendt er det nemt at se, at for $n = 2^k + 1$ er der $k + 1 = \lceil \log_2 n \rceil$ lag. Så der er $\lceil \log_2 n \rceil$ lag for generelt n.

n	7	$8 = 2^3$	9	10	11	12	13	14	15	$16 = 2^4$	17
$\log_2(n)$	2.807	3	3.169	3.321	3.459	3.584	3.700	3.807	3.906	4	4.087
Antal lag	3	3	4	4	4	4	4	4	4	4	5

Mergesort

Mergesort som pseudo-kode, i en variant formuleret med rekursion:

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

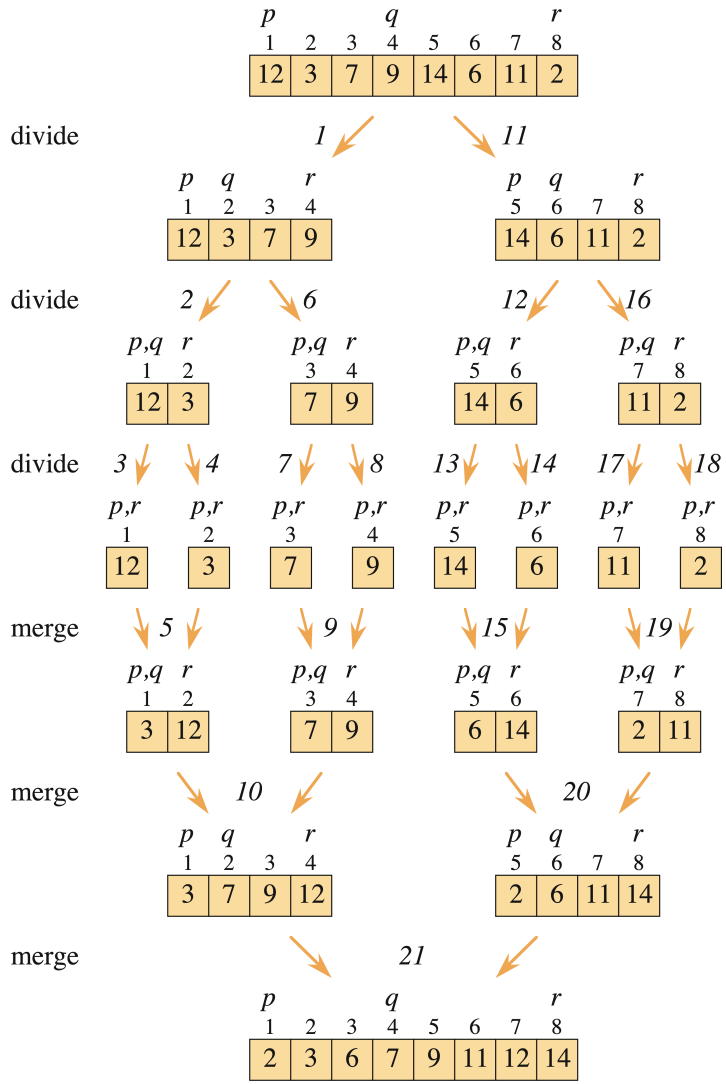
Et kald $\textsc{Merge-Sort}(A,p,r)$ har til opgave at stille elementerne i $A[p \ldots r]$ i sorteret orden.

Første kald er $\textsc{Merge-Sort}(A,1,n)$, som har til opgave at sortere hele A.

Et kald $\textsc{Merge}(A,p,q,r)$ merger de to sorterede del-arrays/lister $A[p \ldots q]$ og $A[q+1 \ldots r]$ sammen til $A[p \ldots r]$.

Mergesort

Eksempel på kørsel:



Quicksort

Mergesort:

- Del input op i to dele X og Y (trivielt)
- Sorter hver del for sig (rekursion)
- Merge de to sorterede dele til én sorteret del (reelt arbejde)

Basistilfælde: $n \leq 1$ (allerede sorteret, gør intet)

Quicksort:

- Del input op i to dele X og Y så $X \leq Y$ (reelt arbejde)
- Sorter hver del for sig (rekursion)
- Returner X efterfulgt af Y (trivielt)

Basistilfælde: $n \leq 1$ (allerede sorteret, gør intet)

[Hoare, 1960]

Quicksort

Som pseudo-kode:

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

Et kald $\textsc{Quicksort}(A,p,r)$ har til opgave at stille elementerne i $A[p \ldots r]$ i sorteret orden.

Første kald er $\textsc{Quicksort}(A,1,n)$, som har til opgave at sortere hele A.

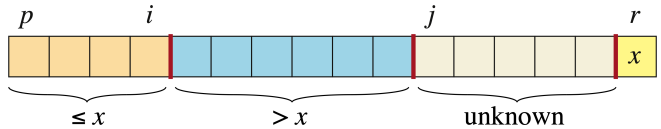
Et kald $\textsc{Partition}(A,p,r)$ vælger et element $x \in A$ og opdeler $A[p \ldots r]$ således at:

$$A[q] = x \qquad A[p \ldots q-1] \leq x \qquad A[q+1 \ldots r] > x$$

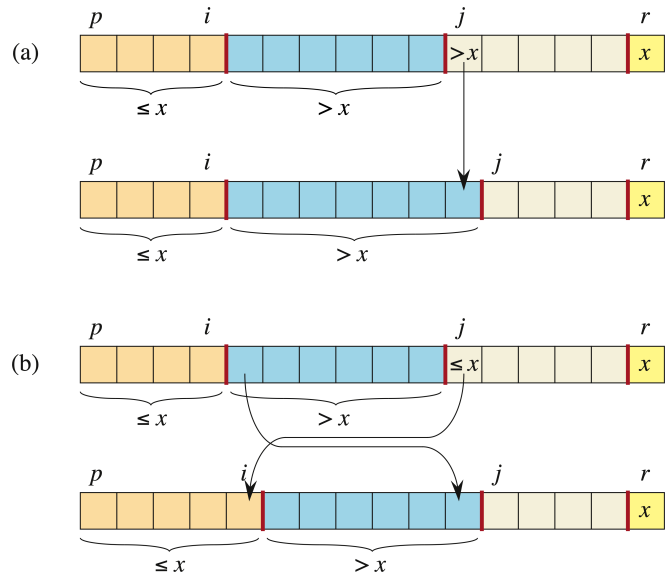
Partition

Hvordan lave PARTITION?

Idé: Vælg et element x fra input at opdele efter (her sidste element i array-del). Opbyg de to dele under et gennemløb af array ud fra flg. princip:

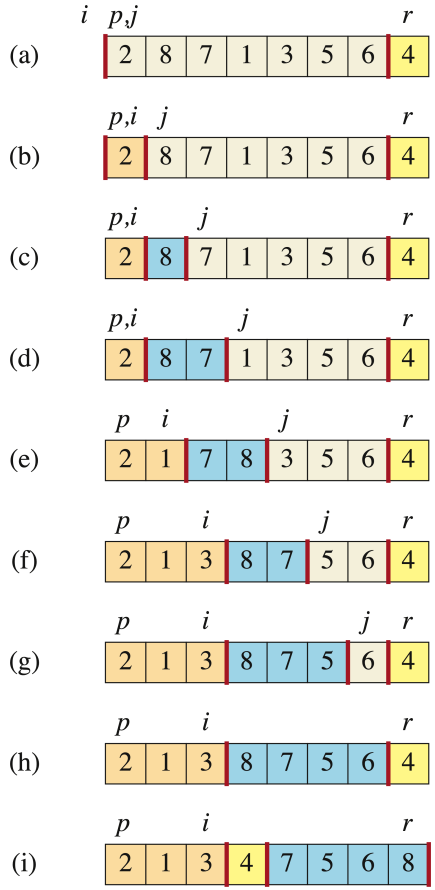


Hvordan tage et skridt under gennemløb?



Partition

Et eksempel på gennemløb:

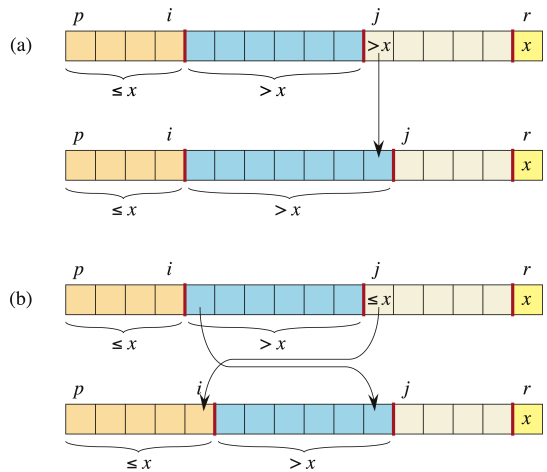


Tid: $O(n)$, hvor n er antal elementer i $A[p \ldots r]$.

Partition

Som pseudo-kode:

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



Quicksort køretid

Hænger på, hvordan partitions gennem rekursionen deler input.

To ekstremer af størrelser på rekursive kald:

- Helt ubalanceret: 0 og $n-1$
- Helt balanceret: $\lceil (n-1)/2 \rceil$ og $\lfloor (n-1)/2 \rfloor$

- Hvis alle partitions er helt balancerede: $O(n \log n)$ (ca. samme analyse som for Mergesort).
- Hvis alle partitions er helt ubalancerede: $O(n + (n-1) + (n-2) + \cdots + 2 + 1) = O(n^2)$.

Man kan vise at dette er henholdsvis best case og worst case for Quicksort.

Quicksort køretid

- I praksis: køretid $O(n \log n)$ for næsten alle input.
- Bemærk dog: allerede sorteret input giver køretid $\Theta(n^2)$, hvis opdelingselement x i Partition vælges som sidste element (hvad bogen gør). Brug ikke det valg i praksis.
- Forslag til mere robuste valg af opdelingselement x: enten som midterelementet, som medianen af flere elementer, som et tilfældigt element, eller som medianen af flere tilfældigt valgte elementer.
- Quicksort er *inplace*: bruger ikke mere plads end input-array'et.
- Kode er meget effektiv i praksis. En godt implementeret Quicksort er ofte bedste all-round sorteringsalgoritme (og valgt i mange biblioteker, f.eks. Java og C++/STL).

Heapsort

En Heap er:

1. et binært træ
2. med heap-orden
3. og heap-facon
4. udlagt i et array

(Note: "heap" bruges også om et hukommelsesområde brugt til allokering af objekter under et programs udførsel. De to anvendelser er urelaterede.)

[Williams, 1964]

1) Binært træ

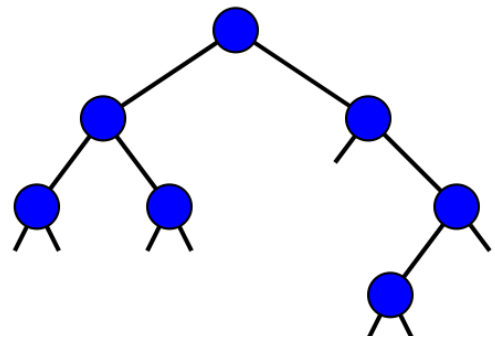
Et binært træ er enten

- det tomme træ

eller

- en knude v samt to undertræer (et højre og et venstre).

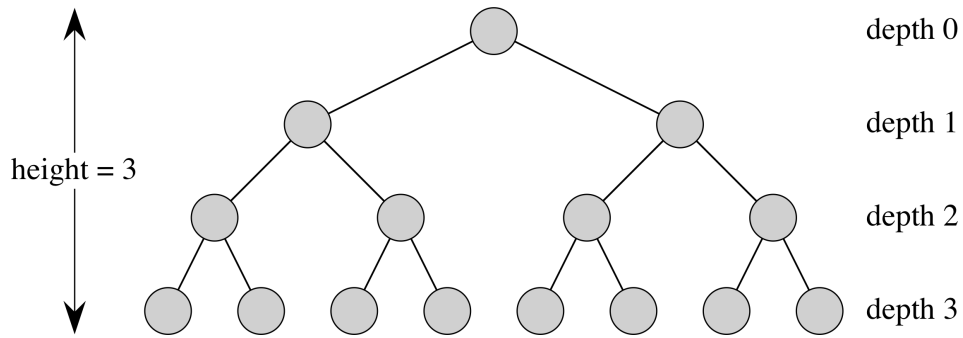
Visualisering:



Knuden v kaldes også rod for træet. Hvis v har et ikke-tomt undertræ, kaldes roden u af dette for et barn af v, og v kaldes u's forælder. Hvis begge v's undetræer er tomme, kaldes v et blad. Stregerne mellem børn og forældre kaldes for kanter. Forældre/barn-begrebet generaliserer på naturlig måde til forfader og efterkommer.

1) Binært træ

- Dybde af knude = antal kanter til rod
- Højde af knude = max antal kanter til blad
- Højde af træ = højde af dets rod
- Fuldt (complete) binært træ = træ hvor alle lag er helt fyldte.



Et fuldt binært træ af højde h har

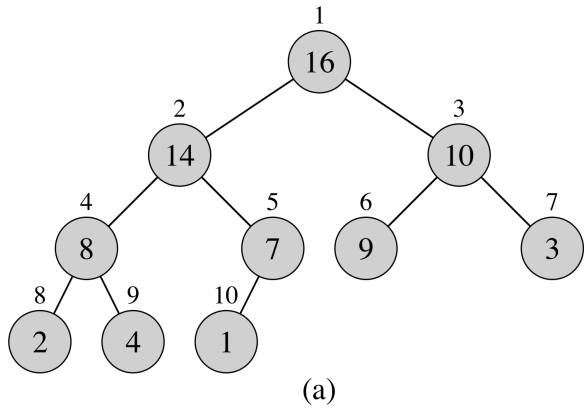
$$1 + 2 + 4 + 8 + \cdots + 2^h = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

knuder (formel A.5 side 1147), heraf 2^h blade.

2) Heap-orden

Et binært træ med værdier i alle knuder er max-heap-ordnet hvis det for enhver knude v med barn u gælder at

$$\text{værdi i } v \geq \text{værdi i } u$$



(a)

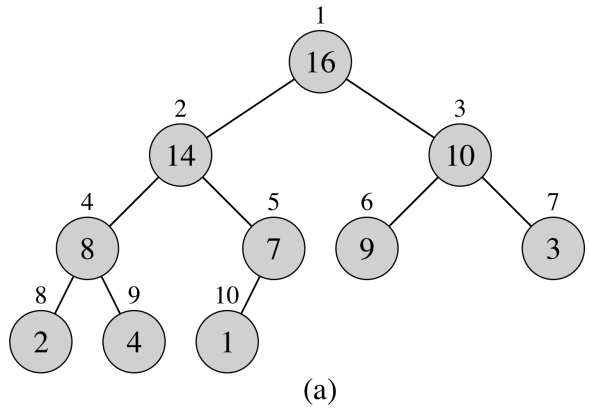
[Bemærk: en ækvivalent definition er, at det for enhver knude knude v med efterkommer u gælder, at værdi i $v \geq$ værdi i u.]

I et max-heap-ordnet træ vil roden indeholde den største værdi i hele heapen.

Et træ er min-heapordnet, hvis ovenstående gælder med $\leq$ i stedet for $\geq$.

3) Heapfacon

Et binært træ har heapfacon hvis alle lag i træet er helt fyldte, undtagen det sidste lag, hvor alle knuder findes længst til venstre. (Specielt har et fuldt træ heapfacon).



(a)

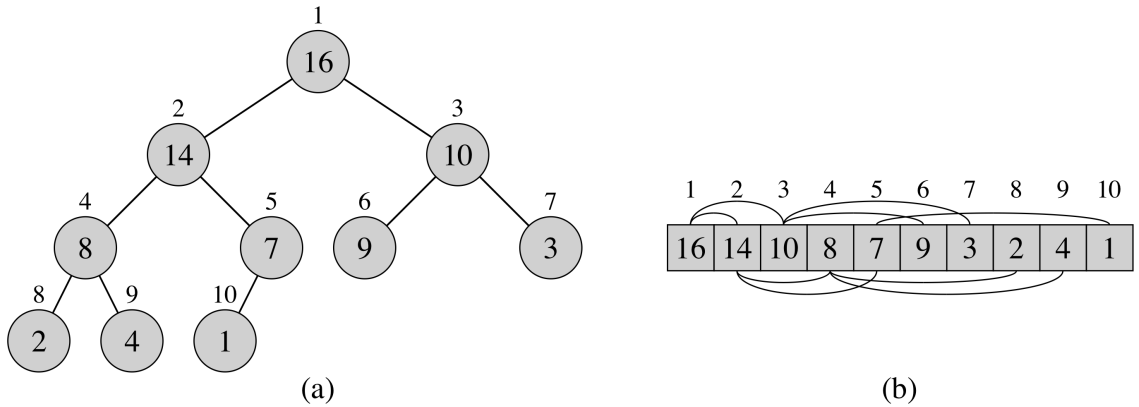
For et træ af heapfacon af højde h med n knuder:

$$n > \text{antal knuder i fuldt træ af højde } h-1 = 2^h - 1$$

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

4) Heap udlagt i et array

Et binært træ i heapfacon kan naturligt udlægges i et array ved at tildele array-indekser til knuder ved et top-down, venstre-til-højre gennemløb af træets lag:



Navigering mellem børn og forældre i array-versionen kan udføres ved simple beregninger: Knuden på plads i har

- Forælder på plads $\lfloor i/2 \rfloor$
- Børn på plads $2i$ og $2i + 1$

(Se figur ovenfor. Formelt bevis til øvelsestimer.)

Operationer på en heap

Vi ønsker at lave følgende operationer på en heap:

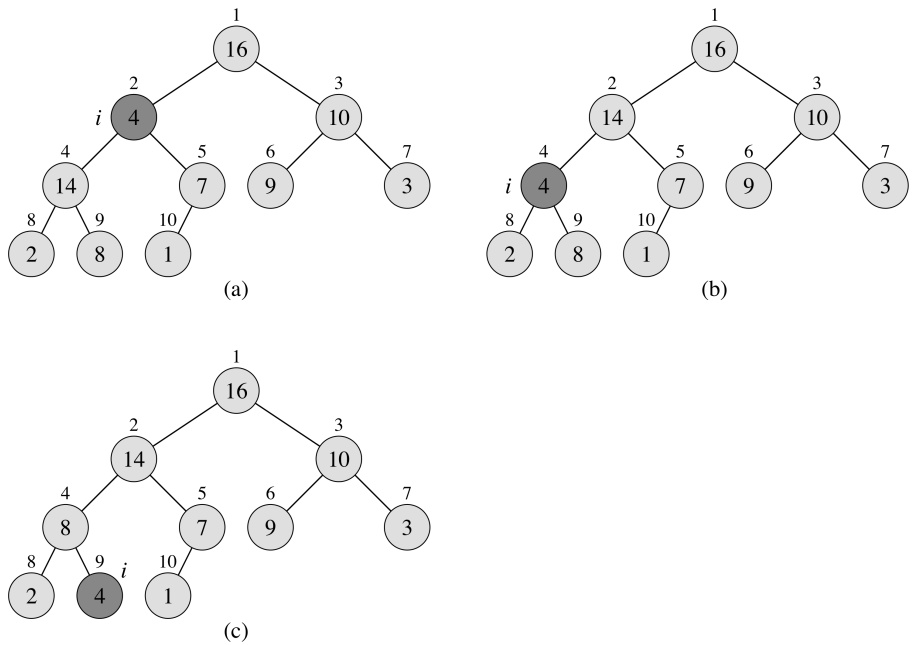
- MAX-HEAPIFY: Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens undertræ til at overholde heap-orden.
- BUILD-MAX-HEAP: Lav n input elementer (uordnede) til en heap.

[Navnene ovenfor er for en max-heap. For en min-heap findes de samme operationer med "min-" i stedet for "max-" i navnet.]

Max-Heapify

Givet en knude med to undertræer, som hver især overholder heap-orden, få hele knudens træ til at overholde heap-orden.

- Problem: knudens værdi mindre end et eller begge børns værdier.
- Løsning: byt plads med barnet med den største værdi, kør derefter MAX-HEAPIFY på dette barn.

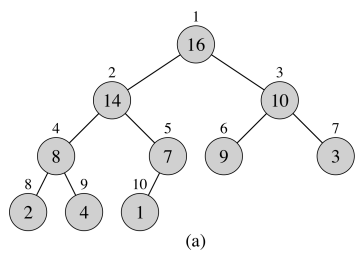


Tid: O(højde af knude).

Max-Heapify

Som pseudo-kode (med indarbejdet check for at man ikke kigger "for langt" i arrayet, dvs. længere end plads n):

MAX-HEAPIFY(A, i, n)
$l = \text{LEFT}(i)$
$r = \text{RIGHT}(i)$
if $l \leq n$ and $A[l] > A[i]$
    $largest = l$
else $largest = i$
if $r \leq n$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
    exchange $A[i]$ with $A[largest]$
    MAX-HEAPIFY$(A, largest, n)$

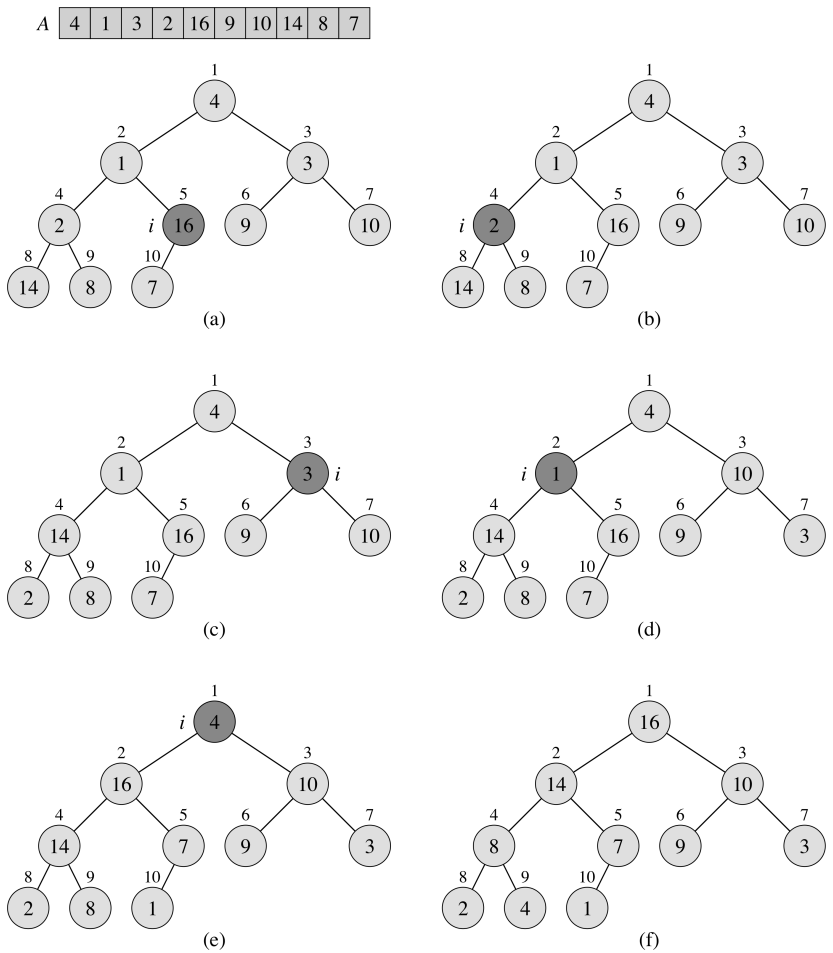


(a)

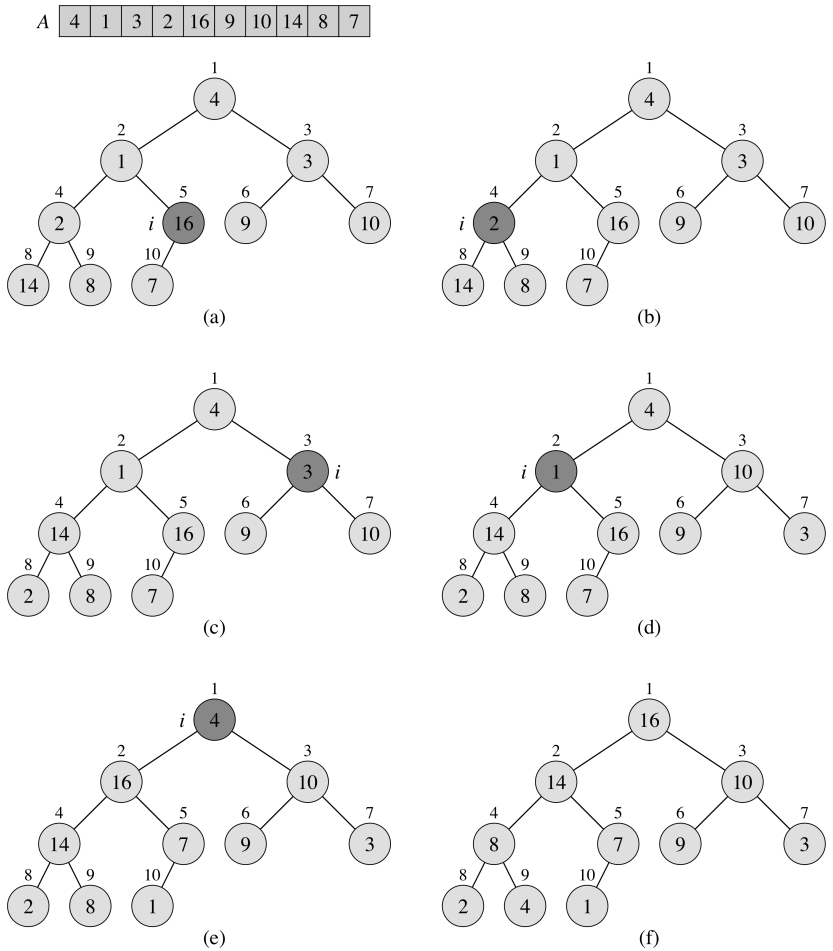
Build-Heap

Lav n input elementer (uordnede) til en heap.

- Ide: arranger elementerne i heap-facon, bring derefter træet i heap-orden nedefra og op.
- Observation: et træ af størrelse én overholder altid heaporder.



Build-Heap

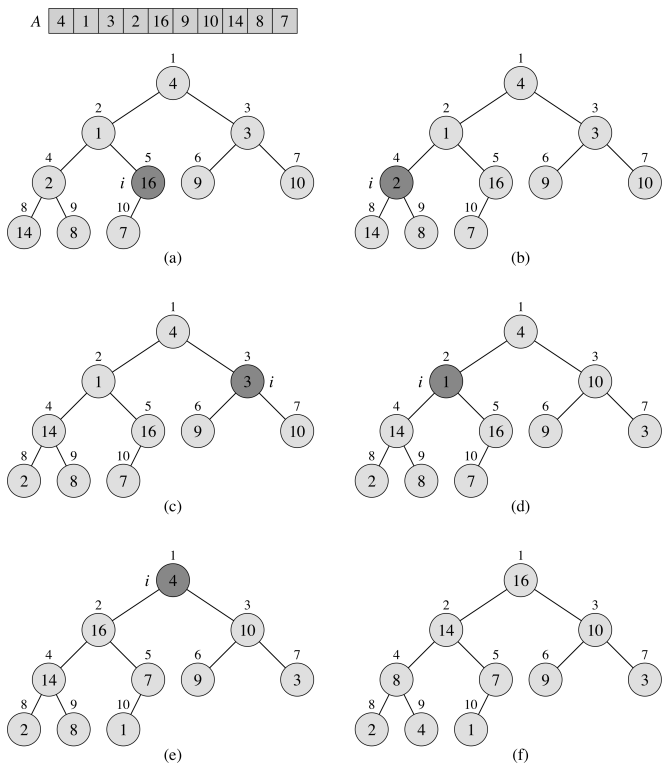


Tid: $O(n \log_2 n)$ klart. Bedre analyse giver $O(n)$.

Build-Heap

Som pseudo-kode:

$$\begin{aligned}&\textsc{Build-Max-Heap}(A, n)\\ &\quad \textbf{for } i = \lfloor n/2 \rfloor \textbf{ downto } 1\\ &\qquad \textsc{Max-Heapify}(A, i, n)\end{aligned}$$



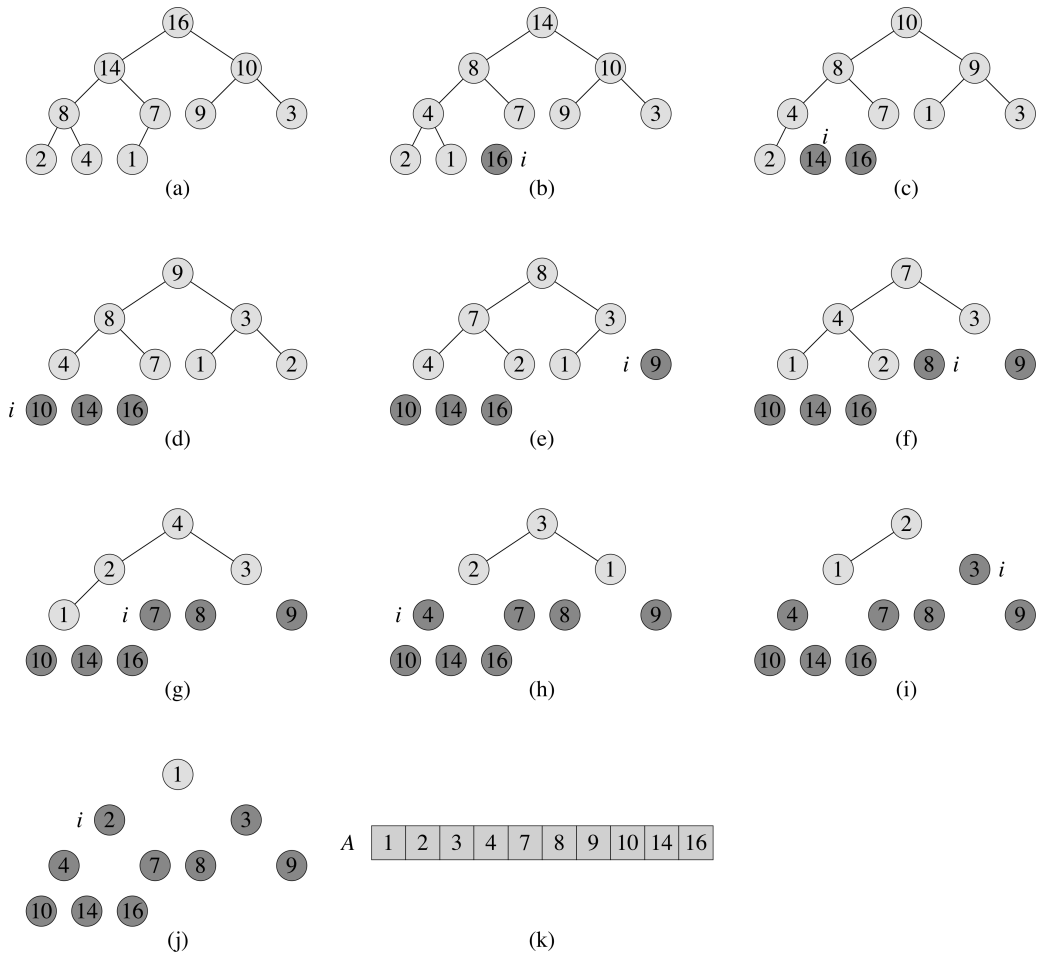
Heapsort

En form for selectionsort hvor der bruges en heap til hele tiden at udtage det største tilbageværende element:

byg en heap
gentag til heap er tom:
  udtag rod (største element i heapen)
  sæt sidste element op som ny rod
  genskab heap-struktur ved MAX-HEAPIFY på ny rod.

Heapsort

Eksempel:



Heapsort

Som pseudo-kode:

```
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)
```

Tid: $O(n) + O(n \log n) = O(n \log n)$

Tre $n \log n$ sorteringsalgoritmer

	Worstcase	Inplace
QuickSort		$\surd$
MergeSort	$\surd$	
HeapSort	$\surd$	$\surd$

Heapsort kører dog langsommere end Mergesort og Quicksort pga. ineffektiv brug af hukommelse (random access).

Introsort [Musser, 1997]: brug Quicksort, men skift under rekursionen til heapsort hvis rekursionen bliver for dyb. Dette giver en inplace, worst case $O(n \log n)$ algoritme, med god køretid i praksis (dette er sorteringsalgoritmen i standardbiblioteket STL for C++).

Asymptotisk analyse af algoritmers køretider

Analyse af køretid

Recall: Vi ønsker at vurdere (analysere) algoritmer på forhånd, inden vi bruger lang tid på at implementere dem.

De to primære **spørgsmål:**

- Løser algoritmen opgaven (er den korrekt)?
- Er algoritmen effektiv (hvad er køretiden)?

Vi fokuserer i disse slides på det andet spørgsmål.

Analyse af køretid

Recall: I vores analyse er en algoritmes køretid lig antallet af basale operationer, som den udfører i RAM-modellen (for worst case input). Dette antal operationer er en voksende funktion $f(n)$ af inputstørrelsen n.

1. Vi vil starte med at undersøge, hvor godt teoretiske analyser i RAM-modellen ser ud til at passe med algoritmers observerede køretid på virkelige computere.
2. Vi vil dernæst introducere et redskab, kaldet asymptotisk analyse, til at sammenligne $f(n)$ for forskellige algoritmer på en tilpas (u)præcis måde.

Mål: at vi kan grovsortere algoritmer efter voksehastigheden af deres køretider, så vi kan undgå at implementere dem, som ikke har en chance for at være hurtigst.

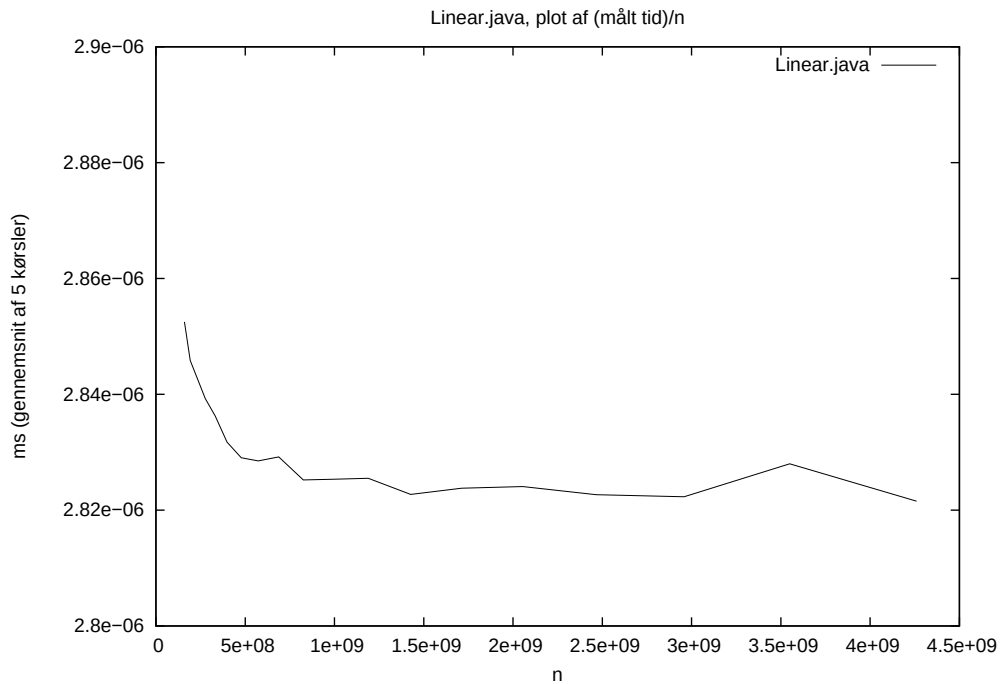
Analyse vs. virkeligheden

```
public class Linear {
  public static void main(String[] args) {

    long time = System.currentTimeMillis();
    long n = Long.parseLong(args[0]);
    long total = 0;
    for(long i=1; i<=n; i++){
      total = total + 1;
    }
    System.out.println(total);
    System.out.println(System.currentTimeMillis() - time);
    }
}
```

Analyse

$$\text{Tid}(n) = c_1 \cdot n + c_0$$

$$\frac{\text{Tid}(n)}{n} = \frac{c_1 \cdot n + c_0}{n} = c_1 + \frac{c_0}{n}$$



Virkelighed

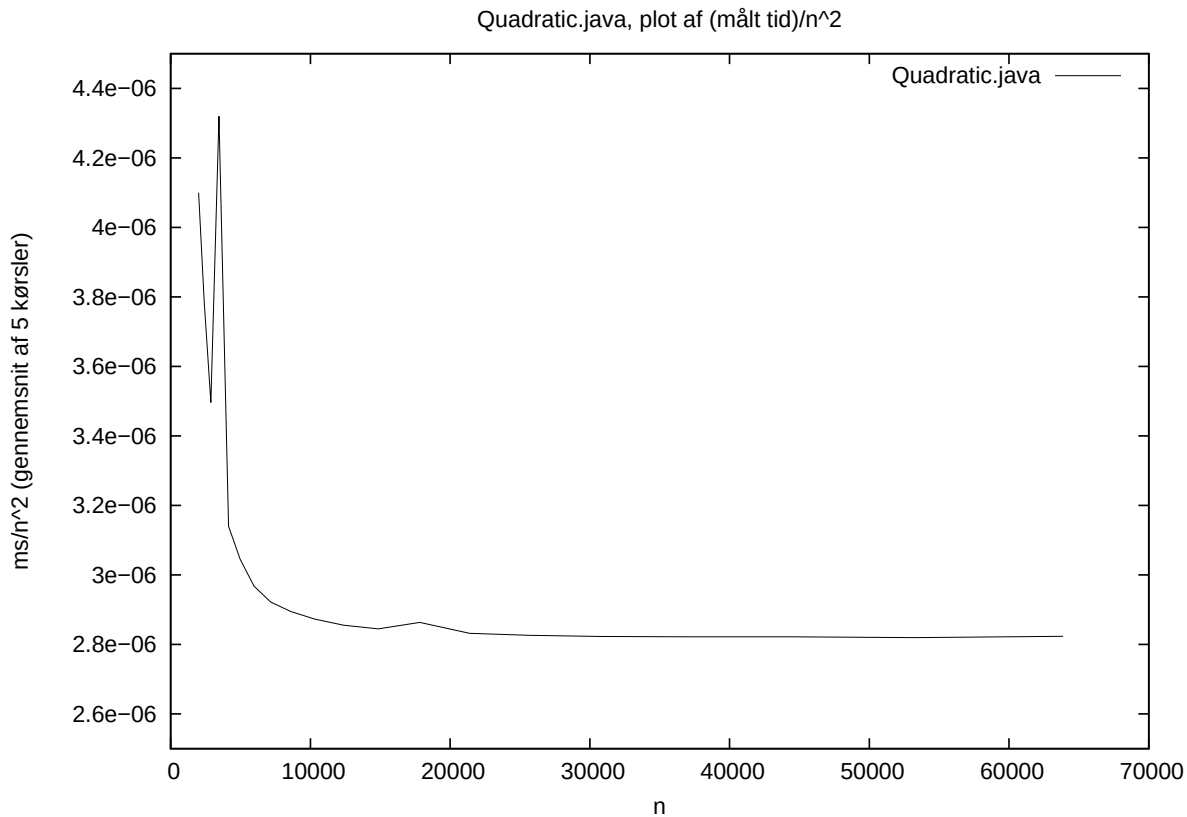
x-akse:
inputstørrelse n

y-akse:
(målt tid)/n

Analyse vs. virkeligheden

```
for(long i=1; i<=n; i++){
  for(long j=1; j<=n; j++){
    total = total + 1;
  }
}
```

$$\begin{aligned}\text{Tid}(n) &= (c_2 \cdot n + c_1) \cdot n + c_0 \\ &= c_2 \cdot n^2 + c_1 \cdot n + c_0\end{aligned}$$



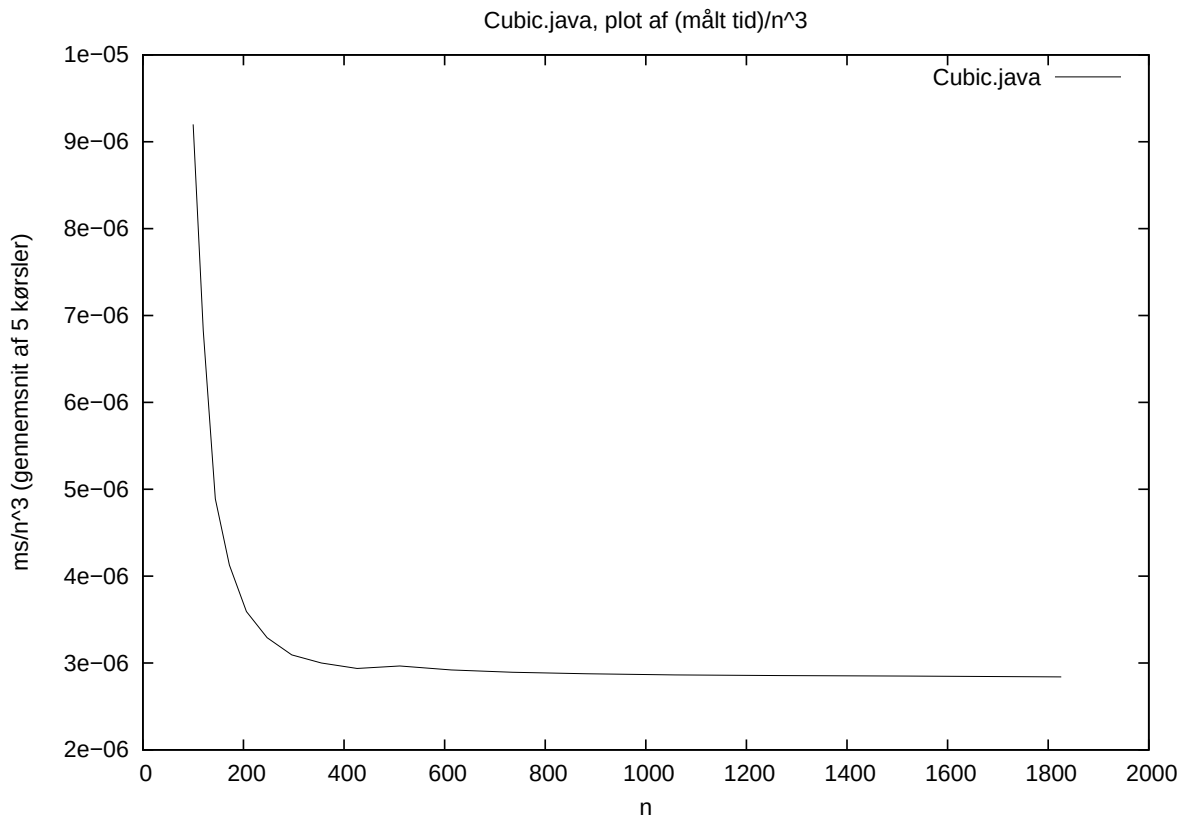
x-akse:
inputstørrelse n

y-akse:
(målt tid)$/n^2$

Analyse vs. virkeligheden

```
for(long i=1; i<=n; i++){
  for(long j=1; j<=n; j++){
    for(long k=1; k<=n; k++){
      total = total + 1;
    }
  }
}
```

$$\begin{aligned}\text{Tid}(n) &= ((c_3 \cdot n + c_2) \cdot n + c_1) \cdot n + c_0 \\ &= c_3 \cdot n^3 + c_2 \cdot n^2 + c_1 \cdot n + c_0\end{aligned}$$



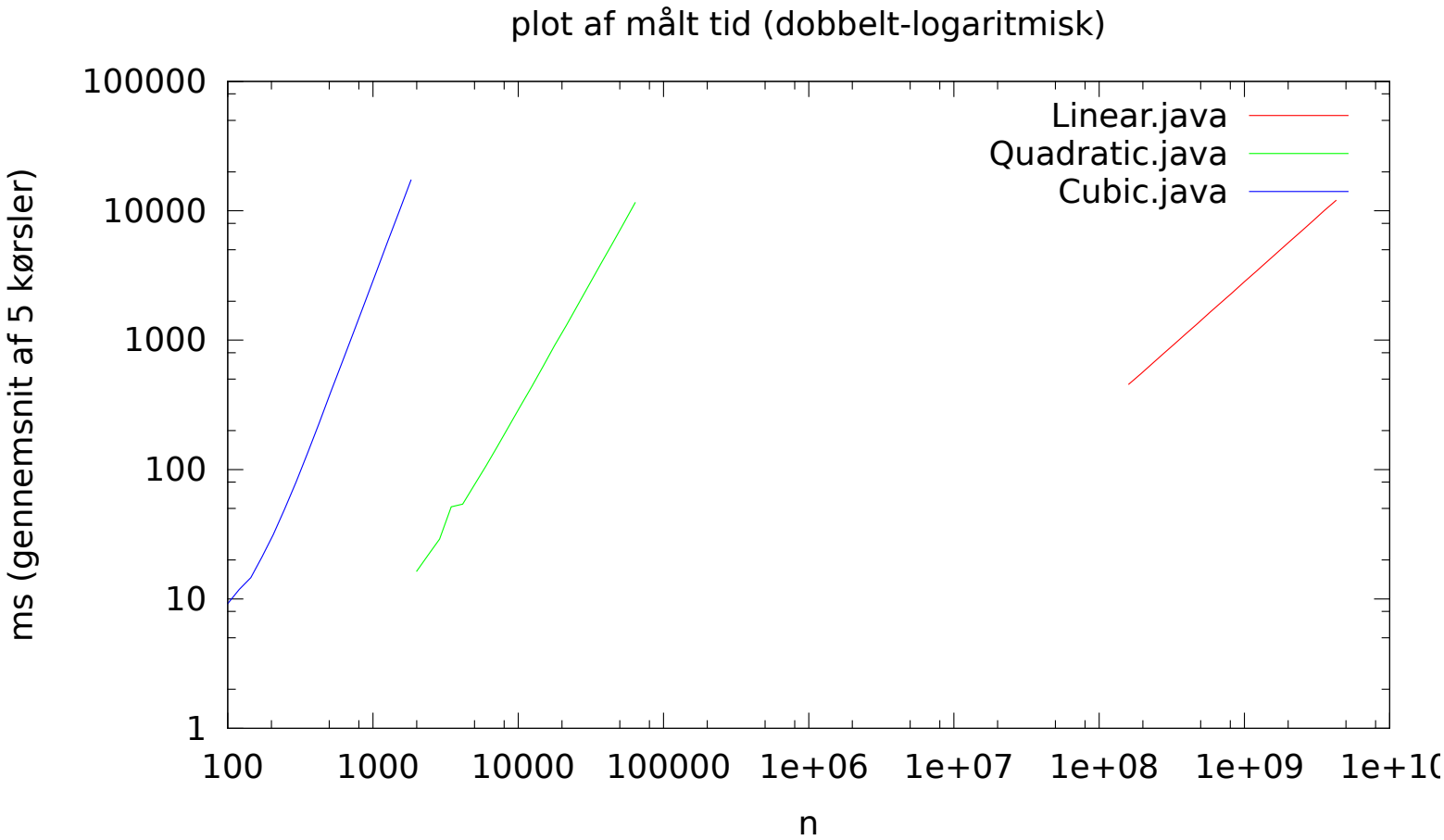
x-akse:
inputstørrelse n

y-akse:
(målt tid)$/n^3$

Analyse vs. virkeligheden

Konklusion: Det ser ud til at analyser i RAM-modellen forudser den rigtige køretid ret godt, i hvert fald for de afprøvede eksempler.

Linear vs. kvadratisk vs. kubisk



Man ser at funktionerne n, n^2 og n^3 står for meget forskellige effektiviter.

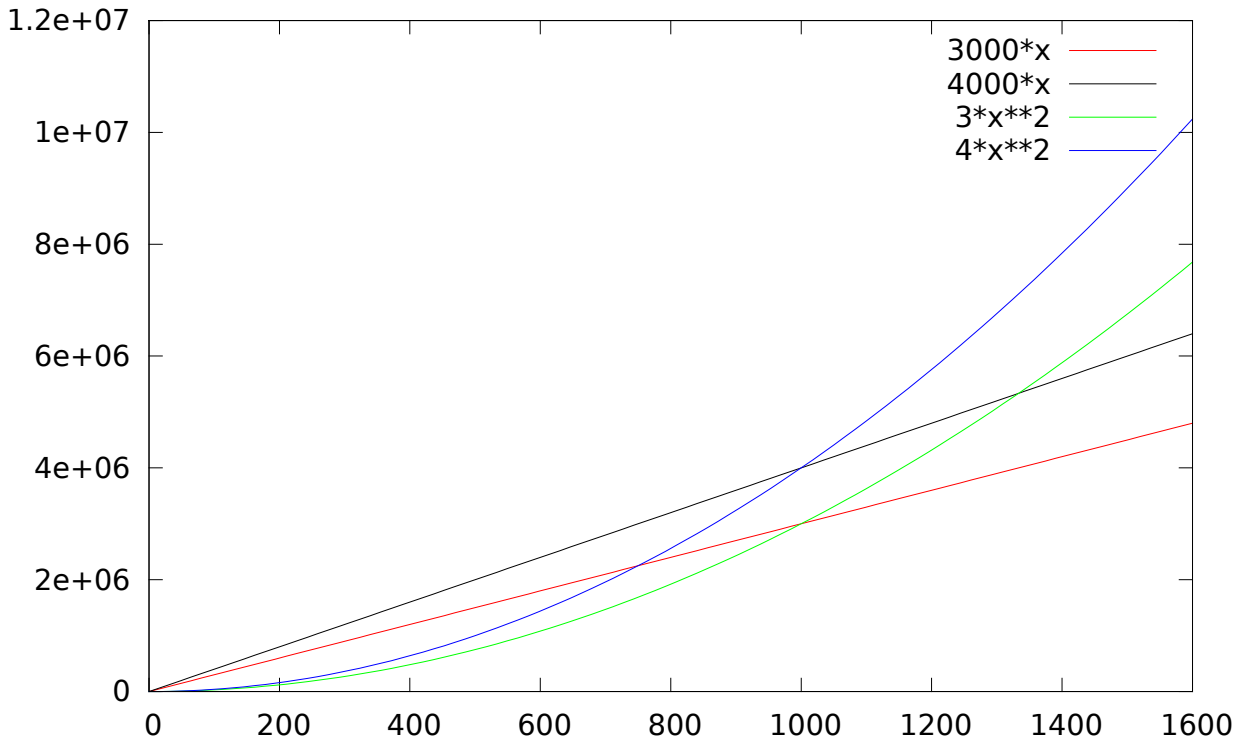
I analysen optræder i virkeligheden en del konstanter (som vi typisk har svært ved at kende præcist), f.eks. $c_1 \cdot n + c_0$. Mon disse betyder noget?

Multiplikative konstanter

Multiplikative konstanter ligegyldige hvis voksehastighed er forskellig:

$f(n) = 3000n$ $\quad$ $h(n) = 3n^2$

$g(n) = 4000n$ $\quad$ $k(n) = 4n^2$



Vinder $3000n$ over $4n^2$? Ja: $\quad 3000n < 4n^2 \Leftrightarrow 3000 < 4n \Leftrightarrow 750 < n$

Faktisk vinder $c_1 \cdot n$ **altid** over $c_2 \cdot n^2$: $\quad c_1 \cdot n < c_2 \cdot n^2 \Leftrightarrow c_1/c_2 < n$

Voksehastighed

Vi ønsker derfor at sammenligne funktioners essentielle voksehastighed på en måde, så der ses bort fra multiplikative konstanter.

