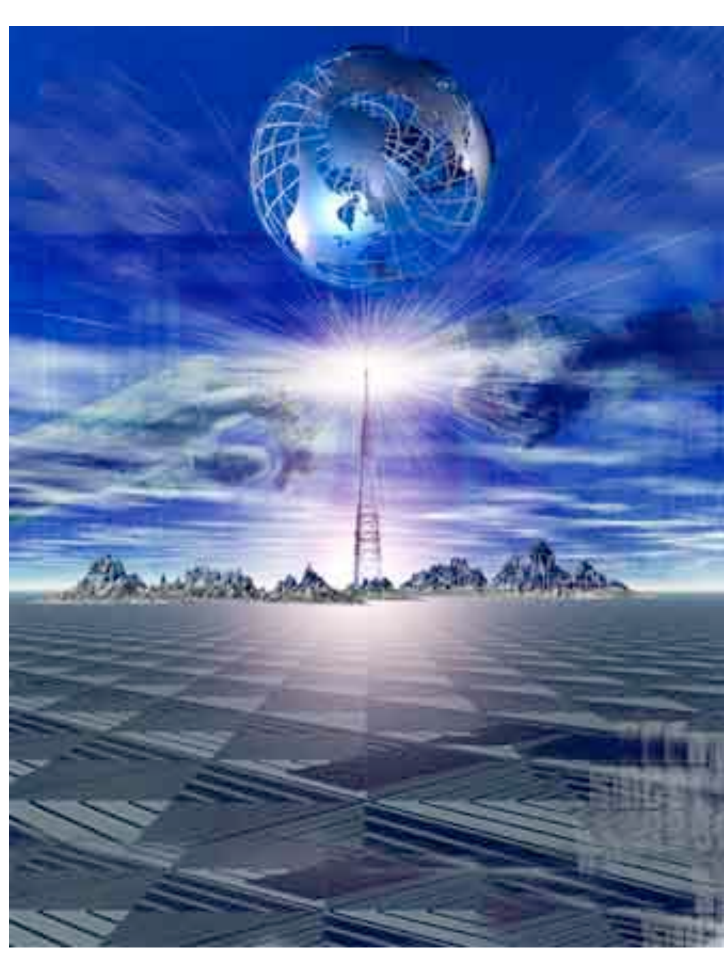
Algoritmedesign

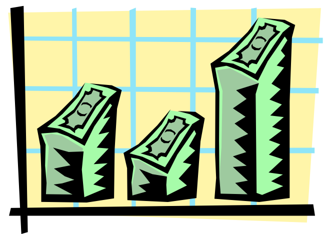


Den grådige metode

Den grådige metode

- Et problem løses ved at foretage en række beslutninger
- Beslutningerne træffes en ad gangen i en eller anden rækkefølge
- Hver beslutning er baseret på et grådighedskriterium
- Når en beslutning er truffet, ændres den (sædvanligvis) ikke senere

Den grådige metode



Den **grådige metode** er et generelt paradigme til algoritmedesign, der bygger på følgende to elementer:

- **Konfigurationer**: forskellige sæt af værdier, der skal findes
- **Objektfunktion**: en score knyttet til konfigurationerne, som vi ønsker at maksimere eller minimere

Metoden virker bedst, når den anvendes på problemer, der har **grådigt-valg-egenskaben**:

> En *globalt* optimal løsning kan altid findes ved en række *lokale* forbedringer ud fra en startkonfiguration

Møntveksling

Problem: Et beløb, der skal opnås, og en samling mønter, der kan bruges for at opnå dette beløb

Konfiguration: Det beløb, der endnu mangler at blive givet til kunden, plus de mønter, der allerede er givet

Objektfunktion: Antallet af mønter (skal minimeres)

Grådig metode: Benyt altid den størst mulige mønt

Eksempel 1: Mønterne er 1, 5, 10 og 25 cent

Har grådigt-valg-egenskaben. Intet beløb over 25 cent kan opnås med et minimalt antal mønter uden brug af 25-cent mønten (tilsvarende for beløb under 25 cent, men over 10 cent, og beløb under 10 cent men, over 5 cent)

Eksempel 2: Mønterne er 1, 5, 10, 21 og 25 cent

Har ikke grådigt-valg-egenskaben. Beløbet 42 cent opnås bedst med to 21 cent mønter, men den grådige algoritme giver en løsning med 5 mønter (hvilke?)

Det fraktionelle rygsækproblem

Givet: En mængde $\boldsymbol{S}$ af $\boldsymbol{n}$ emner, hvor hvert emne $\boldsymbol{i}$ har

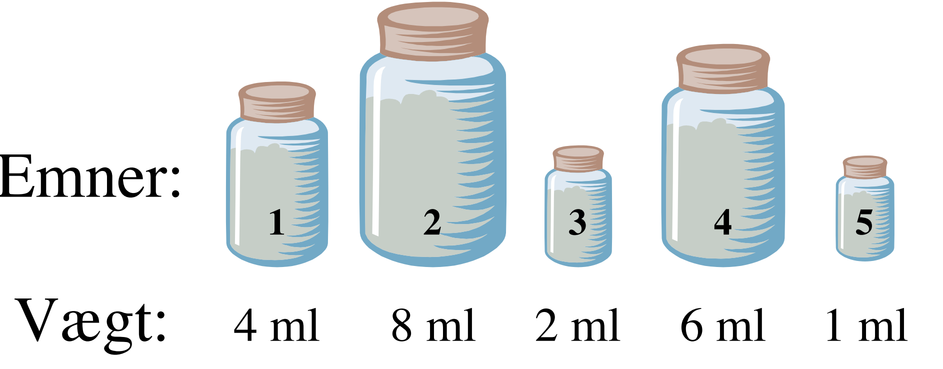
$\boldsymbol{b_i}$ - en positiv nytteværdi (benefit)

$\boldsymbol{w_i}$ - en positiv vægt (weight)

Mål: Vælg emner med maksimal samlet nytteværdi, men med samlet vægt på højst $\boldsymbol{W}$

Hvis det er muligt at tage vilkårlige brøkdele af emnerne, er der tale om **det fraktionelle rygsækproblem**

- Lad $\boldsymbol{x_i}$ betegne den mængde, vi tager af emnet $\boldsymbol{i}$ ($0 \le \boldsymbol{x_i} \le \boldsymbol{w_i}$)
- **Mål**: Maksimer $\sum_{i \in S} b_i (x_i / w_i)$
- **Begrænsning**: $\sum_{i \in S} x_i \le W$



Emner:	1	2	3	4	5
Vægt:	4 ml	8 ml	2 ml	6 ml	1 ml
Nytteværdi:	12 kr	32 kr	40 kr	30 kr	50 kr
Værdi: (kr per ml)	3	4	20	5	50

10 ml

"rygsæk"

Løsning:
- 1 ml af 5
- 2 ml af 3
- 6 ml af 4
- 1 ml af 2

Algoritme for det fraktionelle rygsækproblem

Grådigt valg: Tag altid det emne, der har størst **værdi** (forholdet imellem nytteværdi og vægt)

$$\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$$

Køretid: $O(n \log n)$. Hvorfor?

Korrekthed: Antag, at der en bedre løsning, d.v.s. der findes et emne i med højere værdi end et valgt emne j ($x_j > 0$ og $v_i > v_j$), men hvor $x_i < w_i$. Hvis vi erstatter noget af j med noget af i, kan vi opnå en bedre løsning.

Hvor meget af emne i kan vi erstatte uden at ændre den samlede vægt?

