Forelesning 6

Dynamisk programmering

1. **Eksempel: Stavkapping**
2. **Dyn. prog. › hva er det?**
3. **Eksempel: LCS**
4. **Optimal delstruktur**
5. **Eksempel: Ryggsekk**

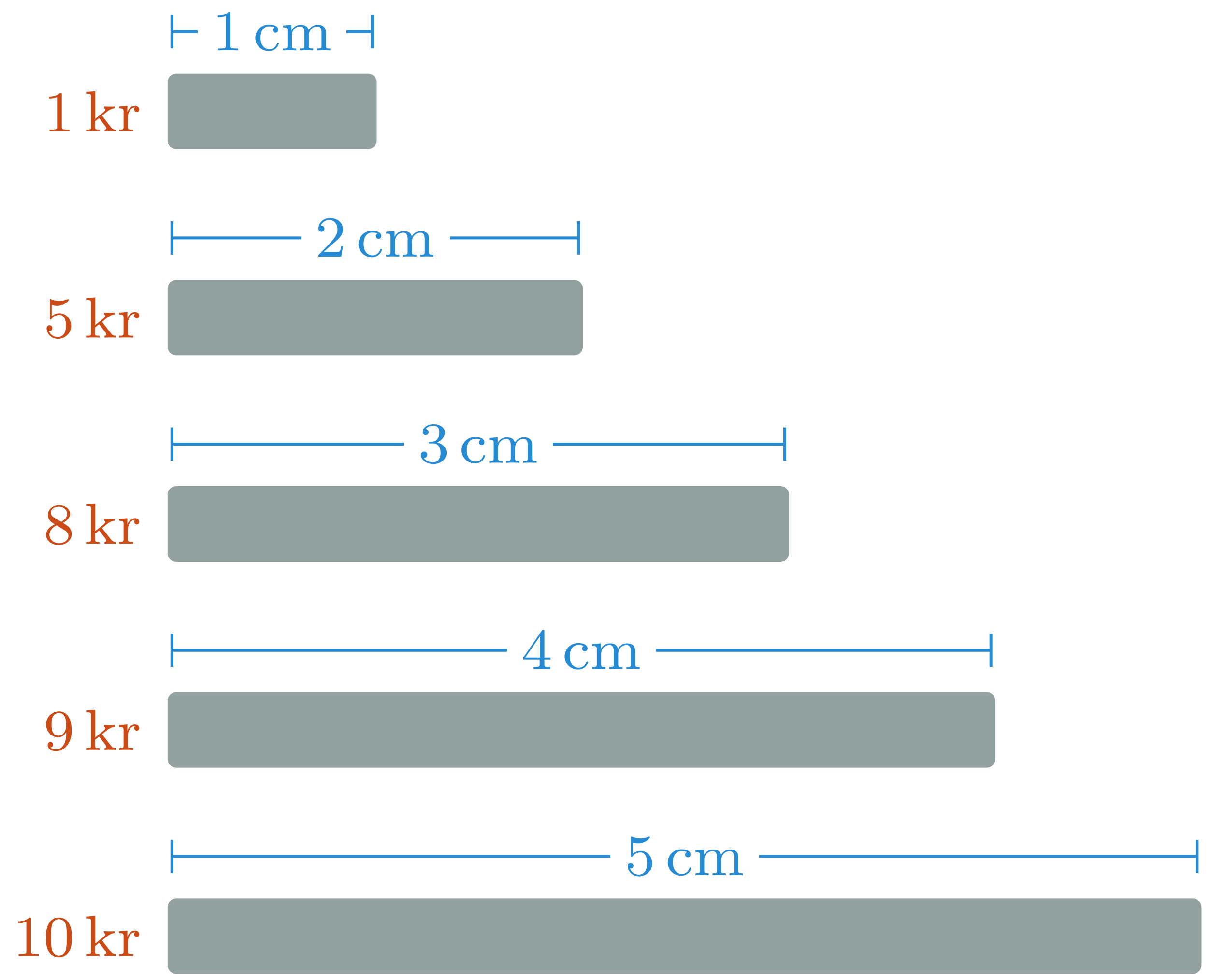
Hovedidé: Rekursiv dekomponering, akkurat som før, men noen rekursive kall går igjen, så vi lagrer svarene og slår dem opp når vi trenger dem.

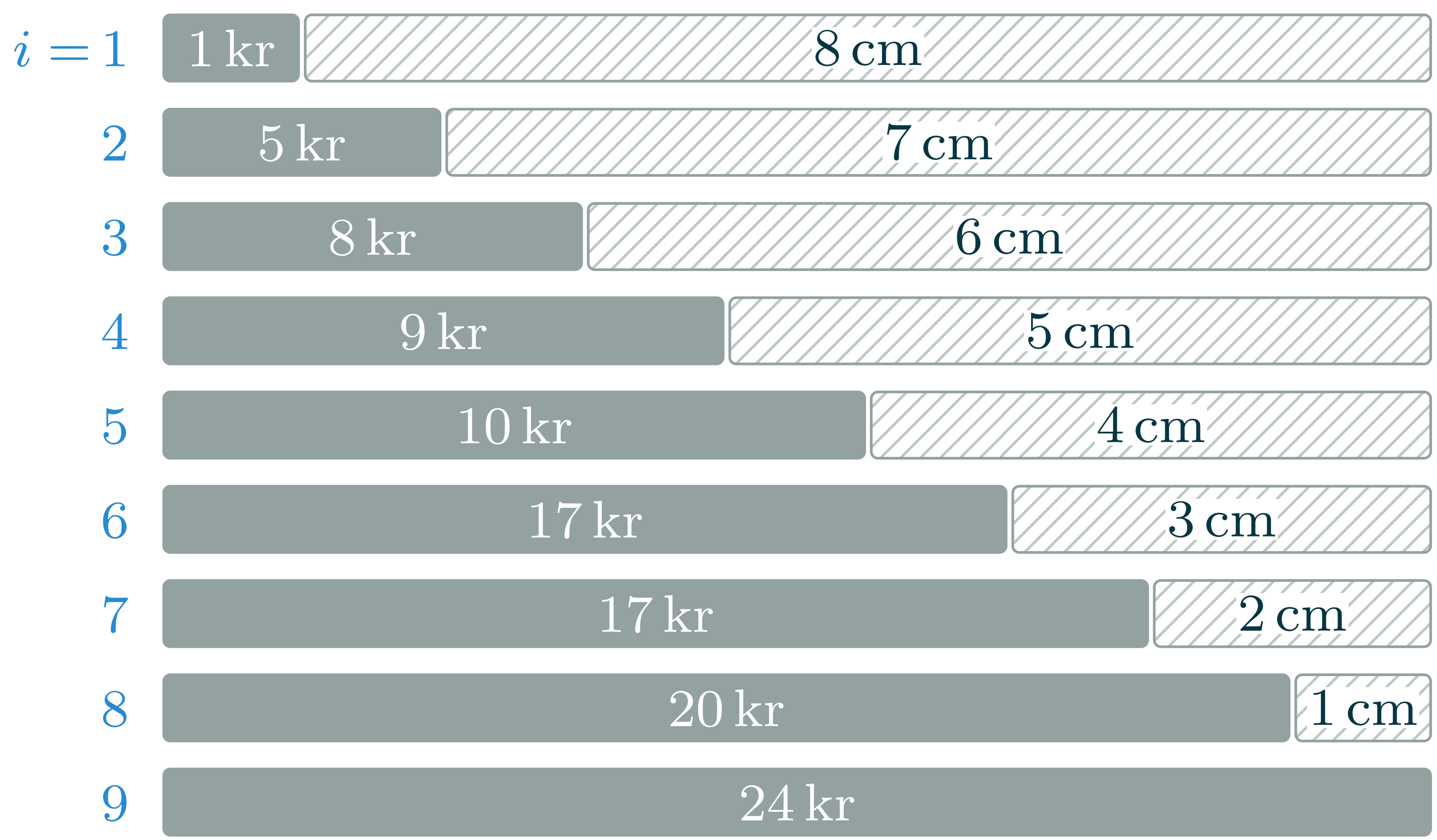
1:5

Eksempel: Stavkapping

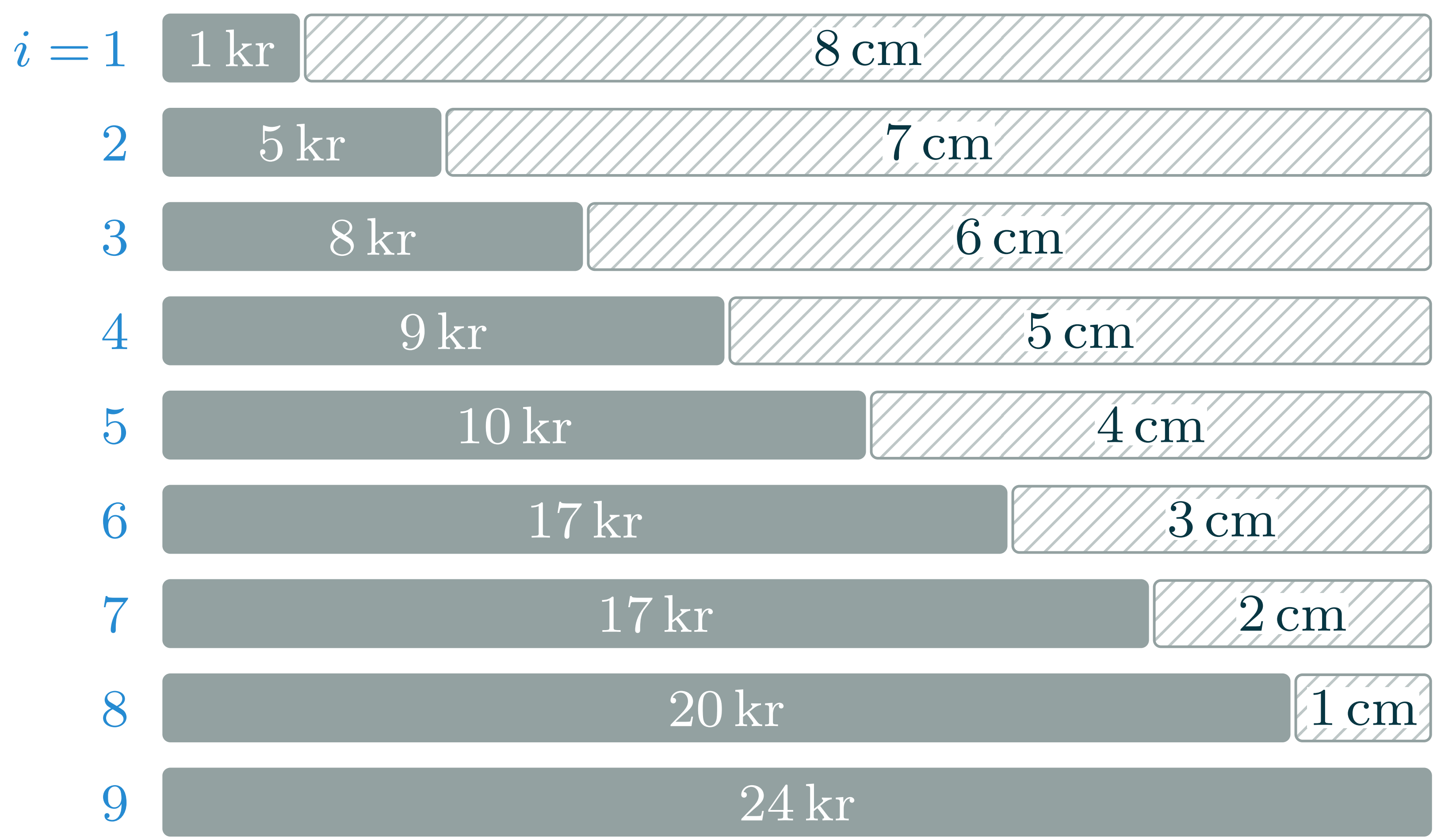
Input: En lengde n og priser p_i for lengder $i = 1, \ldots, n$.

Output: Lengder $\ell_1, \ldots, \ell_k$ der summen av lengder $\ell_1 + \cdots + \ell_k$ er n og totalprisen $r_n = p_{\ell_1} + \cdots + p_{\ell_k}$ er maksimal.

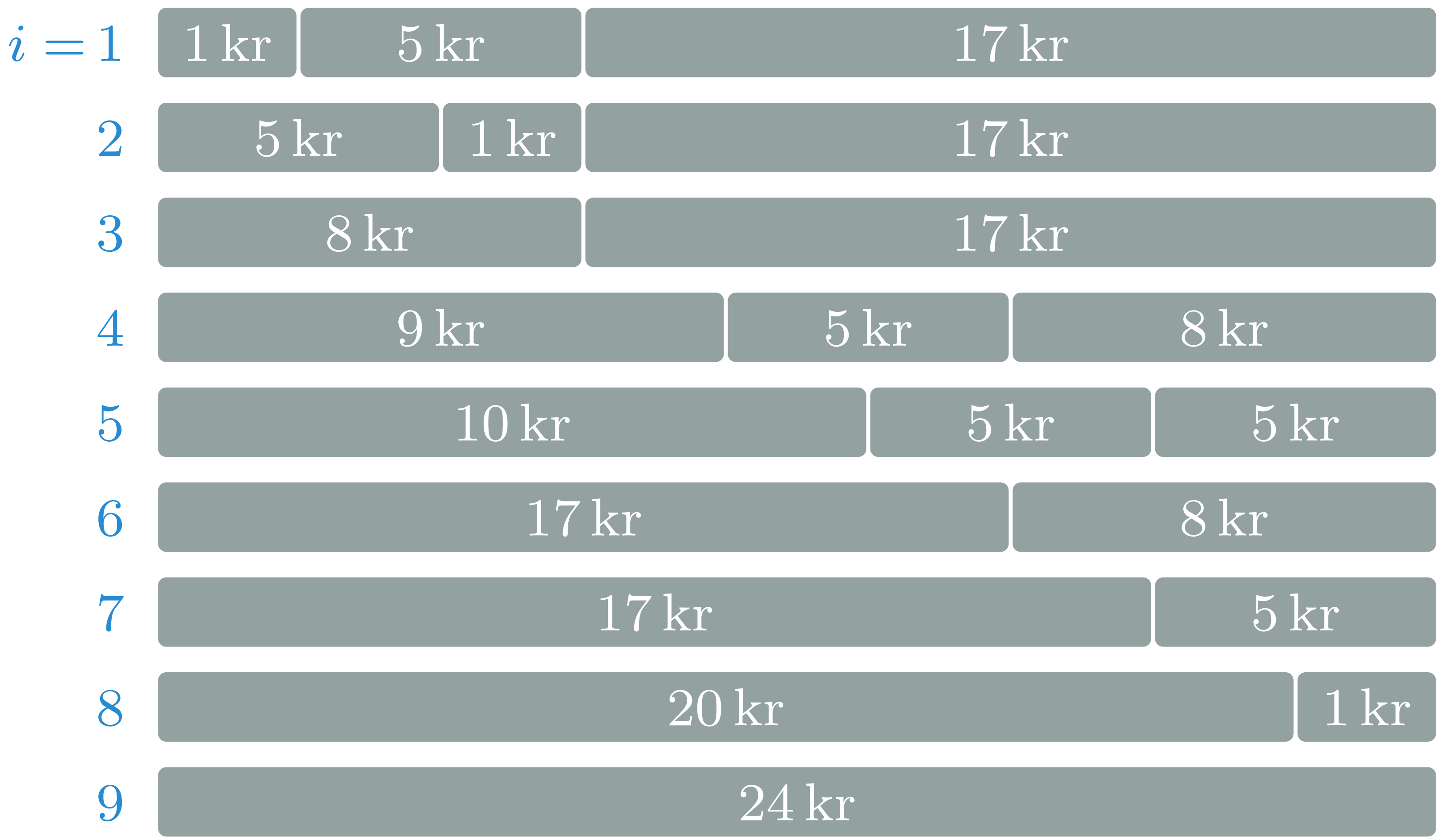

1 cm
1 kr
2 cm
5 kr
3 cm
8 kr
4 cm
9 kr
5 cm
10 kr



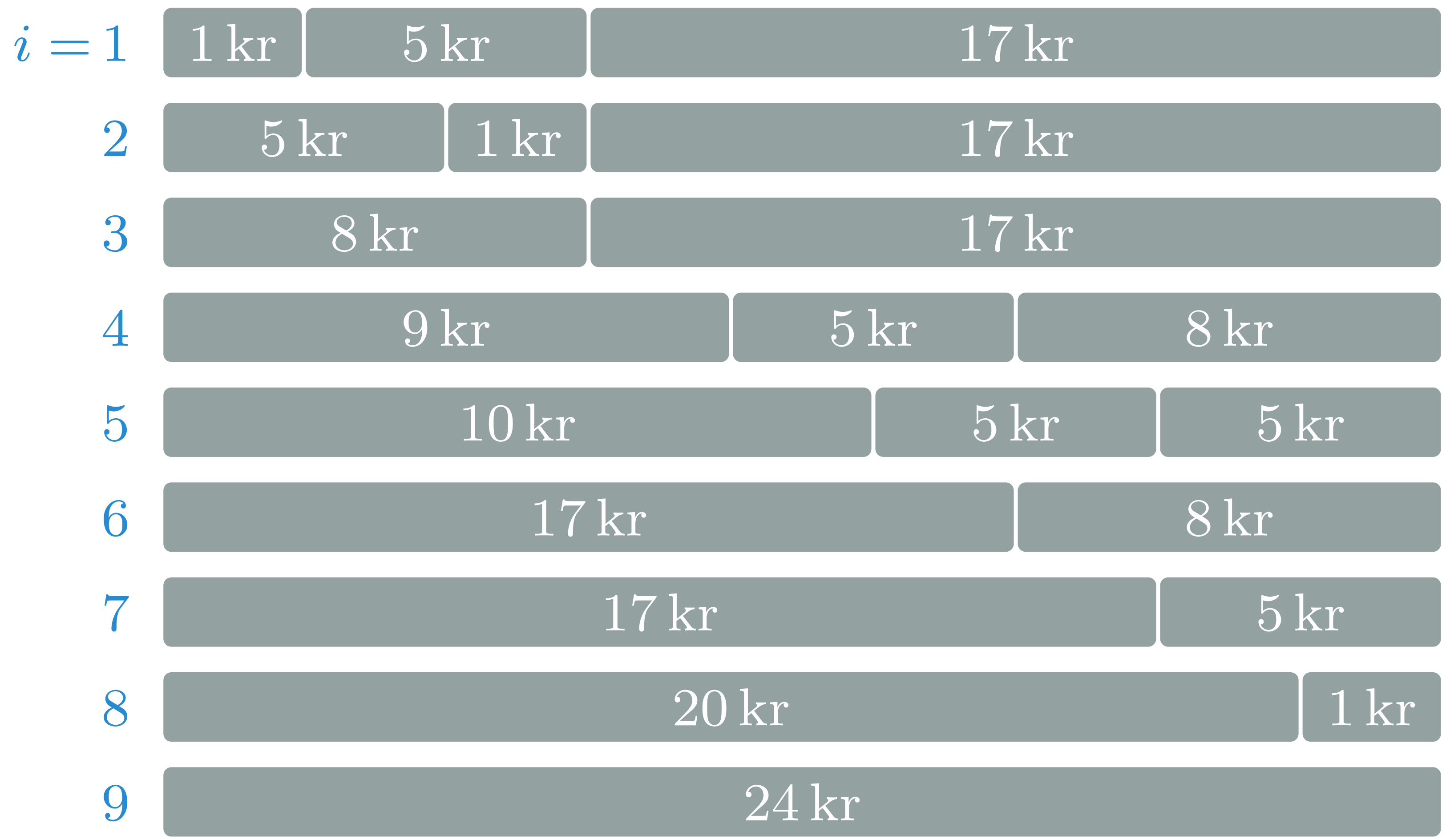
Vi prøver alle mulige (første) kutt



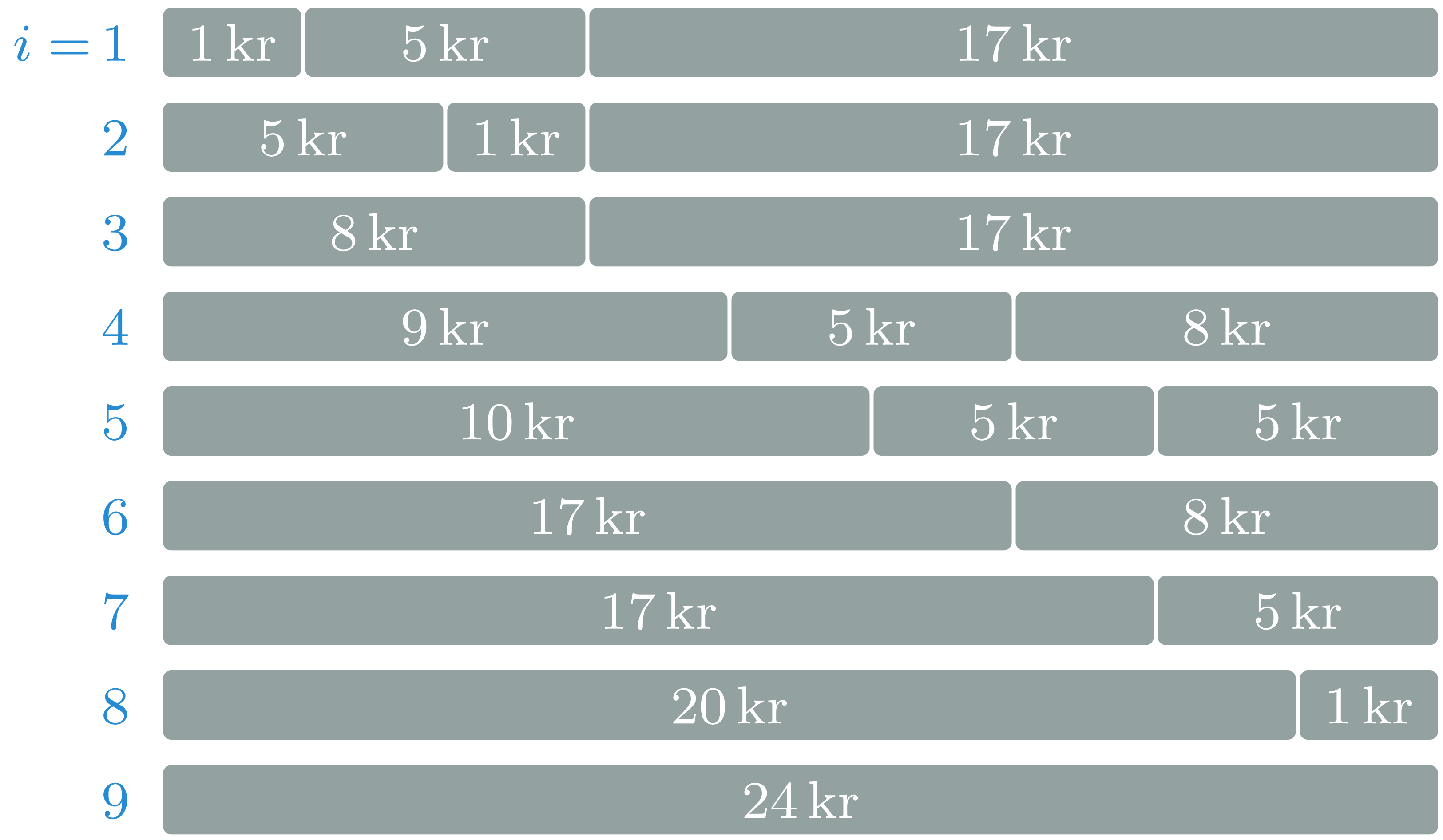
Anta at vi kan kappe resten optimalt (IH)



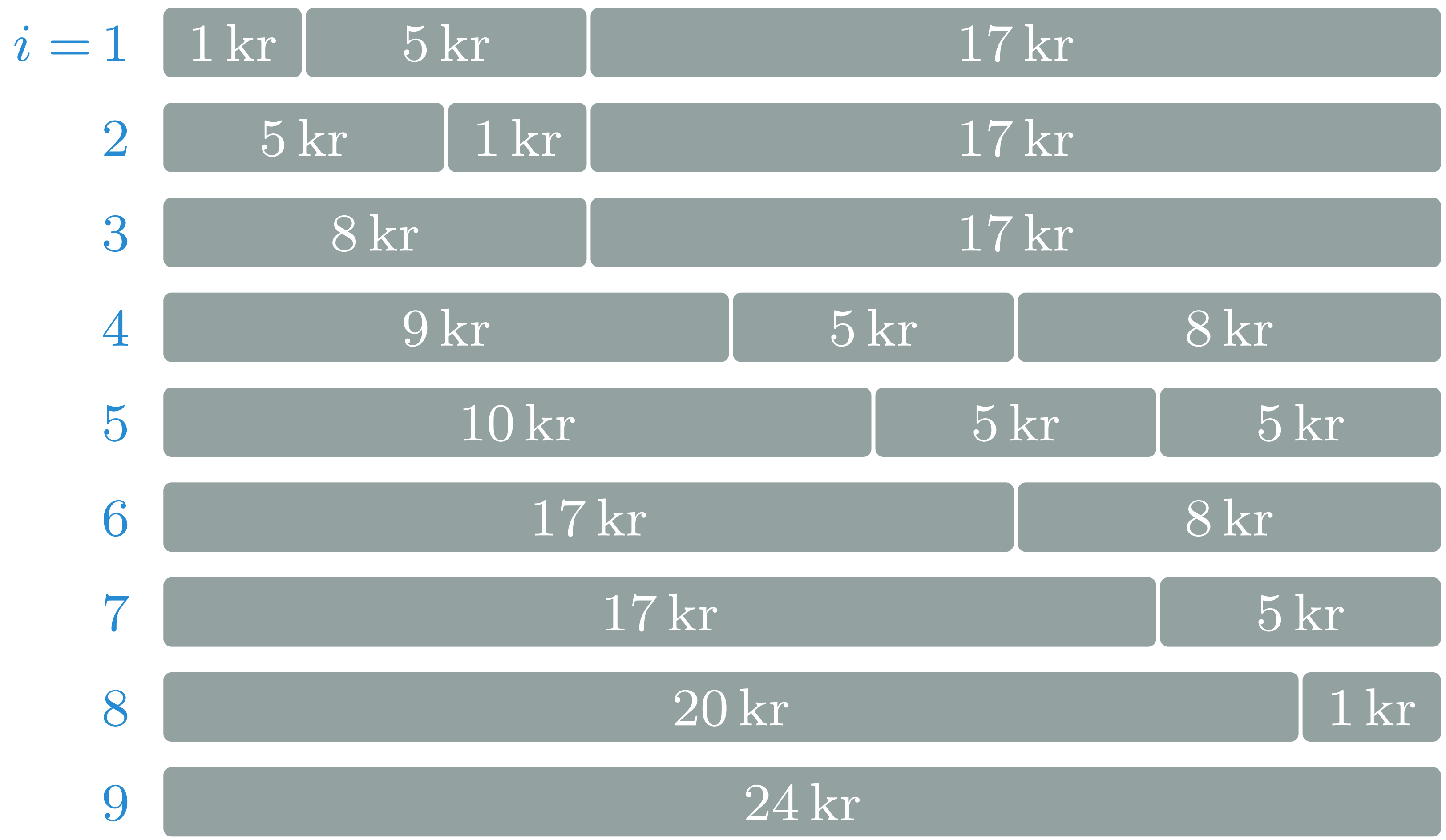
Anta at vi kan kappe resten optimalt (IH)



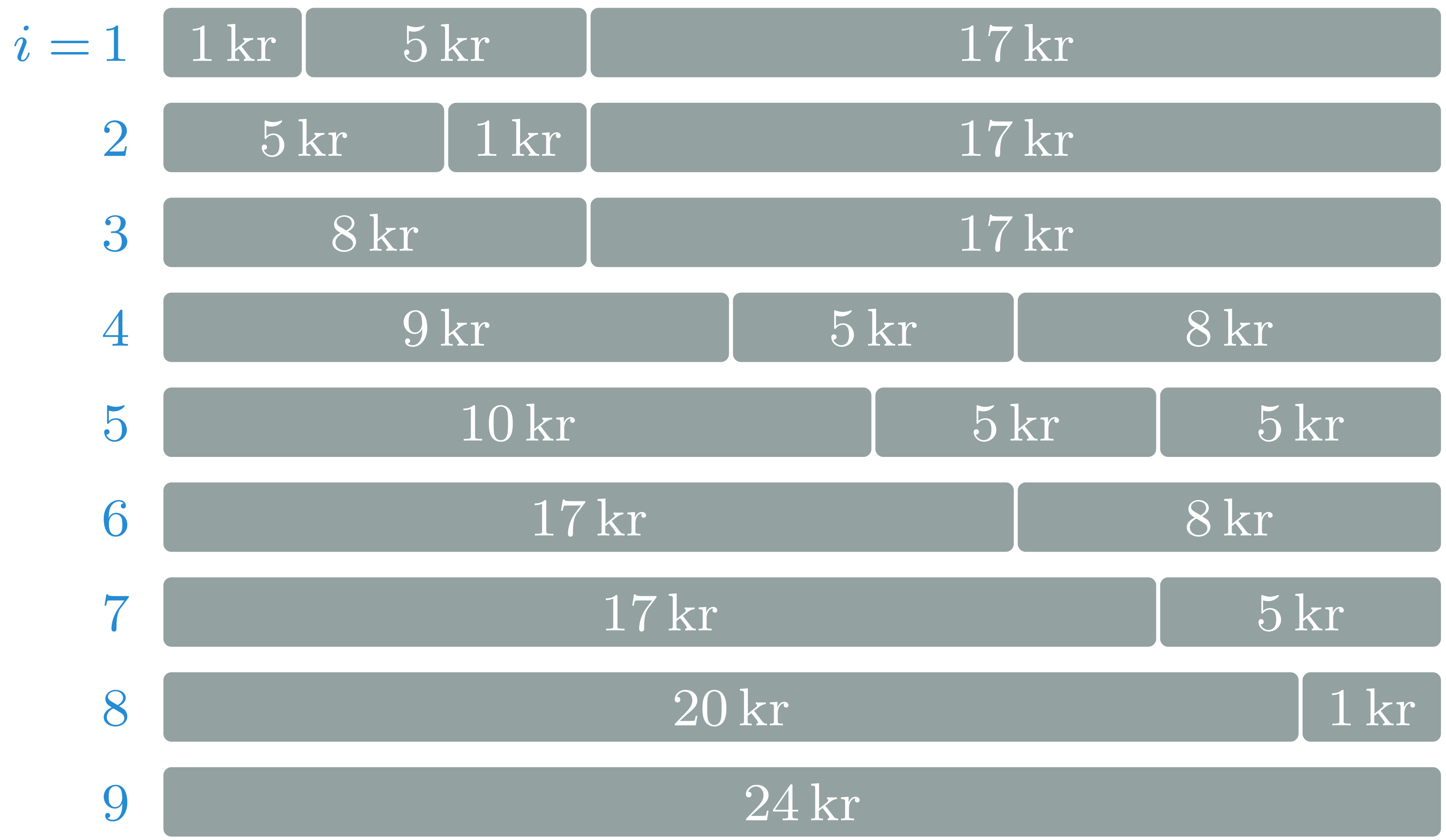
Alt unntatt første kutt er (induktivt) antatt optimalt



