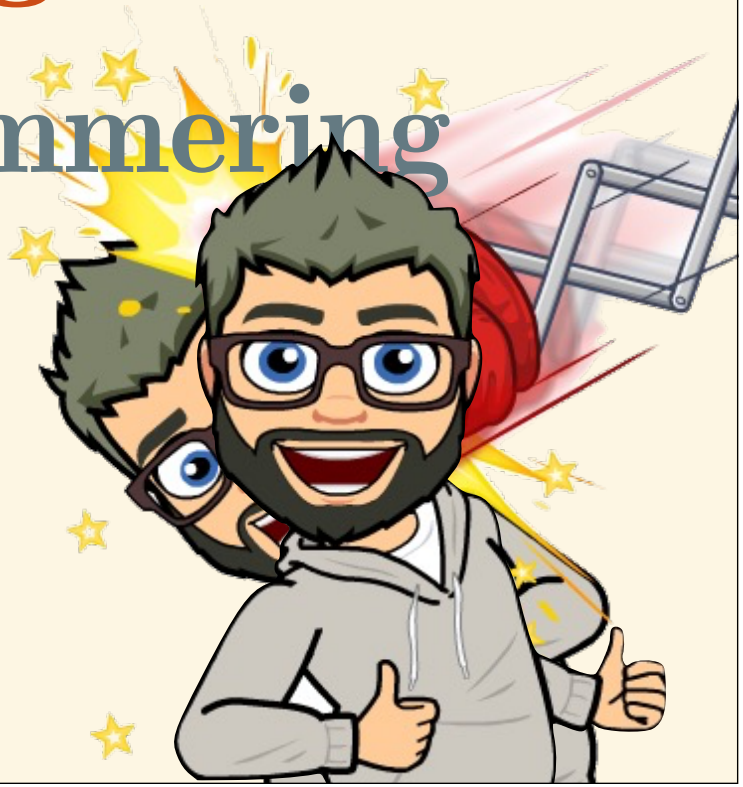
Forelesning 6

Dynamisk programmering



Rep., søketrær

- Etterfølger
- Innsetting
- Sletting
- Balanse

Dyn. Prog.

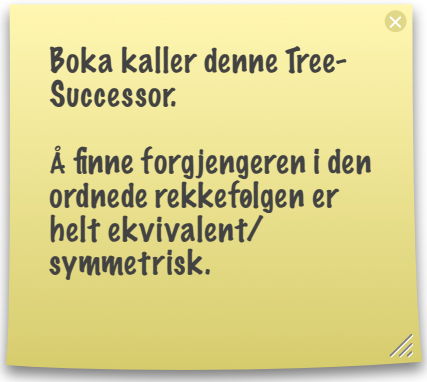
- Hva er det?
- Substruktur

Eksempler

- Stavkutting
- LCS
- Ryggsekk
- Matrisekjede

Søketrær

Søketrær › Etterfølger



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

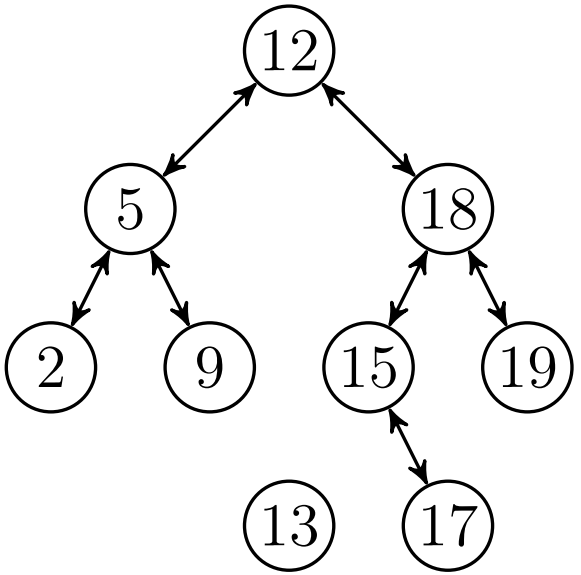
Her bruker har vi en peker «på slep» (trailing pointer), så den peker på den forrige noden hele tiden. Så når vi kommer til en NIL-peker, så trenger vi ikke en foreldrepeker (som jo ikke eksisterer).

Søketrær › Innsetting

Boka kaller denne Tree-Insert.

```
INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```

```
INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```



Her har jeg tegnet piler begge veier på kantene, for å indikere at vi også har foreldrepekere. (Node v har foreldrepeker v.p.)

Søketrær › Sletting

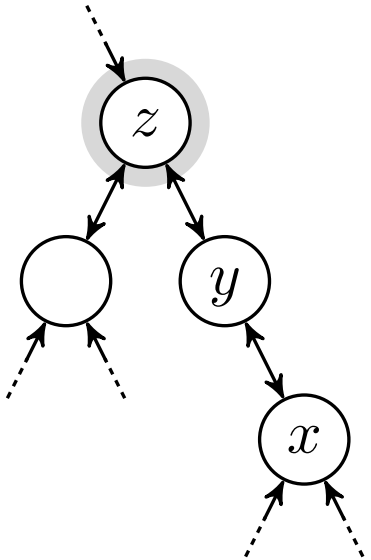
- **Ingen barn?**
  - **Fjern noden**
- **Ett barn?**
  - **Barnet rykker opp**
- **Er etterfølger barn?**
  - **Spleis inn etterfølger**
- **Ellers...**
  - **Etterfølgers barn rykker opp**
  - **Spleis inn etterfølger**

```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2         T.root = v
3   elseif u == u.p.left
4         u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7         v.p = u.p
```

```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```

Har forkortet "Transplant" til "Transp" her, av plasshensyn.