Svar: $\min\{w_i - x_i, x_j\}$

Algorithm *fractionalKnapsack*(***S***, ***W***)

Input: set S of items with benefit b_i and weight w_i; max. weight W

Output: amount x_i of each item i to maximize benefit with weight at most W

```
for each item i in S do
    x_i ← 0
    v_i ← b_i / w_i            {value}
w ← 0                          {total weight}
while w < W do
    remove item i with highest v_i
    x_i ← min{w_i , W - w}
    w ← w + x_i
```

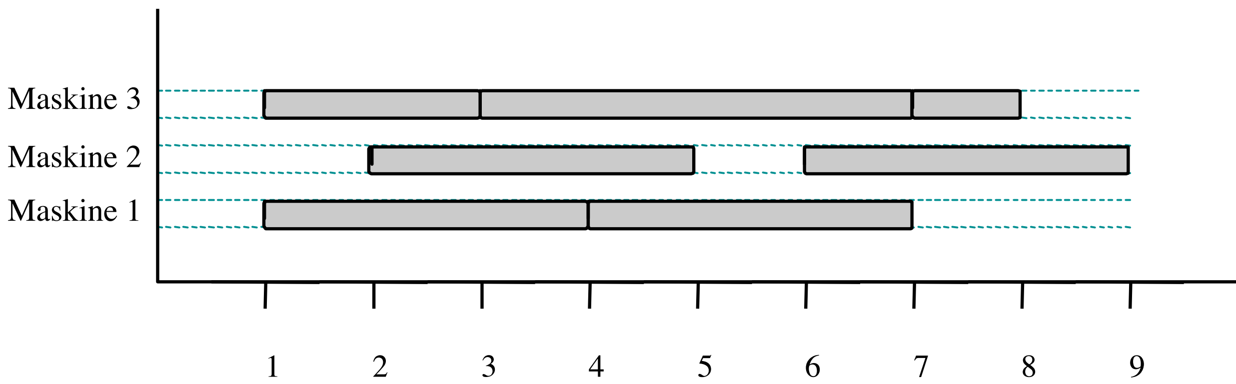
Planlægning af opgaver

Givet: en mængde T af opgaver (tasks), der hver har

et starttidspunkt, s_i

et sluttidspunkt, f_i (hvor $s_i < f_i$)

Mål: Udfør alle opgaverne ved brug af så få maskiner som muligt



Algoritme til opgaveplanlægning

Grådigt valg: ordn opgaverne efter deres starttidspunkt og brug så få maskiner som muligt med denne rækkefølge

Køretid: $O(n \log n)$. Hvorfor?

Korrekthed: Antag, at der findes en bedre plan, d.v.s. algoritmen finder en løsning med m maskiner, men der findes en løsning med m-1 maskiner.

Lad m være den sidste maskine, der allokeres af algoritmen, og lad i være den første opgave, der udføres på denne maskine. Opgave i må være i konflikt med m-1 andre opgaver.

Der findes derfor ingen plan med kun m-1 maskiner.

Algorithm ***taskSchedule***(***T***)
- **Input:** set T of tasks with start time s_i and finish time f_i
- **Output:** non-conflicting schedule with minimum number of machines

```
m ← 0                       {no. of machines}
while T is not empty do
    remove task i with smallest s_i
    if there's a machine j for i then
        schedule i on machine j
    else
        m ← m + 1
        schedule i on machine m
```

Eksempel

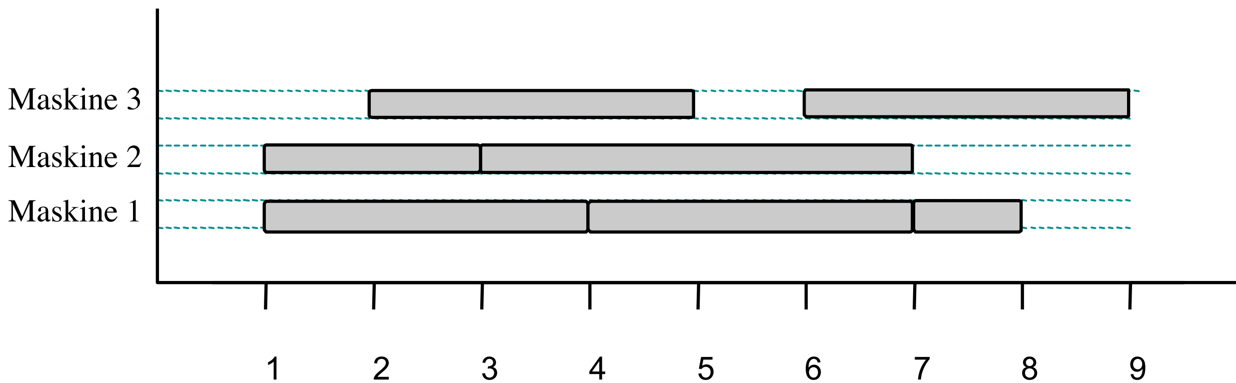
Givet: en mængde *T* af opgaver (tasks), der hver har

et starttidspunkt, s_i

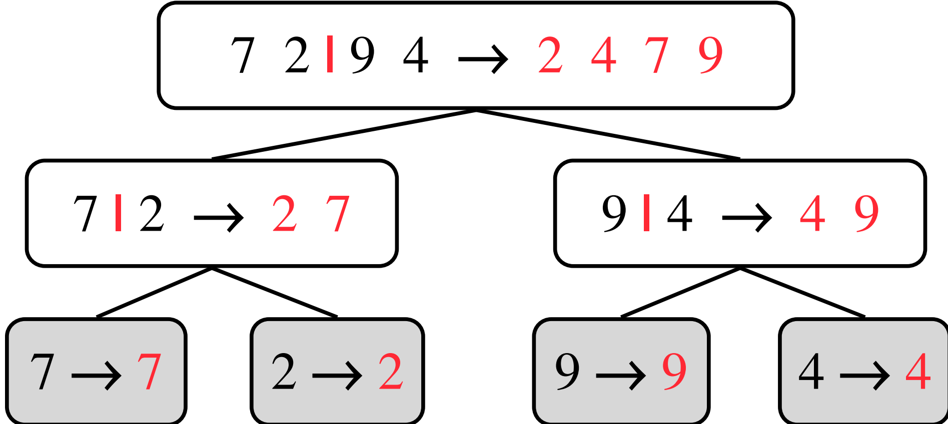
et sluttidspunkt, f_i (hvor $s_i < f_i$)

[1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (ordnet efter starttidspunkt)

Mål: Udfør alle opgaverne ved brug af så få maskiner som muligt



Del-og-hersk



Del-og-hersk

Del-og-hersk er et generelt paradigme til algoritmedesign

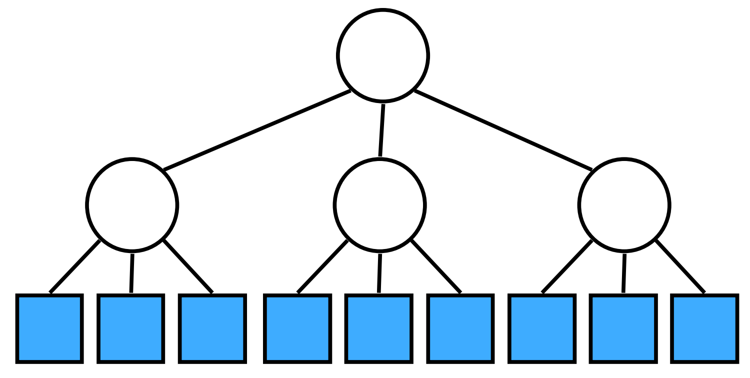
Del: opdel inddata $\boldsymbol{S}$ i to eller flere disjunkte delmængder, $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, ...