En sådan sammenligning kan bruges til at lave en grovsortering af algoritmer, inden vi laver implementationsarbejde.

> Hvis to algoritmer A og B har voksehastigheder, hvor algoritme B altid (for store n) vil tabe til algoritme A, uanset hvilke multiplikative konstanter der er i udtrykkene for voksehastigheder, så vil der som regel ikke være nogen pointe i at implementere algoritme B.

Bemærk: I ovenstående situation behøver vi ikke kende konstanterne for at kunne lave denne vurdering. Vi kan derfor lave køretidsanalyse uden at bekymre os om at kende størrelsen af de indgående konstanter præcist.

Asymptotisk notation

Så vi ønsker et værktøj til at sammenligne funktioners essentielle voksehastighed på en måde, så der ses bort fra multiplikative konstanter.

Princippet for vores værktøj vil være følgende: for en funktion $f(n)$ vil vi betragte alle skaleringer af den

$$\{c \cdot f(n) \mid \text{alle } c > 0\}$$

som lige gode.

I det følgende vil vi kalde denne mængde af funktioner for $f(n)$'s klasse.

Asymptotisk notation

Baseret på dette princip definerer vi for voksehastighed for funktioner fem relationer svarende til de fem klassiske ordens-relationer:

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

De vil, af historiske årsager, blive kaldt for:

$$O \quad \Omega \quad \Theta \quad o \quad \omega$$

Hvilket udtales således:

"O", "Omega", "Theta", "lille o", "lille omega"

De fem definitioner beskrives over de næste fem sider.

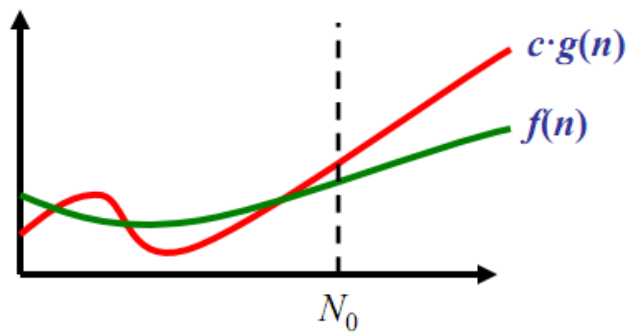
O

Definition: $f(n) = O(g(n))$

hvis $f(n)$ og $g(n)$ er funktioner $N \longrightarrow R$ og

findes $c > 0$ og N_0 så *for alle* $n \geq N_0$:

$$f(n) \leq c \cdot g(n)$$



Mening: $f \leq g$ i voksehastighed

Princip: $f(n)$ vokser højst så hurtigt som funktioner fra $g(n)$'s klasse.

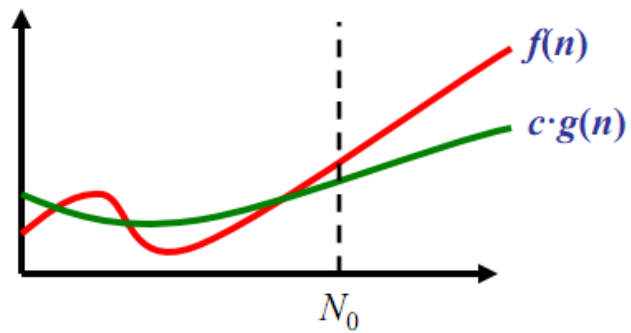
Omega

Definition: $f(n) = \Omega(g(n))$

hvis $f(n)$ og $g(n)$ er funktioner $N \rightarrow R$ og

findes $c > 0$ og N_0 så *for alle* $n \geq N_0$:

$$f(n) \geq c \cdot g(n)$$



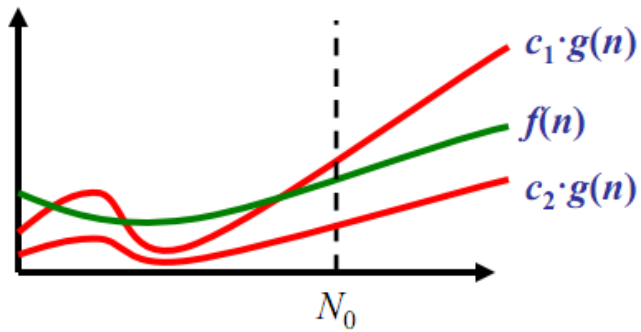
Mening: $f \geq g$ i voksehastighed

Princip: $f(n)$ vokser mindst så hurtigt som funktioner fra $g(n)$'s klasse.

Theta

Definition: $f(n) = \theta(g(n))$

hvis $f(n)$=O($g(n)$) og $f(n)$= $\Omega(g(n)$



Mening: $f = g$ i voksehastighed

Princip: $f(n)$ vokser lige så hurtigt som funktioner fra $g(n)$'s klasse.

Lille o

Definition: $f(n) = o(g(n))$

hvis $f(n)$ og $g(n)$ er funktioner $N \longrightarrow R$ og

for alle $c > 0$, *findes* N_0 så *for alle* $n \geq N_0$:

$$f(n) \leq c \cdot g(n)$$

Mening: $f < g$ i voksehastighed

Princip: $f(n)$ vokser langsommere end alle funktioner fra $g(n)$'s klasse.

Lille omega

Definition: $f(n) = \omega(g(n))$

hvis $f(n)$ og $g(n)$ er funktioner $N \longrightarrow R$ og

for alle $c > 0$, *findes* N_0 så *for alle* $n \geq N_0$:

$$f(n) \geq c \cdot g(n)$$

Mening: $f > g$ i voksehastighed

Princip: $f(n)$ vokser hurtigere end alle funktioner fra $g(n)$'s klasse.

Asymptotisk notation

Man kan nemt vise, at disse definitioner opfører sig som forventet af ordens-relationer. F.eks.:

$$f(n) = o(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x < y \Rightarrow x \leq y)$$

$$f(n) = \Theta(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x = y \Rightarrow x \leq y\ )$$

$$f(n) = O(g(n)) \;\Leftrightarrow\; g(n) = \Omega(f(n)) \qquad (\text{jvf. } x \leq y \Leftrightarrow y \geq x\ )$$

$$f(n) = o(g(n)) \;\Leftrightarrow\; g(n) = \omega(f(n)) \qquad (\text{jvf. } x < y \Leftrightarrow y > x\ )$$

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

Asymptotisk analyse i praksis

De asymptotiske forhold mellem de fleste funktioner f og g kan afklares ved følgende to sætninger (som kan vises ud fra definitionerne):

$$\text{Hvis } \frac{f(n)}{g(n)} \to k > 0 \text{ for } n \to \infty \text{ så gælder } f(n) = \Theta(g(n)) \qquad (1)$$

$$\text{Hvis } \frac{f(n)}{g(n)} \to 0 \text{ for } n \to \infty \text{ så gælder } f(n) = o(g(n)) \qquad (2)$$

Eksempler:

$$\frac{20n^2 + 17n + 312}{n^2} = \frac{20 + 17/n + 312/n^2}{1} \to \frac{20 + 0 + 0}{1} = 20 \text{ for } n \to \infty$$

$$\frac{20n^2 + 17n + 312}{n^3} = \frac{20/n + 17/n^2 + 312/n^3}{1} \to \frac{0 + 0 + 0}{1} = 0 \text{ for } n \to \infty$$

Asymptotisk analyse i praksis

Derudover er det godt at kende følgende fact fra matematik:

$$\text{For alle } a > 0 \text{ og } b > 1 \text{ gælder } \frac{n^a}{b^n} \to 0 \text{ for } n \to \infty \tag{3}$$

Dvs. ethvert polynomium er $o()$ af enhver exponentialfunktion

Eksempelvis giver dette at:

$$\frac{n^{100}}{2^n} \to 0 \text{ for } n \to \infty$$

hvoraf ses

$$n^{100} = o(2^n)$$

Asymptotisk analyse i praksis

Samt følgende fact:

$$\text{For alle } a, d > 0 \text{ og } c > 1 \text{ gælder } \frac{(\log_c n)^a}{n^d} \to 0 \text{ for } n \to \infty \qquad (4)$$

> Dvs. enhver logaritme (selv opløftet i enhver potens) er $o()$ af ethvert polynomium.

Eksempelvis giver dette at:

$$\frac{(\log n)^3}{n^{0.5}} \to 0 \text{ for } n \to \infty, \text{ hvoraf ses } (\log n)^3 = o(n^{0.5})$$

Asymptotisk analyse i praksis

Regel (4) kan i øvrigt nemt ses at være en variant af regel (3) [dette er ikke pensum]:

For $c > 1$ og $d > 0$, sæt $N = \log_c(n)$ og $b = c^d$. Så haves

$$\frac{(\log_c n)^a}{n^d} = \frac{N^a}{(c^{\log_c(n)})^d} = \frac{N^a}{c^{d \log_c(n)}} = \frac{N^a}{(c^d)^{\log_c(n)}} = \frac{N^a}{(c^d)^N} = \frac{N^a}{b^N}$$

Da $\log_c(n)$ er en voksende og ubegrænset funktion, gælder at $n \to \infty$ er det samme som $N = \log_c(n) \to \infty$.

Eksempler på funktioner for voksehastighed

Med regel (1)–(4) kan man vise, at følgende funktioner er sat i stigende voksehastighed (mere præcist, at den ene er $o()$ af den næste):

$$1, \quad \log n, \quad \sqrt{n}, \quad n, \quad n \log n,$$

$$n\sqrt{n}, \quad n^2, \quad n^3, \quad n^{10}, \quad 2^n$$

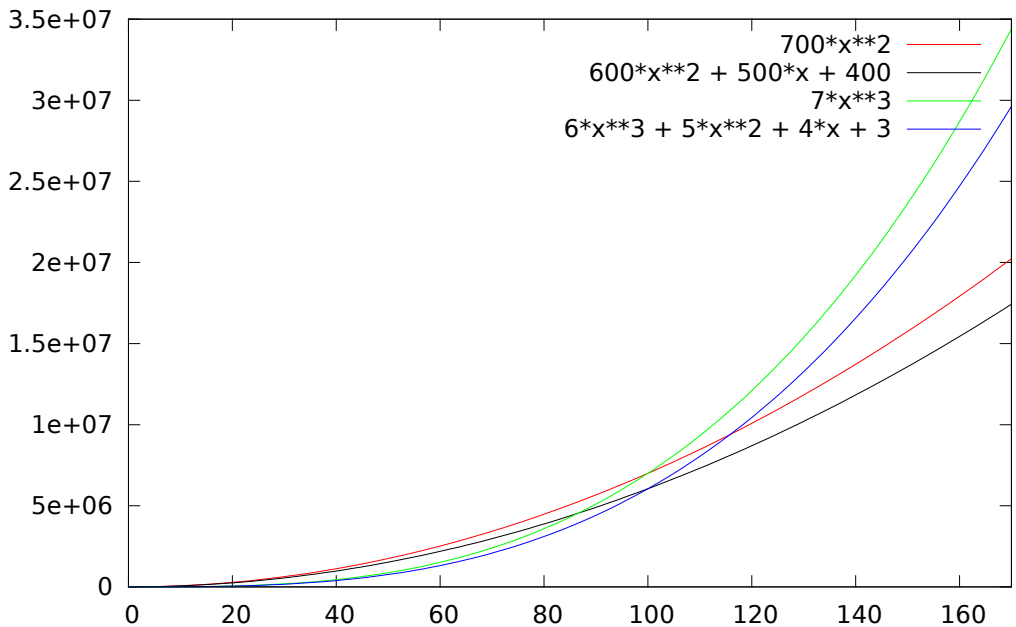
(Dette er en opgave til øvelsestimerne.)

Dominerende led

For funktioner med flere led (dele med plus imellem), vil leddet/ledene med størst voksehastighed bestemme samlet voksehastighed. Eksempel:

$$f(n) = 700n^2 \qquad g(n) = 7n^3$$

$$h(n) = 600n^2 + 500n + 400 \qquad k(n) = 6n^3 + 5n^2 + 4n + 3$$



Figuren passer med beregninger:

Dominerende led

$$\frac{6n^3+5n^2+4n+3}{7n^3} = \frac{6+5/n+4/n^2+3/n^3}{7} \to \frac{6+0+0+0}{7} = 6/7 \text{ for } n \to \infty$$

Dvs. $6n^3+5n^2+4n+3 = \Theta(7n^3)$

$$\frac{600n^2+500n+400}{700n^2} = \frac{600+500/n+400/n^2}{700} \to \frac{600+0+0}{700} = 6/7 \text{ for } n \to \infty$$

Dvs. $600n^2+500n+400 = \Theta(700n^2)$

$$\frac{600n^2+500n+400}{6n^3+5n^2+4n+3} = \frac{600/n+500/n^2+400/n^3}{6+5/n+4/n^2+3/n^3} \to \frac{0+0+0}{6+0+0+0} = 0 \text{ for } n \to \infty$$

Dvs. $600n^2+500n+400 = o(6n^3+5n^2+4n+3)$

Eksempel 1 på asymptotisk analyse af algoritmer

Hvad er den asymptotiske køretid af følgende algoritme?

```
ALGORITME1(n)
    i = 1
    while i ≤ n
        i = i + 2
```

Løkken har $\lceil n/2 \rceil = \Theta(n)$ iterationer.

Hver iteration tager mellem c_1 og c_2 tid for (ukendte) konstanter c_1 og c_2. Dvs. hver iteration tager $\Theta(1)$ tid.

Derfor er køretiden $\Theta(n \cdot 1) = \Theta(n)$.

Vi har her undladt at snakke om tiden for den første linje samt for initialisering af løkken. En mere præcis analyse vil give et udtryk af typen $c_1 \cdot n + c_0$, men vi undlader at snakke om led, som klart er domineret af andre led.

Eksempel 2 på asymptotisk analyse af algoritmer

Hvad er den asymptotiske køretid af følgende algoritme?

```
ALGORITME2(n)
    s = 0
    for i = 1 to n
        for j = i downto 1
            s = s + 1
```

Der er n iterationer af den yderste løkke. For hver af disse er der højst n iterationer af den inderste løkke. Hver iteration af den inderste løkke tager $\Theta(1)$ tid. Så køretiden er $O(n \cdot n \cdot 1) = O(n^2)$.

For $i \geq n/2$ (dvs. for $n/2$ iterationer af den yderste løkke) er der mindst $n/2$ iterationer af den inderste løkke. Så køretiden er $\Omega(n/2 \cdot n/2 \cdot 1) = \Omega(n^2)$.

Derfor er køretiden alt i alt $\Theta(n^2)$.

[Alternativ analyse: den inderste løkke kører $(1 + 2 + 3 + \cdots + n) = (n+1)n/2 = \Theta(n^2)$ gange.]

Eksempel 3 på asymptotisk analyse af algoritmer

Hvad er den asymptotiske køretid af følgende algoritme?

```
ALGORITME3(n)
    s = 0
    for i = 1 to n
        for j = i to n
            for k = i to j
                s = s + 1
```

Der er n iterationer af den yderste løkke. For hver af disse er der højst n iterationer af den midterste løkke. For hver af disse er der højst n iterationer af den inderste løkke. Hver iteration af den inderste løkke tager $\Theta(1)$ tid. Så køretiden er $O(n \cdot n \cdot n \cdot 1) = O(n^3)$.

For $i \leq n/4$ (dvs. for $n/4$ iterationer af den yderste løkke) er der $n/4$ iterationer af den midterste løkke med $j \geq 3n/4$. For disse gælder $j - i \geq n/2$, så for disse har den inderste løkke mindst $n/2$ iterationer. Så køretiden er $\Omega(n/4 \cdot n/4 \cdot n/2 \cdot 1) = \Omega(n^3)$.

Derfor er køretiden alt i alt $\Theta(n^3)$.

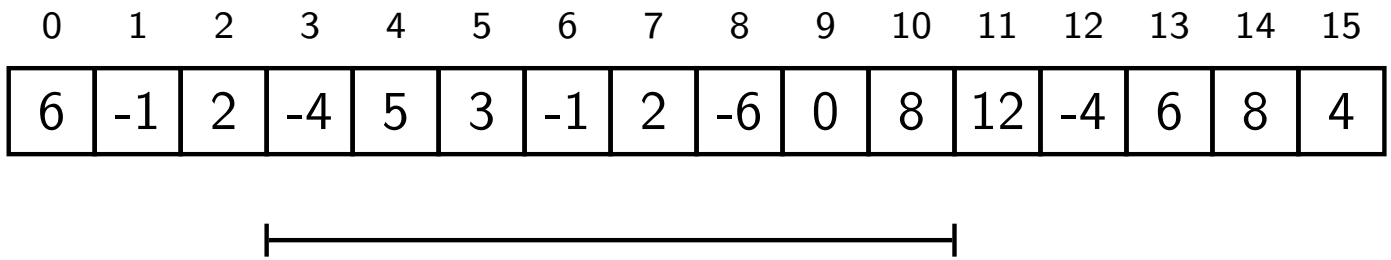
Opsummering

Arbejdsprincip: Sammenlign først voksehastigheder af algoritmer via asymtotisk analyse, og implementer normalt kun den med laveste voksehastighed. For to algoritmer med samme voksehastighed, implementer begge og mål deres køretider.

Algoritmer for Max Sum Problemet

Maximum Sum problemet

Givet et array (liste) af tal kan vi se på summer af segmenter (del-arrays).



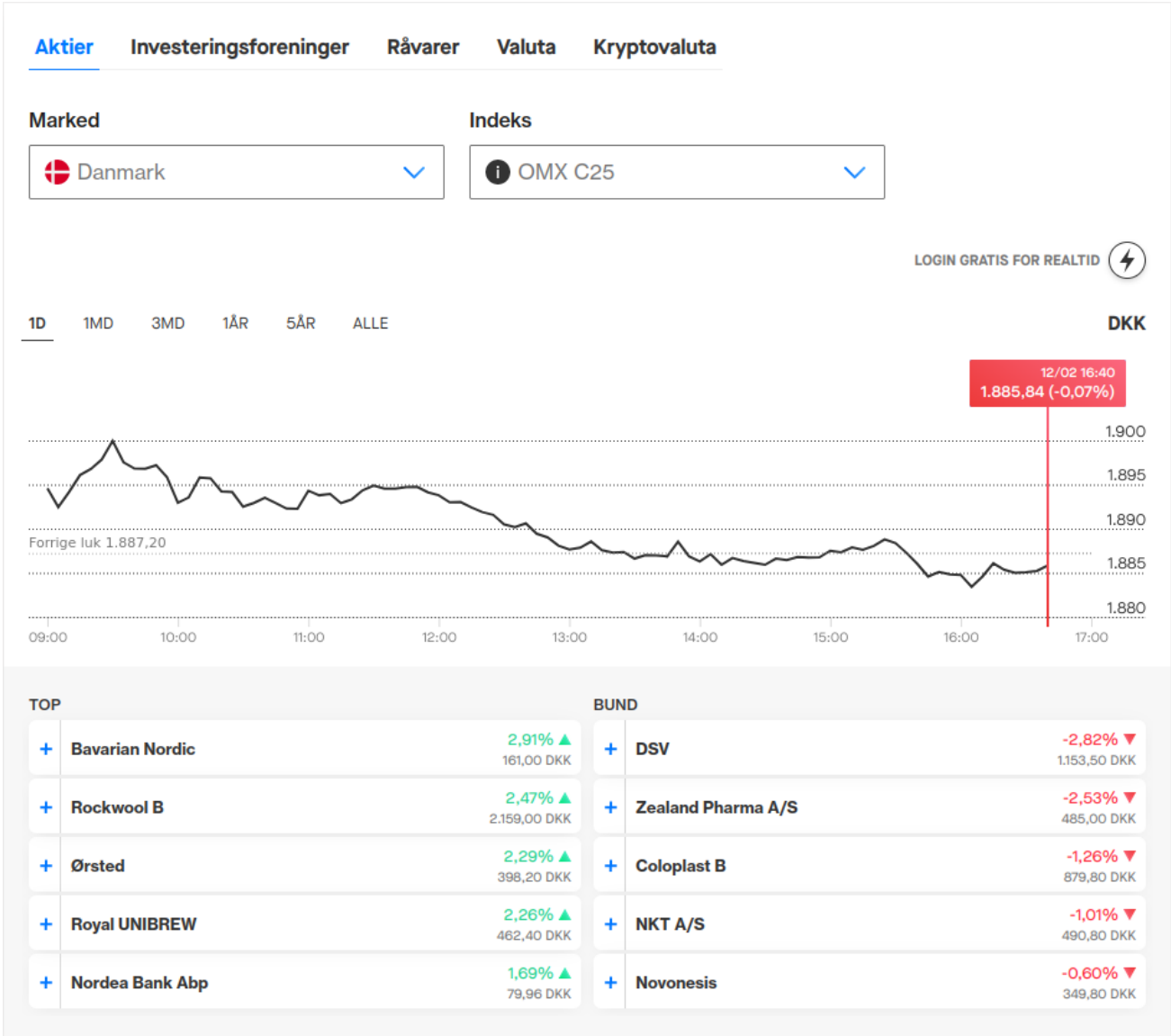
I segmentet ovenfor er summen

$$(-4) + 5 + 3 + (-1) + 2 + (-6) + 0 + 8 = 7$$

Spørgsmål: Hvilket segment har størst sum?

Et simpelt og fundamentalt problem

Mere motivation for MaxSum problemet: aktieanalyse

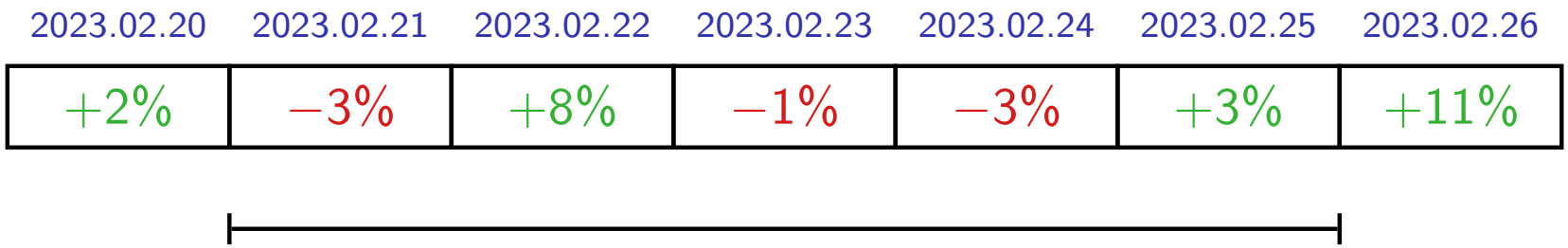


(Fra www.euroinvestor.dk)

Aktieanalyse

Vi har data af følgende type:

Aktie for Firma X:



Spørgsmål: I hvilken periode har det været bedst at eje aktien?

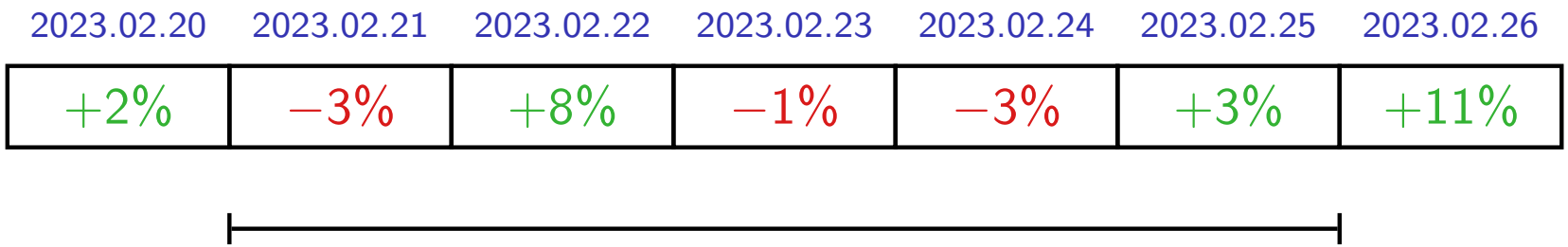
Procentregning

Hvis 1000 kr. stiger med 3% bliver det til $1000 \cdot 1.03 = 1030$ kr.

Hvis 1000 kr. falder med 2% bliver det til $1000 \cdot 0.98 = 980$ kr.

Hvis 1000 kr. først stiger med 3% og derefter falder med 2% bliver det til

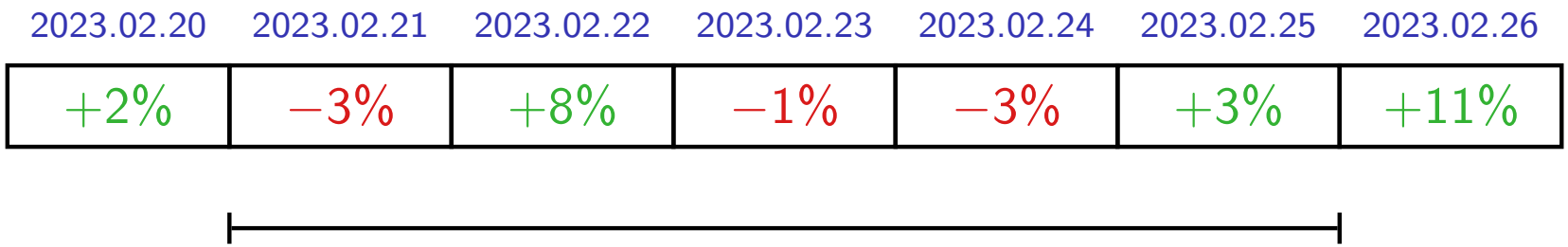
$$1000 \cdot 1.03 \cdot 0.98 \ (= 1009.40) \text{ kr.}$$



I perioden ovenfor har aktien forandret sig med en faktor

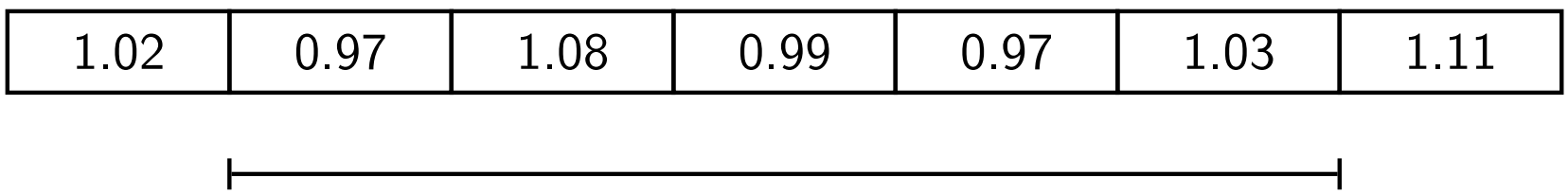
$$0.97 \cdot 1.08 \cdot 0.99 \cdot 0.97 \cdot 1.03$$

Aktieanalyse



Spørgsmål: I hvilken periode har det været bedst at eje aktien?

$$\Downarrow$$



Spørgsmål: Hvilket segment har størst produkt?

Fra maximum produkt til maximum sum

Logaritmer er voksende funktioner. Så

$$0.94 \cdot 1.05 \cdot 0.99 \leq 0.96 \cdot 1.03 \cdot 1.01$$

hvis og kun hvis

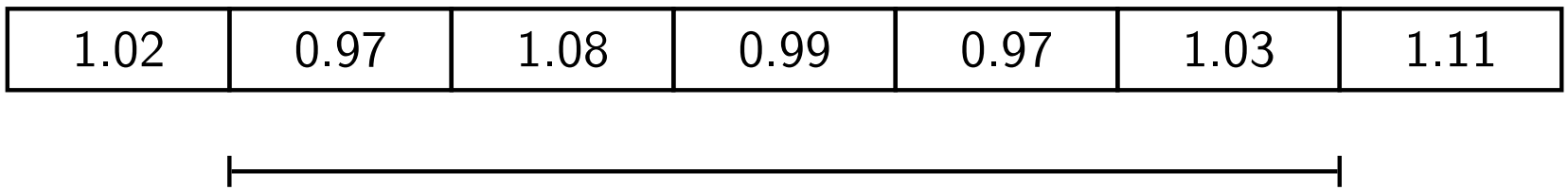
$$\log(0.94 \cdot 1.05 \cdot 0.99) \leq \log(0.96 \cdot 1.03 \cdot 1.01)$$

Da $\log(x \cdot y) = \log(x) + \log(y)$, gælder ovenstående hvis og kun hvis

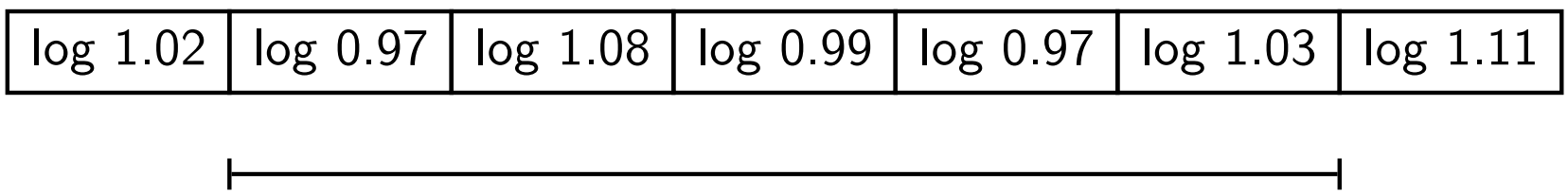
$$\log(0.94) + \log(1.05) + \log(0.99) \leq \log(0.96) + \log(1.03) + \log(1.01)$$

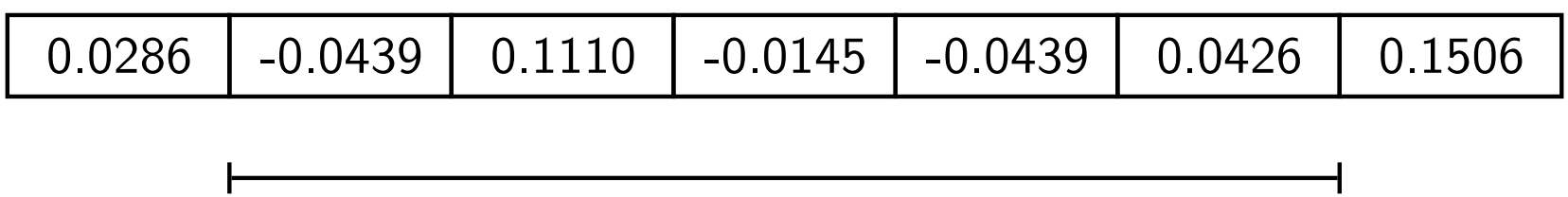
Fra maximum produkt til maximum sum

Så segmentet, der har størst produkt i dette array:



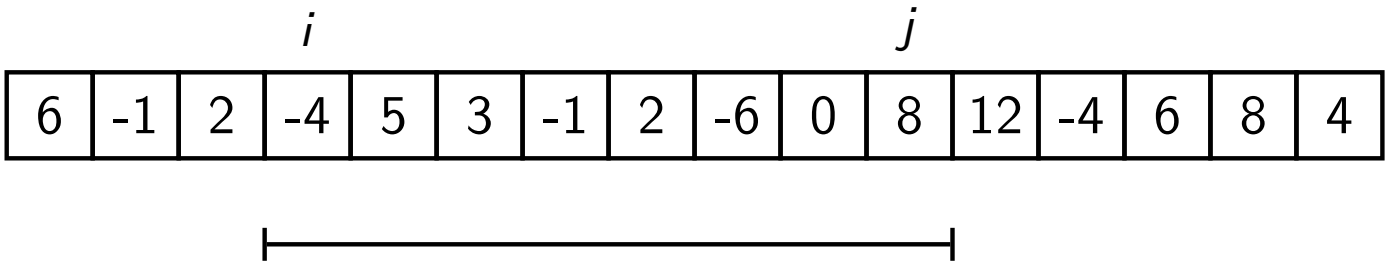
er det samme som segmentet der har størst sum i dette array:





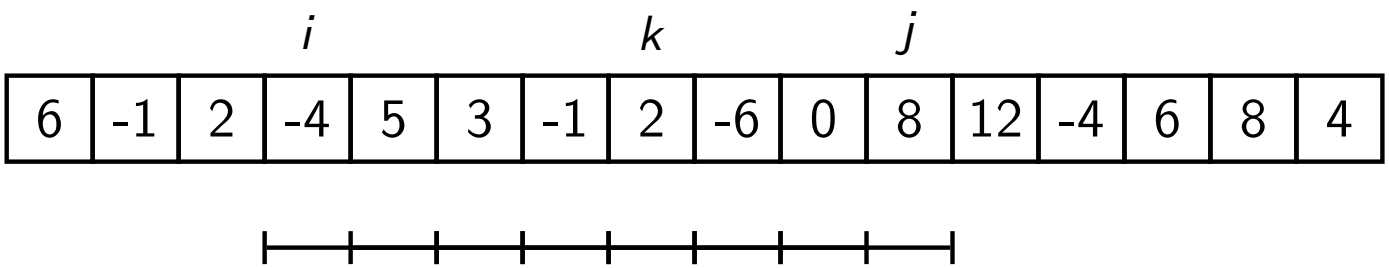
Algoritmer for MaxSum

Vi skal finde summen for alle segmenter:



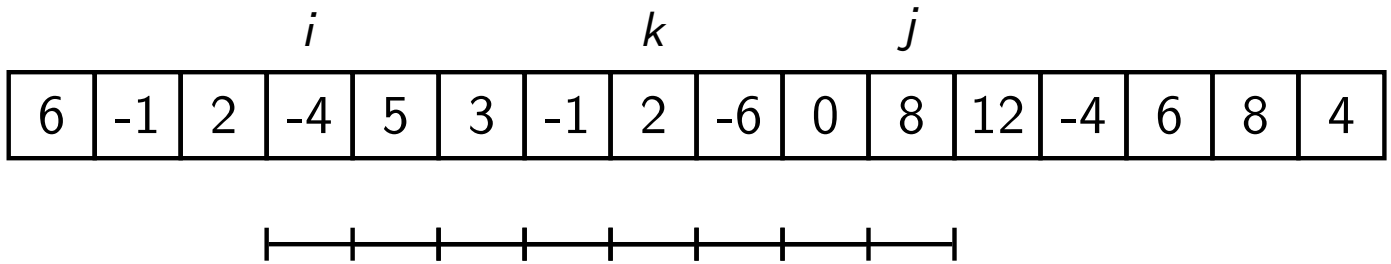
Naturlig algoritme ud fra definitionen:

For alle i og for alle $j \geq i$, lav summen fra i til og med j.



Første algoritme for MaxSum

```
MaxSum1(n)
    maxSoFar = 0
    for i = 0 to n − 1
        for j = i to n − 1
            sum = 0
            for k = i to j
                sum += A[k]
            maxSoFar = max(maxSoFar, sum);
    return maxSoFar
```



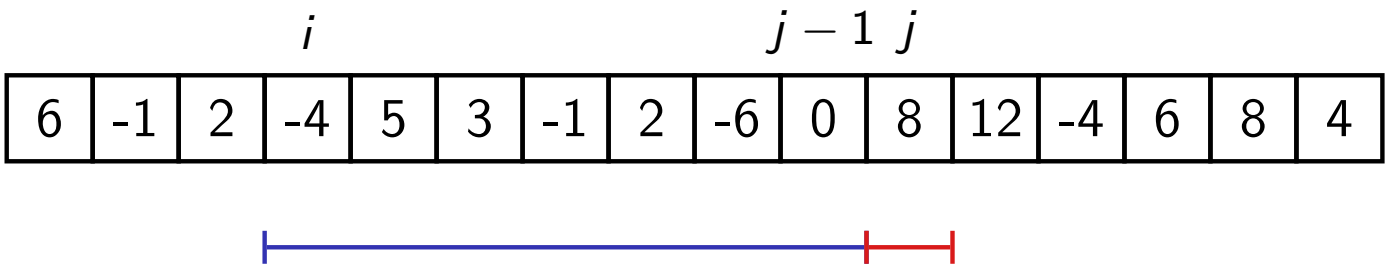
Korrekt? Følger af definition af problem. Køretid? $\Theta(n^3)$, med samme argument som for Algoritme 3 fra asymptotisk analyse eksemplerne (de har helt samme struktur).

Observation

$$(-4)+5+3+(-1)+2+(-6)+0=(-1)$$

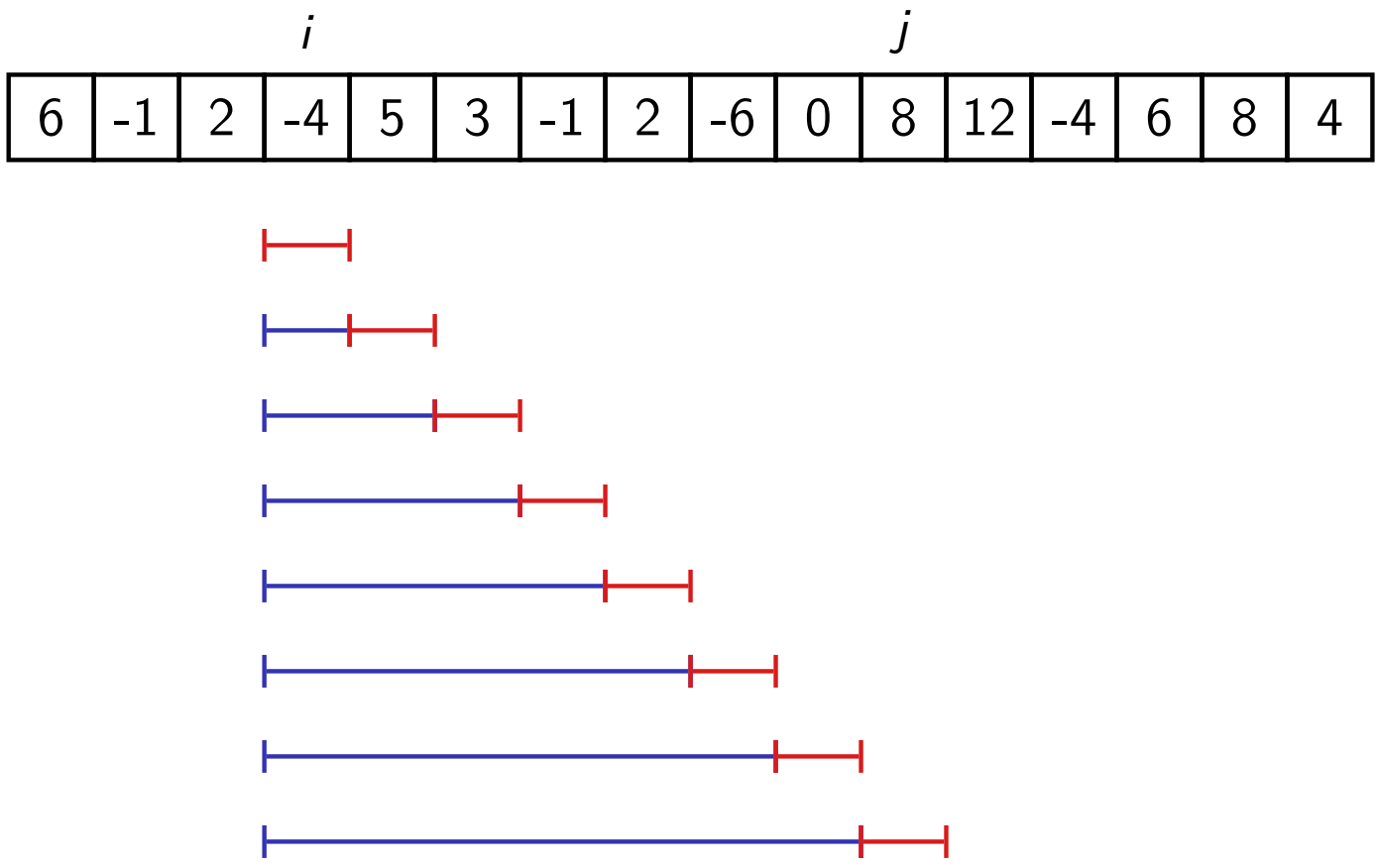
$$\Downarrow$$

$$(-4)+5+3+(-1)+2+(-6)+0+8=(-1)+8=7$$



Idé til forbedret algoritme

Algoritme: For hvert i, beregn summer for stigende j med én ny addition per sum.



Anden algoritme for MaxSum

```
MaxSum2(n)
    maxSoFar = 0
    for i = 0 to n − 1
        sum = 0
        for j = i to n − 1
            sum += A[j]
            maxSoFar = max(maxSoFar, sum);
    return maxSoFar
```

Korrekt? Følger af definition af problem samt observationen ovenfor.

Køretid? $\Theta(n^2)$, med ca. samme argument som for Algoritme 2 fra asymptotisk analyse eksemplerne.

Ny observation

$$x_1 \leq x_2$$

$$\Updownarrow$$

$$x_1 + 2 \leq x_2 + 2$$

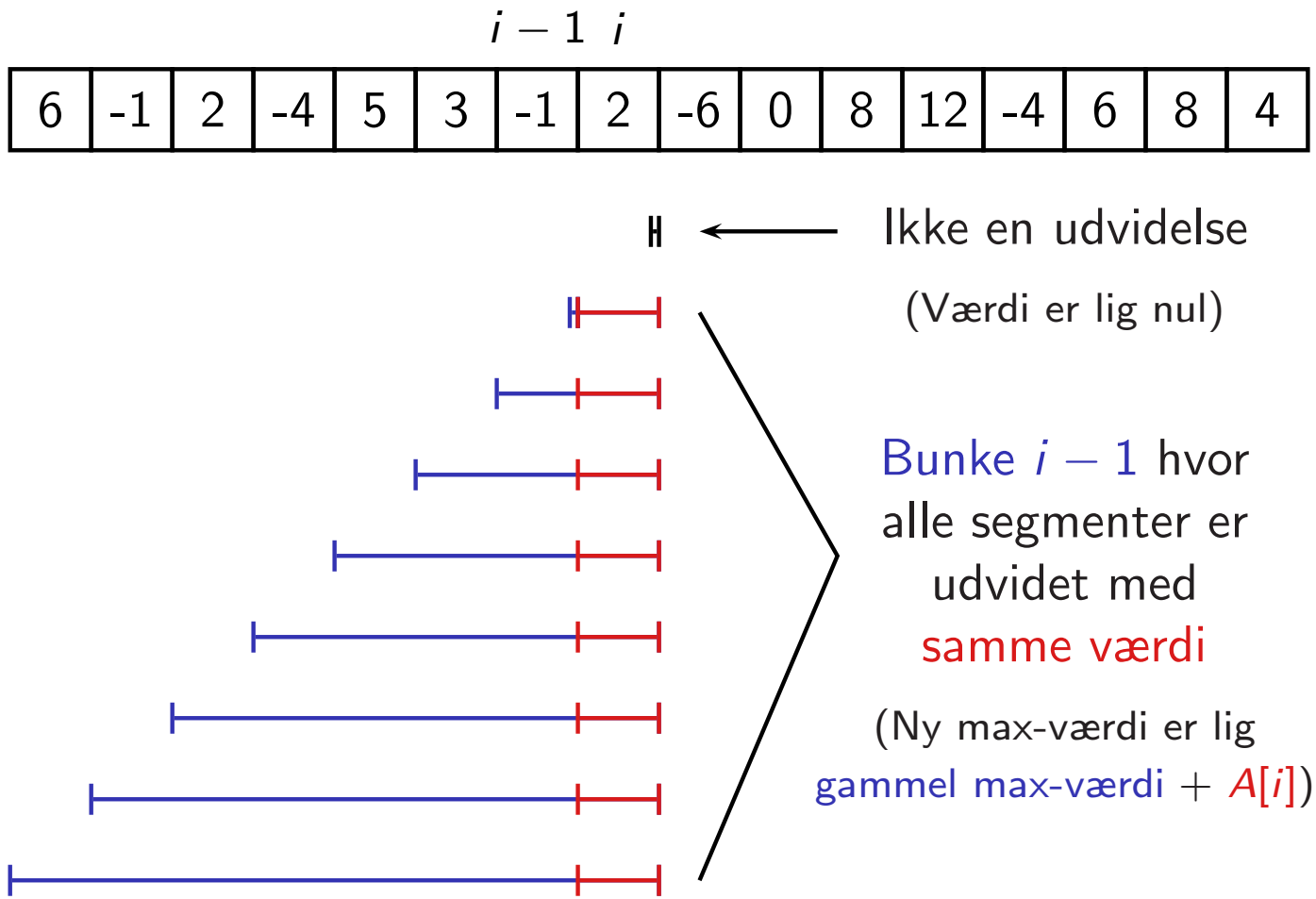
Heraf følger:

$$\max\{x_1 + 2, x_2 + 2, \ldots, x_i + 2\} = \max\{x_1, x_2, \ldots, x_i\} + 2$$

Idé: Kan vi kigge på segmenter i bunker, således at den nye bunke er lig den gamle bunke med alle segmenter udvidet med den samme værdi?

Idé til forbedret algoritme

Lad bunke i være alle segmenter, som ender ved $A[i]$('s højre kant). Så er bunke i det samme som bunke $i-1$ med alle segmenter udvidet med den samme værdi, plus det tomme segment:



Tredie algoritme for MaxSum

```
MaxSum3(n)
    maxSoFar = 0
    maxEndingHere = 0
    for i = 0 to n − 1
        maxEndingHere = max(maxEndingHere + A[i], 0)
        maxSoFar = max(maxSoFar, maxEndingHere);
    return maxSoFar
```

Korrekt? Følger af definition af problem samt den nye observation ovenfor, som sikrer, at vi tager maksimum over alle segmenter (da ethvert segment er med i en bunke, nemlig bunken for det i, hvor segmentet ender).

Køretid? Der er n iterationer, som hver tager $\Theta(1)$ tid. Det giver alt i alt $\Theta(n)$.

Divide-and-Conquer algoritmer

Divide-and-Conquer algoritmer

Det samme som rekursive algoritmer.

En generel algoritme-udviklingsmetode:

1. Opdel problem i mindre delproblemer (af samme type).
2. Løs delproblemerne ved rekursion (dvs. kald algoritmen selv, men med de mindre input).
3. Konstruer en løsning til problemet ud fra løsningen af delproblemerne.

Basistilfælde: Problemer af mindste størrelse løses direkte (uden rekursion).

Divide-and-Conquer

Generel struktur af koden:

If basistilfælde
    Lokalt arbejde (løs problem af basisstørrelse)
Else
    Lokalt arbejde (f.eks. byg et eller flere delproblemer)
    Rekursivt kald
    Lokalt arbejde (f.eks. udnyt svar til at bygge næste delproblem)
    Rekursivt kald
    Lokalt arbejde (løs hovedproblem ud fra svar på delproblemer)

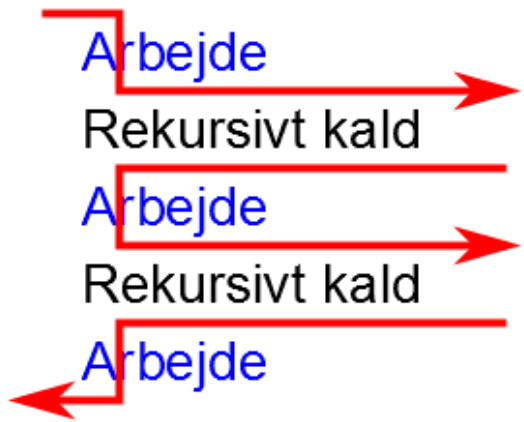
(Der behøver ikke altid være to rekursive kald. Nogle rekursive algoritmer har bare ét, og nogle har flere end to).

Divide-and-Conquer, flow of control

Flow of control (lokalt set, for ét kald af algoritmen):

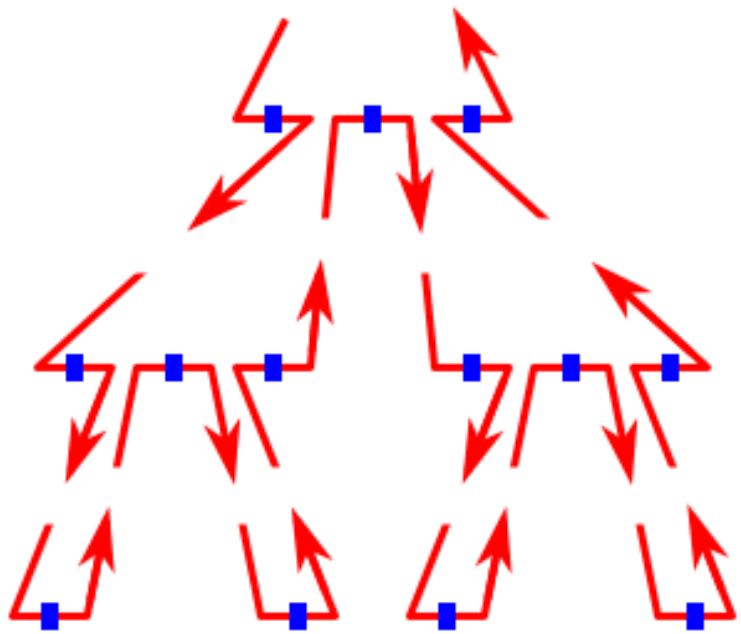
Basistilfælde

Ikke basistilfælde



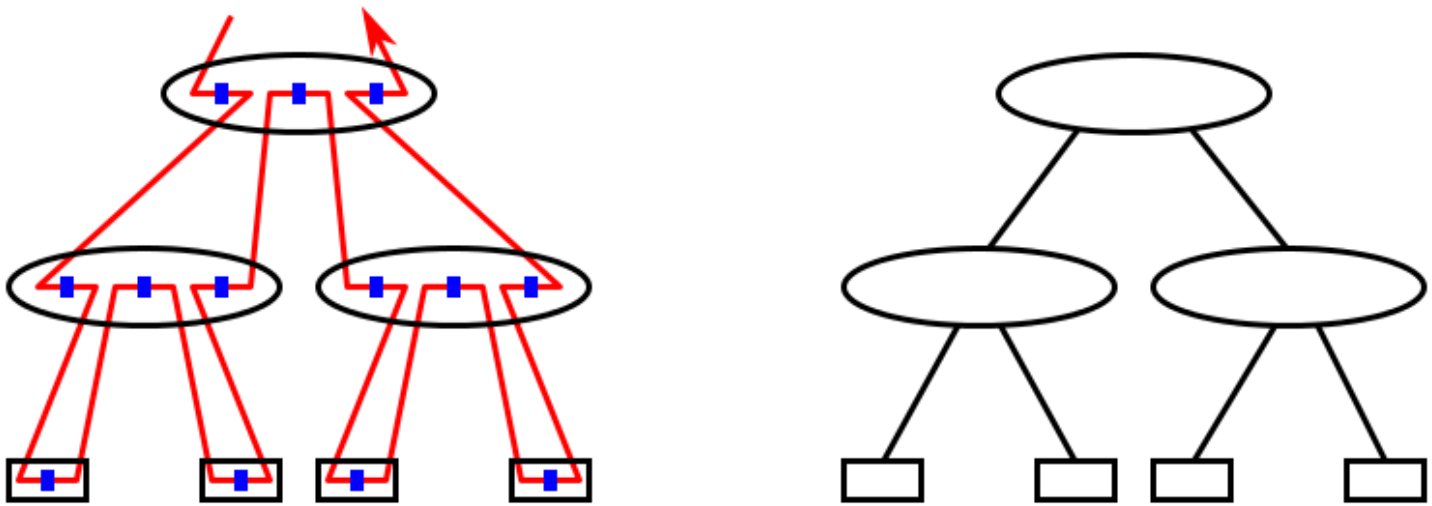
Divide-and-Conquer, udført arbejde

Globalt flow of control:



Rekursionstræer

Globalt flow of control = rekursionstræer:



Én knude = ét kald af algoritmen.