```
DELETE(T, z)
 1  if z.left == NIL
 2        TRANSP(T, z, z.right)
 3  elseif z.right == NIL
 4        TRANSP(T, z, z.left)
 5  else y = MINIMUM(z.right)
 6        if y.p ≠ z
 7              TRANSP(T, y, y.right)
 8              y.right = z.right
 9              y.right.p = y
10        TRANSP(T, z, y)
11        y.left = z.left
12        y.left.p = y
```



Siden y er etterkommer (successor), så vil den alltid ligge i høyre deltre. Dersom den er barn av y vil den altså være høyre barn.

Algoritme	Kjøretid
INORDER-TREE-WALK	$\Theta(n)$
TREE-SEARCH	$\mathrm{O}(h)$
TREE-MINIMUM	$\mathrm{O}(h)$
TREE-SUCCESSOR	$\mathrm{O}(h)$
TREE-INSERT	$\mathrm{O}(h)$
TREE-DELETE	$\mathrm{O}(h)$

Søketrær › **Balanse**

- Tilfeldig input-permutasjon gir logaritmisk forventet høyde
- Worst-case-høyde er lineær!
- Det er mulig å holde treet balansert etter hver innsetting og sletting – i logaritmisk tid
  - Detaljer ikke pensum
- Hvorfor er det umulig, i verste tilfelle, å bygge et binært søketre like raskt som en haug?

Hvis vi klarte det, så hadde vi brutt grensen for sorteringshastighet!

Dynamisk Programmering

DP › Hva er det?

Oppskrift, Cormen et al.

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

Dynamic Programming, Foundations and Principles (2. utg., CRC Press, 2011)

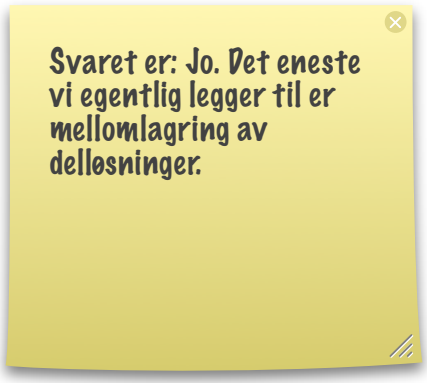
Oppskrift, Sniedovich

1. **Embed** your problem in a family of related problems
2. **Derive** a relationship between the solutions to these problems
3. **Solve** this relationship
4. **Recover** a solution to your problem from this relationship

"**Any** problem admitting of the kind of treatment outlined by this **meta-recipe** is a **dynamic programming problem.**"

— Moshe Sniedovich

Men er det ikke det vi har gjort til nå?

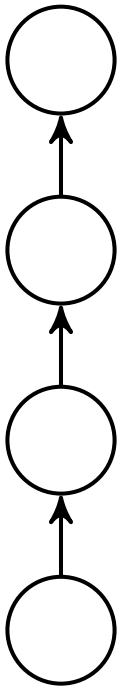


Vi lagrer all mellomregning/alle delløsninger, så de kan brukes om igjen, som i et regneark. (Vi kan naturligvis ikke ha sykliske avhengigheter.)

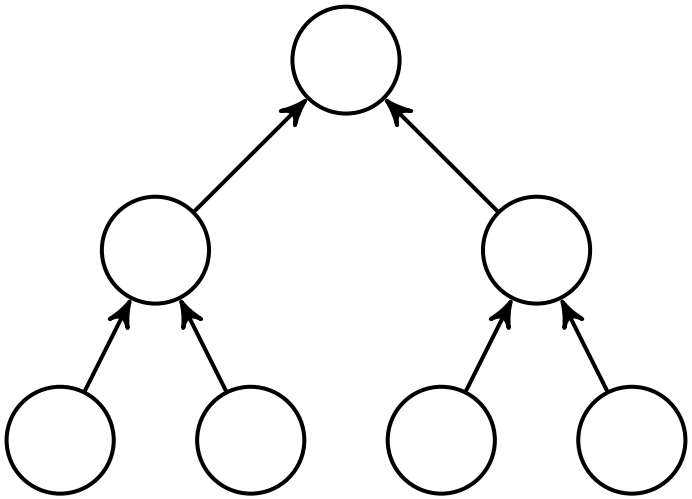
«Time–memory tradeoff»

Tenk regneark

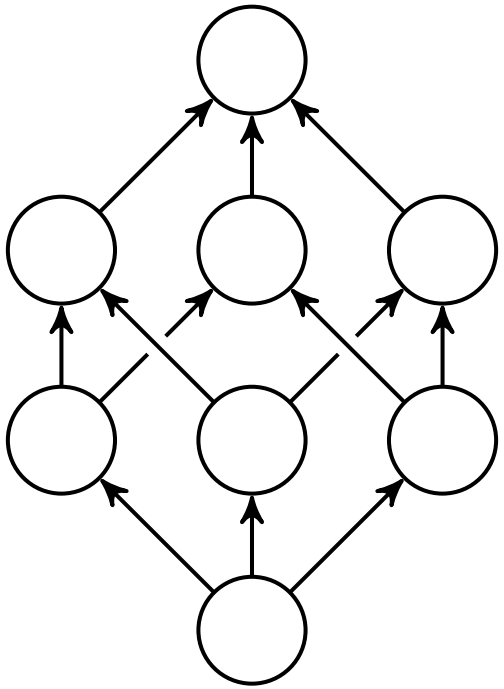
Bare nyttig hvis vi trenger noen av løsningene mer enn én gang, dog...



Inkrementell design

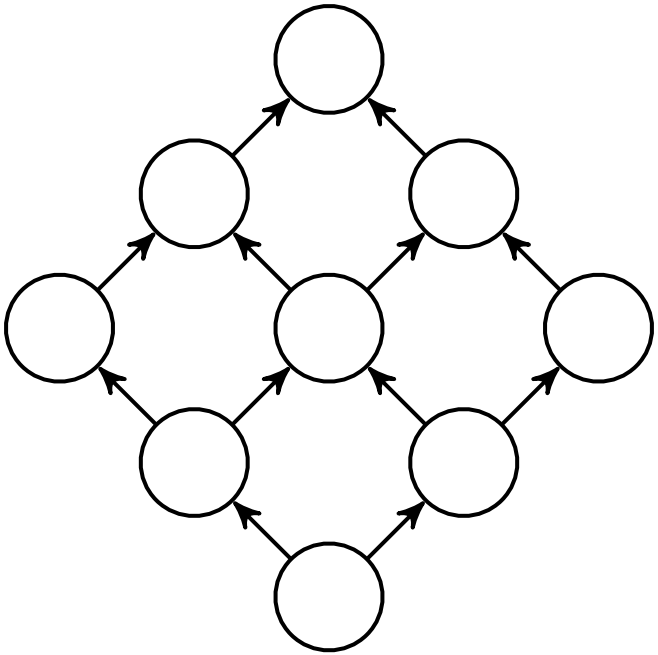


Uavhengige delproblemer: Splitt og hersk

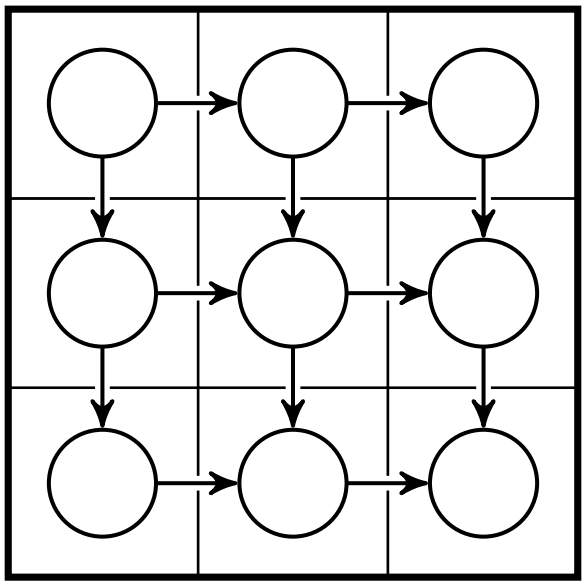


Induksjonen vi bruker her er generelt velfundert induksjon – men ofte kan vi bruke naturlig induksjon basert på problemstørrelse.

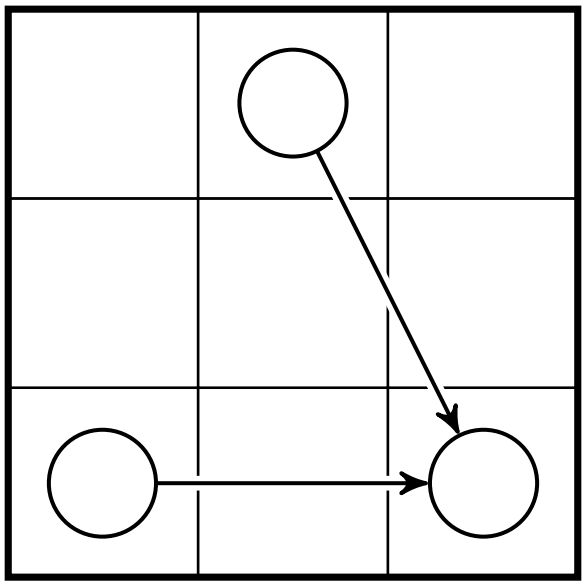
Overlappende delproblemer: Dynamisk programmering



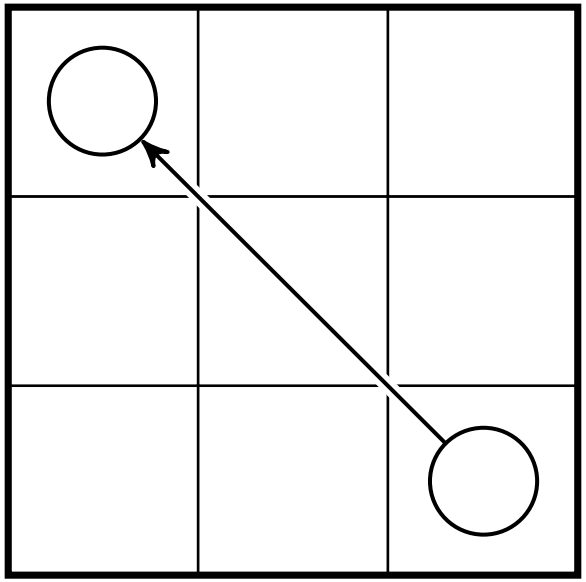
Foreløpig forenkling: Delproblemer i «rutenett»



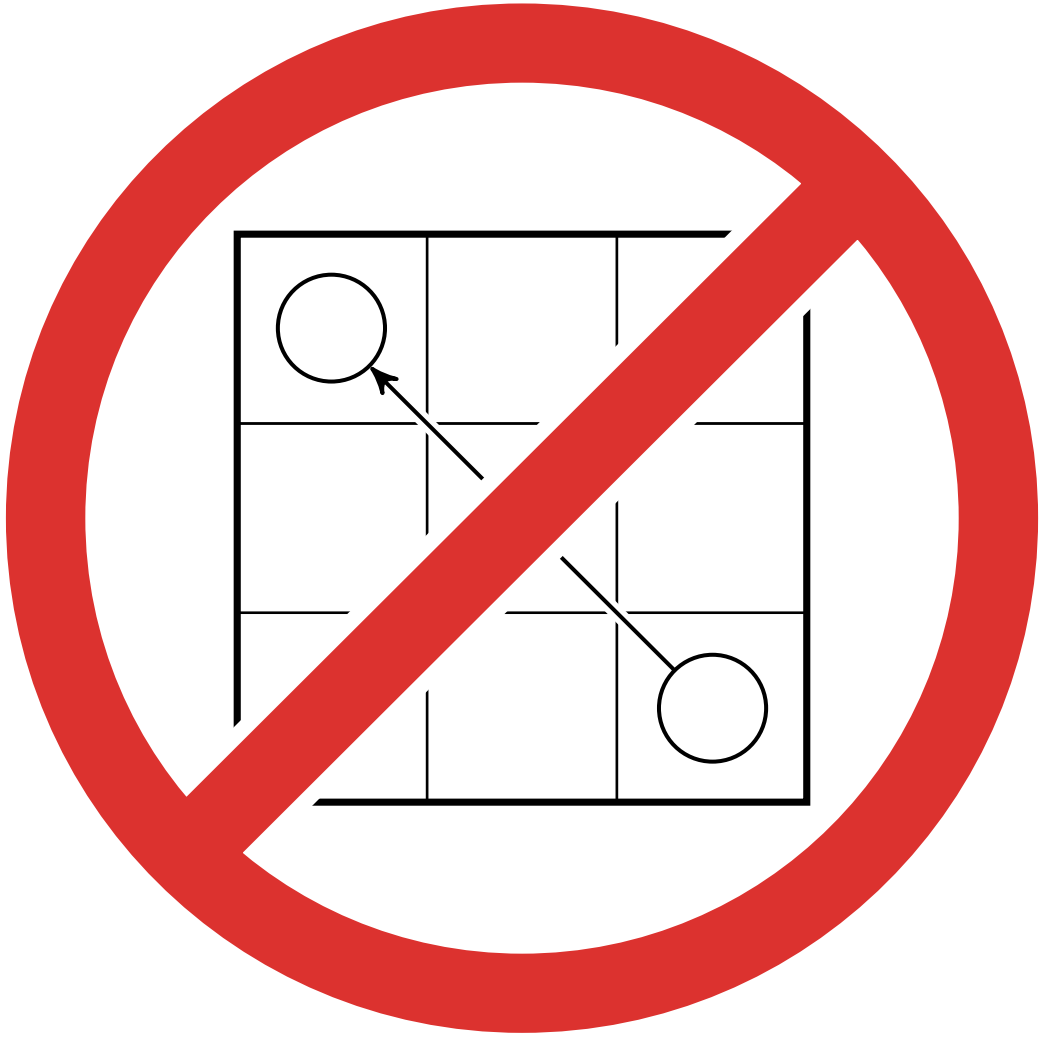
Lar oss lagre løsninger i en tabell



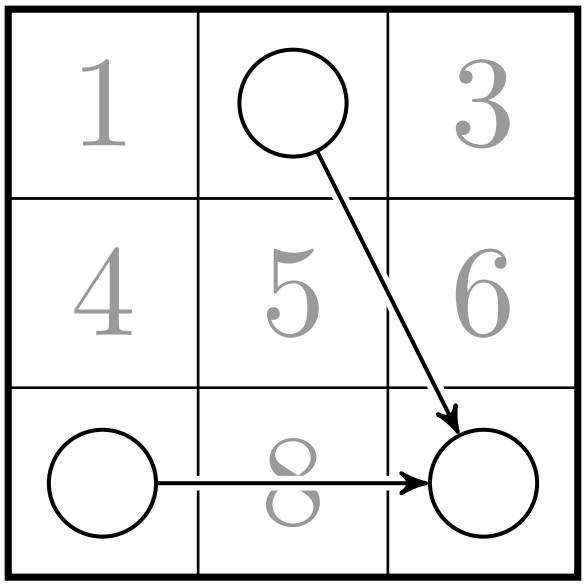
Kan ha mer rotete avhengigheter . . .



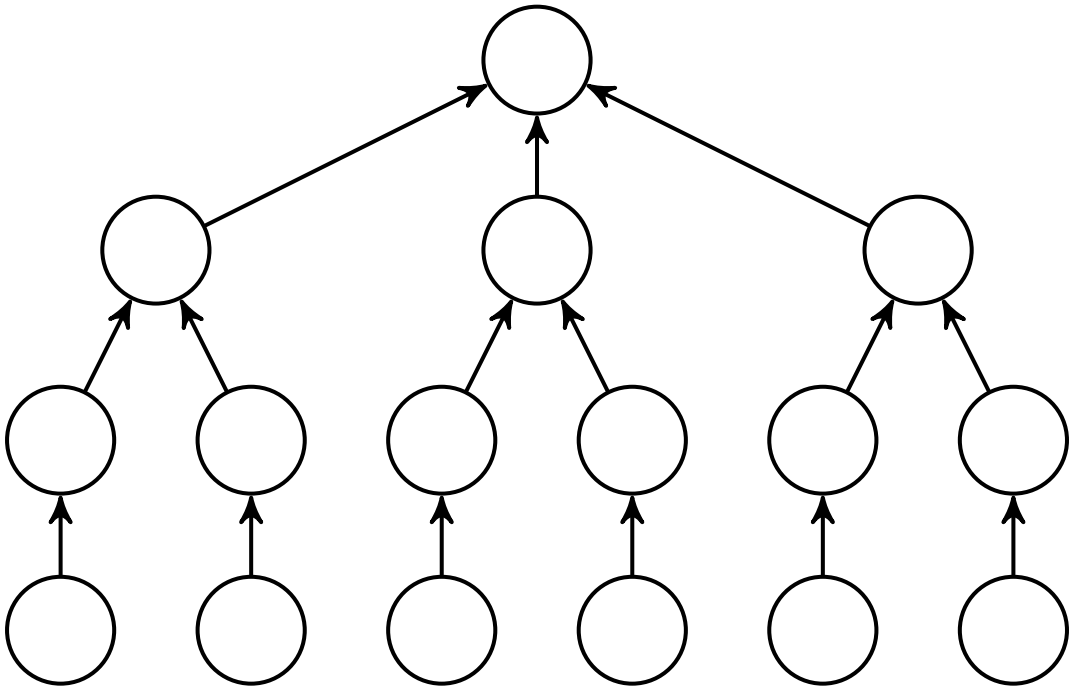
. . . men de kan ikke gå opp eller til venstre . . .



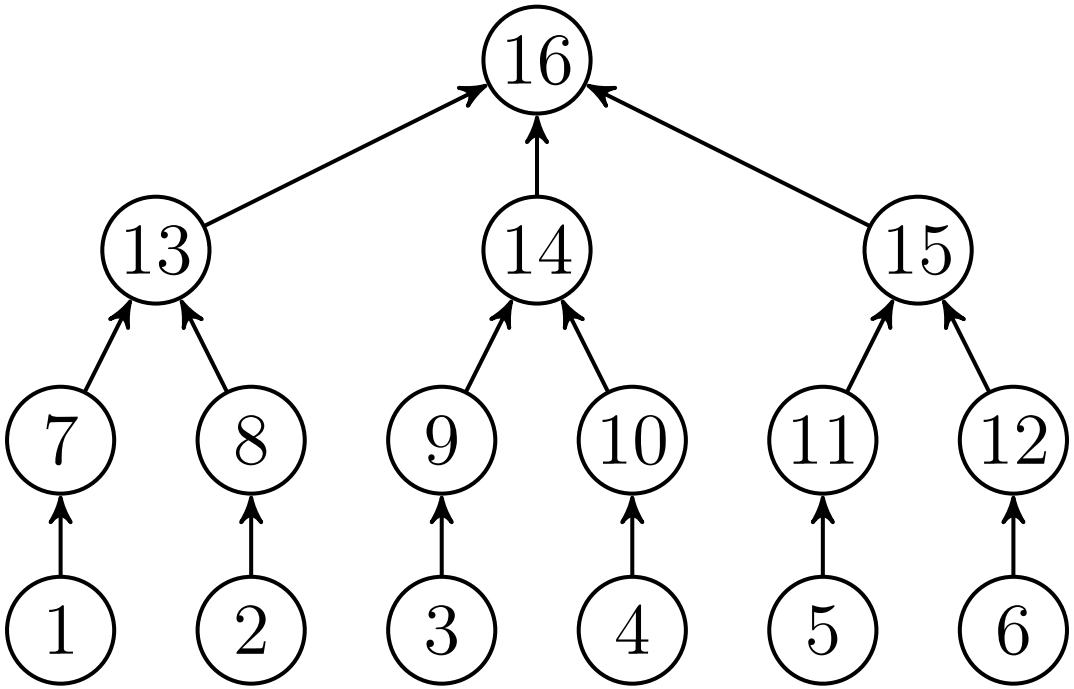
. . . men de kan ikke gå opp eller til venstre . . .



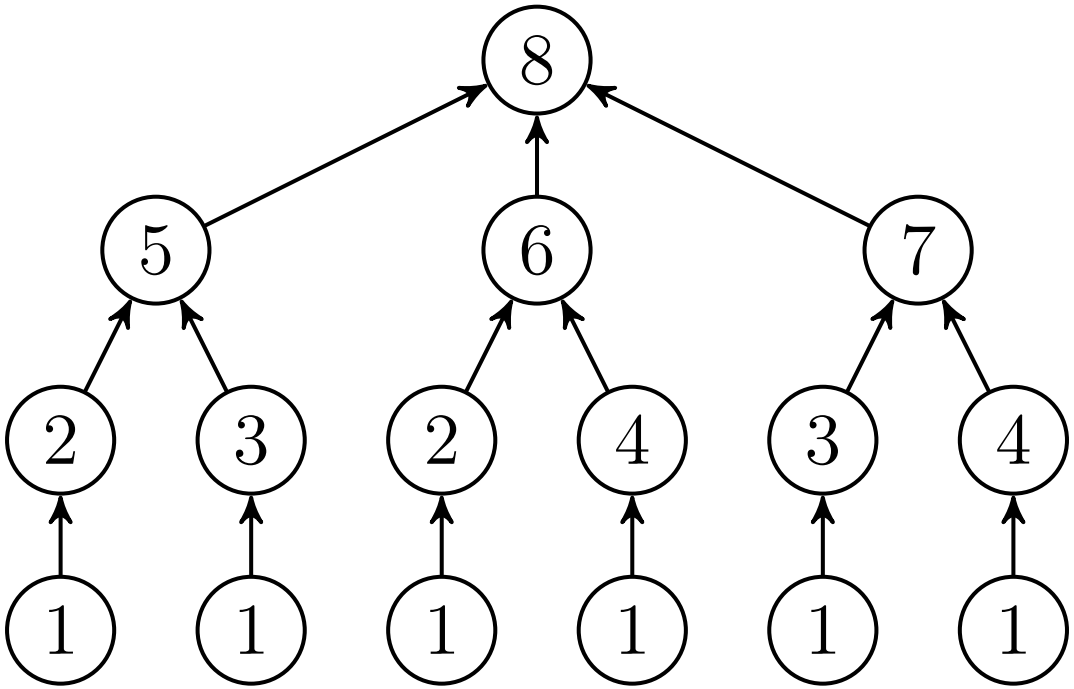
. . . fordi vi vil jobbe radvis (eller kolonnevis)



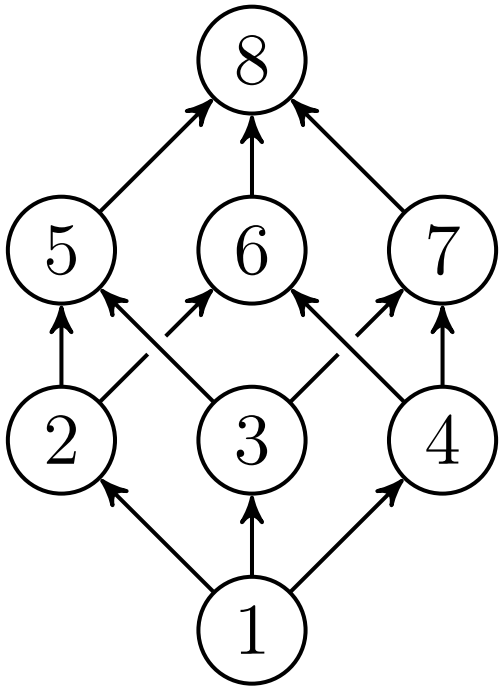
Uavhengige delproblemer: Splitt og hersk



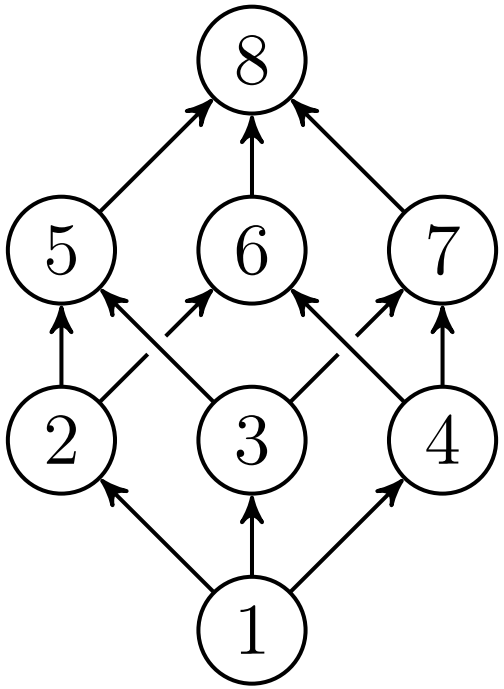
Uavhengige delproblemer: Splitt og hersk



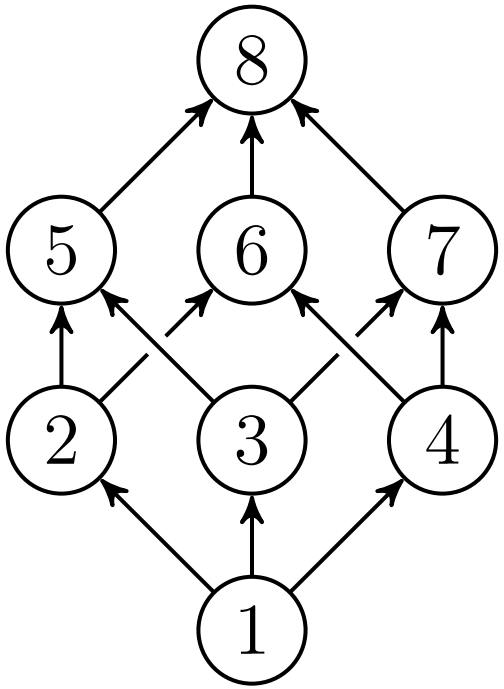
Overlappende og delte delproblemer . . .



Overlappende og delte delproblemer ...



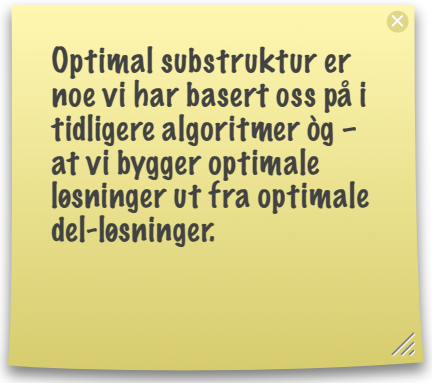
Grafen har like mange stier som treet



Idé: Lagre hvert delsvar!

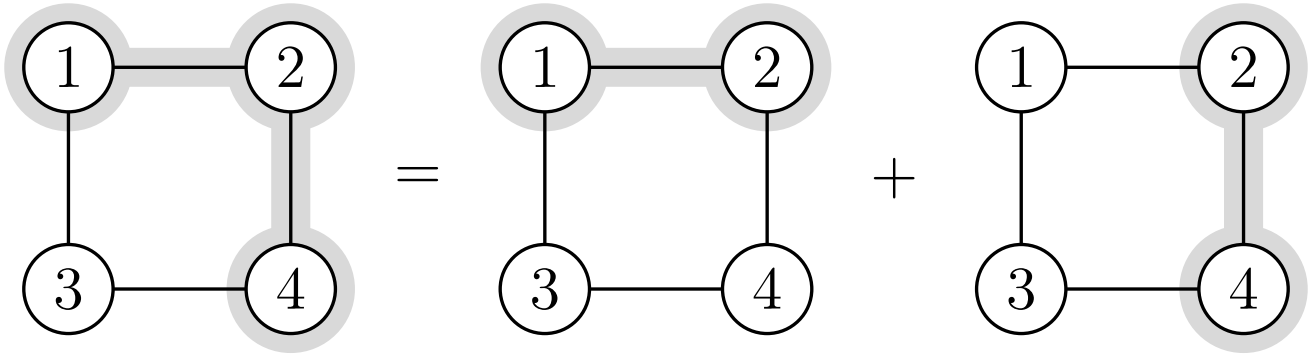
Nyttig når vi har overlappende delproblemer

Korrekt når vi har optimal substruktur

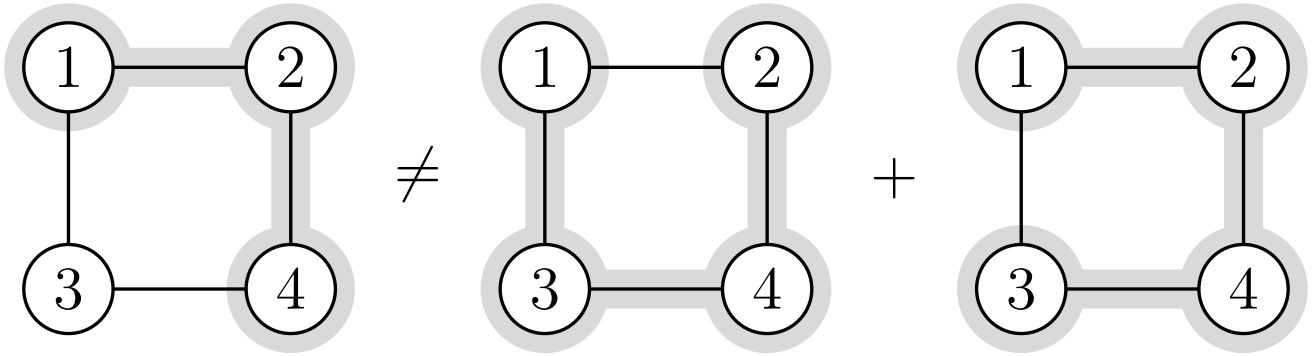


DP ›

Optimal substruktur



Korteste vei består av korteste veier



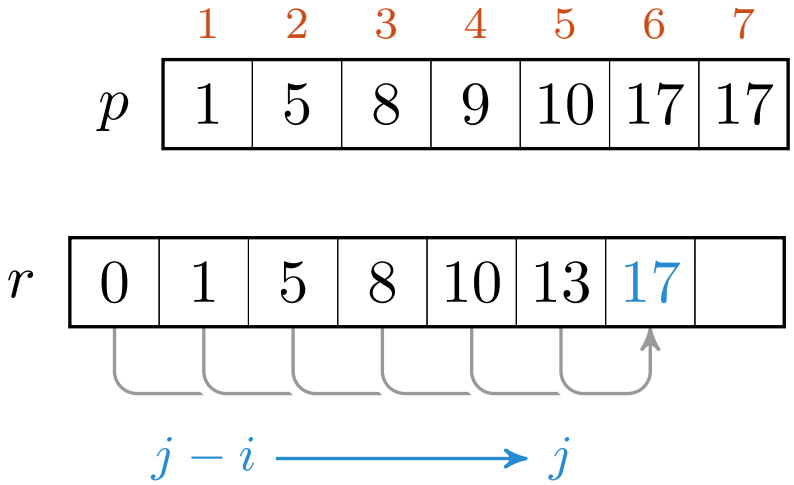
Lengste vei består ikke av lengste veier

Eksempler!

DP › Stavkutting

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.



$p[1] + r[5] = 14$

$p[2] + r[4] = 15$

$p[3] + r[3] = 16$

$p[4] + r[2] = 14$

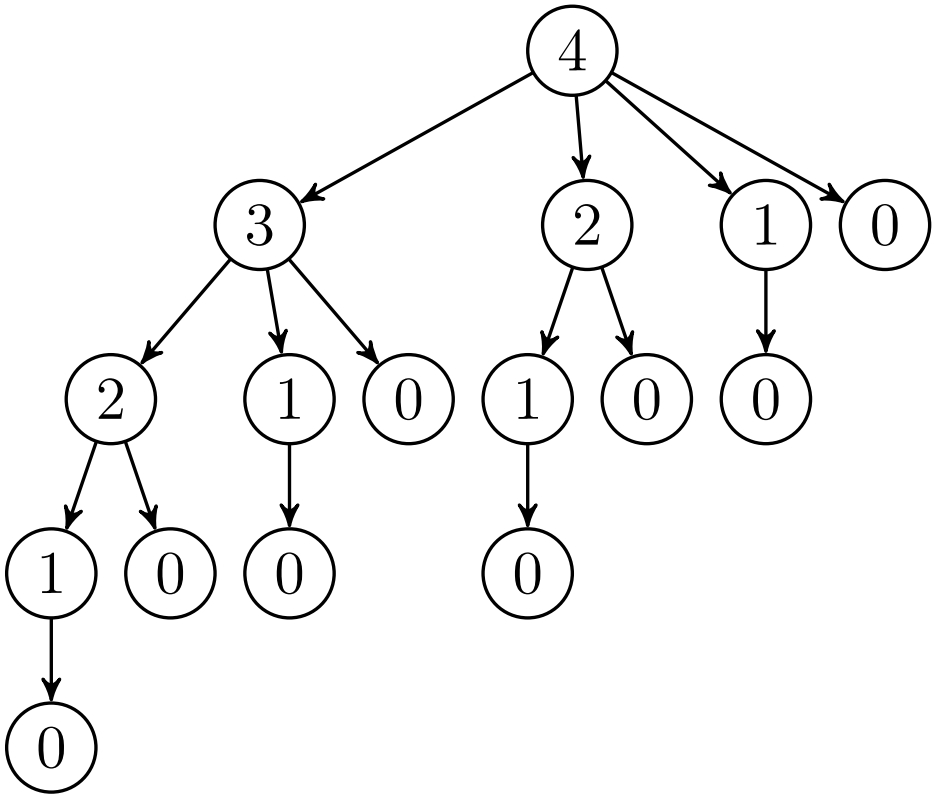
$p[5] + r[1] = 11$

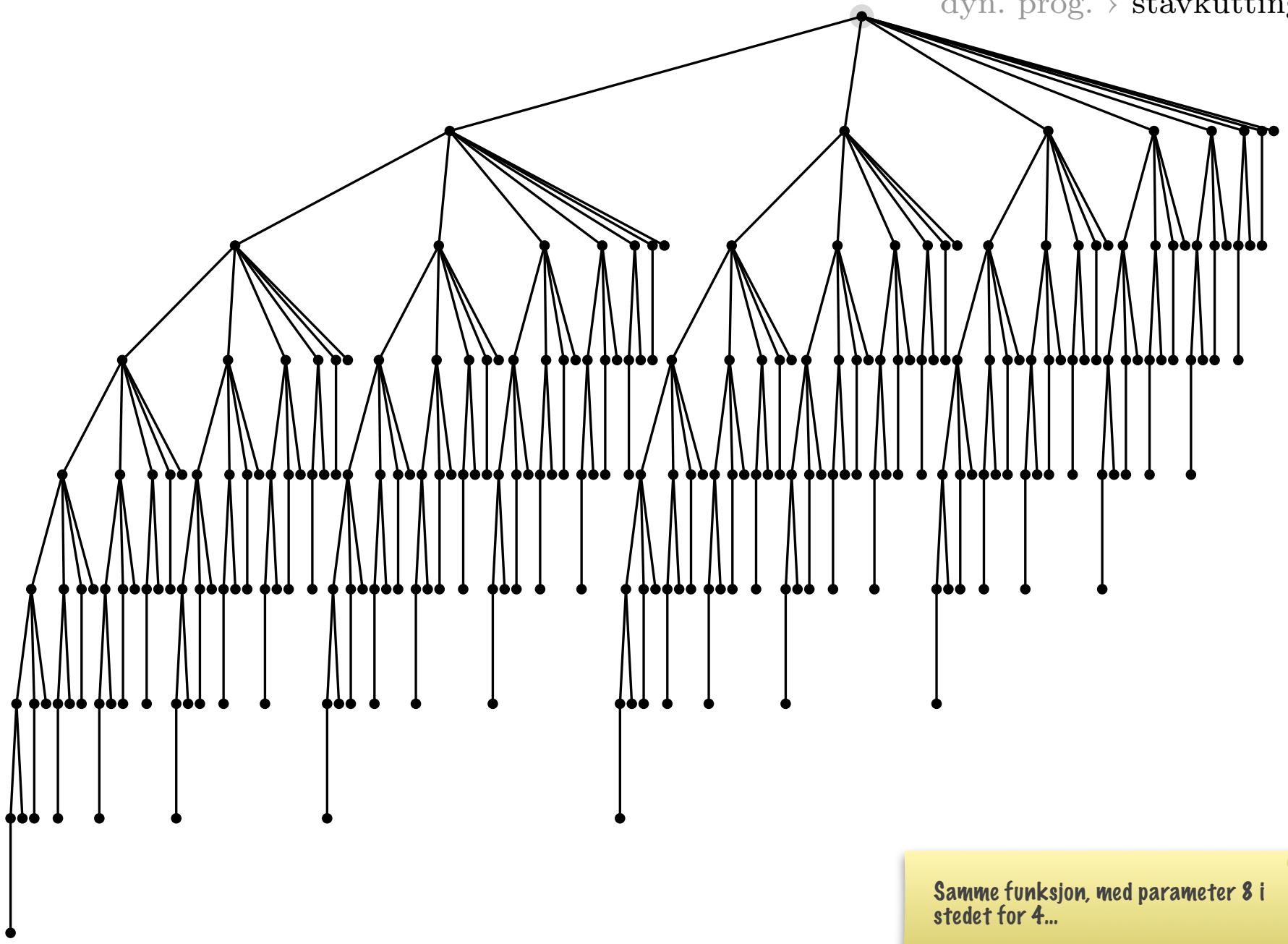
$p[6] + r[0] = 17$

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

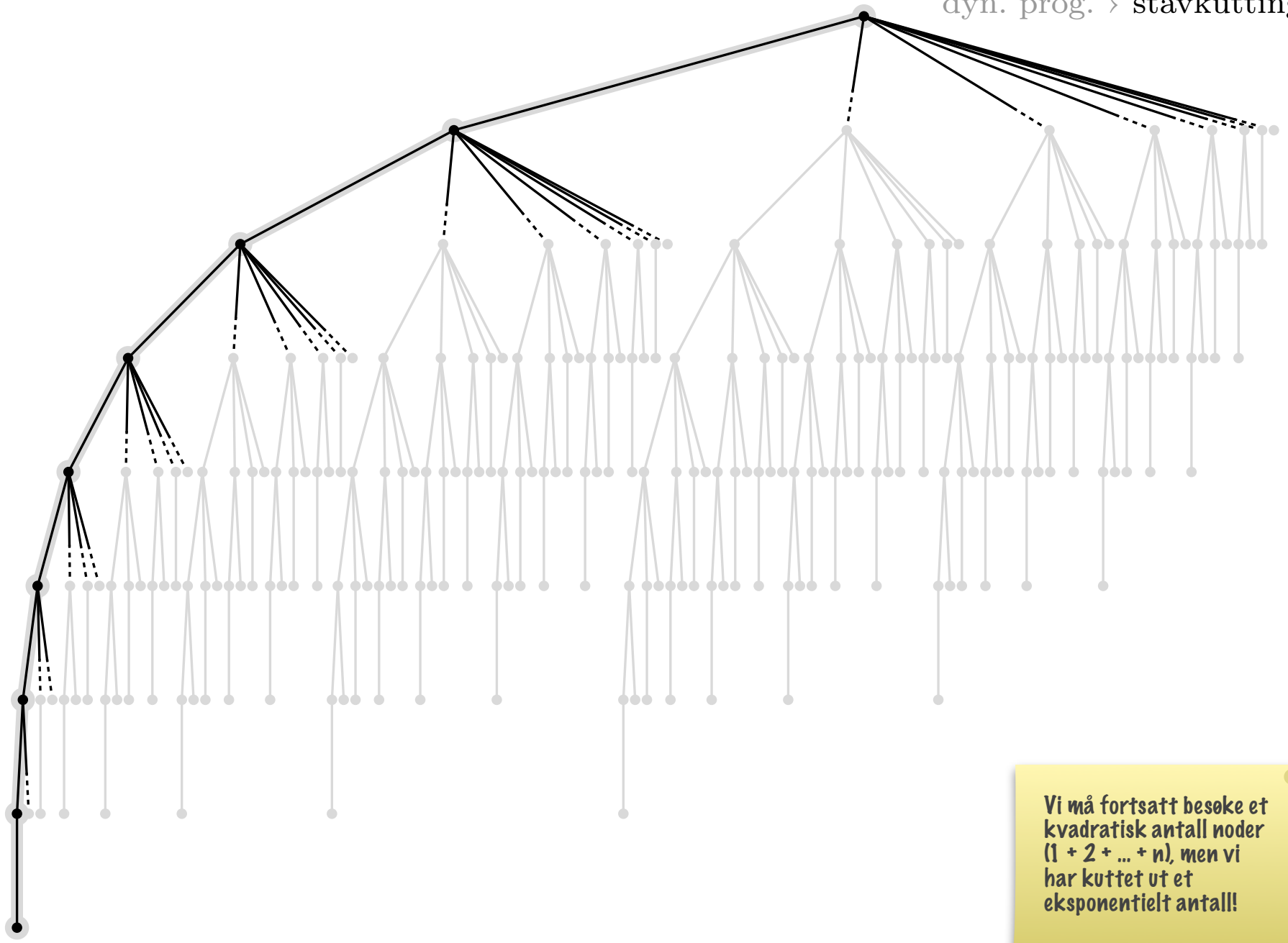
```
CUT(p, n)
1  if n == 0
2        return 0
3  q = −∞
4  for i = 1 to n
5        t = p[i] + CUT(p, n − i)
6        q = max(q, t)
7  return q
```

→ 10





Samme funksjon, med parameter 8 i stedet for 4...

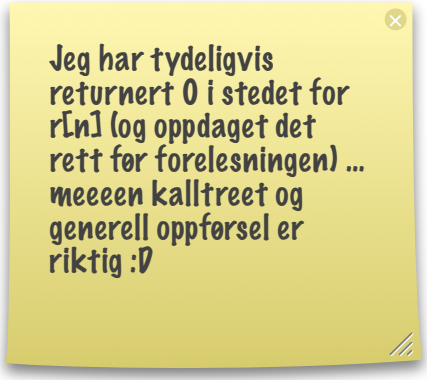


Vi må fortsatt besøke et kvadratisk antall noder (1 + 2 + ... + n), men vi har kuttet ut et eksponentielt antall!