Løs: løs delproblemerne rekursivt

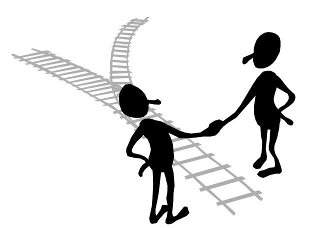
Hersk: kombiner løsningerne for $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, ... til en løsning for $\boldsymbol{S}$

Basistilfældet for rekursionen er delproblemer af "tilpas lille" størrelse

Analyse kan foretages ved hjælp af **rekursionsligninger**



Merge-sort (tilbageblik)



Merge-sort på en input-sekvens $\boldsymbol{S}$ med $\boldsymbol{n}$ elementer består at tre trin:

Del: opdel $\boldsymbol{S}$ i to sekvenser, $\boldsymbol{S}_1$ og $\boldsymbol{S}_2$, hver med cirka $\boldsymbol{n}/2$ elementer

Løs: sorter rekursivt $\boldsymbol{S}_1$ og $\boldsymbol{S}_2$

Hersk: flet $\boldsymbol{S}_1$ og $\boldsymbol{S}_2$ til en sorteret sekvens

Algorithm ***mergeSort***(***S, C***)
  Input sequence $\boldsymbol{S}$ with $\boldsymbol{n}$
    elements, comparator $\boldsymbol{C}$
  Output sequence $\boldsymbol{S}$ sorted
    according to $\boldsymbol{C}$
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    $mergeSort(S_1, C)$
    $mergeSort(S_2, C)$
    $S \leftarrow merge(S_1, S_2)$

Analyse med rekursionsligninger

Hersk-trinet i merge-sort består i fletning af to sorterede sekvenser, hver med $\boldsymbol{n}/2$ elementer

Hvis der benyttes en hægtet liste, kan det gøres med højst $\boldsymbol{bn}$ skridt for en konstant værdi $\boldsymbol{b}$

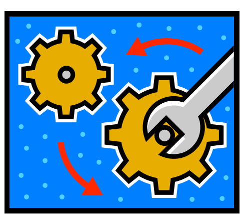
Basistilfældet bruger højst $\boldsymbol{b}$ skridt

Hvis $\boldsymbol{T(n)}$ betegner køretiden for merge-sort, har vi:

$$T(n) = \begin{cases} b & \text{hvis } n < 2 \\ 2T(n/2) + bn & \text{hvis } n \geq 2 \end{cases}$$

Vi kan derfor analysere køretiden for merge-sort ved at finde en løsning på **lukket form** for denne ligning (d.v.s. en løsning, hvor $\boldsymbol{T(n)}$ kun forekommer på venstre side af lighedstegnet)

Iterativ substitution



Ved teknikken **iterativ substitution** anvender vi gentagne gange rekursionsligningen på sig selv for at finde et mønster:

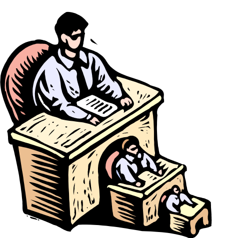
$$\begin{aligned}
T(n) &= 2T(n/2) + bn \\
&= 2(2T(n/2^2) + b(n/2)) + bn \\
&= 2^2 T(n/2^2) + 2bn \\
&= 2^3 T(n/2^3) + 3bn \\
&= 2^4 T(n/2^4) + 4bn \\
&= \ldots \\
&= 2^i T(n/2^i) + ibn
\end{aligned}$$

Basistilfældet indtræffer, når $2^i = n$, d.v.s. $i = \log n$

$$T(n) = bn + bn \log n$$

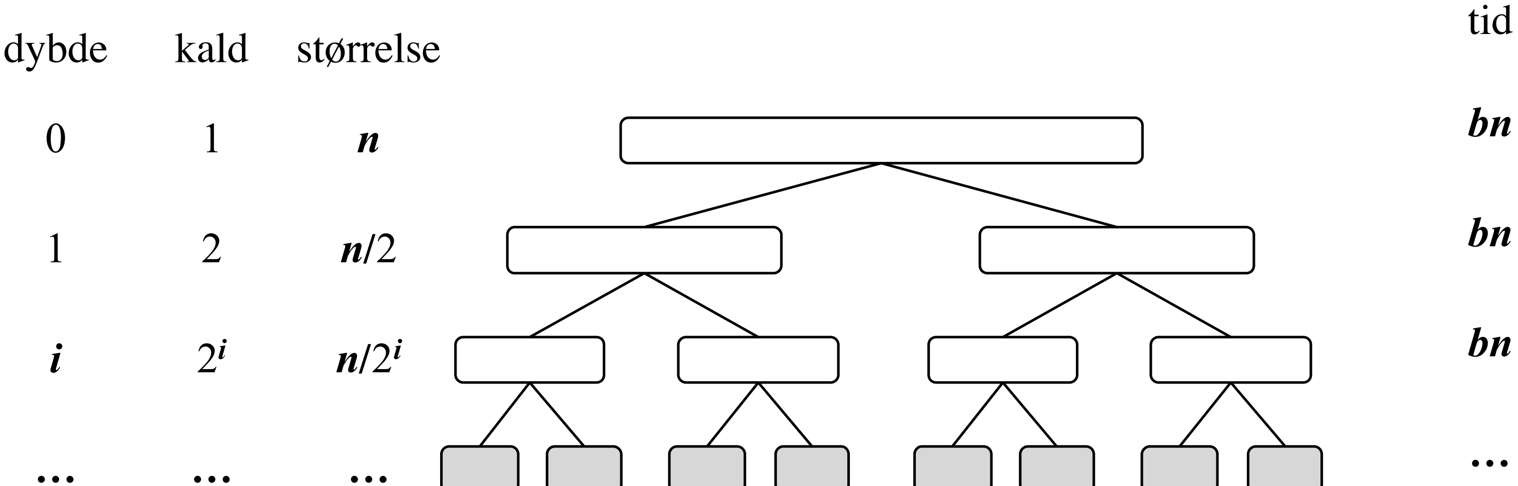
Vi kan derfor konkludere, at $T(n)$ er $O(n \log n)$

Rekursionstræ



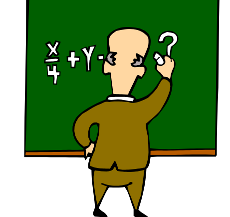
Tegn rekursionstræet for rekursionsligningen og led efter et mønster:

$$T(n) = \begin{cases} b & \text{hvis } n < 2 \\ 2T(n/2) + bn & \text{hvis } n \geq 2 \end{cases}$$



Samlet tid = ***bn*** + ***bn*** log ***n***
(sidste niveau plus alle foregående niveauer)

Gæt-og-test-metoden



Ved gæt og test-metoden gætter vi på en lukket form og prøver at bevise, at den er sand ved hjælp af induktion:

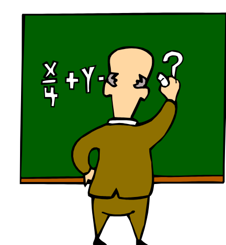
$$T(n) = \begin{cases} b & \text{hvis } n < 2 \\ 2T(n/2) + bn\log n & \text{hvis } n \geq 2 \end{cases}$$

Gæt: $T(n) < cn \log n$

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &< 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

Forkert: vi kan ikke vise, at udtrykket i den sidste linje er mindre end $cn \log n$

Gæt-og-test-metoden (del 2)



$$T(n) = \begin{cases} b & \text{hvis } n < 2 \\ 2T(n/2) + bn\log n & \text{hvis } n \geq 2 \end{cases}$$

Gæt 2: $T(n) < cn \log^2 n$

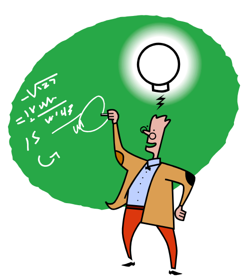
$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &< 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n \end{aligned}$$

hvis $c > b$

Så $T(n)$ er $O(n \log^2 n)$

For at bruge denne metode skal man være god til at gætte og god til at føre induktionsbeviser

Mestermetoden



Mange del-og-hersk rekursionsligninger har formen

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \Rightarrow T(n)$ er $\Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)$ er $\Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \Rightarrow T(n)$ er $\Theta(f(n))$

   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)$ for $n \geq d$

Mestermetoden, eksempel 1

Form

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \Rightarrow T(n)$ er $\Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)$ er $\Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \Rightarrow T(n)$ er $\Theta(f(n))$

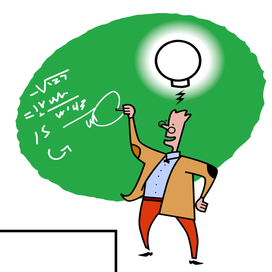
   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)$ *for* $n \geq d$