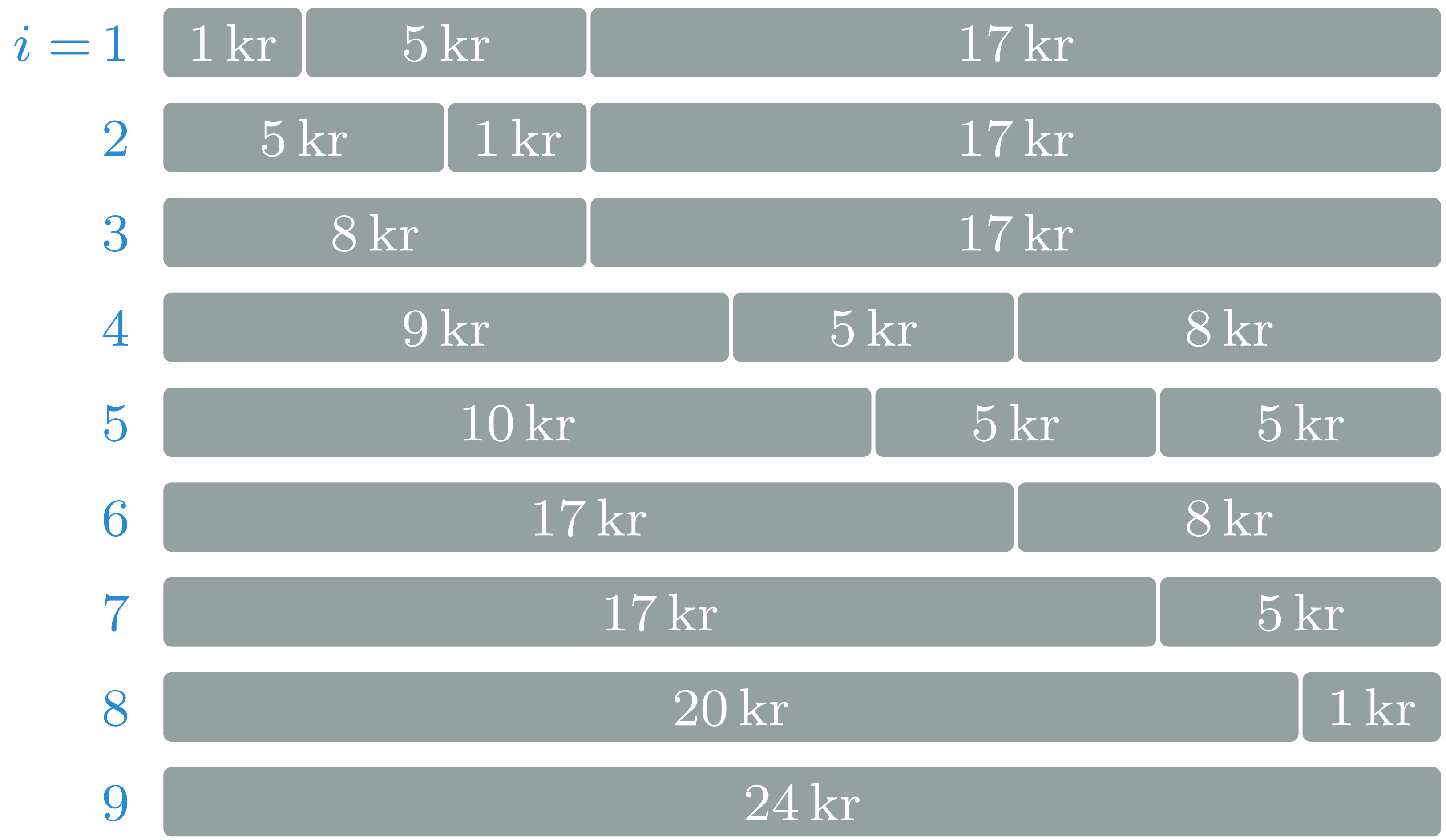
Én av løsningene må dermed være optimal; velg den beste!



Mer teknisk: Velg beste $p[i] + \textsc{Cut}(p, n - i)$



Ind. trinn: Hvis resten er optimalt (IH) så er løsningen optimal



Dermed er løsningen optimal (via induksjon)

$\textsc{Cut}(p, n)$

$p[i]$	pris
n	lengde

Har total lengde n. Pris for lengde i er p_i. Del i biter!

```
CUT(p, n)
if n == 0
```

$p[i]$	pris
n	lengde

```
CUT(p, n)
if n == 0
    return 0
```

$p[i]$	pris
n	lengde

Ingen lengde, ingen fortjeneste

CUT(p, n)
if $n == 0$
    return 0
$q = -\infty$

$p[i]$	pris
n	lengde
q	opt

Skal bli beste løsning

```
Cut(p, n)
if n == 0
        return 0
q = −∞
for i = 1 to n
```

$p[i]$	pris
n	lengde
q	opt
i	splitt

Prøv alle splittpunkter

$\textsc{Cut}(p, n)$

if $n == 0$

    return 0

$q = -\infty$

for $i = 1$ **to** n

    $t = p[i] + \textsc{Cut}(p, n - i)$

$p[i]$	pris
n	lengde
q	opt
i	splitt
t	temp

Uansett valgt splittpunkt i: Resten må kuttes optimalt

```
Cut(p, n)
if n == 0
        return 0
q = −∞
for i = 1 to n
        t = p[i] + Cut(p, n − i)
        q = max(q, t)
```

$p[i]$	pris
n	lengde
q	opt
i	splitt
t	temp

Ble det bedre enn det beste vi har?

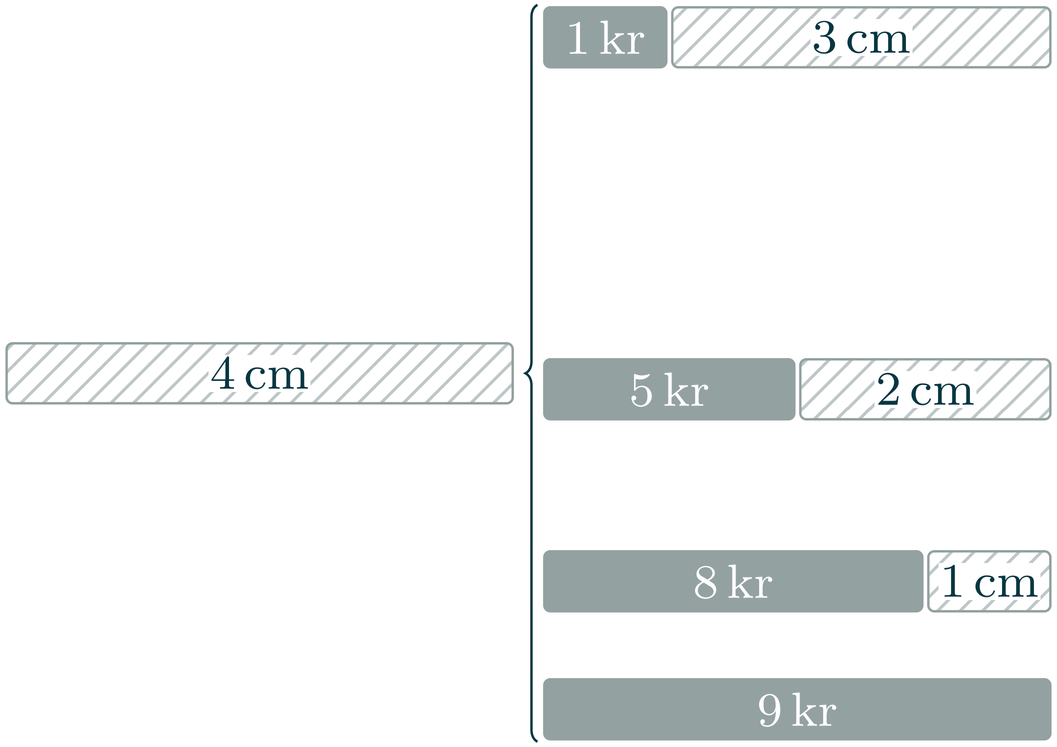
$\textsc{Cut}(p, n)$

if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ **to** n
    $t = p[i] + \textsc{Cut}(p, n - i)$
    $q = \max(q, t)$
return q

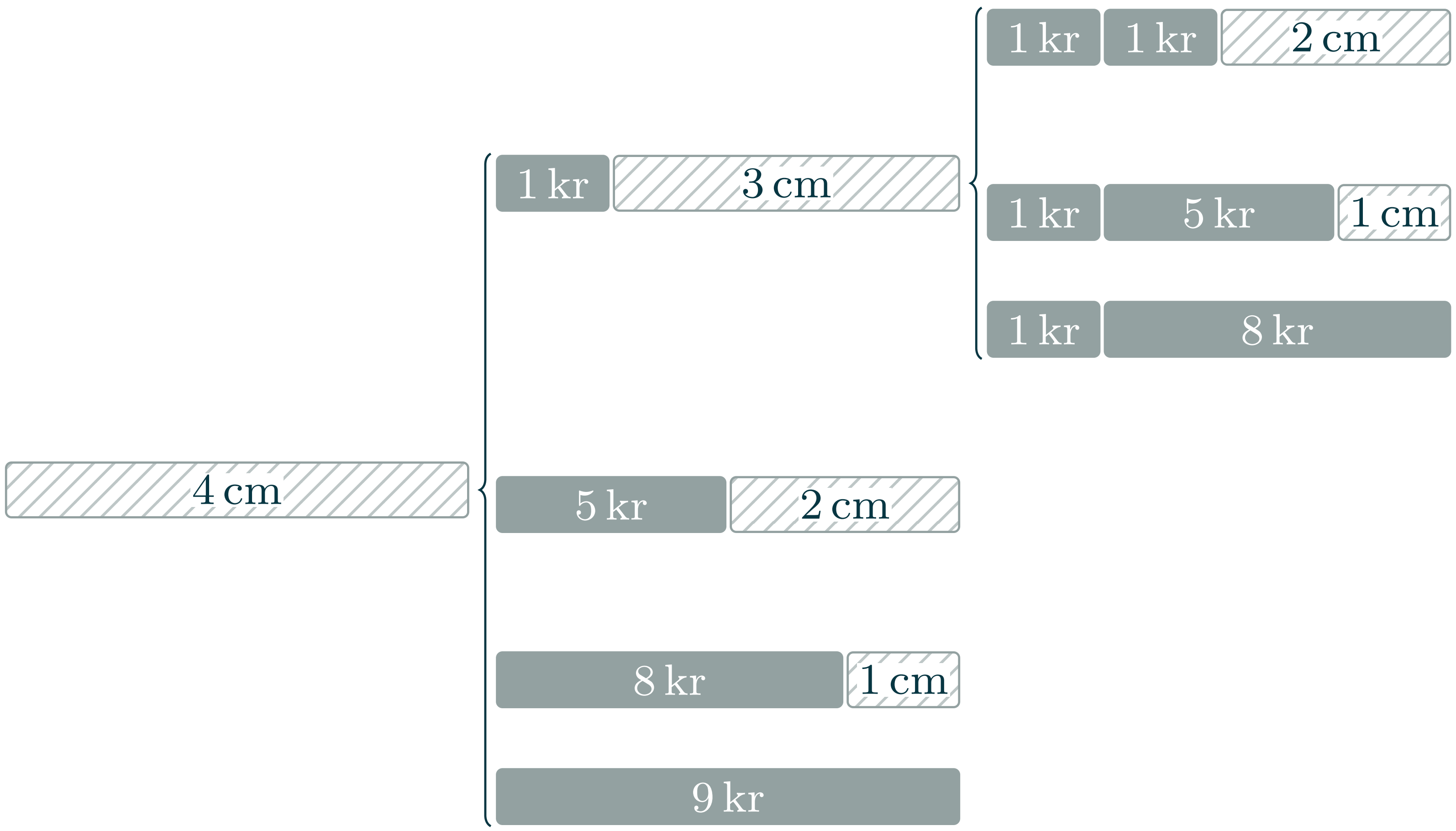
$p[i]$	pris
n	lengde
q	opt
i	splitt
t	temp

Returnér det beste vi fant

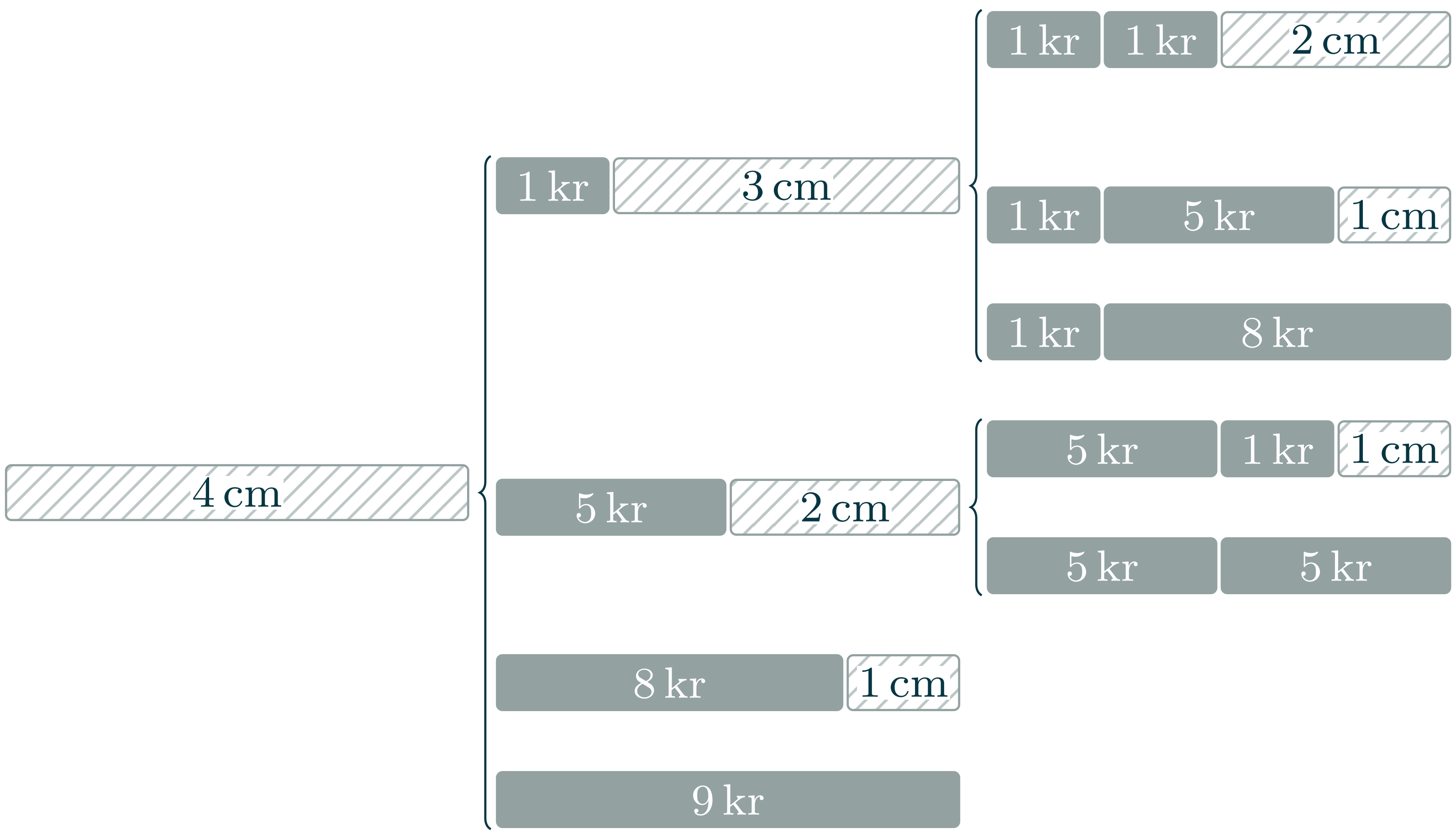
Vi har 4 cm vi ikke har kappet



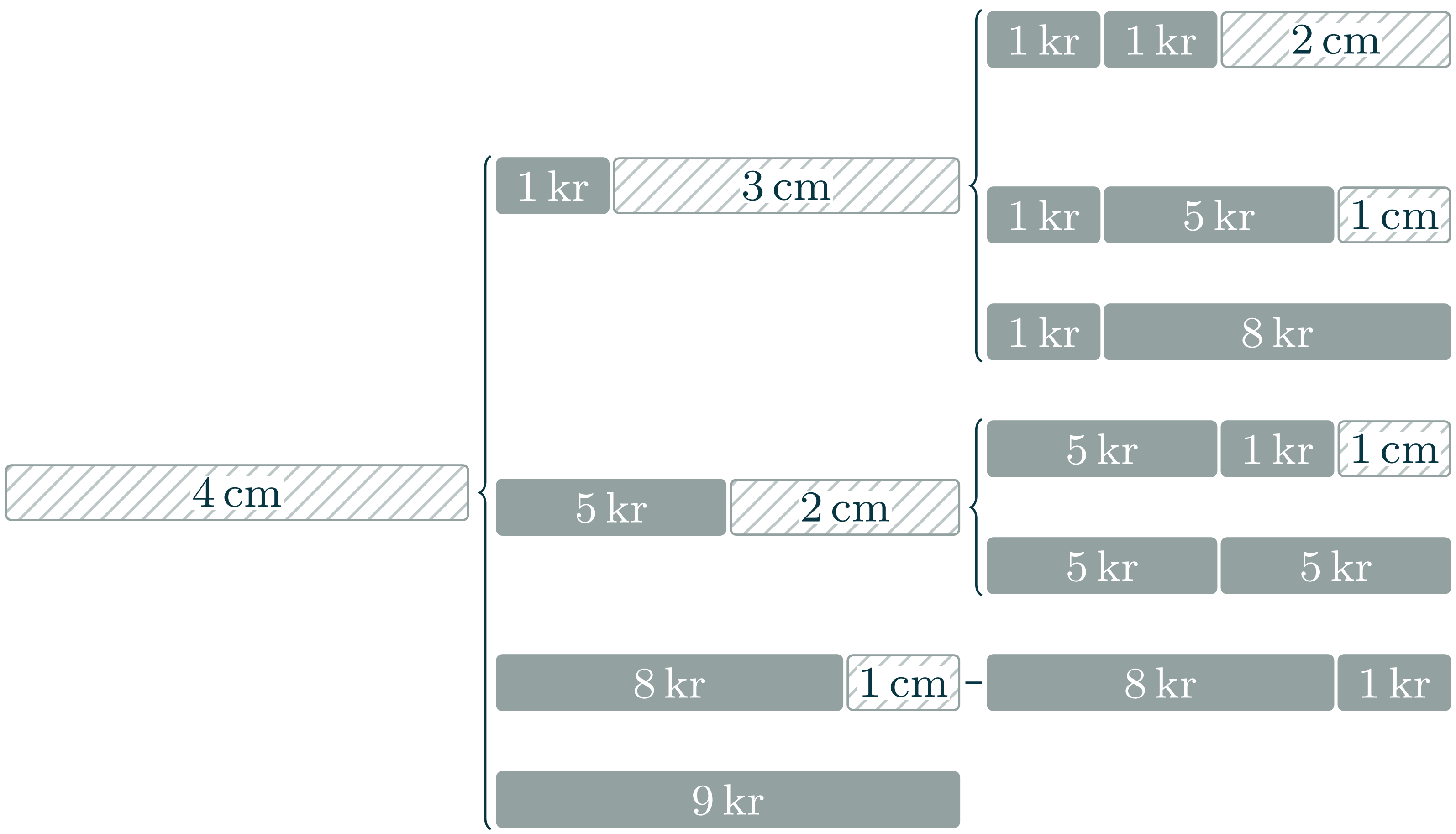
Vi prøver alle muligheter



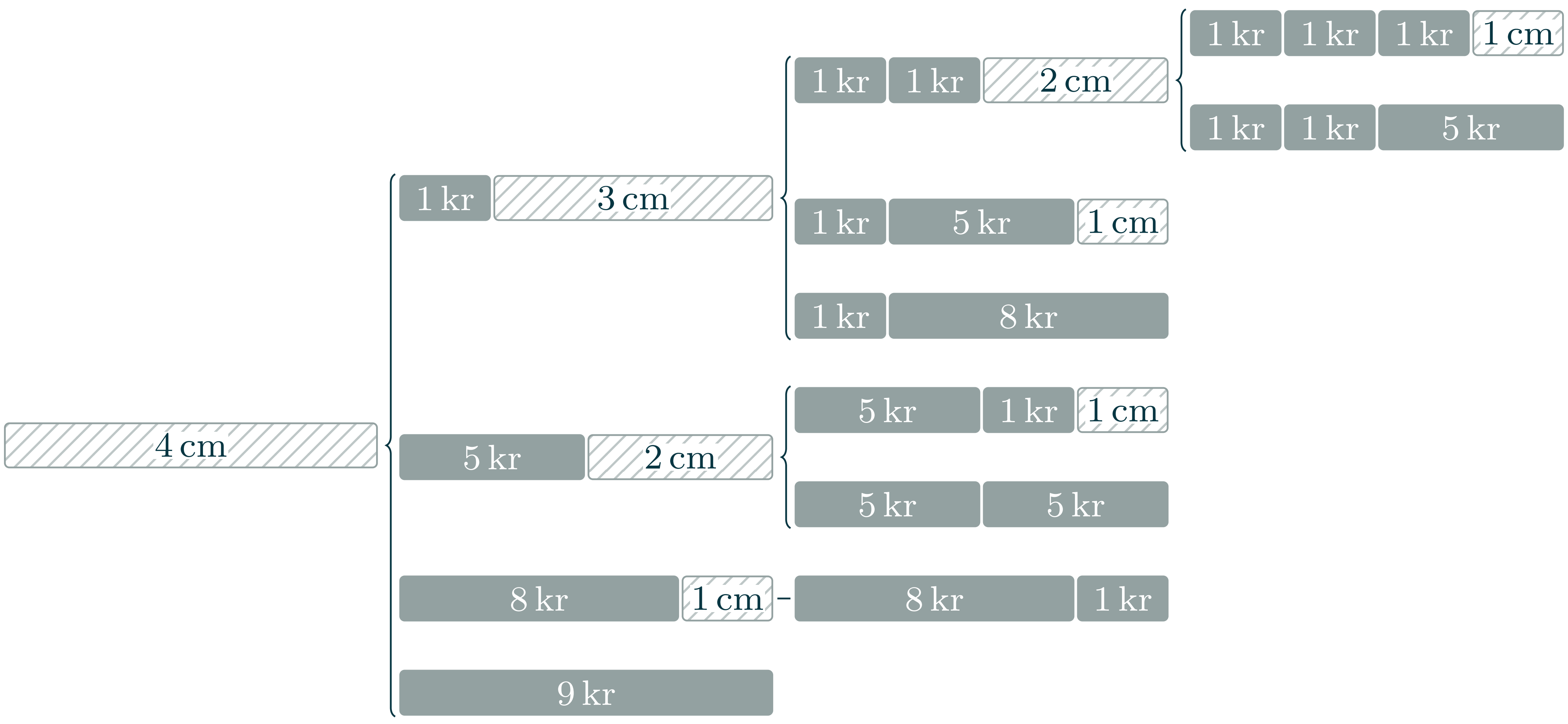
Kutter vi av 1 cm sitter vi igjen med 3


4 cm
1 kr
3 cm
1 kr
1 kr
2 cm
1 kr
5 kr
1 cm
1 kr
8 kr
5 kr
2 cm
5 kr
1 kr
1 cm
5 kr
5 kr
8 kr
1 cm
9 kr

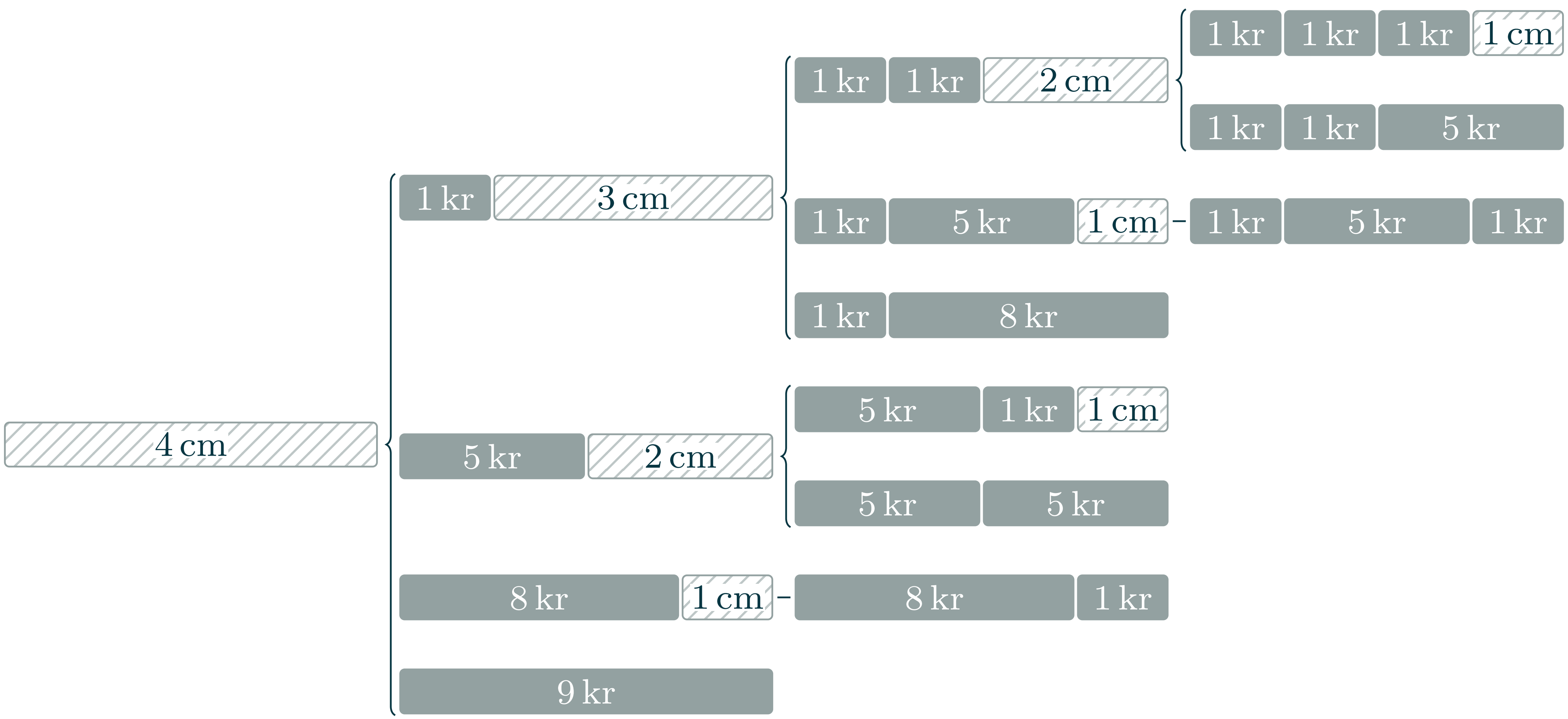
Etc.


1 kr
1 kr
2 cm
1 kr
3 cm
1 kr
5 kr
1 cm
1 kr
8 kr
5 kr
1 kr
1 cm
4 cm
5 kr
2 cm
5 kr
5 kr
8 kr
1 cm
8 kr
1 kr
9 kr

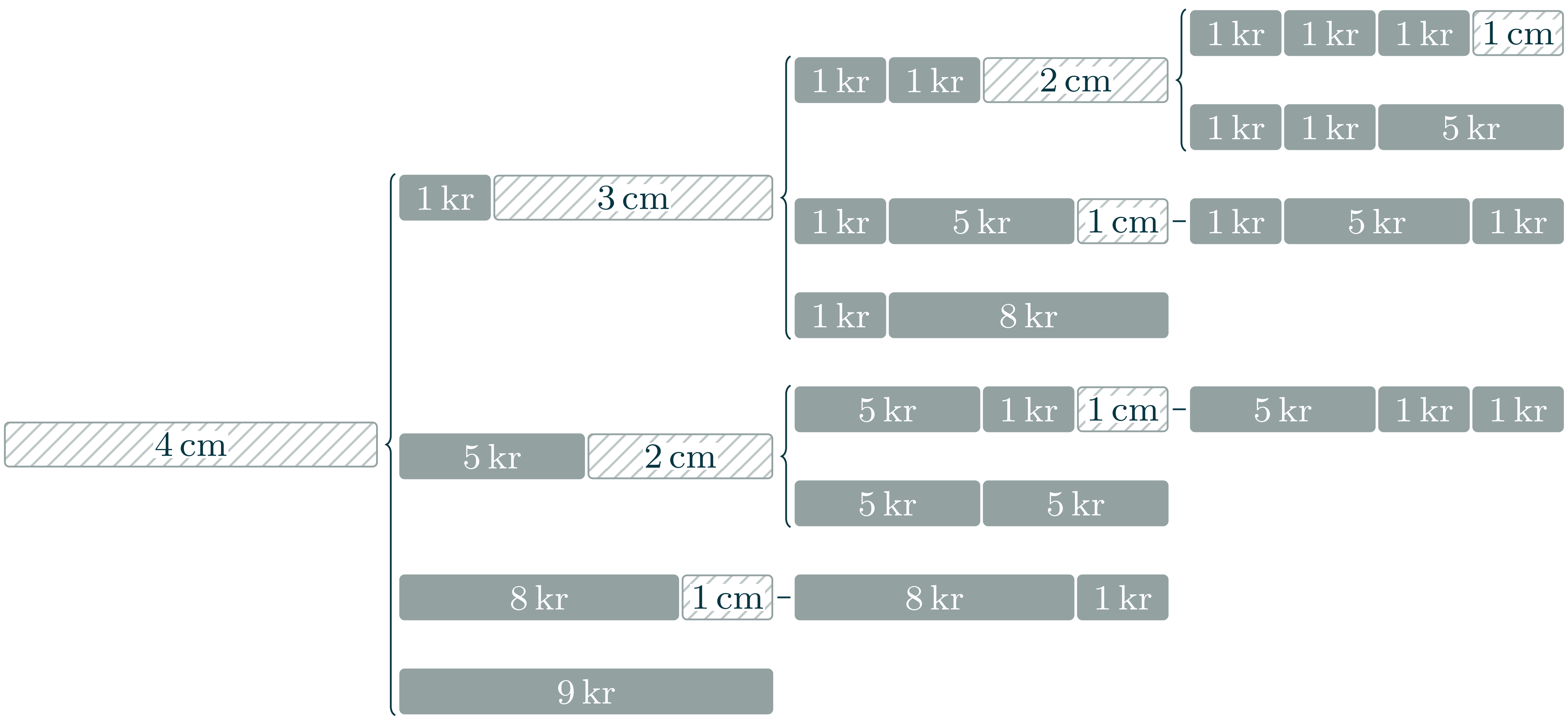
Etc.