```
AUX(p, n, r)
 1  if r[n] ≥ 0
 2        return r[n]
 3  if n == 0
 4        q = 0
 5  else q = −∞
 6        for i = 1 to n
 7              t = p[i] + AUX(p, n − i, r)
 8              q = max(q, t)
 9  r[n] = q
10  return q
```

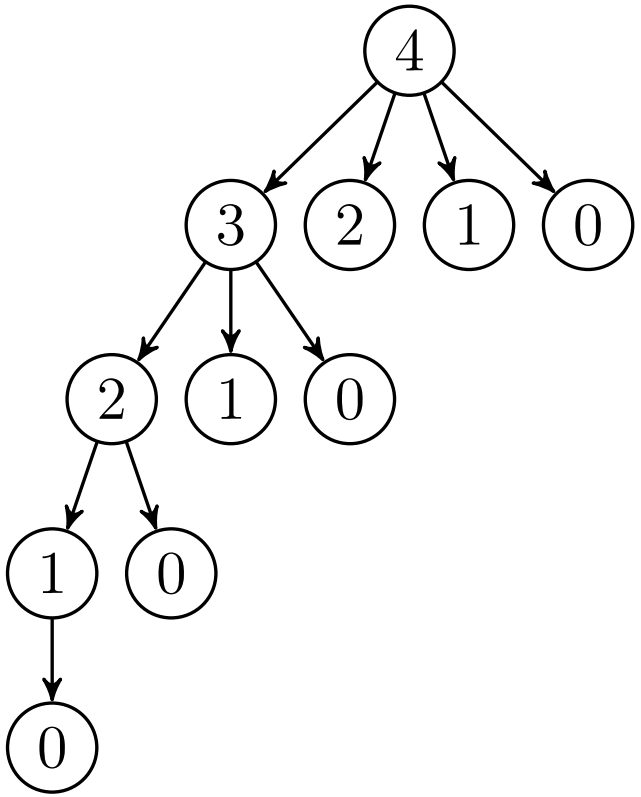
Bug alert!

Returverdiene er feil

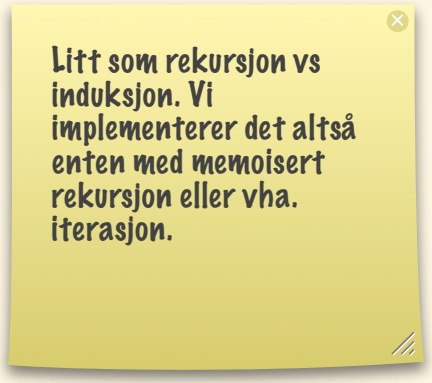