Eksempel:

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

Løsning: $\log_b a = 2$. Tilfælde 1 siger, at $T(n)$ er $\Theta(n^2)$

Mestermetoden, eksempel 2



Form

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \Rightarrow T(n)$ *er* $\Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)$ *er* $\Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \Rightarrow T(n)$ *er* $\Theta(f(n))$

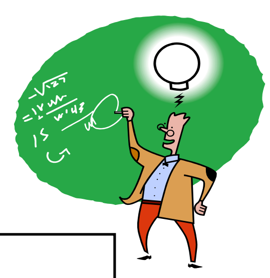
   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)$ *for* $n \geq d$

Eksempel:

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

Løsning: $\log_b a = 1$. Tilfælde 2 siger, at $T(n)$ er $\Theta(n \log^2 n)$

Mestermetoden, eksempel 3



Form

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \Rightarrow T(n)$ er $\Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)$ er $\Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \Rightarrow T(n)$ er $\Theta(f(n))$

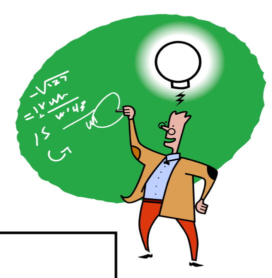
   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)$ *for* $n \geq d$

Eksempel:

$$T(n) = T(n/3) + n \log n$$

Løsning: $\log_b a = 0$. Tilfælde 3 siger, at $T(n)$ er $\Theta(n \log n)$

Mestermetoden, eksempel 4



Form

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \quad \Rightarrow T(n)\ er\ \Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)\ er\ \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \quad \Rightarrow T(n)\ er\ \Theta(f(n))$

   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)\ \ for\ n \geq d$

Eksempel:

$$T(n) = 8T(n/2) + n^2$$

Løsning: $\log_b a = 3$. Tilfælde 1 siger, at $T(n)$ is $\Theta(n^3)$

Mestermetoden, eksempel 5

Form

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \quad \Rightarrow T(n)\ er\ \Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)\ er\ \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \quad \Rightarrow T(n)\ er\ \Theta(f(n))$

   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)\ for\ n \geq d$

Eksempel:

$$T(n) = 9T(n/3) + n^3$$

Løsning: $\log_b a = 3$. Tilfælde 3 siger, at $T(n)$ is $\Theta(n^3)$

Mestermetoden, eksempel 6

Form

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \quad \Rightarrow T(n)$ *er* $\Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)$ *er* $\Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \quad \Rightarrow T(n)$ *er* $\Theta(f(n))$

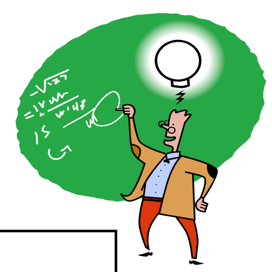
   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)$ *for* $n \geq d$

Eksempel:

$T(n) = T(n/2) + 1$ (binær søgning)

Løsning: $\log_b a = 0$. Tilfælde 2 siger, at $T(n)$ is $\Theta(\log n)$

Mestermetoden, eksempel 7



Form

$$T(n) = \begin{cases} c & \text{hvis } n < d \\ aT(n/b) + f(n) & \text{hvis } n \geq d \end{cases}$$

Mestersætningen:

1. $f(n)$ er $O(n^{\log_b a - \varepsilon}) \quad \Rightarrow T(n)\ er\ \Theta(n^{\log_b a})$
2. $f(n)$ er $\Theta(n^{\log_b a} \log^k n) \Rightarrow T(n)\ er\ \Theta(n^{\log_b a} \log^{k+1} n)$
3. $f(n)$ er $\Omega(n^{\log_b a + \varepsilon}) \quad \Rightarrow T(n)\ er\ \Theta(f(n))$

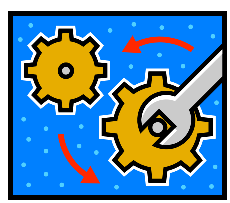
   forudsat $\exists \delta < 1 : af(n/b) \leq \delta f(n)\ for\ n \geq d$

Eksempel:

$$T(n) = 2T(n/2) + \log n \qquad \text{(hob-konstruktion)}$$

Løsning: $\log_b a = 1$. Tilfælde 1 siger, at $T(n)$ is $\Theta(n)$

Iterativt “bevis” for mestersætningen



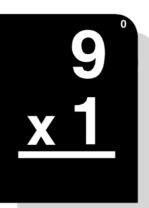
Ved hjælp af iterativ substitution findes et mønster:

$$
\begin{aligned}
T(n) &= aT(n/b) + f(n) \\
&= a(aT(n/b^2)) + f(n/b)) + bn \\
&= a^2T(n/b^2) + af(n/b) + f(n) \\
&= a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \\
&= \ldots \\
&= a^{\log_b n}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i) \\
&= n^{\log_b a}T(1) + \sum_{i=0}^{(\log_b n)-1} a^i f(n/b^i)
\end{aligned}
$$

Vi skelner imellem tre tilfælde:

1. Det første led i summen er dominerende
2. Ledene i summationen er lige dominerende
3. Summationen er en kvotientrække

Heltalsmultiplikation



Algoritme: Multiplicer to n-bit heltal I og J

- Opdeling: Opsplit I og J i højordens- og lavordens-bits:

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

I_h	I_l
J_h	J_l

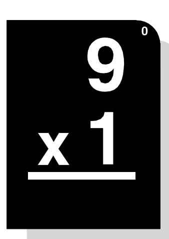
- Vi kan så bestemme $I*J$ ved at multiplicere delene og addere:

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$

$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

- $T(n) = 4T(n/2) + cn$, hvilket medfører, at $T(n)$ er $\Theta(n^2)$
- Men det er ikke bedre end den algoritme, vi lærte i skolen

Forbedret algoritme



Algoritme: Multiplicer to n-bit heltal I og J

- Opdeling: Opsplit I og J i højordens- og lavordens-bits:

$$I = I_h 2^{n/2} + I_l$$
$$J = J_h 2^{n/2} + J_l$$

I_h	I_l
J_h	J_l

- Der findes en anden måde at multiplicere delene:

$$\begin{aligned} I * J &= I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l \\ &= I_h J_h 2^n + [(I_h J_l - I_l J_l - I_h J_h + I_l J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l \\ &= I_h J_h 2^n + (I_h J_l + I_l J_h) 2^{n/2} + I_l J_l \end{aligned}$$

- $T(n) = 3T(n/2) + cn$, hvilket medfører, at $T(n)$ er $\Theta(n^{\log_2 3})$
- $T(n)$ er $\Theta(n^{1.585})$

Med FFT kan opnås en $\Theta(n \log n)$ algoritme

Dynamisk programmmering

Dynamisk programmering

Del-og-hersk (top-til-bund, top-down):

For at løse et stort problem deles problemet op i mindre delproblemer, der løses uafhængigt af hinanden

Dynamisk programmering (bund-til-top, bottom-up):

For at løse et stort problem løses <u>alle</u> mindre delproblemer, og deres løsninger gemmes og benyttes til at løse større problemer. Således fortsættes, indtil problemet er løst

Betegnelsen stammer fra operationsanalysen, hvor "programmering" benyttes om *formulering* af et problem, således at en bestemt metode kan anvendes

Moderne definition

Dynamisk programmering:

Bund-til-top implementering af rekursive programmer med overlappende delproblemer

Top-til-bund implementering er dog også mulig

Dynamisk programmering er baseret på følgende simple princip:

Undgå at gentage beregninger

Beregning af Fibonacci-tal

(Fibonacci, 1202)

F(n) = F(n-1) + F(n-2) for n >= 2

F(0) = 0

F(1) = 1

n	F(n)
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34

n	F(n)
10	55
11	89
12	144
13	233
14	377
15	610
16	987
17	1597
18	2584
19	4181

n	F(n)
20	6765
21	10946
22	17711
23	28657
24	46368
25	75025
26	121393
27	196418
28	317811
29	514229

Talrækken vokser eksponentielt:

F(n)/F(n-1) går imod 1.618... (det gyldne snit = $(1+\sqrt{5})/2$)

Top-til-bund-tilgang

```
int F(int n) {
    return n <= 1 ? n : F(n-1) + F(n-2);
}
```

Simpel, men meget ineffektiv
Antallet af kald, $C(n)$, tilfredsstiller rekursionsligningerne

$C(n) = C(n\text{-}1) + C(n\text{-}2) + 1,$
$C(1) = C(0) = 1$

som har løsningen $C(n) = \mathrm{F}(n+2) + \mathrm{F}(n\text{-}1) - 1$.
$C(n)$ er altså større end det Fibonacci-tal, der skal beregnes!