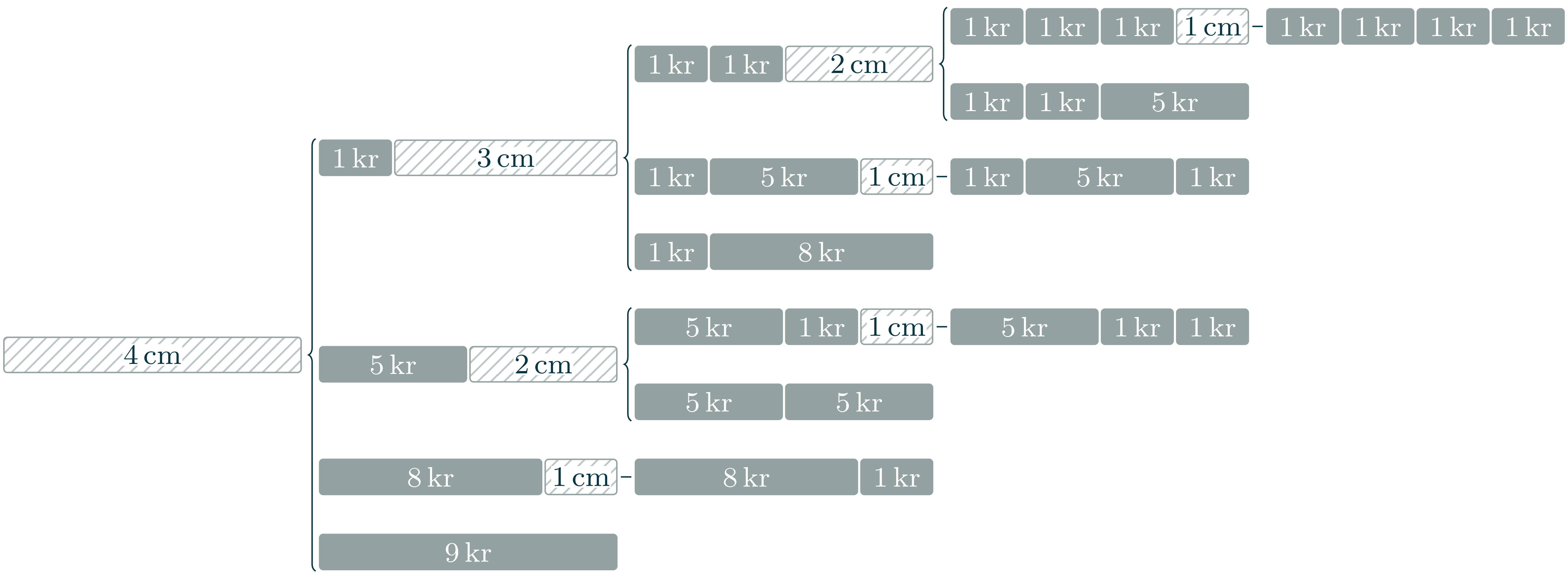


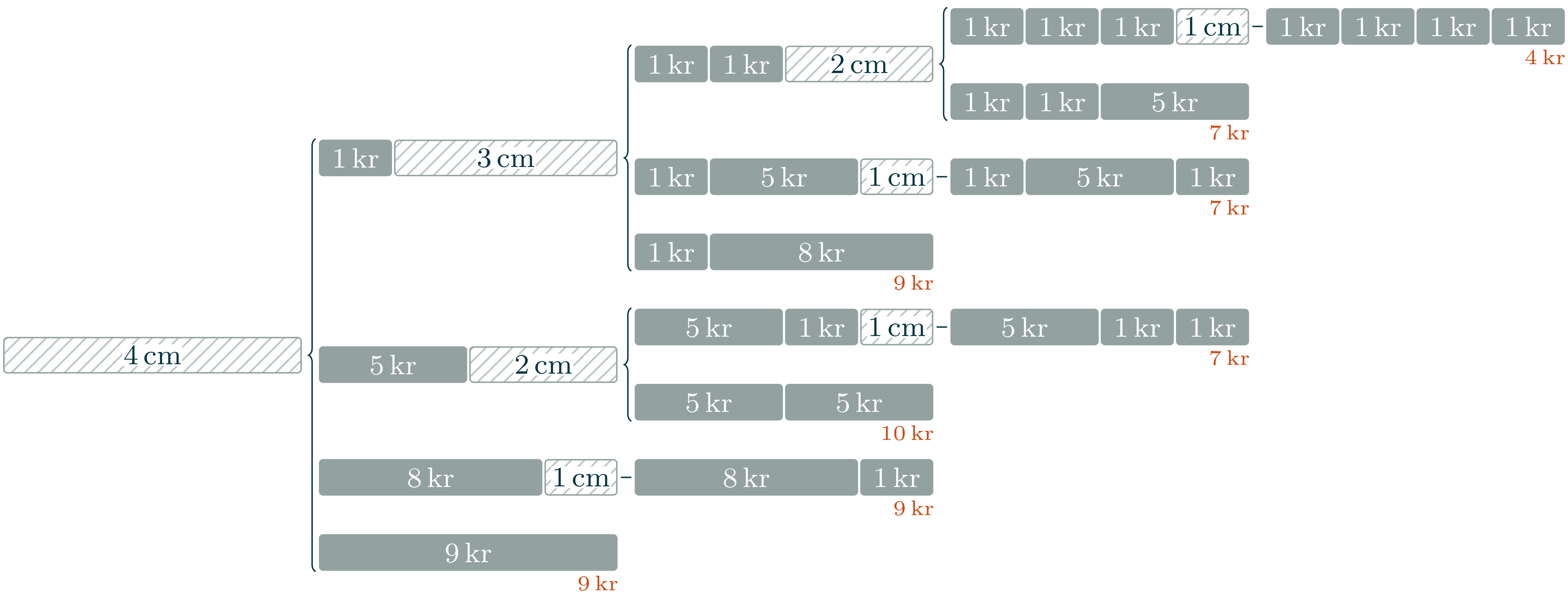
Vi fortsetter å løse resten rekursivt



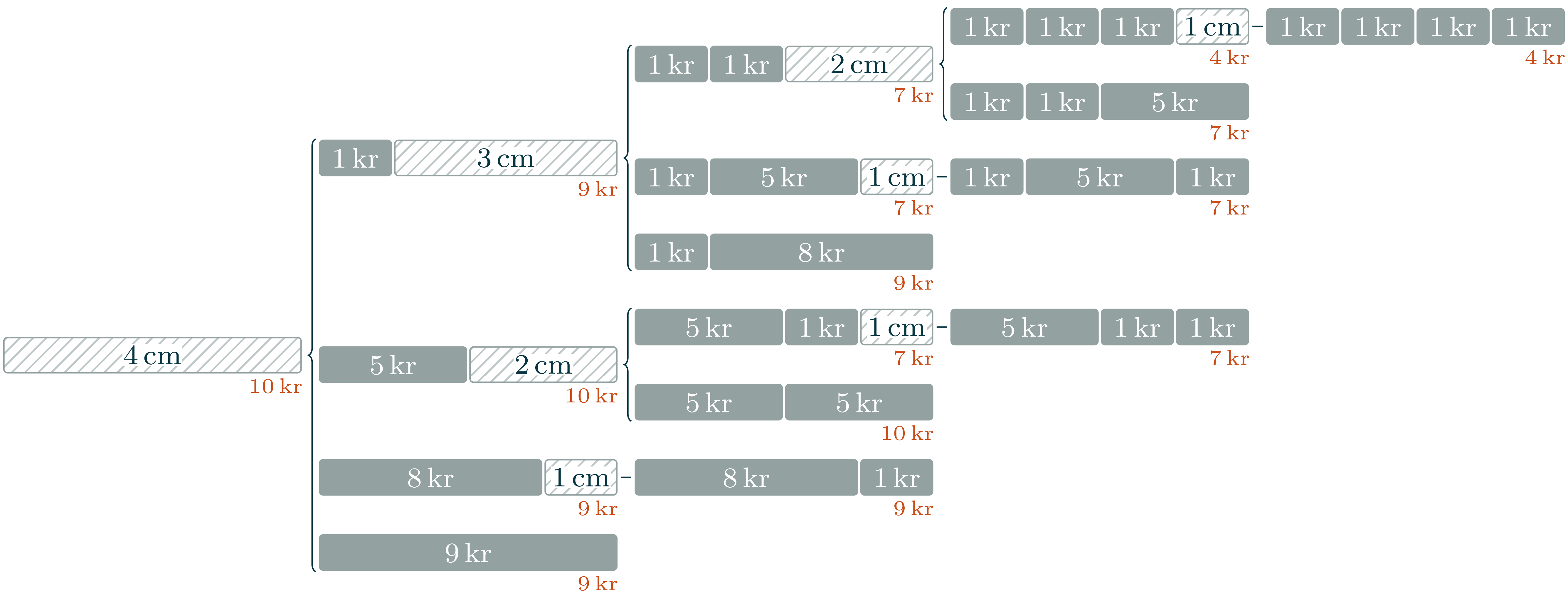
4 cm
1 kr
3 cm
1 kr
1 kr
2 cm
1 kr
1 kr
1 kr
1 cm
1 kr
1 kr
5 kr
1 kr
5 kr
1 cm
1 kr
5 kr
1 kr
1 kr
8 kr
5 kr
2 cm
5 kr
1 kr
1 cm
5 kr
5 kr
8 kr
1 cm
8 kr
1 kr
9 kr


4 cm
1 kr
3 cm
5 kr
2 cm
8 kr
1 cm
9 kr
1 kr
1 kr
2 cm
1 kr
5 kr
1 cm
1 kr
8 kr
5 kr
1 kr
1 cm
5 kr
5 kr
8 kr
1 kr
1 kr
1 kr
1 kr
1 cm
1 kr
1 kr
5 kr
1 kr
5 kr
1 kr
5 kr
1 kr
1 kr

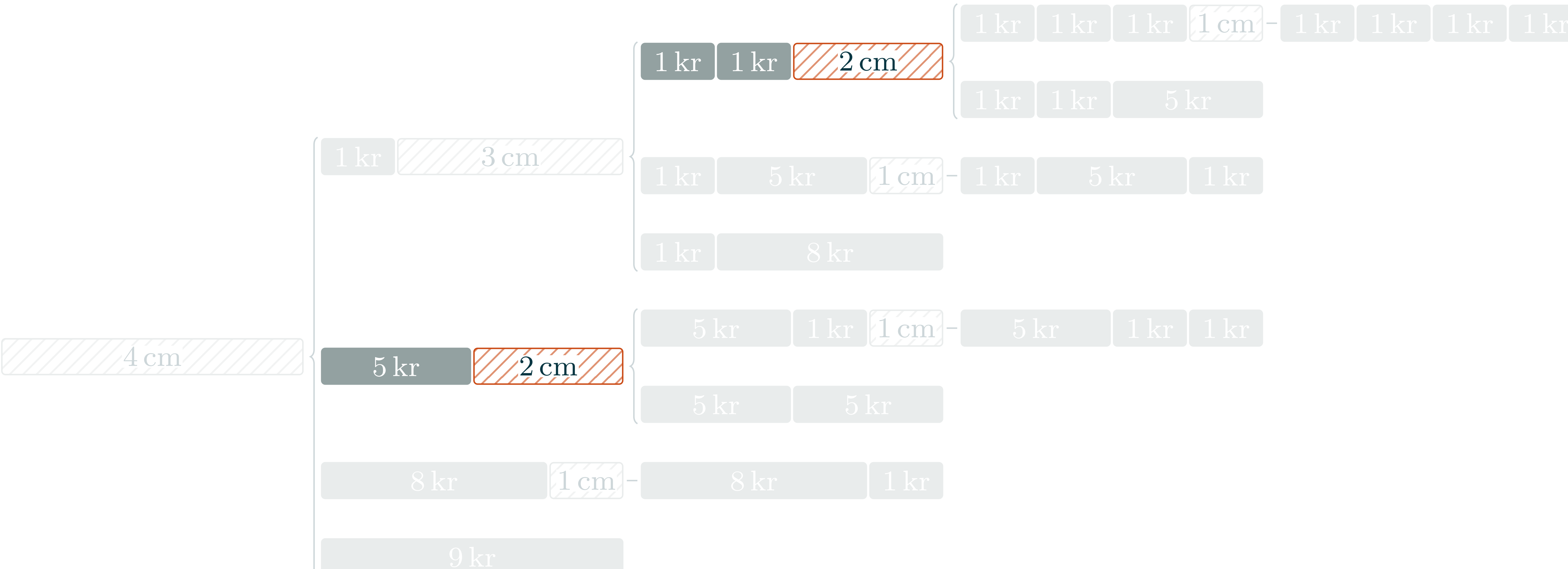

4 cm
1 kr
3 cm
1 kr
1 kr
2 cm
1 kr
1 kr
1 kr
1 cm
1 kr
1 kr
1 kr
1 kr
1 kr
1 kr
5 kr
1 kr
5 kr
1 cm
1 kr
5 kr
1 kr
1 kr
8 kr
5 kr
2 cm
5 kr
1 kr
1 cm
5 kr
1 kr
1 kr
5 kr
5 kr
8 kr
1 cm
8 kr
1 kr
9 kr



Hvert grunntilfelle er bare en sum av priser



Ellers velger vi beste delløsning



Men her har vi løst for $n = 2$ mer enn én gang! Hm . . .

```
Cut(p, n)
if n == 0
        return 0
q = −∞
for i = 1 to n
        t = p[i] + Cut(p, n − i)
        q = max(q, t)
return q
```

$q, t \; = \;$ -, -

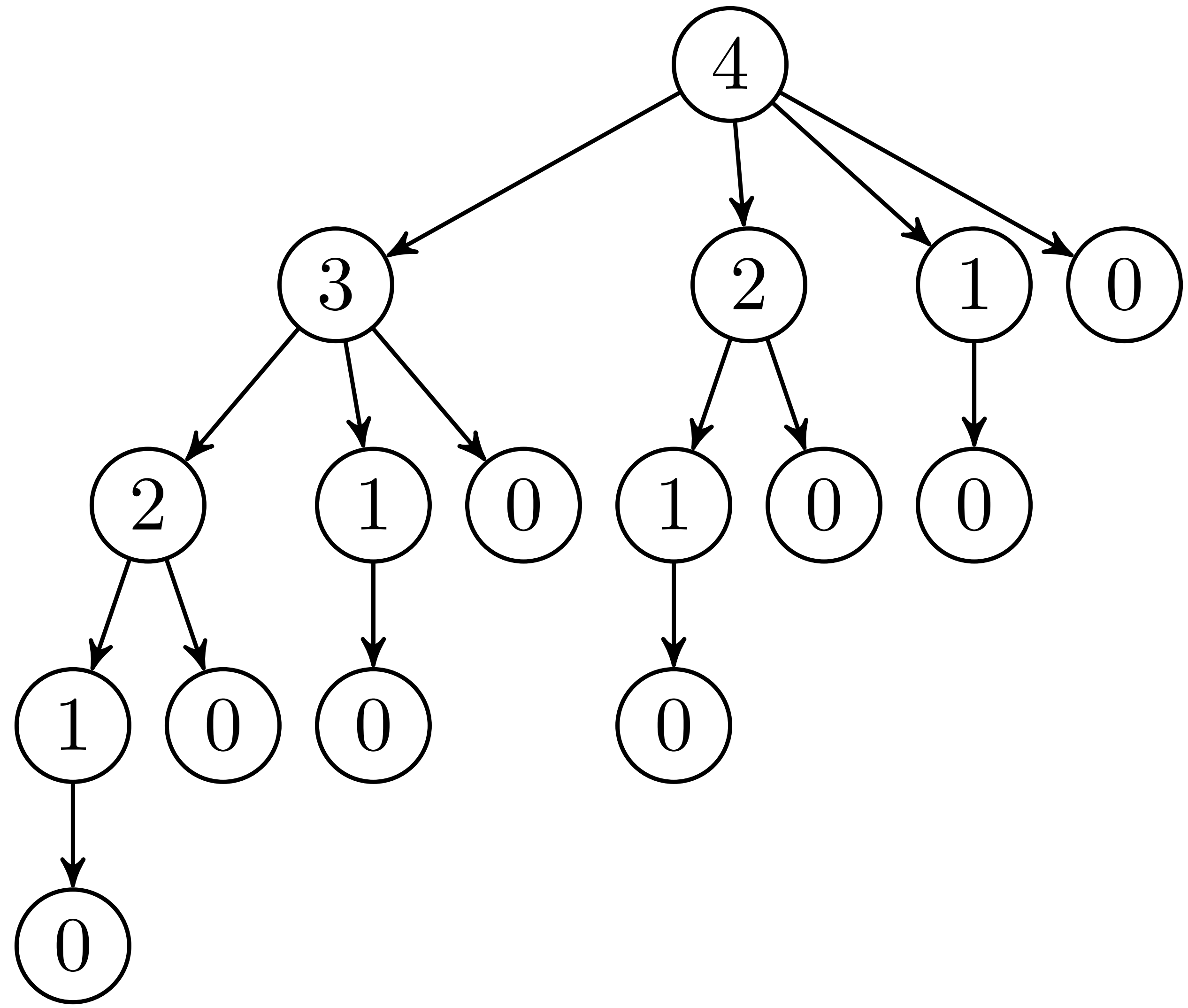
Hver node representerer et kall til Cut, og tallet i noden er verdien til parameteren n.

4

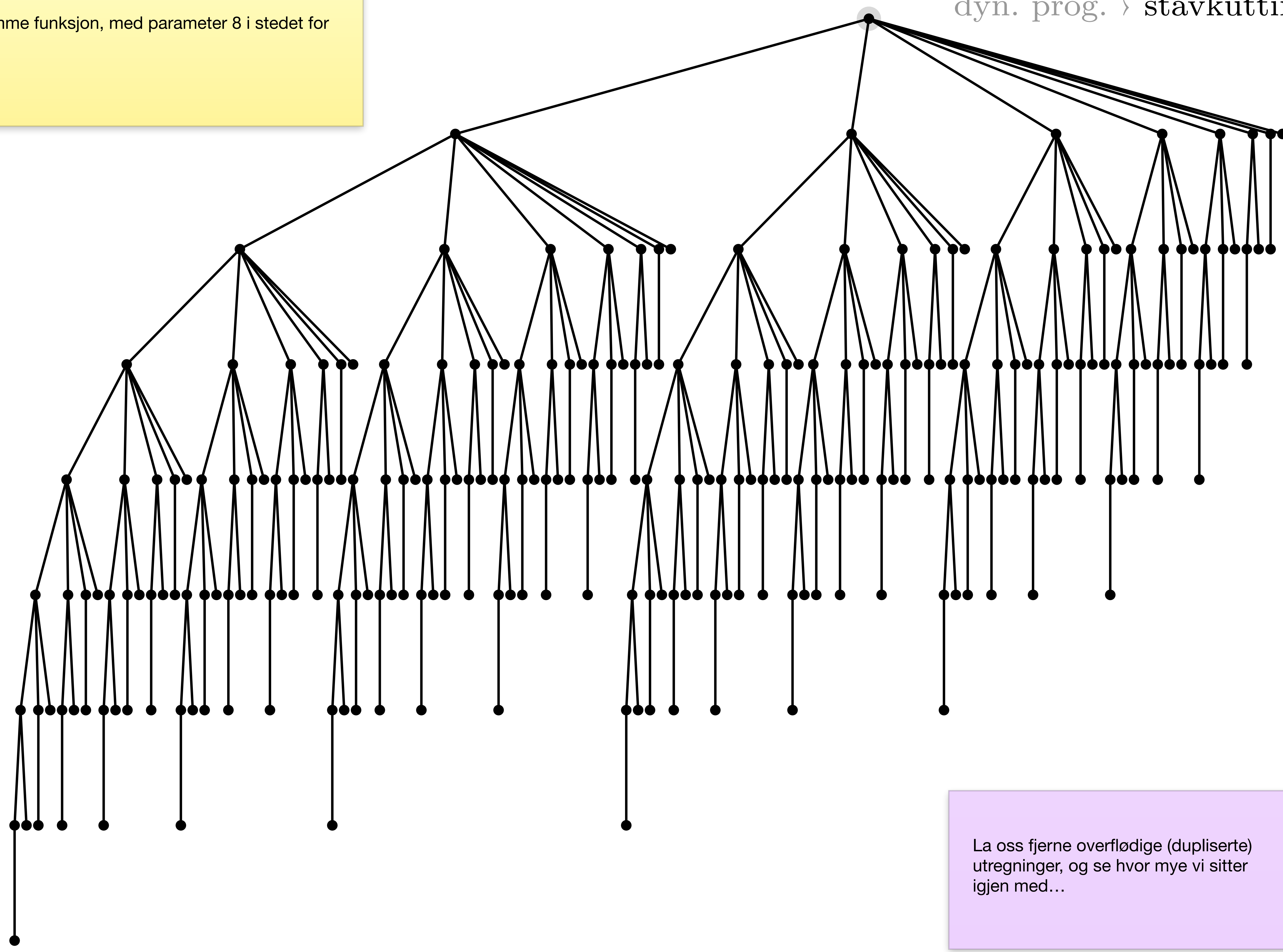
For simulering med innholdet i p, se bonusmateriale.

```
CUT(p, n)
if n == 0
        return 0
q = −∞
for i = 1 to n
        t = p[i] + CUT(p, n − i)
        q = max(q, t)
return q
```

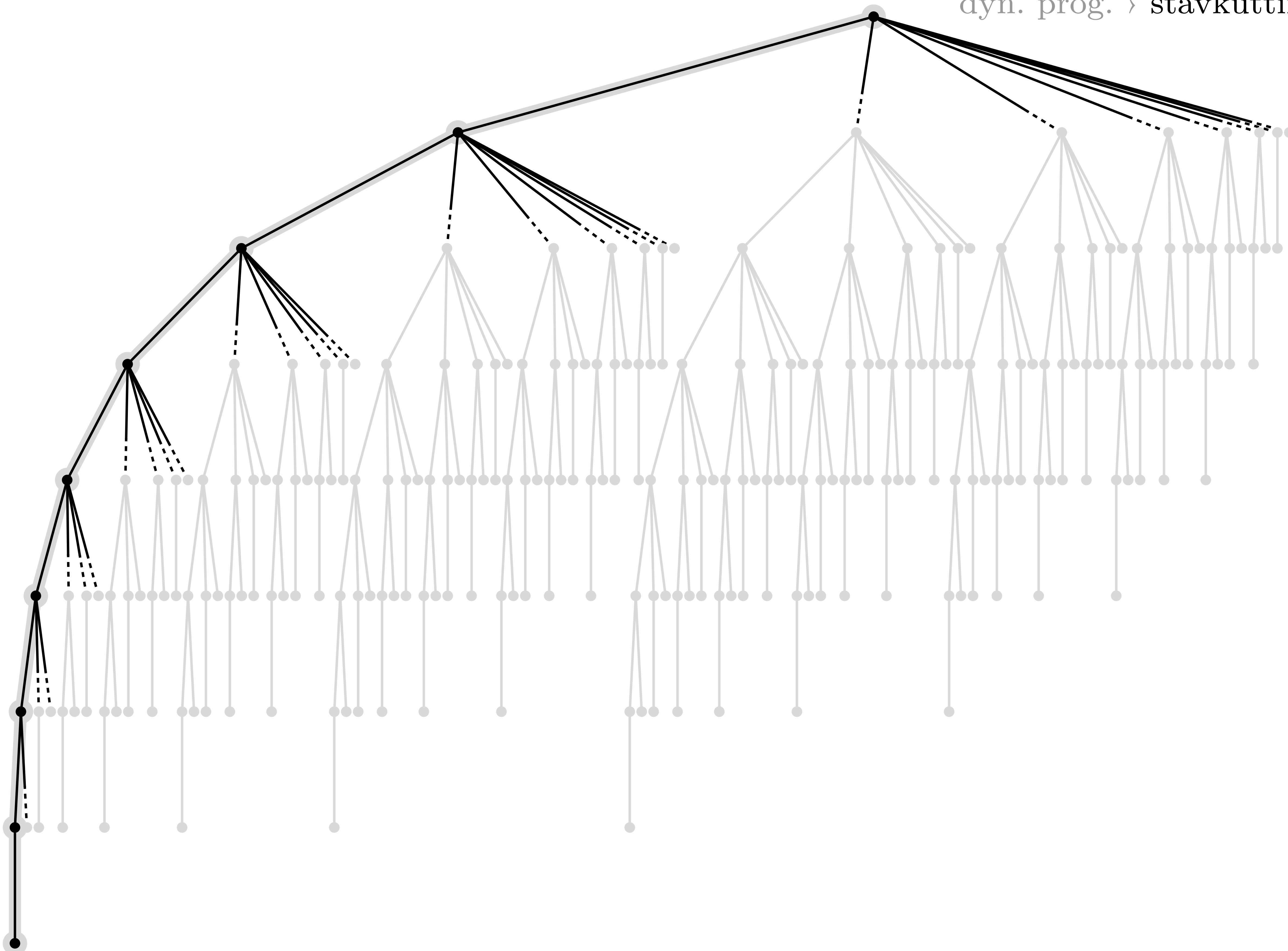
→ 10



Samme funksjon, med parameter 8 i stedet for 4…



La oss fjerne overflødige (dupliserte) utregninger, og se hvor mye vi sitter igjen med…



Vi vil beregne hver delløsning maks én gang. Plasser dem i et «regneark»!

I stedet for rekursjon: Hver celle beregnes basert på andre celler.