```
AUX(p, n, r)
 1  if r[n] ≥ 0
 2        return r[n]
 3  if n == 0
 4        q = 0
 5  else q = −∞
 6        for i = 1 to n
 7              t = p[i] + AUX(p, n − i, r)
 8              q = max(q, t)
 9  r[n] = q
10  return q
```

→ 0



Top-down vs bottom-up

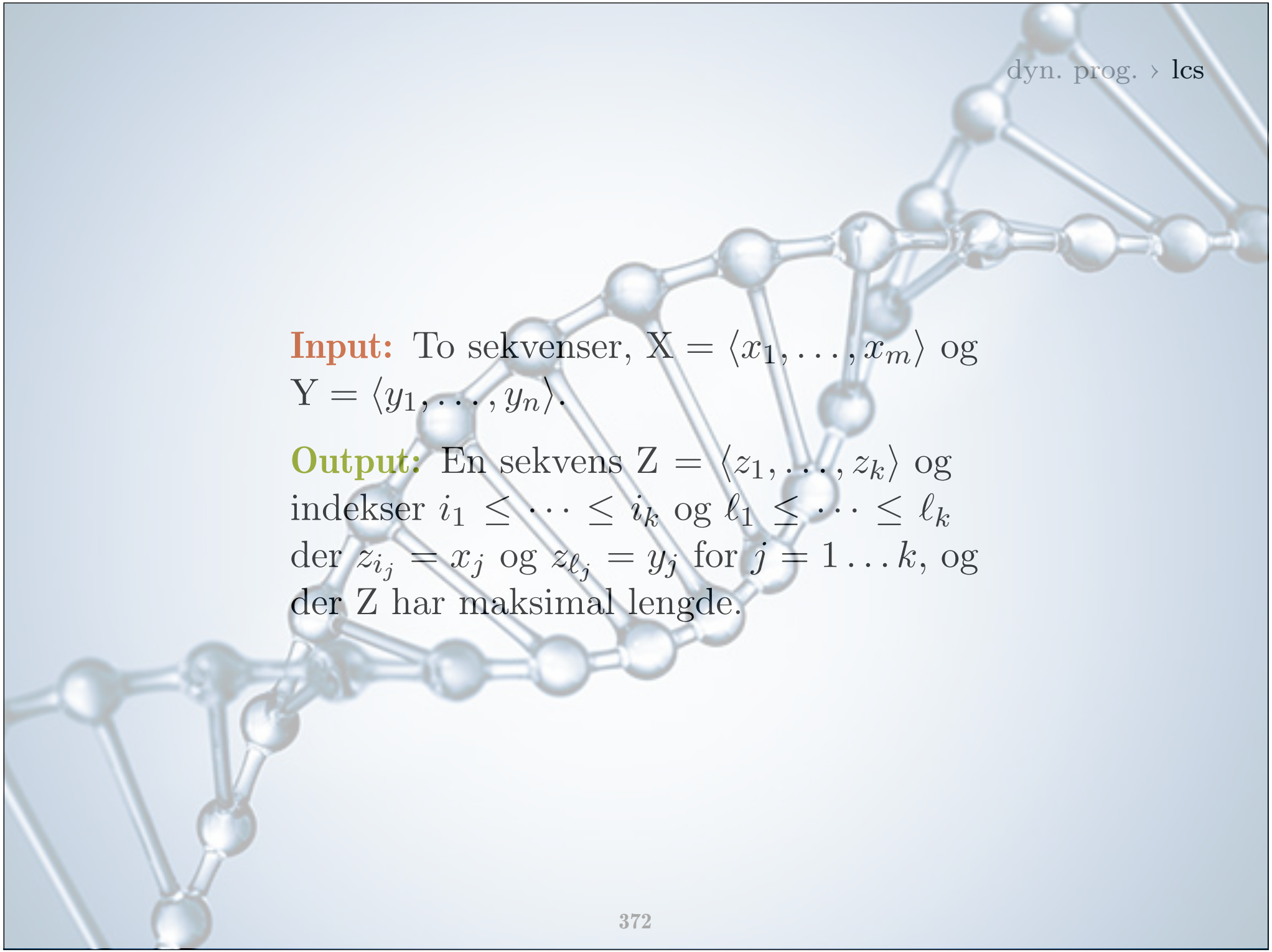


```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

DP › LCS

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

Output: En sekvens $\mathrm{Z} = \langle z_1, \ldots, z_k \rangle$ 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, $\mathrm{X} = \langle x_1, \ldots, x_m \rangle$ og $\mathrm{Y} = \langle y_1, \ldots, y_n \rangle$.

Output: En sekvens $\mathrm{Z} = \langle z_1, \ldots, z_k \rangle$ 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.

klapper takpapp

klapper takpapp

Fjern bokstaver de ikke deler?

kapp akpapp

kapp akpapp

Gjett at app skal brukes?

kapp akpapp

Nytt delproblem i så fall

kapp kapp

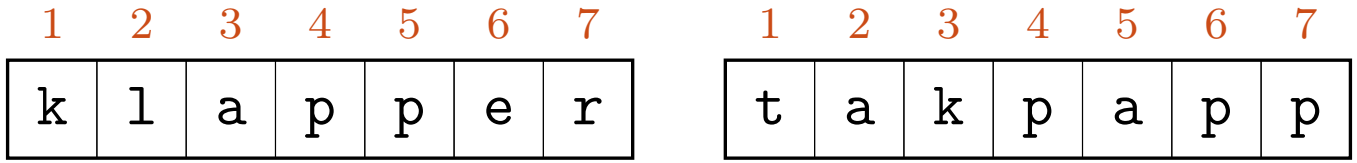
Løst delproblem

kapp kapp

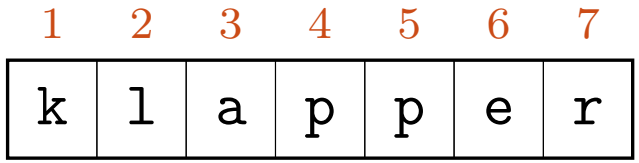
Løsning

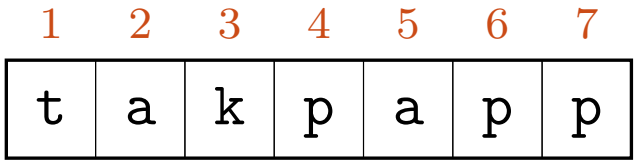
kapp kapp

Men hva om app ikke skulle brukes?

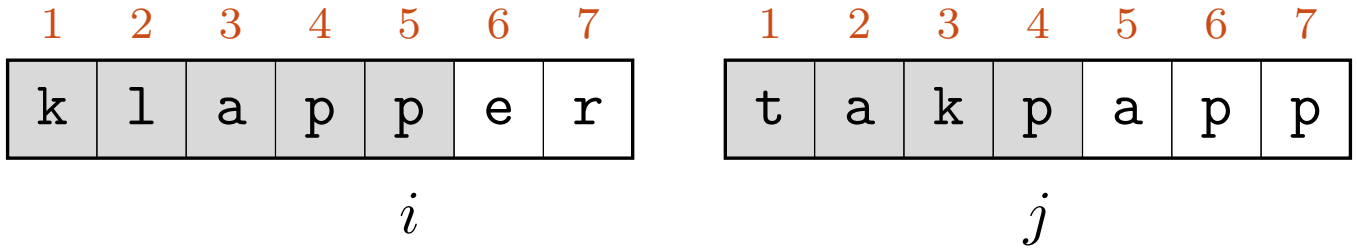


Først: Kartlegg delproblemer

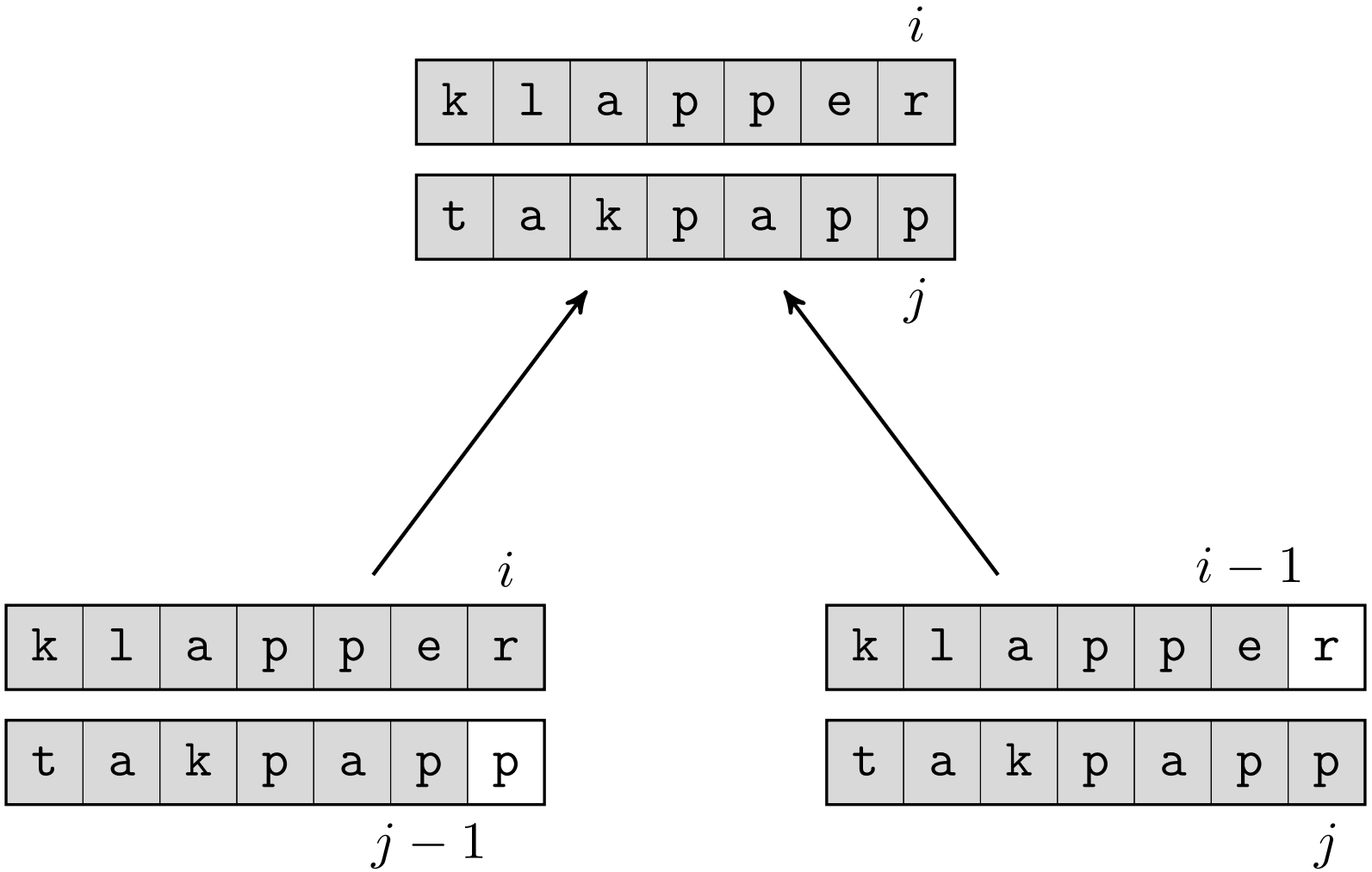




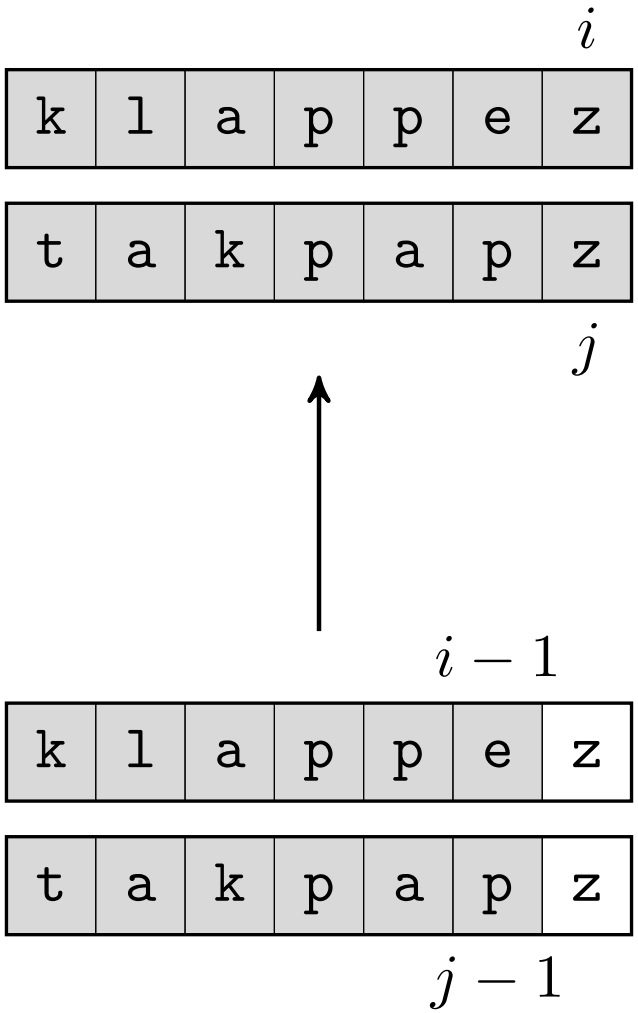
En parameter per sekvens?