Husk: alle kald på en sti fra roden mod aktive kald er "i gang", men sat på pause. Deres lokale variable og anden state opbevares (af operativsystemet) på en stak, så kaldenes udførsel ikke blandes sammen.

- Kald af barn i rekursionstræet = push på stak.
- Afslutning af et barns udførsel = pop fra stak.

Sortering i lineær tid?

Nedre grænse for sammenligningsbaseret sortering

Nedre grænse for *alle* sorteringsalgoritmer. Kræver en præcis definition af sorteringsalgoritme.

Sammenligningbaseret: elementer kan sammenlignes med andre elementer, men ikke deltage i andre operationer.

- Basal handling: sammenlign to elementer i input og foretag derudfra et valg mellem to måder at fortsætte på.
- Svar: den opstilling som skal laves for at få sorteret orden.
- ID for elementer: deres *oprindelige* position (index) i input.

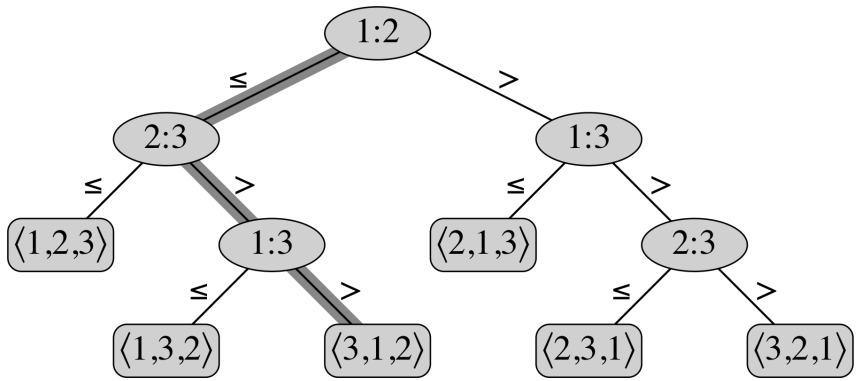
Bemærk: hvis vi starter med at annotere alle input-elementer med deres oprindelige position, kan vi i en konkret algoritme altid følge med i, hvilke to ID'er, som sammenlignes.

Annotering af input:

$$\text{F, A, C, B, E, D} \quad \rightarrow \quad \text{(F,1), (A,2), (C,3), (B,4), (E,5), (D,6)}$$

Decision trees

Præcis model som definerer begrebet "sammenligningsbaserede sorteringsalgoritmer":



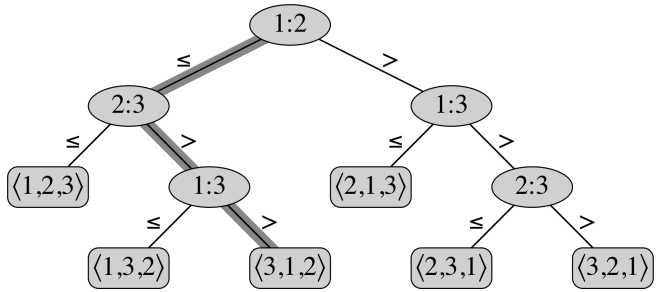
Labels for indre knuder: ID'er (dvs. oprindelige indeks i input) for to input-elementer, som sammenlignes.

Labels for blade (svar når algoritmen stopper): hvilken opstilling som skal laves for at få sorteret orden (angivet med liste af ID'er, dvs. af oprindelige indekser for input-elementer).

Worst-case køretid: længste rod-blad sti = træets højde.

Bemærk: Insertionsort, selectionsort, mergesort, quicksort, heapsort kan alle beskrives sådan.

Nedre grænse for sammenligningsbaseret sortering



For en fast samling af n elementer er der $n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot \ldots \cdot n$ forskellige input (rækkefølger af elementer).

Hvis algoritmen (træet) skal kunne sortere alle disse, skal der være mindst $n!$ blade - ellers vil der være to forskellige input som leder til samme svar, og for det ene input må svaret være forkert.

Et træ af højde h har højst 2^h blade (da det fulde træ af højde h har det).

$$2^h \geq \text{ antal blade } \geq n!$$

$$h \geq \log(n!) = \log(1 \cdot 2 \cdot 3 \ldots \cdot n)$$

$$= \log(1) + \log(2) + \ldots \log(n/2) \cdots + \log(n) \geq \frac{n}{2} \cdot \log(\frac{n}{2}) = \frac{n}{2}(\log(n) - 1)$$

$$\text{Så worst-case køretid } = \text{ træets højde } h = \Omega(n \log n)$$

Counting sort

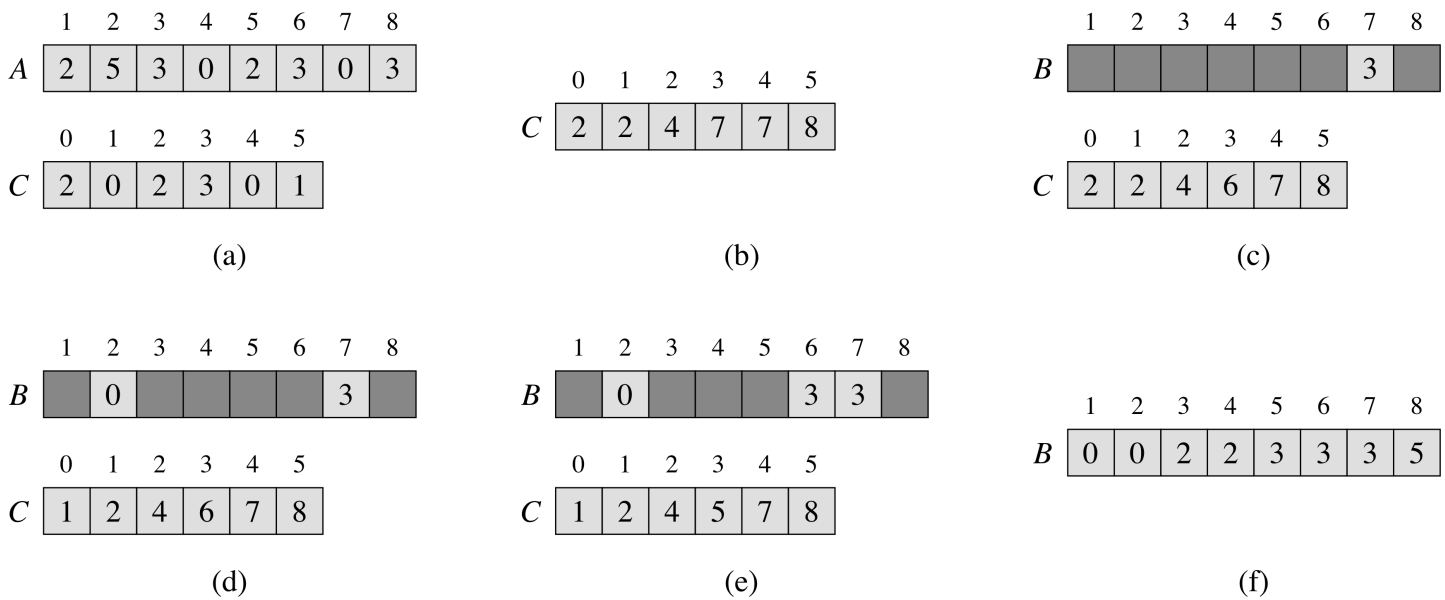
Antager at nøgler er heltal, af størrelse op til k. Derved kan elementer bruges som array-indekser ($\neq$ at bruge sammenligninger på elementer).

Counting sort: Sorterer n heltal af størrelse mellem 0 og k (inkl.).

Input-array A (længde n)

Output-array B (længde n)

Array af tællere for hver mulig elementværdi: C (længde $k + 1$)



Counting sort

```
COUNTING-SORT(A, n, k)
 1  let B[1 : n] and C[0 : k] be new arrays
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to n
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  // Copy A to B, starting from the end of A.
11  for j = n downto 1
12      B[C[A[j]]] = A[j]
13      C[A[j]] = C[A[j]] − 1    // to handle duplicate values
14  return B
```

Tid: $O(n + k)$

Bemærk: stabil, dvs. elementer med ens værdier beholder deres indbyrdes plads (da sidste løkke løber baglæns gennem A (og B for hver værdi)).

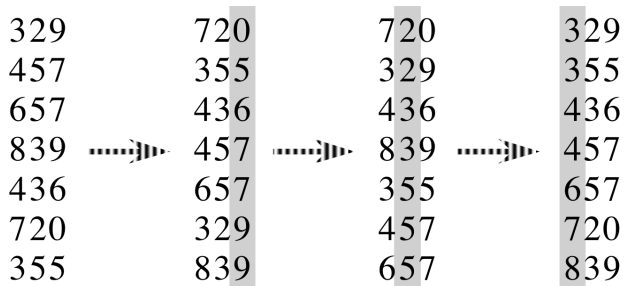
Radix sort

Radix sort: Sorterer n heltal alle med d cifre i base (radix) k.

(dvs. cifrene er heltal i $\{0, 1, 2, \ldots, k-1\}$)

På figuren nedenfor er der 7 heltal med 3 cifre i base 10.

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



Tid: $O(d(n+k))$ hvis der bruges Counting Sort i **for**-løkken.

Korrekthed:

Efter i'te iteration af **for**-løkken er A sorteret hvis man kun kigger på de i cifre mest til højre.

Radix sort

Eksempel: heltal i 10-talsystemet med bredde 12

486 239 123 989

Countingsort sorterer disse i tid $O(n + 10^{12})$

Dette er $O(n)$ hvis $n \geq 10^{12} = 1.000.000.000.000$

Se som 2-cifrede tal i base 10^6 (bemærk: sorteret orden er den samme)

486 239	123 989

Radixsort sorterer disse i tid $O(2(n + 10^6))$

Dette er $O(n)$ hvis $n \geq 10^6 = 1.000.000$

Se som 4-cifrede tal i base 10^3 (bemærk: sorteret orden er den samme)

486	239	123	989

Radixsort sorterer disse i tid $O(4(n + 10^3))$

Dette er $O(n)$ hvis $n \geq 10^3 = 1.000$

Radix sort

Eksempel: heltal i 2-talsystemet med bredde 32

11011001 10011000 01101000 10110101

Countingsort sorterer disse i tid $O(n + 2^{32})$

Dette er $O(n)$ hvis $n \geq 2^{32} = 4.294.967.296$

Se som 2-cifrede tal i base 2^{16} (bemærk: sorteret orden er den samme)

11011001 10011000	01101000 10110101

Radixsort sorterer disse i tid $O(2(n + 2^{16}))$

Dette er $O(n)$ hvis $n \geq 2^{16} = 65.536$

Se som 4-cifrede tal i base 2^8 (bemærk: sorteret orden er den samme)

11011001	10011000	01101000	10110101

Radixsort sorterer disse i tid $O(4(n + 2^8))$

Dette er $O(n)$ hvis $n \geq 2^8 = 256$

Repræsentation af tal

DM573

Rolf Fagerberg

Mål

Målet for disse slides er at beskrive, hvordan tal repræsenteres som bitmønstre i computere.

Bitmønstre

Information = valg mellem forskellig muligheder.

Simpleste situation: valg mellem to muligheder. Kald dem 0 og 1. Denne valgmulighed kaldes en bit.

Relevante for computere fordi to-delte valg er nemmest at repræsentere rent fysisk (1 = strøm, 0 = ikke strøm).

Større samling information: brug flere bits:

$$\boxed{01101011}000110010101011011\ldots$$

F.eks. 8 bits (= 1 byte): valg mellem $2^8 = 256$ muligheder.

Bitmønstre

Bitmønstre skal *fortolkes* for at have en betydning.

$$01101011 = ?$$

Der er brug for et system, som angiver, hvilken mening de forskellige bitmønstre skal tillægges.

Der er lavet sådanne systemer for f.eks.:

- Tal (heltal, kommatal)
- Bogstaver
- Pixels (billedfil)
- Amplitude (lydfil)
- Computerinstruktion (program)
- $\vdots$

Fokus i dag: systemer for heltal og kommatal.

Talsystemer

Tital-systemet:

$$\begin{aligned} 4532 &= 4 \cdot 1000 + 5 \cdot 100 + 3 \cdot 10 + 2 \cdot 1 \\ &= 4 \cdot 10^3 + 5 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0 \end{aligned}$$

Grundtal: 10
Cifre: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (fordi $10 \cdot 10^i = 10^{i+1}$)

Syvtal-systemet:

$$\begin{aligned} 4532_7 &= 4 \cdot 7^3 + 5 \cdot 7^2 + 3 \cdot 7^1 + 2 \cdot 7^0 \\ &= 4 \cdot 343 + 5 \cdot 49 + 3 \cdot 7 + 2 \cdot 1 \\ &= 1640 \end{aligned}$$

Grundtal: 7
Cifre: 0, 1, 2, 3, 4, 5, 6 (fordi $7 \cdot 7^i = 7^{i+1}$)

Total-systemet

$$
\begin{aligned}
1011_2 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 \\
&= 11
\end{aligned}
$$

Grundtal: 2
Cifre: 0, 1 (fordi $2 \cdot 2^i = 2^{i+1}$)

Relevante for computere fordi to-delte valg er nemmest at repræsentere rent fysisk (1 = strøm, 0 = ikke strøm).

Total-systemet kaldes også det *binære talsystem*.

Det giver en naturlig fortolkning af bitmønstre som ikke-negative hele tal.

Hexadecimalt talsystem

Også brugt i datalogi er 16-tal-systemet:

$$\begin{aligned} 4A3F_{16} &= 4 \cdot 16^3 + 10 \cdot 16^2 + 3 \cdot 16^1 + 15 \cdot 16^0 \\ &= 4 \cdot 4096 + 10 \cdot 256 + 3 \cdot 16 + 15 \cdot 1 \\ &= 19007 \end{aligned}$$

Grundtal: 16

Cifre: 0, 1, 2, 3, . . . , 9, A (=10), B (=11), . . . , F (=15)

(fordi $16 \cdot 16^i = 16^{i+1}$)

Hexadecimal notation

16-tals systemet kan også bruges som en simpel/kort måde at beskrive bitstrenge. Gruppér bits i grupper af 4 (dvs. 16 forskellige muligheder):

$$\boxed{0110}\boxed{1010}\boxed{1110}01\ldots$$

Brug de 16 cifre til at beskrive disse muligheder:

0111	7	1111	F
0110	6	1110	E
0101	5	1101	D
0100	4	1100	C
0011	3	1011	B
0010	2	1010	A
0001	1	1001	9
0000	0	1000	8

$$\boxed{0110}\boxed{1010}\boxed{1110}01\ldots = \texttt{6AE}\ldots$$

Addition

Addition fungerer ens i alle talsystemer, blot med grundtal udskiftet.

Tital-systemet:

$$\begin{array}{r} {\scriptstyle 1\,1\,1\,1\,} \\ 5432 \\ +96781 \\ \hline = 102213 \end{array}$$

Total-systemet:

$$\begin{array}{r} {\scriptstyle 1\,1\,1\,} \\ 1110_2 \\ +11100_2 \\ \hline = 101010_2 \end{array}$$

Subtraktion, multiplikation, division fungerer også ens. F.eks.

$$1010_2 \cdot 1110_2 = 10001100_2 \qquad (\text{Check: } 10 \cdot 14 = 140)$$
$$1101011_2 : 101_2 = 10101_2, \text{ rest } 10_2 \qquad (\text{Check: } 107 : 5 = 21, \text{ rest } 2)$$

Konvertering mellem talsystemer

Fra andre grundtal: brug definitionen af talsystemer.

$$
\begin{aligned}
1011_2 &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 \\
&= 11
\end{aligned}
$$

$$
\begin{aligned}
4532_7 &= 4 \cdot 7^3 + 5 \cdot 7^2 + 3 \cdot 7^1 + 2 \cdot 7^0 \\
&= 4 \cdot 343 + 5 \cdot 49 + 3 \cdot 7 + 2 \cdot 1 \\
&= 1640
\end{aligned}
$$

Til andre grundtal: brug gentagen heltalsdivision. Husk hvordan heltalsdivision fungerer:

Heltalsdivision	Kvotient	Rest	Som ligning
$31:7$	4	3	$31 = 7 \cdot 4 + 3$
$25:2$	12	1	$25 = 2 \cdot 12 + 1$

Detaljer for grundtal to: næste side.

Konvertering til binært talsystem

Følgende algoritme finder cifrene *fra højre til venstre* i den binære representation af et positivt heltal N:

$X = N$
Sålænge $X > 0$ gentag:
  Næste ciffer = rest ved heltalsdivision $X : 2$
  X = kvotient ved heltalsdivision $X : 2$

Eksempel: $N = 25$:

Heltalsdivision	Kvotient	Rest
$25 : 2$	12	1
$12 : 2$	6	0
$6 : 2$	3	0
$3 : 2$	1	1
$1 : 2$	0	1

$25 = 11001_2$

Hvorfor virker det?

Heltalsdivision	Kvotient	Rest
$25:2$	12	1
$12:2$	6	0
$6:2$	3	0
$3:2$	1	1
$1:2$	0	1

$$25 = 11001_2$$

$$\begin{aligned}
25 &= 2 \cdot 12 + 1 \\
&= 2(2 \cdot 6 + 0) + 1 \\
&= 2(2(2 \cdot 3 + 0) + 0) + 1 \\
&= 2(2(2(2 \cdot 1 + 1) + 0) + 0) + 1 \\
&= 2(2(2(2(2 \cdot 0 + 1) + 1) + 0) + 0) + 1 \\
&= 2^5 \cdot 0 + 2^4 \cdot 1 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 0 + 2^0 \cdot 1
\end{aligned}$$

Bemærk at sidste division altid er $1:2$ (med kvotient 0 og rest 1). Fordi X bliver 1 på et tidspunkt, da man ved en heltalsdivision med 2 hele tiden gør X mindre, men ikke kan komme fra heltal ≥ 2 til heltal ≤ 0.

Repræsentation af heltal

Talrepræsentationer bruger (næsten altid) et fast antal bits (så operationer kan implementeres effektivt).

$$\boxed{k \text{ bits} = 2^k \text{ forskellige bitmønstre}}$$

Positive heltal: det binære talsystem giver en naturlig repræsentation.

$k = 4$:

0111	7	1111	15
0110	6	1110	14
0101	5	1101	13
0100	4	1100	12
0011	3	1011	11
0010	2	1010	10
0001	1	1001	9
0000	0	1000	8

Hvordan skal disse 2^k bitmønstre fordeles, hvis vi vil repræsentere alle heltal, både negative og positive?

Two's complement

En mulig repræsentation af både negative og positive heltal er følgende:

$k = 4$:

0111	7	1111	-1
0110	6	1110	-2
0101	5	1101	-3
0100	4	1100	-4
0011	3	1011	-5
0010	2	1010	-6
0001	1	1001	-7
0000	0	1000	-8

Dette kaldes "two's complement" (af grunde, som ikke er relevante her).

Det kan også beskrives som at højeste ciffer tæller $-(2^{k-1})$ i stedet for 2^{k-1}:

$$\begin{aligned} 1101_2 &= 1 \cdot (-(2^3)) + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 1 \cdot (-8) + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 \\ &= -3 \end{aligned}$$

Two's complement

Repræsentationen two's complement har mange gode egenskaber:

1. Fortegn kan ses af første bit.
2. Simpel metode til at skifte fortegn findes:

   Kopier bits fra højre til venstre, til og med første 1-bit.
   Resten af bits inverteres (dvs. 1 sættes til 0 og omvendt).

   (Eksempel: $6 = 0110 \rightarrow 1010 =$ -6)

3. Den almindelige metode til addition virker også for negative tal. Ingen ekstra logiske kredsløb for disse (sparer transistorer på CPU).
4. Subtraktion kan laves ved at vende fortegn og addere. Ingen logiske kredsløb for subtraktion (sparer transistorer på CPU).

[Her er 1) og 4) er klare, mens 2) og 3) kræver bevis (ikke pensum).]

Two's complement vælges derfor ofte som repræsentation for heltal. I Java er typen `int` heltal i two's complement ($k = 32$). I Python er dette også grundtypen for heltal.

Repræsentationer af kommatal

Talrepræsentationer bruger (næsten altid) et fast antal bits.

$$\boxed{k \text{ bits} = 2^k \text{ forskellige bitmønstre}}$$

Hvordan bruge k bits til at beskrive kommatal?

Fra tital-systemet kendes

- Fast decimalpunkt (45.32)
- Flydende decimalpunkt ($-6.87 \cdot 10^{-6}$)

Disse kan nemt gentages i total-systemet (grundtal 2). Se næste sider.

I computere bruges oftest flydende decimalpunkt (med grundtal 2). For at forstå disse skal man forstå fast decimalpunkt (med grundtal 2) først.

I Java er typerne `float` ($k = 32$) og `double` ($k = 64$) kommatal i flydende decimalpunkt. I Python er typen `float` det samme ($k = 64$).

Fast decimalpunkt

Tital-systemet:

$$\begin{aligned} 45.32 &= 4 \cdot 10 + 5 \cdot 1 + 3 \cdot 1/10 + 2 \cdot 1/100 \\ &= 4 \cdot 10^1 + 5 \cdot 10^0 + 3 \cdot 10^{-1} + 2 \cdot 10^{-2} \end{aligned}$$

Det binære talsystem:

$$\begin{aligned} 10110.111_2 &= 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 \\ &\quad + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 \\ &\quad + 0 \cdot 1 + 1 \cdot 1/2 + 1 \cdot 1/4 + 1 \cdot 1/8 \\ &= 22\tfrac{7}{8} \\ &= 22.875 \end{aligned}$$

Flydende decimalpunkt

Tital-systemet: få kommaet til at stå efter første ciffer $\neq 0$.

$2340000.0 = 2.34 \cdot 10^6$ $\quad 0.000456 = 4.56 \cdot 10^{-4}$ $\quad -0.0987 = -9.87 \cdot 10^{-2}$

Fortegn:	plus	Fortegn:	plus	Fortegn:	minus
Eksponent:	6	Eksponent:	-4	Eksponent:	-2
Mantisse:	2.34	Mantisse:	4.56	Mantisse:	9.87

Total-systemet: få kommaet til at stå efter første ciffer $\neq 0$ (er altid 1).

$101100.0_2 = 1.011_2 \cdot 2^5$ $\quad -0.01101_2 = -1.101_2 \cdot 2^{-2}$

Der afsættes et fast antal bits til hver af: fortegn, eksponent, mantisse. For $k = 8$ vælger vi: 1, 3 og 4 bits. Eksponent kan være positiv eller negativ, vi bruger two's complement til den. Mantisse fyldes om nødvendigt op med 0'er til højre. Eksempel: for -0.01101_2 fås

Fortegn:	1	(1 for negativt tal, 0 for positivt)
Eksponent:	110	(-2 i two's complement (3 bits))
Mantisse bits:	(1.)1010	(første bit skrives ikke, da den altid er 1)

Så -0.01101_2 repræsenteres som $\boxed{1}\boxed{110}\boxed{1010}$.

Begrænsninger

Heltal og kommatal er uendelige talmængder. Hvis der afsættes et fast antal (k) bits fås et endeligt antal (2^k) forskellige bitmønstre.

Ikke alle tal kan repræsenteres!

Viser sig f.eks. ved

- Overflow
  - maxInt + maxInt = ?
- Rounding errors
  - Stort tal x + meget lille tal y = samme store tal x
  - $(x + y) + z \neq x + (y + z)$ hvis f.eks. $x + y$ ikke kan repræsenteres eksakt.

I praksis opleves sjældent problemer pga. et stort antal bits i talrepræsentationerne.

Alternativt findes programmeringsbiblioteker, der implementerer f.eks. vilkårligt store heltal (under brug af variabelt antal bits, samt tab af effektivitet). Dette sker automatisk i Python for typen `int`.

Prioritetskøer

Prioritetskøer?

En prioritetskø er en datastruktur.

Datastrukturer

$$\boxed{\text{Datastruktur} = \text{data} + \text{operationer herpå}}$$

Data:

- Normalt struktureret som en ID plus yderligere data. ID kaldes også en nøgle (key).
- Vi nævner normalt ikke den yderligere data. Dvs. elementer omtales blot som ID, men er reelt (ID,data) eller (ID,reference til data).
- ID er ofte fra et ordnet univers (har en ordning), f.eks. `int`, `float`, `String`.

Operationer:

- Datastrukturens egenskaber udgøres af de tilbudte operationer, samt deres køretider.
- Målene er fleksibilitet og effektivitet (som regel modstridende mål).

Datastrukturer

Tænk på en datastruktur som et API for adgang til en samling data.

- Datastrukturer niveau 1: de tilbudte operationer (i Java: et interface).
- Datastrukturer niveau 2: en konkret implementation af de tilbudte operationer (i Java: en klasse som implementerer interfacet).

En givent sæt operationer (niveau 1) kan have mange forskellig implementationer (niveau 2), ofte med forskellige køretider.

I dette kursus: katalog af datastrukturer (niveau 1) med bred anvendelse samt effektive implementationer (niveau 2) heraf.

Prioritetskøer

Prioritetskøer, niveau 1:

Data:

- Element = nøgle (ID) fra et ordnet univers samt evt. yderligere data.

Centrale operationer (max-version af prioritetskø):

- Q.Extract-Max(): Returnerer elementet med den største nøgle i prioritetskøen Q (et vilkårligt sådant element, hvis der er flere lige store). Elementet fjernes fra Q.
- Q.Insert(e: element). Tilføjer elementet e til prioritetskøen Q.

Bemærk: vi kan sortere med disse operationer:

$n \times$ Insert
$n \times$ Extract-Max

Prioritetskøer

Prioritetskøer, niveau 1:

Ekstra operationer:

- Q.Increase-Key(r: reference til et element i Q, k nøgle). Ændrer nøglen til $\max\{k, \text{gamle nøgle}\}$ for elementet refereret til af r.
- Q.Build(L: liste af elementer). Bygger en prioritetskø indeholdende elementerne i listen L.

Trivielle operationer for alle datastrukturer:

- Q.CreateNewEmpty(), Q.DestructEmpty(), Q.IsEmpty?().

Vil ikke blive nævnt fremover.

Implementation via heaps

En mulig implementation (niveau 2): brug heapstrukturen fra Heapsort.

Vi har allerede:

- Extract-Max: Er essentielt hvad der bruges under anden del af Heapsort – fjern rod, flyt sidste blad op som rod, kald Heapify. Køretid: $O(\log n)$.
- Build: Brug Heapify gentagne gange bottom-up. Køretid: $O(n)$.

Mangler:

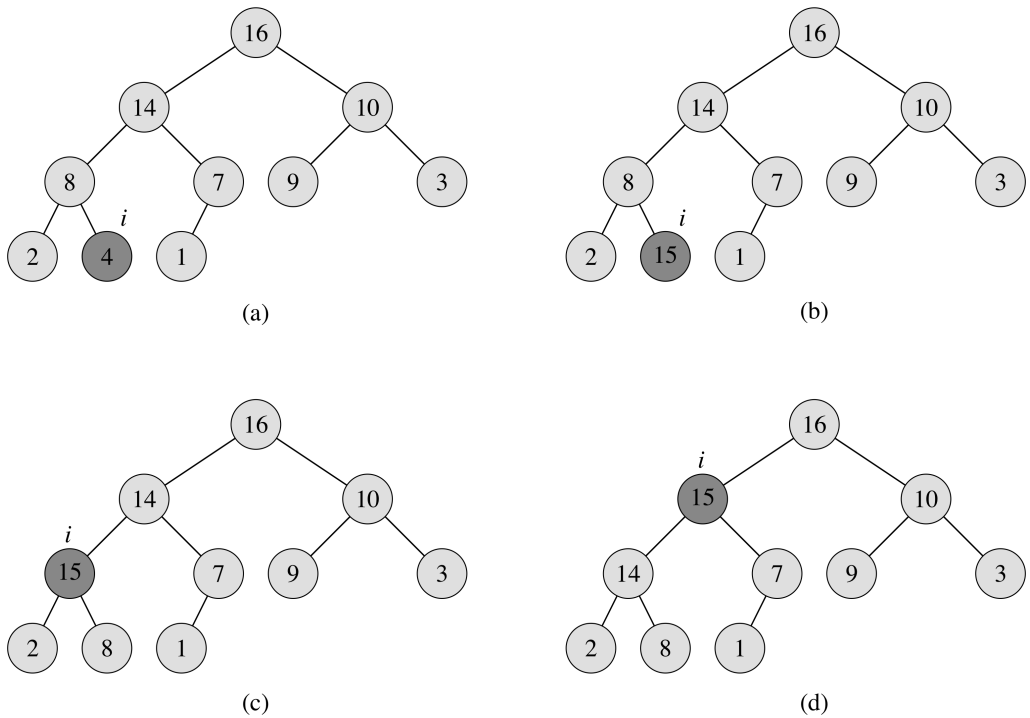
- Insert
- Increase-Key

[Detalje: I Java kræver array-versionen af heaps et kendt maximum for størrelsen af køen. Alternativt kan array erstattes af et extendible array, f.eks. `java.util.ArrayList` i Java eller lister i Python. Man kan også implementere heaptræet via pointere/referencer, lige som vi gør med søgetræer lidt senere.]

Increase-Key

1. Ændre nøgle for element.
2. Genopret heaporden: så længe elementet er stærkere end forælder, skift plads med denne.

Eksempel med INCREASE-KEY(i, 15):



Køretid: Højden af træet, dvs. $O(\log n)$.

Insert

1. Indsæt det nye element sidst ($\Rightarrow$ heapfacon i orden).
2. Genopret heaporden præcis som i Increase-Key: så længe elementet er større end forælder, skift plads med denne.

Køretid: Højden af træet, dvs. $O(\log n)$.

Forskellige implementationer af prioritetskøer

Samme niveau 1, forskellige niveau 2:

	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	$O(\log n)$	$O(1)$	$O(n)$

Ovenstående samling operationer (niveau 1) er for max-prioritetskøer.
Der er naturligvis nemt at lave min-prioritetskøer med operationerne

EXTRACT-MIN, BUILD, DECREASE-KEY, INSERT,

blot ved at vende alle uligheder mellem nøgler i definitioner og algoritmer.

Dictionaries

Datastrukturer (recap)

$$\boxed{\text{Datastruktur} = \text{data} + \text{operationer herpå}}$$

Data:

- En ID (nøgle) + associeret data (ofte underforstået, også i disse slides).

Operationer:

- Datastrukturens egenskaber udgøres af de tilbudte operationer (API for adgang til data), samt deres køretider (forskellige implementationer af samme API kan give forskellige køretider).

I dette kursus: katalog af datastrukturer (niveau 1) med bred anvendelse samt effektive implementationer (niveau 2) heraf.

Datastrukturer (recap)

Vi har allerede set Priority Queue. Datastruktur som understøtter (niveau 1) operationerne:

- Extract-Min(): Fjern et element med mindste nøgle fra prioritetskøen og returner det.
- Insert(key): Tilføj nyt element til prioritetskøen.

- Build(liste af elementer): Byg en prioritetskø indeholdende elementerne.
- Decrease-Key(key, reference til element i kø): Sætter nøglen for elementet til min{key,gamle key}.

Dictionaries

Emne for disse slides: Dictionary. Datastruktur som understøtter (niveau 1) operationerne:

- Search(key): returner element med nøglen key (eller fortæl hvis det ikke findes).
- Insert(key): Indsæt nyt element med nøglen key.
- Delete(key): Fjern element med nøglen key.

- Predecessor(key): Find elementet med højeste nøgle $<$ key.
- Successor(key): Find elementet med laveste nøgle $>$ key.
- OrderedTraversal(): Udskriv elementer i sorteret orden.

For de sidste tre operationer kræves at nøglerne har en ordning.

Hvis kun de tre første operationer skal understøttes, kaldes det en unordered dictionary. Hvis alle seks understøttes, kaldes det en ordered dictionary.

Dictionaries

Dictionaries i Java: interface `Map`.

Dictionaries i Python: `dict`.

Implementationer (niveau 2) som vi møder i dette kursus:

- Balancerede binære søgetræer: Understøtter alle ovenstående operationer (samt mange flere, f.eks. ved at tilføje ekstra information i knuderne) i $O(\log n)$ tid.
- Hashing: understøtter de tre første operationer forventet tid $O(1)$.

Disse implementationer findes i Java som henholdsvis `TreeMap` og `HashMap`. I Python er den indbyggede datatype `dict` implementeret med hashing. Der er ingen balancerede binære søgetræer i Pythons standard moduler, men man kan finde moduler med dem fra andre kilder.

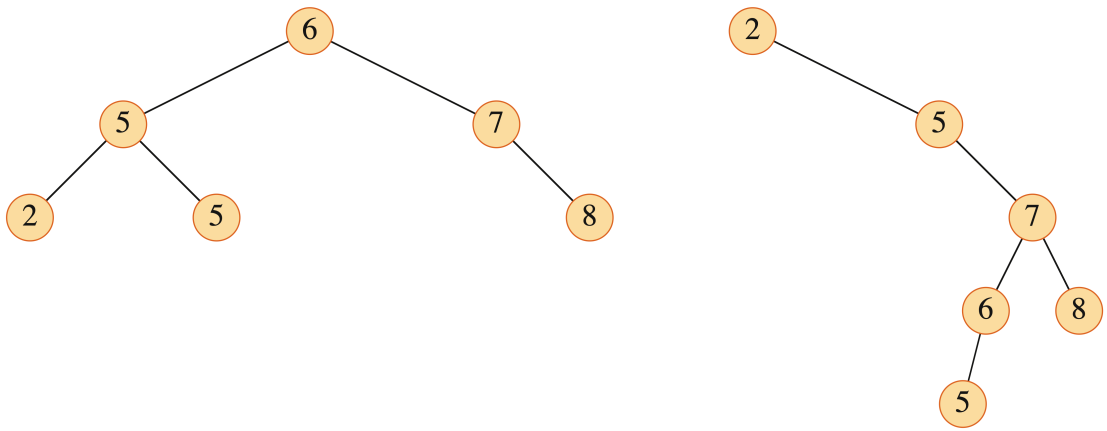
Binært søgetræ

- et binært træ
- med knuder i inorder

Et binært træ med nøgler i alle knuder overholder inorder hvis det for alle knuder v gælder:

nøgler i v's venstre undertræ $\leq$ nøgle i v $\leq$ nøgler i v's højre undertræ

Eksempler:

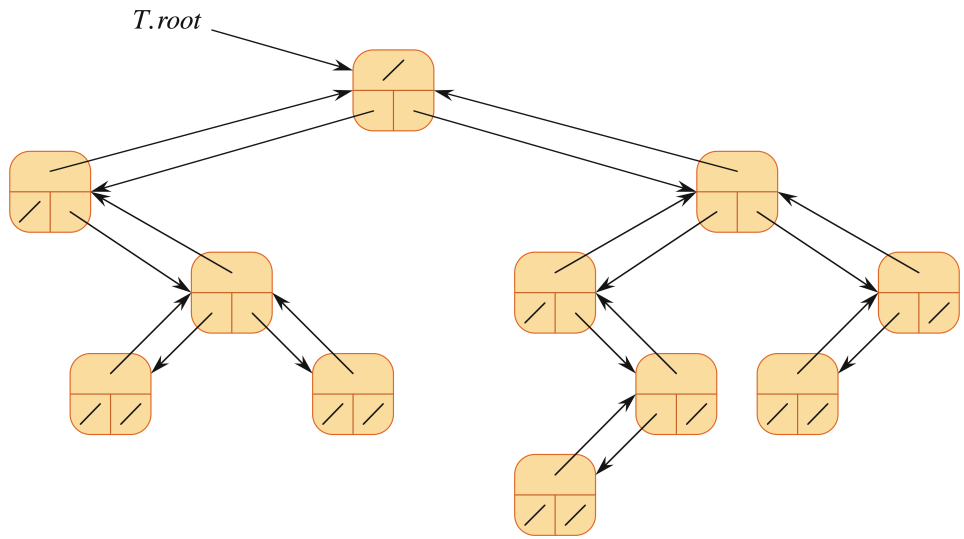


Binære søgetræer

Typisk implementation: Knude-objekter med:

- Reference til forælder
- Reference til venstre undertræ
- Reference til højre undertræ

samt ét træ-objekt med reference til roden. (Java: reference, bog: pointer).



Knude-objekter

Java-klasse for knuder:

```
class Node {
  int key;
  Node leftchild;
  Node rightchild;
  Node parent;
  .
  .
  (constructor)
  (andre metoder)
  .
}
```

Python-klasse for knuder:

```
class Node:
  def __init__(self):
    self.key = None
    self.leftchild = None
    self.rightchild = None
    self.parent = None
  .
  .
  (andre metoder)
  .
```

Python via lister:

```
Node = [key,leftchild,rightchild,parent]
```

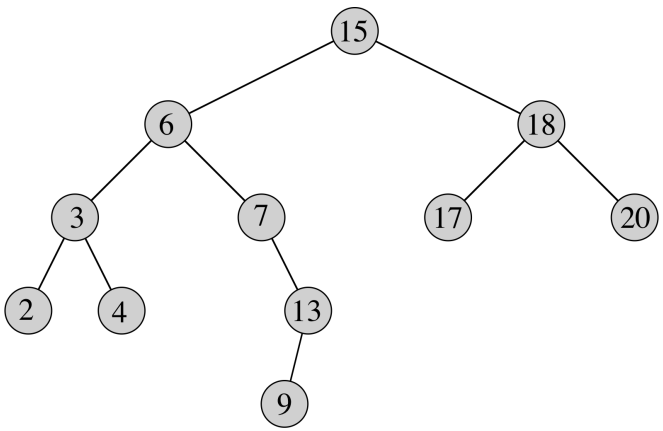
Binære søgetræer

Pga. definitionen af inorder

nøgler i v's venstre undertræ $\leq$ nøgle i v $\leq$ nøgler i v's højre undertræ

kan binære søgetræer siges at indeholde data i sorteret orden.

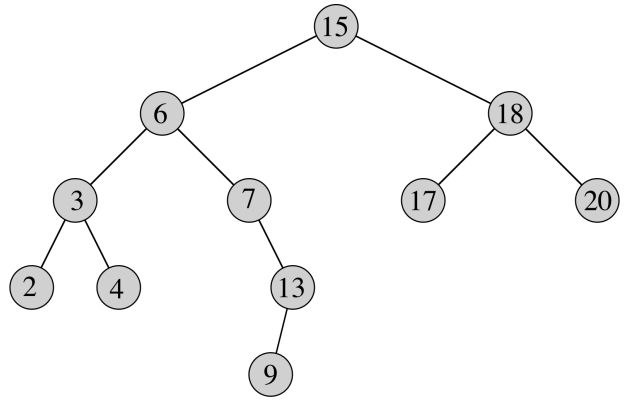
Mere præcist: inorder gennemløb vil udskrive nøgler i sorteret orden:




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

Køretid: $O(n)$ (der laves O(1) arbejde per knude i træet).

Søgning i binære søgetræer



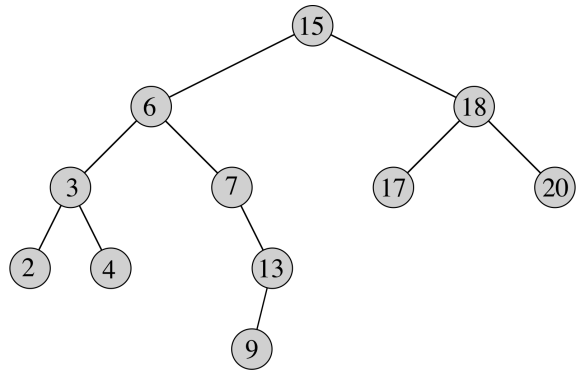
TREE-SEARCH(x, k)

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

Princip:

Hvis søgte element findes, er det i det undertræ, vi er kommet til

Flere slags søgninger i binære søgetræer



TREE-MAXIMUM(x)

while $x.right \neq$ NIL
  $x = x.right$
return x

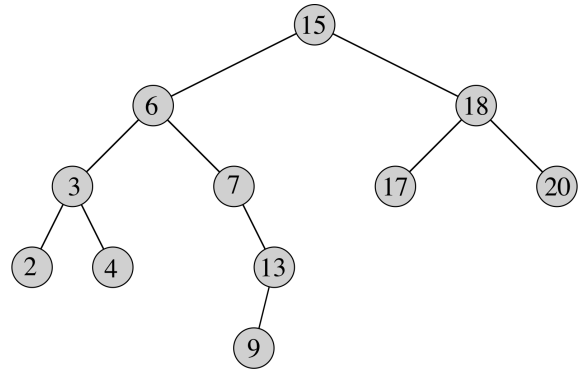
TREE-MINIMUM(x)

while $x.left \neq$ NIL
  $x = x.left$
return x

Princip:

Det søgte element findes i det undertræ, vi er kommet til

Flere slags søgninger i binære søgetræer

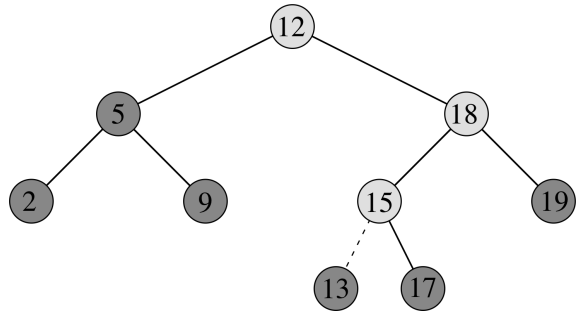



```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

Princip:

> Se på stien fra x til rod. Ingen side-træer på den kan indeholde det søgte element (pga. in-order).

Indsættelser i binære søgetræer



- Søg nedad fra rod: gå i hver knude v mødt videre ned i det undertræ (højre/venstre), hvor nye element skal være iflg. inorder-krav for v.
- Når blad (NIL/tomt undertræ) nås, erstat dette med den nye knude (med to tomme undertræer).

Inorder er overholdt for knuder på søgesti (pga. søgeregel), og for alle andre knuder (fordi de ikke har fået nogle nye efterkommere i deres to undertræer).

Sletninger i binære søgetræ

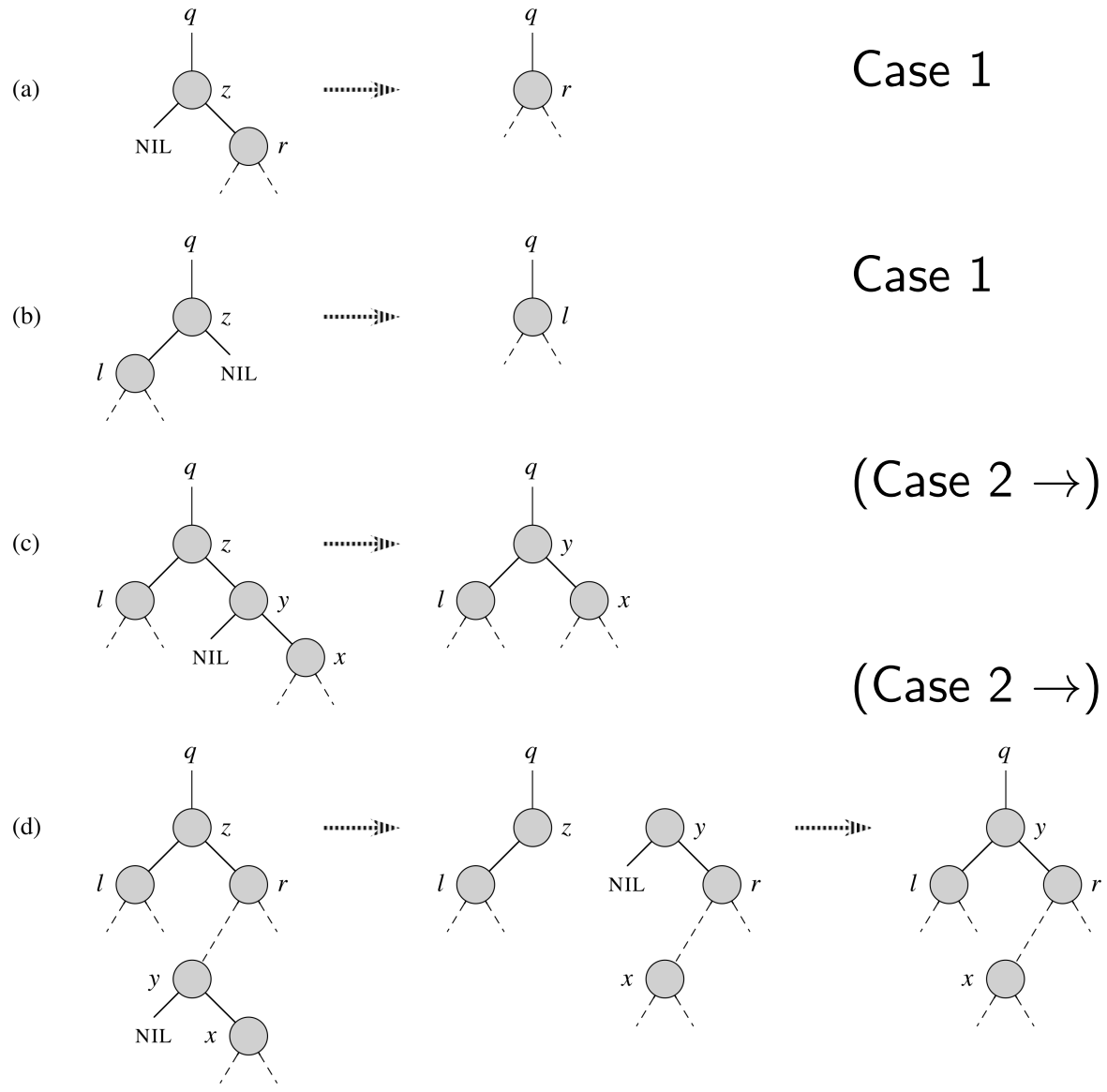
Sletning af knude z:

- Case 1: Mindst ét barn er NIL: Fjern z samt dette barn, lad andet barn tage z's plads.
- Case 2: Ingen børn er NIL: Da er successor-knuden y til z den mindste knude i z's højre undertræ. Fjern y (som er en Case 1 fjernelse, da dens venstre barn er NIL), og indsæt den på z's plads.

Begge cases efterlader træet i inorder: I Case 1 får ingen knuder nye efterkommere i deres to undertræer. I Case 2 får y (og kun y) nye efterkommere i sine to undertræer, men da y er z's successor, er der ingen nøgler i træet med værdi mellem z's og y's nøgler, så nøglerne i y's nye undertræer overholder inorder i forhold til y, eftersom de gjorde i forhold til z.

Bemærk at strukturelt i træet er alle sletninger en Case 1 sletning.

Sletninger i binære søgetræ (bogens cases)



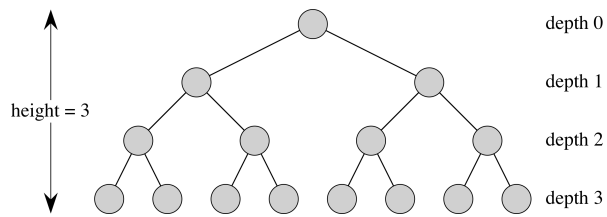
Tid for operationer i binære søgetræ

For alle operationer (undtagen inorder gennemløb):

Gennemløb sti fra rod til blad.

Dvs. køretid = O(højde).

Et træ med højde h kan ikke indeholde flere knuder end det fulde træ med højde h. Dette indeholder $2^{h+1}-1$ knuder (jvf. slides om heaps).



Så for et træ med n knuder og højde h gælder:

$$n \leq 2^{h+1} - 1 \quad \Leftrightarrow \quad \log_2(n+1) - 1 \leq h$$

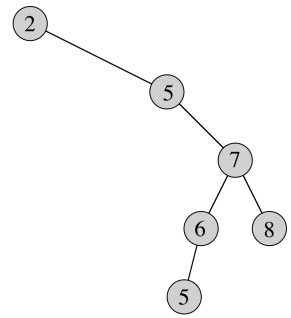
Dvs. den bedst mulige højde er $\log_2 n$ (± 1)

Kan vi holde højden tæt på optimal – f.eks. $O(\log n)$ – under updates (indsættelser og sletninger)?

Balancerede binære søgetræer

Kan vi holde højden $O(\log n)$ under updates (indsættelser og sletninger)?

Kræver rebalancering (omstrukturering af træet) efter updates, da dybe træer ellers kan opstå:



Vedligehold af $O(\log n)$ højde første gang opnået med AVL-træer [1961].

Mange senere forslag. Et forslag består af:

- Balanceinformation i knuder.
- Strukturkrav baseret på balanceinformation, som sikrer $O(\log n)$ højde.
- Algoritmer, som genopretter strukturen efter en update.

I dette kursus: rød-sorte træer.

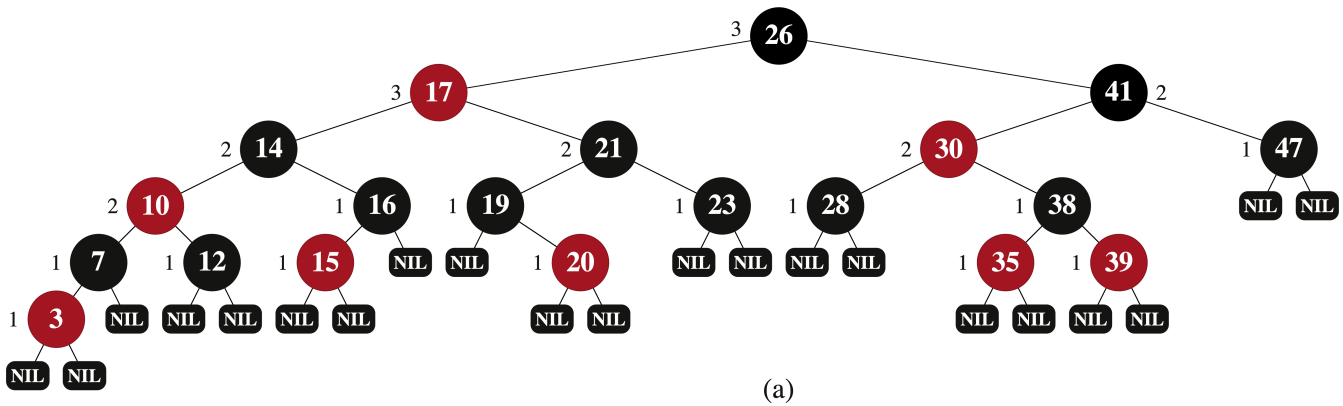
Rødsorte træer

Balanceinformation i knuder: 1 bit (kaldet rød/sort farve).

Strukturkrav:

- Rod og blade sorte.
- Samme antal sorte på alle rod-blad stier.
- Ikke to røde i træk på nogen rod-blad sti.

Eksempel:

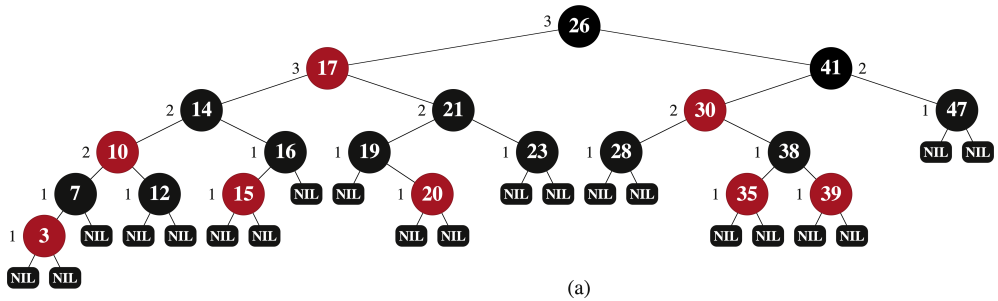


(a)

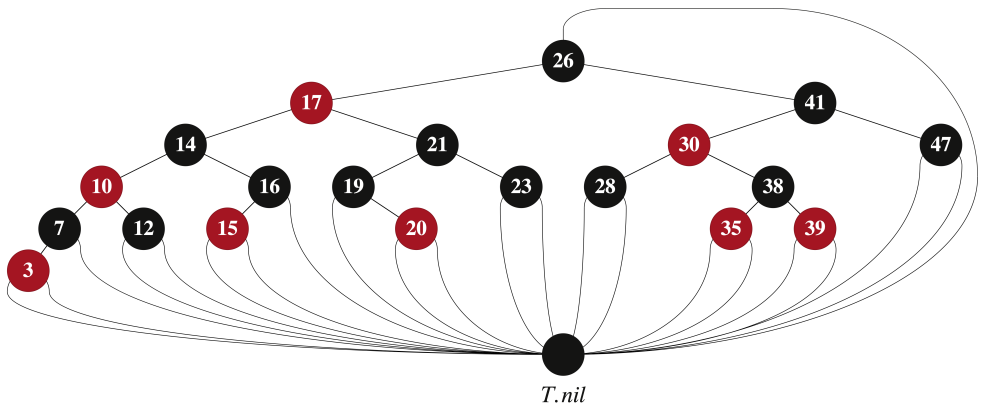
NB: begrebet blade bliver i rød-sorte træer af brugt om NIL-undertræer (hvilket teknisk set øger højden af et træ med én).

Rødsorte træer

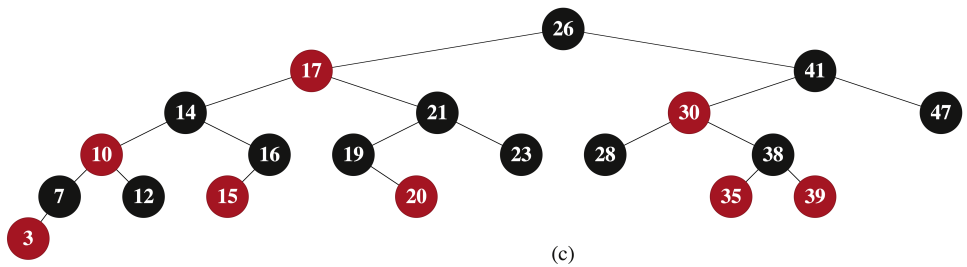
Andre repræsentationer i bogen (samme træ):



(a)



(b)



(c)

Rødsorte træer

Strukturkrav (recap):

- Rod og blade sorte.
- Samme antal sorte på alle rod-blad stier.
- Ikke to røde i træk på nogen rod-blad sti.

Sikrer disse strukturkrav sikrer $O(\log n)$ højde? Ja:

Lad antal sorte på alle rod-blad stier være k (dvs. $k-1$ sorte knuder plus et sort blad). Så indeholder alle rod-blad stier mindst $k-1$ knuder (de sorte plus evt. nogle røde). Der er derfor mindst $k-1$ fulde lag af knuder i træet.

Derfor er $n \geq 2^0 + 2^1 + 2^2 + \cdots + 2^{k-2} = 2^{k-1} - 1$.

Heraf følger $\log(n+1) \geq k-1$.

Da der ikke er to røde knuder i træk, indholder den længste rod-blad sti højst $2(k-1)$ kanter.

Så højden er højst $2(k-1) \leq 2\log(n+1) = O(\log n)$.

Indsættelse

1. Indsæt en knude i træet
2. Fjern evt. opstået ubalance (overtrædelse af rød-sort strukturkravene).

Recall indsættelse: et blad (NIL) erstattes af en knude med to blade som børn.

Ubalance?

Recall indsættelse: et blad (NIL) erstattes af en knude med to blade som børn.

Overtrædelse af rød-sorte strukturkrav?

- Rod og blade sorte.
- Samme antal sorte på alle rod-blad stier.
- Ikke to røde i træk på nogen rod-blad sti.

De to nye blade skal være sorte.

Vi vælger at gøre den nye indsatte knude rød.

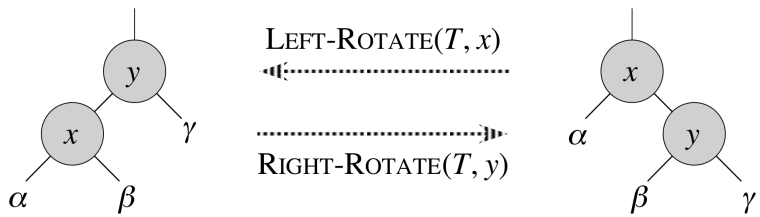
> *Mulige overtrædelse af strukturkrav er nu: To røde knuder i træk på en rod-blad sti ét sted i træet.*

Idé til plan: Kan problemet ikke løses umiddelbart, så skub det opad i træet indtil det kan (forhåbentligt nemt at gøre, hvis det når roden).

Rebalancering

Detaljeret plan: skub rød-rød problem opad i træet, under brug af omfarvninger og restruktureringer af træet.

Den basale restrukturering vil være en rotation (α, β, γ er undertræer, evt. tomme):



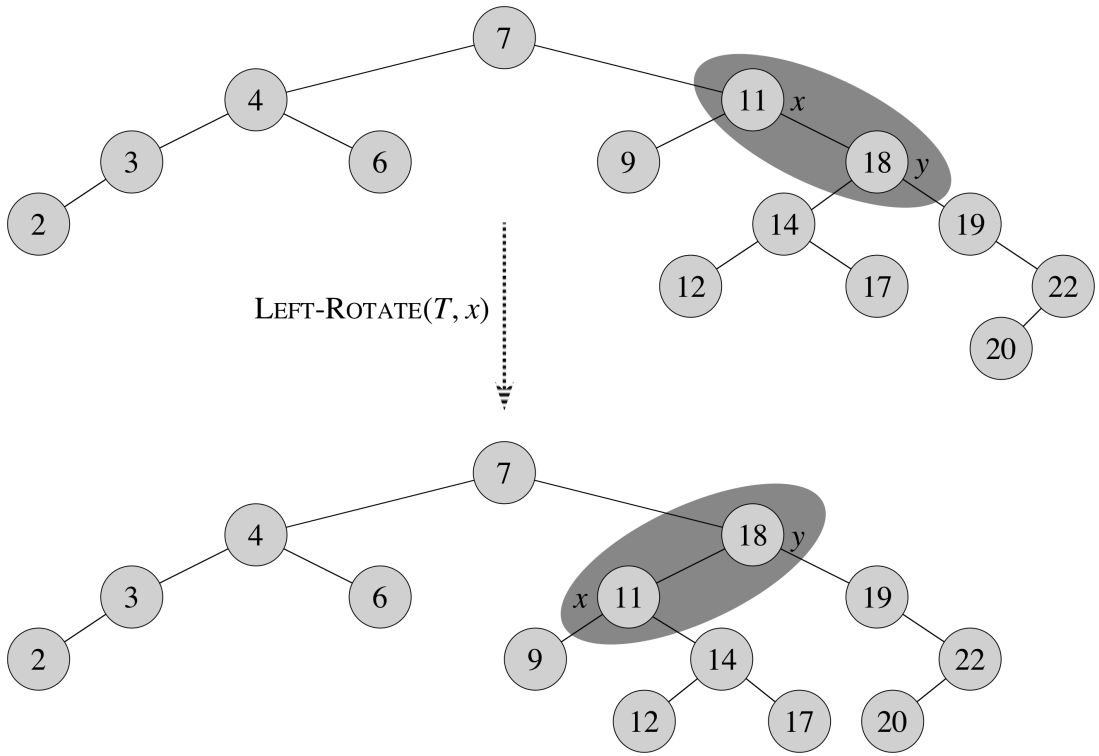
Central observation: Rotationer kan ikke ødelægge in-order i træet:

Kun x og y kan få inorder overtrådt (alle andre knuder har de samme elementer i deres undertræer før og efter). Men dette sker ikke, da følgende gælder både før og efter en rotation:

$$\text{keys i } \alpha \leq x \leq \text{keys i } \beta \leq \text{y} \leq \text{keys i } \gamma$$

Så vi skal ikke bekymre os om bevarelse af inorder, hvis vi kun restrukturerer vha. rotationer.

Eksempel på rotation



Plan for rebalancering (efter indsættelse)

Recap af plan: skub rød-rød problem opad i træet, under brug af omfarvninger og restruktureringer (rotationer) af træet.

Princip undervejs:

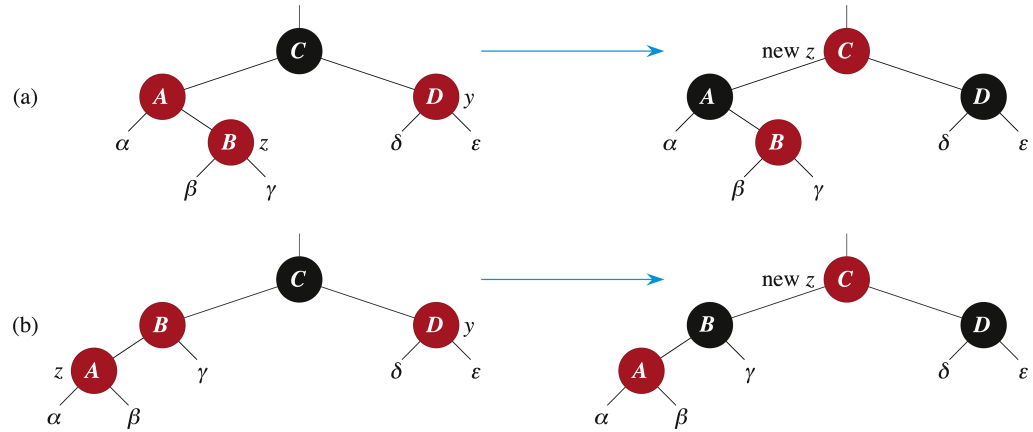
- To røde knuder i træk på en rod-blad sti højst ét sted i træet.
- Bortset herfra er de rød-sorte krav overholdt.

Mål: I $O(1)$ tid, fjern problemet eller skub det ét skridt nærmere roden.

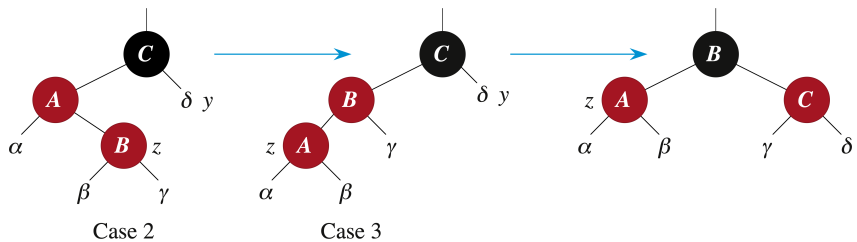
Dette vil give rebalancering i $O(\text{højde}) = O(\log n)$ tid.

Cases i rebalancering (efter indsættelse)

Case 1: Rød onkel til z (onkel = forælders søsken y).



Case 2: Sort onkel til til z (onkel = forælders søsken y).

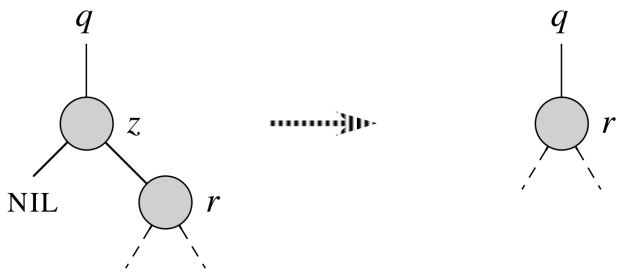


Her er z nederste knude i rød-rød problemet. Kontrollér at princip vedligeholdes. Kontrollér at problem fjernes eller flyttes nærmere roden (én færre sort knude på sti til rod). Hvis z bliver lig roden, kan den blot farves sort ($\Rightarrow$ *alle* stier får en sort mere).

Sletning

1. Slet en knude i træet
2. Fjern evt. opstået ubalance (overtrædelse af rød-sort kravene).

Recall sletning: der fjernes strukturelt set altid én knude hvis ene barn er et blad (NIL), som også fjernes.



Ubalance?

Overtrædelse af rød-sorte krav?

- Rod og blade sorte.
- Samme antal sorte på alle rod-blad stier.
- Ikke to røde i træk på nogen rod-blad sti.

Fjernet knude rød: Alle rød-sorte krav stadig overholdt.

Fjernet knude sort: Ikke længere samme antal sorte på alle stier.

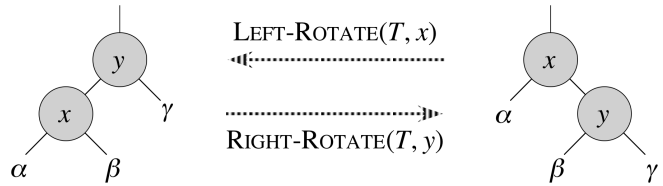
Meget brugbar formulering:

> *Lad den fjernede knudes andet barn være "sværtet" og gælde for "én mere" sort end dens farve angiver, når vi tæller sorte på stier (sværtet sort = 2 sorte, sværtet rød = 1 sort). Så er kravene overholdt, bortset fra eksistensen af en sværtet knude.*

Idé til plan: Kan problemet ikke løses umiddelbart, så skub det opad i træet til det kan (forhåbentligt nemt at gøre, hvis det når roden).

Rebalancering

Skub sværtet knude opad i træet, under brug af omfarvninger og rotationer:



Princip undervejs:

- Højst én knude i træet er sværtet.
- Hvis sværtningen tælles med, er de rød-sorte krav overholdt.

Nemme stoptilfælde:

- Sværtet knude er rød $\Rightarrow$ sværtning kan fjernes ved at farve knuden sort.
- Sværtet knude er rod $\Rightarrow$ sværtning kan bare fjernes ($\Rightarrow$ *alle* stier får en sort mindre).

(Så nok at se på tilfældet at den sværtede knude er sort.)

Rebalancering

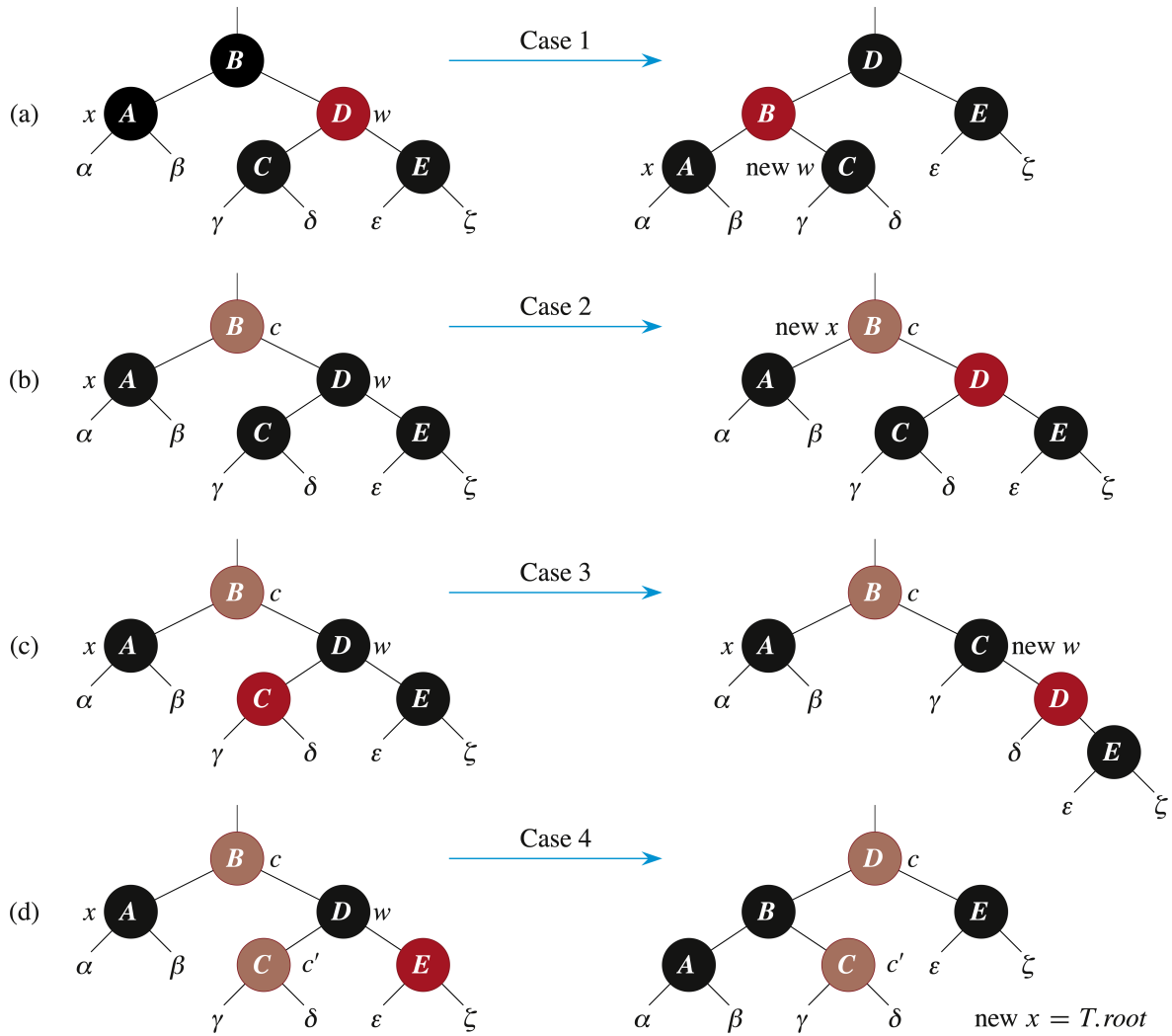
Mål: I $O(1)$ tid, fjern problemet eller skub det ét skridt nærmere roden.

Dette vil give rebalancering i $O(\text{højde}) = O(\log n)$ tid.

Cases for sværtet sort knude x med søsken w.

1. Rød søsken.
2. Sort søsken, og denne har to sorte børn.
3. Sort søsken, og dennes nærmeste barn er rødt, det fjerneste sort.
4. Sort søsken, og dennes fjerneste barn er rødt.

Cases i rebalancering (efter sletning)



Her er x sværtet knude. Kontrollér at princip vedligeholdes. Kontrollér $O(1)$ tid før sværtning fjernes eller flyttes ét skridt nærmere roden.

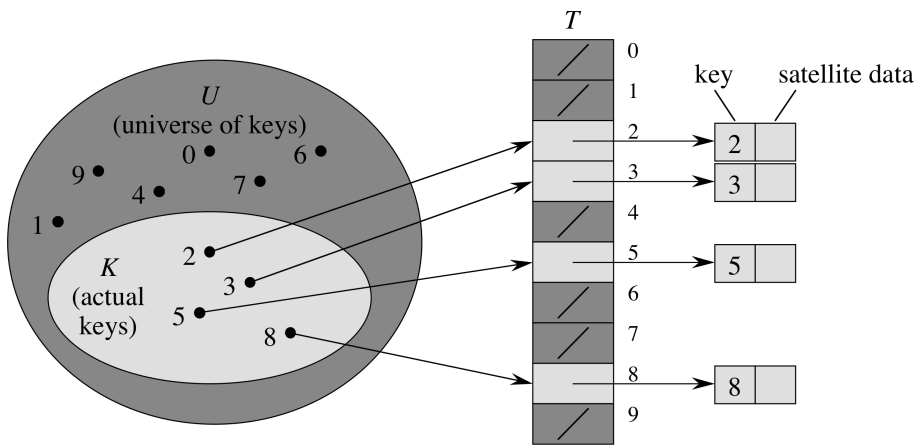
Hashing

Vi antager i hashing, at keys er heltal op til en max-grænse u. [For at bruge hashing på andre datatyper må elementer tildeles en unik heltalsværdi, jvf. `hashCode()` i Java og `hash()` i Python.]

Dvs. at vi har et univers af mulige heltalskeys: $U = \{0, 1, 2, \ldots u - 1\}$.

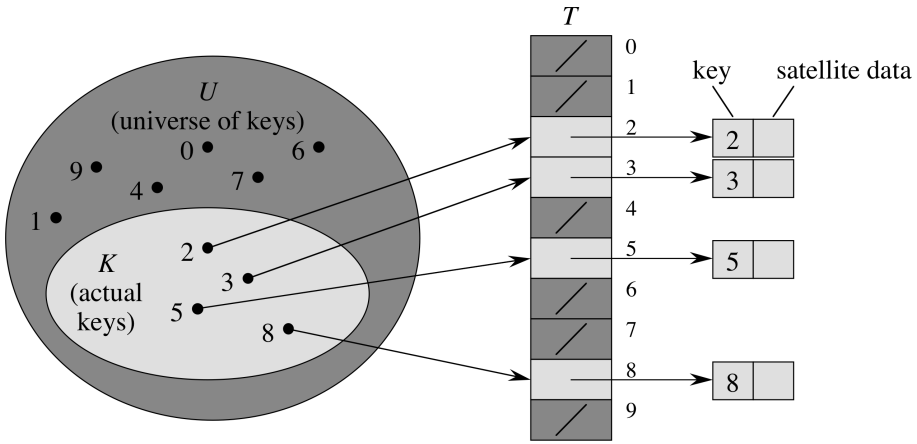
En dictionary gemmer en delmængde $K \subseteq U$, f.eks. $K = \{2, 3, 5, 8\}$

Udgangspunktet i hashing er (lige som i Counting sort) idéen om at bruge keys som indekser i et array:



Dette tager $O(1)$ tid for Search, Insert og Delete.

Problem: pladsforbrug



Problem: Denne idé kan nemt generere pladsspild, fordi array-størrelse er $u = |U|$, mens antallet $n = |K|$ af gemte elementer ofte er meget mindre.

Eksempel: gem 5 CPR-numre. Dvs. keys af typen 180781-2345, der kan ses som heltal i $U = \{0, 1, 2, \ldots 10^{10} - 1\}$. Her er $u = 10^{10}$, mens $n = 5$. Så vi bruger mindst 10^{10} bytes ($>$ 8 Gb RAM) for at gemme 5 CPR-numre.

Gemmes 32 eller 64 bits heltal er $u = 2^{32} \approx 10^{10}$ eller $u = 2^{64} \approx 10^{20}$. Dvs. samme situation eller meget værre.

Hash-funktioner

En løsning på problemet med pladsforbruget: find en funktion h, som mapper fra key's store univers U til et mindre:

$$h : U \to \{0, 1, 2, \ldots m - 1\}.$$

Her er m den ønskede array-størrelse. Ofte vælges $m = O(n)$, så bruges der ikke mere plads på array'et end på elementerne selv.

En sådan funktion kaldes en hash-funktion. Et eksempel på en hash-funktion kan være:

$$h(k) = k \bmod m$$

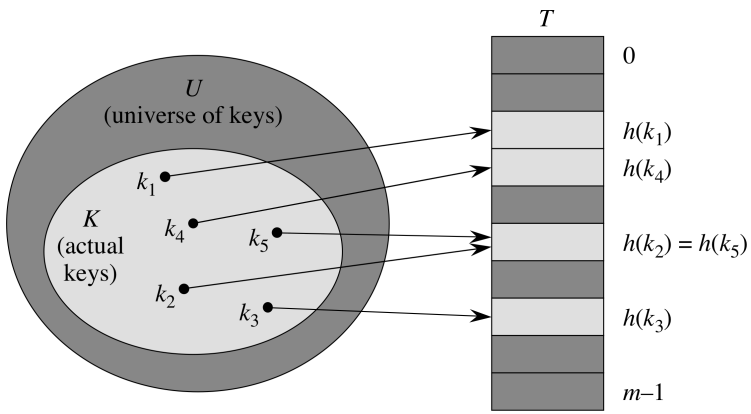
Konkret eksempel med $m = 41$:

$$\begin{aligned} h(12) &= 12 \bmod 41 = 12 && (\text{da } 0 \cdot 41 + 12 = 12) \\ h(100) &= 100 \bmod 41 = 18 && (\text{da } 2 \cdot 41 + 18 = 100) \\ h(479869) &= 479869 \bmod 41 = 5 && (\text{da } 11704 \cdot 41 + 5 = 479869) \end{aligned}$$

Kollisioner

Hashfunktioner løser problemet med pladsforbrug. Men de genererer et andet problem: To keys kan hash'er til samme array index.

$$h(479869) = 479869 \bmod 41 = 5 \quad (\text{da } 11704 \cdot 41 + 5 = 479869)$$
$$h(46) = 46 \bmod 41 = 5 \quad (\text{da } 1 \cdot 41 + 5 = 46)$$

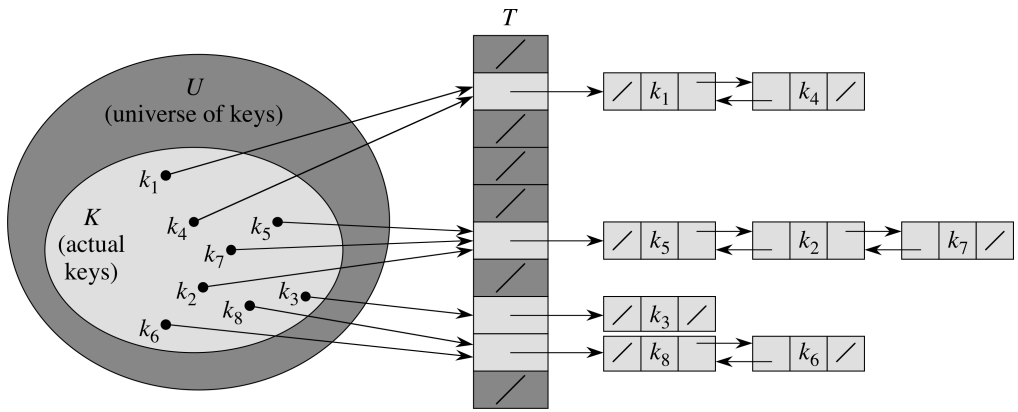


Dette kaldes en kollision.

Når h skal mappe U ind i en mindre mængde $\{0, 1, 2, \ldots m-1\}$, vil der altid være nogle keys k og k' hvor $h(k) = h(k')$. Så kollisioner kan ikke undgås, og vi skal derfor finde en løsning.

Chaining

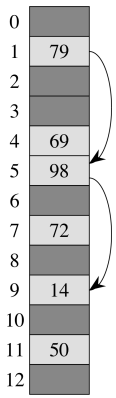
Én simpel løsning: en array-indgang indeholder starten på en lænket liste med alle de indsatte elementer, hvis key hash'er til denne array-indgang. Det kaldes chaining.



Prisen er, at lænkede lister skal gennemløbes under Search og Delete, hvorved tiden for disse operationer stiger fra $\Theta(1)$ til $\Theta(|\text{liste}|)$. Insert er stadig $O(1)$, da vi bare kan indsætte forrest i listen.

Open addressing

Open addressing er et alternativ til chaining: Forsøg at finde tom slot i selve array'et.



Linear probing (alias linear hashing):
$h(k,i) = (h_1(k) + i) \mod m$

Quadratic hashing:
$h(k,i) = (h_1(k) + c_1 \cdot i + c_2 \cdot i^2) \mod m$
[Ikke med i Cormen et al. 4. udgave, udgår af pensum.]

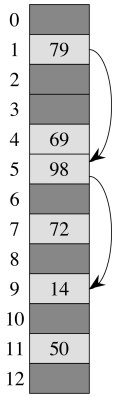
Double hashing
$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$

Her er $h_1(k)$ og $h_2(k)$ to hash-funktioner (kaldet "auxiliary" i bog).

Insert: $i = 0, 1, 2, \ldots$ forsøges til en empty slot findes.

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

Open addressing, bemærkninger



Linear probing (alias linear hashing):
$h(k,i) = (h_1(k) + i) \mod m$

Double hashing
$h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m$

- At implementere Delete er mere kompliceret. En simpel løsning: lad slettede elementer stå, mærk dem som slettede, ryd op en gang i mellem ved at genbygge hashtabellen (indsæt de tilbageværende elementer forfra). For linear hashing findes en mere direkte metode, se Cormen et al. 4. udgave, sektion 11.5.1 (ikke pensum).
- Det er nødvendigt at $n \leq m$ (da alle n elementer ligger i array'et). Gerne $n \approx m/4$ for at undgå at køretider degenererer.
- De gennemsøgte indexer $\{h(k,0), h(k,1), h(k,2), \ldots, h(k,m-1)\}$ bør være $\{0,1,2,\ldots,m-1\}$ for alle k (så hele array'et gennemsøges). Dette kan f.eks. sikres ved at vælge m som et primtal.

Køretid for hashing

Vi ønsker en hash-funktion h som spreder keys fra det konkrete input godt ud over $\{0, 1, 2, \ldots m-1\}$, så der bliver få kollisioner. Man tænker ofte på gode hash-funktioner, som nogle der mapper keys til indekser på en tilsyneladende tilfældig måde.

Man kan dog for en given hashfunktion altid finde mindst $|U|/m$ (dvs. u/m) keys, som hash'er til samme array-index. Ofte er $u/m > n$, hvad der gør worst case tiden til $\Theta(n)$.

I praksis håber man ofte bare på, at dette ikke sker for ens konkrete input og konkrete valg af hashfunktion. Der findes faktisk metoder til at garantere, at dette sjældent sker - se næste slide om universal hashing.

> I praksis kan man regne med at hashing understøtter Search, Insert og Delete i $O(1)$ køretid, medmindre man er meget uheldig.

Man kan desuden sænke worst case tiden til $O(\log n)$ ved at bruge balancerede søgetræer i stedet for lænkede lister i chaining. Dette gør Java, når den lænkede liste bliver stor.

Universal hashing (ikke pensum)

Betragt følgende familie af hashfunktion:

$$h(k) = ((a \cdot k + b) \mod p) \mod m,$$

hvor p er et fast primtal større end $|U|$ og a, b er faste, men tilfældigt valgte heltal med $1 \leq a \leq p-1$ og $0 \leq b \leq p-1$.

Man kan bevise (ikke pensum) at ovenstående hashfunktion er god i følgende forstand (kaldet universal hashing):

> For alle datasæt kan vi for et tilfældigt valg af a og b forvente så få kollisioner, at operationerne (Search, Insert, Delete) tager $O(1)$ tid.

I Cormen et al. 4. udgave angives en familie mere af hashfunktioner med samme egenskab (og bedre konstanter i udregningshastighed af funktionen), se formel (11.2) på side 285 og formel (11.5) side 290. Denne kan være et godt valg, hvis man selv skal implementere hashtabeller.

Andre anvendelser af hashfunktioner (ikke pensum)

Hashfunktioner har mange anvendelsesområder, udover som implementation (niveau 2) af unordered dictionaries.

Nogle eksempler (ikke pensum):

- Fingerprint af filer og dokumenter.
- Digital signatures.
- Load balancing
- Coordinated sampling

De forskellig anvendelser har hver deres specifikke krav til hashfunktionerne.

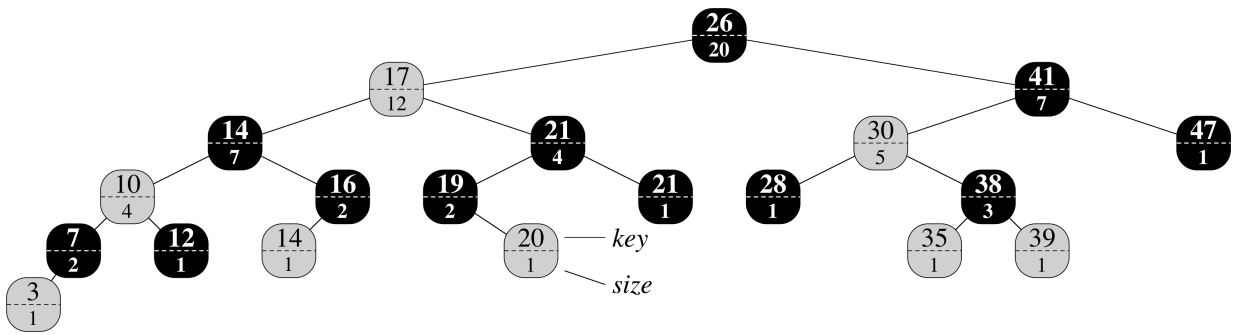
Binære søgetræer med ekstra information i knuderne

Tilføj ekstra information i knuderne

Konkret eksempel på ekstra information i knuder:

> Alle knuder gemmer størrelsen af deres undertræ (dvs. antallet af knuder i deres undertræ).

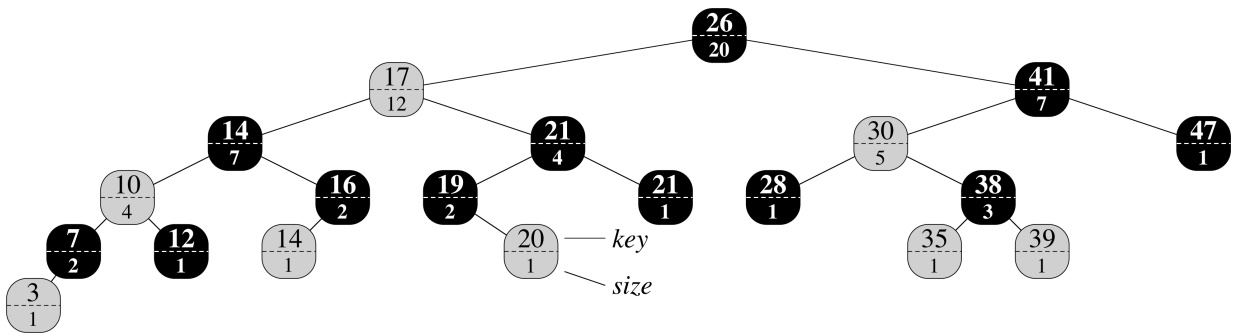
Et binært søgetræ (rød-sort) med denne information tilføjet:



Ny funktionalitet

Målet med ekstra information i knuderne er at tilføje ekstra funktionalitet. F.eks. kan man i eksemplet ovenfor (med størrelse af undertræer gemt i knuder) udføre flg. operationer i $O(\log n)$ tid:

- Find rang af en given nøgle.
- Find nøgle som har en given rang.

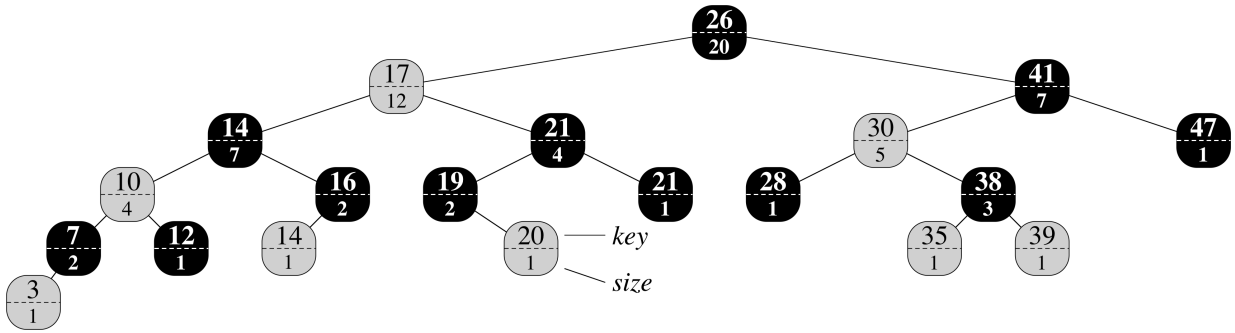


Her er rang nøglens nummer i sorteret orden (blandt de gemte).

Nøgle:	3	7	10	12	14	14	16	17	19	20	21	⋯
Rang:	1	2	3	4	5	6	7	8	9	10	11	⋯

Ny funktionalitet, implementation

- Find rang af en given nøgle.
- Find nøgle som har en given rang.



OS-RANK(T, x)

$r = x.left.size + 1$
$y = x$
while $y \neq T.root$
  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$(x.right, i - r)$

[NB: denne pseudo-kode fra bogen antager, at der bruges et eksplict knude-objekt til at repræsentere NIL (blade).]

Vedligehold den ekstra information i knuderne

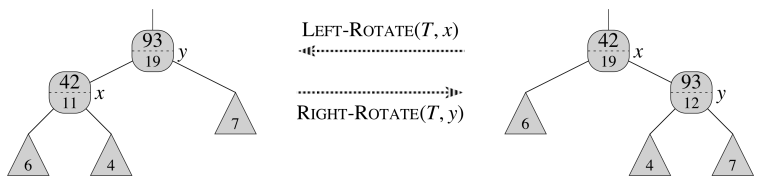
Antag følgende egenskab gælder for den ekstra information:

> Hvis en knudes to børns værdier k_1 og k_2 allerede er korrekte, kan knudes egen værdi k beregnes i $O(1)$ tid. Blades værdier kan beregnes i $O(1)$ tid.

I eksemplet ovenfor: k kan findes som $1 + k_1 + k_2$.

Af antagelsen følger, at værdierne kan vedligeholdes under updates i rød-sorte søgetræer, og dette uden at ændre $O(\log n)$ køretiden:

- Knuder uden for sti fra indsat/slettet knude til rod skal ikke have deres værdi ændret (bottom-up beregning giver uændret resultat).
- Under rebalancering: genberegn for berørte knuder efter hver rotation og efter hvert skridt opad. Til sidst: genberegn på sti fra sidste rebalanceringssted til rod.



Andre eksempler

Vedligeholdelsessystemet ovenfor vil altid fungere når følgende gælder:

> Hvis en knudes to børns værdier k_1 og k_2 allerede er korrekte, kan knudes egen værdi k beregnes i $O(1)$ tid. Blades værdier kan beregnes i $O(1)$ tid.

Flere eksempler på information som opfylder dette (vi antager, at hvert element udover søgenøglen har noget associeret data, som er et tal):

- Maksimum af data-værdier i undertræet.
- Minimum af data-værdier i undertræet.
- Sum af data-værdier i undertræet.

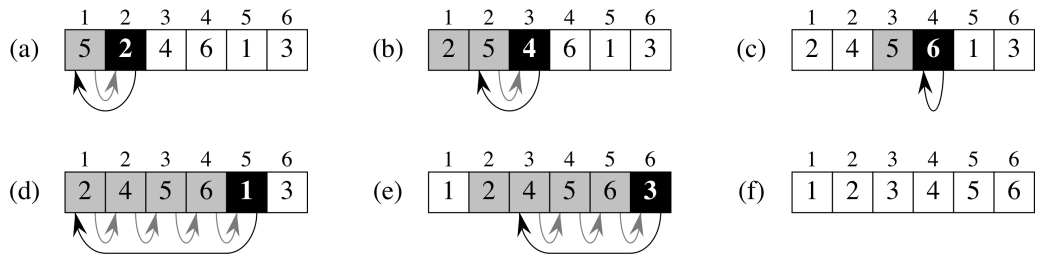
Via sådan information kan man tilføje yderligere funktionalitet til træet. F.eks. kan man søge efter elementer med den maksimale dataværdi. Eller man kan finde gennemsnittet af dataværdierne i en knudes undertræ (gennemsnit = sum af dataværdier / antal knuder).

Invarianter

Invarianter

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

Eksempel: Insertionsort:



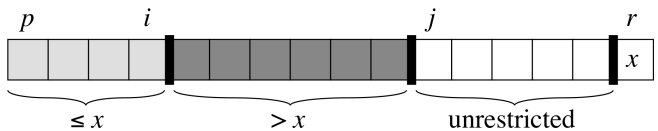
Invariant: Alt til venstre for det sorte felt er sorteret.

Når løkken stopper: hele array'et er til venstre for det sorte felt.

Af invarianten følger så: hele array'et er sorteret. Dvs. algoritmen er korrekt.

Invarianter

Eksempel: Partition fra Quicksort:



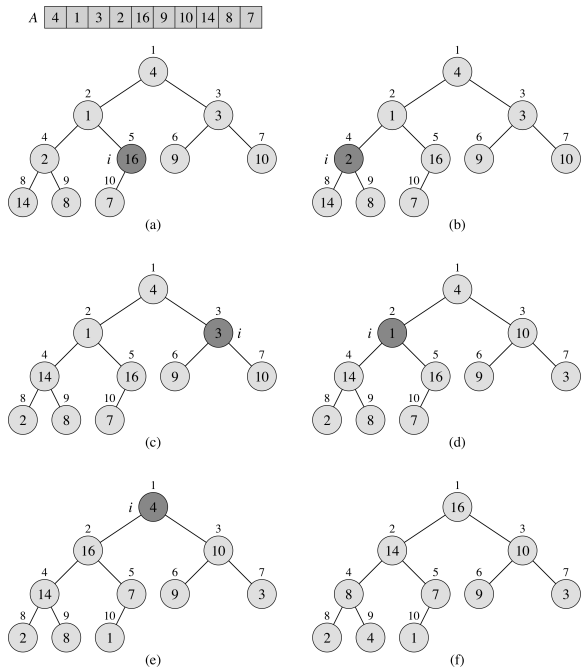
Invariant: $\boxed{\text{Lysegrå del} \leq x < \text{mørkegrå del.}}$

Når løkken stopper: Kun x er hvid, resten enten lysegrå eller mørkegrå.

Af invarianten følger så: array'et er delt i tre dele: "$\leq x$", "$> x$" og x selv. Dvs. algoritmen er korrekt.

Invarianter

Eksempel: Build-Heap:



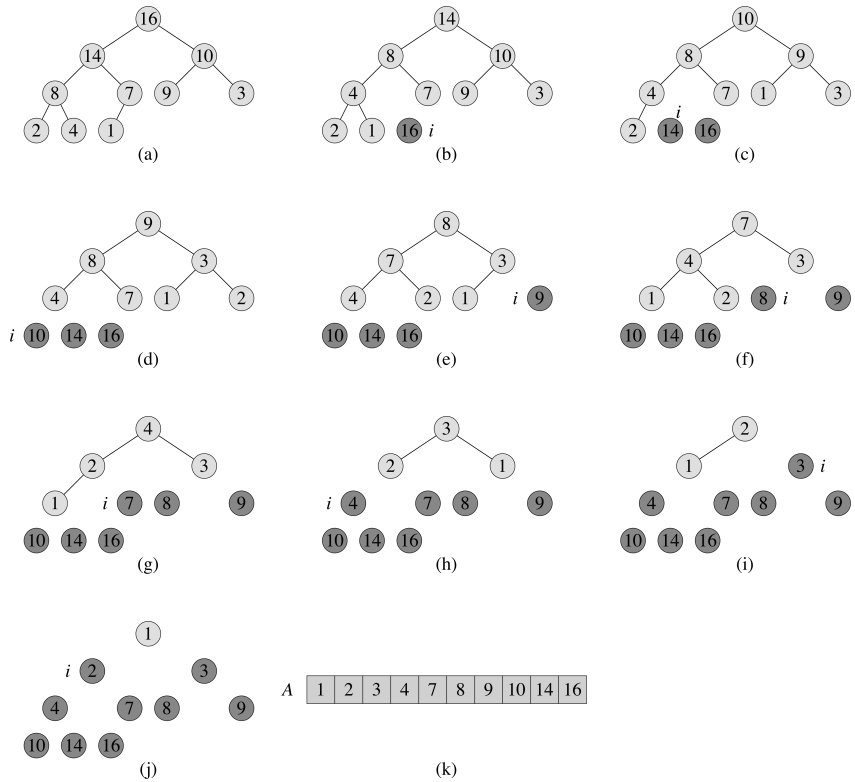
Invariant:

Undertræer, hvis rod har indeks større end den mørke knude, overholder heaporder.

Når løkken stopper: roden af hele træet har indeks større end den mørke knude. Af invarianten følger så: Hele træet overholder heaporder. Dvs. algoritmen er korrekt.

Invarianter

Eksempel: Heapsort (og enhver Selectionsort-baseret sortering):



Invariant: Det mørke er sorteret, og alt i det lyse er $\leq$ det mørke.

Når løkken stopper: hele array'et er mørkt. Af invarianten følger så: hele array'et er sorteret. Dvs. algoritmen er korrekt.

Invarianter

Eksempel: søgning i binære søgetræer.

Invariant: | Hvis søgte element k findes, er det i det undertræ, vi er kommet til. |

Algoritmen må stoppe fordi vi kigger på mindre og mindre undertræer. Når algoritmen stopper: enten er k fundet eller vi er endt i et tomt undertræ.

I det sidste tilfælde følger så af invarianten: k findes ikke i træet. Dvs. algoritmen er korrekt (i begge tilfælde).

Invarianter

Invariant under rebalancering efter indsættelse i et rød-sort træ:

> Der kan være to røde knuder i træk på en sti ét sted i træet, derudover er de rød-sorte krav overholdt. Efter k iterationer er der k færre sorte mellem problemet og roden end i starten.

Invariant under rebalancering efter sletning i et rød-sort træ:

> Der kan være én sværtet knude et sted i træet, og hvis sværtningen tælles med, er de rød-sorte krav overholdt. Efter k iterationer er der k færre sorte mellem problemet og roden end i starten.

Invarianten viser her flere ting, som tilsammen gør algoritmen korrekt:

- Samme case analyse virker hver gang (dækker altid alle muligheder, så algoritmen kan ikke gå i stå, så længe problemet ikke er løst).
- Algoritmen må stoppe, enten ved at problemet er forsvundet, eller at det har nået roden (hvor det let løses).

Invarianter, mere formelt

Invariant for algoritme:

- Et udsagn om indholdet af hukommelsen (variable, arrays,. . .) som er sandt efter alle skridt.
- Ved algoritmens afslutning kan korrekthed udledes af udsagnet (samt de omstændigheder som fik algoritmen til at stoppe).

Induktion

At en invariant gælder efter alle skridt, vises ved hjælp af induktion:

1) Invariant overholdt i starten 2) Invariant overholdt før et skridt $\Rightarrow$ invariant overholdt efter skridtet	$\Rightarrow$	Invariant altid overholdt

(hvor "skridt" ofte er en iteration af en løkke). Dvs: Vis 1) og 2).

Induktion $\sim$ "Dominoprincippet":

1) Brik 1 falder 2) Brik k falder $\Rightarrow$ brik $k+1$ falder	$\Rightarrow$	Alle brikker falder

Brug af invarianter

Invarianter kan bruges på to forskellige detalje-niveauer (med en glidende overgang imellem dem):

1. Som værktøj til at udvikle algoritme-ideer: Med den rette invariant fanges essensen af metoden, og algoritmen skal "blot" skrives ud fra at denne invariant skal vedligeholdes.
2. Som værktøj til at nedskrive kode (eller detaljeret pseudo-kode) og vise denne konkrete kode korrekt.

På niveau 1 er blødere beskrivelser (tekst, figur) passende. Eksemplerne tidligere illustrerer niveau 1.

På niveau 2 må man nedskrive invarianten præcist i termer af konkrete variable fra koden samt argumentere via den konkrete kodes ændringer af disse.

Vi illustrerer nu niveau 2 på et simpelt eksempel med at finde største element i et array

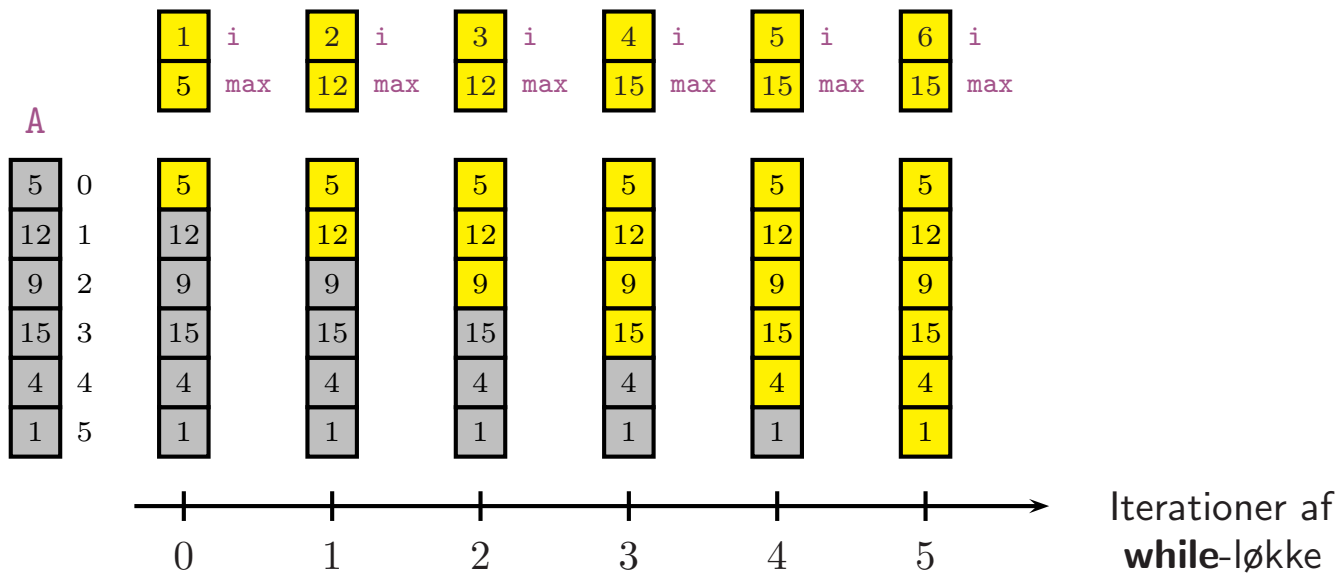
Eksempel

Find største element i array:

```
max = A[0]
i = 1
while i < A.length
  max = maximum(max,A[i])
  i++
```

Invariant: Efter den k'te iteration af while-løkken indeholder `max` den største værdi af $A[0..(i-1)]$.

Vises ved induktion på k.



Divide-and-Conquer algoritmer

Analyse

Divide-and-Conquer algoritmer

Tidligere:

Divide-and-Conquer som algoritme-udviklings-teknik.

I dag:

Analyse af Divide-and-Conquer algoritmer mht. korrekthed og (især) køretid.

Divide-and-Conquer algoritmer, repetition

Det samme som rekursive algoritmer.

Grundidé:

1. Opdel problem i mindre delproblemer (af samme type).
2. Løs delproblemerne ved rekursion (dvs. kald algoritmen selv, men med de mindre input).
3. Konstruer en løsning til problemet ud fra løsningen af delproblemerne.

Basistilfælde: Problemer af størrelse $O(1)$ løses direkte (uden rekursion).

Dette er en generel algoritme-udviklingsmetode, med mange anvendelser.

For hver ny algoritme skal punkt 1 og punkt 3 udvikles. Punkt 2 er altid det samme. Løsning for basistilfældet skal også udvikles, men er som regel triviel.

Generel struktur af Divide-and-Conquer kode

Hvis basistilfælde ($n = O(1)$):

- Arbejde

Hvis ikke basistilfælde:

- Arbejde
- Rekursivt kald
- Arbejde
- Rekursivt kald
- Arbejde

(Der behøver ikke altid være to rekursive kald. Nogle rekursive algoritmer har bare ét, og nogle har flere end to).

Divide-and-Conquer eksempler

Mergesort:

- Del input op i to dele X og Y (trivielt).
- Sorter hver del for sig (rekursion).
- Merge de to sorterede dele til én sorteret del (reelt arbejde).

Basistilfælde: $n \leq 1$ (trivielt).

Quicksort:

- Del input op i to dele X og Y så $X \leq Y$ (reelt arbejde).
- Sorter hver del for sig (rekursion).
- Returner X efterfulgt af Y (trivielt)

Basistilfælde: $n \leq 1$ (trivielt).

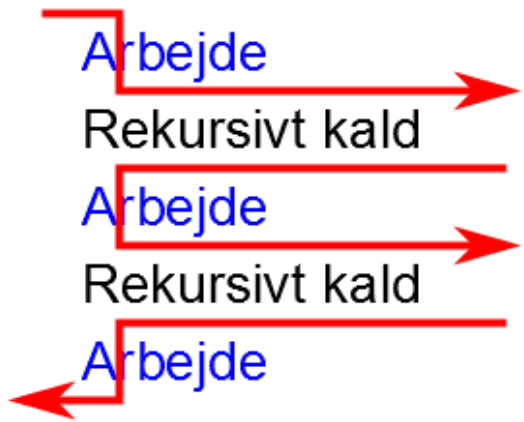
Et tredie eksempel er inorder gennemløb af et binært søgetræ: To rekursive kald (på højre og venste undertræ) med $O(1)$ arbejde imellem sig (udskrivning af nøgle i aktuelle knude).

Hvad sker der når Divide-and-Conquer algoritmer udføres?

Flow of control (lokalt set, for ét kald af algoritmen):

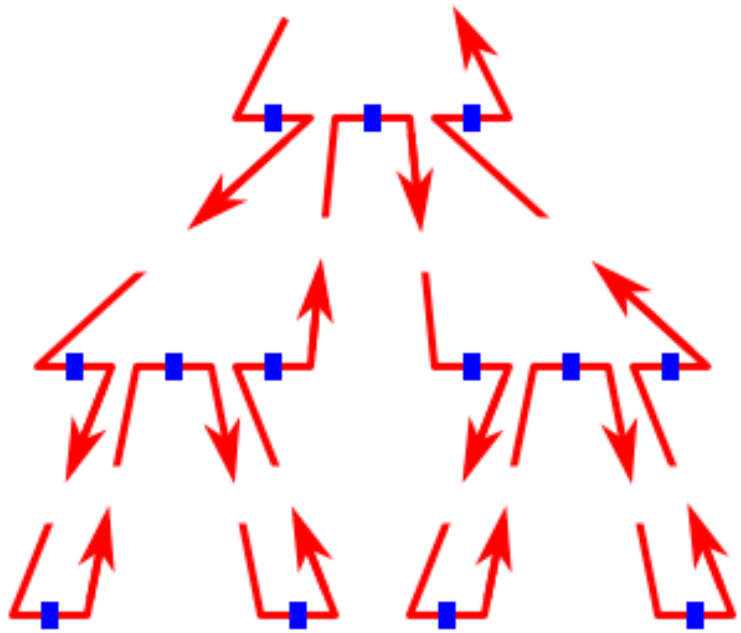
Basistilfælde

Ikke basistilfælde



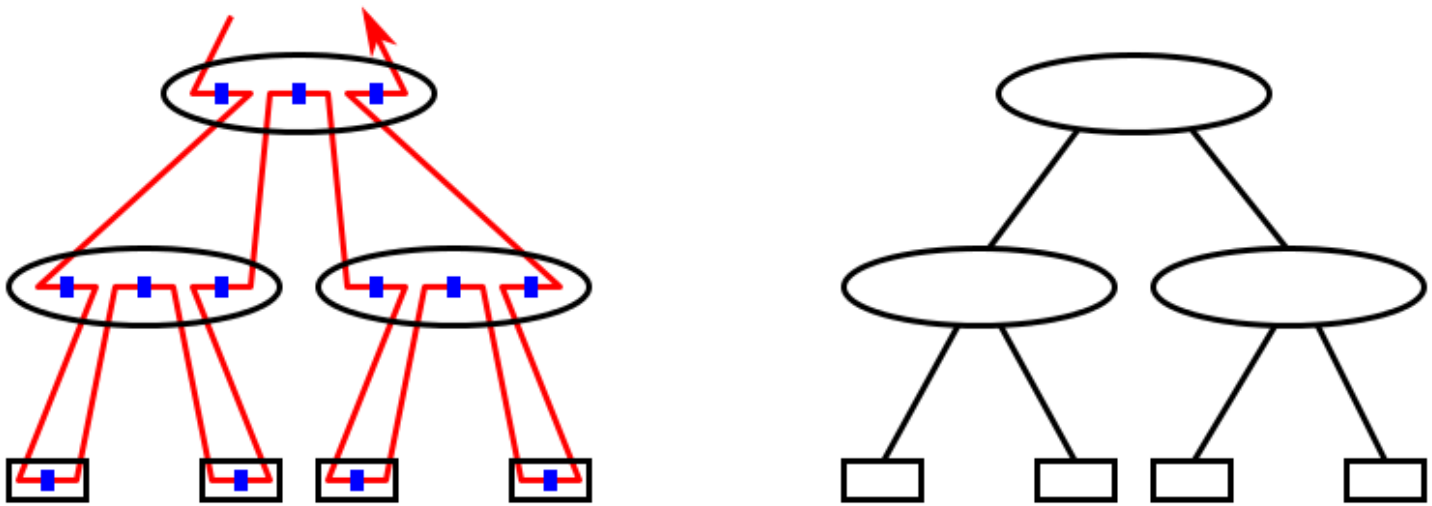
Hvad sker der når Divide-and-Conquer algoritmer udføres?

Her ses det globale flow of control, som opstår:



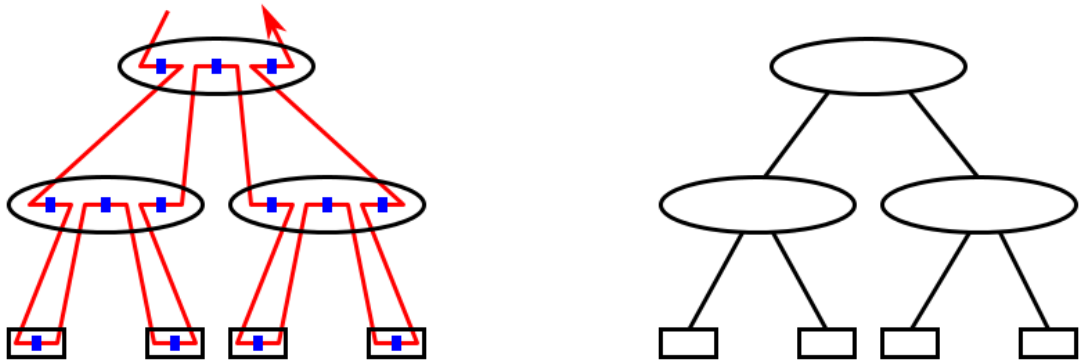
Rekursionstræer

Denne struktur for flow of control kan illustreres via træer, som har én knude for hvert kald af koden. Basistilfældene er blade, ikke-basistilfælde er indre knuder, og træets fanout (antal børn af knuder) er lig med antal rekursive kald i koden.



Disse træer kaldes rekursionstræer.

Hvad sker der når Divide-and-Conquer algoritmer udføres?



På ethvert givet tidspunkt er den samlede udførsel (den røde sti) nået til en eller anden knude v.

På dette tidspunkt er det kaldet, som v svarer til, der udføres handlinger fra. Alle kald svarende til knuder på stien fra v til roden er sat på pause. Hvert af disse kalds tilstand (indhold af deres variable, hvilken kommando de er nået til, m.m.) opbevares af computeren på en stak, så kaldenes udførsel ikke blandes sammen. Kald svarende til alle andre knuder i træet er enten helt færdige eller slet ikke begyndt.

- Kald af barn i rekursionstræet = push på stak.
- Afslutning af en knudes udførsel = pop fra stak.

Korrekthed af Divide-and-Conquer algoritmer

Korrekthed af Divide-and-Conquer algoritmer argumenteres nedefra og op i rekursionstræet:

- Argumentér for, at base case kald svarer korrekt (som regel trivielt).
- For et ikke-base case kald, argumentér for, at hvis de rekursive kald svarer korrekt, så vil dette sammen med det lokale arbejde gøre, at der svares korrekt i det aktuelle kald.

Dvs. at korrekthed af kald med store input følger af korrekthed af kald med mindre input samt handlingerne involveret i at konstruere en løsning for det store input ud fra løsningerne for de mindre input.

[Formelt kan man også sige, at korrekthed vises via induktion på inputstørrelsen. Basistilfældet for rekursionen er basistilfældet for induktionsbeviset.]

Selve argumentet er individuelt for hver algoritme. Det bliver som regel udviklet sammen med algoritmen (det er i praksis svært at få ideen til algoritmen uden at have ideen til, hvorfor den virker).

Rekursionsligninger

Vi vil nu kigge på køretider for rekursive algoritmer. Vi skal først se, hvordan disse kan beskrives ved rekursionsligninger.

Kald worst case køretiden på input af størrelse n for $T(n)$. Hvis en rekursiv algoritme for problemer af størrelse n laver a rekursive kald, som alle er på delproblemer af størrelse n/b, og laver $\Theta(f(n))$ lokalt arbejde, må der for køretiden $T(n)$ gælde:

$$T(n) = \begin{cases} a \cdot T(n/b) + \Theta(f(n)) & \text{hvis } n > 1 \\ \Theta(1) & \text{hvis } n \leq 1 \end{cases}$$

Sidste linie er altid den samme og udelades derfor ofte. Ofte er det også underforstået, at vi bruger asymptotisk notation (så vi skriver $f(n)$ i stedet for $\Theta(f(n))$ og 1 i stedet for $\Theta(1)$).

Eksempel: Mergesort har to rekursive kald af størrelse $n/2$ og laver $\Theta(n)$ lokalt arbejde. Dens rekursionsligning skrives derfor

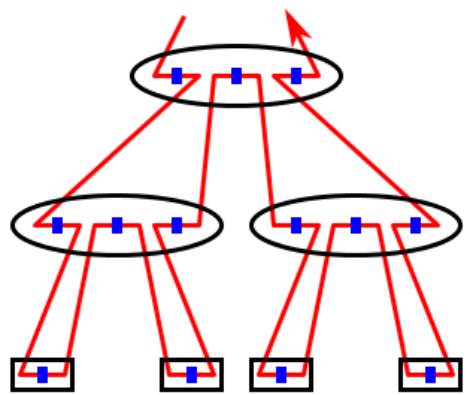
$$T(n) = 2T(n/2) + n$$

Køretid for Divide-and-Conquer algoritmer

En rekursionsligning beskriver strukturen af en rekursiv algoritme og giver en rekursiv beskrivelse af dens køretid (dvs. at $T(n)$ beskrives ud fra T på mindre inputstørrelse).

Men vi ønsker naturligvis også at finde et direkte udtryk (en funktion af n) for algoritmens køretid $T(n)$.

En rekursiv algoritmes samlede arbejde er lig summen af lokalt arbejde i algoritmens rekursiontræ:



Dvs. at vi ønsker at finde denne sum.

Rekursionstræsmetoden til beregning af køretid

For en rekursiv algoritme (eller en rekursionsligning), annotér knuderne i rekursionstræet (for algoritmen eller ligningen) med

- Input størrelsen for kaldet til knude.
- Det resulterende arbejde udført i denne knude.

Find derefter summen af arbejdet i alle knuder på følgende måde:

- Find først højden af træet (antal lag).
- Sum hvert lag i rekursionstræet sammen for sig.
- Sum de resulterende værdier for alle lag.

Eksempler følger.

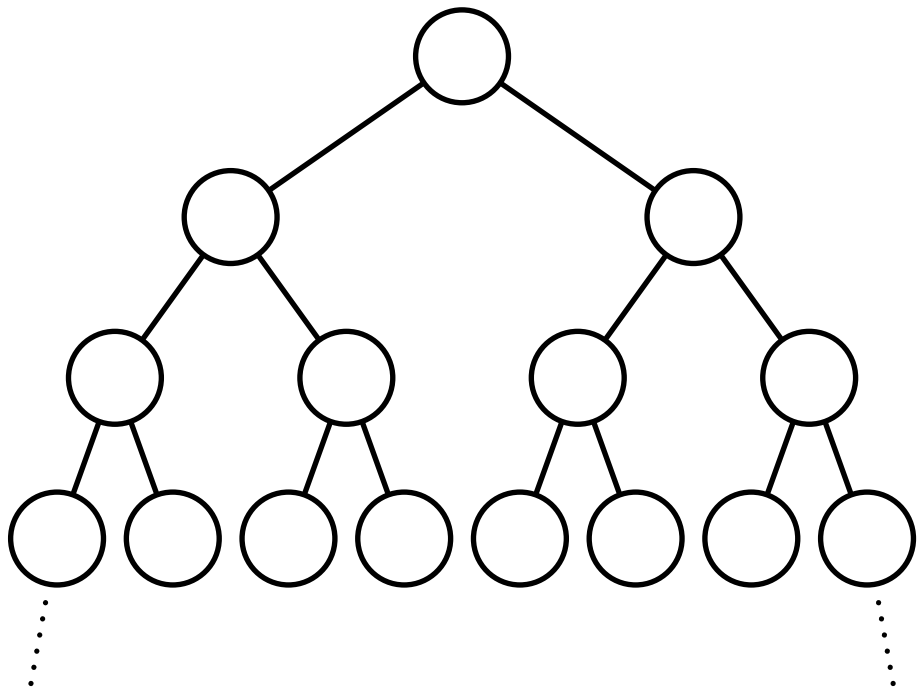
Eksempel 1

$$T(n) = 2T(n/2) + n$$

$$a = 2$$

$$b = 2$$

$$f(n) = n \; T(n) = 2T(n/2) + n$$
$$a = 2,\ b = 2,\ f(x) = x$$



Nogle matematiske facts til brug for de næste eksempler

Som bekendt gælder følgende sætning for $c > 1$:

$$1 + c + c^2 + c^3 + \cdots + c^k = \frac{c^{k+1} - 1}{c - 1} = c^k \cdot \frac{c - 1/c^k}{c - 1} = \Theta(c^k).$$

Konkret eksempel: For $c = 2$ siger sætningen at

$$1 + 2 + 4 + 8 + \cdots + 2^k = 2^{k+1} - 1 = 2 \cdot 2^k - 1 = \Theta(2^k).$$

Denne sætning bør huskes sådan:

> Hvis elementerne i en sum ændrer sig eksponentielt, så er hele summen domineret af det største led.

Eksponentielt voksende/faldende: største led = sidste/første led.

Bevis (ikke pensum): Sæt $S = 1 + c + c^2 + c^3 + \cdots + c^k$, så haves at $S(c-1) = S \cdot c - S$ er lig $(c + c^2 + c^3 + \cdots + c^{k+1}) - (1 + c + c^2 + c^3 + \cdots + c^k) = c^{k+1} - 1$, så $S = (c^{k+1} - 1)/(c - 1)$.

Nogle matematiske facts til brug for de næste eksempler

Andre facts:

- $(a^b)^c = a^{bc} = (a^c)^b$
- $a^{\log_b n} = n^{\log_b a}$
- $\log_b a = \log_c a / \log_c b$ (f.eks. $\log_b a = \ln a / \ln b$).

Sidste fact giver os en måde at beregne $\log_b a$ på via lommeregner (hvor den naturlige logaritme ln findes).

Sidste fact kan også skrives som $\log_b x = \frac{1}{\log_c b} \cdot \log_c x$, hvilket viser, at logaritmer med forskellige grundtal (her b og c) er en konstant faktor fra hinanden: $\log_b x = \Theta(\log_c x)$.)

Bevis (ikke pensum) for midterste fact: $a^{\log_b n} = (b^{\log_b a})^{\log_b n} = (b^{\log_b n})^{\log_b a} = n^{\log_b a}$.

Bevis (ikke pensum) for sidste fact: $\log_b a = \log_c a / log_c b \Leftrightarrow log_c b \cdot \log_b a = \log_c a \Leftrightarrow c^{log_c b \cdot \log_b a} = c^{\log_c a} \Leftrightarrow (c^{log_c b})^{\log_b a} = a \Leftrightarrow b^{\log_b a} = a \Leftrightarrow a = a$.

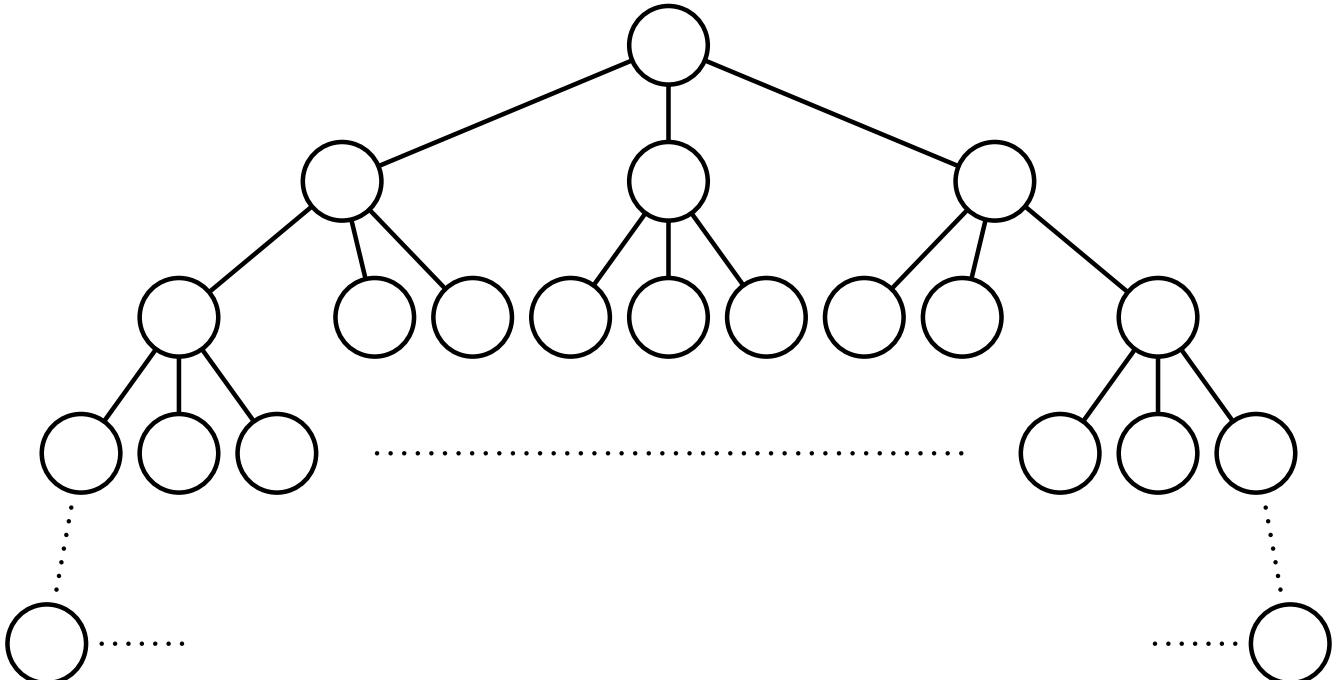
Eksempel 2

$$T(n) = 3T(n/2) + n$$

$$a = 3$$

$$b = 2$$

$$f(n) = n \; T(n) = 3T(n/2) + n$$
$$a = 3,\ b = 2,\ f(x) = x$$



Eksempel 3

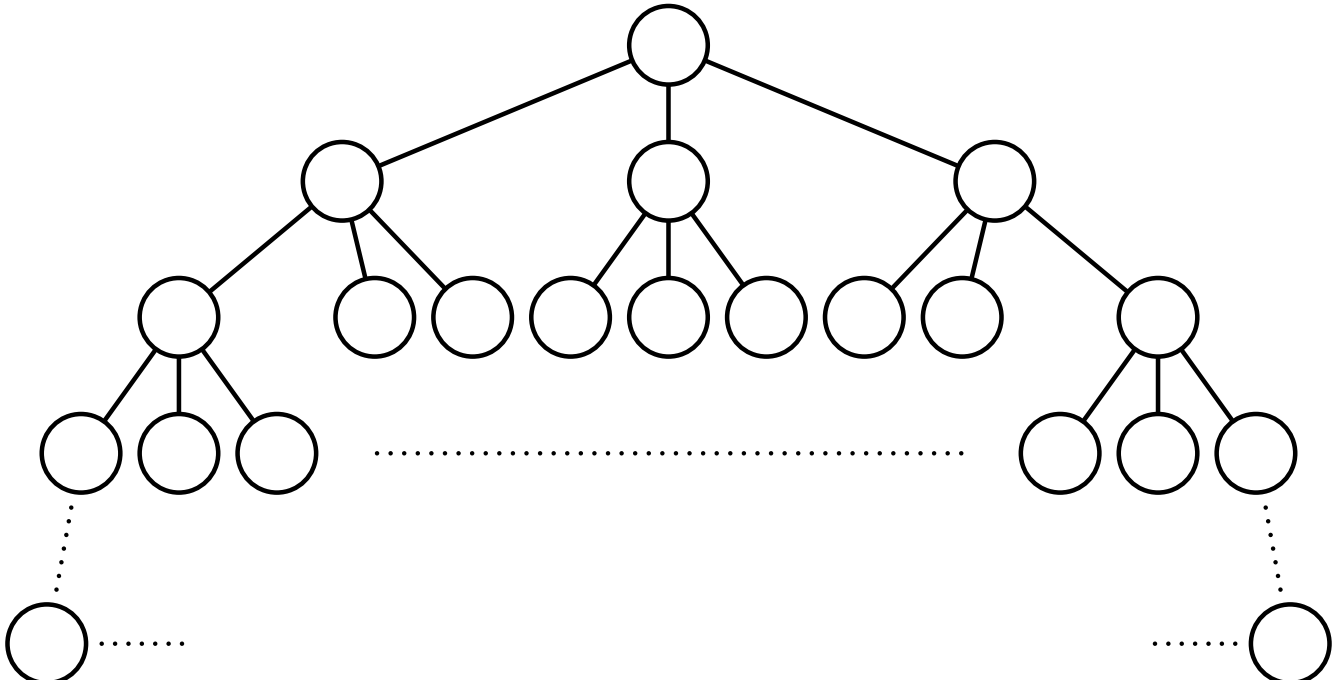
$$T(n) = 3T(n/4) + n^2$$

$$a = 3$$

$$b = 4$$

$$f(n) = n^2 \;\; T(n) = 3T(n/4) + n^2$$

$$a = 3,\ b = 4,\ f(x) = x^2$$



Divide-and-Conquer eksempler

Disse tre eksempler er repræsentative. Dvs. ofte gælder én af følgende:

- Alle lag har ca. lige stor sum, hvorved den samlede sum er antal lag (træets højde) gange denne sum.
- Lagenes sum vokser eksponentiel nedad gennem lagene, hvorved nederste lag dominerer. For at finde dette lags sum, skal man kende træets højde.
- Lagenes sum aftager eksponentiel nedad gennem lagene (= vokser eksponentielt opad gennem lagene), hvorved øverste lag dominerer.

> En generisk løsning af disse tre cases er indholdet af bogens sætning side 102–3 [3. udgave: side 94], kaldet Master Theorem.

De fleste rekursive algoritmer beskrives ved en rekursionsligning, der passer ind i Master Theorem. Hvis den ikke passer i Master Theorem, må man forsøge rekursionstræsmetoden direkte (det kan man også gøre, hvis den passer i Master Theorem, naturligvis).

Master Theorem

Rekursionsligningen

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

har følgende løsning, hvor $\alpha = \log_b a$:

1. Hvis $f(n) = O(n^{\alpha-\epsilon})$ for et $\epsilon > 0$ så gælder $T(n) = \Theta(n^{\alpha})$.
2. Hvis $f(n) = \Theta(n^{\alpha}(\log n)^k)$ for et $k \geq 0$ så gælder $T(n) = \Theta(n^{\alpha}(\log n)^{k+1})$.
3. Hvis $f(n) = \Omega(n^{\alpha+\epsilon})$ for et $\epsilon > 0$ så gælder $T(n) = \Theta(f(n))$.

Ekstra betingelse: I case 3 skal der også gælde, at der findes et $c < 1$ og et n_0 som opfylder at $a \cdot f(n/b) \leq c \cdot f(n)$ når $n \geq n_0$.

> Kort sagt: case afgøres af forholdet mellem voksehastighederne for $f(n)$ og for $n^{\alpha}(\log n)^k$ (hvor $k \geq 0$).

Er de ens, har vi case 2. Er $f(n)$ mindre (med mindst en faktor n^{ϵ}), har vi case 1. Er $f(n)$ større (med mindst en faktor n^{ϵ}), har vi case 3 (hvis ekstrabetingelsen er opfyldt.).

Master Theorem brugt på de tre eksempler

Eksempel 1:

$$T(n) = 2T(n/2) + n$$

- $a = 2$
- $b = 2$
- $f(n) = n$
- $\alpha = \log_2 2 = 1$
- $f(n) = n = \Theta(n^1) = \Theta(n^\alpha (\log n)^0)$ [dvs. $k = 0$]

Så vi er i case 2, så der gælder $T(n) = \Theta(n^\alpha (\log n)^{0+1}) = \Theta(n \log n)$.

Master Theorem brugt på de tre eksempler

Eksempel 2:

$$T(n) = 3T(n/2) + n$$

- $a = 3$
- $b = 2$
- $f(n) = n$
- $\alpha = \log_2 3 = \ln(3)/\ln(2) = 1.5849\ldots$
- $f(n) = n = O(n^{1.5849\cdots-\epsilon}) = O(n^{\alpha-\epsilon})$ for f.eks. $\epsilon = 0.1$.

Så vi er i case 1, så der gælder $T(n) = \Theta(n^{\alpha}) = \Theta(n^{1.5849\ldots})$.

Master Theorem brugt på de tre eksempler

Eksempel 3:

$$T(n) = 3T(n/4) + n^2$$

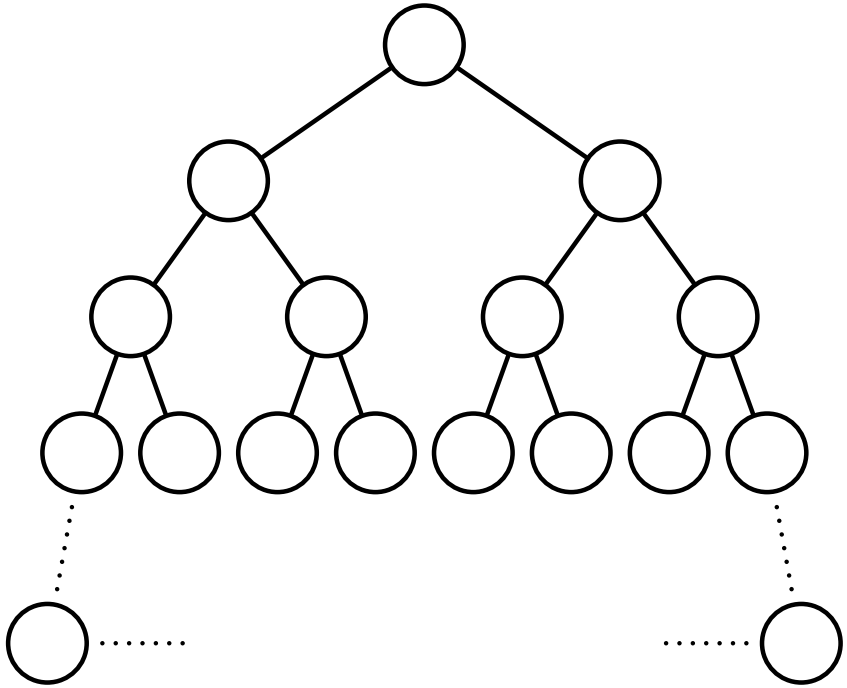
- $a = 3$
- $b = 4$
- $f(n) = n^2$
- $\alpha = \log_4 3 = \ln(3)/\ln(4) = 0.7924\ldots$
- $f(n) = n^2 = \Omega(n^{0.7924\cdots+\epsilon}) = \Omega(n^{\alpha+\epsilon})$ for f.eks. $\epsilon = 0.1$.

Så vi er i case 3, så der gælder $T(n) = \Theta(f(n)) = \Theta(n^2)$.

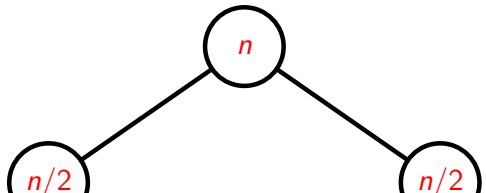
I case 3 skal vi dog også checke den ekstra betingelse: Med $c = 3/16 < 1$ og $n_0 = 1$ opfyldes at $a \cdot f(n/b) = 3(n/4)^2 = 3/16 \cdot n^2 \leq c \cdot f(n) = c \cdot n^2$ for alle $n \geq n_0$.

Eksempel 4

$T(n) = 2T(n/2) + n \log n, \qquad a = 2,\ b = 2,\ f(x) = x \log x$



Tegn træ med fanout $a = 2$.



Eksempel 4

Vi har set at det i'te lag i rekursionstræet laver $n \log(n/2^i)$ arbejde, hvilket kan skrives som $n(\log(n) - \log(2^i)) = n(\log(n) - i)$. Dette udtryk falder, når i stiger, så alle lag laver *højst* det samme arbejde som det første lag ($i = 0$). Det første lag laver $n \log n$ arbejde og der er $\log n$ lag i alt, så det samlede arbejde er $O(n \log^2 n)$.

Vi ser nu på den øverste halvdel af lagene, dvs. på lag i for $i = 1, 2, \ldots, k = \frac{1}{2} \log n$.

Arbejdet for lag i falder, når i stiger, så hvert af disse lag laver *mindst* det samme arbejde som det sidste af dem (lag k). Arbejdet for lag k er

$$n(\log(n) - k) = n(\log(n) - \tfrac{1}{2} \log n) = \tfrac{1}{2} n \log n.$$

Der er derfor $k = \frac{1}{2} \log n$ lag, som hver laver mindst $\frac{1}{2} n \log n$ arbejde. Det samlede arbejde er derfor $\Omega(n \log^2 n)$.

I alt har vi vist at det samlede arbejde er $\Theta(n \log^2 n)$.

Master Theorem brugt på eksempel 4

Eksempel 1:

$$T(n) = 2T(n/2) + n \log n$$

- $a = 2$
- $b = 2$
- $f(n) = n \log n$
- $\alpha = \log_2 2 = 1$
- $f(n) = n \log n = \Theta(n^1 (\log n)^1) = \Theta(n^\alpha (\log n)^1)$ [dvs. $k = 1$]

Så vi er i case 2, så der gælder $T(n) = \Theta(n^\alpha (\log n)^{1+1}) = \Theta(n(\log n)^2)$.

[Bemærk: $(\log n)^2$ skrives oftest som $\log^2 n$.]

Master Theorem kan ikke altid bruges

Master Theorem kan ses som en forudberegnet rekursionstræsmetode, der dækker mange rekursionsligninger.

Mht. løsning af rekursionsligninger er det til eksamen i DM507 nok at kunne følgende:

- Bruge Master Theorem i situationer, hvor det kan anvendes.
- Genkende situationer, hvor Master Theorem ikke kan anvendes.

Af hensyn til det andet punkt giver vi nu et eksempel på en rekursionsligning, hvor Master Theorem *ikke* kan anvendes.

For det viste tilfælde *er* det faktisk muligt at gennemføre rekursionstræsmetoden og finde køretiden på den måde. Vi viser også hvordan (men dette er ikke pensum til eksamen).

Eksempel 5

$$T(n) = T(n/3) + T(2n/3) + n$$

Denne rekursionsligning angiver køretiden for en rekursiv algoritme, som laver $O(n)$ lokalt arbejde og som laver to rekursive kald, et med størrelse $n/3$ og et med størrelse $2n/3$.

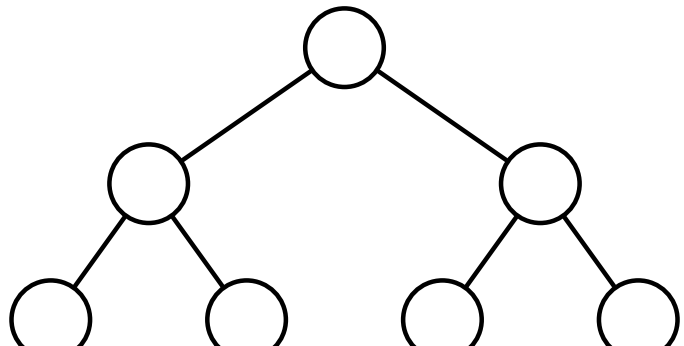
Denne rekursionsligning er desværre *ikke* af den type, som Master Theorem omhandler. For dem skal alle rekursive kald have samme størrelse (n/b):

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

Vi kan derfor ikke bruge Master Theorem.

Vi viser nu, hvordan vi alligevel kan løse rekursionsligningen med rekursionstræsmetoden.

$$T(n) = T(n/3) + T(2n/3) + n$$



Eksempel 5

En knude med input af størrelse x laver $f(x) = x$ arbejde. Den har to børn med input af størrelse $x/3$ og $2x/3$, som derfor laver $f(x/3) = x/3$ og $f(2x/3) = 2x/3$ arbejde. Da $x/3 + 2x/3 = x$, ser vi at knudens arbejde er lig summen af den børns arbejde.

Deraf følger at summen af arbejde i lag k er lig summen af arbejdet i lag $k + 1$, hvis lag $k + 1$ er et fuldt lag (alle knuder i lag k har to børn).

Så alle fulde lag har samme sum af arbejde. For rodens lag er denne sum klart n, så for alle fulde lag er summen af arbejdet i laget lig n. Der er $\log_3 n$ fulde lag, så det samlede arbejde er $\Omega(n \log n)$.[1]

For ikke-fulde lag kan summen kun være mindre (blade har ingen børn, så deres input-størrelse overføres ikke til næste lag), dvs. højst n. Der er $\log_{3/2} n$ lag i alt (fulde og ikke-fulde), så det samlede arbejde er $O(n \log n)$.[1]

Alt i alt har vi vist at det samlede arbejde er $\Theta(n \log n)$.

[1] Her bruges at logaritmer med forskellige grundtal er en konstant faktor fra hinanden, jvf. matematisk fact på side 16.

Floors og ceilings i rekursionsligninger

Der er en detalje, som vi indtil nu ikke har snakket om. Vi bruger Mergesort som eksempel.

Rekursionsligningen for Mergesort har vi skrevet som

$$T(n) = 2T(n/2) + n. \tag{1}$$

Men de to rekursive kald kan jo ikke være helt ens i størrelse hvis n er ulige. Dvs. at rekursionsligningen faktisk hedder

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n. \tag{2}$$

Betyder denne forskel noget for vores beregning af køretiden?

Svaret er nej, som vi skal se.

Floors og ceilings i rekursionsligninger

For en sti ned gennem rekursiontræet lader vi n_i betegne størrelsen af et input på lag nummer i (rodens lag sættes til nummer 0).

For rekursionsligningen $T(n) = 2T(n/2) + n$ gælder:

$$n_0 = n$$

$$n_1 = n_0/2 = n/2$$

$$n_2 = n_1/2 = (n/2)/2 = n/2^2$$

$$n_3 = n_2/2 = (n/2^2)/2 = n/2^3$$

$$\vdots$$

$$n_i = n/2^i$$

Floors og ceilings i rekursionsligninger

Vi ser nu på rekursionsligningen $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$.

Eftersom

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1.$$

får vi:

$$\begin{gathered}
n_0 = n \\
n_1 \leq \lceil n_0/2 \rceil < n_0/2 + 1 = n/2 + 1 \\
n_2 \leq \lceil n_1/2 \rceil < n_1/2 + 1 < (n/2 + 1)/2 + 1 = n/2^2 + 1/2 + 1 \\
n_3 \leq \lceil n_2/2 \rceil < n_2/2 + 1 < (n/2^2 + 1/2 + 1)/2 + 1 = n/2^3 + 1/2^2 + 1/2 + 1 \\
\vdots \\
n_i < n/2^i + 1/2^{i-1} + \cdots + 1/2 + 1
\end{gathered}$$

og vi får også:

$$\begin{gathered}
n_0 = n \\
n_1 \geq \lfloor n_0/2 \rfloor > n_0/2 - 1 = n/2 - 1
\end{gathered}$$

Floors og ceilings i rekursionsligninger

Så for rekursionsligningen $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ har vi vist

$$n_i < n/2^i + (1/2^{i-1} + \cdots + 1/2 + 1),$$
$$n_i > n/2^i - (1/2^{i-1} + \cdots + 1/2 + 1).$$

Udtrykket i parents er lig $1 + c + c^2 + \cdots + c^{i-1}$ for $c = 1/2$, og er derfor (jvf. slide med matematiske facts på side 15) lig med:

$$\frac{(1/2)^i - 1}{1/2 - 1} = \frac{(1/2)^i - 1}{-1/2} = \frac{1 - (1/2)^i}{1/2} = 2 - (1/2)^{i-1} < 2.$$

For rekursionsligningen $T(n) = 2T(n/2) + n$ viste vi

$$n_i = n/2^i.$$

Floors og ceilings i rekursionsligninger

Så i de to rekursionstræer for

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$$

$$T(n) = 2T(n/2) + n$$

afviger inputstørrelserne n_i i knuderne på et vilkårligt lag i kun med plus/minus 2.

For alle de funktioner f, som vi møder, gælder $f(n+2) = O(f(n))$.

> For sådanne funktioner får vi samme asymptotiske køretid, når vi analyserer rekursionsligninger med og uden floors/ceilings.

Eksempler: for $f(n) = n$ gælder $f(n+2) = n+2 = O(n) = O(f(n))$. For $f(n) = n^2$ gælder $f(n+2) = (n+2)^2 = n^2 + 2n + 4 = O(n^2) = O(f(n))$, for $f(n) = \log n$ gælder $f(n+2) = \log(n+2) \leq \log 2n = 1 + \log n = O(\log n) = O(f(n))$, og for $f(n) = 2^n$ gælder $f(n+2) = 2^{n+2} = 2^2 \cdot 2^n = 4 \cdot 2^n = O(2^n) = O(f(n))$.

Strassens algoritme

Matricer (repetition)

Matrix = firkant af tal:

$$\begin{bmatrix} 1 & 6 & 4 \\ 2 & 5 & 7 \\ 9 & 1 & 1 \end{bmatrix}$$

Ovenstående er en 3×3 matrix.

I dag: alle matricer er $n \times n$ kvadratiske matricer. (Dvs. n angiver sidelængden af matricerne.)

Matricer

Plus for matricer:

$$\begin{bmatrix} 1 & 6 & 4 \\ 2 & 5 & 7 \\ 9 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 2 & 1 \\ 4 & 3 & 2 \\ 5 & 4 & 3 \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} 1 & 6 & 4 \\ 2 & 5 & 7 \\ 9 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 2 & 1 \\ 4 & 3 & 2 \\ 5 & 4 & 3 \end{bmatrix} = \begin{bmatrix} 1+3 & 6+2 & 4+1 \\ 2+4 & 5+3 & 7+2 \\ 9+5 & 1+4 & 1+3 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 5 \\ 6 & 8 & 9 \\ 14 & 5 & 4 \end{bmatrix}$$

Tid? $\Theta(n^2)$.

Optimalt, da output er af størrelse n^2.

Matricer

Gange for matricer:

$$\begin{bmatrix} 1 & 6 & 4 \\ 2 & 5 & 7 \\ 9 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 2 & 1 \\ 4 & 3 & 2 \\ 5 & 4 & 3 \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix}$$

$$\begin{bmatrix} 1 & 6 & 4 \\ \cancel{2} & \cancel{5} & \cancel{7} \\ 9 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 2 & \cancel{1} \\ 4 & 3 & \cancel{2} \\ 5 & 4 & \cancel{3} \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & 33 \\ ? & ? & ? \end{bmatrix}$$

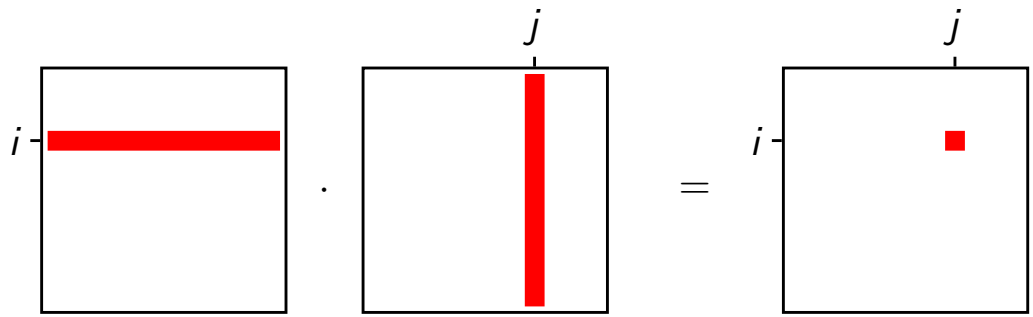
$$33 = 2 \cdot 1 + 5 \cdot 2 + 7 \cdot 3$$

$$\begin{bmatrix} 1 & 6 & 4 \\ 2 & 5 & 7 \\ \cancel{9} & \cancel{1} & \cancel{1} \end{bmatrix} \cdot \begin{bmatrix} 3 & \cancel{2} & 1 \\ 4 & \cancel{3} & 2 \\ 5 & \cancel{4} & 3 \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & 33 \\ ? & 25 & ? \end{bmatrix}$$

$$25 = 9 \cdot 2 + 1 \cdot 3 + 1 \cdot 4$$

Tid? $\Theta(n^3)$. Optimalt?? Andre algoritmer??

Rekursiv algoritme for multiplikation?



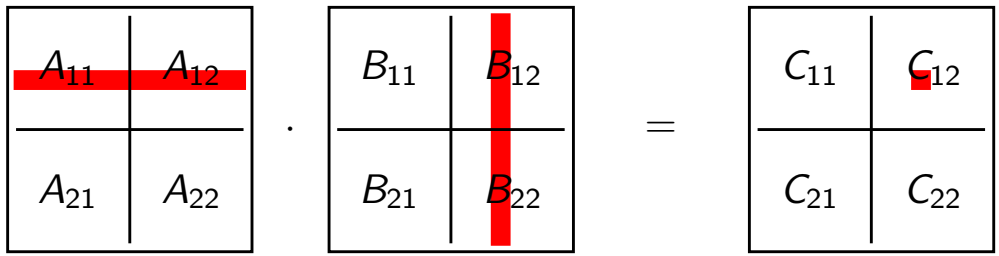
$$\begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$

$$\begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array}$$

Bemærk:

$$A_{11} \cdot B_{12} + A_{12} \cdot B_{22} = C_{12}$$

Rekursiv algoritme for multiplikation



$$A_{11} \cdot B_{11} + A_{12} \cdot B_{21} = C_{11}$$
$$A_{11} \cdot B_{12} + A_{12} \cdot B_{22} = C_{12}$$
$$A_{21} \cdot B_{11} + A_{22} \cdot B_{21} = C_{21}$$
$$A_{21} \cdot B_{12} + A_{22} \cdot B_{22} = C_{22}$$

Matrix addition: $O(n^2)$

Matrix multiplikation: Rekursivt kald til matrixmultiplikation på $n/2 \times n/2$ matricer. (Base case: $n = 1 \Rightarrow$ multiplikation af tal.)

$$T(n) = 8\,T(n/2) + n^2$$

Rekursiv algoritme for multiplikation

$$T(n) = 8T(n/2) + n^2$$

Master theorem:

- $\alpha = \log_b(a) = \log_2(8) = 3$
- $f(n) = n^2$

$n^2 = O(n^{\alpha - 0.1}) \Rightarrow$ Case 1

$$T(n) = \Theta(n^\alpha) = \Theta(n^3)$$

Det samme som den almindelige algoritme. Øv.

Strassen [1969]

Beregn:

$$S_1 = B_{12} - B_{22} \qquad S_6 = B_{11} + B_{22}$$
$$S_2 = A_{11} + A_{12} \qquad S_7 = A_{12} - A_{22}$$
$$S_3 = A_{21} + A_{22} \qquad S_8 = B_{21} + B_{22}$$
$$S_4 = B_{21} - B_{11} \qquad S_9 = A_{11} - A_{21}$$
$$S_5 = A_{11} + A_{22} \qquad S_{10} = B_{11} + B_{12}$$

Tid: $O(n^2)$, da både addition og subtraktion tager denne tid.

Strassen [1969]

Beregn:

$P_1 = A_{11} \cdot S_1$

$P_2 = S_2 \cdot B_{22}$

$P_3 = S_3 \cdot B_{11}$

$P_4 = A_{22} \cdot S_4$

$P_5 = S_5 \cdot S_6$

$P_6 = S_7 \cdot S_8$

$P_7 = S_9 \cdot S_{10}$

7 rekursive kald til matrixmultiplikation på $n/2 \times n/2$ matricer.

Strassen [1969]

Check nu at der gælder:

$$\begin{aligned} P_5+P_4-P_2+P_6 &= A_{11}\cdot B_{11}+A_{12}\cdot B_{21} \\ P_1+P_2 &= A_{11}\cdot B_{12}+A_{12}\cdot B_{22} \\ P_3+P_4 &= A_{21}\cdot B_{11}+A_{22}\cdot B_{21} \\ P_5+P_1-P_3-P_7 &= A_{21}\cdot B_{12}+A_{22}\cdot B_{22} \end{aligned}$$

Dvs. output kan beregnes i $O(n^2)$ tid ud fra $P_1,\ldots,P_7$, eftersom

$A_{11}\cdot B_{11}+A_{12}\cdot B_{21}=C_{11}$

$A_{11}\cdot B_{12}+A_{12}\cdot B_{22}=C_{12}$

$A_{21}\cdot B_{11}+A_{22}\cdot B_{21}=C_{21}$

$A_{21}\cdot B_{12}+A_{22}\cdot B_{22}=C_{22}$

$$T(n)=7T(n/2)+n^2$$

Strassen [1969]

$$T(n) = 7T(n/2) + n^2$$

Master theorem:

- $\alpha = \log_b(a) = \log_2(7) = 2.80735\ldots$
- $f(n) = n^2$

$n^2 = O(n^{\alpha - 0.1}) \Rightarrow$ Case 1

$$T(n) = \Theta(n^\alpha) = O(n^{2.81})$$

Bedre end den almindelige algoritme!

Dynamisk programmering

Kombinatoriske optimeringsproblemer

Kombinatorisk struktur: En struktur opbygget af et endeligt antal enkeltdele. Eksempler:

- Rute fra A til B.
- Pakning af lastbil eller containerskib.
- Undervisningsskema.
- Produktionsplan for y ordrer og x produktionsmaskiner.

Kombinatorisk optimeringsproblem: man ønsker at finde den bedste kombinatoriske struktur blandt mange mulige. Eksempler:

- Hurtigste rute fra A til B.
- Mest profitable pakning af lastbil eller containerskib.
- Undervisningsskema med mindst mulig undervisning efter kl. 16.
- Produktionsplan for y ordrer og x produktionsmaskiner med færrest overskridelser af leveringsfrister.

Dynamisk programmering

Dynamisk programmering [Bellman, 1950-57]: en metode til at udvikle algoritmer til kombinatoriske optimeringsproblemer.

Dynamisk programmering er et specialtilfælde af Divide-and-Conquer metoden—dvs. er en rekursiv metode, som opbygger løsninger til større problemer ud fra løsninger til mindre problemer.

Observation:

- Normalt i rekursive metoder: delproblemer typisk halvt så store og der er ingen gentagelser af delproblemer forskellige steder i rekursionstræet.
- Nogle rekursive metoder: Kald af delproblemer hvis størrelse kun er reduceret med én. Der vil så ofte opstå gentagelser af delproblemer forskellige steder i rekursionstræet, hvilket ofte gør køretiden eksponentiel.

Dynamisk programmering

Kernen i dynamisk programmering er følgende idé:

- Lav en tabel over løsninger på delproblemer, så disse kun skal løses én gang hver. Dette ændrer normalt køretiden fra eksponentiel til polynomiel.

Mere generelt bruges begrebet dynamisk programmering om

- Udvikling af rekursive løsninger for optimeringsproblemer, hvor nogle delproblemer i rekursionen kun er reduceret $O(1)$ i størrelse. Man bruger så idéen ovenfor til at implementere den rekursive løsning effektivt.

Den kreative del er at finde den rekursive beskrivelse af løsningen. At derefter bruge idéen ovenfor er ret ens fra problem til problem.

Dynamisk programmering

Den kreative del er at finde den rekursive beskrivelse af løsningen.

Følgende er ofte en god angrebsvinkel:

1. Hvad kunne være en god beskrivelse af størrelsen af et problem, udtrykt ved ét, to eller evt. flere heltalsindekser? Det giver så en tabel med én, to eller flere dimensioner.
2. Analysér hvordan en optimal løsning for en given problemstørrelse må bestå af en "sidste del" og "resten", hvor man om "resten" kan argumentere, at denne må være en optimal løsning for et mindre problem af samme type. Derved kan fås en rekursiv beskrivelse af løsninger.

Den sidste egenskab kaldes, at der er "optimale delproblemer".

Princippet forstås bedst gennem eksempler.

Eksempel: Maltes problem

En masse guldkæder i overskud:

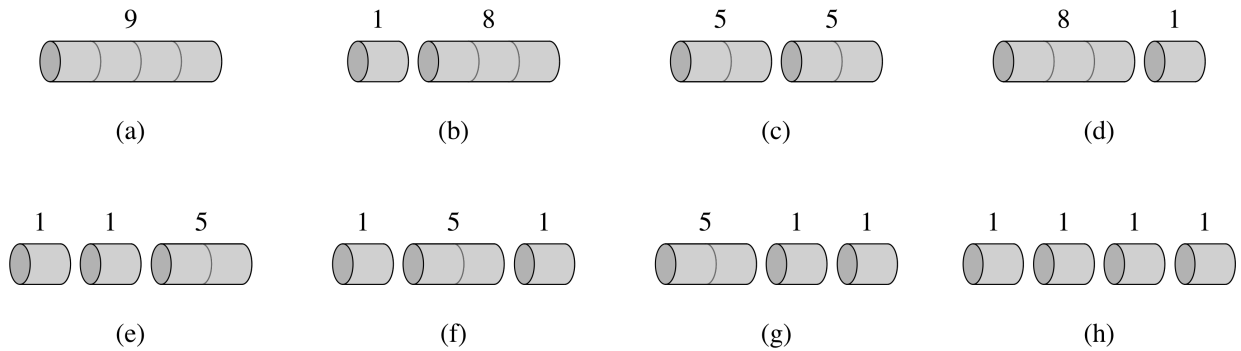
Foto: © Kaspar Wenstrup

Eksempel: Maltes problem

Du har en guldkæde med n led. Den kan deles i mindre længder (med n led tilsammen, dvs. ingen led går tabt). Guldsmeden køber guldkæder af forskellige længder til forskellige priser:

længde i (antal led):	1	2	3	4	5	6	7	8	9	$\cdots$
pris p_i (1.000 kr.):	1	5	8	9	10	17	17	20	24	$\cdots$

Hvordan skal du opdele din lange guldkæde for at optimere din salgspris?



Der er 2^{n-1} forskellige opdelinger, så det er ikke en effektiv algoritme blot at prøve dem alle.

Optimale delproblemer

Enhver opdeling af en kæde af længde n må bestå af:

- Et sidste stykke af længde $k \leq n$.
- En opdeling af resten, dvs. en opdeling af en kæde af længde $n - k$.

Observation (optimale delproblemer):

> For en *optimal* opdelingen af kæden af længde n, må opdelingen af resten selv være optimal for en kæde af længde $n - k$. For hvis der fandtes en ægte bedre opdeling af resten, kunne man bruge den i stedet og derved forbedre den optimale opdeling af kæden af længde n.

Kald *værdien* af en optimal opdeling af en kæde af længde n for $r(n)$.

Det er klart at $r(0) = 0$. Vi vil gerne finde $r(n)$ for $n > 0$.

Rekursiv formel for $r(n)$

Opsummering: en optimal opdeling T for længde n består af:

- Et sidste stykke med længde $k \leq n$.
- En optimal opdeling af resten, dvs. en optimal opdeling af en kæde af længde $n - k$.

Værdien $r(n)$ af T er derfor lig $p_k + r(n - k)$, så det lugter af rekursion.

Men: vi kender desværre ikke k!

Rekursiv formel for $r(n)$

Værdien $r(n)$ af T er derfor lig $p_k + r(n-k)$, så det lugter af rekursion, men vi kender desværre ikke k.

Derfor gør vi sådan:

Lad T_i (for $i = 1 \ldots n$) være en opdelingen bestående af et sidste stykke af længde i, samt en optimal opdeling af resten.

Værdien af T_i er $p_i + r(n-i)$.

T_k har værdi $p_k + r(n-k)$ lige som T og er derfor optimal for længde n.

- Så (mindst) én af T_1, T_2, T_3,..., T_n er optimal for længde n.
- Naturligvis kan ingen T_i have en værdi bedre end optimal.

Heraf:

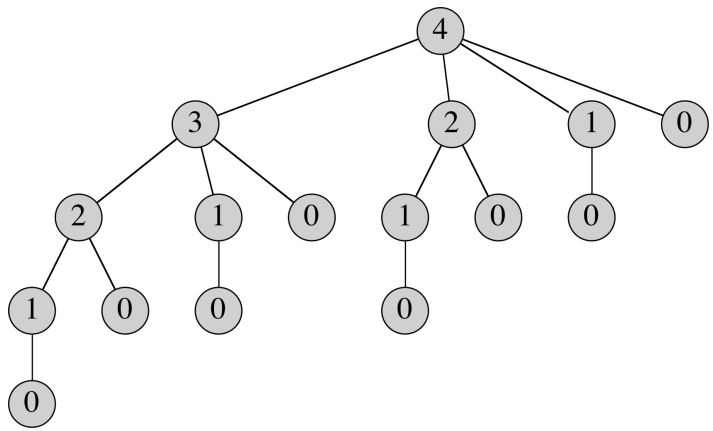
$$r(n) = \max_{1 \le i \le n} (p_i + r(n-i)), \qquad r(0) = 0$$

Beregne de optimale værdier

$$r(n) = \max_{1 \le i \le n} (p_i + r(n-i)), \qquad r(0) = 0$$

Dvs. $r(n)$ er (matematisk set) rekursivt defineret ud fra mindre instanser.

Er rekursion også en god løsning, algoritmisk set?



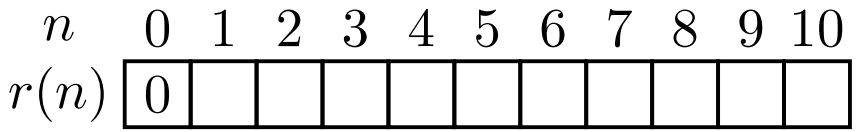
Man kan vise via induktion at der er $1 + (1 + 2 + 4 + \cdots + 2^{n-1}) = 2^n$ knuder i rekursionstræet. Så køretiden vil blive $\Theta(2^n)$

Problemet er gentagelser blandt delproblemers delproblemer.

Brug en tabel

Fokusér i stedet på en tabel over værdien af de optimale løsninger.

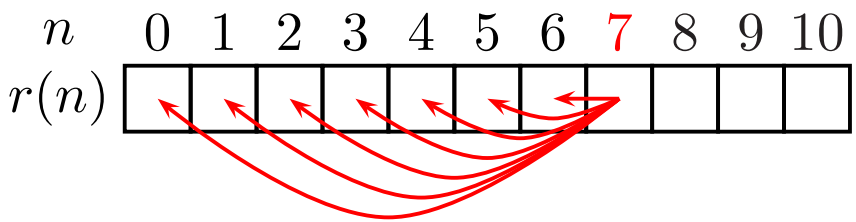
Start: $r(0) = 0$



Et felt kan fyldes ud via

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

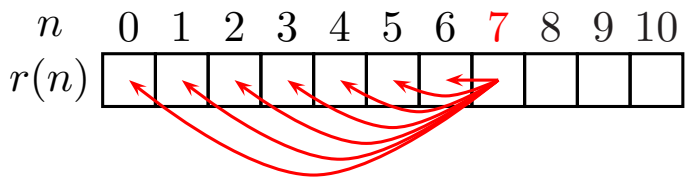
hvis de foregående felter er fyldt ud. Denne afhængighed kan vi illustrere:



Deraf følger, at vi kan beregne $r(n)$ bottom-up, dvs. for stigende n.

Køretid

$$r(n) = \max_{1 \le i \le n} (p_i + r(n-i)), \qquad r(0) = 0$$



Denne beregning kan laves med to simple for-loops (det ydre går gennem tabellen felt for felt, det inderste finder max for hvert felt):