Kode og simulering: Se bonusmateriale.

$$p[i] + \textsc{Cut}(p, n - i)$$

Optimum, om vi først kapper av i cm

$$p[i] + \textsc{Cut}(p, n - i)$$

Optimum for resten $(n - i)$ beregnes når vi trenger det

$$p[i] + \textsc{Cut}(p, n - i)$$

Vi kan også beregne det *på forhånd*, og lagre svaret i $r[0\,..\,n]$

$$p[i] + \textsc{Cut}(p, n - i)$$

$$p[i] + r[n - i]$$

Vi kan nå slå opp svaret i konstant tid

$$p[i] + \textsc{Cut}(p, n - i)$$

$$p[i] + r[n - i]$$

Ikke sirkulært: Vi trenger bare $r[0 \mathinner{.\,.} n - 1]$ for å beregne $r[n]$

$$r[n] = \max_i \; p[i] + r[n-i]$$

Ikke sirkulært: Vi trenger bare $r[0\,..\,n-1]$ for å beregne $r[n]$

BOTTOM-UP-CUT-ROD(p, n)

$p[i]$	pris
n	tot. lengde

Finn optimum for hver lengde fra 0 til n

BOTTOM-UP-CUT-ROD(p, n)
1 let $r[0\,..\,n]$ be a new array

$p[i]$ pris
n tot. lengde
$r[j]$ opt, lengde j

Til lagring av delløsninger, i stedet for rekursjon

Bottom-Up-Cut-Rod(p, n)
1 let $r[0\,..\,n]$ be a new array
2 $r[0] = 0$

$p[i]$	pris
n	tot. lengde
$r[j]$	opt, lengde j

Ingen lengde, ingen fortjeneste

BOTTOM-UP-CUT-ROD(p, n)
let $r[0\,..\,n]$ be a new array
$r[0] = 0$
for $j = 1$ **to** n

$p[i]$	pris
n	tot. lengde
$r[j]$	opt, lengde j
j	lengde

For hver lengde: Har alt løst for mindre lengder

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
        q = −∞
```

$p[i]$	pris
n	tot. lengde
$r[j]$	opt, lengde j
j	lengde
q	skal bli $r[j]$

Skal bli beste løsning

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
```

$p[i]$	pris
n	tot. lengde
$r[j]$	opt, lengde j
j	lengde
q	skal bli $r[j]$
i	splitt

Prøv alle splittpunkter

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
```

$p[i]$	pris
n	tot. lengde
$r[j]$	opt, lengde j
j	lengde
q	skal bli $r[j]$
i	splitt

Uansett valgt splittpunkt i: Resten må kuttes optimalt

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
```

$p[i]$	pris
n	tot. lengde
$r[j]$	opt, lengde j
j	lengde
q	skal bli $r[j]$
i	splitt

Beste løsning for lengde j

```
Bottom-Up-Cut-Rod(p, n)
let r[0 . . n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

$p[i]$	pris
n	tot. lengde
$r[j]$	opt, lengde j
j	lengde
q	skal bli $r[j]$
i	splitt

Beste løsning for lengde n

mlh@ntnu.no

	1	2	3	4	5	6	7
p	1	5	8	9	10	17	17

r	0	1	5	8	10	13		

$$r[j] = \max_{i=1\ldots j}\big(p[i] + r[j-i]\big)$$

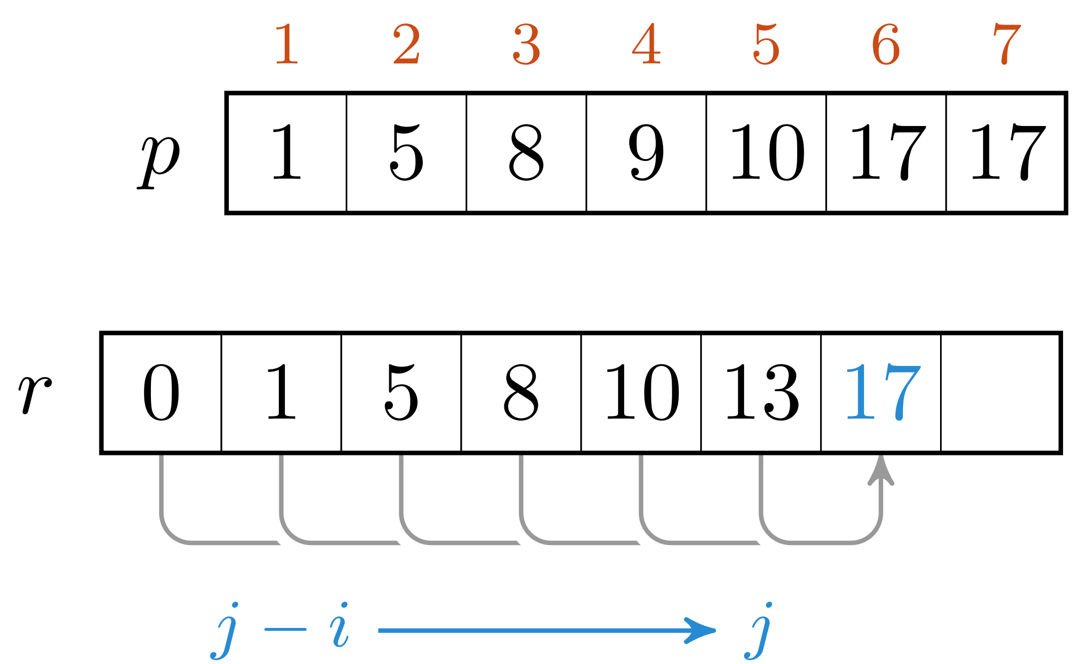
	1	2	3	4	5	6	7
p	1	5	8	9	10	17	17

r	0	1	5	8	10	13		

$$j - i \longrightarrow j$$

Her har vi j delinstanser – én for hvert mulig kuttsted. Vi kombinerer løsningene vha. max.

$$r[j] = \max_{i=1\ldots j} \left(p[i] + r[j-i]\right)$$



$p[1] + r[5] = 14$
$p[2] + r[4] = 15$
$p[3] + r[3] = 16$
$p[4] + r[2] = 14$
$p[5] + r[1] = 11$
$p[6] + r[0] = 17$

$$r[j] = \max_{i=1\ldots j} \left(p[i] + r[j-i] \right)$$

Oppgave

Nå vet vi hva den beste prisen er, men ikke hvordan vi skal kappe opp staven.

Hvordan vil du endre prosedyren for å finne ut dette?

Vent med s. 368–369 i boka til etterpå.

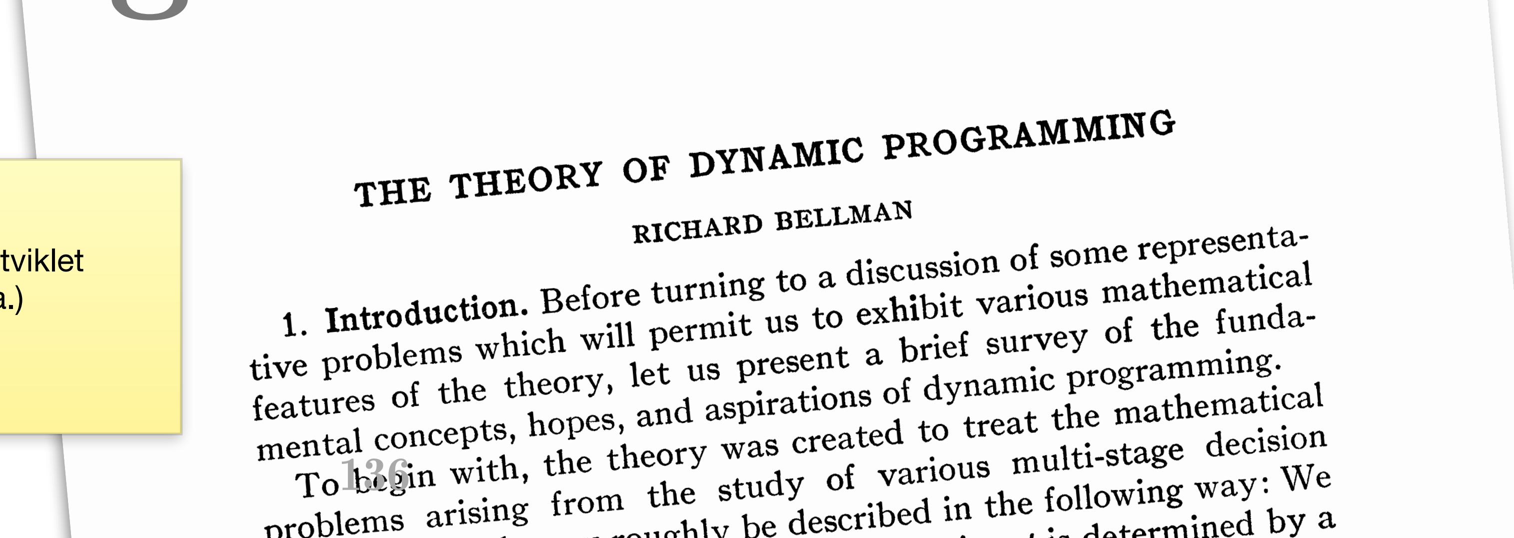
Tenk selv 0:30
Jobb sammen 2:00
Svar fra dere
Svar fra meg
Refleksjon 1:00

```
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Hva var egentlig metoden vi brukte for å løse stavkappingsproblemet...?

2:5

Dyn. prog. › Hva er det?



THE THEORY OF DYNAMIC PROGRAMMING

RICHARD BELLMAN

1. **Introduction.** Before turning to a discussion of some representative problems which will permit us to exhibit various mathematical features of the theory, let us present a brief survey of the fundamental concepts, hopes, and aspirations of dynamic programming.

To begin with, the theory was created to treat the mathematical problems arising from the study of various multi-stage decision

Fra 1954. (Han hadde utviklet ideene en del frem til da.)

Oppskrift fra boka

1. **Characterize the structure** of an optimal solution
2. **Recursively define the value** of an optimal solution
3. **Compute the value** of an optimal solution
4. **Construct** an optimal solution from computed information

FUNCTION(A)

A instans

En (ganske) generell algoritme!

Function(A)
1 S = Divide(A)

A instans
S delinstanser

Del i delinstanser/delproblemer (*subproblems*)

```
Function(A)
S = Divide(A)
n = S.length
```

A instans
S delinstanser

FUNCTION(A)

1 S = DIVIDE(A)
2 n = S.*length*
3 let R[$1 \,..\, n$] be a new array

A instans
S delinstanser
R delsvar

Mellomlagring av delsvar/resultater

FUNCTION(A)
S = DIVIDE(A)
n = S.*length*
let R[1 . . n] be a new array
for i = 1 **to** n

A instans
S delinstanser
R delsvar

```
FUNCTION(A)
  S = DIVIDE(A)
  n = S.length
  let R[1..n] be a new array
  for i = 1 to n
      R[i] = FUNCTION(S[i])
```

A instans
S delinstanser
R delsvar

Løs delinstanser rekursivt

```
FUNCTION(A)
    S = DIVIDE(A)
    n = S.length
    let R[1 .. n] be a new array
    for i = 1 to n
        R[i] = FUNCTION(S[i])
    return COMBINE(R)
```

A instans
S delinstanser
R delsvar

Kombinér delsvar til ett svar og returnér

```
FUNCTION(A)
S = DIVIDE(A)
n = S.length
let R[1..n] be a new array
for i = 1 to n
    R[i] = FUNCTION(S[i])
return COMBINE(R)
```

A instans
S delinstanser
R delsvar

Ofte vil A og S[i] være tupler av argumenter

Grunntilfellet er når n = 0.

```
FUNCTION(A)
S = DIVIDE(A)
n = S.length
let R[1 . . n] be a new array
for i = 1 to n
        R[i] = FUNCTION(S[i])
return COMBINE(R)
```

A instans
S delinstanser
R delsvar

Vi trenger bare DIVIDE og COMBINE!

Men er det ikke det vi har gjort til nå?

Svaret er: Jo. Det eneste vi egentlig legger til er mellomlagring av delløsninger.

A wide array of combinatorial optimization problems that are hard in general have been shown to be polynomially solvable in special cases via recursive computations usually termed *dynamic programming* in the discrete optimization literature and *divide-and-conquer* in computer science. The full problem is attacked by decomposing it into a recursive sequence of smaller ones, solving the latter subproblems in turn, and assembling a solution for the full problem from the results.

Med andre ord: Navnet har vært brukt litt forskjellig, og handlet opprinnelig om en spesifikk type optimeringsproblemer. Vi behandler det bare som enda et navn på det samme vi har gjort hele veien – men i den mest generelle formen, der vi tillater at samme delinstanser brukes på flere måter i dekomponeringen, i motsetning til det vi (i motsetning til det Martin, Rardin & Campbell sier) kaller divide-and-conquer, der vi *ikke* har overlapp melom delinstanser.

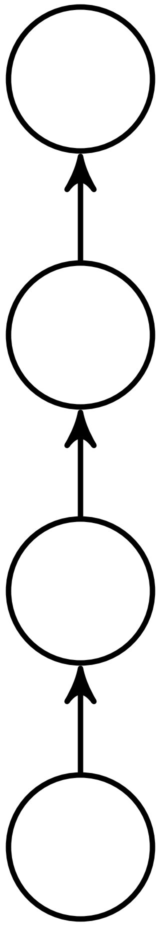
Martin, Rardin & Campbell (1990): Polyhedral Characterization of Discrete Dynamic Programming.

Vi lagrer all mellomregning/alle delløsninger, så de kan brukes om igjen, som i et regneark. (Vi kan naturligvis ikke ha sykliske avhengigheter.)

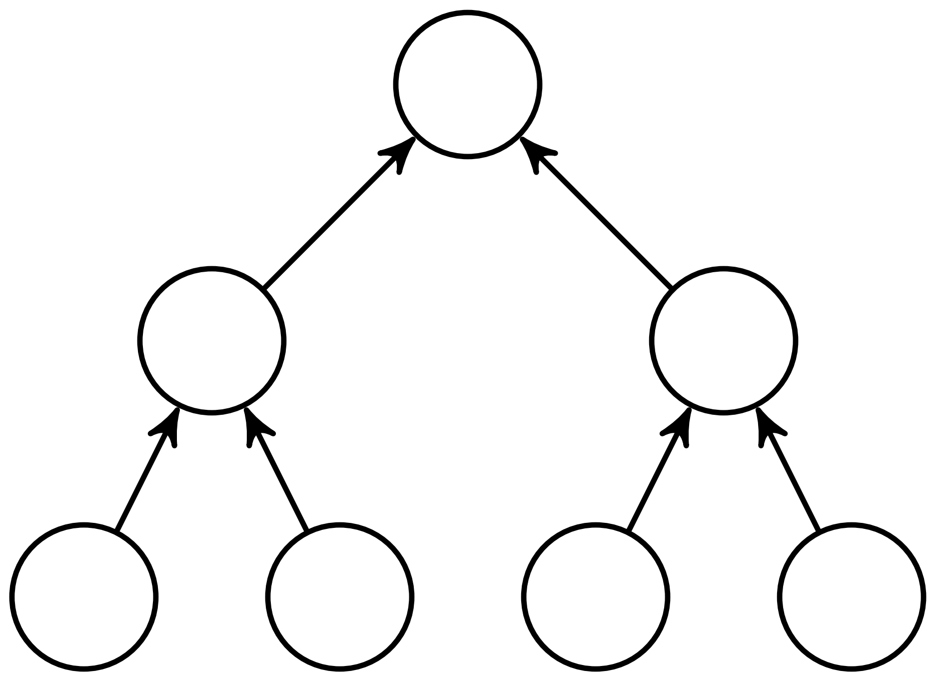
«Time–memory tradeoff»

Tenk regneark

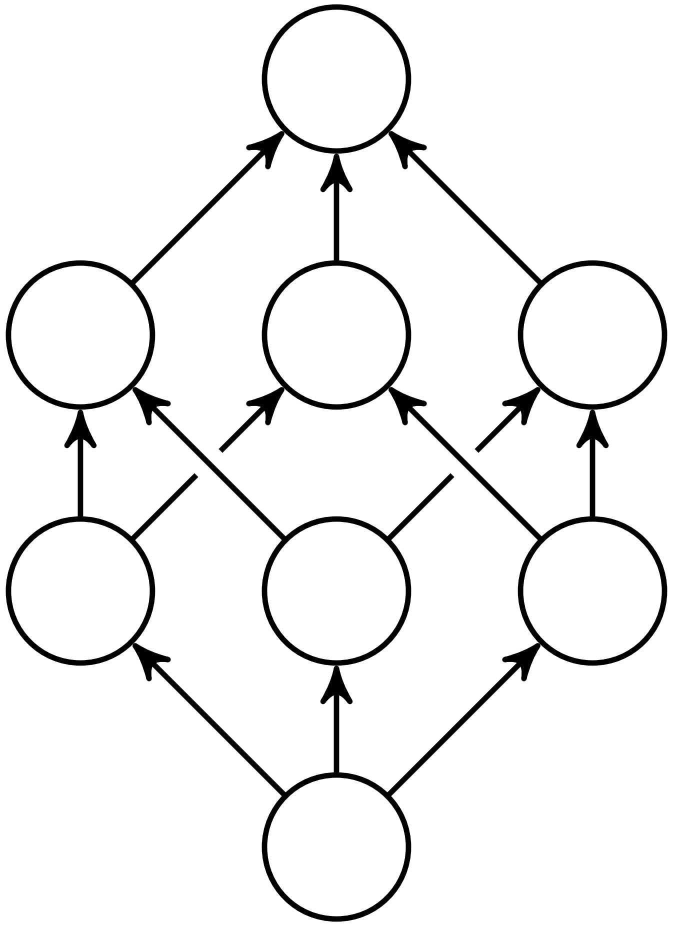
Bare nyttig hvis vi trenger noen av løsningene mer enn én gang, dog…



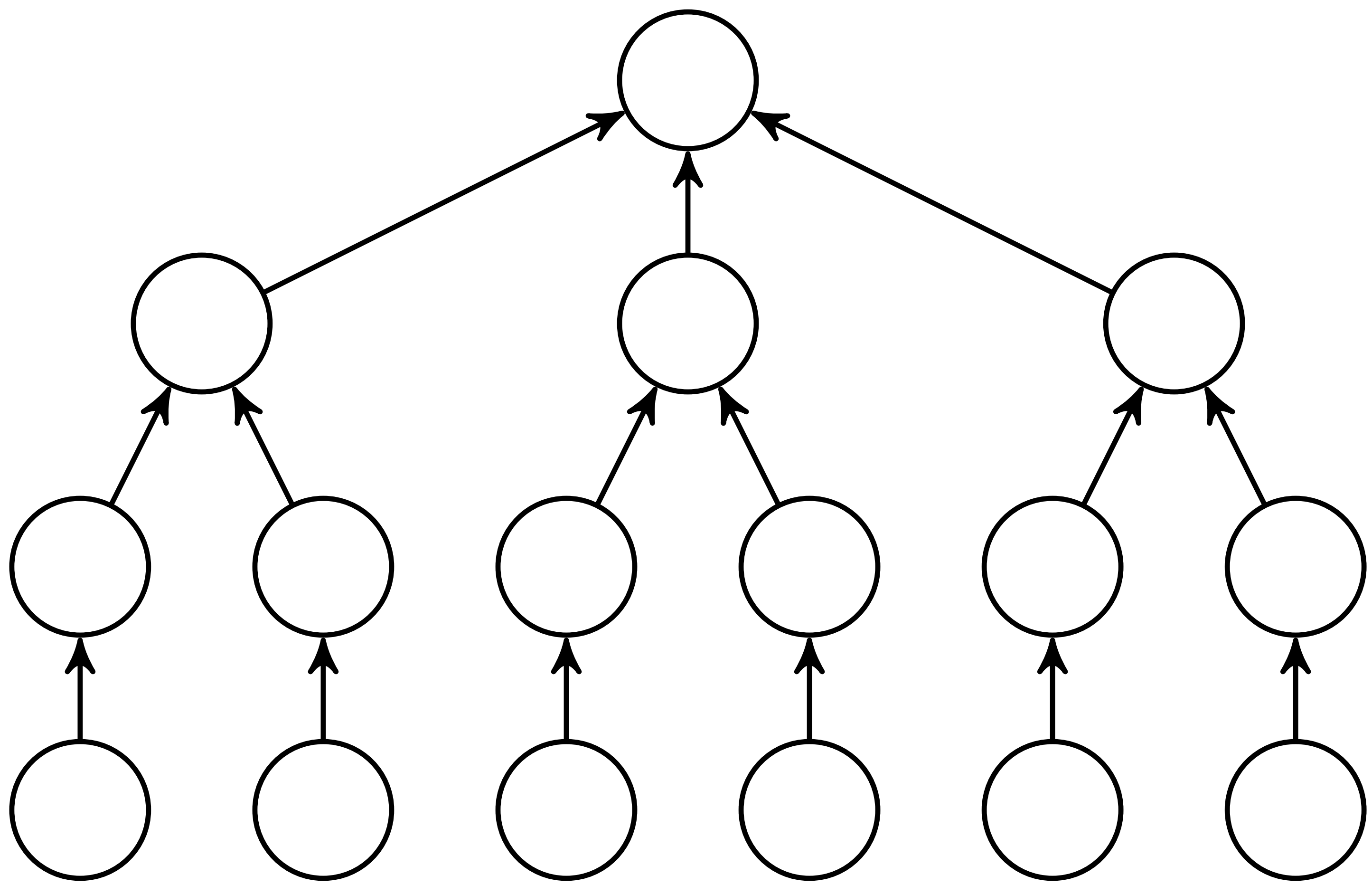
Inkrementell design



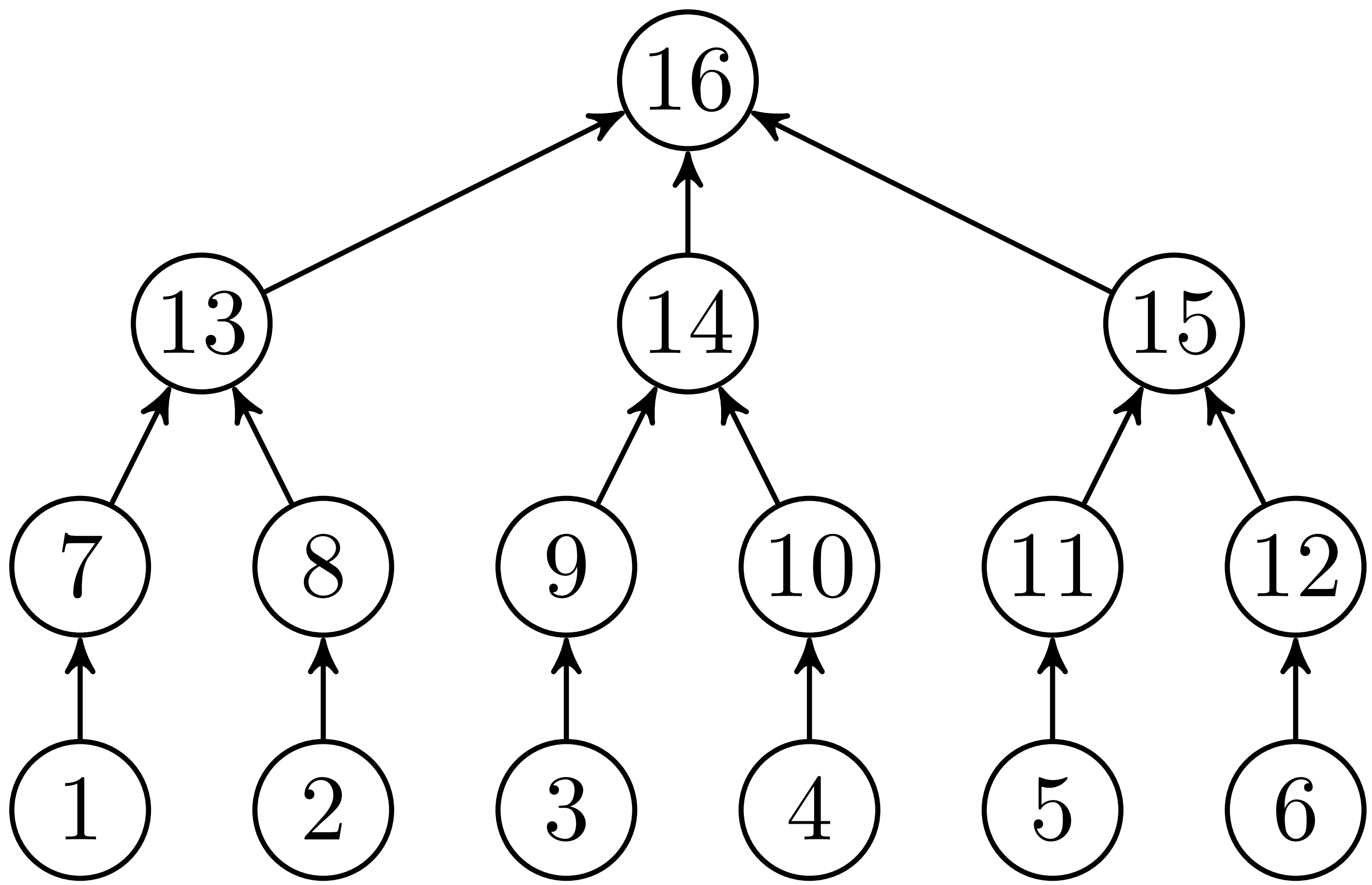
Uavhengige delproblemer: Splitt og hersk



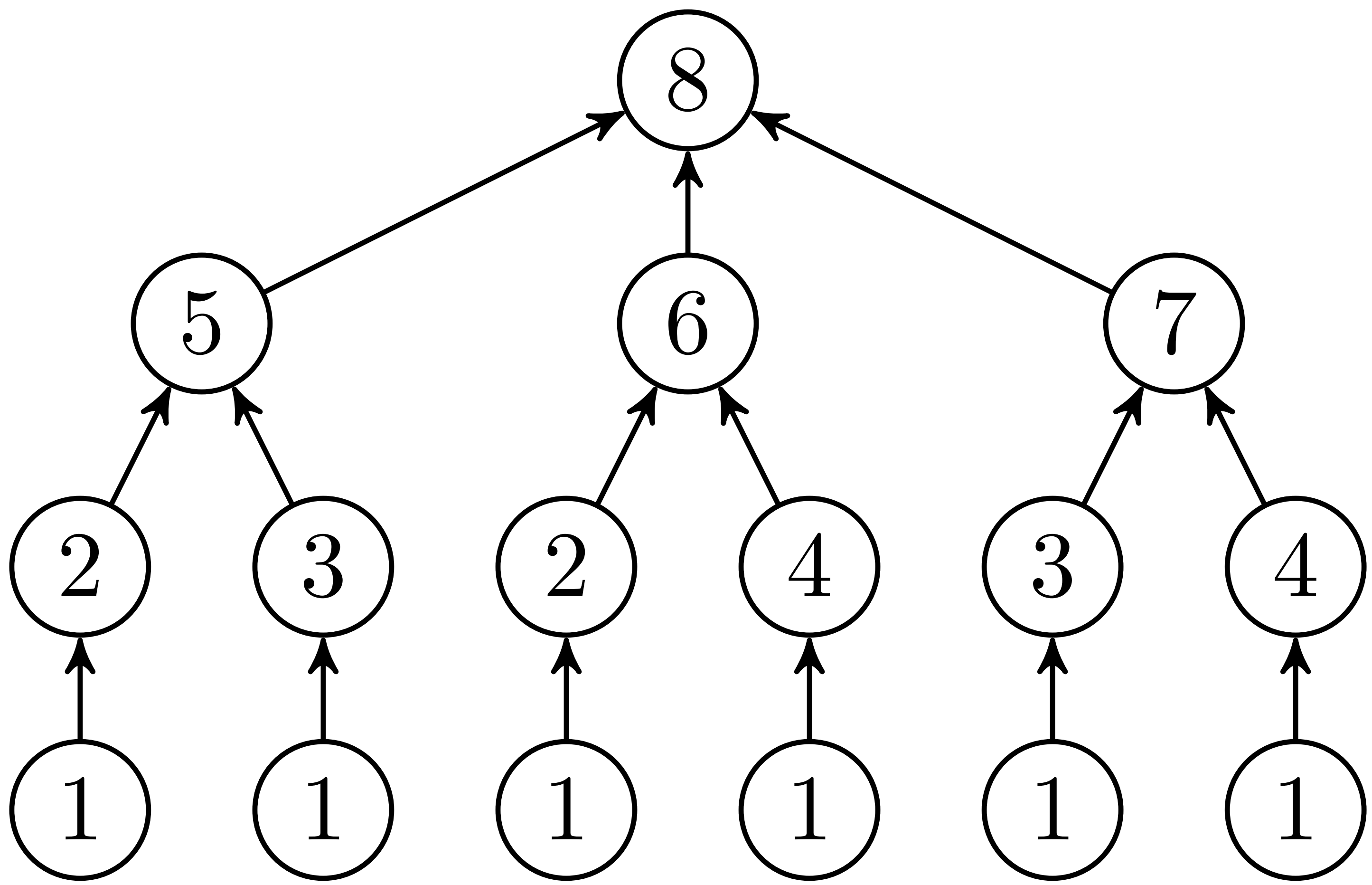
Overlappende delproblemer: Dynamisk programmering



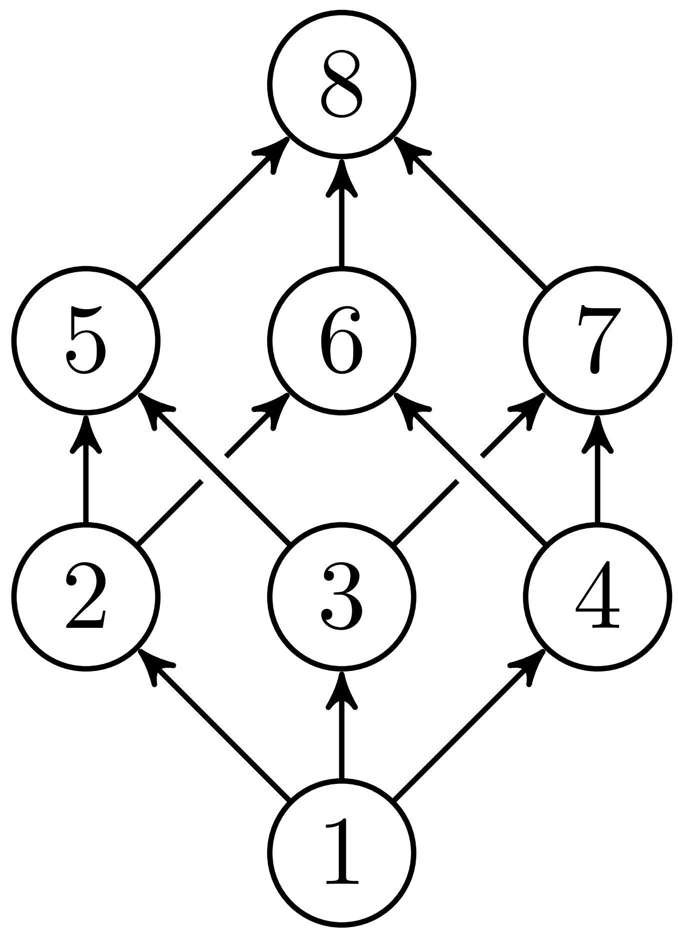
Uavhengige delproblemer: Splitt og hersk



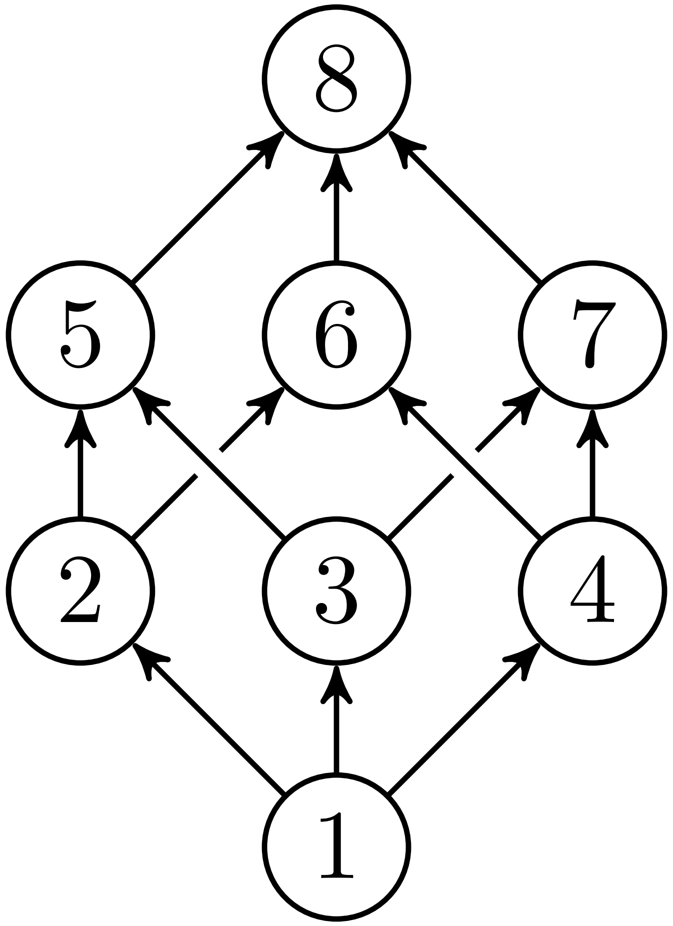
Uavhengige delproblemer: Splitt og hersk



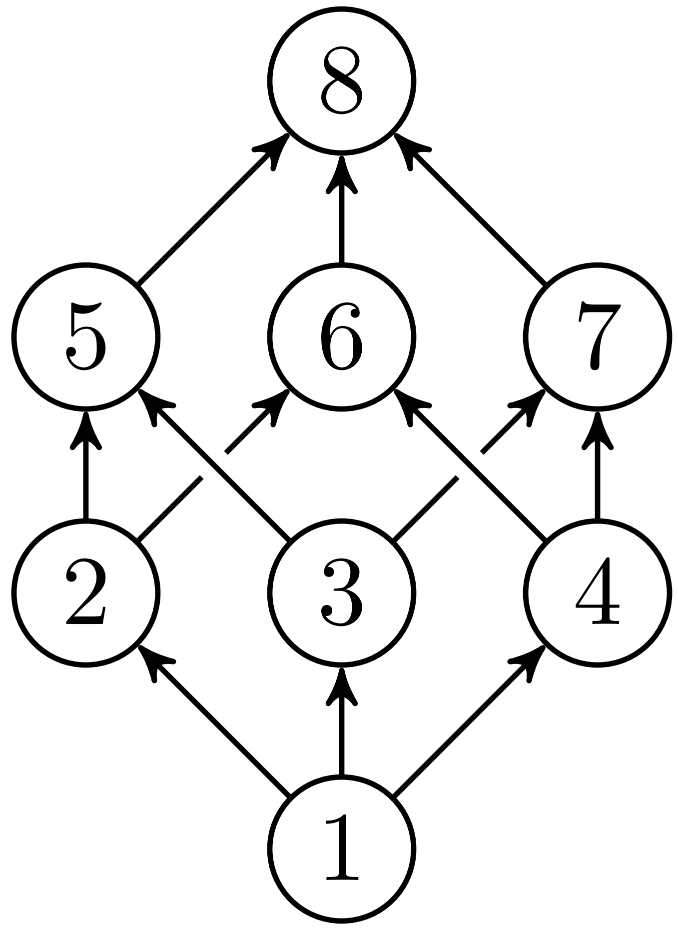
Overlappende og delte delproblemer ...