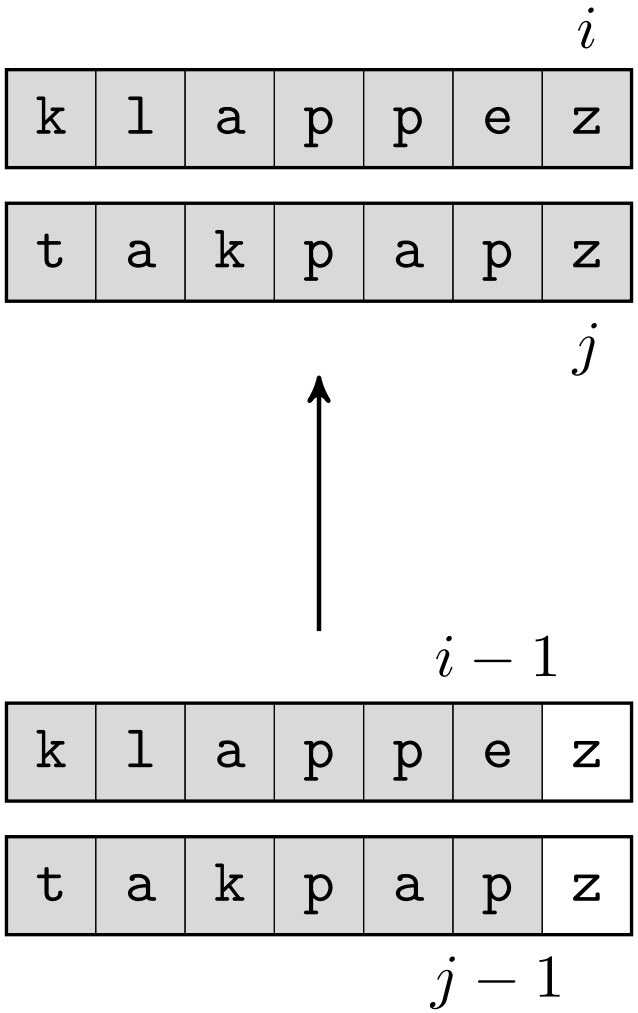
Prefiks, som i Rod-Cutting?



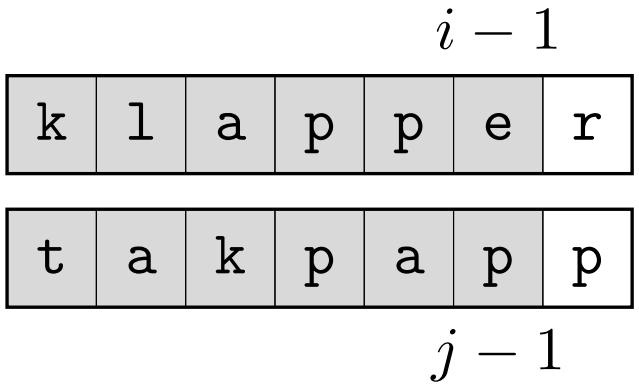
Ulike siste-elementer gir to delproblemer