Ineffektiviteten skyldes, at de samme delproblemer løses mange gange

F.eks. F(9) = F(8) + F(7) = <u>F(7)</u> + F(6) + <u>F(7)</u> =
<u>F(6)</u> + F(5) + <u>F(6)</u> + <u>F(6)</u> + F(5)

Undgå genberegninger

(benyt “caching”)

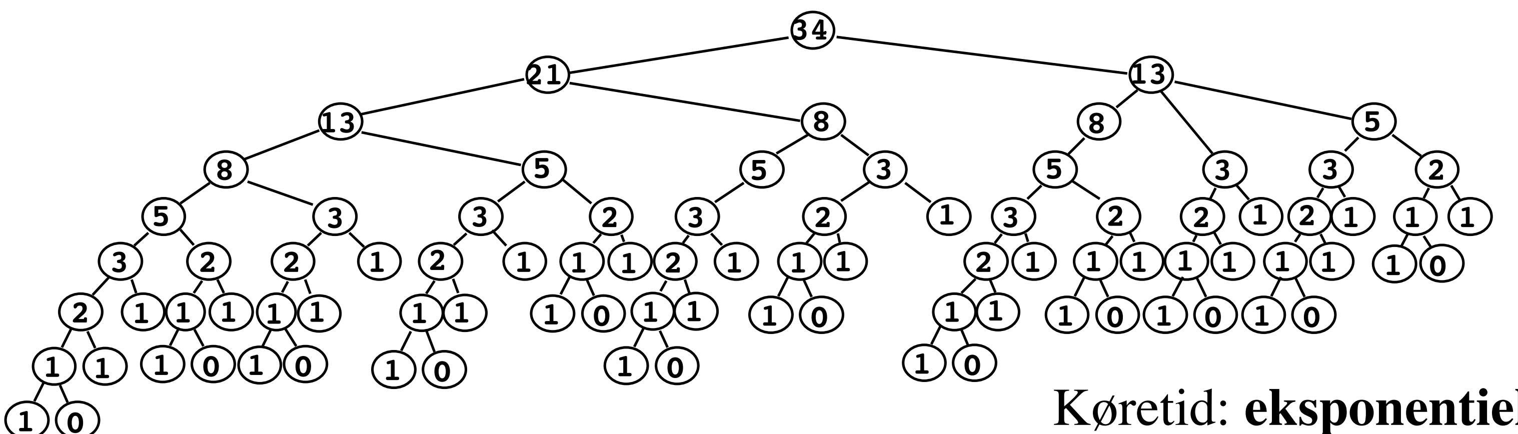
Vedligehold en tabel (indiceret ved parameterværdien) indeholdende
* 0, hvis den rekursive metode endnu ikke er kaldt med denne parameterværdi
* ellers det resultat, der skal returneres

Første kald af metoden for en given parameterværdi: beregn som før, men gem desuden resultatet
Efterfølgende kald med samme parameterværdi: returner resultatet fra det første kald

```
int F(int n) {
    if (Fknown[n] != 0)
        return Fknown[n];
    int r = n <= 1 ? n : F(n-1) + F(n-2);
    Fknown[n] = r;
    return r;
}
```

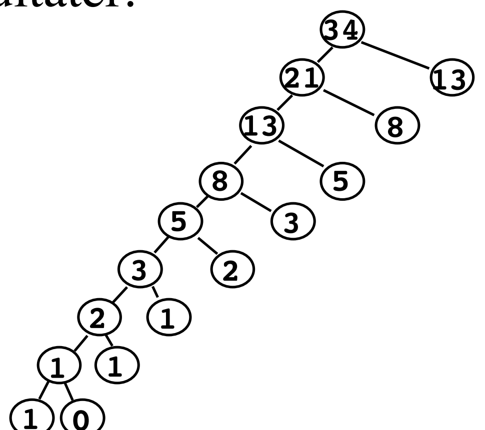
Effektivitetsforbedring

Simpel rekursiv metode: F(9)



Køretid: **eksponentiel**

Husk kendte resultater:



Køretid: **lineær**

Bund-til-top-tilgang

Dynamisk programmering (traditionel):

* Tabellæg løsningerne til delproblemerne
* Opbyg tabellen i stigende orden af problemstørrelse
* Benyt gemte løsninger til at bestemme nye løsninger

Eksempel: Dynamisk programmering til beregning
af Fibonacci-tal:

```
F[0] = 0; F[1] = 1;
for (i = 2; i <= n; i++)
    F[i] = F[i-1] + F[i-2];
```

Tidsforbruget er lineært

Optimal møntveksling

Udbetal et beløb i mønter, således at antallet af mønter er minimalt

Eksempel:

Hvis beløbet skal udbetales i amerikanske cents, kan mønterne 1-, 5-, 10- og 25-cent benyttes
Veksling af 63 cents kan da foretages med **6** mønter, nemlig **to** 25-cent, **en** 10-cent og **tre** 1-cent

For disse mønter vil en **grådig** algoritme altid give en optimal løsning
Men hvis der indføres en 21-cent-mønt, virker denne metode ikke

Top-til-bund-løsning

Beregn for hver mønt det minimale antal mønter, der kan benyttes til at veksle det resterende beløb. Tag minimum

```
int[] coins = {1, 5, 10, 21, 25};
```

```
int makeChange(int change) {
    if (change == 0)
        return 0;
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < coins.length; i++)
        if (change >= coins[i])
            min = Math.min(min,
                    1 + makeChange(change - coins[i]));
    return min;
}
```

Benyt **ikke** denne algoritme! Eksponentielt tidsforbrug.
Undgå genberegninger.

Genbrug kendte løsninger

```
int makeChange(int change) {
    if (change <= 0)
        return 0;
    if (minKnown[change] > 0)
        return minKnown[change];
    int min = Integer.MAX_VALUE;
    for (int i = 0; i < coins.length; i++)
        if (change >= coins[i])
            min = Math.min(min,
                  1 + makeChange(change - coins[i]));
    minKnown[change] = min;
    return min;
}
```

Udskrivning af mønterne i en optimal veksling

Gem i en tabel, **`lastCoin`**, for ethvert beløb den sidst valgte mønt i en optimal veksling af beløbet

```
while (change > 0) {
    System.out.println(lastCoin[change]);
    change -= lastCoin[change];
}
```

```
int makeChange(int change) {
    if (change <= 0)
        return 0;
    if (minKnown[change] > 0)
        return minKnown[change];
    int min = Integer.MAX_VALUE, minCoin = 0;
    for (int i = 0; i < coins.length; i++)
        if (change >= coins[i]) {
            int m = 1 + makeChange(change - coins[i]);
            if (m < min)
              { min = m; minCoin = coins[i]; }
        }
    lastCoin[change] = minCoin;
    minKnown[change] = min;
    return min;
}
```

(uden rekursion)

Brug fundne løsninger til at bestemme den næste løsning

```
int makeChange(int change) {
    minKnown[0] = 0;
    for (int c = 1; c <= change; c++) {
        int min = Integer.MAX_VALUE;
        for (int i = 0; i < coins.length; i++)
            if (c >= coins[i])
                min = Math.min(min,
                          1 + minKnown[change - coins[i]]);
        minKnown[c] = min;
    }
    return minKnown[change];
}
```