Overlappende og delte delproblemer …



Grafen har like mange stier som treet



Idé: Lagre hvert delsvar!

Nyttig når vi har overlappende delproblemer

Korrekt når vi har optimal substruktur

Optimal substruktur er noe vi har basert oss på i tidligere algoritmer òg – at vi bygger optimale løsninger ut fra optimale del-løsninger.

Memoisering

Memo-isering

Memo—isering

Memo———isering

Memo———isering

Gi funksjonen hukommelse: «Har jeg fått disse argumentene før?»

Sjekk f.eks. pakken Memoized.jl til Julia – med en @memoize-makro som gjør jobben for deg!

Se f.eks. https://algdat.idi.ntnu.no/faq/2019/08/14/hvordan-memoiserer-jeg-i-julia.html

Memo———————isering

Hvis ja: Returnér svaret du lagret sist gang!

FUNCTION'(A)

A instans

Samme beregning, men vi beskjærer overflødige deler av kalltreet

FUNCTION'(A)

8 **return** F[A]

A instans
F memo

Jeg kaller tabellen (evt. hashtabellen) «en memo», her. Det er ikke nødvendigvis 100% standard terminologi, men det er praktisk å ha et navn på den. (Man kunne argumentere for at det burde være «et memo», siden ordet «memoisering» stammer fra «memo» som i «memorandum»; men siden dette er en annen betydning enn dette, er det kanskje like greit å bruke en egen bøyninsform – som matcher andre lignende objekter som «hashtabell», e.l., selv om det naturligvis er litt vilkårlig.)

Har vi alt svaret? Returnér det; beregning overflødig!

```
Function'(A)
  if F[A] == nil

  return F[A]
```

A instans
F memo

Hvis ikke . . .

```
FUNCTION'(A)
if F[A] == NIL
    S = DIVIDE(A)
    n = S.length
    let R[1..n] be a new array
    for i = 1 to n
        R[i] = FUNCTION'(S[i])
    F[A] = COMBINE(R)
return F[A]
```

A instans
F memo
S delinstanser
R delsvar

Samme prosedyre som før!

```
FUNCTION'(A)
if F[A] == NIL
    S = DIVIDE(A)
    n = S.length
    let R[1..n] be a new array
    for i = 1 to n
        R[i] = FUNCTION'(S[i])
    F[A] = COMBINE(R)
return F[A]
```

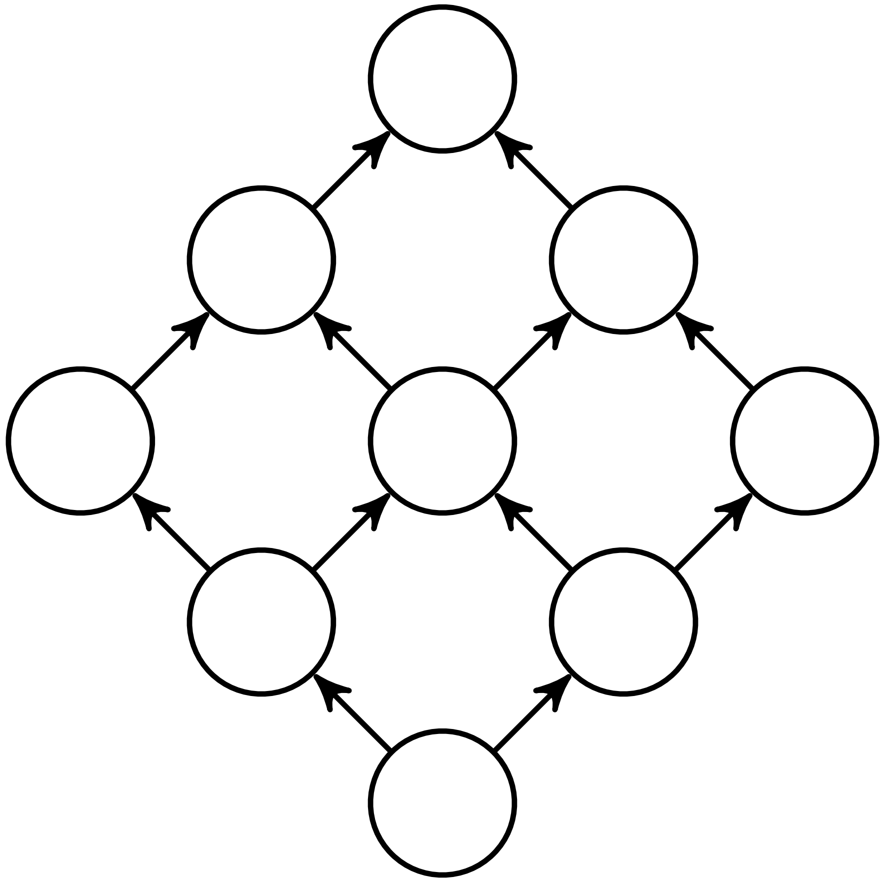
A instans
F memo
S delinstanser
R delsvar

Som før: Vi trenger bare DIVIDE og COMBINE!

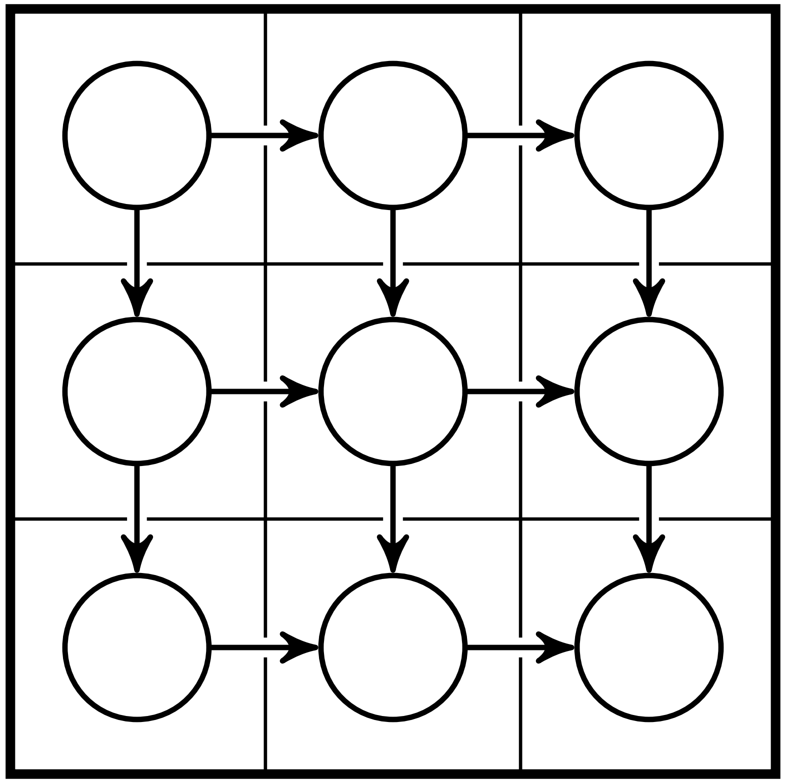
Bottom-up

Iterasjon over alle delinstanser. I stedet for rekursjon: Slå opp i løsninger du alt har regnet ut og lagret i en tabell.

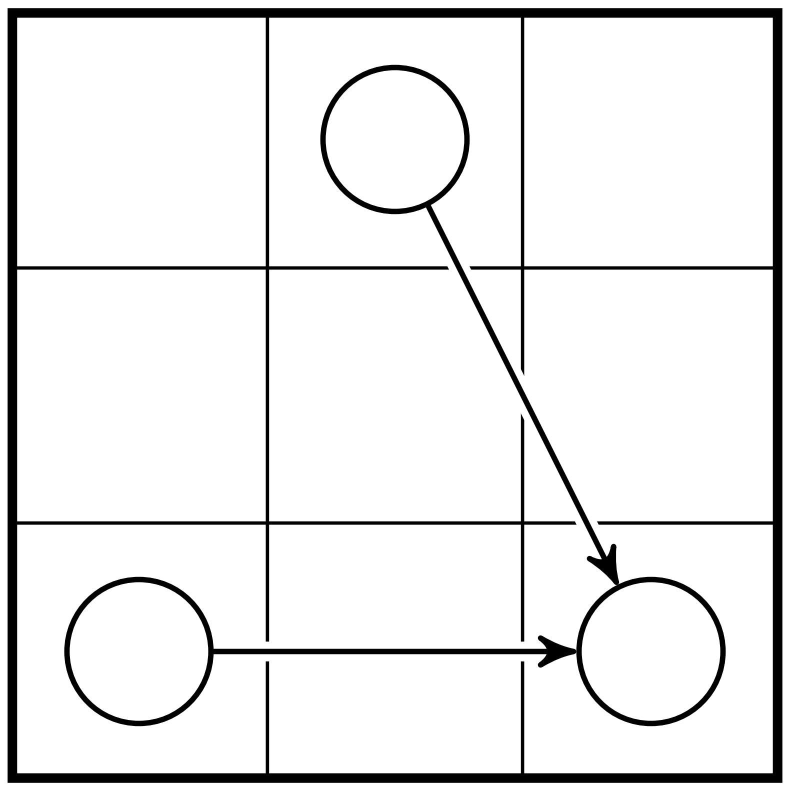
Vi trenger ikke begrense oss til én dimensjon!



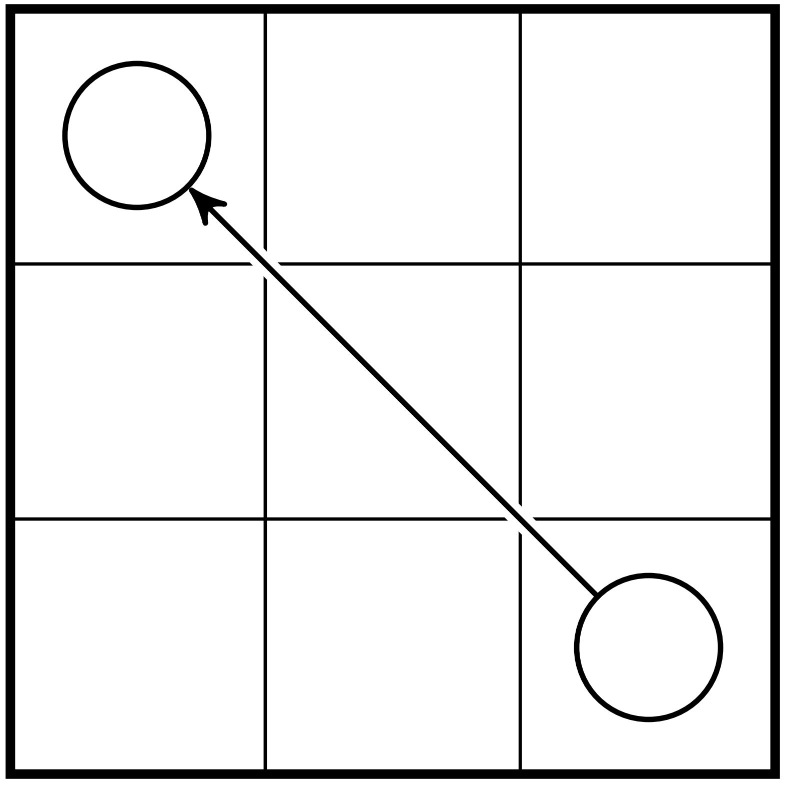
Foreløpig forenkling: Delproblemer i «rutenett»



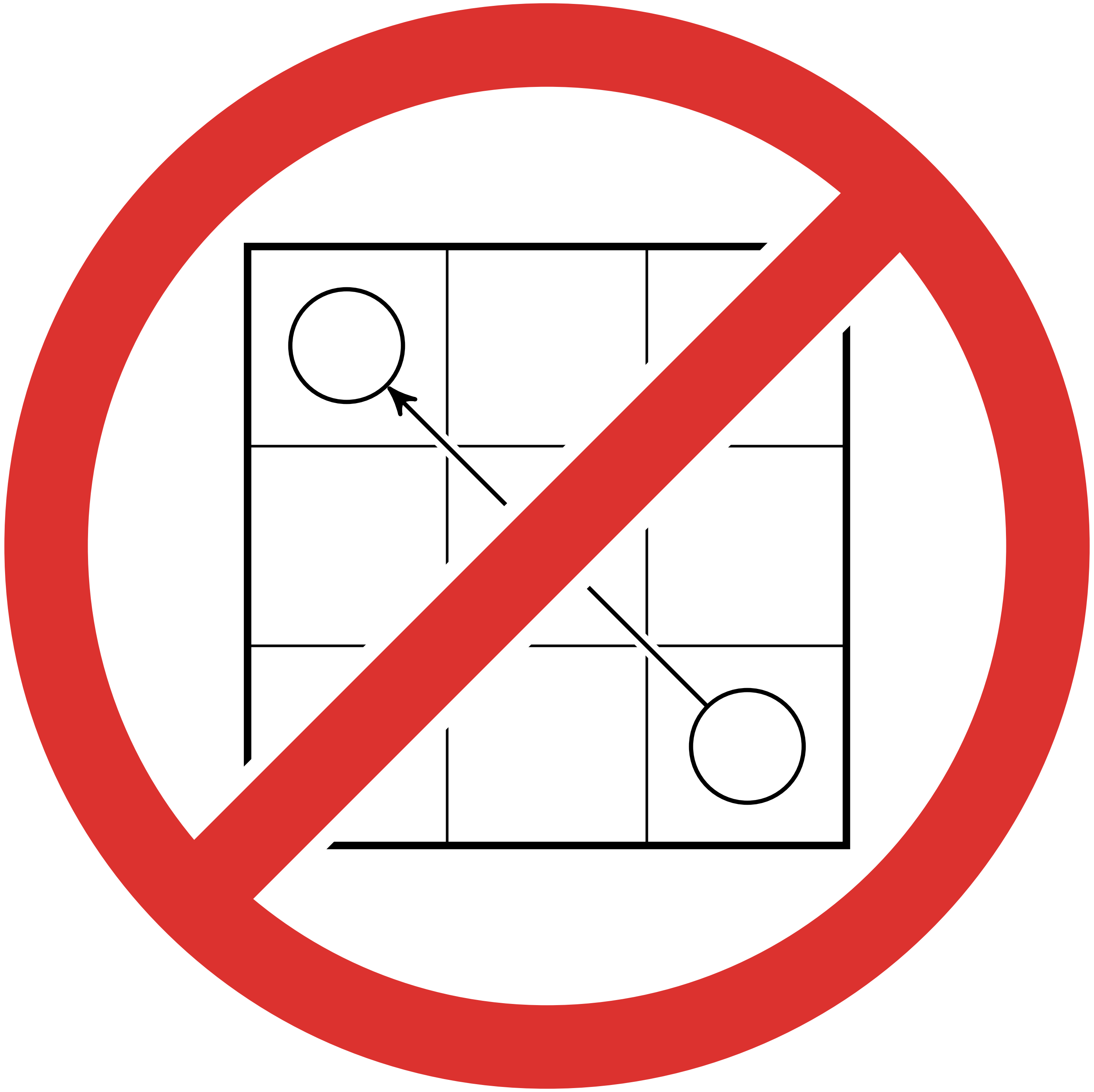
Lar oss lagre løsninger i en tabell



Kan ha mer rotete avhengigheter . . .

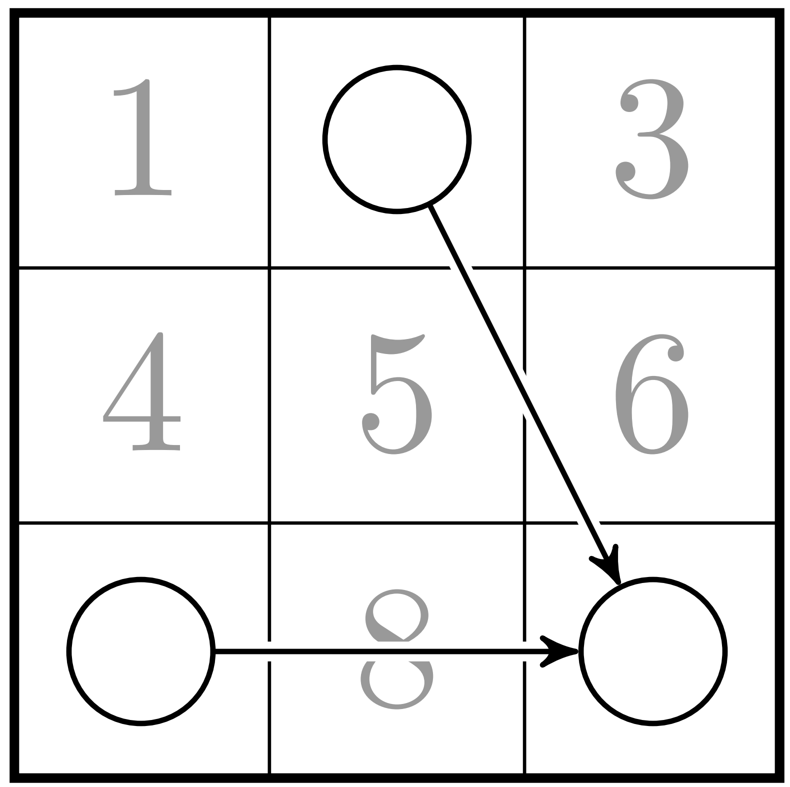


. . . men de kan ikke gå opp eller til venstre . . .



... men de kan ikke gå opp eller til venstre ...

Det er bare i spesielt ryddige tilfeller at vi kan organisere grafen som her, der hver kombinasjon av delinstans-parametre gir oss en celle i en tabell (array). Det er nyttig om vi vil lagre delløsninger i en slik tabell – mer generelt kan vi lagre løsninger i en hashtabell, eller direkte i strukturen vi ser på (f.eks. attributter i noder i en graf vi jobber med, e.l.). Det er i grunnen bare en implementasjonsdetalj.



… fordi vi vil jobbe radvis (eller kolonnevis)

3:5

Eksempel: LCS

Input: To sekvenser, $X = \langle x_1, \ldots, x_m \rangle$ og $Y = \langle y_1, \ldots, y_n \rangle$.

Output: En sekvens $Z = \langle z_1, \ldots, z_k \rangle$ og indekser $i_1 \leq \cdots \leq i_k$ og $\ell_1 \leq \cdots \leq \ell_k$ der $z_{i_j} = x_j$ og $z_{\ell_j} = y_j$ for $j = 1 \ldots k$, og der Z har maksimal lengde.

Input: To sekvenser, $X = \langle x_1, \ldots, x_m \rangle$ og $Y = \langle y_1, \ldots, y_n \rangle$.

Output: En sekvens $Z = \langle z_1, \ldots, z_k \rangle$ og indekser $i_1 \leq \cdots \leq i_k$ og $\ell_1 \leq \cdots \leq \ell_k$ der $z_{i_j} = x_j$ og $z_{\ell_j} = y_j$ for $j = 1 \ldots k$, og der Z har maksimal lengde.

1	2	3	4	5	6	7
k	l	a	p	p	e	r

1	2	3	4	5	6	7
t	a	k	p	a	p	p

Først: Kartlegg delproblemer

1	2	3	4	5	6	7
`k`	`l`	`a`	`p`	`p`	`e`	`r`

i

1	2	3	4	5	6	7
`t`	`a`	`k`	`p`	`a`	`p`	`p`

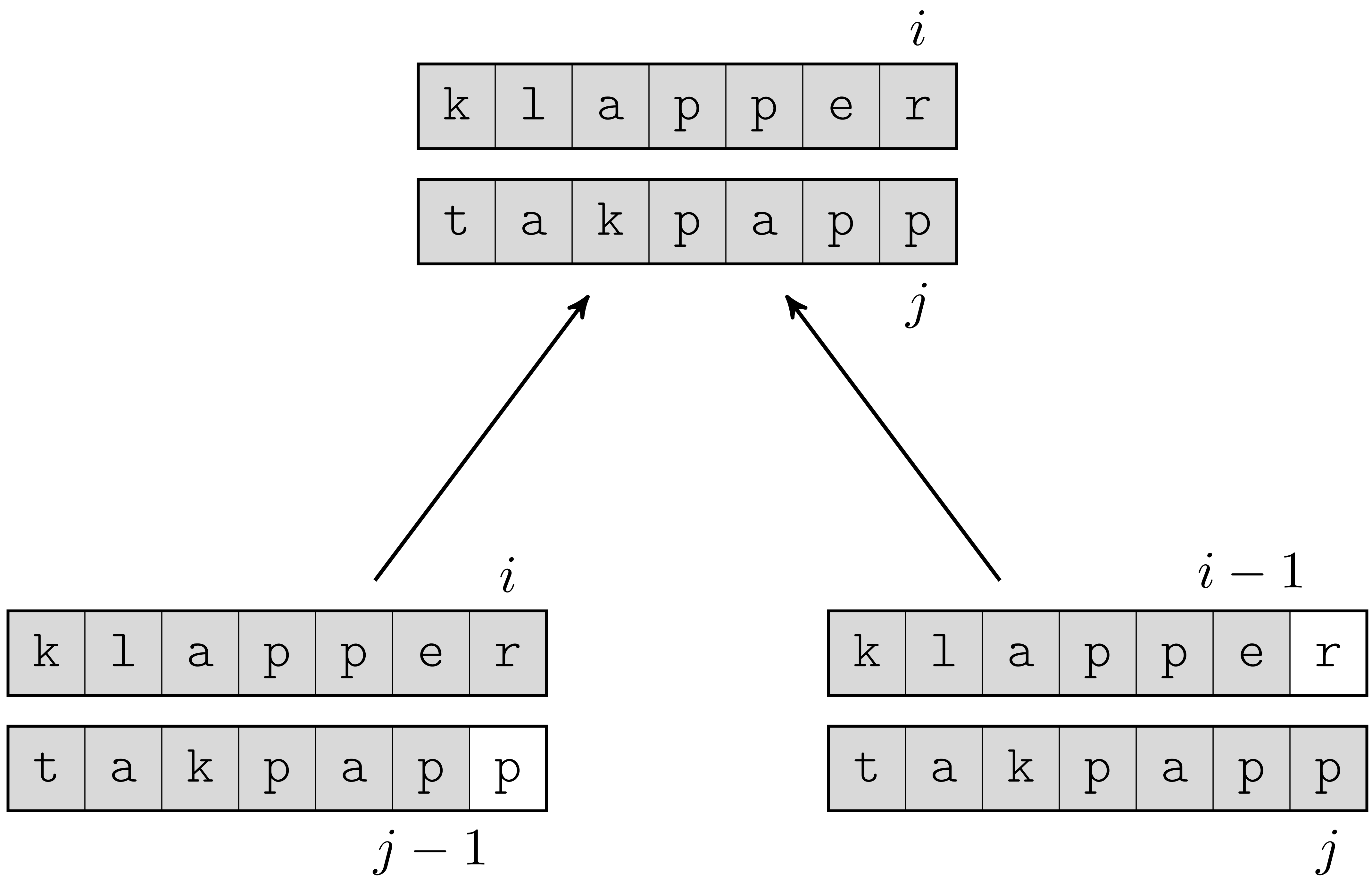
j

En parameter per sekvens?

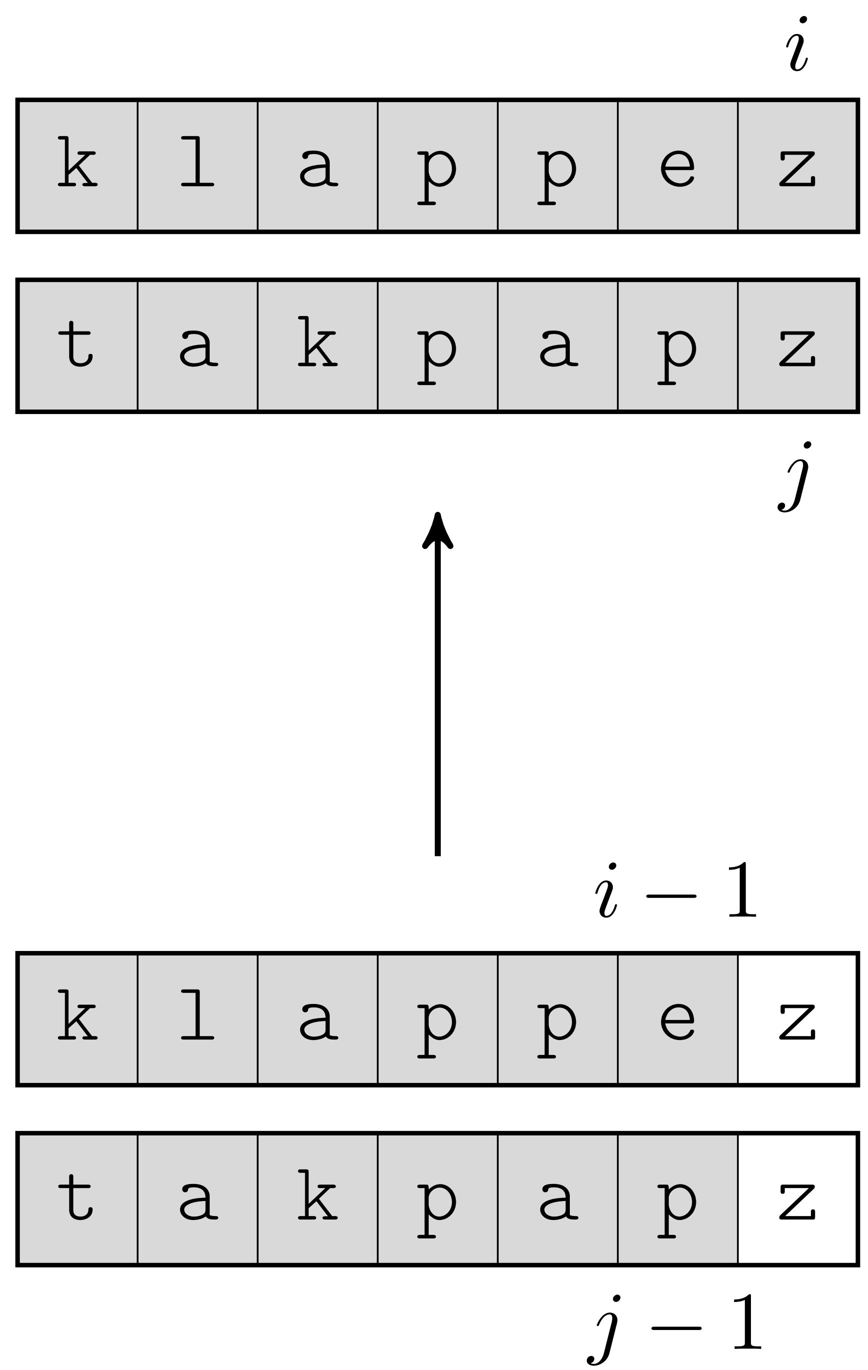
1	2	3	4	5	6	7
k	l	a	p	p	e	r
				i		

1	2	3	4	5	6	7
t	a	k	p	a	p	p
			j			

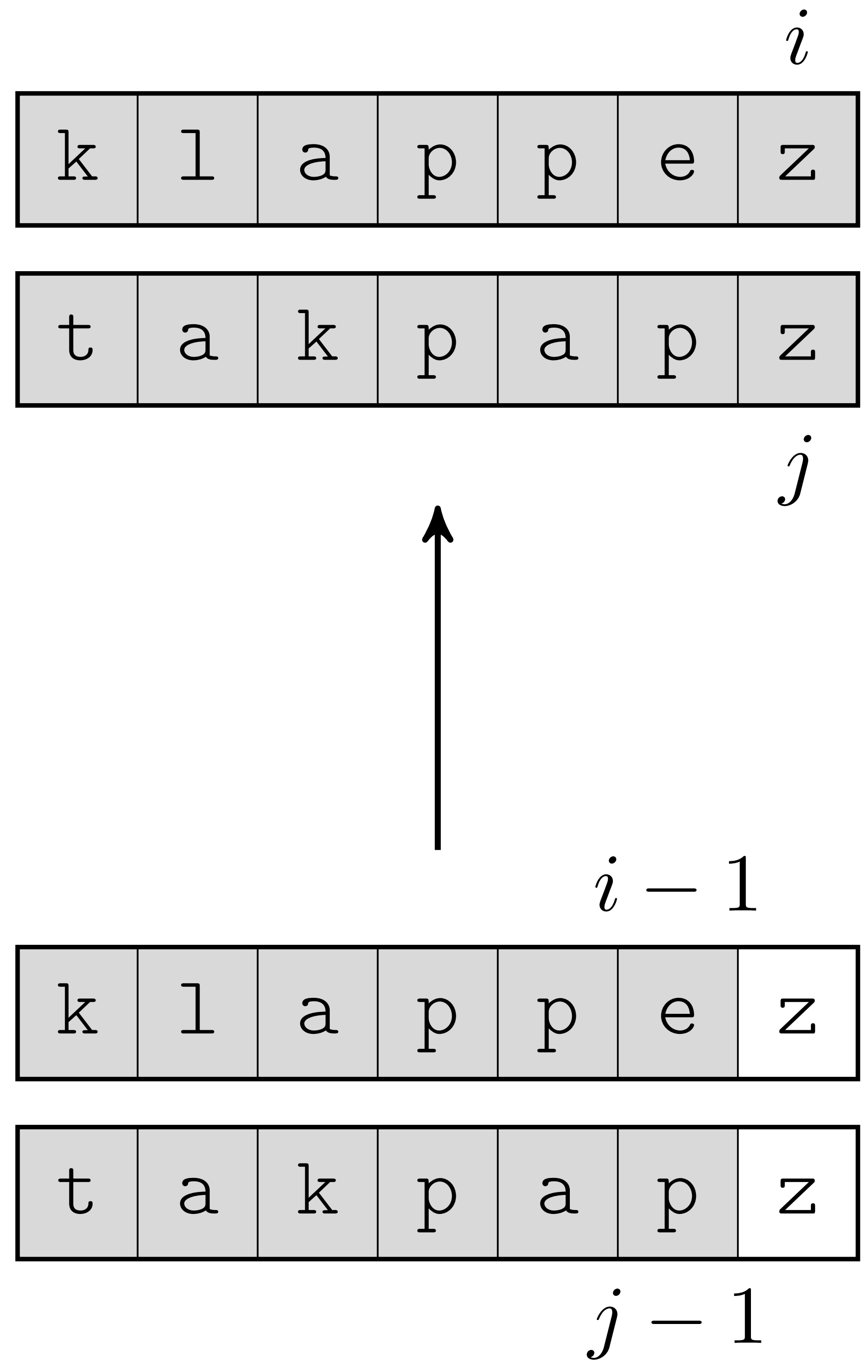
Prefiks, som i Rod-Cutting?