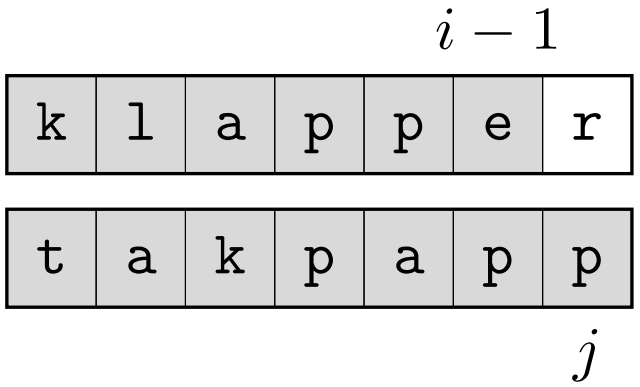


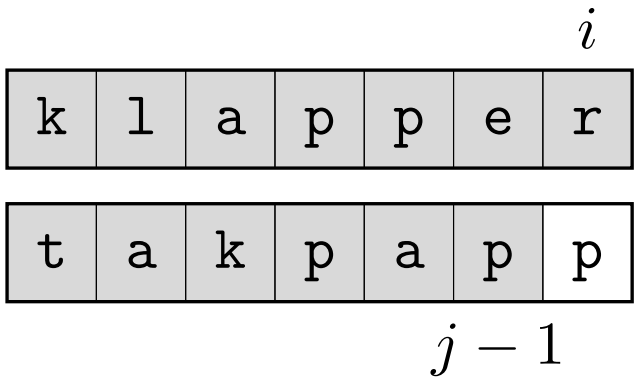
Like siste-elementer gir ett delproblem

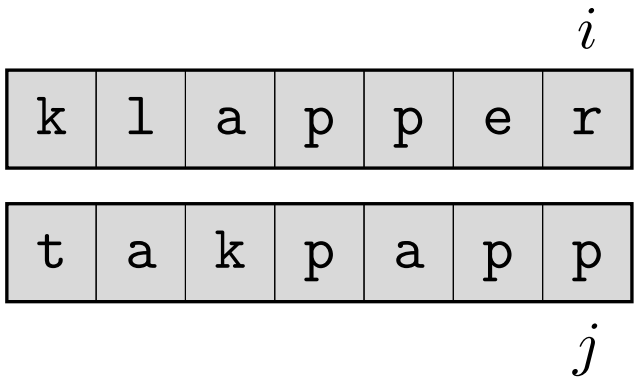


Lønner seg alltid å inkludere like siste-elementer i løsningen

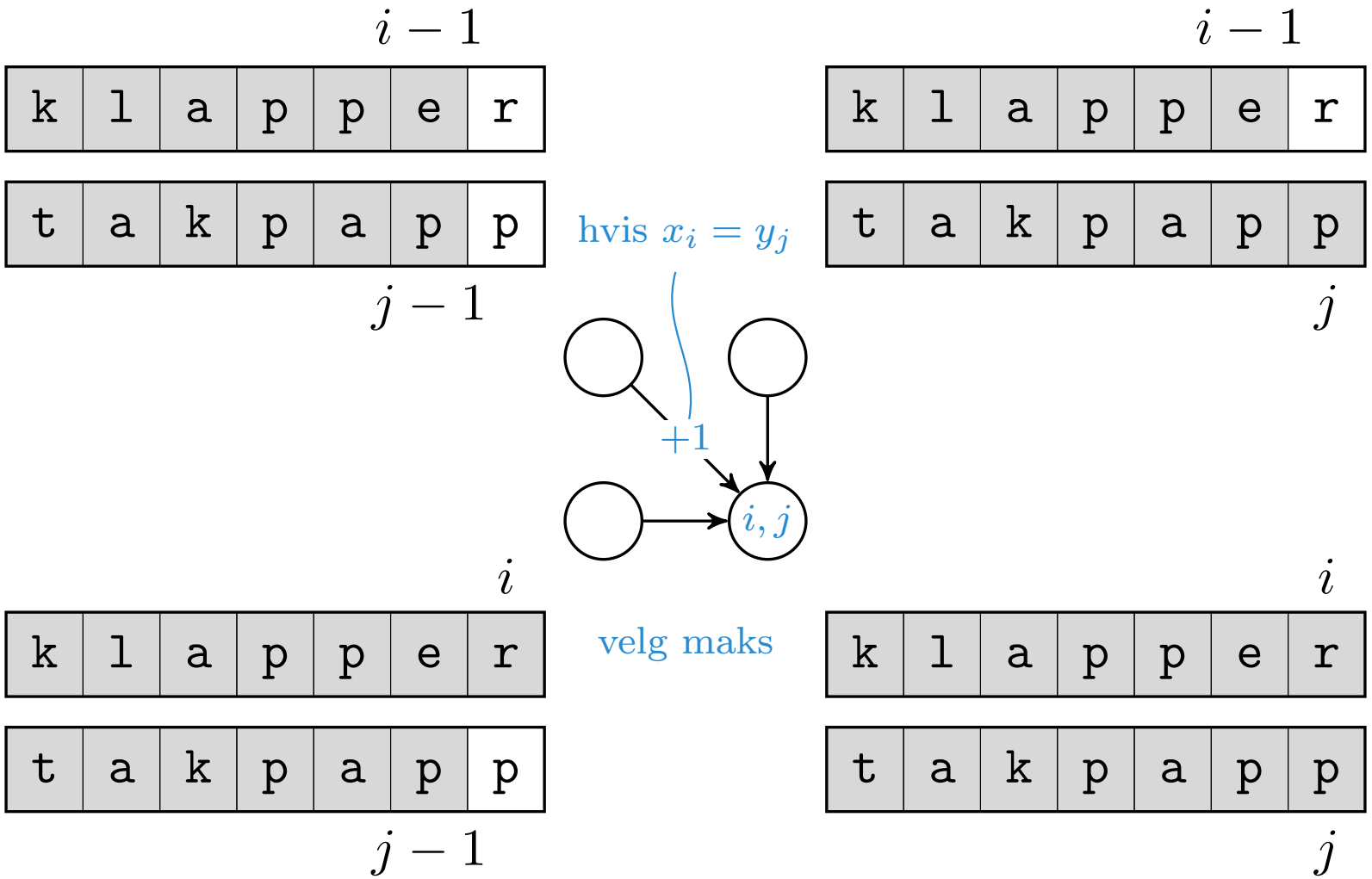






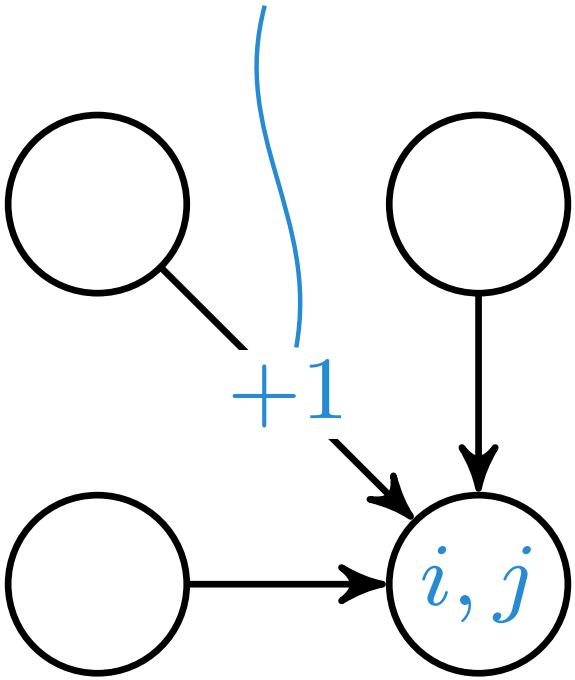


Endelig dekomponering



Endelig dekomponering

hvis $x_i = y_j$



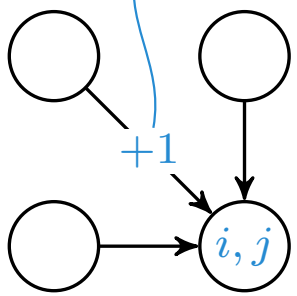
velg maks

			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

Y

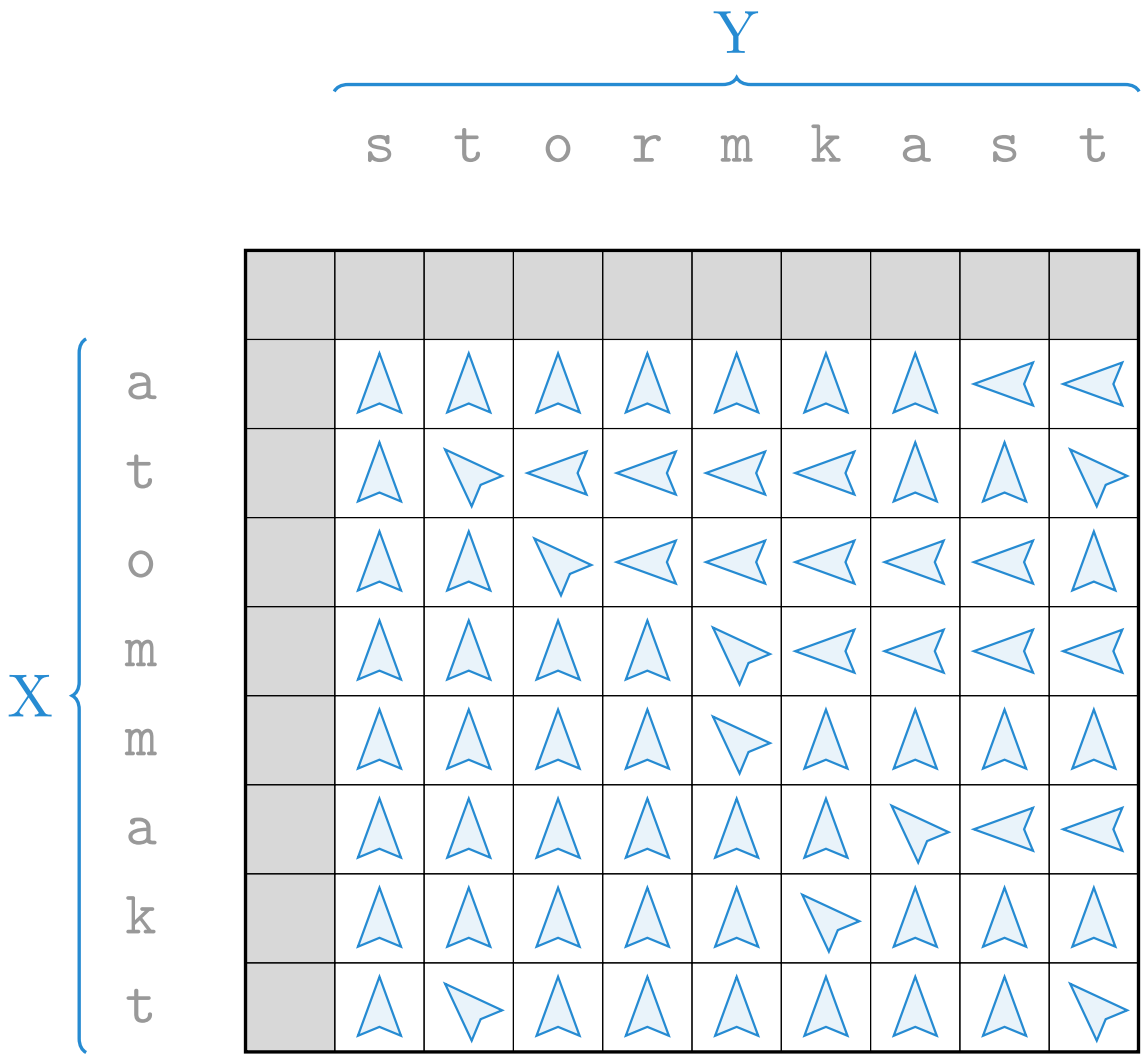
X

hvis $x_i = y_j$

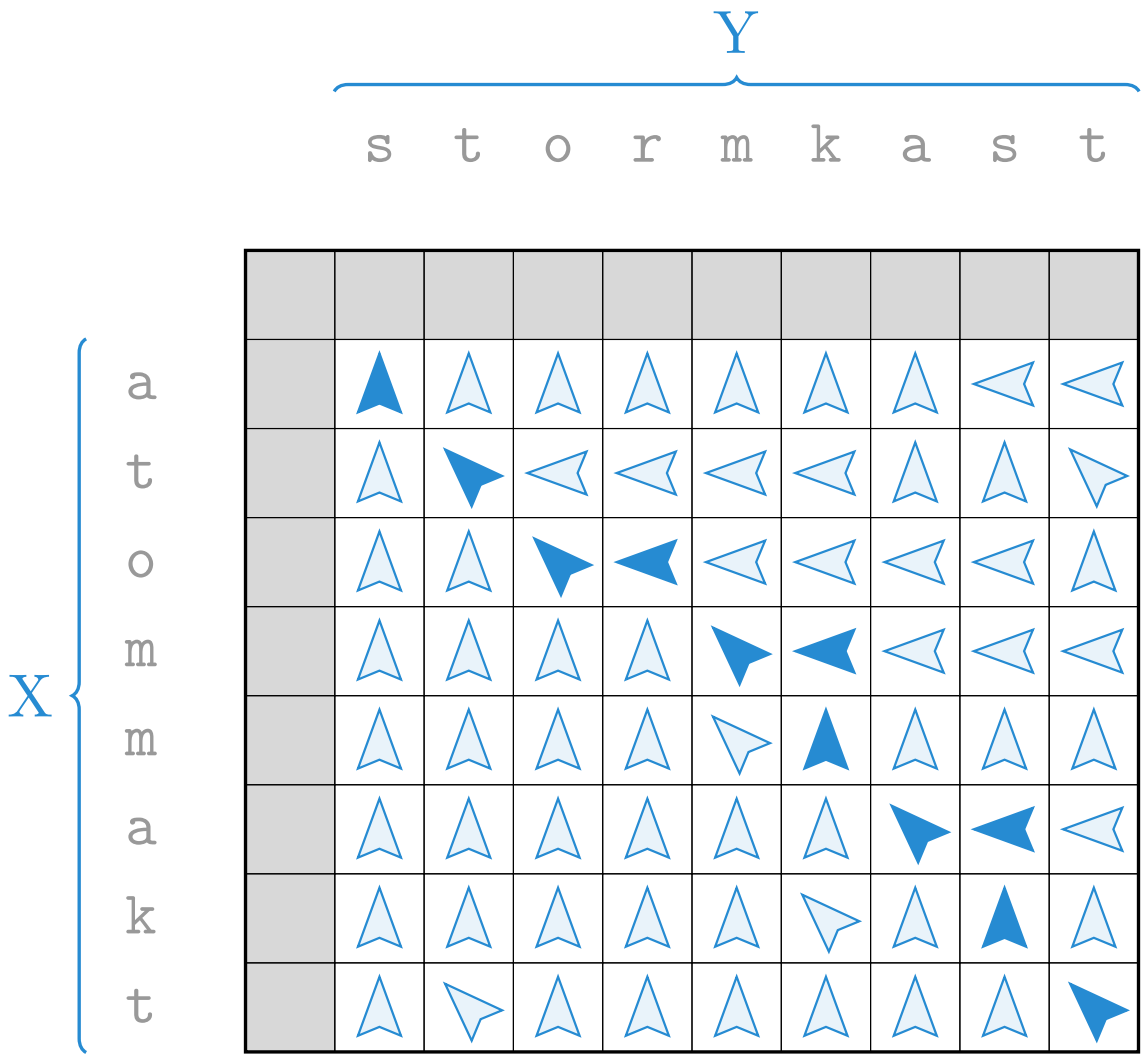


velg maks

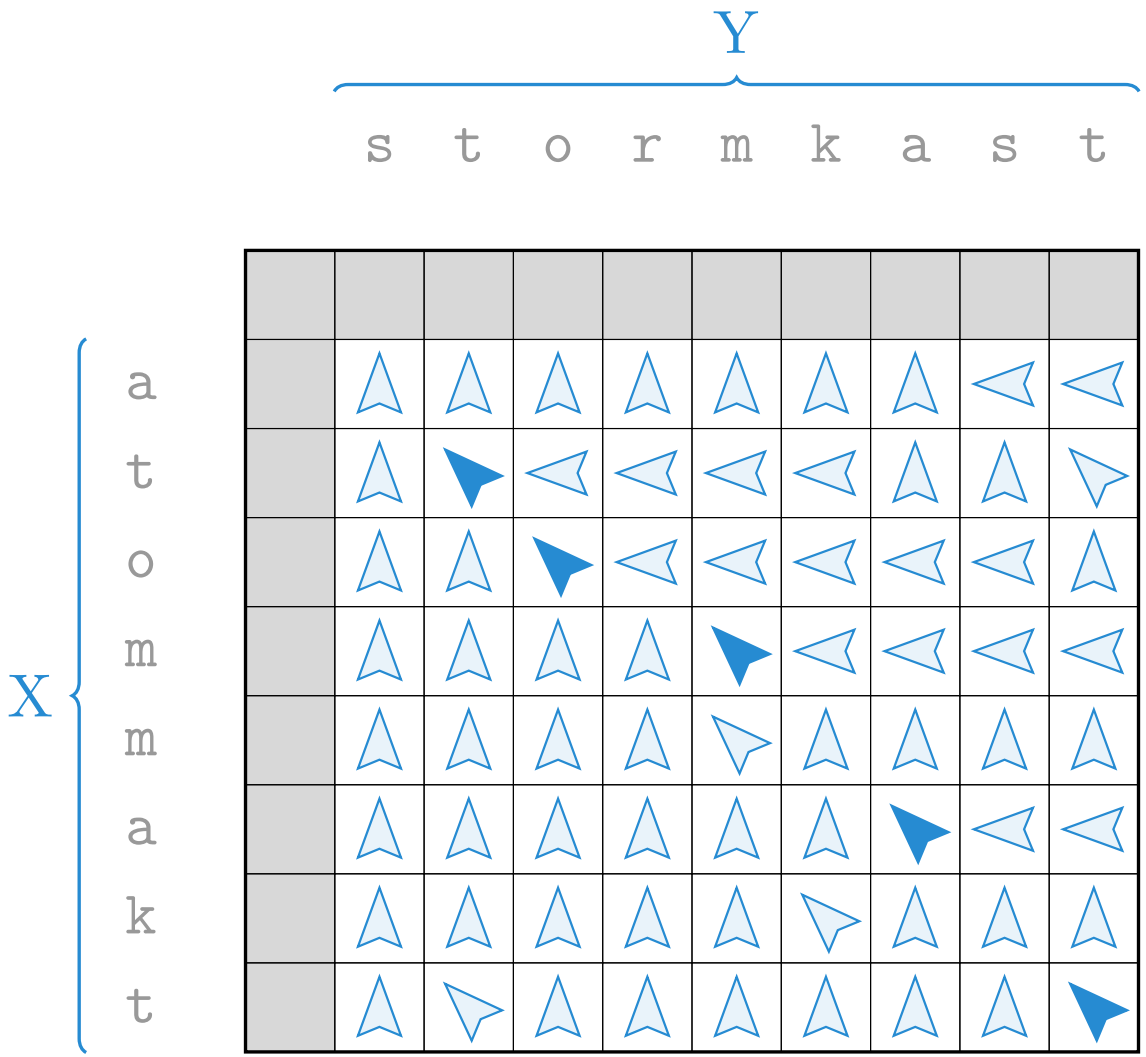
Skal vi hoppe over x_i og/eller y_j?



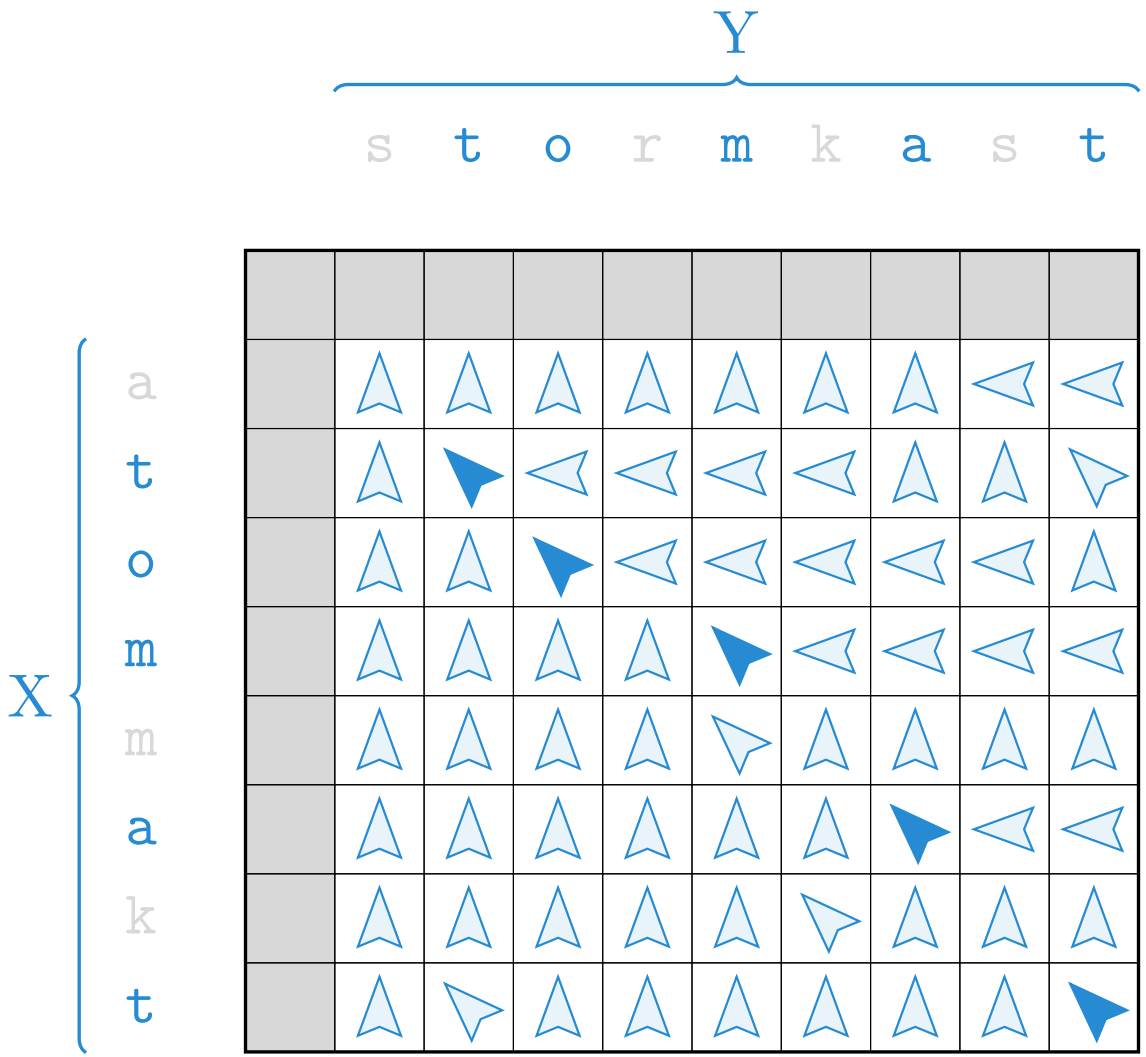
Hvilken delløsning bygger løsning (i, j) på?



Hvilke delløsninger bidro til løsning (n, m)?



Hvilke elementer hoppet vi ikke over?

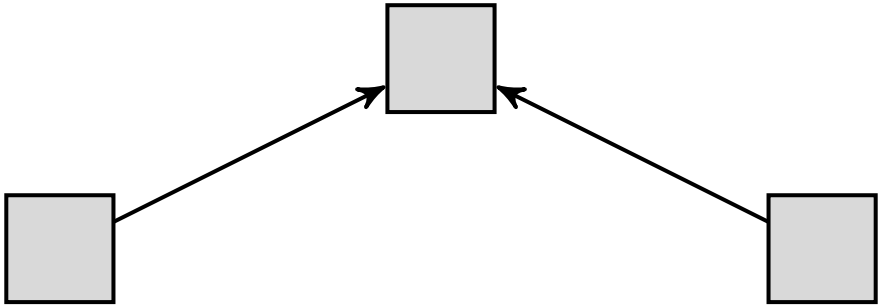


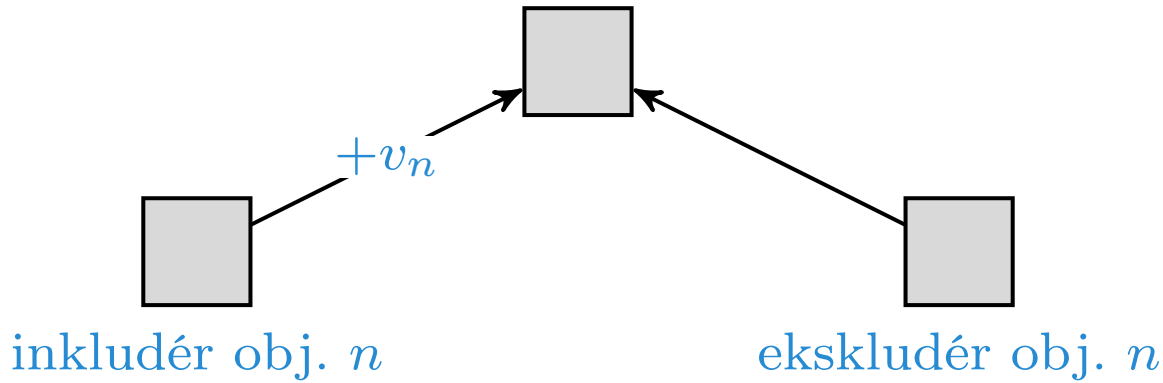
Hvilke elementer hoppet vi ikke over?

DP › Ryggsekk

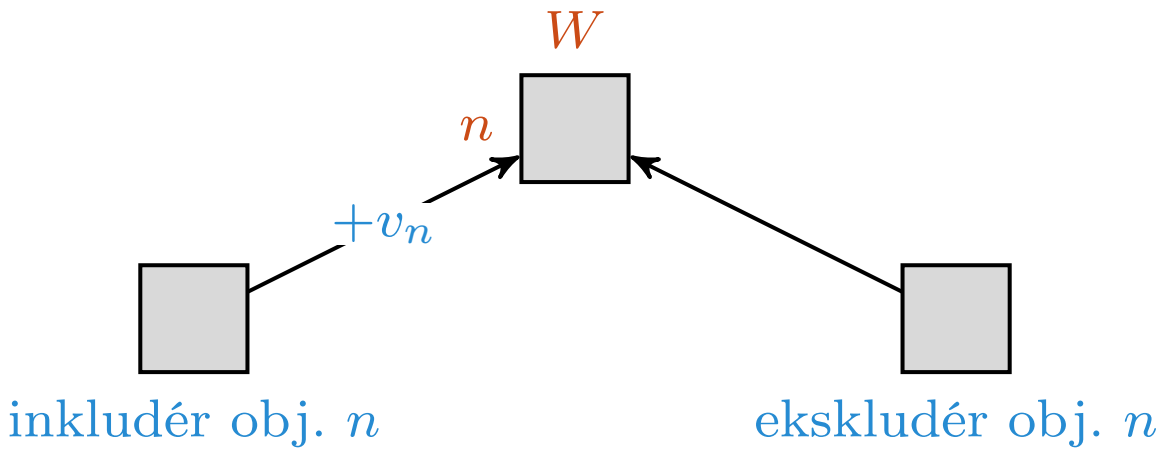
Input: Verdier $v_1, \ldots, v_n$, vekter $w_1, \ldots, w_n$ og en kapasitet W.