Køretiden er proportional med **`change*coins.length`**

Matrix-kædeprodukter

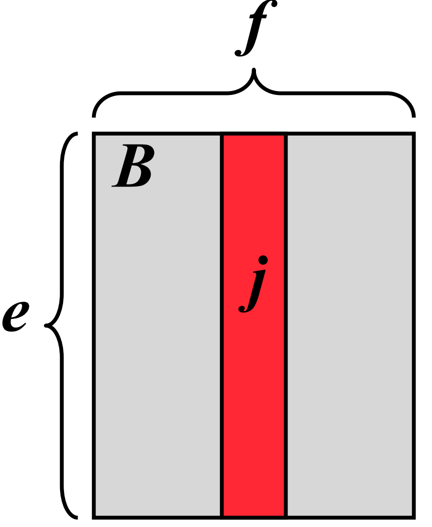
Dynamisk programmering er et generelt paradigme til algoritmedesign

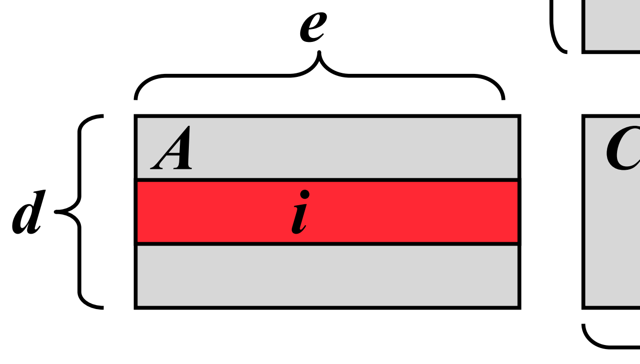
Matrixmultiplikation:

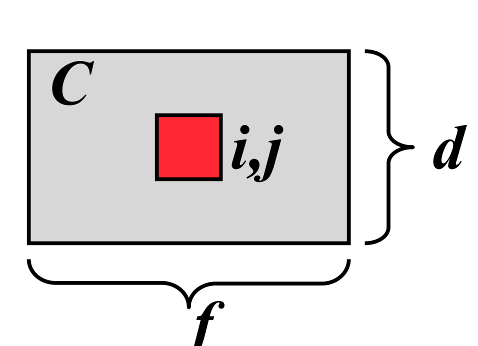
- ***C*** = ***A********B***
- ***A*** er ***d*** x ***e***, og ***B*** er ***e*** x ***f***

$$C[i,j] = \sum_{k=0}^{e-1} A[i,k] * B[k,j]$$

- Køretiden er ***O***(***def***)







Matrix-kædeprodukter

Matrix-kædeprodukt:

Beregn $A = A_0 * A_1 * \ldots * A_{n-1}$

A_i er d_i x d_{i+1}

Problem: Hvor skal parenteserne placeres?

Eksempel:

B er 3 x 100

C er 100 x 5

D er 5 x 5

(B*C)*D bruger 1500 + 75 = 1575 multiplikationer

B*(C*D) bruger 1500 + 2500 = 4000 multiplikationer

Et større eksempel

A er en 13 x 5 matrix

B er en 5 x 89 matrix

C er en 89 x 3 matrix

D er en 3 x 34 matrix

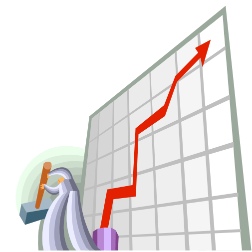
((AB)C)D :

AB :	13*5*89 = 5785 multiplikationer
(AB)C :	13*89*3 = 3471 multiplikationer
((AB)C)D :	13*3*34 = 1326 multiplikationer
i alt	10582 multiplikationer

Nogle andre muligheder:

(AB)(CD) :	54201 multiplikationer	
A((BC)D) :	4055 multiplikationer	
A(B(CD)) :	26418 multiplikationer	
(A(BC))D :	**2856** multiplikationer	**(optimum)**

En opregnende metode



Matrix-kædeprodukt algoritme:

- Prøv alle mulig måder at sætte parenteser i
  $A_0*A_1*\ldots*A_{n-1}$
- Beregn antallet af operationer for hver mulighed
- Vælg den, der er bedst

Køretid: Antallet af muligheder er lig med antallet af binære træer med n knuder

Dette tal vokser **eksponentielt**! Det kaldes for det *catalanske tal* og er næsten 4^n $(\Omega(4^n / n^{3/2}))$

Dette er er en rædselsfuld algoritme!

En grådig metode

Ide 1: Vælg produkterne et ad gangen. Vælg altid det produkt, der (for)bruger flest multiplikationer

Modeksempel:

A er 10 x 5
B er 5 x 10
C er 10 x 5
D er 5 x 10

Ide 1 giver (A*B)*(C*D),
som bruger 500+1000+500 = 2000 multiplikationer

Men A*((B*C)*D) bruger færre,
nemlig 500+250+250 = 1000 multiplikationer

En anden grådig metode

Ide 2: Vælg produkterne et ad gangen. Vælg altid det produkt, der kræver færrest multiplikationer

Modeksempel:

A is 101 x 11
B is 11 x 9
C is 9 x 100
D is 100 x 99

Ide 2 giver A*((B*C)*D),
som bruger 109989+9900+108900=228789 multiplikationer

Men (A*B)*(C*D) bruger færre,
nemlig 9999+89991+89100=189090 multiplikationer

En rekursiv metode

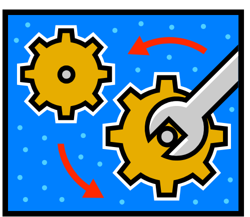
Definer **delproblemer**:

- Find den **bedste** parentesplacering for $A_i*A_{i+1}*\ldots*A_j$
- Lad $N_{i,j}$ betegne antallet af multiplikationer, der foretages for dette delproblem
- Det minimale antal multiplikationer for hele problemet er $N_{0,n-1}$

Optimalitet af delproblemer: Den optimale løsning kan defineres i termer af optimale delproblemer

- Der må nødvendigvis være en sidste multiplikation for den optimale løsning (rod i udtrykstræet)
- Antag at det sidste produkt sker ved indeks i: $(A_0*\ldots*A_i)*(A_{i+1}*\ldots*A_{n-1})$
- Da er den optimale løsning $N_{0,n-1}$ lig med summen af $N_{0,i}$ og $N_{i+1,n-1}$, plus antal multiplikationer for den sidste matrixmultiplikation

En karakteriserende ligning



Det globale optimum må være defineret i termer af optimale delproblemer, der afhænger af, hvor den sidste multiplikation foretages

Prøv alle mulige placeringer for den sidste multiplikation:

- Der mindes om, at A_i er en d_i x d_{i+1} matrix
- En karakteriserende ligning for $N_{i,j}$ er:

$$N_{i,j} = \min_{i \leq k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

Bemærk, at delproblemerne ikke er uafhængige - **delproblemerne overlapper**

En dynamisk programmeringsalgoritme

Da delproblemerne overlapper, bruger vi ikke rekursion

I stedet for konstrueres en optimal løsning "bottom-up"

Værdierne $N_{i,i}$ er lette at bestemme, så dem starter vi med

Så beregnes værdier for delproblemer med længde 2, 3, ... o.s.v

Køretid: $O(n^3)$

Algorithm ***matrixChain***(***S***):

Input: sequence ***S*** of ***n*** matrices to be multiplied

Output: number of operations in an optimal paranethization of ***S***

for ***i*** ← 1 **to** ***n-1*** **do**

$N_{i,i} \leftarrow 0$

for ***b*** ← 1 **to** ***n-1*** **do**

for ***i*** ← 0 **to** ***n-b-1*** **do**

j ← ***i+b***

$N_{i,j} \leftarrow$ **+infinity**

for ***k*** ← ***i*** **to** ***j-1*** **do**

$N_{i,j} \leftarrow \min\{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

Visualisering af algoritmen

$$N_{i,j} = \min_{i \le k < j} \{ N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \}$$