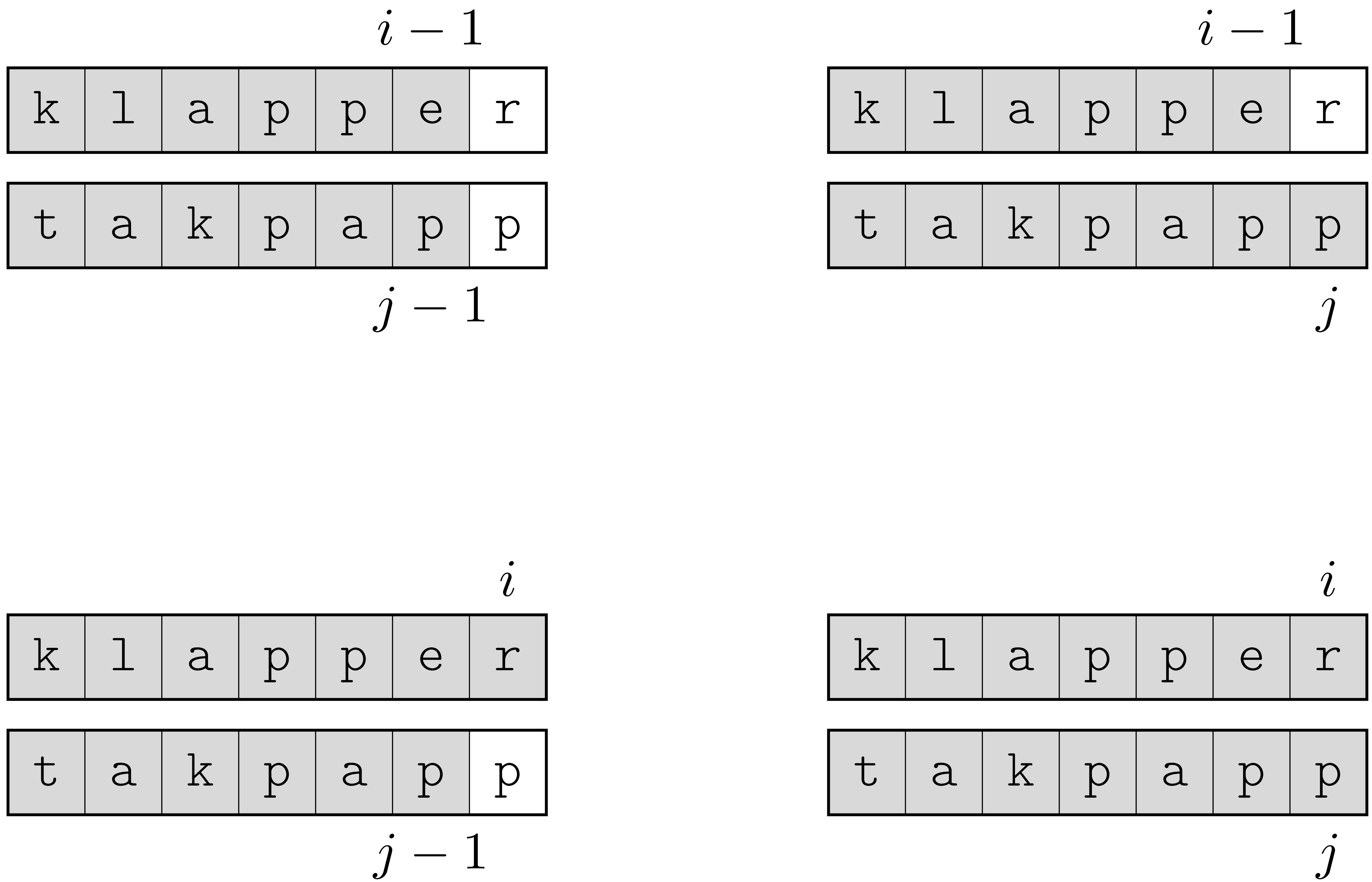
Ulike siste-elementer gir to delproblemer



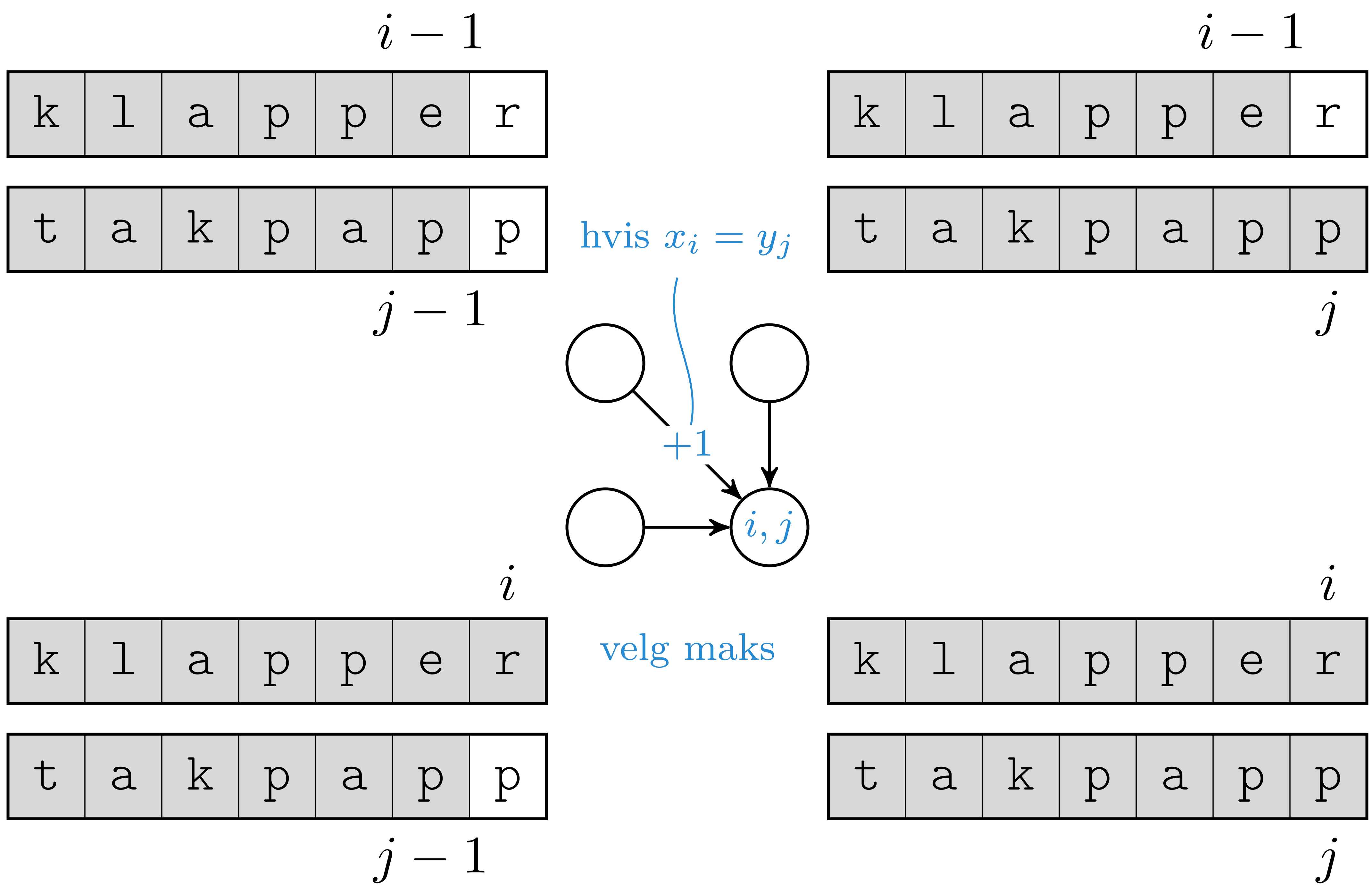
Like siste-elementer gir ett delproblem



Lønner seg alltid å inkludere like siste-elementer i løsningen

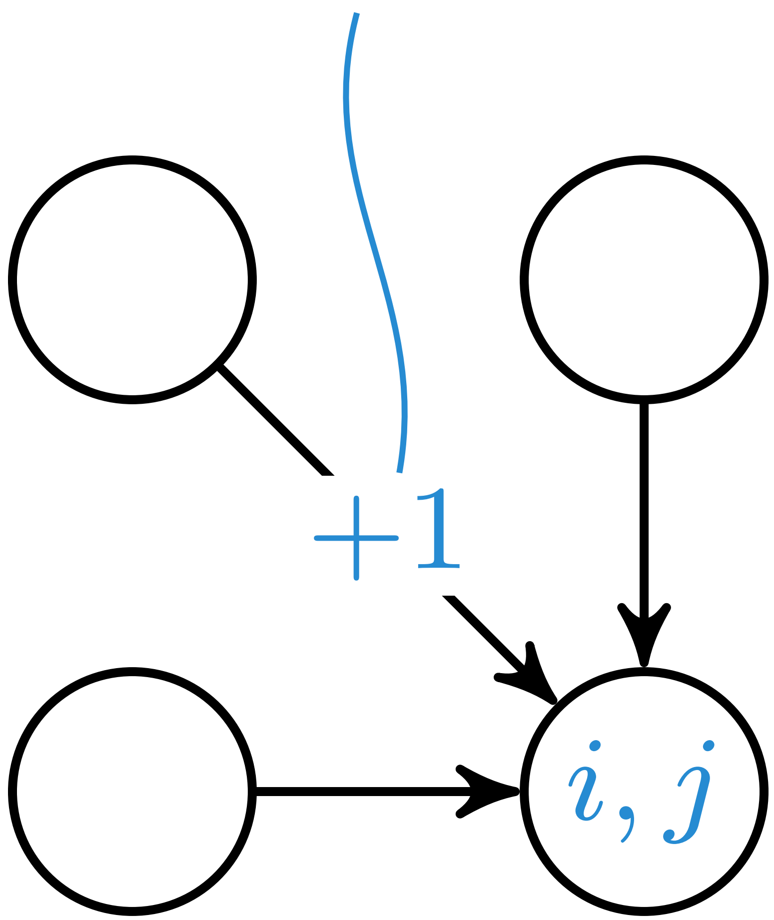


Endelig dekomponering



Endelig dekomponering

hvis $x_i = y_j$

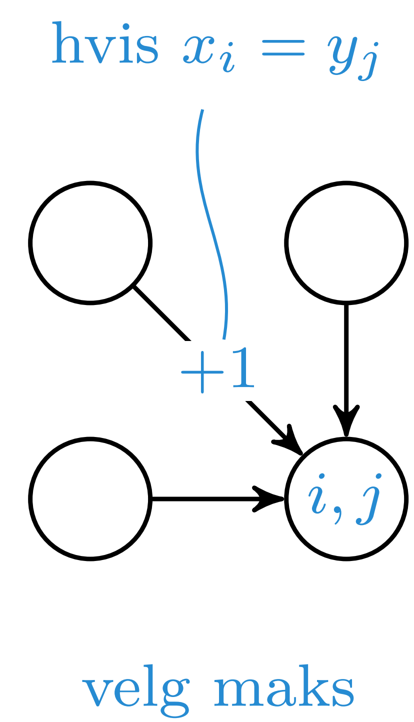


velg maks

Y

			s	t	o	r	m	k	a	s	t
		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
a	1	0									
t	2	0									
o	3	0									
m	4	0									
m	5	0									
a	6	0									
k	7	0									
t	8	0									

X

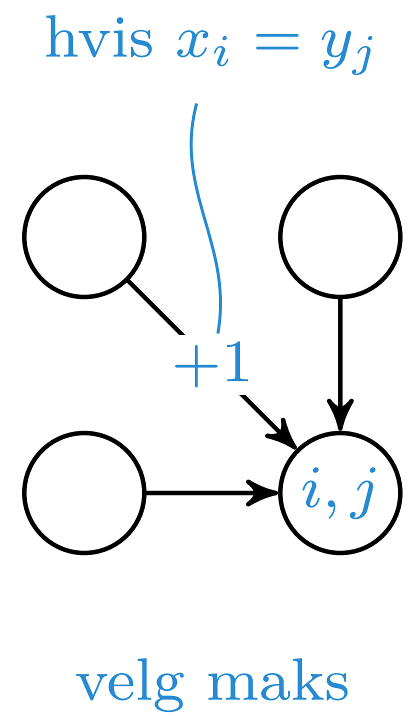


Skal vi hoppe over x_i og/eller y_j?

Y

		s	t	o	r	m	k	a	s	t
		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	1	1	1
t	2	0	0	1	1	1	1	1	1	1	2
o	3	0	0	1	2	2	2	2	2	2	2
m	4	0	0	1	2	2	3	3	3	3	3
m	5	0	0	1	2	2	3	3	3	3	3
a	6	0	0	1	2	2	3	3	4	4	4
k	7	0	0	1	2	2	3	4	4	4	4
t	8	0	0	1	2	2	3	4	4	4	5

X



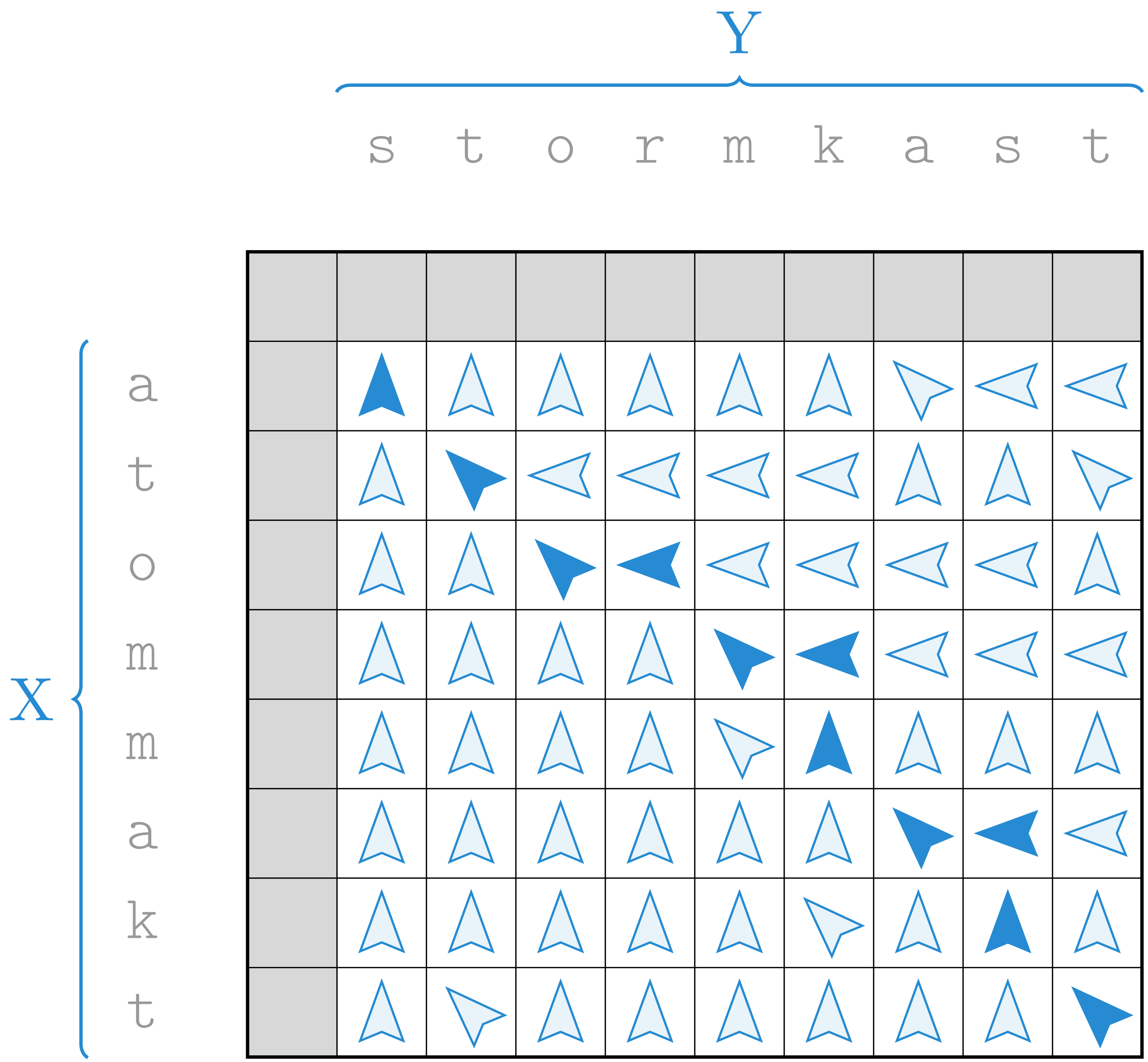
Skal vi hoppe over x_i og/eller y_j?

Y

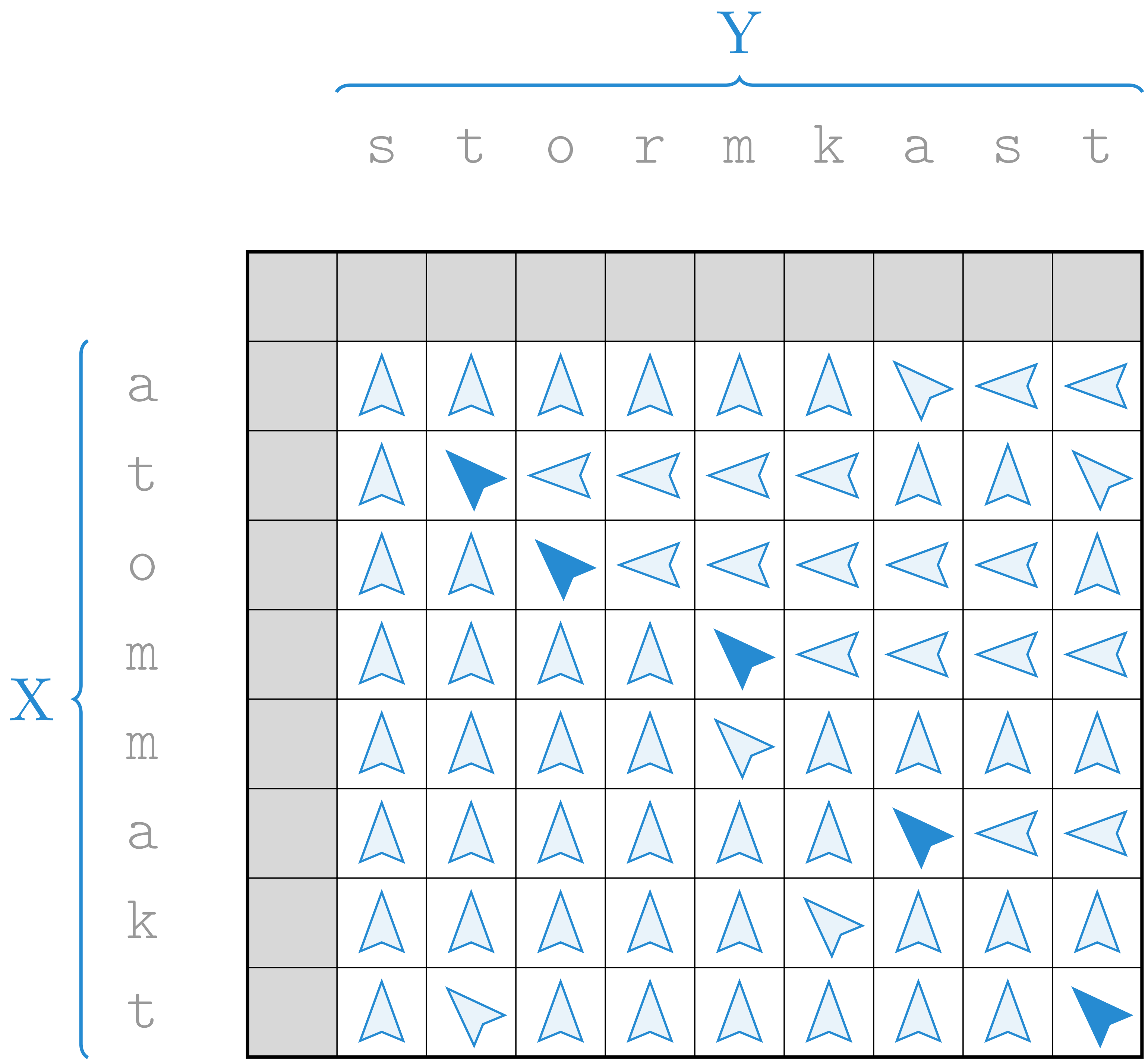
X

		s	t	o	r	m	k	a	s	t
a		↑	↑	↑	↑	↑	↑	↖	←	←
t		↑	↖	←	←	←	←	↑	↑	↖
o		↑	↑	↖	←	←	←	←	←	↑
m		↑	↑	↑	↑	↖	←	←	←	←
m		↑	↑	↑	↑	↖	↑	↑	↑	↑
a		↑	↑	↑	↑	↑	↑	↖	←	←
k		↑	↑	↑	↑	↑	↖	↑	↑	↑
t		↑	↖	↑	↑	↑	↑	↑	↑	↖

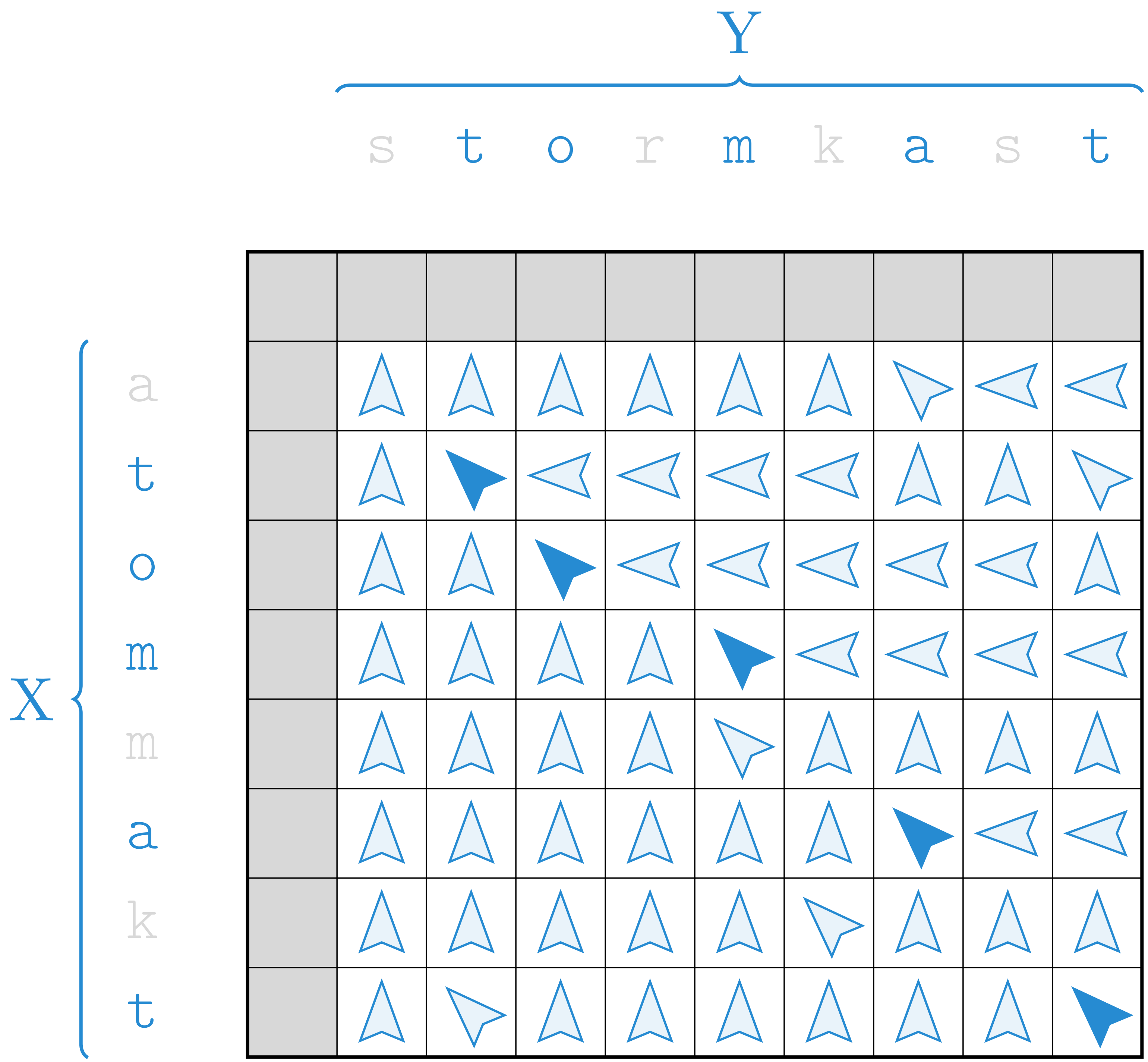
Hvilken delløsning bygger løsning (i, j) på?



Hvilke delløsninger bidro til løsning (n, m)?



Hvilke elementer hoppet vi ikke over?



Hvilke elementer hoppet vi ikke over?

Oppgave

Hvor mange slettinger og innsettinger kreves for å gjøre den første strengen om til den andre?

Hvordan kan vi løse problemet generelt, hvis vi også tillater å erstatte tegn?

Se også oppg. 15-5 i boka.

a	s
t	t
o	o
m	r
m	m
a	k
k	a
t	s
	t

Tenk selv	0:30
Jobb sammen	2:00
Svar fra dere	
Svar fra meg	
Refleksjon	1:00

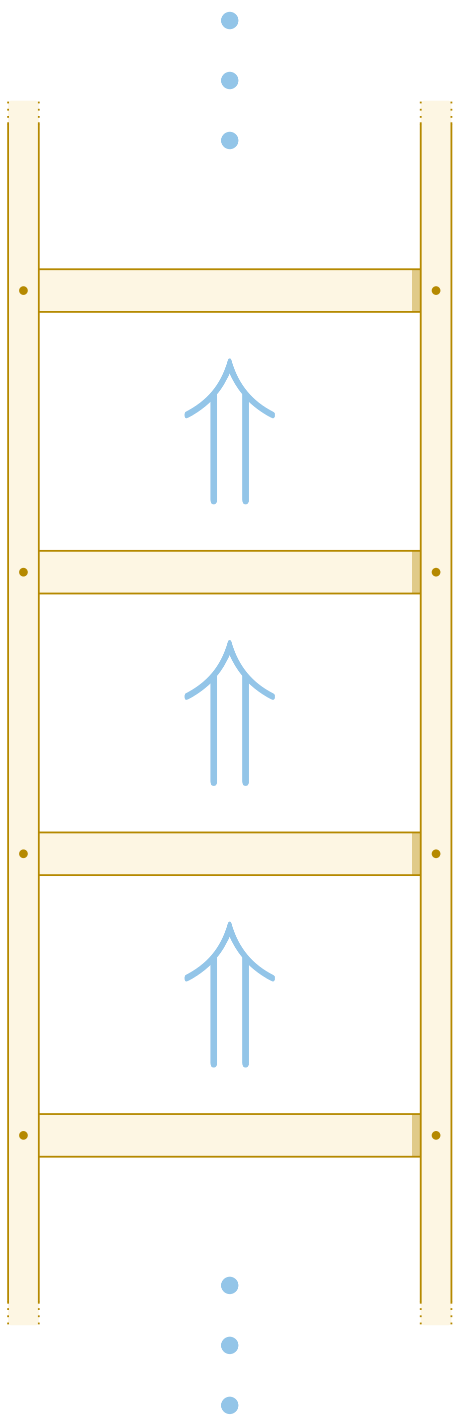
4:5

Optimal delstruktur

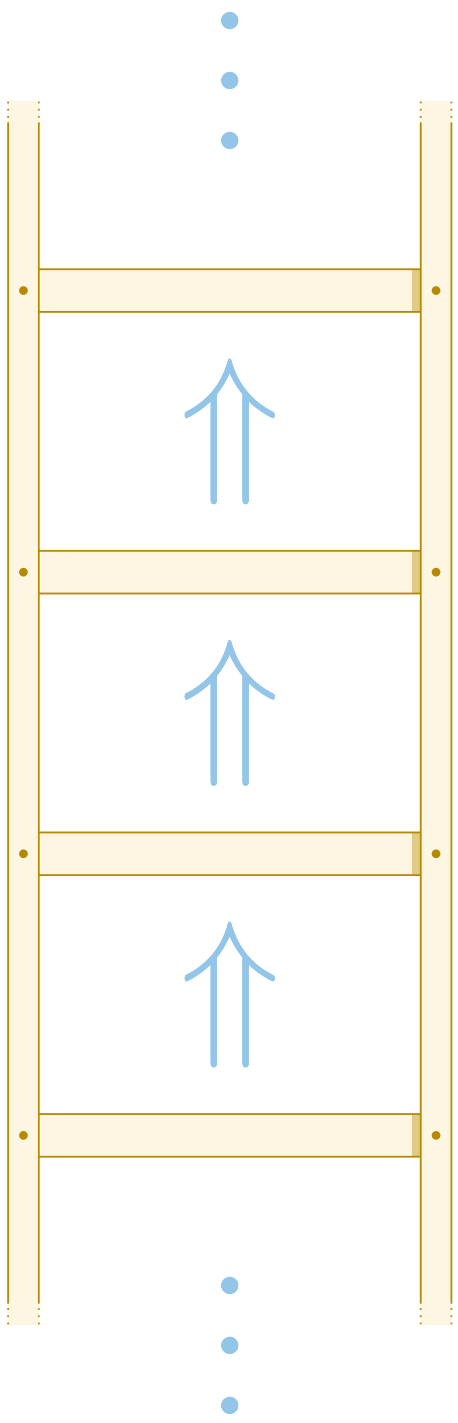
Det han omtaler som «remaining decisions» er det vi tenker på som delproblemer (selv om det høres litt bakvent ut). Hva vi gjr i vårt ene induktive trinn kommer an på optimale løsninger på delproblemene – og det forteller oss hva konsekvensene blir.

