Dynamisk programmering

Metoden ble formalisert av Richard Bellmann (RAND Corporation) på 50-tallet.

- ”*Programmering*” i betydningen *planlegge*, *ta beslutninger*.
  (Har ikke noe med kode eller å skrive kode å gjøre.)

- ”*Dynamisk*” for å indikere at det er en stegvis prosess. Men også et pynteord.

Hvilke problemer?

- Dynamisk programmering brukes typisk til å løse optimaliseringsproblemer. (Problemer hvor det kan være mange mulige løsninger, og hvor vi ønsker å finne den beste – vi ønsker å optimere en *objektivfunksjon.*

- Skal vi kunne løse et problem med dynamisk programmering, må det kunne deles opp i mindre og mindre biter…

  … helt til vi kommer til et delproblem så lite at løsningen lett kan finnes.

  Så starter vi med løsningene på små delproblemer, og kombinerer disse til løsninger på større delproblemer, helt til vi har løsningen på hele problemet.

- Skal dynamisk programmering være en egnet metode (rask, godt svar) må:

  1. *optimalitetsprinsippet* holde,
  2. delproblemer må overlappe, og
  3. antall delproblemer være polynomisk.

1) Optimalitetsprinsippet

Gitt et optimaliseringsproblem, og en funksjonen *combine* som kombinerer løsninger på delproblemer til løsninger på større problemer, så sier vi at *optimalitetsprinsippet* holder hvis følgende alltid er sant:

Hvis
$S = combine(S_1, S_2, \ldots, S_m)$, og
S er en optimal løsning på sin probleminstans,

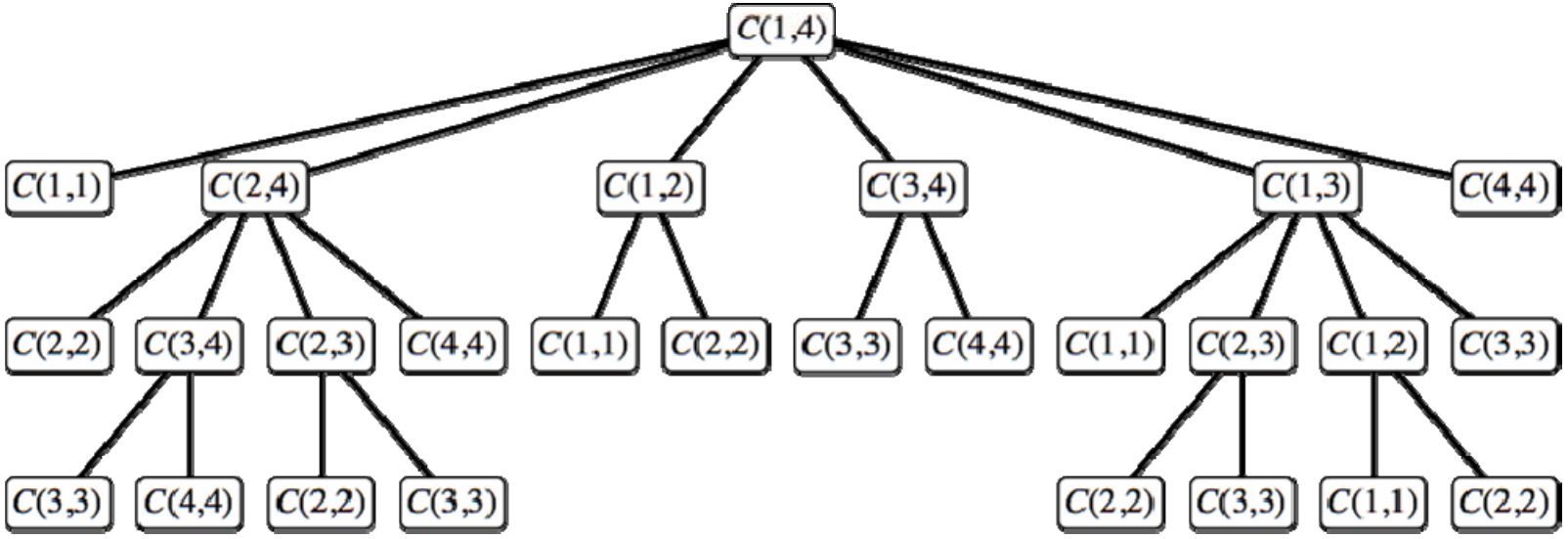
så er
$S_1, S_2, \ldots, S_m$ optimale løsninger på sine respektive probleminstanser.

Hvis optimalitetsprinsippet holder, er problemet egnet for løsning med dynamisk programmering. Da genereres bare optimale løsninger på delproblemer når vi starter fra optimale løsninger på de minste delproblemene, og kombinerer disse til løsninger på større problemer.

2) Overlappende delproblemer

En dynamisk programmerings-algoritme starter med å løse de minste delproblemene, og setter sammen disse løsningene til løsninger på større og større problemer.