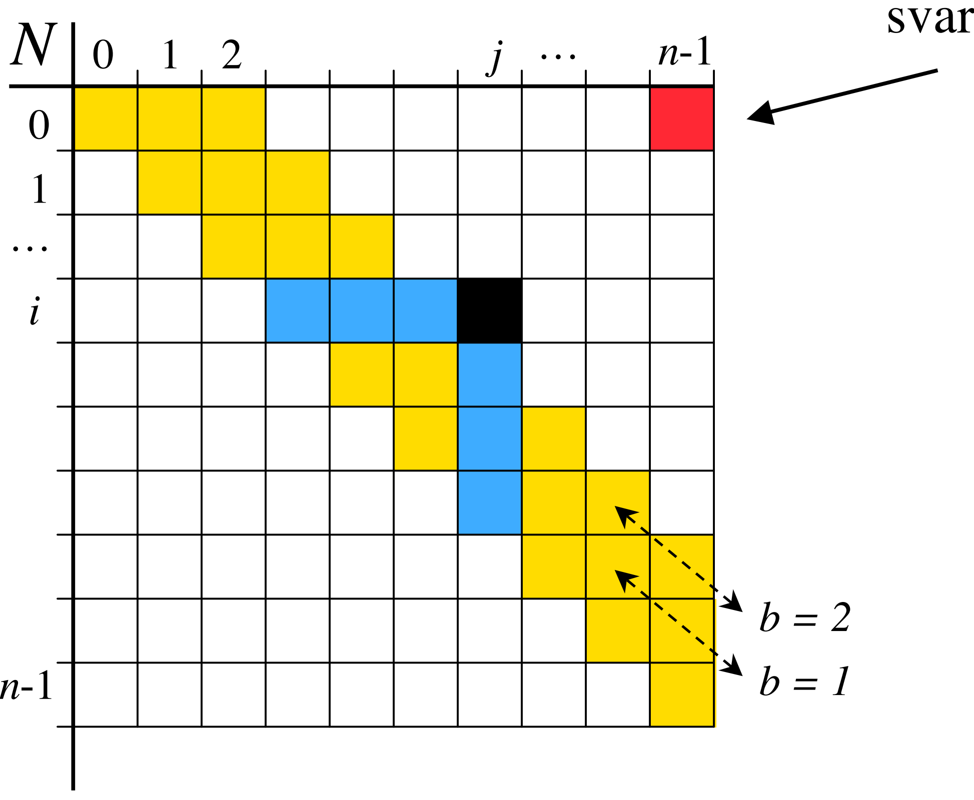
Arrayet N fyldes diagonalvis

$N_{i,j}$ får værdi fra tidligere indgange i den i'te række og j'te søjle

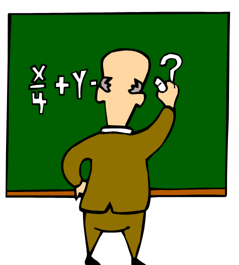
Beregning af hvert element i tabellen tager $O(n)$ tid

Køretid: $O(n^3)$

Den fundne parentesplacering kan gøres tilgængelig ved at huske k's værdi for hvert element i N



Den generelle teknik til dynamisk programmering



Anvendes på et problem, der først synes at kræve en masse tid (muligvis eksponentiel), forudsat at vi har

- **Simple delproblemer**: delproblemer kan defineres i termer af få variable, såsom i, j, k, l, m o.s.v.

- **Delproblem-optimalitet**: det globale optimum kan defineres i termer af optimale delproblemer

- **Delproblem-overlap**: delproblemerne er ikke uafhængige, men overlapper (og skal derfor konstrueres “bottom-up”)

0-1-rygsækproblemet

Givet: En mængde ***S*** af ***n*** emner, hvor hvert emne har

b_i - en positiv nytteværdi (benefit)

w_i - en positiv vægt (weight)

Mål: Vælg emner med maksimal samlet nytteværdi, men med samlet vægt på højst ***W***

Hvis det **ikke** er muligt at tage vilkårlige brøkdele af emnerne, er der tale om **0-1-rygsækproblemet**

Lad ***T*** betegne mængden af de emner, vi vælger

- **Mål**: Maksimer $\sum_{i \in T} b_i$
- **Begrænsning**: $\sum_{i \in T} w_i \leq W$

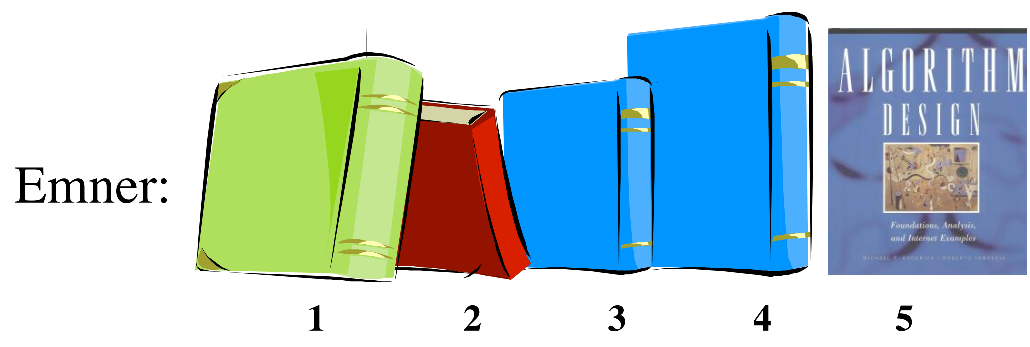
Eksempel

Givet: En mængde $\boldsymbol{S}$ af $\boldsymbol{n}$ emner, hvor hvert emne har

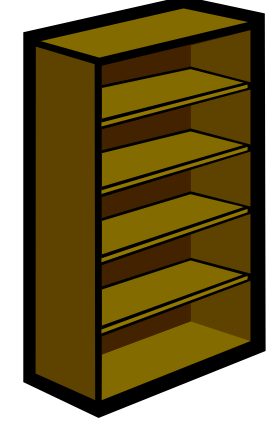
- $\boldsymbol{b_i}$ - en positiv nytteværdi (benefit)
- $\boldsymbol{w_i}$ - en positiv vægt (weight)

Mål: Vælg emner med maksimal samlet nytteværdi, men med samlet vægt på højst $\boldsymbol{W}$

Emner:



	1	2	3	4	5
Vægt:	8 cm	4 cm	4 cm	12 cm	4 cm
Nytteværdi:	20 kr	3 kr	6 kr	25 kr	80 kr



18 cm

"rygsæk"

Løsning:

- 5 (4 cm)
- 3 (4 cm)
- 1 (8 cm)

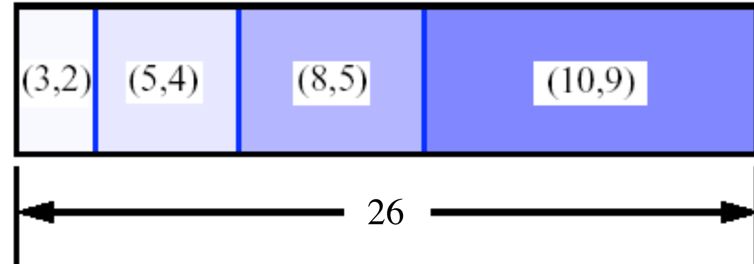
En algoritme (første forsøg)

- Lad S_k være mængden af de emner i S, der er nummereret fra 1 til k
- Definer $B[k]$ = værdi af bedste delmængde af S_k
- Der er **ikke** delproblem-optimalitet:

  Betragt S={(3,2), (5,4), (8,5), (4,3), (10,9)} vægt-nytteværdi par

Bedste for S_4:

Bedste for S_5:



En algoritme (andet forsøg)

- Lad S_k være mængden af de emner i S, der er nummereret fra 1 til k
- Definer $B[k,w]$ = værdi af bedste delmængde af S_k med vægt w
- Der **er** delproblem-optimalitet