PRINCIPLE OF OPTIMALITY. *An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decisions.*

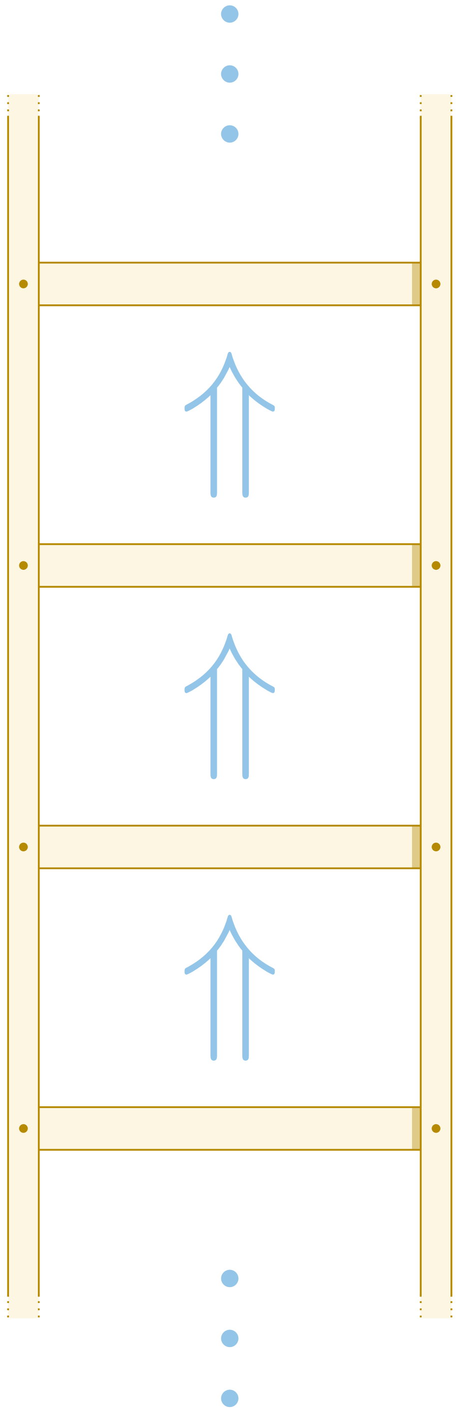
Richard Bellman, «The theory of dynamic programming», Bull. Amer. Math. Soc. 60 (1954), 503-515.



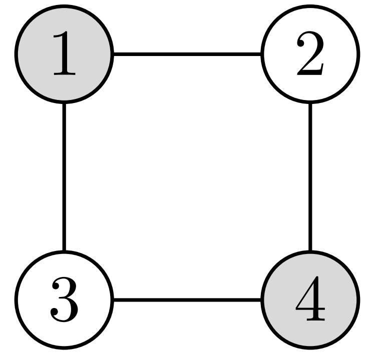
Opt. delstrukt.: Det finnes opt. løsning bestående av opt. delløsninger



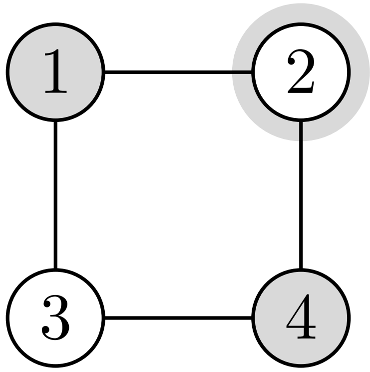
Dette gir «smitte-effekten» vi trenger, dvs., det induktive trinnet!



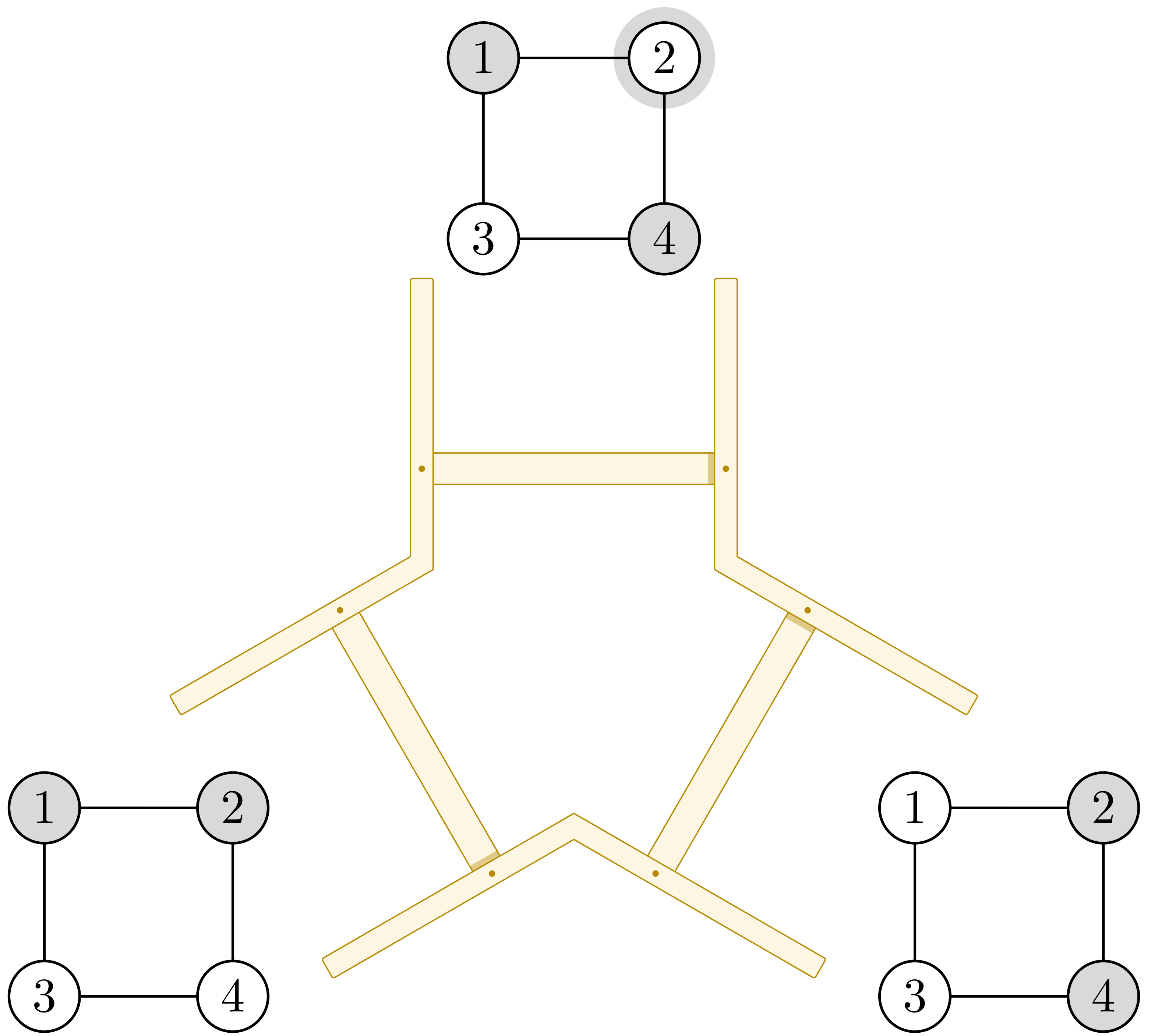
Finner optimale delløsninger; konstruerer optimal løsning



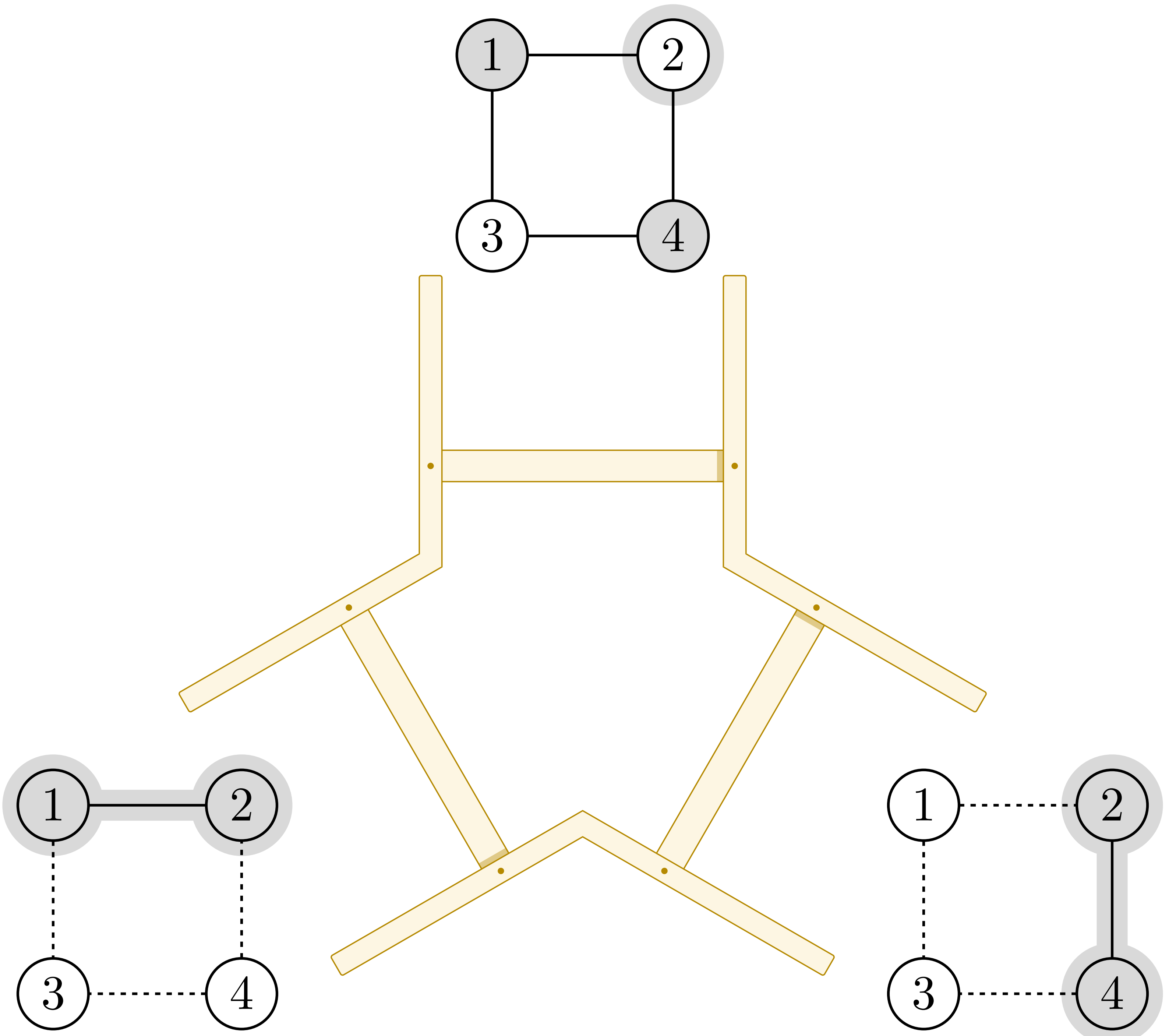
Vi vil finne korteste vei fra 1 til 4



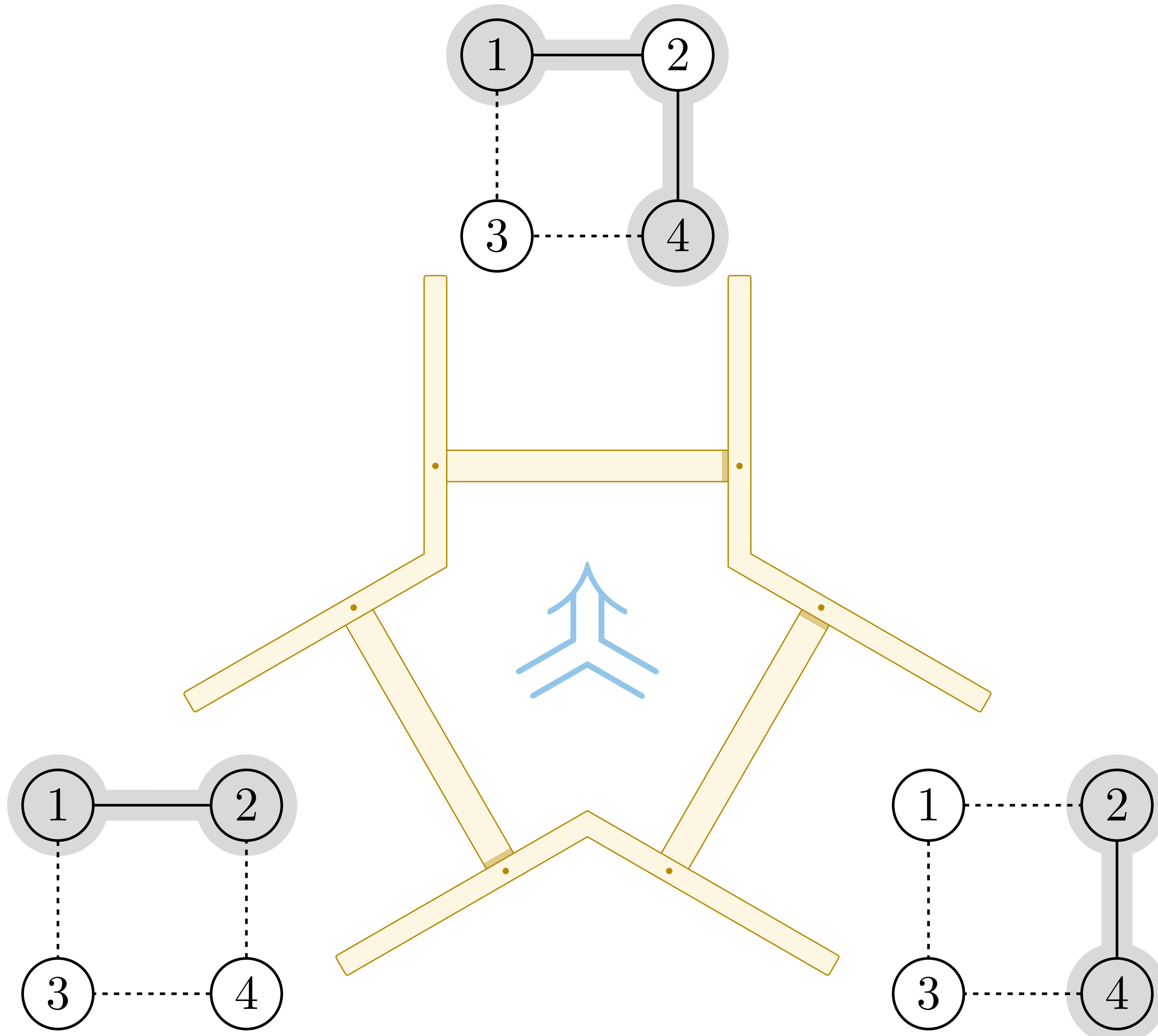
Vi velger tentativt å gå innom 2



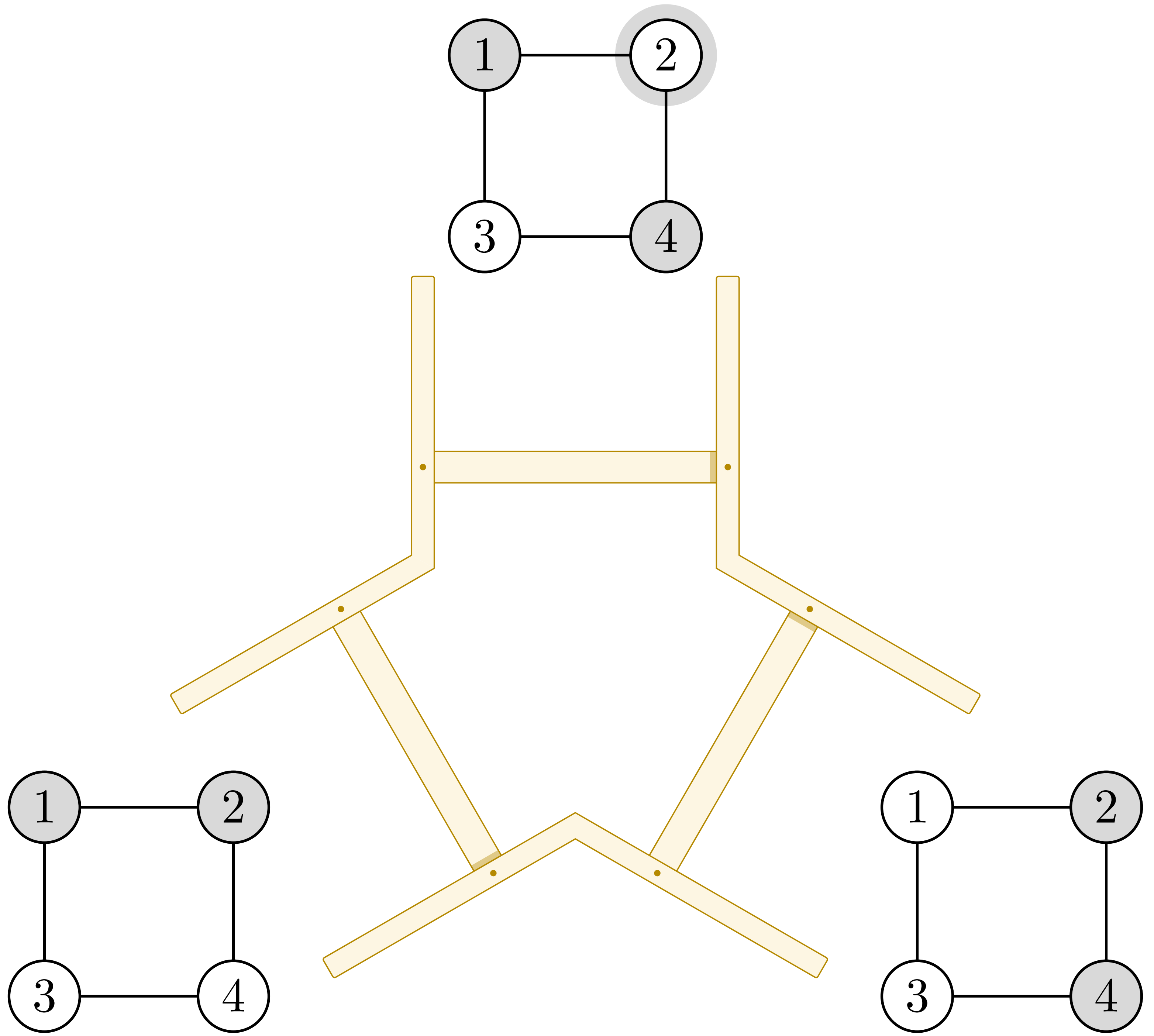
Kan dekomponere i korteste veier $1 \rightsquigarrow 2$ og $2 \rightsquigarrow 4$



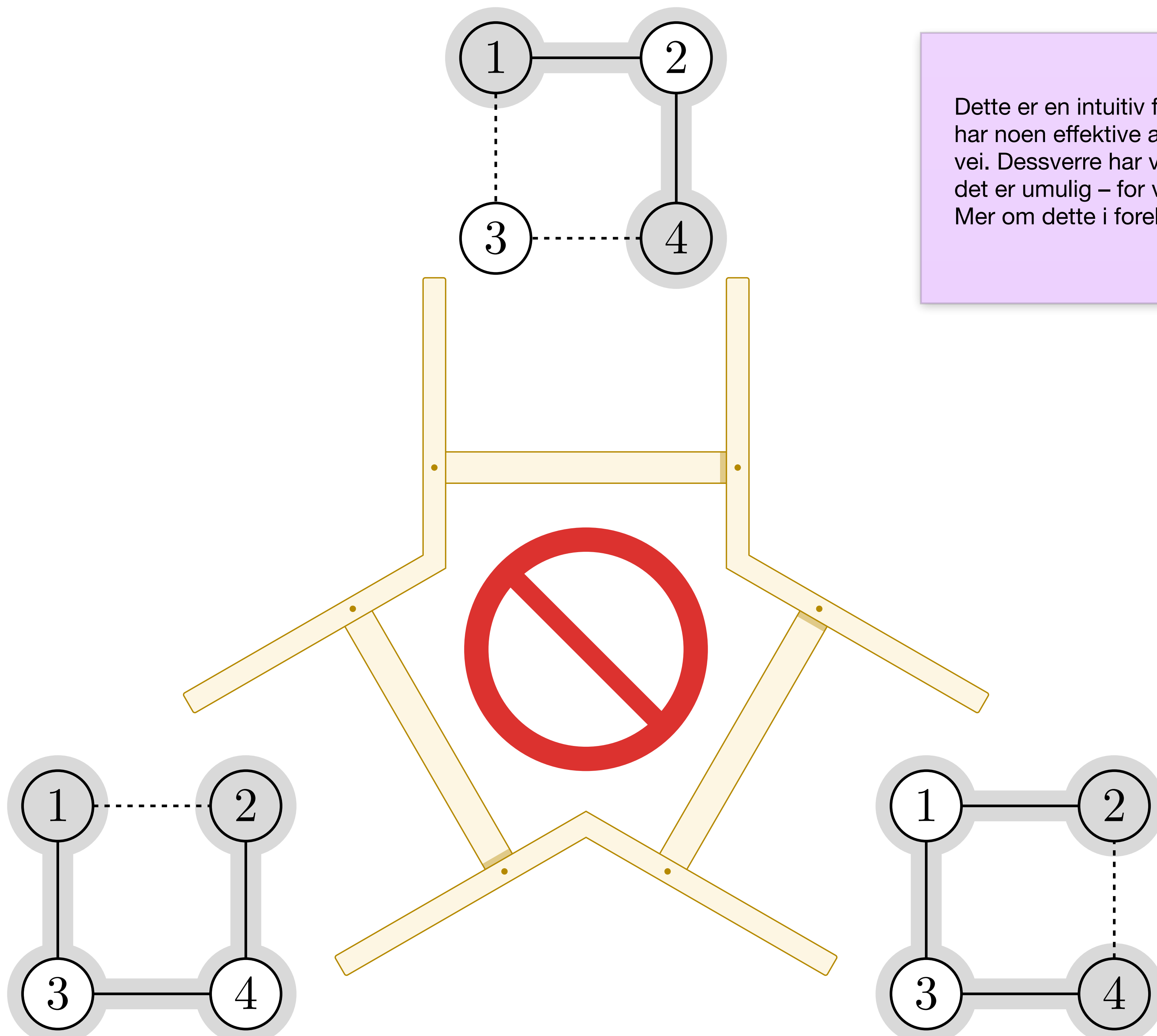
Om vi finner disse . . .



. . . så kan vi konstruere korteste vei $1 \leadsto 4$



Lengste vei: Prøver samme dekomponering



Dette er en intuitiv forklaring på hvorfor vi ikke har noen effektive algoritmer for å finne lengste vei. Dessverre har vi ingen forklaring på hvorfor det er umulig – for vi vet ikke om det *er* umulig! Mer om dette i forelesning 13!

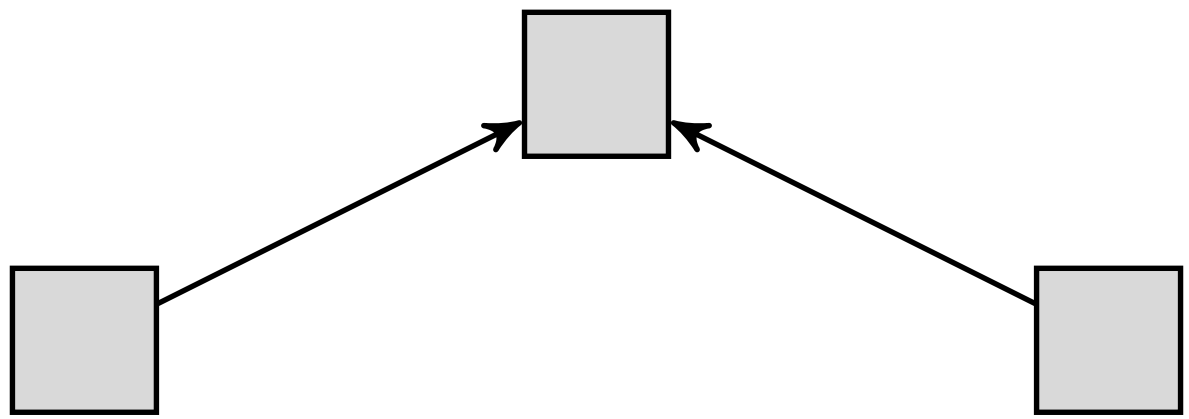
Fungerer ikke! En lengste vei kan ikke dekomponeres i lengste veier

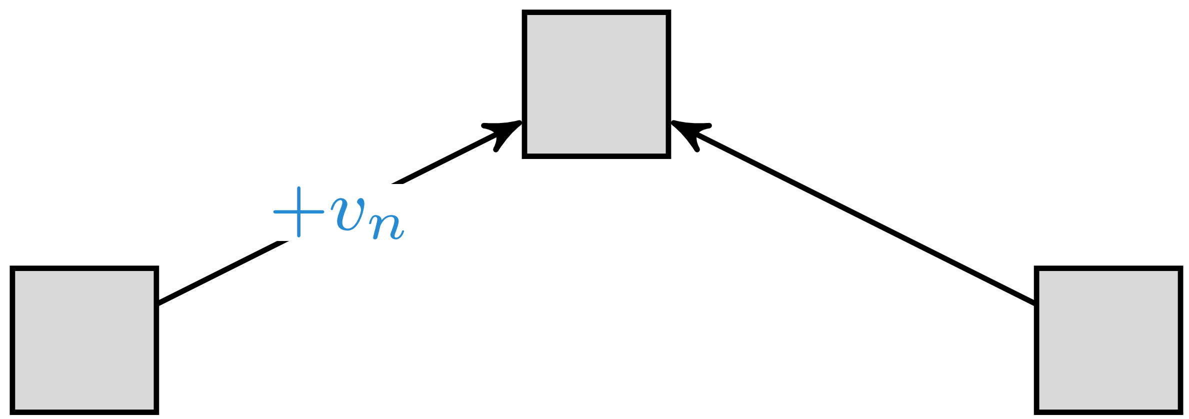
5:5

Eksempel: Ryggsekk

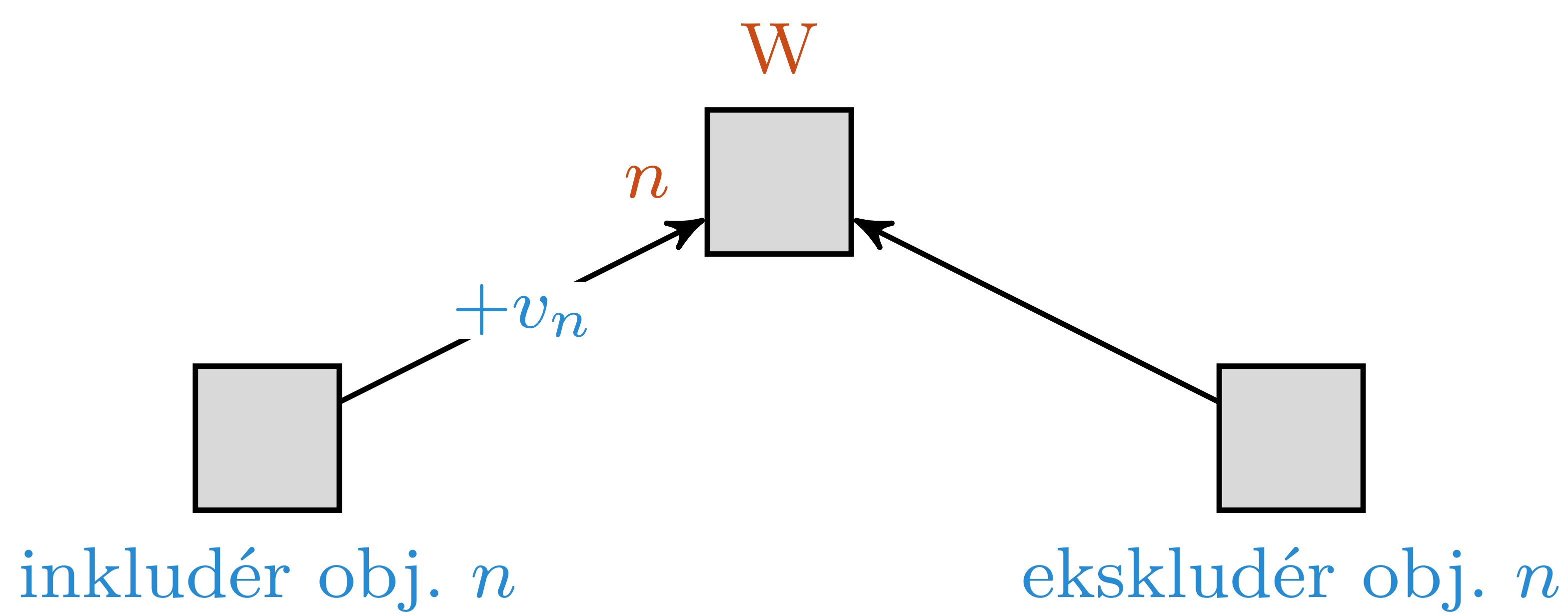
Input: Verdier $v_1, \ldots, v_n$, vekter $w_1, \ldots, w_n$ og en kapasitet W.