```
r[0] = 0
for k = 1 to n:
    max = −∞
    for i = 1 to k:
        x = p[i] + r[k − i]:
        if x > max:
            max = x
    r[k] = max
```

Tid: $O(1 + 2 + 3 + 4 + \cdots + n) = \Theta(n^2)$

Eksempel

Brug

$$r(n) = \max_{1 \le i \le n} (p_i + r(n-i)), \qquad r(0) = 0$$

og priserne p_i:

længde i	1	2	3	4	5	6	7	8	9	10
pris p_i	1	5	8	9	10	17	17	20	24	26

til at udfylde tabellen over $r(n)$ fra højre mod venstre:

n	0	1	2	3	4	5	6	7	8	9	10
$r(n)$	0	1	5	8	10	13	17	18	22	..25	..27

Find selve løsningen

Tallet $r(n)$ er kun *værdien* af den optimale løsning. Hvad hvis vi gerne vil have *selve løsningen* (de enkelte længder, guldkæden skal brydes op i)?

Gem længden $s(n)$ af det *sidste stykke* for en optimal løsning for længde n. Dvs. gem det i som giver max i den rekursive ligning.

$$r(n) = \max_{1 \le i \le n} (p_i + r(n-i)), \qquad r(0) = 0$$

længde i	1	2	3	4	5	6	7	8	9	10
pris p_i	1	5	8	9	10	17	17	20	24	26

længde n	0	1	2	3	4	5	6	7	8	9	10
optimal værdi $r(n)$	0	1	5	8	10	13	17	18	22	25	27
sidste længde $s(n)$	0	1	2	3	2	2	6	1	2	3	2

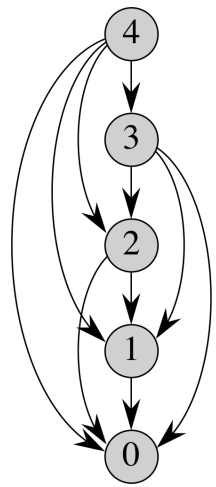
while $n > 0$
  print $s[n]$
  $n = n - s[n]$

Memoization

Rekursion: $\Theta(2^n)$. Struktureret tabeludfyldning: $\Theta(n^2)$

Kan de to kombineres? Ja.

```
GuldKæde(n)
    if n = 0
        return 0
    else if r[n] allerede udfyldt i tabel
        return r[n]
    else
        x = max_{1≤i≤n}(p_i + GuldKæde(n − i))
        r[n] = x
        return x
```



En pil i figuren, der viser et delproblems afhængighed af andre, vil blive en kant i rekursionstræet præcis én gang (første gang delproblemet nås).

Så samme køretid $\Theta(n^2)$ og pladsforbrug $\Theta(n)$ som for bottom-up udfyldning af tabellen. Men nok en lidt dårligere konstant i praksis.

Dynamisk programmering

Flere eksempler

Eksempel 1: Længste fælles delsekvens

Alfabet = mængde af tegn:

$\{a,b,c,\ldots,z\}$, $\{A,C,G,T\}$, $\{0,1\}$

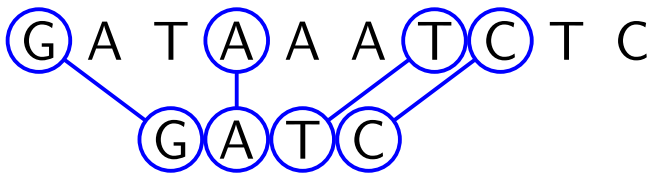
Streng = sekvens $x_1 x_2 x_3 \ldots x_n$ af tegn fra et alfabet:

helloworld

GATAAATCTGGTCTTATTTCC

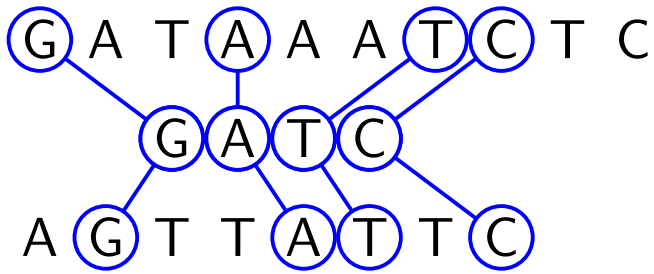
00101100101010001111

Delsekvens = delmængde af tegnene i streng, i uændret rækkefølge:

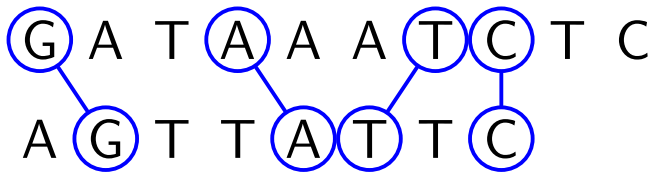


Længste fælles delsekvens

Fælles delsekvens for to strenge:



Eller blot:



Længste fælles delsekvens (Longest Common Subsequence, LCS):

For to strenge

$$X = x_1 x_2 x_3 \ldots x_m$$
$$Y = y_1 y_2 y_3 \ldots y_n$$

af længde m og n, find en længste fælles delsekvens for dem.

Længden af denne kan ses som et mål for similaritet mellem strenge (f.eks. dna-strenge).

Rekursiv løsning?

Vi vil arbejde på at lave en rekursiv løsning. Vi definerer derfor en størrelse for del-problemer:

- $X_i = x_1 x_2 x_3 \ldots x_i$ for $1 \leq i \leq m$.
- $Y_j = y_1 y_2 y_3 \ldots y_j$ for $1 \leq j \leq n$.
- X_0 og Y_0 er den tomme streng.
- lcs(i, j) er længden af længste fælles delsekvens af X_i og X_j.

Vi vil gerne finde lcs(m, n).

Mere generelt: Vi søger en rekursiv formel for lcs(i, j).

Basistilfælde: Det er klart at lcs(0, j) = lcs(i, 0) = 0.

Optimale delprobleme r I

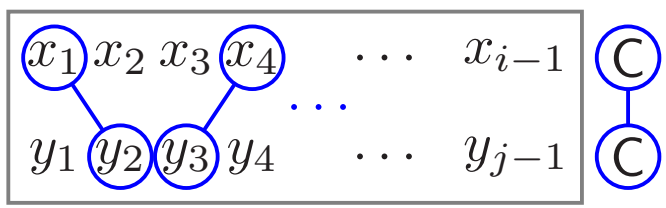
Formel for lcs(i, j): Case I: $x_i = y_j$

Observation: en fælles delsekvens Z for X_i og Y_j af længde k består af

- Et sidste tegn z_k.
- En streng $Z' = z_1 z_2 z_3 \ldots z_{k-1}$ af længde $k-1$, som må være en fælles delsekvens af X_{i-1} og Y_{j-1} (tegnene i Z skal komme i samme rækkefølge som i X og Y, så kun sidste tegn i Z har mulighed for at være x_i og y_j).

Observation (optimale delproblemer) for Case I:

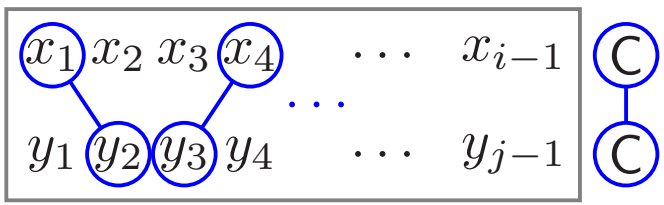
Hvis Z er en længste fælles delsekvens for for X_i og Y_j, må Z' være en længste fælles delsekvens af X_{i-1} og Y_{j-1}. For hvis der fandtes en længere fælles delsekvens for X_{i-1} og Y_{j-1}, kunne den tilføjes tegnet x_i $(= y_j)$ og blive en længere fælles delsekvens for X_i og Y_j.



Optimale delproblemer I

Af observationen haves i Case I ($x_i = y_j$):

- lcs(i, j) = lcs(i − 1, j − 1) + 1
- En længste fælles delsekvens for X_{i-1} og Y_{j-1} tilføjet tegnet x_i (= y_j) er en længste fælles delsekvens for X_i og Y_j.



Optimale delprobleme r II

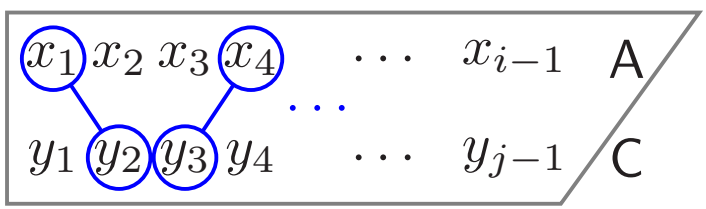
Formel for lcs(i, j): Case II: $x_i \neq y_j$

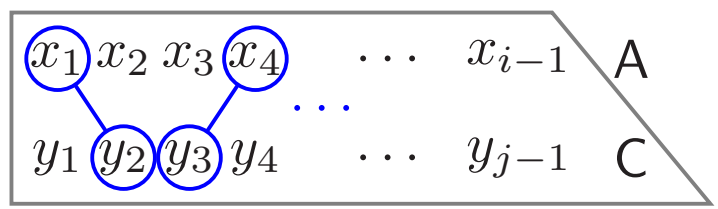
Observation: en fælles delsekvens $Z = z_1 z_2 z_3 \ldots z_k$ for X_i og Y_j kan ikke have z_k værende en parring af x_i og y_j (da disse er forskellige).

Så Z må være en fælles delsekvens for *enten* X_{i-1} og Y_j *eller* for X_i og Y_{j-1} (eller evt. begge).

Observation (optimale delproblemer) for Case II:

> Hvis Z er en længste fælles delsekvens for X_i og Y_j, må den være en længste fælles delsekvens for enten X_{i-1} og Y_j eller for X_i og Y_{j-1} (eller evt. begge). For hvis der fandtes en længere fælles delsekvens for enten X_{i-1} og Y_j eller for X_i og Y_{j-1}, ville denne også være en længere fælles delsekvens for X_i og Y_j.





Optimale delproblemer II

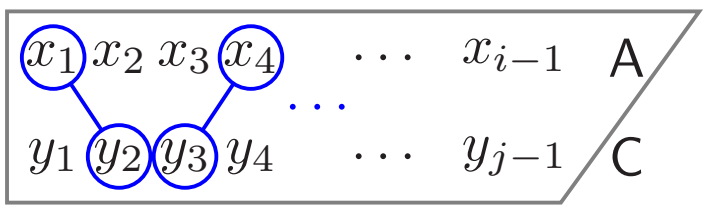
Lad T_1 være en længste fælles delsekvens for X_{i-1} og Y_j, og lad T_2 være en længste fælles delsekvens for X_i og Y_{j-1}.

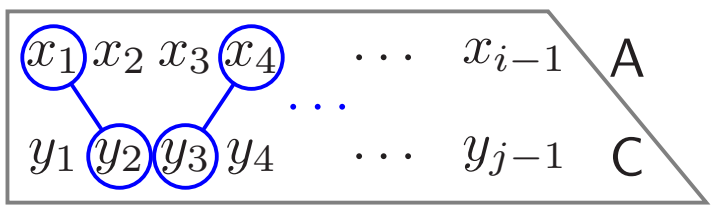
Af observationen i Case II ($x_i \neq y_j$) haves, at blandt T_1 og T_2 er der (mindst) én, som er en længste fælles delsekvens for X_i og Y_j.

Ingen af T_1 og T_2 kan være længere end den længste fælles delsekvens for X_i og Y_j (da de begge er delsekvenser af X_i og Y_j).

Så af observationen haves i Case II ($x_i \neq y_j$):

- $\mathrm{lcs}(i, j) = \max(\mathrm{lcs}(i-1, j), \mathrm{lcs}(i, j-1))$
- Hvis $\mathrm{lcs}(i-1, j) \geq \mathrm{lcs}(i, j-1)$, er en længste fælles delsekvens for X_{i-1} og Y_j også en længste fælles delsekvens for X_i og Y_j. Et symmetrisk udsagn gælder for "$\leq$" og X_i og Y_{j-1}.





Rekursiv formel for lcs(i, j)

Alt i alt har vi fundet flg. rekursive formel for lcs(i, j):

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

Den giver anledning til en naturlig, simpel rekursiv algoritme.

MEN: det er nemt at se at der er gentagelser blandt delproblemers delproblemer.

Så samme delproblemer bliver gentagne gange beregnet forskellige steder i rekursionstræet, og køretiden bliver meget dårlig.

Kan evt. løses med memoization: hav en tabel med plads til svaret på alle de mulige delproblemer lcs(i, j), og gem svaret, når det er beregnet første gang. Siden, slå det bare op.

Dynamisk programmering: udfyld i stedet direkte denne tabel bottom-up på struktureret måde.

Dynamisk programmering

Dynamisk programmering: udfyld tabel over lcs(i, j) bottom-up på struktureret måde.

i \ j	0	1	2	·	·	·	n
0							
1							
2							
·							
·							
m							

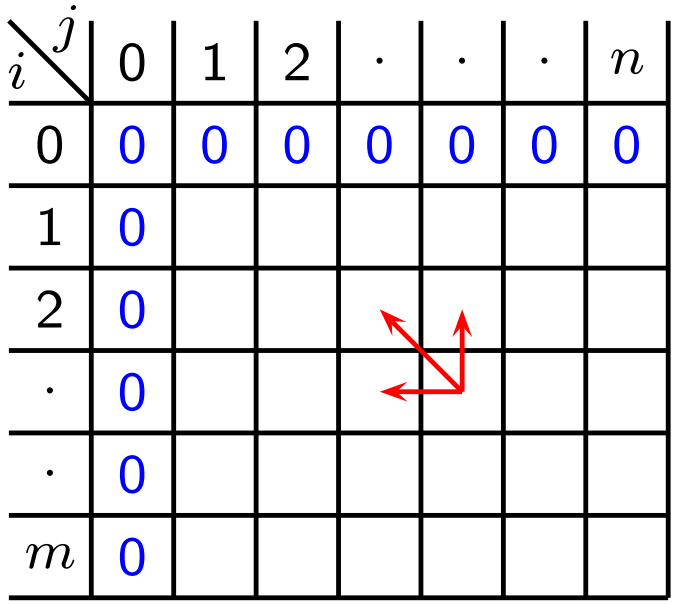
i \ j	0	1	2	·	·	·	n
0	0	0	0	0	0	0	0
1	0						
2	0						
·	0						
·	0						
m	0						

i \ j	0	1	2	·	·
0	0	0	0	0	0
1	0				
2	0				
·	0				
·	0				
m	0				

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

Køretid

Dynamisk programmering: udfyld tabel over lcs(i, j) bottom-up på struktureret måde.



Tabelstørrelse: mn

Udfyld tabelindgang: O(max størrelse af røde graf) $= O(1)$.

Tid i alt: O(produktet af de to) $= O(mn)$.

Find en konkret løsning

lcs(m, n) er længden af en længste fælles delsekvens for $X = X_m$ og $Y = Y_n$.

Hvis vi gerne vil finde en konkret fælles delsekvens af denne længde: Gem for hvert felt i tabellen hvilken af de tre røde pile som gav lcs(i, j)-værdien i dette felt.

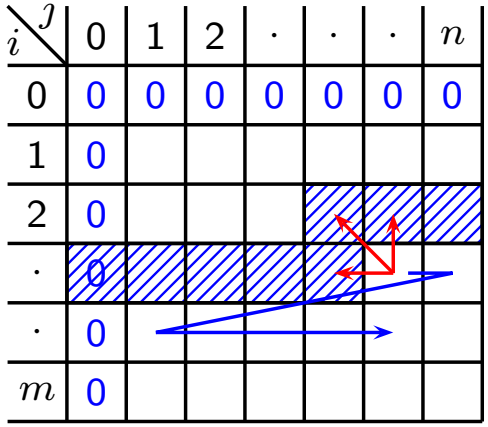
i	j	0	1	2	3	4	5	6
	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Følg gemte pile baglæns fra lcs(m, n). Når en skrå pil følges er det en Case I, og x_i $(=y_j)$ udskrives. Ellers er den en Case II, og intet udskrives.

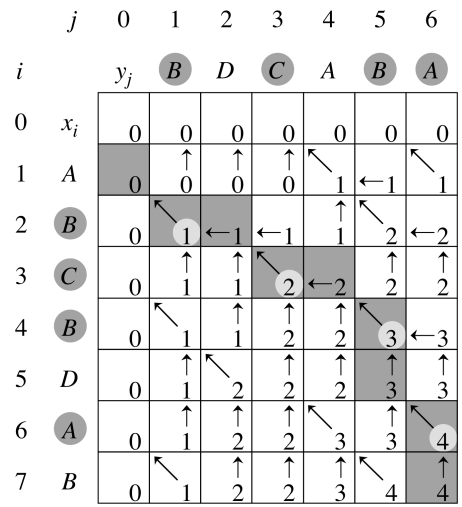
I alt udskrives en længste fælles delsekvens for X og Y i baglæns orden i tid $O(m + n)$.

Pladsforbrug for LCS

Hvis vi kun skal bruge længden af længste fælles delsekvens, kan vi nøjes med $\min\{m, n\}$ plads:



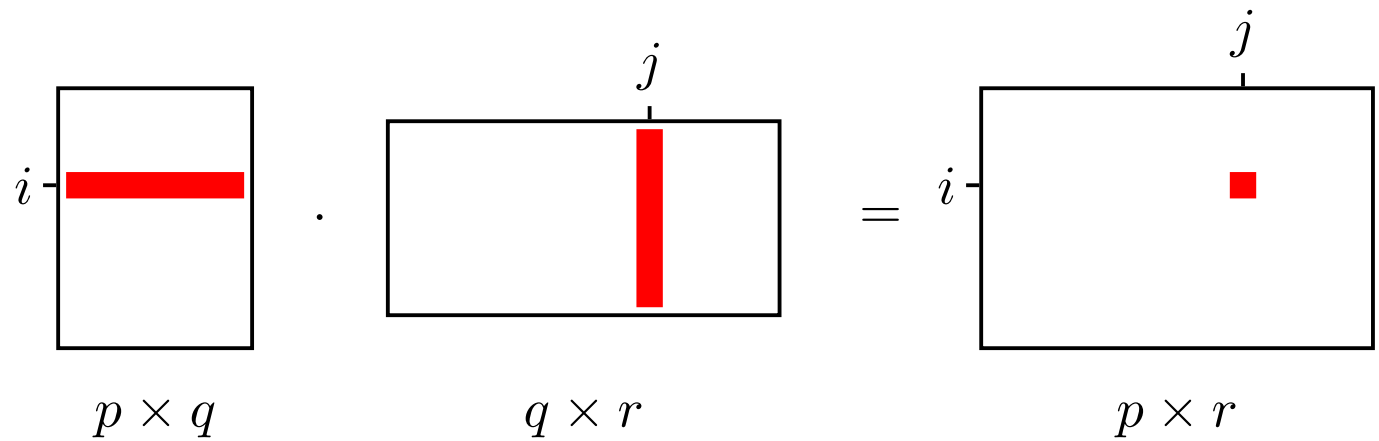
Hvis vi skal bruge en længste fælles delsekvens, må vi gemme hele tabellen, dvs. bruge $\Theta(mn)$ plads (da vi ikke kender stien tilbage, må vi gemme hele tabellen):



[Hirschberg gav i 1975 en metode til også at opnå dette med $\min\{m, n\}$ plads, men det er ikke pensum i DM507.]

Eksempel 2: Multi-Matrix-multiplikation

En $p \times q$ matrix A_1 og en $q \times r$ matrix A_2 kan multipliceres i tid $O(pqr)$. Resultatet er en $p \times r$ matrix.



Matrix-multiplikation er associativ:

$$A_1 \cdot (A_2 \cdot A_3) = (A_1 \cdot A_2) \cdot A_3$$

Multi-Matrix-multiplikation

Matrix-multiplikation er associativ:

$$A_1 \cdot (A_2 \cdot A_3) = (A_1 \cdot A_2) \cdot A_3$$

Men køretiden er IKKE ens. Eksempel:

	A_1	A_2	A_3
	10×100	100×5	5×50
$(A_2 \cdot A_3)$:			100×50
$(A_1 \cdot A_2)$:		10×5	

Tid for $A_1 \cdot (A_2 \cdot A_3)$ er $10 \cdot 100 \cdot 50 + 100 \cdot 5 \cdot 50 = 75.000$

Tid for $(A_1 \cdot A_2) \cdot A_3$ er $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7.500$

Multi-Matrix-multiplikation

Spørgsmålet:

For et produkt af n matricer

$$A_1 \cdot A_2 \cdot A_3 \cdot \ldots \cdot A_{n-1} \cdot A_n$$

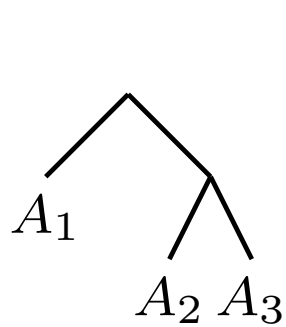
med kombatible dimensioner

A_1	A_2	A_3	$\ldots$	A_{n-1}	A_n
$p_0 \times p_1$	$p_1 \times p_2$	$p_2 \times p_3$	$\ldots$	$p_{n-2} \times p_{n-1}$	$p_{n-1} \times p_n$

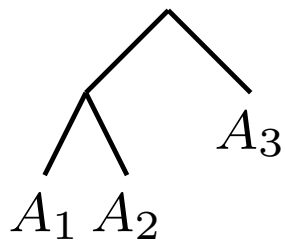
hvad er den billigste rækkefølge at gange dem sammen i?

Beregningstræer

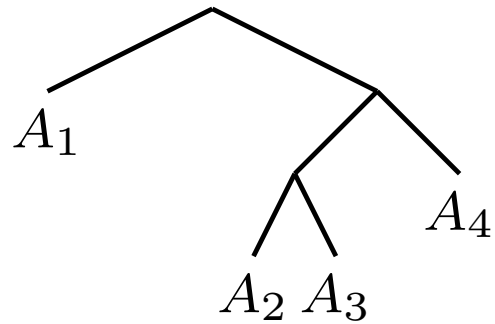
Rækkefølge = parentes-sætning = binært beregningstræ:



$A_1(A_2A_3)$



$(A_1A_2)A_3$



$A_1((A_2A_3)A_4)$

Optimale delprobleme r og rekursiv ligning

Lad $m(i,j)$ være prisen for bedste måde at gange $A_i, \ldots, A_j$ sammen på.

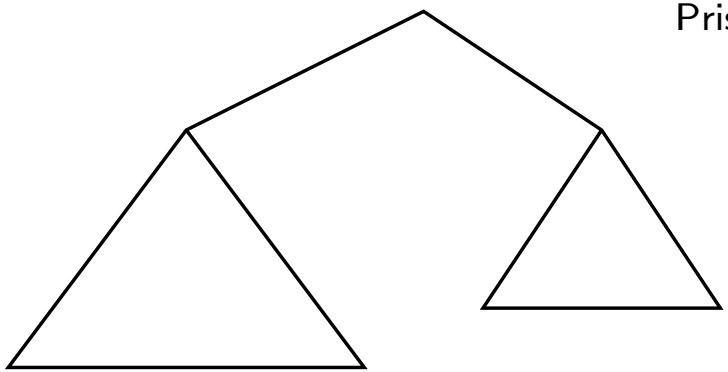
Observation (optimale delprobleme r):

> Undertræerne for roden af et optimalt træ må selv være optimale beregningstræer.

Pris for rod: $p_{i-1}p_k p_j$

Pris for venstre undertræ: $m(i,k)$

Dimension af output af venstre undertræ: $p_{i-1} \times p_k$



Pris for højre undertræ: $m(k+1,j)$

Dimension af output af højre undertræ: $p_k \times p_j$

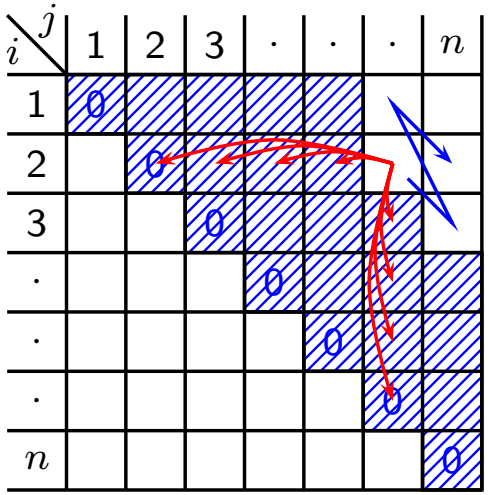
Prøv alle placeringer af rod, dvs. alle split $A_i, \ldots, A_k$ og $A_{k+1}, \ldots, A_j$:

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m(i,k) + m(k+1,j) + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

Tabel

Gentagelser blandt delproblemers delproblemer. Lav tabel og udfyld systematisk. Målet er at kende $m(1, n)$.

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j}\{m(i,k) + m(k+1,j) + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$



Tabelstørrelse: $O(n^2)$.

Udfyld tabelindgang: $O(\text{max størrelse af røde graf}) = O(n)$.

Tid i alt: $O(\text{produktet af de to}) = O(n^3)$.

Find konkret løsning: følg de optimale valg baglæns.

Grådige algoritmer

Grådige algoritmer

Et generelt algoritme-konstruktionsprincip ("paradigme") for kombinatoriske optimeringsproblemer.

Ideen er simpel:

- Opbyg løsningen bid for bid ved hele tiden af vælge, hvad der lige nu ser ud som "bedste valg" (uden at tænke på resten af løsningen).

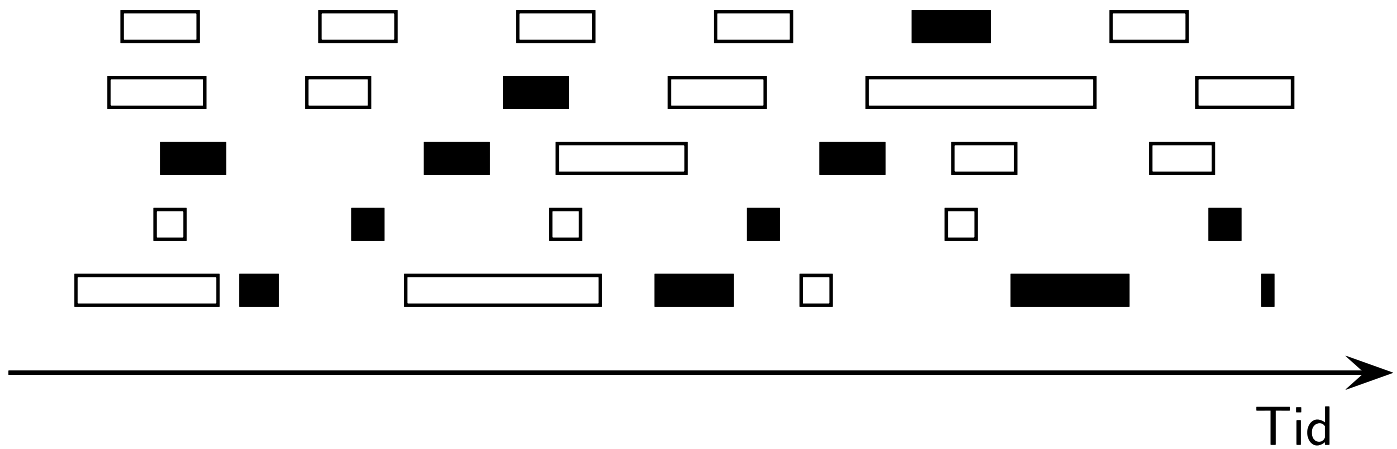
Dvs. man håber på at lokal optimering giver global optimering.

Mere præcist kræver metoden en definition af "bedste valg" (også kaldet "grådig valg"), samt et bevis for at gentagen brug af dette ender med en optimal løsning.

Eksempel: et simpelt skeduleringsproblem

Input: en samling booking-ønsker for en ressource, hver med en starttid og sluttid.

Output: en størst mulig mængde ikke-overlappende bookings.



Her er 12 ikke-overlappende booking-ønsker.

Er 12 det maksimale antal for dette input?

Ja, men det er måske ikke nemt at se. Der er brug for en algoritme til at finde det maksimale antal. Vi forsøger med den grådige metode.

Eksempel: et simpelt skeduleringsproblem

Forslag til grådigt valg: den som slutter først (blandt de tilbageværende uden overlap med allerede valgte). Som kode:

Sorter aktiviteter efter stigende sluttid
For hver aktivitet *a* taget i den rækkefølge:
  If *a* overlapper allerede valgte aktiviteter:
    Skip *a*
  Else
    Vælg *a*

Analyse

Vi vil vise følgende invariant:

> Der findes en optimal løsning OPT som indeholder de af algoritmen indtil nu valgte aktiviteter.

Når algoritmen er færdig, følger korrekthed af ovenstående invariant:

Algoritmens valgte aktiviteter ligger inden i en optimal løsning OPT. Pga. algoritmens virkemåde vil alle ikke-valgte aktiviteter overlappe en af de valgte. Altså kan algoritmens løsning ikke udvides og stadig være en løsning. Specielt kan OPT ikke være større end algoritmens valgte løsning, som derfor er lig med OPT.

Analyse

Bevis for invariant ved induktion:

Basis:
Klart før første iteration af **for**-løkken (da ingen aktiviteter er valgt).

Skridt:
Lad OPT være den optimale løsning fra induktionsudsagnet før iterationen. Vi skal vise at der findes en optimal løsning OPT' i induktionsudsagnet efter iterationen.

If-case: Her vælger algoritmen intet nyt, så OPT kan bruges som OPT'.

Analyse

Else-case:

Se på aktiviteterne sorteret efter stigende sluttider: $a_1, a_2, \ldots, a_n$. Lad a_i være algoritmens senest valgte aktivitet og lad a_j være aktiviteten, som vælges i denne iteration.

Pga. invarianten indeholder OPT a_i. I OPT kan a_i ikke være den sidste (for så kunne OPT udvides med a_j). Lad a_k være den næste i OPT efter a_i. Da a_j er den første aktivitet (i ovenstående sortering) efter a_i som ikke overlapper a_i, må $j \leq k$.

Hvis $j = k$ kan OPT bruges som OPT'. Ellers skifter vi a_j i OPT ud med a_k. Dette giver ikke overlap med andre aktiviteter i OPT (de stopper enten før a_i eller stopper efter a_k - i sidste tilfælde må de starte efter a_k og dermed efter a_j) og bevarer størrelsen. Vi har derfor efter udskiftningen en ny optimal løsning OPT' som opfylder invarianten.

[NB: I første iteration findes a_i ikke, men vi kan sætte $j = 1$ og sige noget tilsvarende.]

Køretid: Sortering + $O(n)$.

Rygsæksproblemet

Rygsæk som kan bære W kg.

Ting med værdi og vægt.

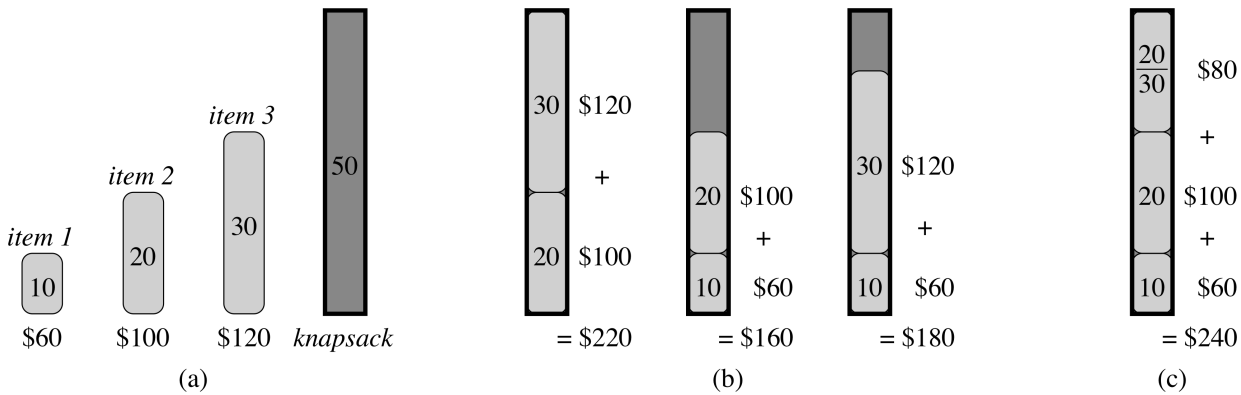
Ting nr. i	1	2	3	4	5	6	7
Vægt w_i	4	6	2	15	7	4	5
Værdi v_i	45	32	12	50	23	9	15

Mål: tag mest mulig værdi med uden at overskride vægtgrænsen.

Rygsæksproblemet

"Fractional" version af problemet (dele af ting kan medtages i rygsækken) kan løses med en grådig algoritme: vælg tingene efter aftagende "nytte" = værdi/vægt. Et simplet udskiftningsargument viser, at den optimale løsning kun kan være som den af algoritmen valgte.

NB: Denne grådige algoritme virker IKKE for 0-1 versionen af problemet (hvor kun hele ting kan medtages):



Eksempel på at "grådige valg" ikke bare kan antages at virke for alle problemer (lokal optimering giver ikke altid global optimering).

Bitmønstre

$$\boxed{01101011}000110010101011011\ldots$$

Bitmønstre skal fortolkes for at have en betydning:

- Bogstaver
- Tal (heltal, kommatal)
- Computerinstruktion (program)
- Pixels (billedfil)
- Amplitude (lydfil)
- $\vdots$

Fokus i dag: bogstaver (og andre tegn).

Repræsentation af tegn

En klassisk repræsentation: ASCII.

$$
\begin{array}{ll}
\vdots & \\
\texttt{a}: & 1100001 \\
\texttt{b}: & 1100010 \\
\texttt{c}: & 1100011 \\
\texttt{d}: & 1100100 \\
\vdots &
\end{array}
$$

Alle tegn fylder 7 bits (fixed-width codes).

Huffman-koder

Er fixed-width kodning den kortest mulige repræsentation af en fil af tegn?

Det kommer an på filens indhold! Eksempel:

	`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

Fixed-width version:

$$3 \cdot (45.000 + 13.000 + \cdots + 5.000) = 300.000 \text{ bits}$$

Variable-width version:

$$1 \cdot 45.000 + 3 \cdot 13.000 + \cdots + 4 \cdot 5.000 = 224.000 \text{ bits}$$

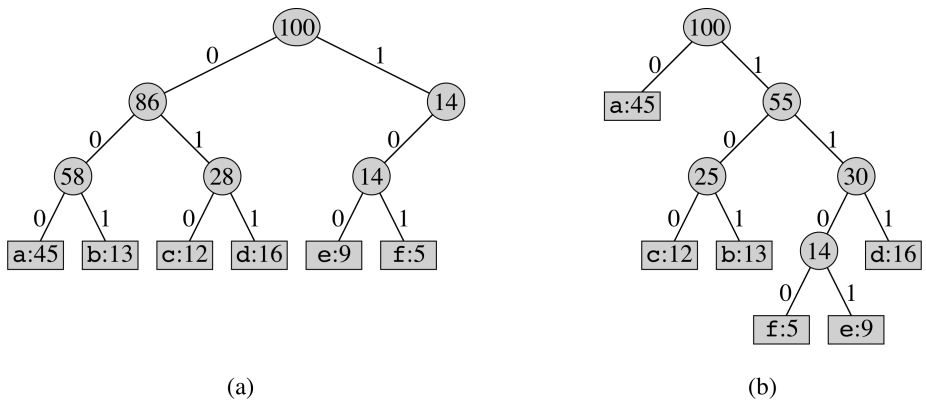
Ønske: kort(est mulig?) repræsentation af en fil. Sparer plads på disk, tid på transport over netværk.

Prefix-kode = træer

Kodeord = sti i binært træ: 0 ∼ gå til venstre, 1 ∼ gå til højre

Prefix-fri kode: ingen kode for et tegn er starten (prefix) af koden for et andet tegn (⇒ dekodning utvetydig). Så tegn svarer til knuder med nul børn (blade).

	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



(a) (b)

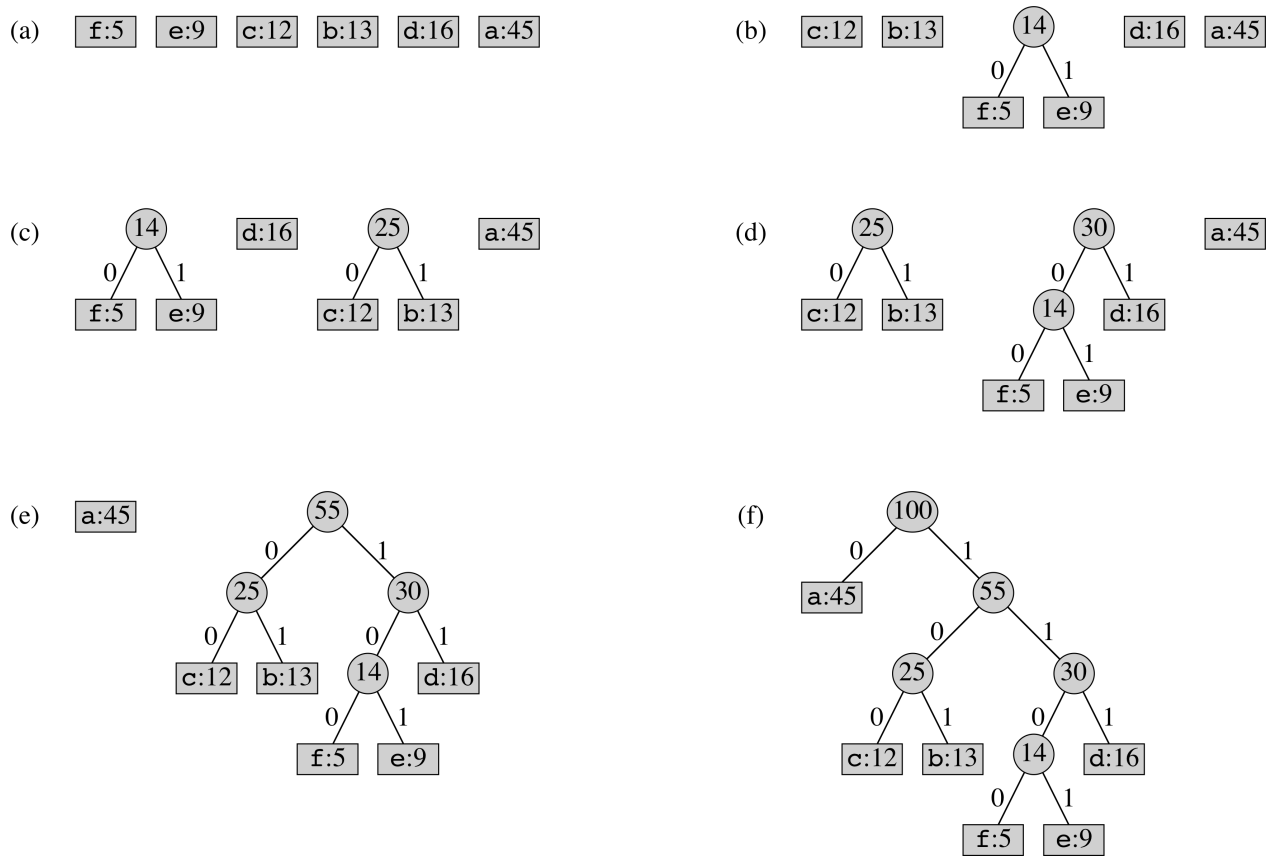
For en givet fil (tegn og deres frekvenser), find bedste variable-width prefix-kode. Dvs. for Cost(tree) = |kodet fil|, find træ med lavest cost.

Optimale træer kan ikke have knuder med kun ét barn (alle tegn i undertræet for en sådan knude kan forkortes med en bit, jvf. (a) ovenfor). Så kun knuder med to eller nul børn findes.

Huffmans algoritme

[David Huffman, 1952]

Byg op nedefra (fra mindste til største frekvenser) ved hele tiden at lave flg. "grådige valg": slå de to deltræer med de to mindste samlede frekvenser sammen:



Køretid

Givet en tabel med n tegn og deres frekvenser laver Huffmans algoritme

- $n - 1$ iterationer.

(Der er n træer til start, ét træ til slut, og hver iteration mindsker antallet med præcis én.)

Ved at bruge en (min-)prioritetskø, f.eks. implementeret ved en heap, kan hver iteration udføres med:

- to ExtractMin-operationer
- én Insert-operation
- $O(1)$ andet arbejde.

Hver prioritetskø-operation tager $O(\log n)$ tid.

Så samlet køretid for de n iterationer er $O(n \log n)$.

Korrekthed

Resumé:

Huffmans algoritme vedligeholder en samling træer F. Vægten af et træ er summen af hyppigheden i dets blade. I hvert skridt slår Huffman to træer med mindste vægte sammen, indtil der kun er ét træ.

Vi vil bevise følgende invariant:

> Træerne i F kan slås sammen til et optimalt træ.

Når algoritmen stopper, indeholder F kun ét træ, som derfor ifølge invarianten må være et optimalt træ.

Korrekthed

Vi vil bevise følgende invariant:

> Træerne i F kan slås sammen til et optimalt træ.

Beviset er via induktion over antal skridt i algoritmen.

Basis: Ingen skridt er taget. Da består F af n træer, som hvert kun er et blad. Disse er netop bladene i ethvert optimalt træ, så invarianten er oplagt opfyldt.

Induktionsskridt: antag invarianten er opfyldt før et skridt i algoritmen, og lad os vise at den er opfyldt efter skridtet.

Korrekthed

Lad træerne i F (før skridtet) være

$$t_1, t_2, t_3, \ldots, t_k.$$

hvor algoritmen slår t_1 og t_2 sammen. Dvs. at med hensyn til vægt gælder følgende for træerne:

$$t_1 \leq t_2 \leq t_3 \leq \cdots \leq t_k,$$

Ifølge induktionsantagelsen kan træerne slås sammen til et optimalt træ. Lad T være toppen af dette træ—dvs. et træ, hvis blade er rødderne i $t_1, t_2, t_3, \ldots, t_k$.

Korrekthed

Case 1: Rødderne i t_1 og t_2 er søskende i T.

Her kan den nye samling træer (efter skridtet, hvor t_1 og t_2 slåes sammen) stadig bygges sammen til det samme optimale træ nævnt i invarianten før skridtet.

Så invarianten gælder igen efter skridtet.

Korrekthed

Case 2: Rødderne i t_1 og t_2 er *ikke* søskende i T.

Her vil vi finde et andet top-træ T' (dvs. en anden sammenlægning af træerne i F), som også giver et optimalt træ, og hvor t_1 og t_2 er søskende. For dette optimale træ er vi i Case 1 og dermed færdige.

Vi finder T' ud fra T således:

Se på et blad i T af størst dybde. Da $k \geq 2$ (ellers var Huffman algoritmen færdig), har bladet en forælder. Denne forælder har mindst ét undertræ, altså har den to (i optimale træer har ingen knuder ét undertræ, som bemærket tidligere). Dens andet undertræ må være et blad, ellers er det første blad ikke et dybeste blad.

Altså findes to blade i T som er søskende og begge er af størst dybde. Lad disse indeholde rødderne af t_i og t_j, med $i < j$.

Korrekthed

Mulige situationer:

1	2	3 . . .
i	j	
i		j
	i	j
		i, j

Handling som giver T' fra T:

Ingen (da vi er i Case 1)

Byt t_2 og t_j

Byt t_1 og t_j

Byt t_1 og t_i, samt t_2 og t_j

For et byt af t_1 og t_i gælder, at eftersom t_i's rod mindst har samme dybde som t_1's rod, og den samlede frekvens af t_i er mindst lige så stor som den samlede frekvens af t_1, vil der være flere tegn i filen, som får kortere kodeord, end der er tegn som får længere kodeord. Desuden er forandringerne i længde af kodeord den samme for både forlængelse og forkortelse (nemlig forskellen i dybde mellem t_1 og t_i).

Så den kodede fils længde stiger ikke ved byt af t_1 og t_i, dvs. træet kan ikke blive dårlige ved byt af t_1 og t_i.

Tilsvarende kan vises for et byt af t_1 og t_j, og for et byt af t_2 og t_j.

Da træet var optimalt før byt, er det også efter. Og t_1 og t_2 er nu søskende. □

Disjoint Sets

Partition

En partition (dansk: disjunkt opdeling) af en mængde S er en samling ikke-tomme delmængder A_i, $i = 1, \ldots, k$, som er disjunkte og tilsammen udgør S:

$$A_i \neq \emptyset \text{ for alle } i$$

$$A_i \cap A_j = \emptyset \text{ for } i \neq j$$

$$A_1 \cup A_2 \cup \cdots \cup A_k = S$$

Eksempel:

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

Datastrukturen Disjoint Sets

Disjunkte opdeling som datastruktur? Følgende samling operationer har vist sig relevante i mange anvendelser (herunder nogle senere i kurset):

MAKE-SET(x):

Opret $\{x\}$ som en ny mængde (x må ikke være element i andre mængder).

UNION(x, y):

Slå $\{a, b, c, \ldots, x\}$ og $\{h, i, j, \ldots, y\}$ sammen til $\{a, b, c, \ldots, x, h, i, j, \ldots, y\}$.

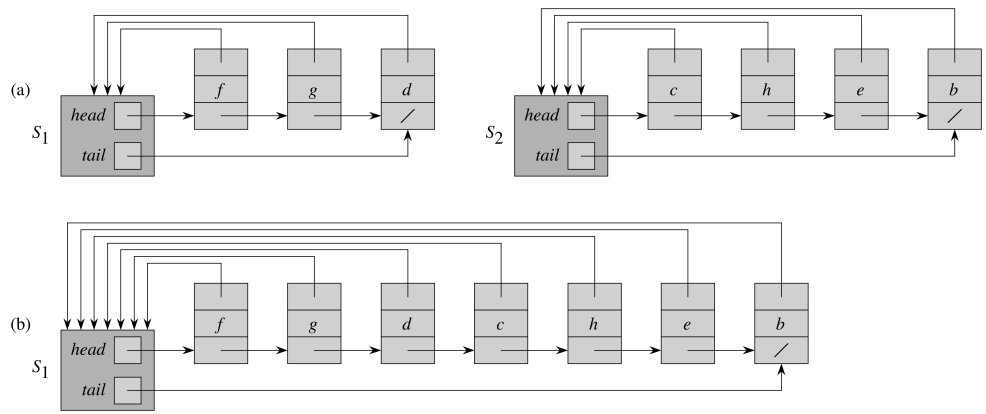
FIND-SET(x):

Returner (en ID for) mængden indeholdende x.

NB: Vi har ingen krav til ID for mængder. Den skal blot være ens for alle x i samme mængde, således at vi kan checke om to elementer x og y ligger i samme mængde.

Disjoint Sets implementeret via lænkede lister

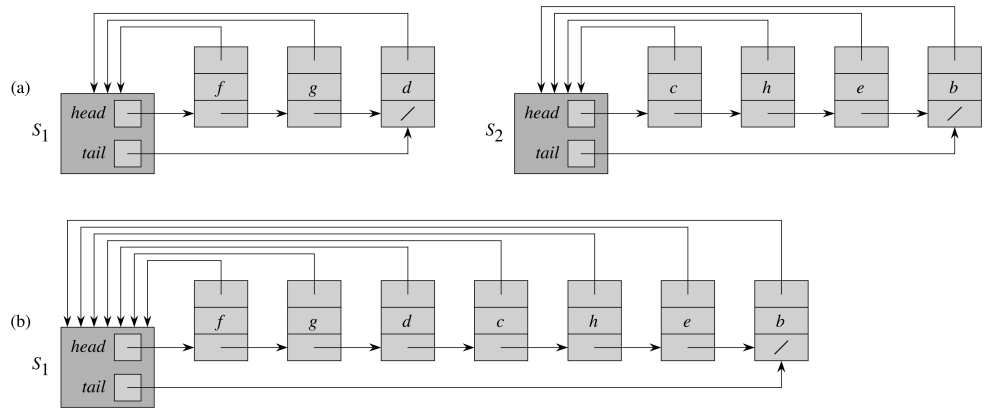
Hver mængde er en lænket liste af elementer, ID for mængde er første element i listen:



- FIND-SET(x): returner (via header-pointer) første element i listen.
- MAKE-SET(x): opret ny liste.
- UNION(x, y): slå lister sammen, behold én header, ændrer alle header-pointere i den anden liste.

Disjoint Sets implementeret via lænkede lister

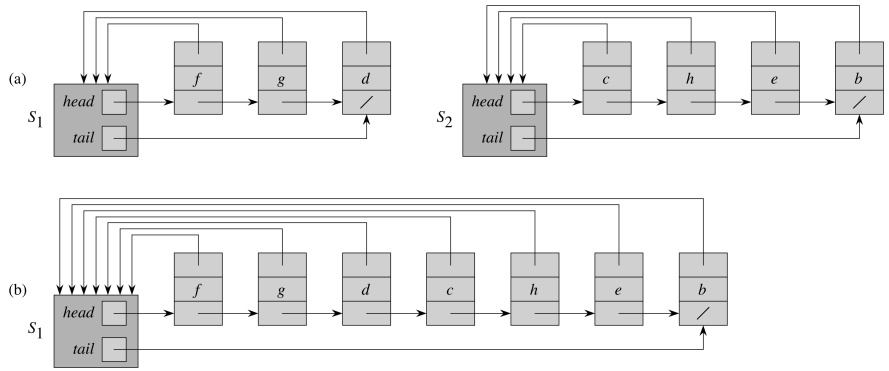
Køretid (n er antal elementer, dvs. antal MAKE-SETs udført)?



- FIND-SET(x): returner (via header-pointer) første element i listen: $O(1)$.
- 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: $O(n)$.

Naiv analyse: n MAKE-SET, op til $n-1$ UNION, og m FIND-SET koster $O(m+n^2)$.

Disjoint Sets implementeret via lænkede lister, version 2



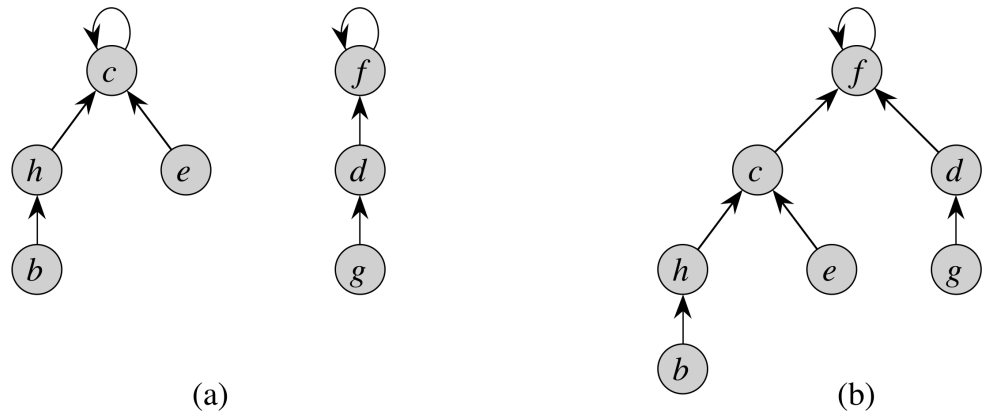
- FIND-SET(x): returner (via header-pointer) første element i listen: $O(1)$.
- MAKE-SET(x): opret ny liste: $O(1)$.
- UNION(x, y): slå lister sammen, behold header af længste liste, ændre alle header-pointere i korteste liste: $O(n)$.

Observér at nu gælder: en knude kan kun ændre sin header-pointer log n gange, da størrelsen af dens mængde hver gang vokser mindst en faktor to ($1 \cdot 2^k \leq n \Leftrightarrow k \leq \log n$).

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

Disjoint Sets implementeret via træer

Hver mængde er et træ med elementer i knuder, rod er ID for mængde:

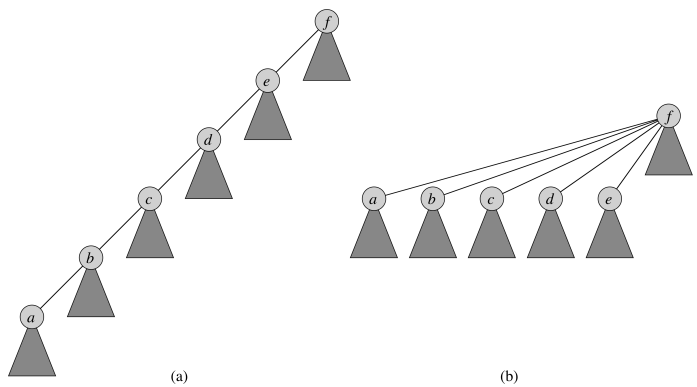


- FIND-SET(x): gå til rod.
- MAKE-SET(x): opret nyt træ.
- UNION(x, y): gør rod af ét træ til barn af andet træ.

Disjoint Sets implementeret via træer, version 2

Union by rank: Tilføj et heltal (rank) til alle rødder. Sættes til 0 ved nye træer (under $\textsc{Make-Set}(x)$). Kontrollerer forældre-barn beslutningen ved $\textsc{Union}(x, y)$: rod med størst rank får den anden rod som barn. Ved ens rank, lad y få x som barn og øg y's rank med 1.

Path compression: Under $\textsc{Find-Set}(x)$, sæt alle passerede knuder op som børn af roden. For $\textsc{Find-Set}(a)$:



Union by rank og path compression $\Rightarrow$ meget tæt på $O(m + n)$ tid. Mere præcist $O(m \cdot \alpha(n) + n)$, hvor $\alpha(n)$ er en meget langsomt voksende funktion.

Funktionen $\alpha(n)$ samt beviset for køretiden findes i afsnit 19.4, som ikke er pensum (gennemgås i et senere kursus på datalogistudiet).

Disjoint Sets implementeret via træer, version 2

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

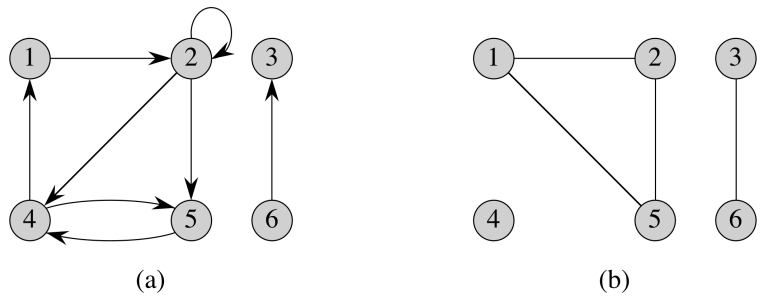
```
MAKE-SET(x)
  x.p = x
  x.rank = 0