Den bedste delmængde af S_k med vægt w er enten
den bedste delmængde af $S_{k\text{-}1}$ med vægt w, eller
den bedste delmængde af $S_{k\text{-}1}$ med vægt $w\text{-}w_k$, plus emnet k:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{hvis } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{ellers} \end{cases}$$

Algoritmen for 0-1-rygsækproblemet

$$B[k,w] = \begin{cases} B[k-1,w] & \text{hvis } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{ellers} \end{cases}$$

Da $B[k,w]$ er defineret i termer af $B[k\text{-}1,*]$, kan vi bruge et endimensionalt array

Køretid: $O(nW)$

Er ikke en polynomiel algoritme, hvis W er stor

Der er tale om en såkaldt **pseudo-polynomiel** algoritme

Algorithm ***01Knapsack***(***S, W***):

Input: set S of items w/ benefit b_i and weight w_i; max. weight W

Output: benefit of best subset with weight at most W

for $w \leftarrow 0$ **to** W **do**

  $B[w] \leftarrow 0$

for $k \leftarrow 1$ **to** n **do**

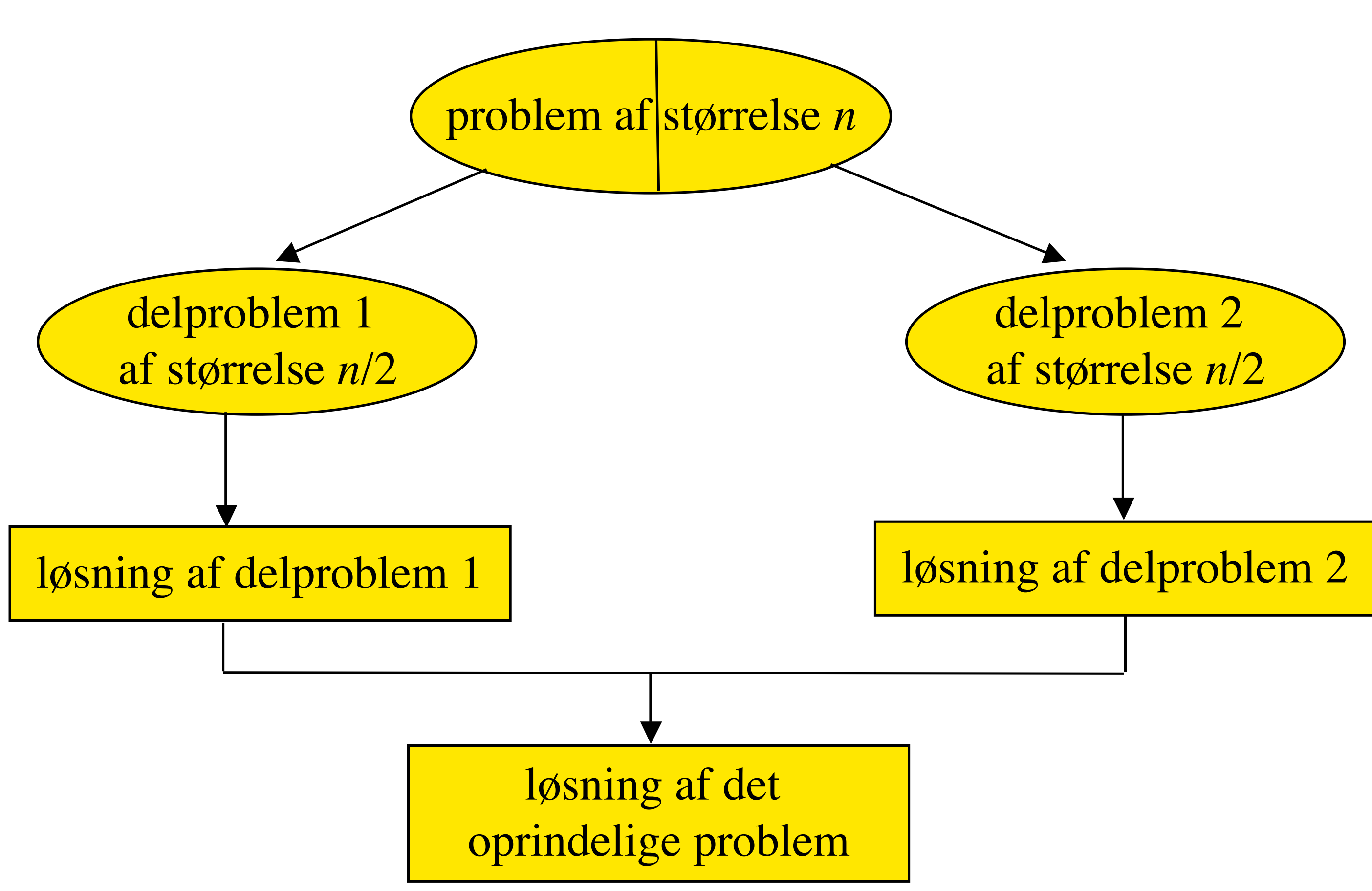
  for $w \leftarrow w_k$ **to** W **do**

    $B[w] = \max\{B[w], B[w-w_k]+b_k\}$

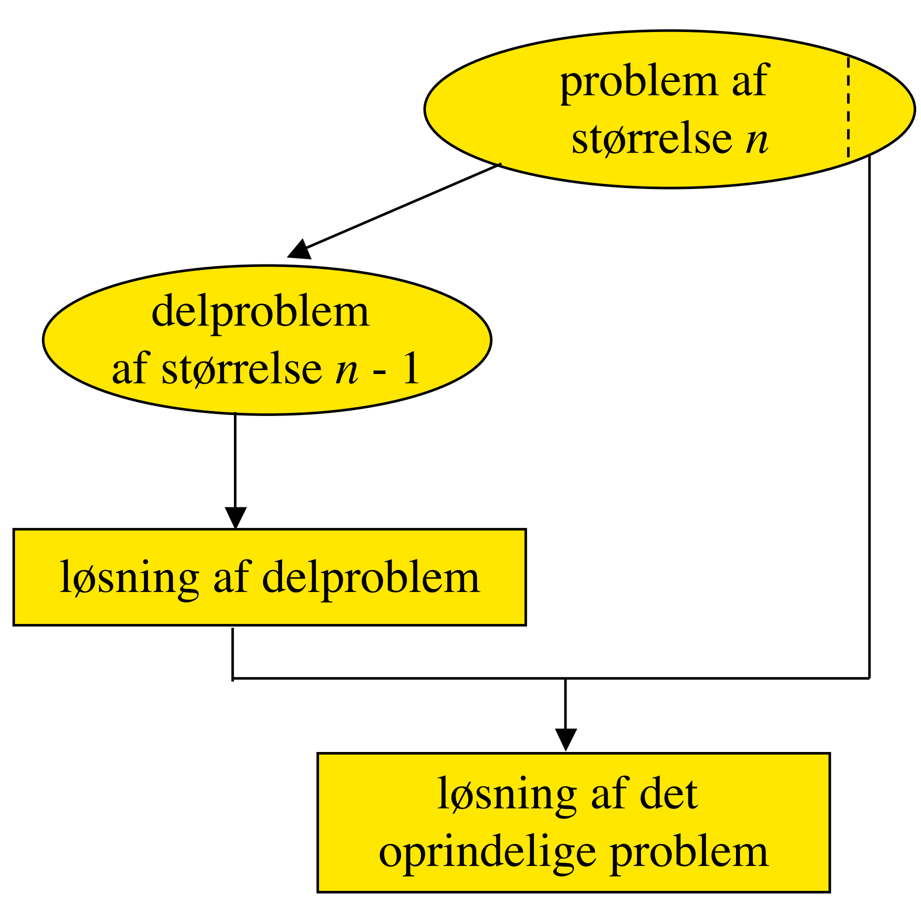
Fejl i lærebogen

Designteknikker

Del-og-hersk



Formindsk-og-hersk



Transformer-og-hersk