Output: Indekser $i_1, \ldots, i_k$ slik at $w_{i_1} + \cdots + w_{i_k} \leq$ W og totalverdien $v_{i_1} + \cdots + v_{i_k}$ er maksimal.

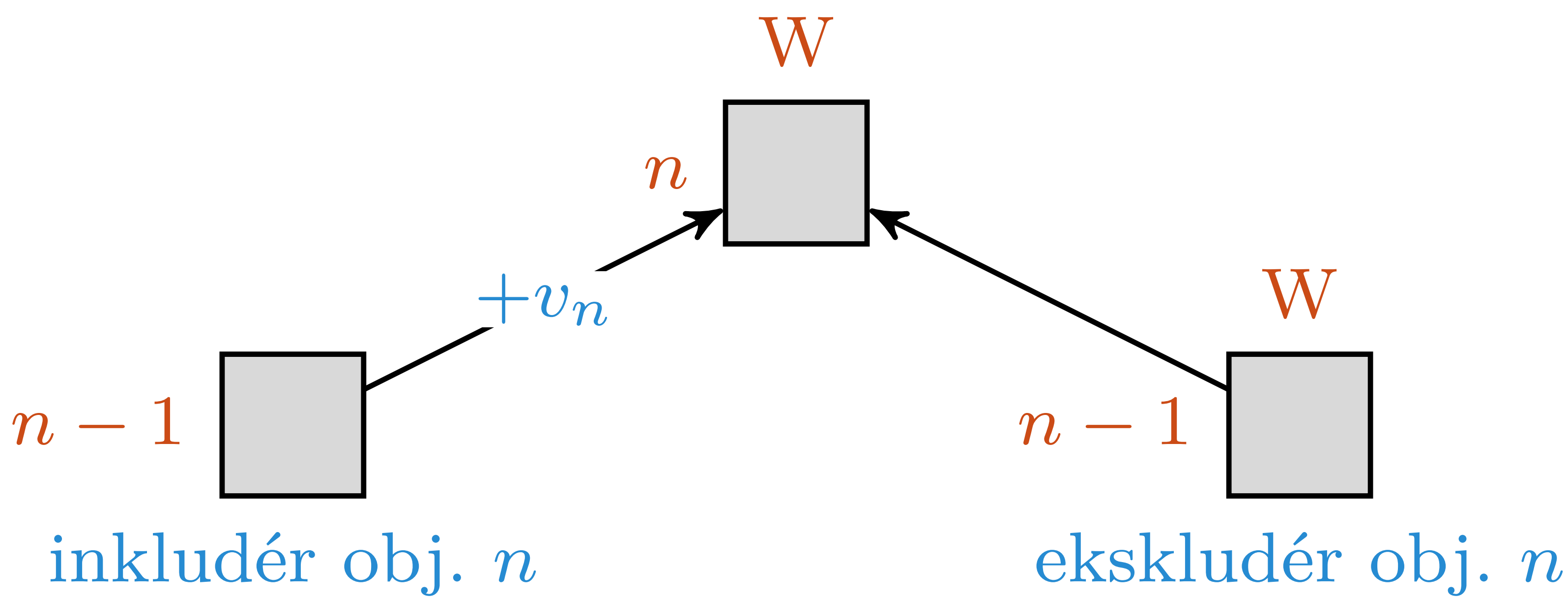




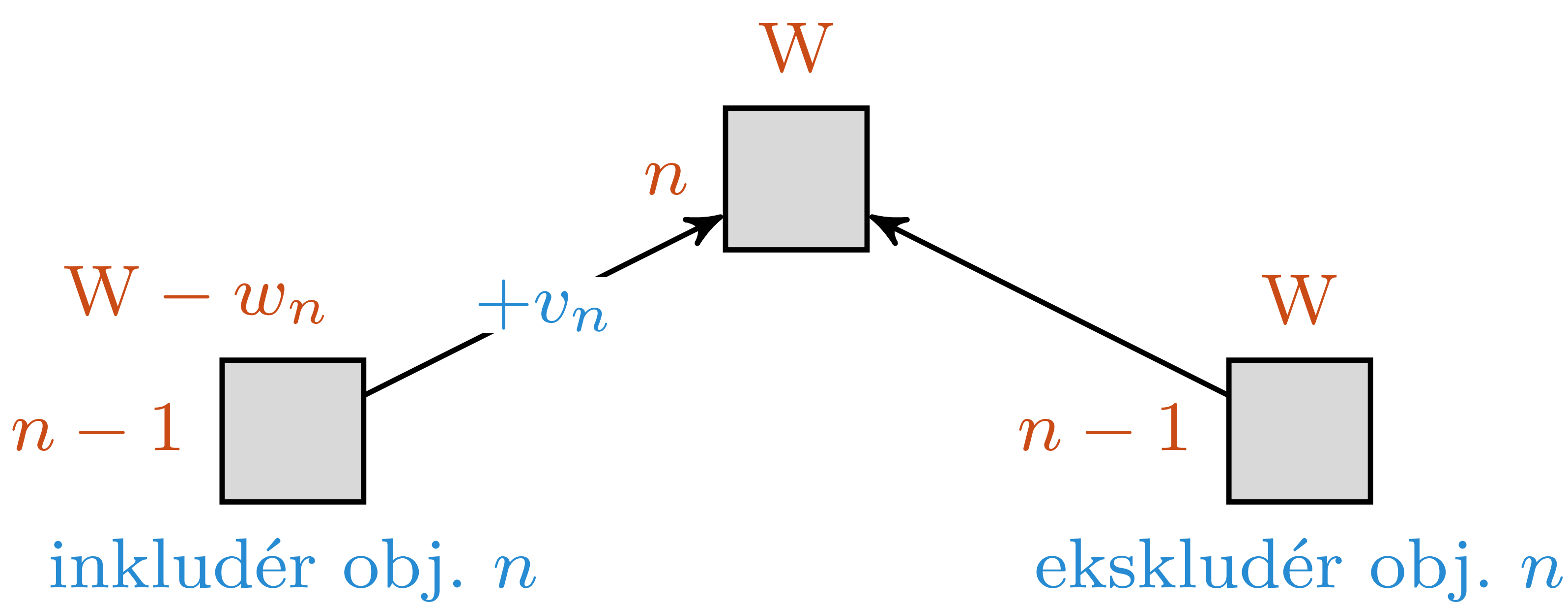
Objekt n bidrar med verdi v_n



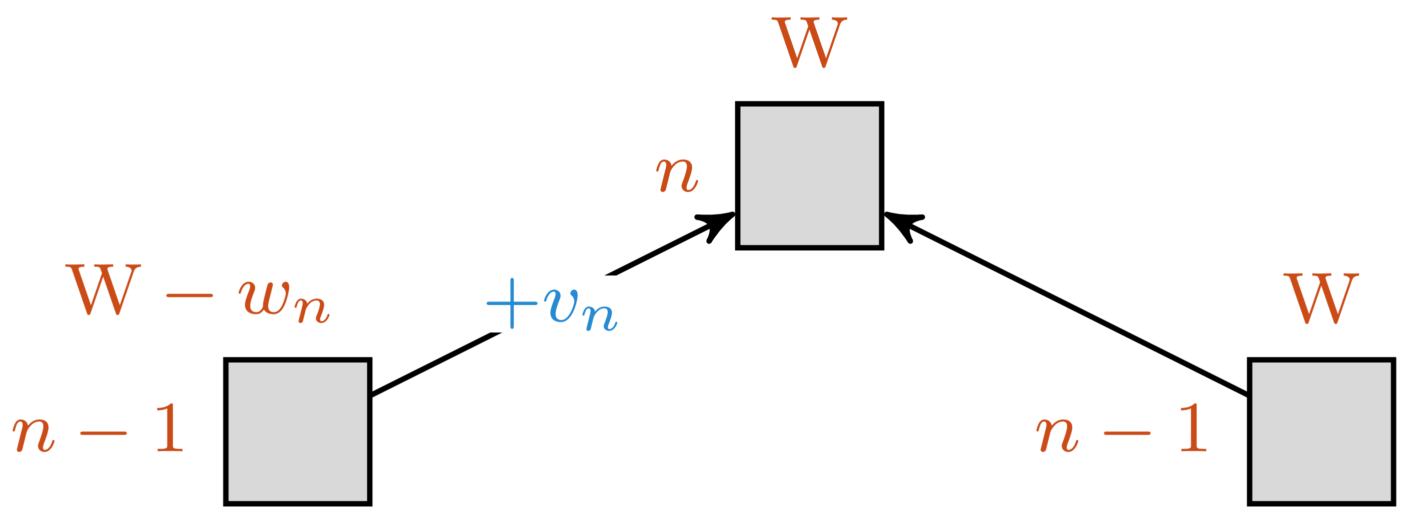
Vi parametriserer delproblemer vha. n og W



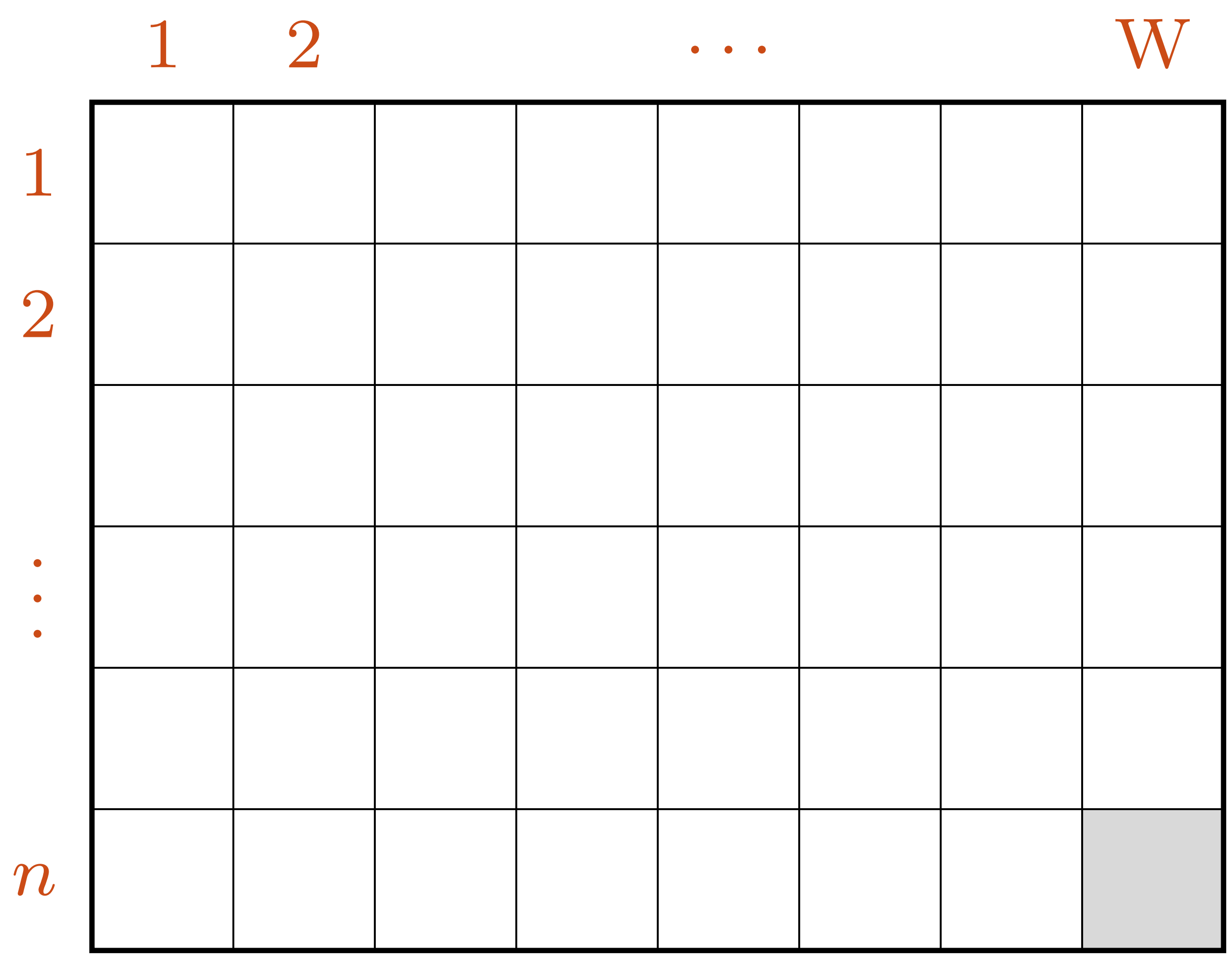
Ser nå bare på objekter $1 \ldots n-1$



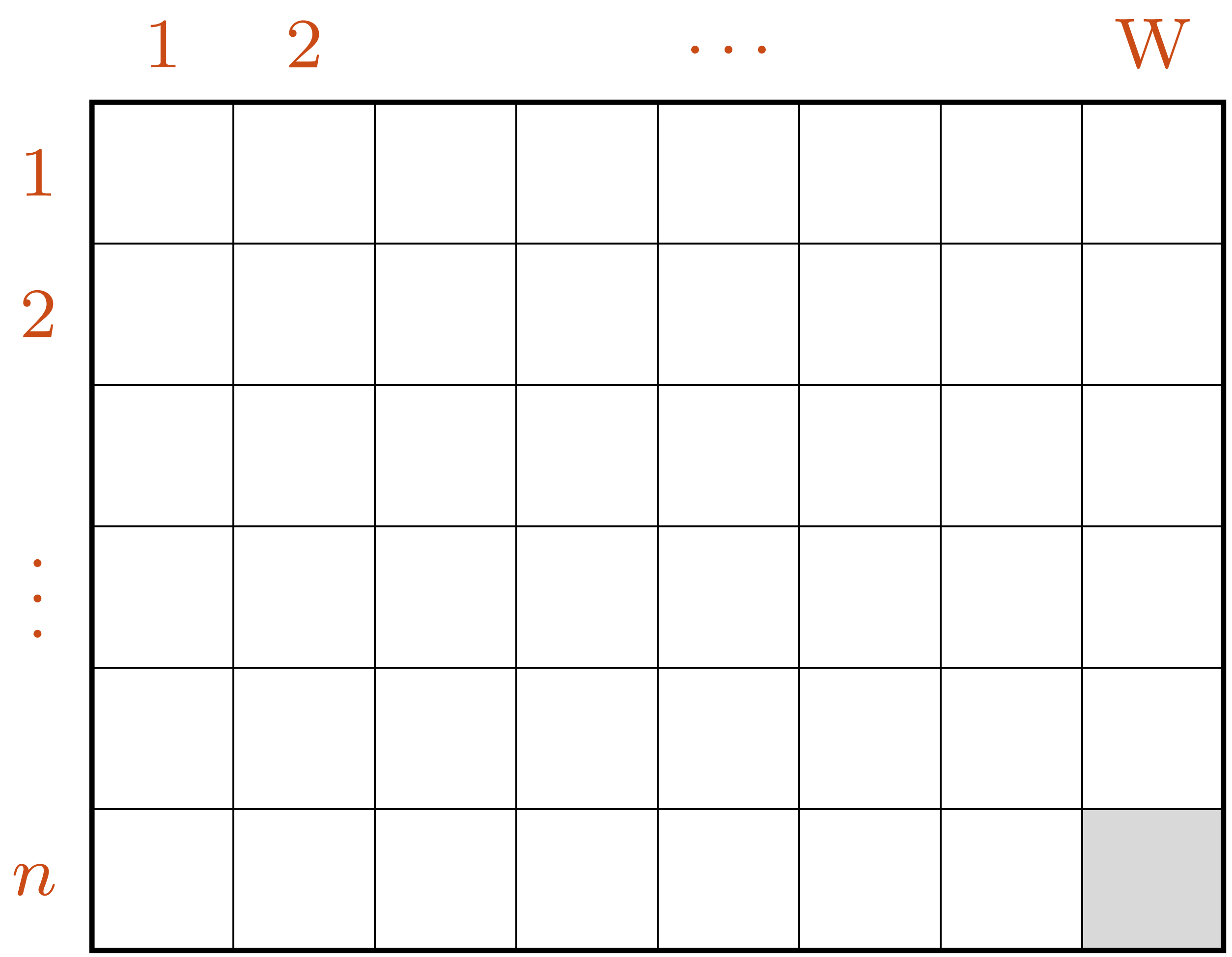
Objekt n bruker opp w_n av W



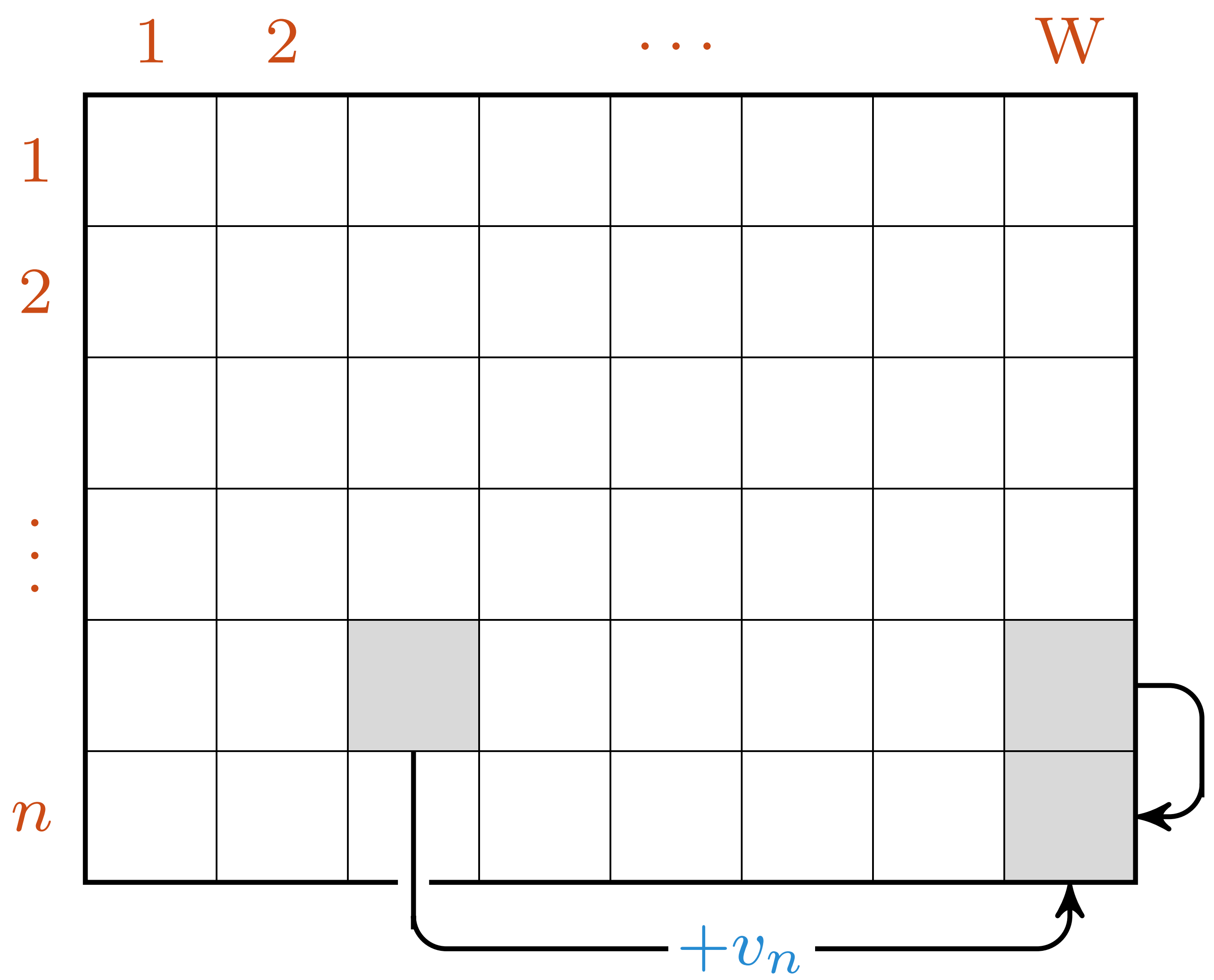
W
n
$W - w_n$
$+v_n$
W
$n-1$
$n-1$



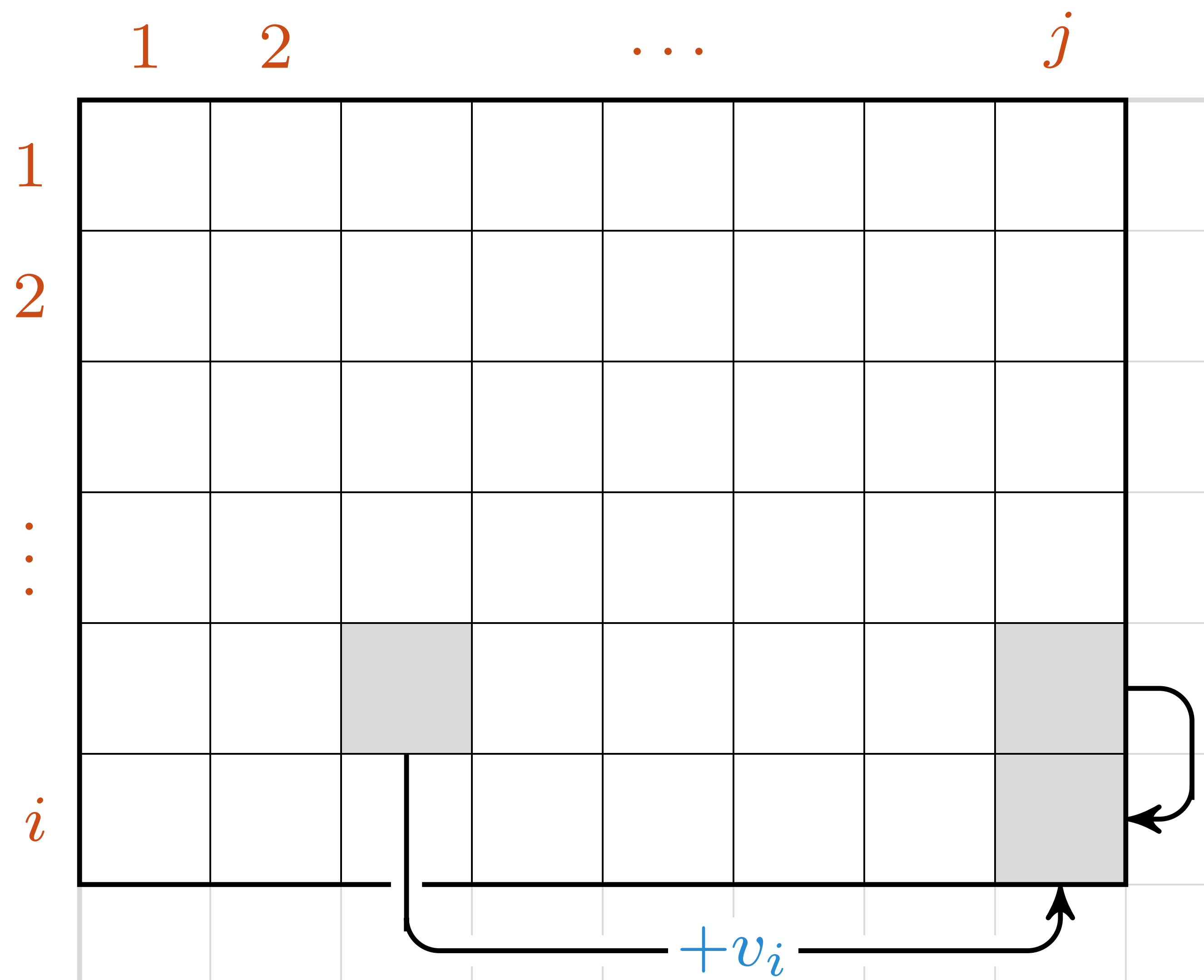
Lagre delløsninger i $n \times$ W-tabell



La f.eks. $w_n = 5$.



Dekomponering som før; kan løses radvis



Samme dekomponering for alle delproblemer

$\textsc{Knapsack}(n, \mathrm{W})$

n antall
W kapasitet

Utvalg: Maksimal total verdi, med vekt-kapasitet W

KNAPSACK(n, W)

let K[$0\,..\,n, 0\,..\,$W] be a new array

n	antall
W	kapasitet
K	memo

K[i, j] er optimum blant objekter 1 til i med kapasitet j

```
KNAPSACK(n, W)
let K[0..n, 0..W] be a new array
for j = 0 to W
```

n	antall
W	kapasitet
K	memo
j	kapasitet

For hver mulig kapasitet …

$\textsc{Knapsack}(n, \mathrm{W})$

```
let K[0..n, 0..W] be a new array
for j = 0 to W
    K[0, j] = 0
```

n	antall
W	kapasitet
K	memo
j	kapasitet

Ingen objekter, ingen verdi

```
KNAPSACK(n, W)
let K[0..n, 0..W] be a new array
for j = 0 to W
    K[0, j] = 0
for i = 1 to n
```

n	antall
W	kapasitet
K	memo
i	objekt
j	kapasitet

Vurdér objekt i: Velg eller forkast?

Knapsack(n, W)

let $\mathrm{K}[0 \,..\, n, 0 \,..\, \mathrm{W}]$ be a new array
for $j = 0$ **to** W
    $\mathrm{K}[0, j] = 0$
for $i = 1$ **to** n
    for $j = 0$ **to** W

n	antall
W	kapasitet
K	memo
i	objekt
j	kapasitet

Anta at kapasiteten er j

```
Knapsack(n, W)
let K[0 .. n, 0 .. W] be a new array
for j = 0 to W
    K[0, j] = 0
for i = 1 to n
    for j = 0 to W
        x = K[i − 1, j]
```

n antall
W kapasitet
K memo
i objekt
j kapasitet

x uten i

Det beste vi fikk til med objekter 1 til $i - 1$

KNAPSACK(n, W)

let $\mathrm{K}[0\,..\,n, 0\,..\,\mathrm{W}]$ be a new array
for $j = 0$ **to** W
  $\mathrm{K}[0, j] = 0$
for $i = 1$ **to** n
  for $j = 0$ **to** W
    $x = \mathrm{K}[i-1, j]$
    if $j < w_i$

n	antall
W	kapasitet
K	memo
i	objekt
j	kapasitet
w_i	vekt
x	uten i

Er objekt i tyngre enn hele kapasiteten?

```
KNAPSACK(n, W)
let K[0..n, 0..W] be a new array
for j = 0 to W
    K[0, j] = 0
for i = 1 to n
    for j = 0 to W
        x = K[i − 1, j]
        if j < w_i
            K[i, j] = x
```

n antall
W kapasitet
K memo
i objekt
j kapasitet
w_i vekt

x uten i

Dropp objekt i, og behold forrige løsning

```
Knapsack(n, W)
let K[0..n, 0..W] be a new array
for j = 0 to W
    K[0, j] = 0
for i = 1 to n
    for j = 0 to W
        x = K[i - 1, j]
        if j < w_i
            K[i, j] = x
        else y = K[i - 1, j - w_i] + v_i
```

n	antall
W	kapasitet
K	memo
i	objekt
j	kapasitet
w_i	vekt
v_i	verdi
x	uten i
y	med i

Ellers: Ta med i, og løs resten «rekursivt»

```
KNAPSACK(n, W)
let K[0..n, 0..W] be a new array
for j = 0 to W
    K[0, j] = 0
for i = 1 to n
    for j = 0 to W
        x = K[i − 1, j]
        if j < w_i
            K[i, j] = x
        else y = K[i − 1, j − w_i] + v_i
            K[i, j] = max(x, y)
```

n	antall
W	kapasitet
K	memo
i	objekt
j	kapasitet
w_i	vekt
v_i	verdi
x	uten i
y	med i

Velg det beste av å ta med i eller ikke

Svaret er altså i siste rute, K[6,5], dvs., 15.

```
Knapsack(n, W)
let K[0..n, 0..W] be a new array
for j = 0 to W
    K[0, j] = 0
for i = 1 to n
    for j = 0 to W
        x = K[i − 1, j]
        if j < w_i
            K[i, j] = x
        else y = K[i − 1, j − w_i] + v_i
            K[i, j] = max(x, y)
```

$i, j, x, y \;=\; -, -, -, -$

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	1	1	1	1
2	0	1	5	6	6	6
3	0	4	5	9	10	10
4	0	4	5	9	10	10
5	0	4	6	9	11	12
6	0	4	6	10	12	15

w	v
1	1
2	5
1	4
3	3
1	2
2	6

Vi kan spore oss tilbake til hvilke elementer som er med på samme måte som i LCS, hvis vi tar vare på valget som gjøres av max(x,y) i hver iterasjon.

Hovedidé: Rekursiv dekomponering, akkurat som før, men noen rekursive kall går igjen, så vi lagrer svarene og slår dem opp når vi trenger dem.

1. Eksempel: Stavkapping
2. Dyn. prog. › hva er det?
3. Eksempel: LCS
4. Optimal delstruktur
5. Eksempel: Ryggsekk