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

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
```

Grafer og graf-gennemløb

Grafer

En mængde V af *knuder* (vertices).

En mængde $E \in V \times V$ af *kanter* (edges). Dvs. par af knuder.



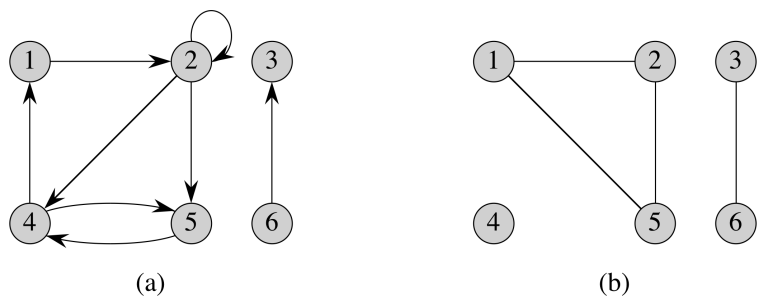
- Orienterede grafer: kanter er ordnede par.
- Uorienterede grafer: kanter er uordnede par.
- Vægtede grafer: hver kant har et tal tilknyttet.
- Notation: $n = |V|$, $m = |E|$.
- Bemærk at $0 \leq m \leq n^2$ for orienterede grafer og $0 \leq m \leq (n^2 + n)/2$ for uorienterede grafer.

Læs yderligere om graf-terminologi i de to første sider af appendix B i lærebogen.

Grafer

Modeller for mange ting:

- Ledningsnet (telefon, strøm, olie, vand,...).
- Vejnet (vejkryds er knuder, veje mellem kryds er kanter)
- Venner på SoMe.
- Følgere på SoMe.
- WWW-sider.
- Medforfatterskaber.

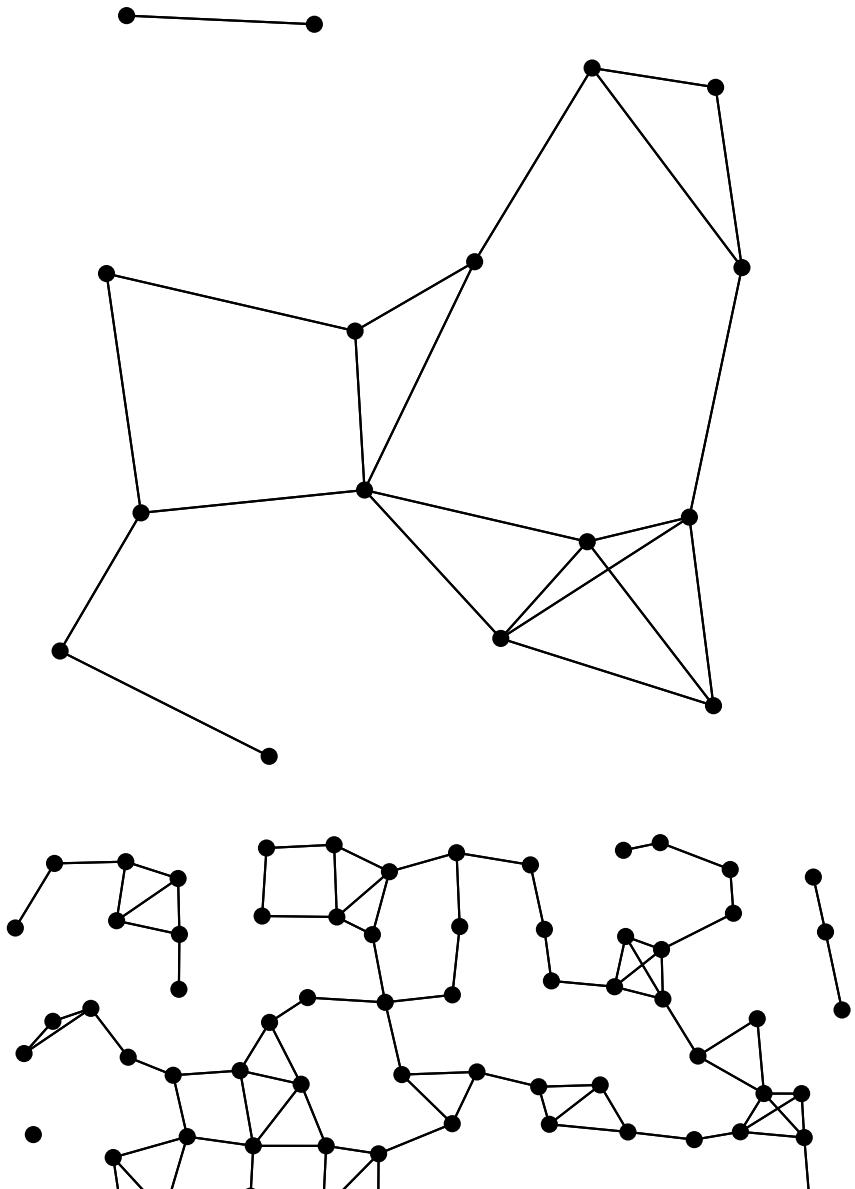


Masser af algoritmiske spørgsmål på grafer

- Hvordan repræsentere grafer på en computer (datastruktur)?
- Findes der en sti mellem to angivne knuder?
- Hvad er en korteste sti mellem to angivne knuder?
- Hvad er en mindste delmængde af kanter, som stadig holder alle knuder forbundet?
- Hvad er en største samling kanter, hvor ingen kanter har fælles knuder?
- $\vdots$

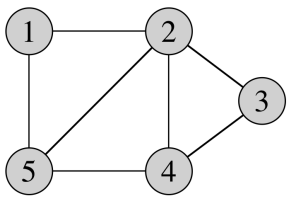
Eksempel på algoritmisk spørgsmål

Afgør om der findes en sti mellem to givne knuder.

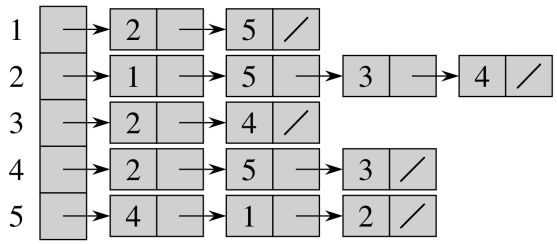


Datastrukturer for grafer

Adjacency lists og adjacency matrix



(a)



(b)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Adjacency lists: listen for u indeholder v for alle kanter $(u, v) \in E$.
Knuder er repræsenteret som heltal mellem 1 og n (eller 0 og $n - 1$).

Plads: $O(n + m)$ for adjacency lists, $O(n^2)$ for adjacency matrix.

Hvis ikke andet oplyses, bruges adjacency lists repræsentationen i algoritmer i dette kursus.

En kant i en uorienterede graf repræsenteres som to orienterede kanter (så mht. implementation er uorienterede grafer bare et specialtilfælde af orienterede grafer).

Grafgennemløb

Opgave: givet en graf i adjacency lists repræsentation, besøg alle knuder og kanter. Målet er, at lære om forskellige egenskaber for grafen.

Generel idé: Besøg en startknude s. Brug kanter i nabolisterne for besøgte knuder til at besøge flere knuder.

Marker knuder undervejs for at holde styr på processen:

- Hvide knuder: endnu ikke besøgt
- Grå knuder: besøgt, men ikke alle kanter i naboliste brugt
- Sorte knuder: besøgt, alle kanter i naboliste brugt

Grafgennemløb

Generisk algoritme til grafgennemløb:

```
GenericGraphTraversal1(s)
    Gør s grå og resten af knuderne hvide
    while der findes grå knuder:
        vælg en grå knude v
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste
            if u hvid:
                gør u grå
```

En knudes livs-cyklus: hvid $\rightarrow$ grå $\rightarrow$ sort. Når algoritmen stopper, er alle knuder enten hvide eller sorte.

Grafgennemløb

Vi skal senere i kurset møde tre varianter, som har forskellige strategier for at vælge næste kant (v, u) at bruge, dvs. for valgene (*):

```
GenericGraphTraversal1(s)
    Gør s grå og resten af knuderne hvide
    while der findes grå knuder:
        vælg en grå knude v (*)
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste (*)
            if u hvid:
                gør u grå
```

- Breadth-First-Search (BFS)
- Depth-First-Search (DFS)
- Priority-Search (Dijkstras algoritme, A*)

Hvor langt når vi rundt i grafen?

Vi når alt, som kan nås fra s:

Sætning: Hvis der er en sti fra s til v, vil v være sort (og dermed besøgt) når $\textsc{GenericGraphTraversal1}(s)$ stopper.

Bevis: Når algoritmen stopper, er alle knuder enten hvide eller sorte. Da s startede grå, må den nu være sort. Antages at v er hvid, må der være mindst én kant (u, w) på stien med u sort og w hvid. Men u kan kun være sort hvis (u, w) er blevet brugt, hvorved w blev grå og nu må være sort. Så antagelsen kan ikke gælde og v må være sort.

For at nå rundt i *hele* grafen:

```
GenericGraphTraversalGlobal()
    Gør alle knuder hvide
    for alle knuder s:
        if s hvid:
            GenericGraphTraversal2(s)
```

```
GenericGraphTraversal2(s)
    Gør s grå ~~og resten af knuderne hvide~~
    while der findes grå knuder:
        vælg en grå knude v (*)
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste (*)
            if u hvid:
                gør u grå
```

Hvis (*) tager tid $O(1)$, er samlet køretid $O(n+m)$. [En kant kan kun vælges én gang, så alt arbejde udført i **else**-del tager $O(m)$ tid i alt. Resten tager $O(n)$ tid i alt.]

Hvor langt når vi rundt i grafen per kald?

Sætning: Hvis der ved starten af et kald til $\textsc{GenericGraphTraversal2}(s)$ er en sti fra s til v bestående af hvide knuder (inkl. v), vil v være sort (og dermed besøgt) når $\textsc{GenericGraphTraversal2}(s)$ stopper.

Bevis: Det samme som før for $\textsc{GenericGraphTraversal1}(s)$.

Husk hvem der opdagede hvem:

Når en knude u ($\neq s$) besøges første gang, gemmer den i variablen $u.\pi$ knuden, som opdagede u (predecessor). Bemærk, at $u.\pi$ højst bliver sat én gang (efter initialisering til NIL), da u gøres grå samtidig.

```
GenericGraphTraversalGlobalWithParents()
    Gør alle knuder hvide og sæt deres π til NIL
    for alle knuder s:
        if s hvid:
                GenericGraphTraversal3(s)
```

```
GenericGraphTraversal3(s)
    Gør s grå
    while der findes grå knuder:
        vælg en grå knude v (*)
        if v's naboliste er brugt op
            gør v sort
        else
            vælg en ubrugt kant (v, u) fra v's naboliste (*)
            if u hvid:
                gør u grå
                sæt u.π lig v
```

Husk hvem der opdagede hvem:

Sætning: De knuder, som er opdaget (gjort ikke-hvide) i et kald GenericGraphTraversal3(s), udgør et træ med s som rod og π i opdagede knuder som parent pointers. For hver sti fra en knude v til roden i træet findes den samme sti i grafen, men i modsat retning (fra s til v).

Bevis: Det er nemt at se, at dette udsagn er en invariant som vedligeholdes under kørslen af GenericGraphTraversal3(s).

Bemærk, at i GenericGraphTraversalGlobalWithParents() kaldes GenericGraphTraversal3(s) gentagne gange. Hvert kald giver ét træ. Træerne fra forskellige kald deler ikke knuder, og tilsammen indeholder de alle knuder i grafen.

Bredde-Først-Søgning (BFS)

Strategi: Hold de grå knuder i en KØ, brug nabolister op med det samme.

Tilføj også en variabel $v.d$ til alle knuder v (d for distance.)

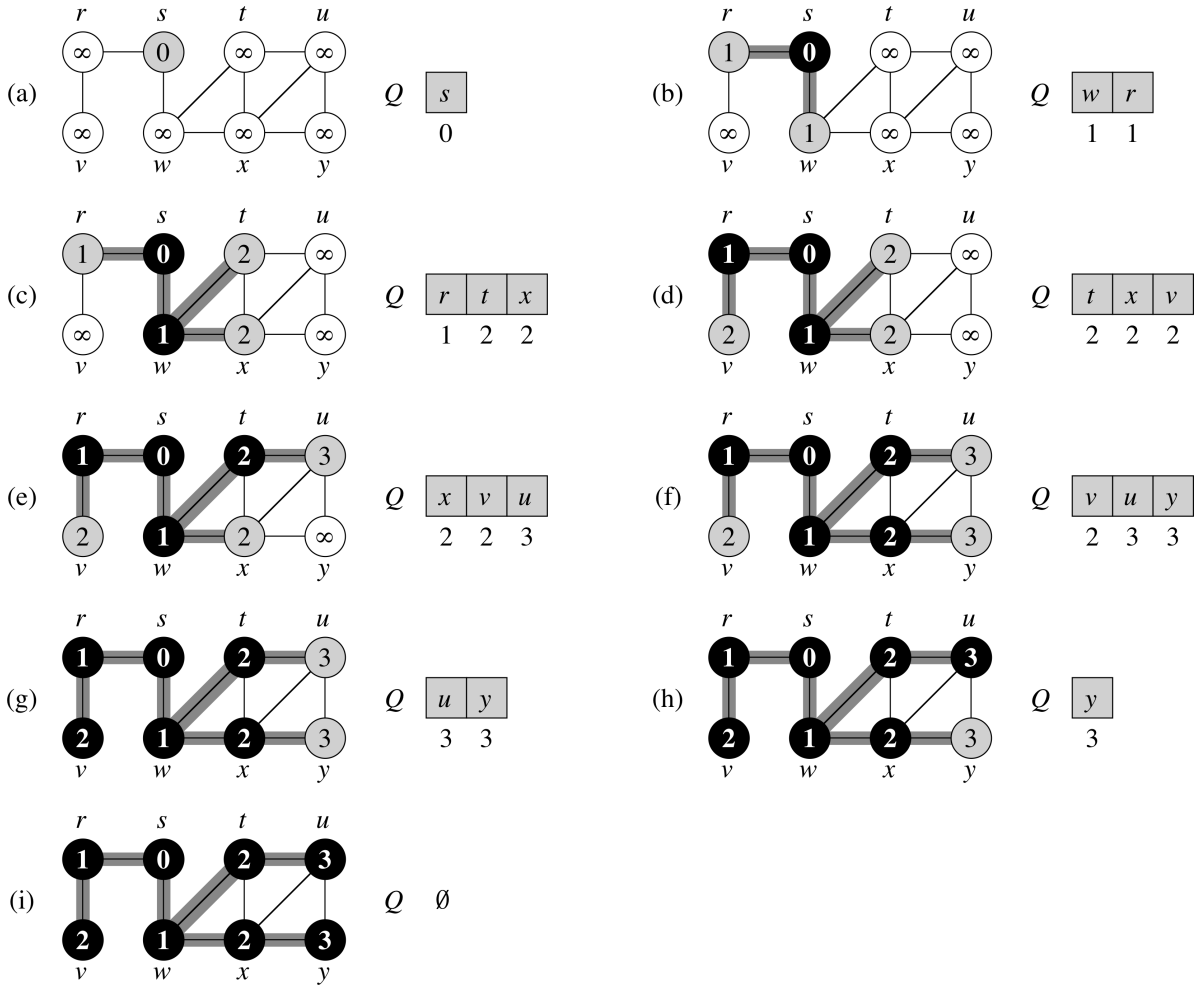
Mest brugt er versionen uden GLOBAL-del (for BFS er vi ofte mere interesserede i ét bestemt s fremfor at komme hele grafen rundt):

```
BFS(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  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
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                 ENQUEUE(Q, v)
18       u.color = BLACK
```

Invariant:
kø = alle grå knuder.

Bredde-Først-Søgning (BFS)

Eksempel:



Bredde-Først-Søgning (BFS)

For BFS kan sætningen om $\textsc{GenericGraphTraversal3}(s)$ udvides til også at sige noget om værdierne af $v.d$:

Sætning: De knuder, som er opdaget (gjort ikke-hvide) i et kald $\textsc{GenericGraphTraversal3}(s)$, udgør et træ med s som rod og π i opdagede knuder som parent pointers. For hver sti fra en knude v til roden i træet findes den samme sti i grafen, men i modsat retning (fra s til v) og $v.d$ er lig antal kanter på denne sti.

Bevis: Det er nemt at se, at dette udsagn er en invariant som vedligeholdes under kørslen af $\mathrm{BFS}(G, s)$.

Bemærk, at $v.d$ højst bliver sat én gang (efter initialisering til $-\infty$): $v.d$ sættes kun, når v er hvid, og v gøres ikke-hvid samtidig med at $v.d$ sættes.

Egenskaber for BFS

Køretid: $O(n+m)$.

Beviset er det samme som under GenericGraphTraversalGlobal, da valgene (*) i BFS tager $O(1)$ tid. I BFS bruger man som sagt ofte kun at kalde på én startknude s, dvs. uden at bruge Global-delen. Men køretiden kan kun falde ved dette.

Definition: Vi definerer $\delta(s,v)$ som længden af en korteste sti, *målt i antal kanter*, fra startknuden s til knuden v. Findes ingen sti, defineres $\delta(s,v) = \infty$.

> Sætning: Når BFS stopper, gælder $v.d = \delta(s,v)$ for alle knuder.

Dvs. BFS kan finde korteste veje (målt i antal kanter) fra s til alle v.

Bevis for sætning

De mulige værdier for $\delta(s, v)$ er $0, 1, 2, 3, \ldots$ samt ∞.

For knuder v med $\delta(s, v) = \infty$ findes der ikke en sti fra s til v. Så kan v ikke være opdaget (som vist tidligere er der en sti i grafen fra s til alle opdagede knuder). Derfor kan værdien $v.d = \infty$ sat under initialisering ikke ændres, så når BFS stopper, gælder $v.d = \delta(s, v)$ for disse knuder.

For resten af knuderne er $\delta(s, v) = i < \infty$. For dem viser vi, via induktion på i, at

$$\delta(s, v) = i$$

$$\Downarrow$$

$v.d = i$ når BFS stopper

Tilsammen giver dette sætningen.

Observationer

1. Pga. virkemåden for en kø vil BFS-algoritmen for $i = 0, 1, 2, 3, \ldots$ udtage alle knuder med d-værdi lig i mens den indsætter alle knuder med d-værdi lig $i + 1$ (og derefter fortsætter den med næste værdi for i).

   Heraf ses, at d-værdierne for de udtagne knuder stiger monotont.
2. Vi ved allerede at $\delta(s, v) \leq v.d$, eftersom vi tidligere har vist, at der er en sti af længde $v.d$ i grafen, når $v.d < \infty$ (dvs. når v er ikke-hvid).

Induktionsbevis

Basis ($i = 0$): Hvis $\delta(s, v) = 0$ er $v = s$. BSF sætter $s.d = 0$.

Induktionssskridt ($i > 0$): Vi antager, at $\delta(s, v) = i - 1 \Rightarrow v.d = i - 1$ er sandt, og skal vise at $\delta(s, v) = i \Rightarrow v.d = i$ er sandt.

Hvis $\delta(s, v) = i$, eksisterer en sti fra s til v af længde i. For næstsidste knude u på denne sti gælder $\delta(s, u) = i - 1$ (hvis u havde en kortere vej, ville v også have det).

Fra induktionsantagelsen har vi $u.d = \delta(s, u)$. Da u blev taget ud af køen, var v (en nabo til u) *enten* uopdaget (hvid) og bliver nu opdaget af u, *eller* v var allerede opdaget fra en knude t, som derfor allerede var taget ud af køen og derfor (via observation 1) har $t.d \leq u.d$.

I BFS tildeles v en d-værdi, som er én større end d-værdien af knuden, som opdager den.

Så hvad enten v bliver optaget af u eller v, bliver $v.d$ derfor sat til *højst* $u.d + 1 = \delta(s, u) + 1 = (i - 1) + 1 = i = \delta(s, v)$. Vi ved (via observation 2) at $v.d$ er *mindst* $\delta(s, v)$. I alt har vi $v.d = \delta(s, v)$.

Dybde-Først-Søgning (DFS)

Strategi: Hold de grå knuder i en STAK, avancér minimalt i nabolisten hver gang vi kigger på en knude.

Stakken er implicit i den rekursive formulering nedenfor (dvs. er lig rekursionsstakken), men kan også kodes eksplicit. Mere præcist: elementerne på stakken er de grå knuder, hver med en delvist gennemløbet naboliste, nemlig gennemløbet i for-løkken i DFS-VISIT. [Bemærk: koden til venstre svarer til GLOBAL-delen i terminologien fra tidligere.]

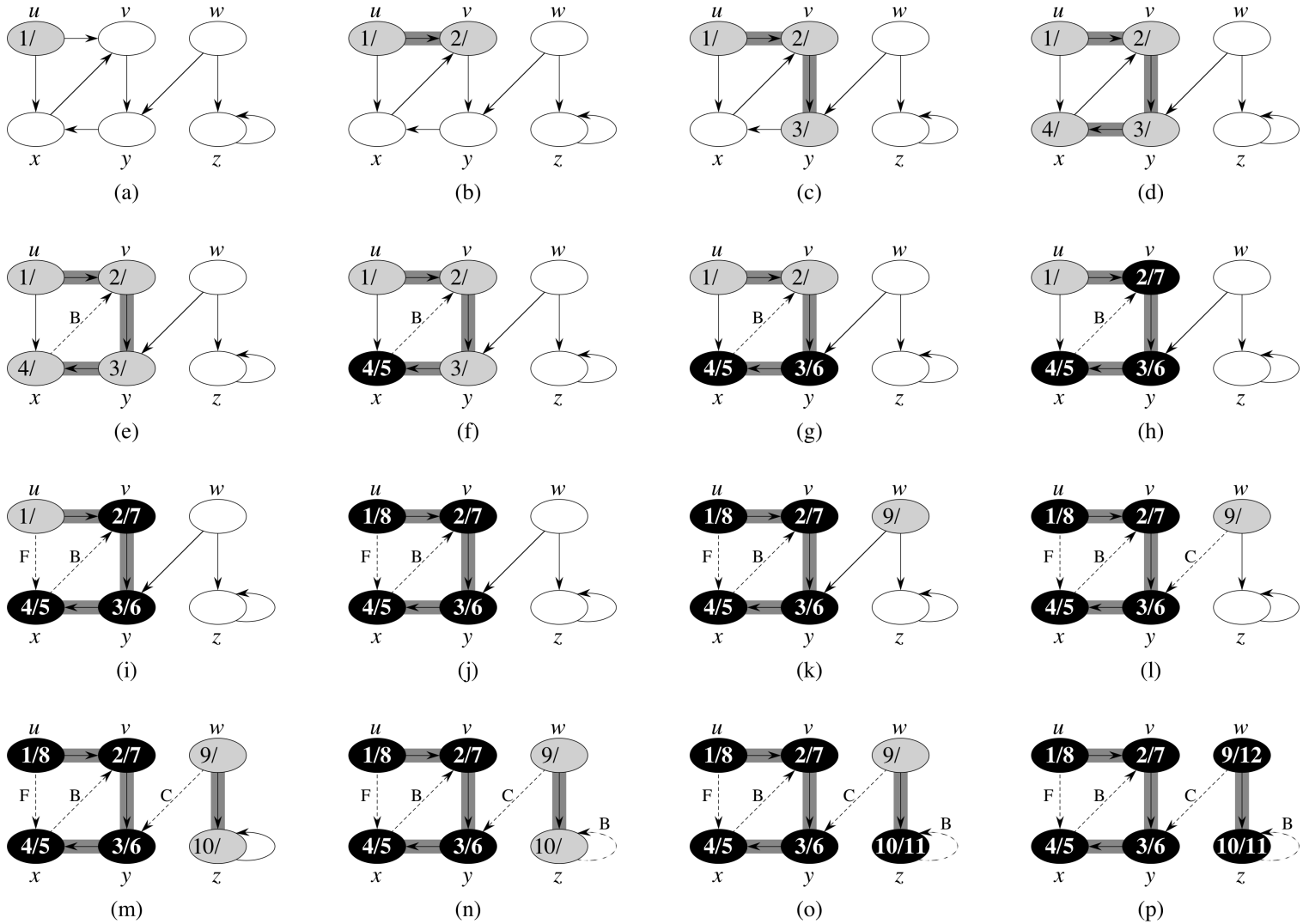
DFS tilføjer også timestamps $u.d$ for "discovery" (hvid $\rightarrow$ grå) og $u.f$ for "finish" (grå $\rightarrow$ sort) til alle knuder u. [$u.d$ er *ikke* "distance" i DFS.]

```
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 edge (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
```

Dybde-Først-Søgning (DFS)

Eksempel:



Egenskaber

Køretid: $O(n+m)$.

Beviset er det samme som under GenericGraphTraversalGlobal, da valgene (*) i DFS tager $O(1)$ tid.

Observér:

- Discovery (hvid $\rightarrow$ grå) af v = sæt $v.d$ = kald af DFS-Visit på v = PUSH af v på stakken.
- Finish (grå $\rightarrow$ sort) af v = sæt $v.f$ = retur fra kald af DFS-Visit på v = POP af v fra stakken.

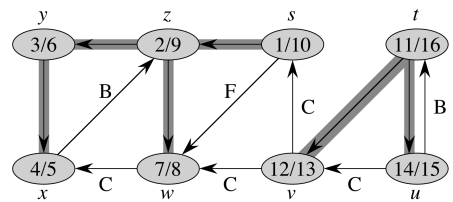
Værdien $v.\pi$ sættes ved kald af DFS-Visit på v. Af dette, samt ovenstående, følger at:

- Kanterne $(v.\pi, v)$ udgør præcis rekursionstræerne for DFS-Visit (ét træ for hvert kald fra DFS).
- Intervallet $[v.d, v.f]$ er den periode v er på stakken.
- Knuden v er grå hvis og kun hvis den er på stakken.

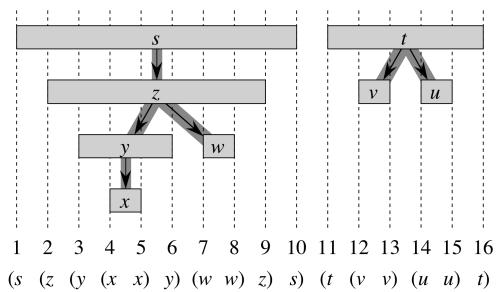
Egenskaber

Af måden en stak virker: Hvis to knuder u og v på et tidpunkt er på stakken samtidig, og v er øverst, må v poppes før u kan poppes.

Intervallet $[v.d, v.f]$ er den periode v er på stakken. Det følger derfor, at for alle par af knuder u og v må intervallerne $[u.d, u.f]$ og $[v.d, v.f]$ *enten* være disjunkte (u og v var aldrig på stakken samtidig) *eller* det ene interval må være helt indeholdt i den andet (u og v var på stakken samtidig, knuden med det største interval kom på først).



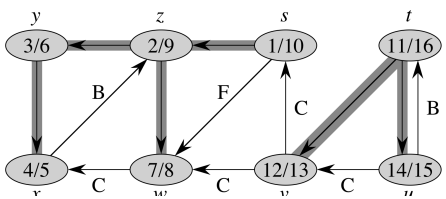
Discovery- og finish-tider er derfor nestede som parenteser er det.

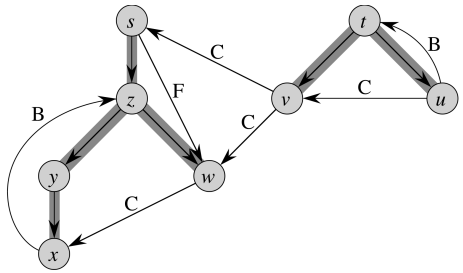


Egenskaber

Når en kant (u, v) undersøges fra u haves flg. tilfælde:

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).





Egenskaber

I lidt større detalje:

Når en kant (u, v) undersøges fra u haves flg. tilfælde:

1. *tree-kanter:* v hvid. Her er $u.d < v.d = \text{nu} < v.f < u.f$.
2. *back-kanter:* v er grå (er på stak – det må være under u, som er toppen af stakken (evt. $u = v$ hvis self-loop)). Her er $v.d \leq u.d \leq \text{nu} < u.f \leq v.f$.
3. *forward-kanter:* v er sort (den er ikke længere på stak, men har været det sammen med u). Her er $u.d < v.d < v.f \leq \text{nu} < u.f$.
4. *cross-kanter:* v er sort (den er ikke længere på stak, og har ikke været det sammen med u). Her er $v.d < v.f < u.d \leq \text{nu} < u.f$.

Bemærk, at disse cases kan genkendes, når en kant undersøges under DFS, nemlig via hvid/grå/sort-farvningen samt d-værdierne i de kantens to knuder (og vi ser ovenfor, at disse d-værdier *er* blevet sat, når kanten undersøges).

Egenskaber

For *uorienterede grafer* er der kun tree-kanter og back-kanter (såfremt en kant kategoriseres første gang den undersøges fra én af dens ender).

Dette følger af, at u allerede må være blevet undersøgt fra v hvis v er sort (hele nabolisten er gennemløbet) og kanten (v, u) må derfor allerede være kategoriseret. Derved kan 3 og 4 ikke opstå.

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).

Hvid-sti lemma

Hvid-sti lemma:

> Hvis findes en sti af hvide knuder (inkl. w) fra u til w til tid $u.d$, da gælder $u.d < w.d < w.f < u.f$.

Bevis:

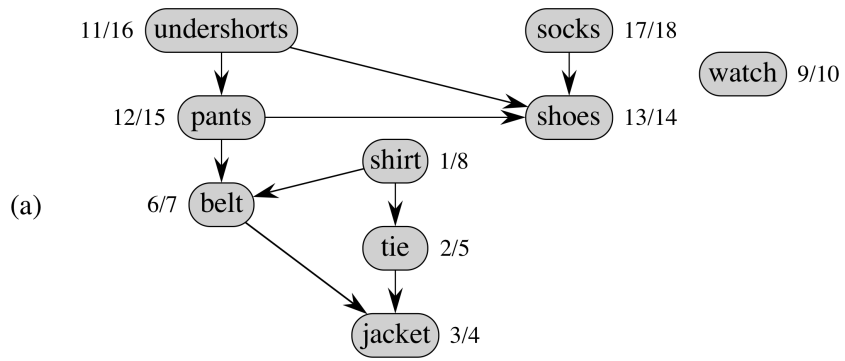
Da stien er hvid til tid $u.d$, gælder $u.d \leq v.d$ for alle knuder v på stien. Af parentesstrukturen for d- og f-tider gælder så enten 1) $u.d \leq v.d < v.f \leq u.f$ eller 2) $u.d < u.f < v.d < v.f$.

Antag, at 2) forekommer, og lad y være den første knude på stien, som opfylder 2) med y indsat på v's plads. Da har y en forgænger x, som opfylder 1) med x indsat på v's plads [evt. er x lig u, som jo opfylder 1)]. Men pga. kanten (x, y) må y opdages inden tid $x.f$, hvilket er i modstrid med at y opfylder 2). □

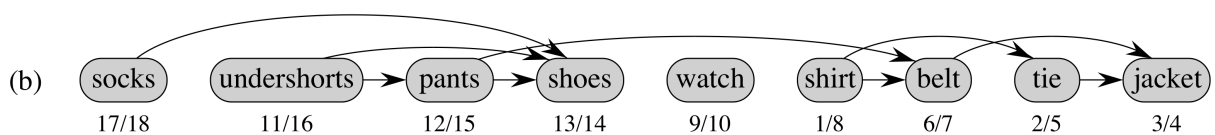
DAGs og topologisk sortering

DAG = Directed Acyclic Graph. En orienteret graf uden kredse (cycles).

Bruges ofte til at modellere afhængigheder. Eksempel:



Topologisk sortering af en DAG: en lineær ordning af knuderne så alle kanter går fra venstre til højre.



DAGs og topologisk sortering

Lemma: En orienteret graf har en kreds (cycle) $\Leftrightarrow$ der findes back-edges under et DFS-gennemløb.

Bevis:

$\Rightarrow$: DFS (med GLOBAL ydre loop) opdager alle knuder. Se på første knude v i kredsen som bliver grå. Dvs. at til tid $v.d$ er alle andre knuder hvide.

Af hvid-sti lemmaet fås så $v.d < u.d < u.f < v.f$ for den sidste knude u i kredsen (som peger på v), hvorved kanten (u, v) erklæres en backedge (v er grå, når denne kant undersøges).

$\Leftarrow$: Når en back-edge findes: Der er en kreds af nul eller flere trækanter (mellem knuder, som lige nu er på stakken) og én back-kant.

DAGs og topologisk sortering

Lemma: For en kant (u, v) gælder $u.f \leq v.f \Leftrightarrow$ kanten er en back-edge.

Bevis: Check de fire cases for kanter (tree, back, forward, cross) og deres ordning af $u.f$ og $v.f$, se tidligere slide.

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.

Så følgende algoritme finder en topologisk sortering i en DAG:

```
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
```

Tid: $O(n + m)$.

Sammenhængskomponenter i grafer

Ækvivalensrelationer

Repetition:

En relation R på en mængde S er en delmængde af $S \times S$. Når $(x, y) \in R$ siges x at stå i relation til y. Ofte skrives $x \sim y$, og relationen selv betegnes "$\sim$".

Relation kaldes en ækvivalensrelation hvis der for alle $x, y, z \in S$ gælder:

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

En ækvivalensrelation deler S i disjunkte delmængder (hver bestående af elementer som er i relation til hinanden, men ikke til andre elementer), og kaldes derfor også en partition.

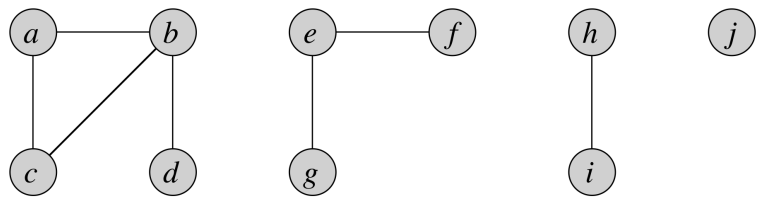
Ækvivalensrelationer på en grafs knuder

Uorienterede grafer:

For $v, u \in V$:

$$v \sim u \Leftrightarrow \text{ der er en (uorienteret) sti mellem } u \text{ og } v$$

Giver en partition af grafens knuder V:



De kaldes grafens sammenhængskomponenter (CC'er).

Finde dem? Via DFS eller BFS med GLOBAL ydre loop. Hvert kald fra ydre loop opdager præcis knuderne i én sammenhængskomponent (fra tidligere sætninger om GENERICGRAPHTRAVERSAL(s) ses, at et kald opdager præcis de knuder, som kan nås fra s via en sti af hvide knuder på tidspunktet for kaldet).

Tid? $O(n + m)$.

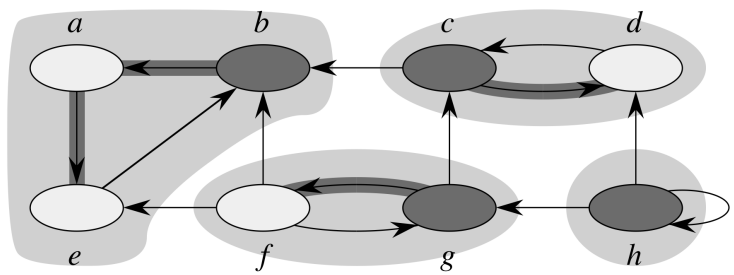
Ækvivalensrelationer på en grafs knuder

Orienterede grafer:

For $v, u \in V$:

$$v \sim u \quad \Leftrightarrow \quad \begin{array}{c} \text{der er en (orienteret) sti fra } u \text{ til } v \\ \text{OG} \\ \text{der er en (orienteret) sti fra } v \text{ til } u \end{array}$$

Giver en partition af grafens knuder V:



De kaldes grafens stærke sammenhængskomponenter (SCC'er).

Finde dem?

Finde stærke sammenhængskomponenter

Algoritme:

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

Her er G^T grafen G med alle kanter vendt.

Tid? $O(n + m)$.

Korrekthed? De næste sider...

Korrekthed af SCC algoritme

Sætning:

> Algoritmen SCC ovenfor er korrekt, dvs. træerne returneret fra det andet kald til DFS repræsenterer præcis G's SCC'er.

Bevis: Over de næste sider.

Bemærk først at

$$\text{Der er en sti } u \rightsquigarrow v \text{ i } G \quad \Leftrightarrow \quad \text{Der er en sti } v \rightsquigarrow u \text{ i } G^T$$

Heraf følger

$$u \text{ og } v \text{ i samme SCC i } G \quad \Leftrightarrow \quad u \text{ og } v \text{ i samme SCC i } G^T$$

Så G og G^T har de samme SCC'er.

Korrekthed af SCC algoritme

For en knudemængde $C \subseteq V$ defineres $f(C) = \max_{v \in C} v.f$ (hvor f angiver tiden fra første DFS i SCC-algoritmen).

Lemma 1:

> Hvis C, C' er to forskellige SCC'er i G, og (x, y) er en kant i G med $x \in C$ og $y \in C'$, da gælder $f(C) > f(C')$.

Bevis for Lemma 1 gives på næste side.

Da G^T er G med alle kanter vendt, og da SCC'erne er de samme i G^T og G, kan lemmaet også formuleres således:

Lemma 2:

> Hvis C, C' er to forskellige SCC'er i G^T, og (x, y) er en kant i G^T med $x \in C$ og $y \in C'$, da gælder $f(C) < f(C')$.

Korrekthed af SCC algoritme

Bevis (Lemma 1):

Lad u være den første knude i $C \cup C'$ som opdages.

Case 1: $u \in C$. Her er der en sti fra u til w for alle $w \in C \cup C'$, så udsagnet følger af hvid-sti lemma.

Case 2: $u \in C'$. Her er der en sti fra u til w for alle $w \in C'$, så af hvid-sti lemma følger $f(C') = u.f$.

Antag at der fandtes en knude $v \in C$ med $v.d < u.f$. Da $u.d < v.d$ (eftersom u var den først opdagede i $C \cup C'$) giver parentesstrukturen for d- og f-tider at $u.d < v.d < v.f < u.f$. Dvs. at v og u er på stakken samtidig, med v øverst (push'et senest). Da det er en invariant under DFS at der i grafen findes en sti mellem knuderne på stakken (fra tidligere til senere push'ede knuder), ville dette betyde en sti fra $u \in C'$ til $v \in C$. Sammen med kanten (x, y) ville dette medføre at alle knuder i $C \cup C'$ var i samme SCC, i modstrid med at C og C' er to forskellige SCC'er.

Derfor haves $v.d > u.f$ for alle $v \in C$, så $f(C) > u.f = f(C')$. $\square$

Korrekthed af SCC algoritme

Vi viser nu sætningen om korrekthed af SCC-algoritmen ved at vise at for alle k gælder:

> Knuderne i de k første træer genereret under den anden DFS i SCC-algoritmen udgør hver især en SCC i G^T.

Da SCC'erne i G og G^T er de samme, og da alle knuder i grafen er i et af træerne, viser dette korrektheden.

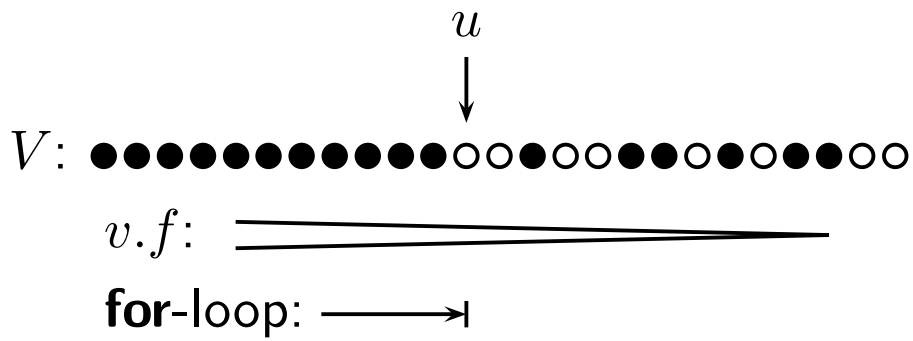
Vi viser ovenstående udsagn via induktion på k.

Skridt: Antag sandt for k, vis sandt for $k+1$.

Det $(k+1)$'te træ genereres ved det $(k+1)$'te kald til DFS-Visit i **for**-løkken i det ydre loop i DFS. Lad u være knude, der kaldes på.

Hvis vi stiller knuderne op i **for**-løkkens rækkefølge (efter aftagende $v.f$-værdi), ser situationen sådan ud på tidspunktet for dette kald:

Korrekthed af SCC algoritme



Sorte knuder er de indtil nu opdagede under DFS, hvide er de uopdagede.

Lad C være SCC'en indeholdende u, og lad T være træet genereret af kaldet på u. Af induktionsantagelsen udgør de sorte knuder præcis k af grafens SCC'er. Derfor må alle andre SCC'er ligge inden i de hvide knuder, og C er en af disse (da u er hvid).

Eftersom der ved starten af kaldet er en hvid sti fra u til alle $w \in C$, giver hvid-sti lemma at $C \subseteq T$.

Korrekthed af SCC algoritme

Lad C' være en vilkårlig hvid SCC forskellig fra C. Pga. **for**-løkkens rækkefølge ses $u.f = f(C) > f(C')$.

Hvis der var en kant i G^T, som gik fra C til C', ville Lemma 2 give $f(C) < f(C')$.

Så ingen kant i G^T kan gå fra C til C'. Da DFS-VISIT ikke besøger de sorte knuder, kan den derfor ikke forlade C. Heraf ses $T \subseteq C$.

Vi har i alt vist $T = C$, hvilket viser udsagnet for $k + 1$.

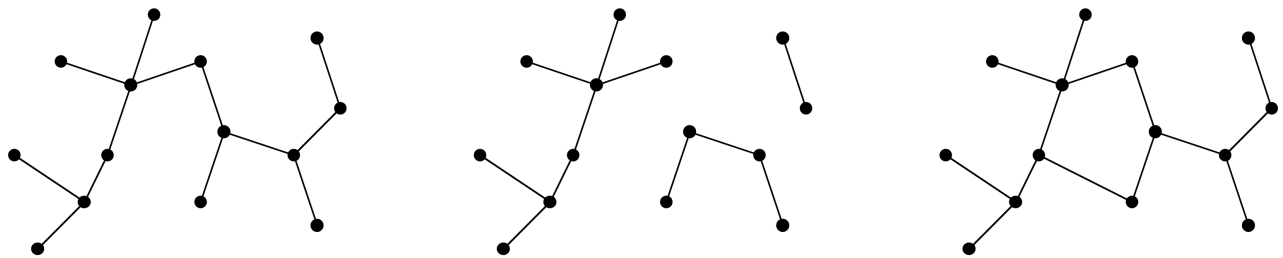
Basis: Samme argument, blot lidt simplere (der er ingen sorte knuder, og u er første knude i rækkefølgen), viser udsagnet for $k = 1$. □

Minimum udSpændende Træer (MST)

Træer

Et (frit/u-rodet) træ er en uorienteret graf $G = (V, E)$ som er

- Sammenhængende: der er en sti mellem alle par af knuder.
- Acyklisk: der er ingen kreds af kanter.



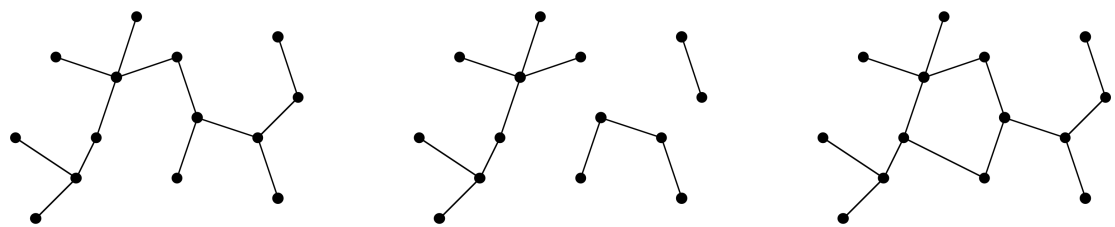
Træ Skov Graf med kreds (ikke træ)

(Uorienteret, acyklisk graf = skov af træer.).

Træer

Sætning (B.2): For uorienteret graf $G = (V, E)$ er flg. ækvivalent (gælder det ene, gælder det andet):

1. G er et træ (dvs. sammenhængende og acyklisk).
2. G er sammenhængende, men er det ikke hvis nogen kant fjernes.
3. G er sammenhængende og $m = n - 1$.
4. G er acyklisk, men er det ikke hvis nogen kant tilføjes.
5. G er acyklisk og $m = n - 1$.
6. Mellem alle par af knuder er der præcis én vej.



Bevis (ikke pensum): se appendix B.5.

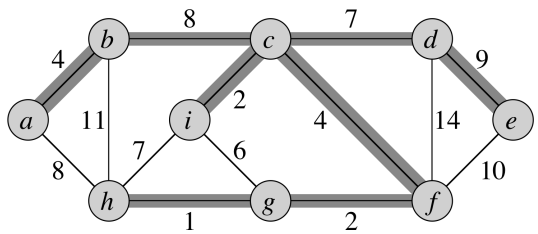
Læs (pensum) appendix B.4 og B.5 for basale definitioner for grafer.

Minimum Spanning Tree (MST)

Udspændende træ for uorienteret, sammenhængende graf $G = (V, E)$:

En delgraf $T = (V, E')$, $E' \subseteq E$, som er et træ.

NB: *samme* knudemængde V. Vi tænker fra nu af på T blot som E'.



Iflg. sætning B.2 har alle udspændende træer $n - 1$ kanter.

Minimum udSpændende Træ (MST) for en *vægtet* uorienteret sammenhængende graf G: et udspændende træ for G som har mindst mulig sum af kantvægte (dvs. intet udspændende træ har mindre sum).

Motivation: forbind punkter i et forsyningsnetværk (elektricitet, olie,...) billigst muligt. Kant i G: mulig forbindelse, vægt: pris for at etablere forbindelse. Dette var motivationen for den første algoritme for problemet (Borůvka, 1926, Østrig-Ungarn, nu Tjekkiet).

Algoritmer for MST

Grundidé er grådig algoritme: byg MST ved at vælge kanterne én efter én ved hjælp af en passende regel.

Korrekthed: via den sædvanlige invariant for korrekthed af grådige algoritmer: "Hvad vi har bygget indtil nu er en del af en optimal løsning".

Dvs. følgende invariant, hvor $A \subseteq E$ er de indtil nu valgte kanter:

> Der findes et MST, som indeholder A.

Terminologi: safe kant for A er en kant, som kan tilføjes uden at ødelægge invarianten (mindst én må findes, når invarianten gælder og $|A| < n - 1$).

Algoritmer for MST

Invariant: Der findes et MST, som indeholder A.