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 ma[illegible]

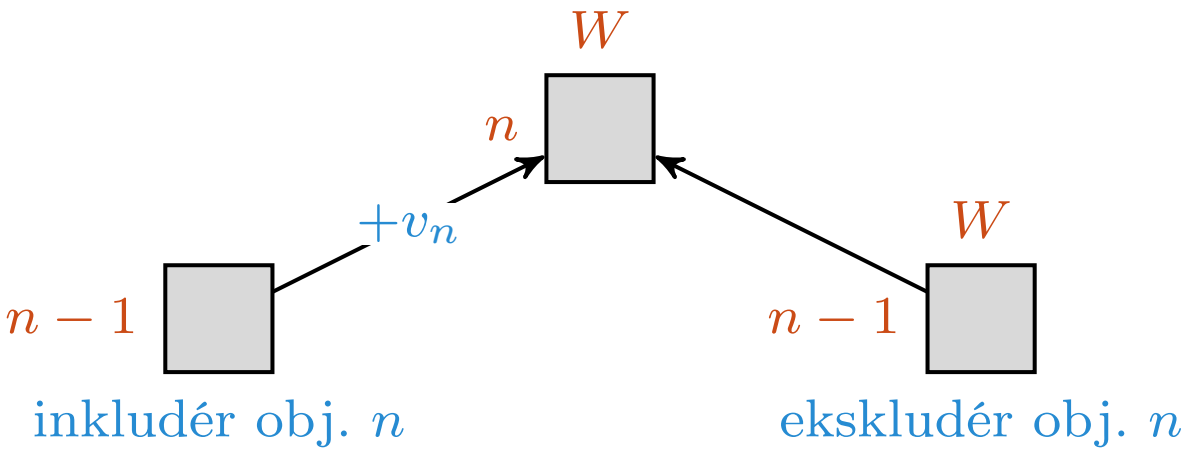




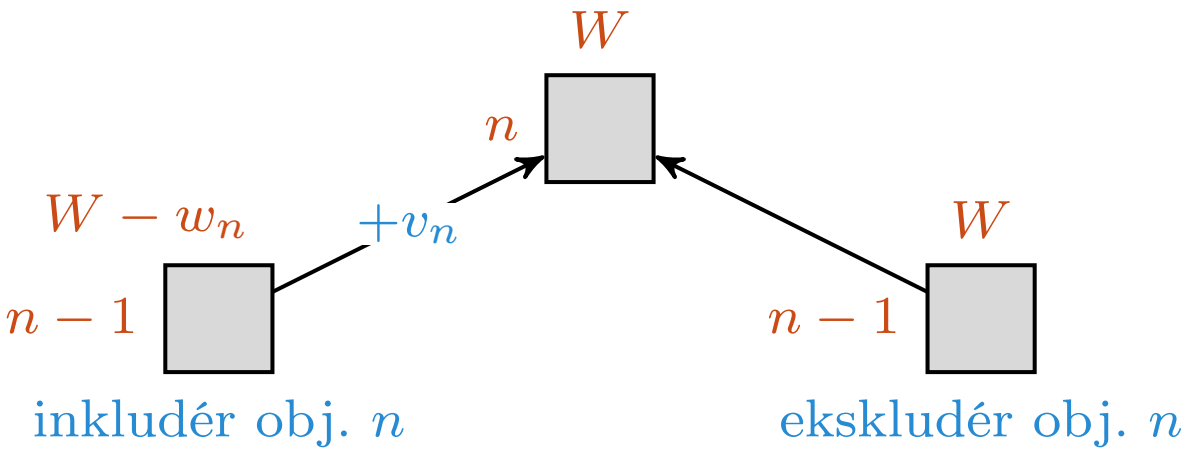
Objekt n bidrar med verdi v_n



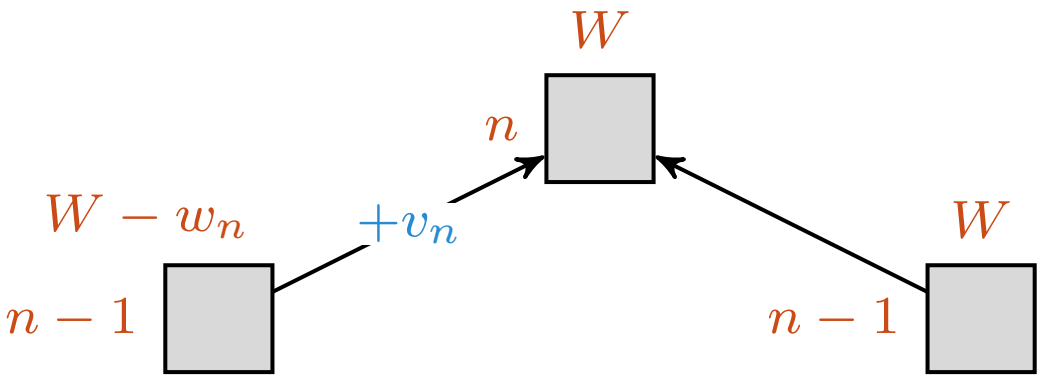
Vi parametriserer delproblemer vha. n og W

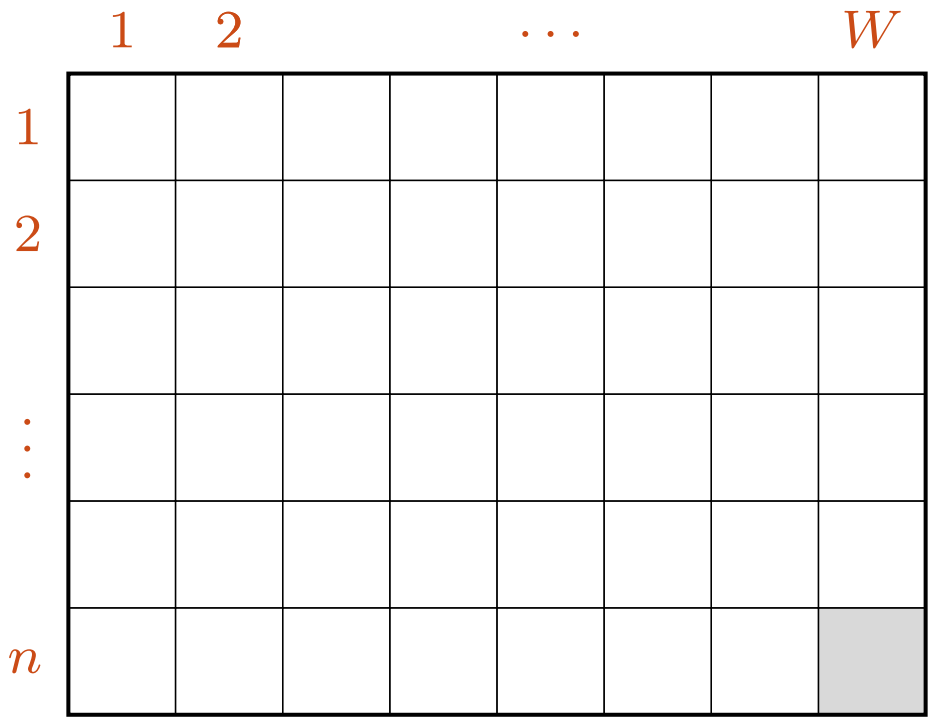


Ser nå bare på objekter $1 \ldots n-1$

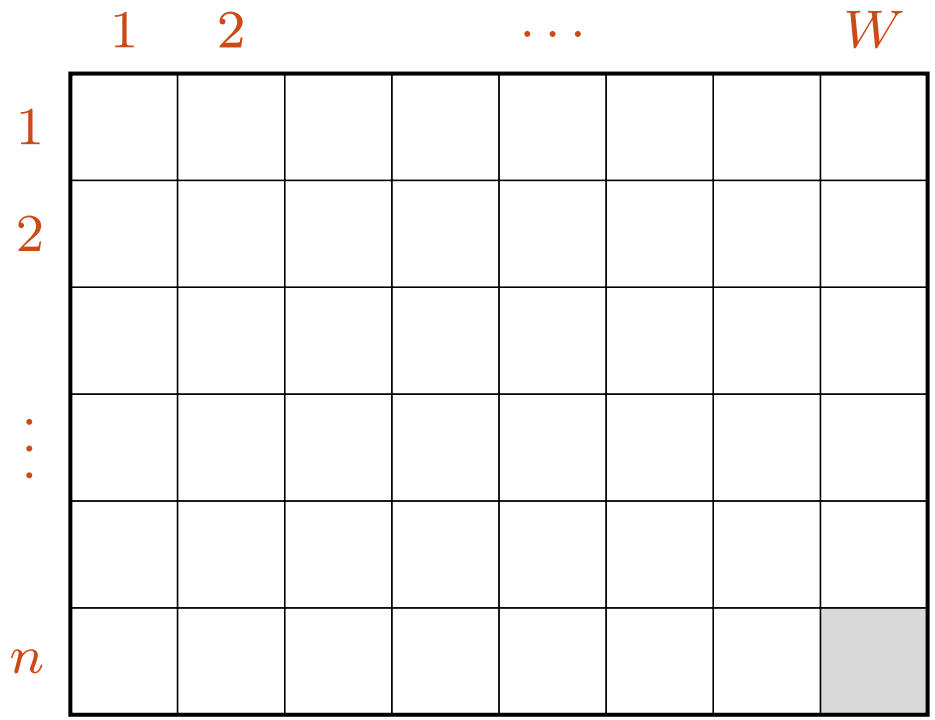


Objekt n bruker opp w_n av W

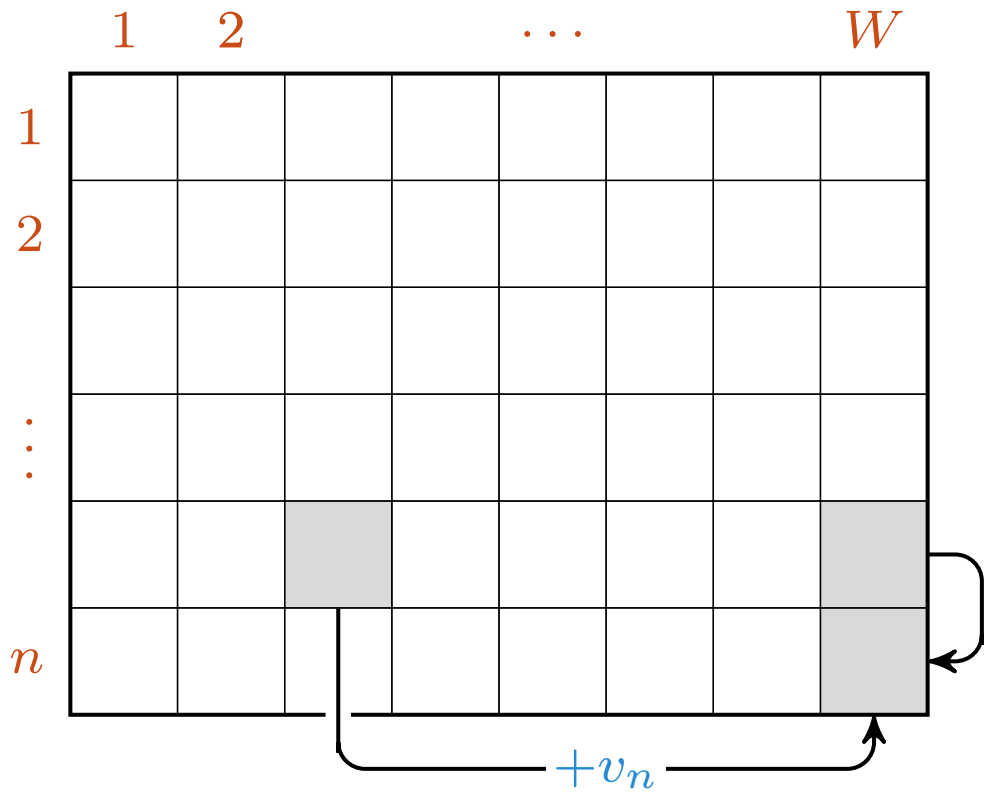

W
n
$W - w_n$
$+v_n$
W
$n-1$
$n-1$



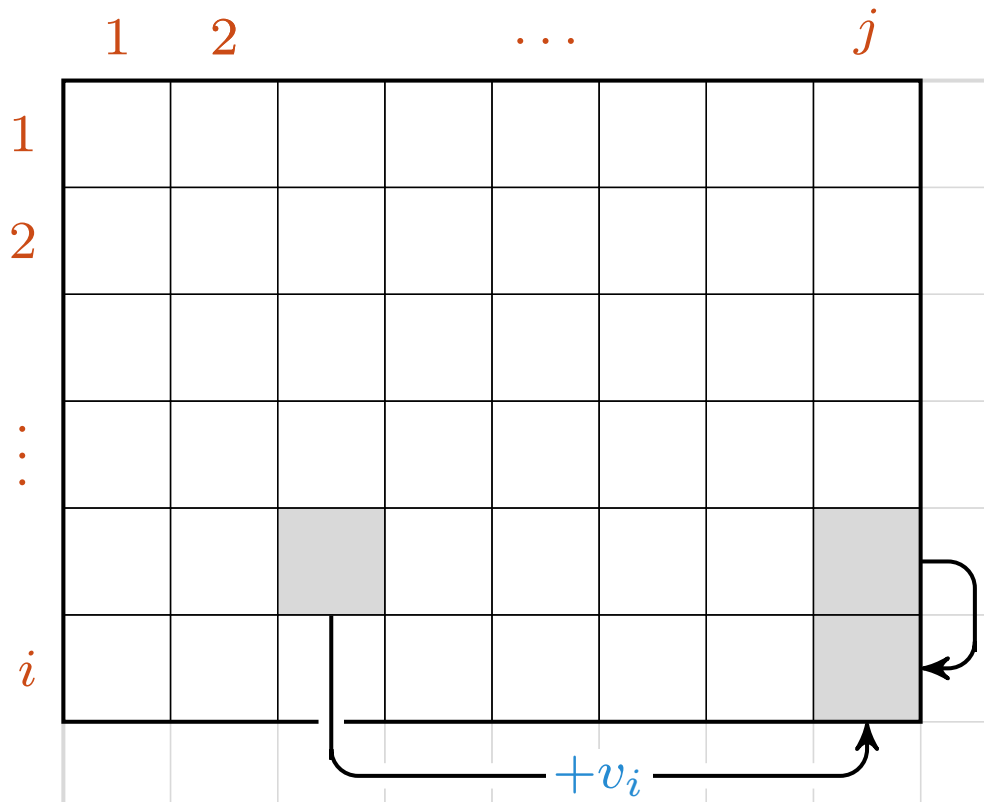
Lagre delløsninger i $n \times W$-tabell



La f.eks. $w_n = 5$.



Dekomponering som før; kan løses radvis

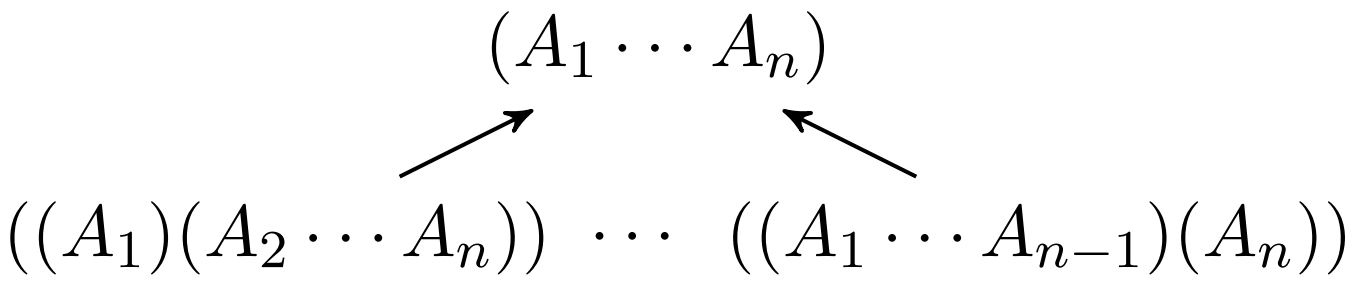


Samme dekomponering for alle delproblemer

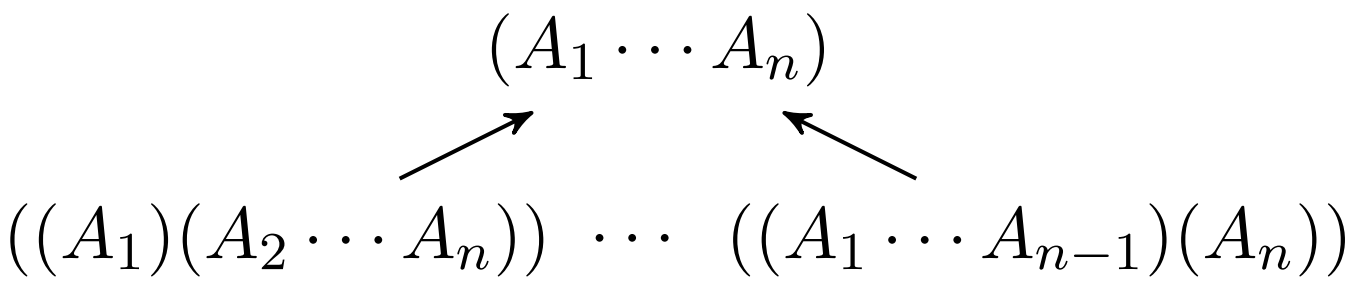
DP › Matrisekjede

Input: En sekvens $\langle A_1, \ldots, A_n \rangle$ med matriser.

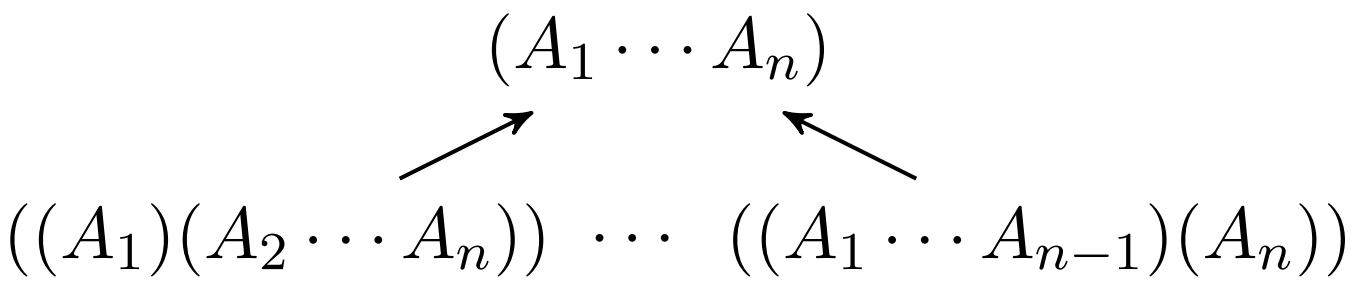
Output: Full parantessetting av produktet $A_1 A_2 \cdots A_n$ slik at antall operasjoner er minimalt.



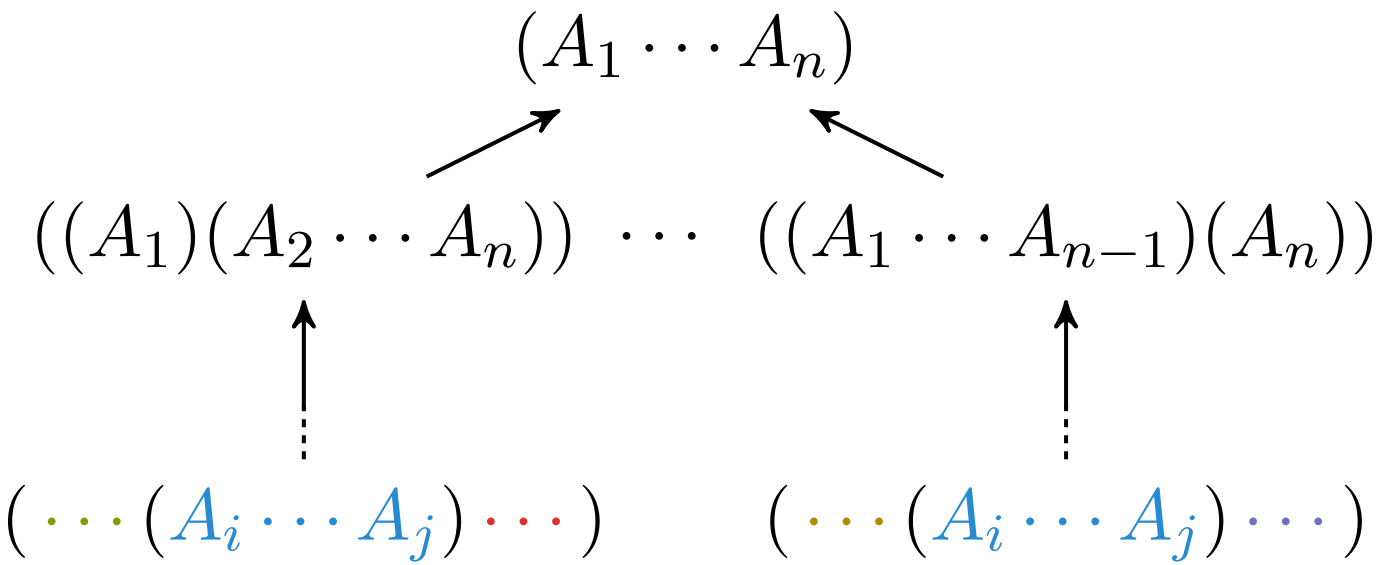
Prøv alle splittpunkter; velg beste



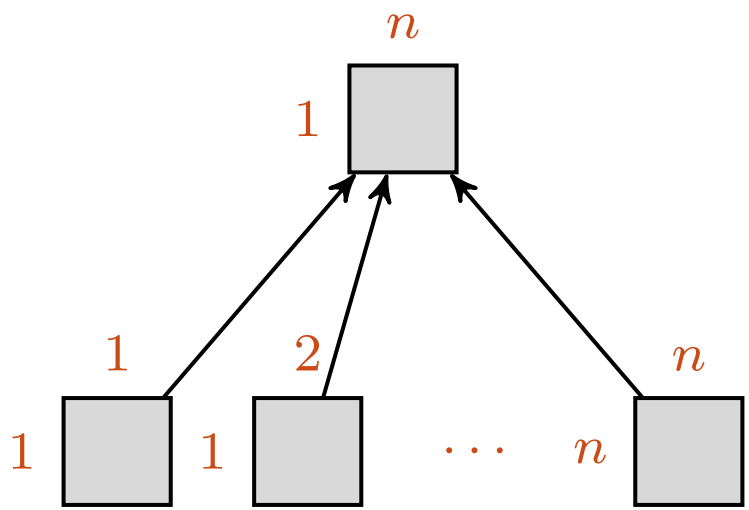
Ett delproblem per splittpunkt?



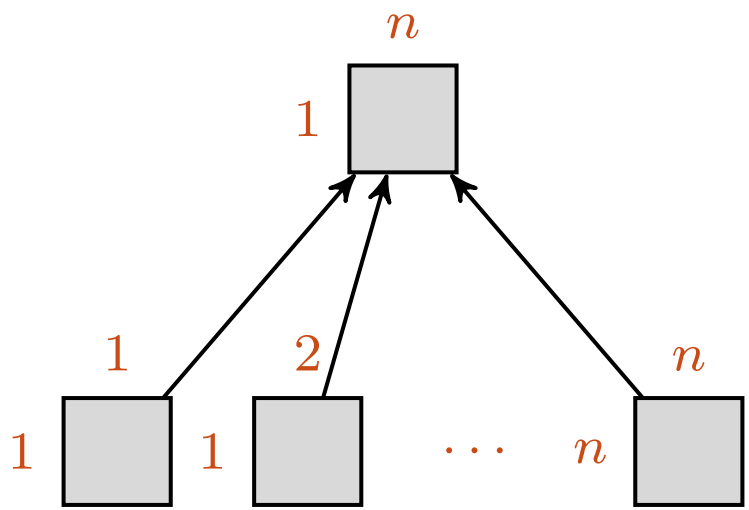
Blir eksponentielt mange nedover; finn overlapp!



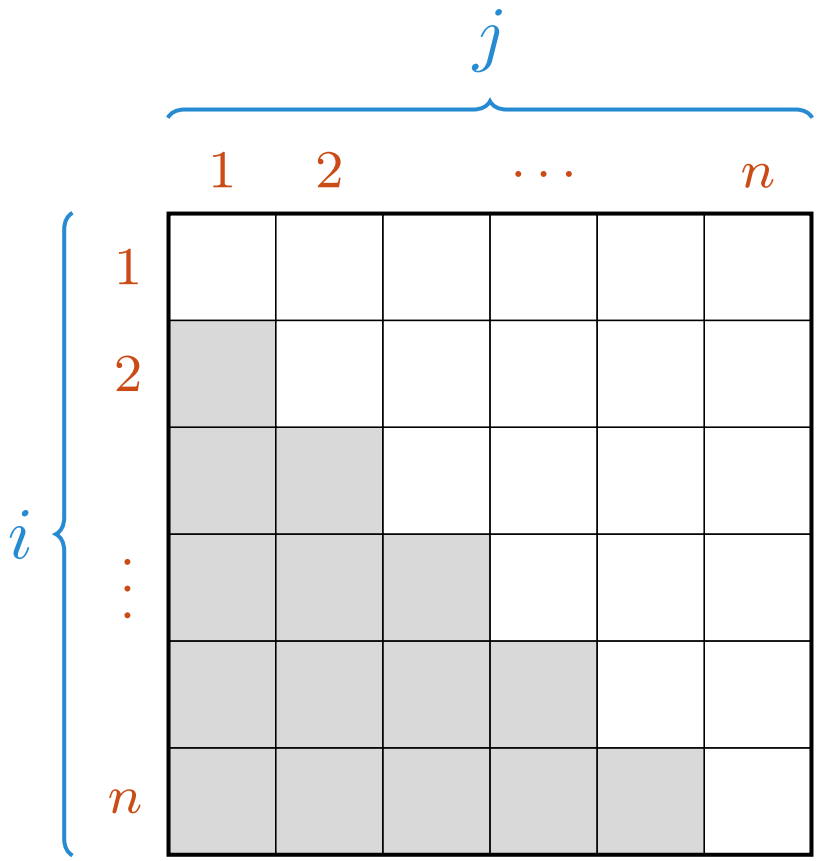
La hvert *intervall* være et delproblem



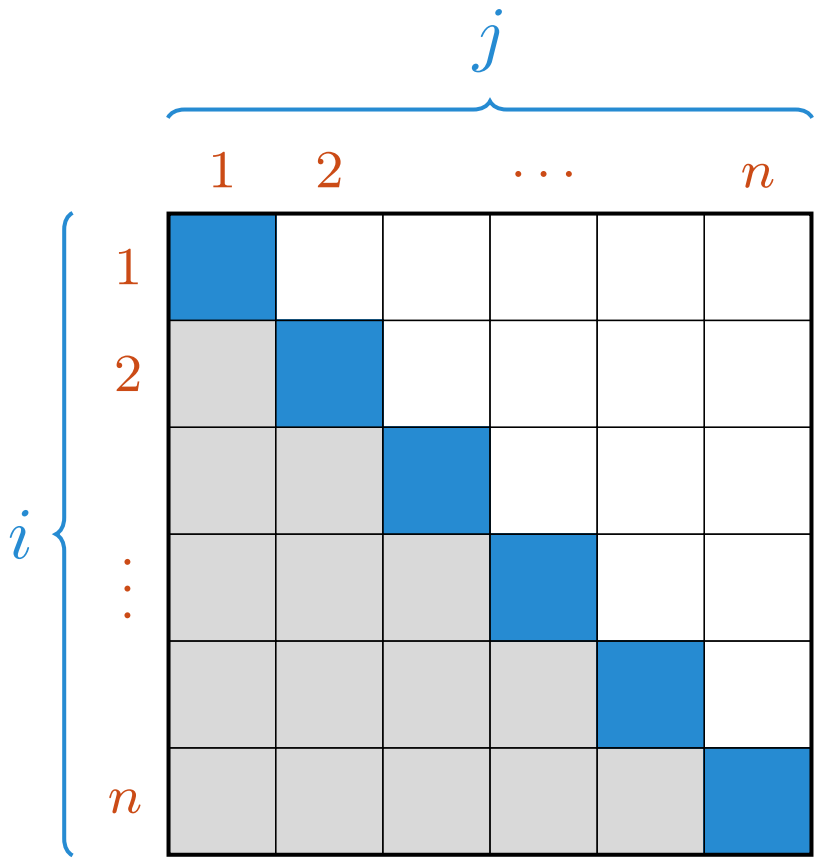
Har funnet optimum for alle kortere intervaller



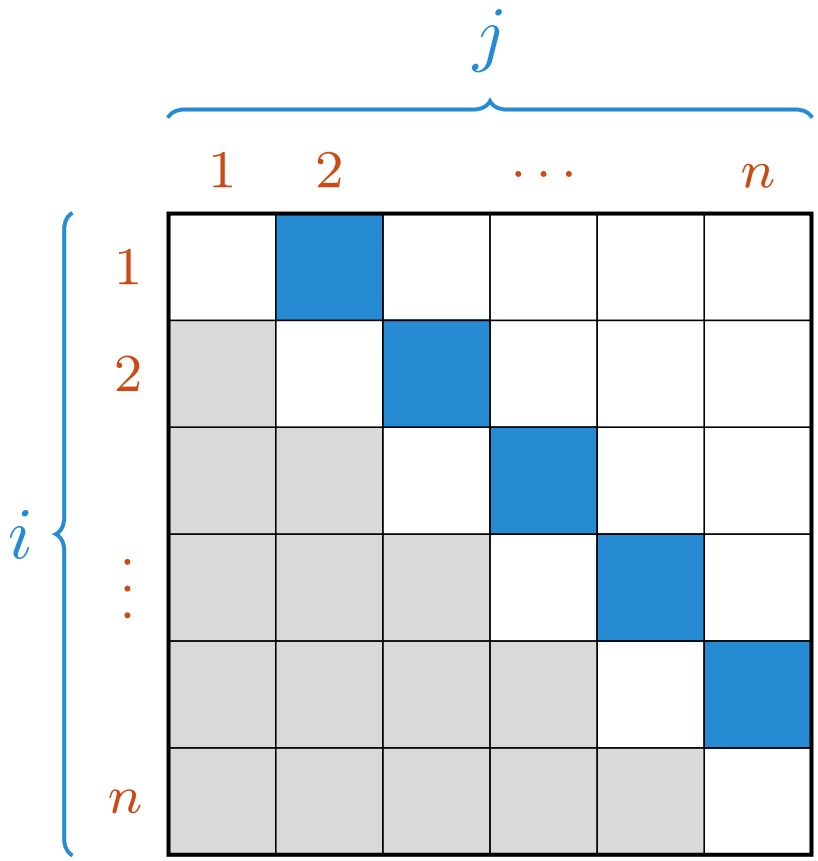
Prøv alle splittpunkter; velg beste



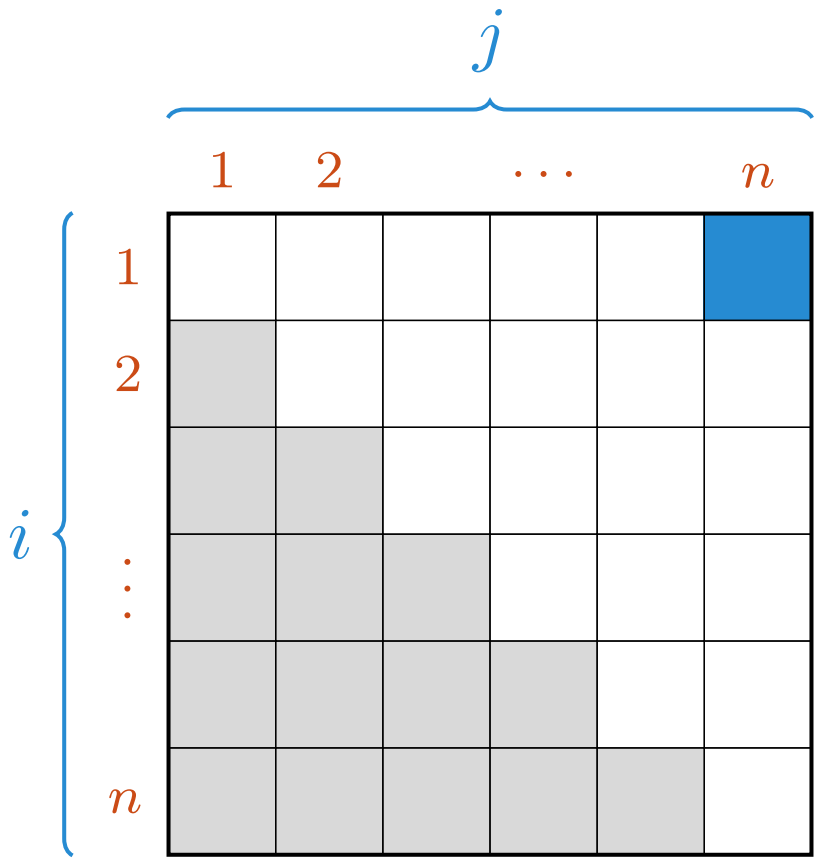
Løsning for $A_i \cdots A_j$ ligger i celle i, j



Løs alle av lengde 1 først



Løs for lengde 2, etc.; avhengigheter er alt ferdige



Fortsett til du har løst $A_1 \cdots A_n$