```
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
```

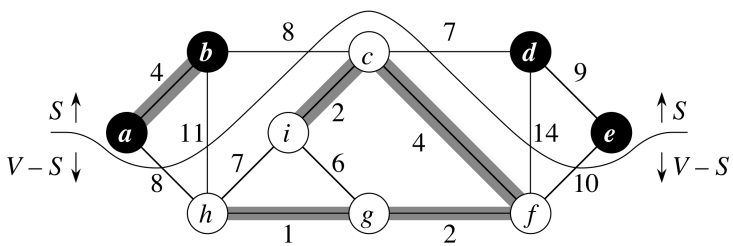
- Initialisering: Enhver sammenhængende graf har et mindst ét ST (via sætningen fra B.5, punkt 2 - fjern kanter til betingelsen nås), og har derfor et MST. Dette indeholder kantmængden $\emptyset$.
- Vedligeholdelse: Er det samme, som at den valgte kant er safe.
- Terminering: ethvert (M)ST indeholder præcis $n-1$ kanter. Da A vokser med én kant per iteration, giver invarianten, at algoritmen terminerer, og at A da er et MST (A er indeholdt i et MST, og har samme antal kanter som dette, så A er lig dette).

Cuts

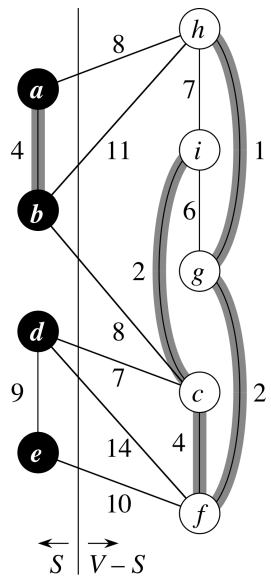
MEN: Hvordan finde en safe kant?

Cut: En delmængde $S \subseteq$ af knuderne.

Kan ses som en to-deling af knuderne i to mængder S og $V - S$.



(a)



(b)

Kant henover cut: en kant i $S \times (V - S)$.

Cut-sætning

Sætning:

Hvis

- der eksisterer et MST, som indeholder A,
- S er et cut, som A ikke har kanter henover,
- e er en letteste kant blandt kanterne henover cuttet,

så

- er e safe for A (dvs. der der eksisterer et MST som indeholder $A \cup \{e\}$).

Cut-sætning

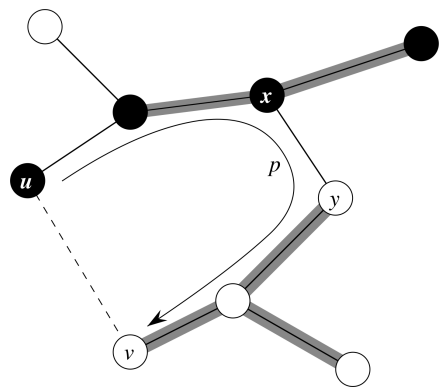
Bevis:

- Der findes et MST T som indeholder A.
- Vi skal lave et MST T' som indeholder $A \cup \{e\}$.

Lad $e = (u, v)$ være en letteste kant henover cuttet S.

Da T er sammenhængende, må der være en sti i T mellem u og v, hvorpå der er mindst én kant (x, y) henover cuttet S.

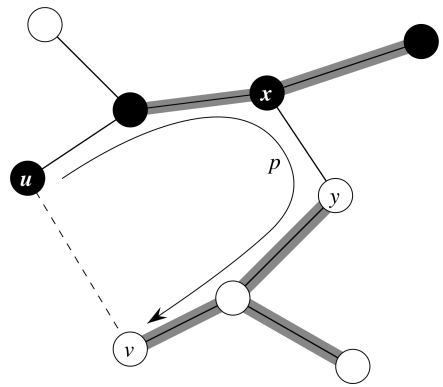
Lad T' være T med (x, y) udskiftet til $e = (u, v)$:



(Viste kanter = T, fede kanter = A, cut er angivet med knudefarver.)

Cut-sætning

Lad T' være T med (x, y) udskiftet til $e = (u, v)$:



Som T er T' stadig sammenhængende (i alle stier kan (x, y) erstattes af resten af stien fra u til v, samt kanten (u, v)), og har n knuder og $n - 1$ kanter. T' er derfor et træ (pga. sætning tidligere). Det kan kun være lettere end T. Derfor er T' også et MST.

T' indeholder $A \cup \{e\}$, da den fjernede kant (x, y) ikke er i A, eftersom A ingen kanter har henover cuttet. □

Brug af cut-sætning i MST-algoritmer

```
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
```

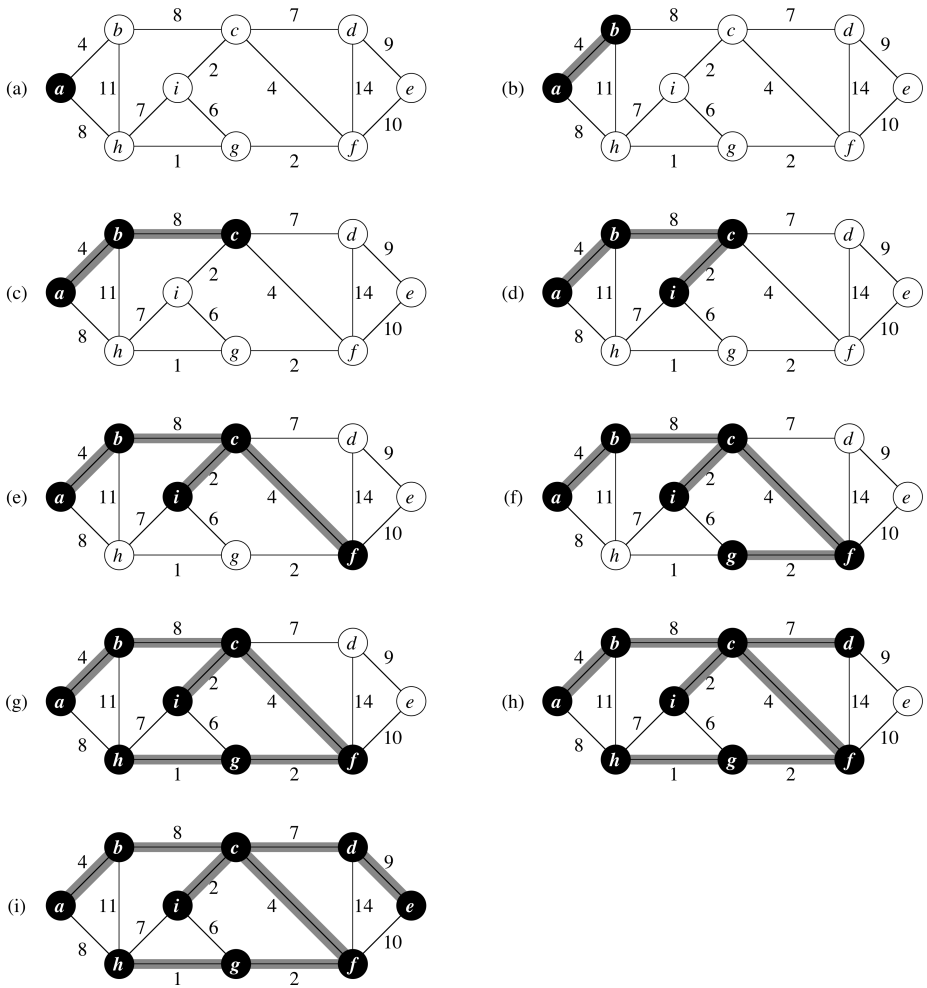
Invariant: Der findes et MST som indeholder de valgte kanter A.

- En ny kant (u, v) med begge endepunkter i *samme* sammenhængskomponent i $G' = (V, A)$ vil introducere en kreds og dermed ødelægge invarianten. Sådanne er derfor aldrig safe.
- En ny kant (u, v) med endepunkterne i *forskellige* sammenhængskomponenter C_1 og C_2 i $G' = (V, A)$ er safe, hvis den er en letteste kant ud af C_1: brug cut-sætning på cuttet C_1.

Man ser nemt, at hvis A udvides med en kant med endepunkterne i forskellige sammenhængskomponenter C_1 og C_2 i G', vil det ændre sammenhængskomponenterne i G' ved at C_1 og C_2 slås sammen til én sammenhængskomponent.

Prim-Jarnik MST-algoritmen (Prim 1957, Jarnik 1930)

Tager udgangspunkt i en (vilkårlig) startknude r. Udvider hele tiden r's sammenhængskomponent i .



Prim-Jarnik MST-algoritmen

Tager udgangspunkt i en (vilkårlig) startknude r. Udvider hele tiden r's sammenhængskomponent C i $G' = (V, A)$.

En knude $v \in V - C$ gemmer information om sin korteste kant henover cuttet C i felterne v.key og $v.\pi$. Mængden A er $\{(v, v.\pi) \mid v \in C - \{r\}\}$.

Knuderne i $V - C$ opbevares i en (min-)prioritetskø Q.

```
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))
```

Korrekthed: via cut-sætningen og invarianten.

Køretid: n INSERT, n EXTRACTMIN, m DECREASEKEY på prioritetskø af størrelse $O(n)$, i alt $O(m \log n)$, da $m \geq n - 1$ (G sammenhængende).

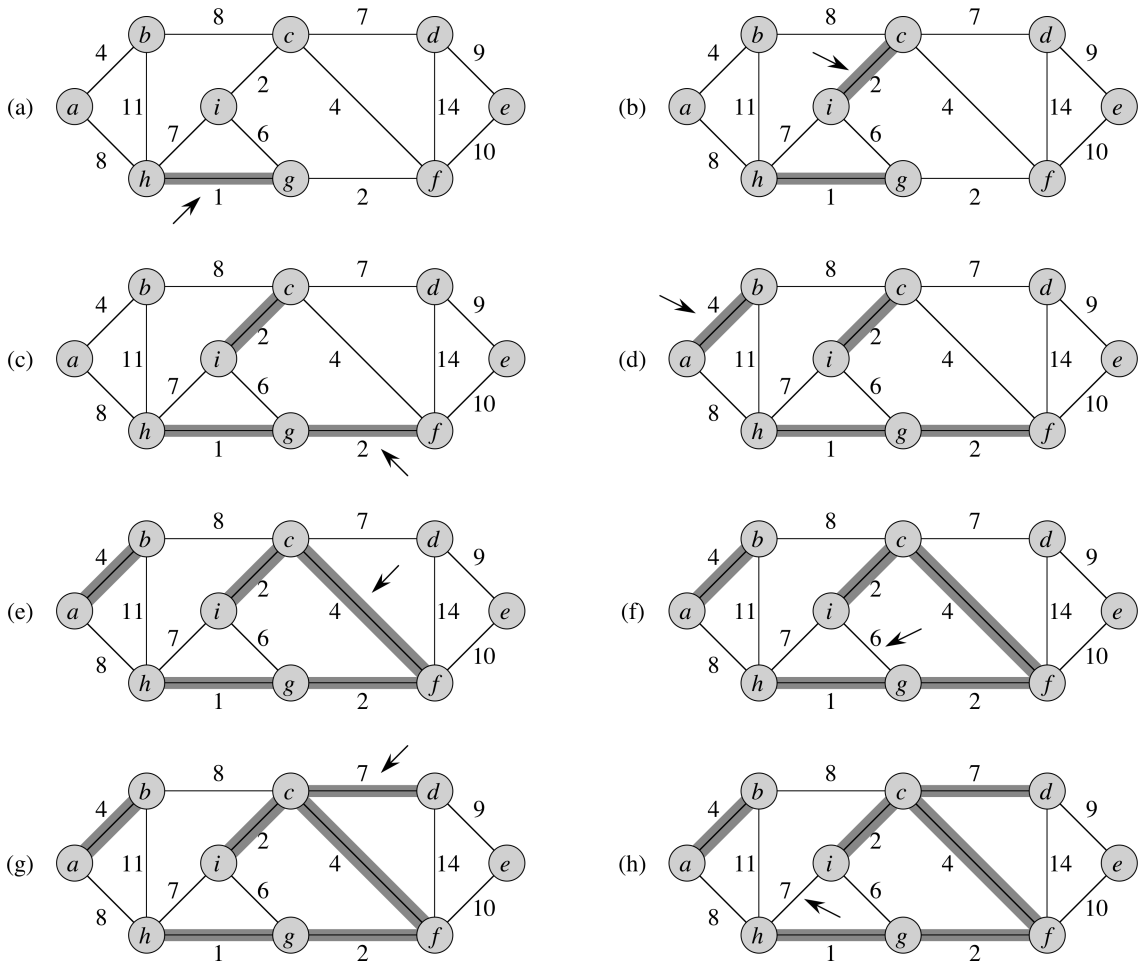
Kruskal MST-algoritmen (1956)

Forsøger at tilføje kanter til A i global letteste-først orden.

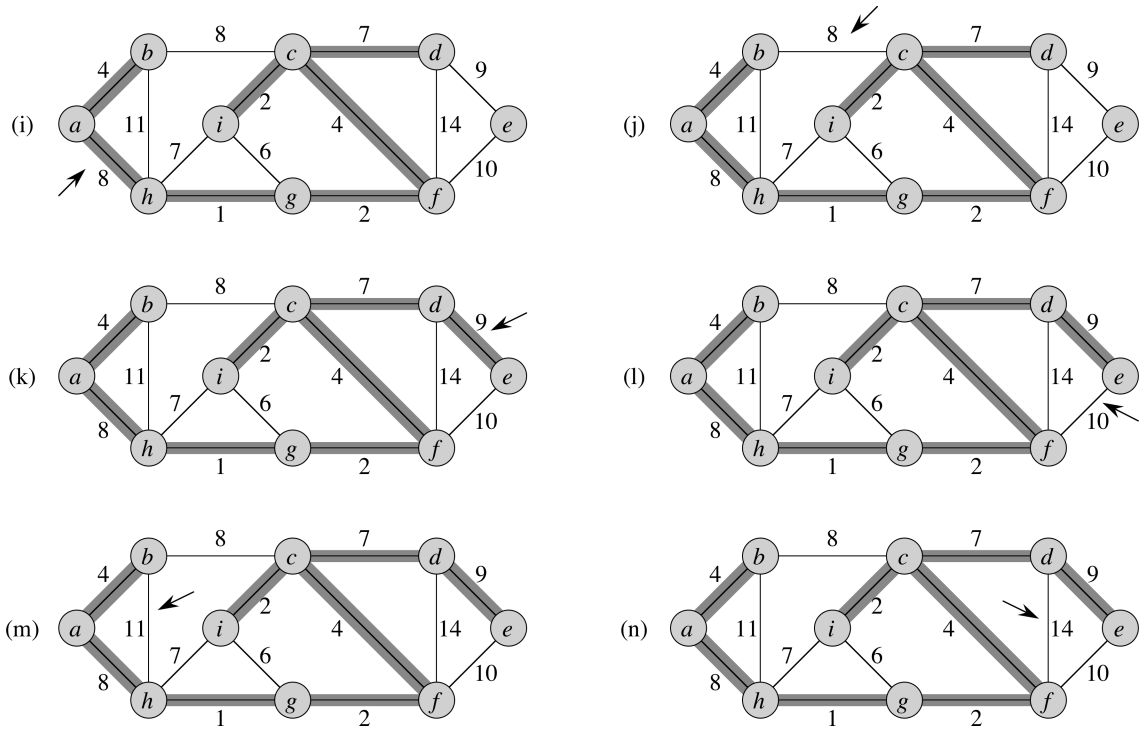
Recap fra side 11:

1. En kant (u, v) kan aldrig tilføjes til A, hvis u og v ligger i samme sammenhængskomponent i $G' = (V, A)$.
2. Hvis en kant (u, v) mellem to forskellige sammenhængskomponenter tilføjes til A, vil disse to sammenhængskomponenter blive til én bagefter.

Kruskal MST-algoritmen



Kruskal MST-algoritmen



Kruskal MST-algoritmen

Vedligeholder sammenhængskomponenterne i $G' = (V, A)$ ved hjælp af en *disjoint-set* datastruktur på V:

$$\textsc{Make-Set}(x),\ \textsc{Union}(x, y)\ \textsc{Find-Set}(x)$$

Mere præcist:

```
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
```

Kruskal MST-algoritmen

```
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
```

Klart ud fra recap side 14 om sammenhængskomponenter at:

1. Datastrukturen vedligeholder sammenhængskomponenterne i $G' = (V, A)$.
2. En kant undersøgt (IF-sætningen) har begge endepunkter i samme sammenhængskomponent efter undersøgelsen, uanset udfaldet af testen i IF-sætningen. Da sammenhængskomponenter i G' kun slås sammen undervejs, gælder dette også for kanten i resten af algoritmen.

Kruskal, korrekthed

På det tidspunkt, hvor algoritmen tilføjer en kant (u, v) til A, ligger u og v i forskellige sammenhængskomponenter C_1 og C_2 i $G' = (V, A)$. [Dette følge af punkt 1 samt testen i IF-sætningen.]

Vi ser på cuttet givet ved u's sammenhængskomponent C_1. Alle lettere kanter er allerede undersøgt, og har derfor begge endepunkter i samme sammenhængskomponent i G' [punkt 2]. Derfor er (u, v) en letteste kant henover dette cut, og vi kan derfor bruge cut-sætningen.

Når algoritmen stopper, er alle kanter undersøgt. Enhver kant i inputgrafen $G = (V, E)$ har derfor begge endepunkter i samme sammenhængskomponent i $G' = (V, A)$ [punkt 2]. Sådanne kanter kan ikke tilføjes A uden at introducere en kreds.

Så A er selv det MST (fra invarianten), som indeholder A, da ingen kanter kan tilføjes A. Så algoritmen er korrekt.

Bemærk at der lavet præcis $n - 1$ UNION operationer undervejs, da hver tilføjer én kant til A, og da et MST har $n - 1$ kanter.

Kruskal, køretid

Arbejde:

Sortér m kanter
Lav n MAKE-SET, $n-1$ UNION, m FIND-SET.

Fra tidligere: der findes en datastruktur for disjoint-sets hvor

- n MAKE-SET(x)
- $n-1$ UNION(x, y)
- m FIND-SET(x)

tager i alt $O(m + n \log n)$ tid.

Samlet køretid for Kruskal er

$$O(m \log m)$$

eftersom $m \geq n - 1$, da inputgrafen er sammenhængende.

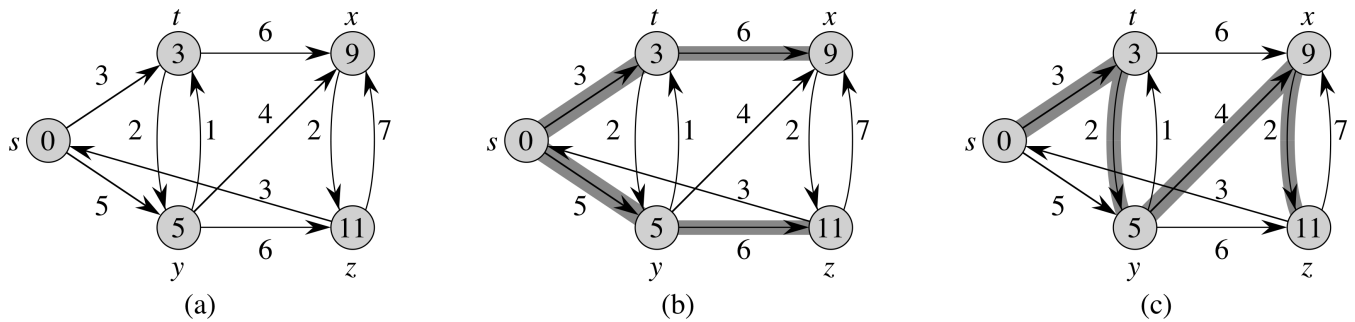
Korteste veje

Korteste veje i vægtede grafer

Længde af sti = sum af vægte af kanter på sti.

$\delta(u, v)$ = længden af en korteste sti fra u til v. Sættes til ∞ hvis ingen sti findes.

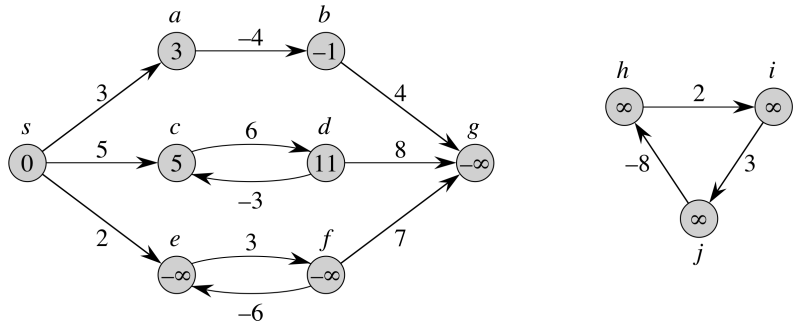
Single-source shortest-path problemet: Givet $s \in V$, find $\delta(s, v)$ (og en konkret sti af denne længde) for alle $v \in V$.



Bemærk at prefixer af korteste veje selv må være korteste veje: hvis $v_1, v_2, \ldots, v_k$ er en korteste vej fra v_1 til v_k, så er $v_1, v_2, \ldots, v_i$ en korteste vej fra v_1 til v_i for alle $i \leq k$ (ellers kan vejen fra v_1 til v_k gøres kortere).

Korteste veje i vægtede grafer

Problemet er ikke veldefineret, hvis der findes kredse (som kan nås fra s) med negativ sum, idet der så findes veje med vilkårlig lav længde:



Omvendt: hvis der ikke findes sådanne negative kredse, kan vi nøjes med at se på simple stier (ingen gentagelser af knuder på stien). Der er et endeligt antal sådanne stier (højst $n!$), så "længde af korteste sti" er veldefineret.

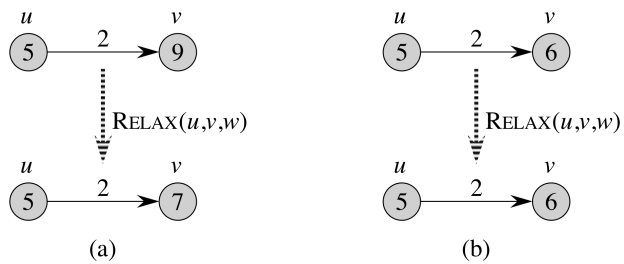
Relaxation – en generel teknik til at finde korteste veje

Idé: Brug kanter til at udbrede information fra knude til knude om længder af kendte stier. Kaldes RELAX af kanten.

Hvis u har information om, at der er en sti fra s til u af længde $u.d$, og (u, v) er en kant af med vægt w, så findes der en sti af længde $u.d + w$ til v. Er det bedre information for v, end hvad den har lige nu?

```
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
```



[Detalje: Implementation af "∞", skal fungere matematisk korrekt med "$>$" og "$+$" (bare at bruge `Integer.MAX_VALUE` som "∞" er ikke nok).]

Relaxation

```
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
```

Vi vil møde en række algoritmer, som starter med INIT-SINGLE-SOURCE og derefter kun ændrer $v.d$ og $v.\pi$ via RELAX.

For sådanne algoritmer gælder følgende invariant (ses nemt ved induktion på antal RELAX):

> Hvis $v.d < \infty$ findes en sti fra s til v af længde $v.d$.

Relaxation

Hvis $v.d < \infty$ findes en sti fra s til v af længde $v.d$.

Derfor gælder altid $\delta(s, v) \leq v.d$. Argument: hvis $v.d < \infty$ følger det af ovenstående, hvis $v.d = \infty$ gælder $\delta(s, v) \leq v.d$ uanset værdien af $\delta(s, v)$.

Da $v.d$ kun kan falde ved brug af $\textsc{Relax}$, følger det, at hvis på et tidspunkt $\delta(s, v) = v.d$, vil $v.d$ ikke kunne ændres senere (og dermed kan heller ikke $v.\pi$ ændres, da $v.d$ og $v.\pi$ ændres på samme tidspunkt i koden).

Specielt gælder, at hvis $\delta(s, v) = v.d$ for alle knuder, vil ingen kant (u, v) kunne relaxeres (dvs. $v.d \leq u.d + w(u, v)$ gælder for alle kanter (u, v)).

De korteste stier kan findes via $v.\pi$-pointers

Invariant:

- Mængden S af knuder v med $\delta(s, v) = v.d < \infty$ udgør et træ med $v.\pi$ som parent pointers og s som rod.
- For en knude v i træet vil stien mod roden svare til et baglæns gennemløb af en sti i grafen fra s til v af længde $\delta(s, v)$.

Dette vises ved induktion på antal RELAX.

Basis: Lige efter initialisering er s den eneste knude v, som har $v.d < \infty$. Da $s.d = 0$ og $s.\pi = \text{NIL}$ efter initialisering, er $S = \{s\}$ og invarianten opfyldt med et træ af størrelse én, hvis blot $\delta(s, s) = 0$.

[Bemærk at $\delta(s, s) \neq 0$ kun er muligt hvis s ligger på en negativ kreds. Men så gælder for alle knuder v enten $\delta(s, v) = -\infty$ (hvis v kan nås fra s) eller $\delta(s, v) = \infty$ (hvis v ikke kan nås fra s). Hvis $v.d < \infty$, svarer $v.d$ til længden af en konkret sti (se tidligere), og er derfor forskellig fra $-\infty$. Så S er altid tom, og der er intet at vise.]

De korteste stier kan findes via $v.\pi$-pointers

Induktionsskridt: For en $\textsc{Relax}$ som ikke ændrer noget, er der intet at vise. For en $\textsc{Relax}$, som ændrer $v.d$ (og dermed $v.\pi$): her kan v ikke være i S før $\textsc{Relax}$ (da $\delta(s, v) < v.d$ på det tidspunkt), så alle eksisterende knuder i træet er uændrede (inkl. deres parent pointers). Hvis v indlemmes i S pga. denne $\textsc{Relax}$, så lad (u, v) være kanten, som blev relaxeret, og lad w være dens vægt. Vi har så $\delta(s, v) = v.d = u.d + w$. Derfor må $\delta(s, u) = u.d$ gælde (hvis der var en vej kortere end $u.d$ til u, var der en vej kortere end $u.d + w$ til v) og $u.d < \infty$ gælder også (ellers ville $\textsc{Relax}$ ikke ske). Så u er med i S, får v som barn, og sætningen gælder klart igen.

Dijkstras algoritme [1959]

Grådig algoritme som trinvis opbygger mængde S af knuder med korrekte $v.d$ og $v.\pi$. Bruger en prioritetskø Q. Kræver alle kantvægte ≥ 0.

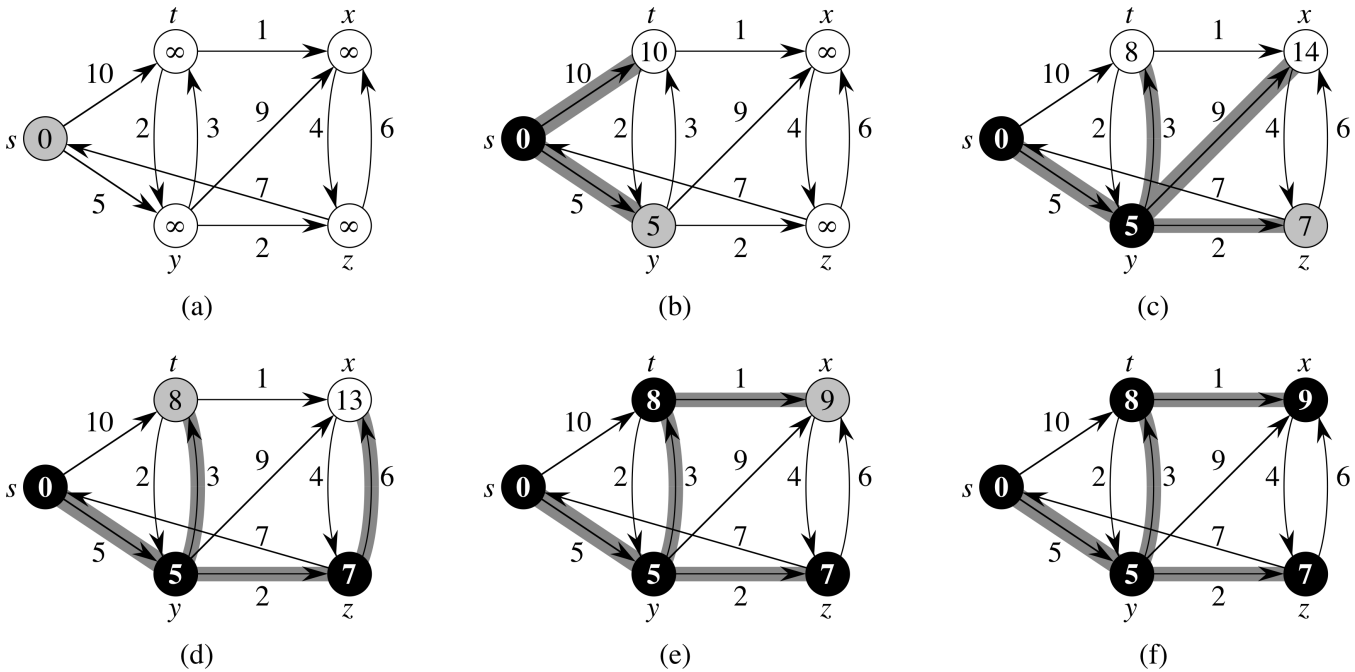
```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid: n INSERT (eller én BUILD-HEAP), n EXTRACT-MIN og m DECREASE-KEY (i RELAX). I alt $O(m \log n)$ hvis prioritetskøen implementeres med en heap.

Invariant: Når u indlemmes i S (dvs. udtages med en EXTRACT-MIN) er $u.d = \delta(s, u)$ (hvis alle kantvægte er ≥ 0).

Bevis for invariant: Et induktionsbevis (gennemgået på tavle). Af invarianten følger at algoritmen er korrekt (da alle knuder er i S til sidst).

Dijkstra, eksempel



Path-relaxation lemma

Dijkstra antager ikke-negative vægte. Vi kigger nu på algoritmer, som kan klare negative vægte (men naturligvis ikke negative kredse, som gør korteste vej problemet udefineret).

Vi starter med flg. lemma:

Lemma: Hvis $s = v_1, v_2, \ldots, v_k = v$ er en korteste vej fra s til v, og en algoritme laver $\textsc{Relax}$ på kanterne $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ efter tur (med en vilkårlig mængde $\textsc{Relax}$ af andre kanter mellem disse $\textsc{Relax}$), da er $\delta(s, v) = v.d$ efter den sidste af disse $\textsc{Relax}$.

Bevis: Det ses ved induktion på i, at efter der er lavet $\textsc{Relax}$ på kanten (v_{i-1}, v_i) i sekvensen ovenfor, kan $v_i.d$ højst være lig summen af vægtene af de første $i - 1$ kanter i stien.

Så efter den sidste af disse $\textsc{Relax}$ er $v.d \leq \delta(s, v)$, eftersom stien er en korteste sti til v. Da $\delta(s, v) \leq v.d$ altid gælder, er $\delta(s, v) = v.d$.

Algoritme for DAGs [unknown]

Recall: DAG = Directed Acyclic Graph.

Recall: En topologisk sortering kan findes via DFS i tid $O(n+m)$.

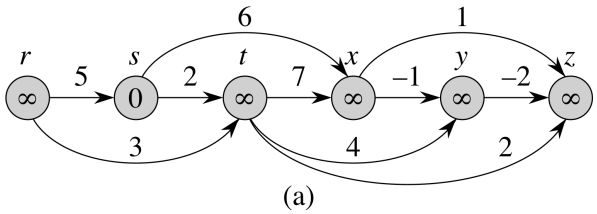
```
DAG-SHORTEST-PATHS(G, w, s)
  topologically sort the vertices
  INIT-SINGLE-SOURCE(G, s)
  for each vertex u, taken in topologically sorted order
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Køretid: $O(n+m)$.

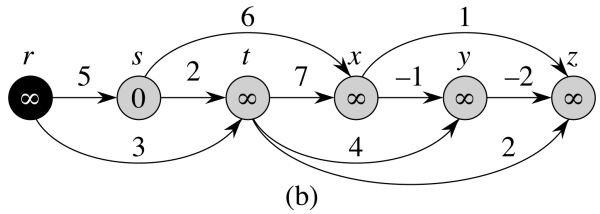
Sætning: Når algoritmen stopper er $v.d = \delta(s,v)$ for alle $v \in V$.

Bevis: For en knude v med en sti fra s til v: alle knuder på en korteste sti er blevet relaxeret i rækkefølge (hvorved korrekte δ-værdier sættes på denne sti pga. path-relaxation lemma). For alle andre knuder gælder $\infty = \delta(s,v)$ så korrekthed her følger af $\delta(s,v) \leq v.d$.

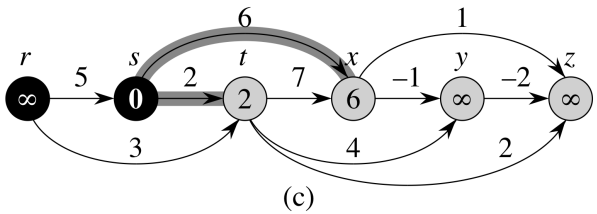
Algoritmen for DAG, eksempel



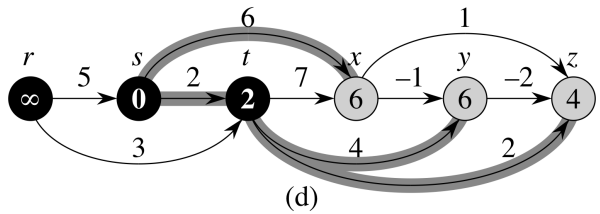
(a)



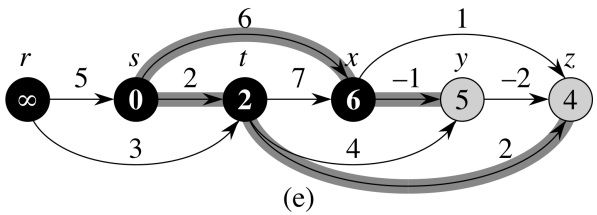
(b)



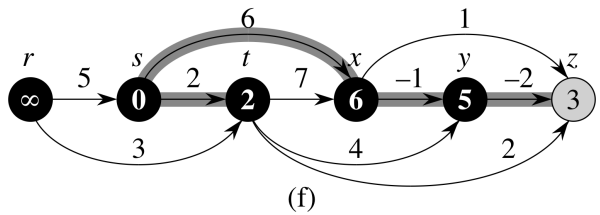
(c)



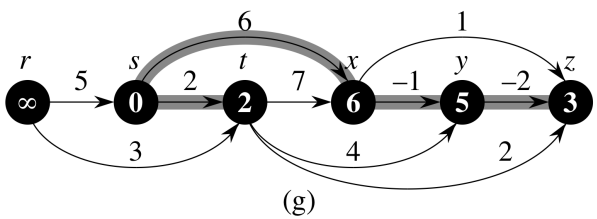
(d)



(e)



(f)



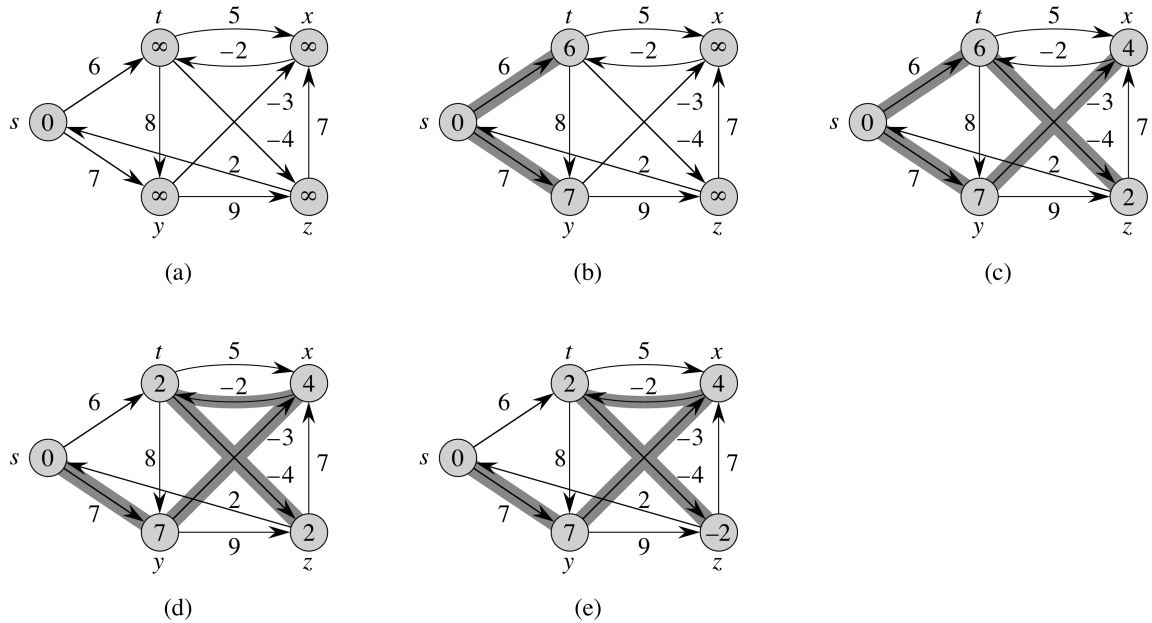
(g)

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

```
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
```

Køretid: $O(nm)$

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



(a) (b) (c) (d) (e)

[Relaxeringsrækkefølge:
$(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.]

Bellman-Ford, korrekthed

Idéen bag Bellman-Ford er, at den vedligeholder følgende invariant:

> Efter i iterationer af første **for**-løkke gælder $v.d = \delta(s, v)$ for alle knuder v, der har en korteste vej med højst i kanter.

Denne invariant følger direkte af path-relaxation lemmaet.

Mere præcist gælder følgende for Bellman-Ford:

Sætning: Hvis der findes en negativ kreds, som kan nås fra s, svarer Bellman-Ford FALSE. Ellers svarer den TRUE, og $v.d = \delta(s, v)$ for alle $v \in V$ når den stopper.

Bellman-Ford, korrekthed

Bevis:

Case 1: der er ingen negative kredse, som kan nås fra s.

Så har alle knuder, som kan nås fra s, en simpel korteste vej (en vej uden gentagelser af knuder). En sådan vej har højst n knuder, og derfor højst $n-1$ kanter. Af invarianten ovenfor gælder $v.d = \delta(s, v)$ for disse knuder, når den første **for**-løkke slutter. For knuder, der ikke kan nås fra s, gælder dette allerede efter initialiseringen i starten. Så $v.d = \delta(s, v)$ for alle knuder, når den første **for**-løkke slutter.

Når $v.d = \delta(s, v)$ for alle knuder, kan $\textsc{Relax}$ ikke ændre nogen $v.d$ længere (jvf. tidligere observation). Derfor svarer bliver **if**-casen i den anden **for**-løkke aldrig sand, og algoritmen svarer TRUE.

Bellman-Ford, korrekthed

Case 2: der er en negativ kreds C, som kan nås fra s.

Vi bemærker først, at hvis en knude v kan nås fra s, kan den også nås via en simpel sti (en sti uden gentagelser blandt knuder). En sådan sti har højst n knuder.

Det er let at se via induktion over i, at efter i iterationer af første **for**-løkke er d-værdien endelig for de første $i + 1$ knuder på denne simple sti. Derfor gælder $v.d < \infty$ ved **for**-løkkens afslutning.

Specielt gælder dette alle knuder på C (de kan alle nås fra s).

Bellman-Ford, korrekthed

Antag, at Bellman-Ford i Case 2 ikke svarer FALSE. Så gælder ved algoritmens afslutning

$$v_{i+1}.d \leq v_i.d + w(v_i, v_{i+1})$$

for $1 \leq i \leq k$ (med $v_{k+1} = v_1$). Og dermed gælder

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_i.d + \sum_{i=1}^{k} w(v_i, v_{i+1}).$$

Da $v_i.d < \infty$ for alle i, er de to første summer ikke bare ens, men også $< \infty$, så de kan trækkes fra og give

$$0 \leq \sum_{i=1}^{k} w(v_i, v_{i+1}),$$

i modstrid med at kredsen er negativ. Så algoritmen må svare FALSE. □

Korteste veje mellem alle par af knuder

All-pairs shortest-path problemet: For alle $s \in V$, find $\delta(s, v)$ (og en konkret sti) for alle $v \in V$.

Én mulighed: køre Dijkstra fra hver source $s \in V$ (kræver ikke-negative vægte): $O(nm \log n)$ tid.

Eller: køre Bellman-Ford-Moore fra hver source $s \in V$ (hvis der er negative vægte): $O(n^2 m)$ tid.

En anden mulighed: Floyd-Warshalls algoritme. $O(n^3)$ tid. Klarer negative vægte.

Endnu en mulighed: Johnsons algoritme. Kører i $O(nm \log n)$ tid. Klarer negative vægte.

Floyd-Warshalls algoritme [1962]

Bruger ikke RELAX, er i stedet baseret på dynamisk programmering.

Input er grafen i adjacency-matrix repræsentationen i en variant W med vægte på kanter: $w_{ii} = 0$, $w_{ij} = w(i,j)$ hvis $(i,j) \in E$, $w_{ij} = \infty$ ellers.

Output er også på matrice-form:

$D = (d_{ij})$, $d_{ij} = \delta(v_i, v_j) =$ længden af en korteste sti fra v_i til v_j. Sættes til ∞ hvis ingen sti findes.

$\Pi = (\pi_{ij})$, $\pi_{ij} =$ sidste knude før v_j på en korteste sti fra knude v_i til knude v_j. Sættes til NIL hvis ingen sti findes.

Floyd-Warshalls algoritme

(Kun konstruktion af D-matricen vises, se bogen for Π-matricen.)

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 $D^{(n)}$

Køretid: $O(n^3)$. Plads: $O(n^2)$ (kun forrige $D^{(k)}$ matrice behøves gemmes).

Sætning: Når algoritmen stopper er d_{ij} og π_{ij} i den sidste matrice sat korrekt for alle $v_i, v_j \in V$ (hvis ingen negativ kreds er i grafen).

Bevis: Invarianten er, at $D^{(k)}$ indeholder længden af korteste vej mellem v_i og v_j som kun passerer knuderne $v_1, v_2, \ldots, v_k$ (udover endepunkterne v_i og v_j). Vises ved induktion på k.

Johnsons algoritme [1977]

Bruger:

- Kører Bellman-Ford-Moore én gang på let udvidet graf.
- Herudfra justering af kantvægte så alle bliver positive uden essentielt at ændre korteste veje (se lemma på næste side).
- Kører Dijkstra fra alle knuder.

Kører i $O(nm \log n + nm) = O(nm \log n)$ tid, klarer negative vægte.

Re-weighting

Se på situationen, hvor vi til alle knuder $v \in V$ tildeler et tal $\phi(v)$.

Ud fra ϕ kan vi lave nye vægte $\tilde{w}$ i grafen på følgende måde:

$$\tilde{w}(u, v) = w(u, v) + \phi(u) - \phi(v).$$

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

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

da summen er teleskoperende.

Med andre ord er stilængden under de nye vægte lig stilængden under de gamle, med en additiv korrektion baseret på stiens endepunkter.

Dvs. denne korrektion er *den samme* for alle stier fra s ($= v_1$) til t ($= v_k$).

Så (d)en korteste sti fra s til t er *den samme sti* under både w og $\tilde{w}$.

Endvidere er der en negativ kreds under w hvis og kun hvis der en negativ kreds under $\tilde{w}$ (da $v_k = v_1$ i en kreds, så $\phi(v_k) - \phi(v_1) = 0$).

A^* [Hart, Nilsson, Raphael, 1968]

A^*-algoritmen kan ses som en tunings-metode til Dijkstra for det (ofte forekommende) tilfælde, at man søger efter sti fra s til en *specifik* målknude t:

Ny ingrediens: Forsøg til alle knuder v at lave en *gæt* $h(v)$ på den korteste afstand fra v til t, dvs. et gæt på $\delta(v, t)$. Man kalder også $h(v)$ for en *heuristik*.

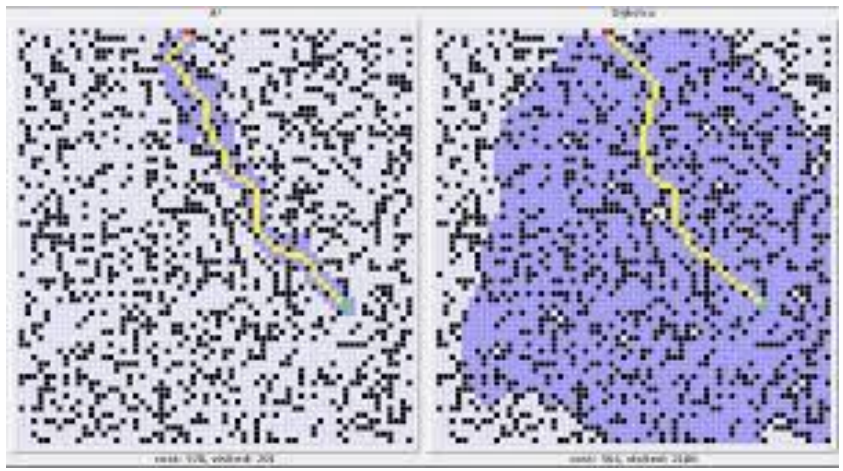
Intuition: hvis $v.d$ (som i Dijkstra, når v udtages af PQ) er lig $\delta(s, v)$, da er $v.d + h(v)$ et gæt på $\delta(s, v) + \delta(v, t)$, hvilket er længden af den korteste vej fra s til t gennem v.

Idé: gå frem som i Dijkstra (inkl. samme update af $v.d$-værdier), men lad nøgle i PQ være $v.d + h(v)$.

Dvs. udvid søgning via knuder, som *gættes* at være på *korteste sti fra s til t*. Til sammenligning kan Dijkstra siges at udvide via knuder, som *vides* at være de *nærmeste til s*.

A^* i praksis

Eksempel med grid-baseret graf. Knuder = de hvide grid-celler, kanter med længde én mellem hvide naboceller. Heuristikken $h(v)$ er lig Euklidisk afstand (fugleflugt) fra celle v til målcellen t.



Dijkstra (højre figur): Undersøger jævnt i alle retninger.

A^* med ovenstående heuristik (venstre figur): Undersøger mere mod målet. Færre knuder besøges, derfor hurtigere i praksis.

Korrekthed og worst case køretid af A^*?

En heuristik kaldes konsistent hvis der for alle knuder v og alle kanter (v, u) i v's naboliste gælder:

$$h(v) \leq w(v, u) + h(u)$$

Dvs. at heuristikkens gæt (på korteste vej til t) for v's naboer ikke er i modstrid med heuristikkens gæt (på korteste vej til t) for v.

Man kan vise, at med en konsistent heuristik er A^* det samme som Dijkstra på en graf med justerede vægte.

Heraf kan man vise korrekthed (at den korteste vej mellem s og t returneres af A^*), og at worst case køretiden er lig worst case køretiden for Dijkstra.