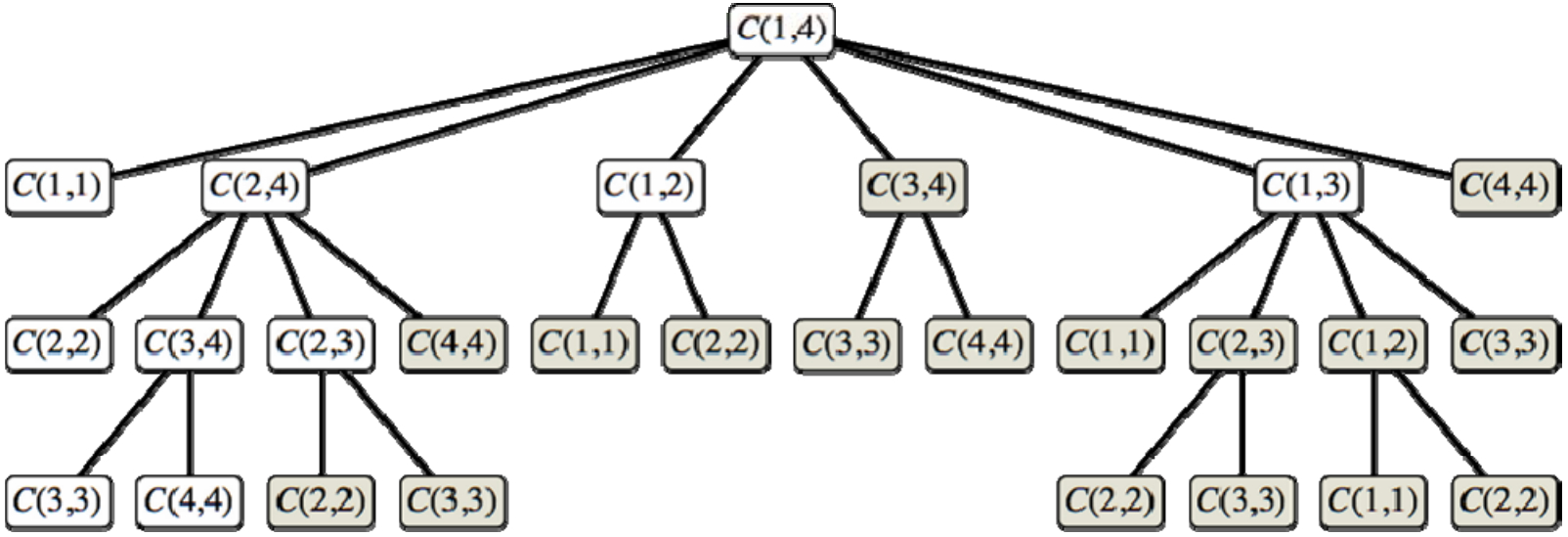
Hvis samme delproblem forekommer flere ganger, unngår vi dobbeltarbeid. Løsningene på delproblemene lagres nemlig, og vi slipper å løse dette delproblemet flere ganger.



2) Overlappende delproblemer

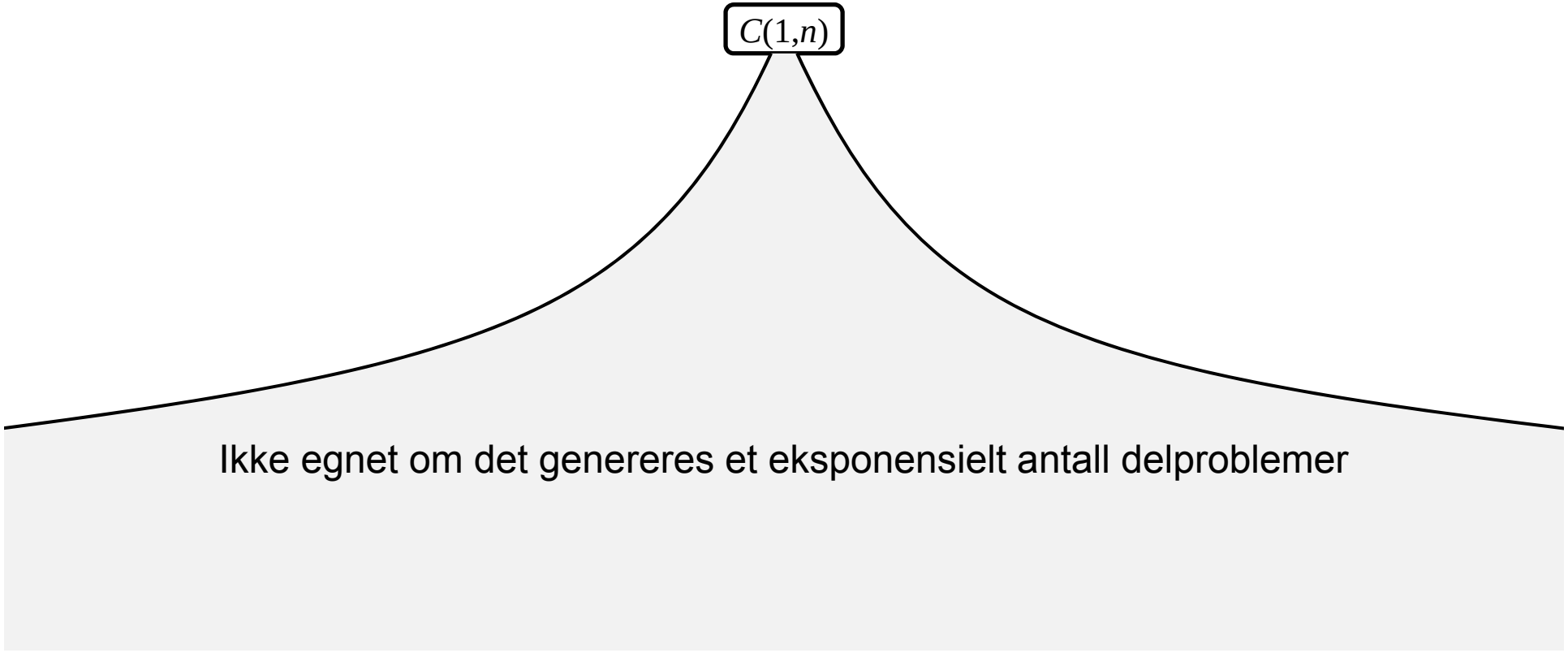
En dynamisk programmerings-algoritme starter med å løse de minste delproblemene, og setter sammen disse løsningene til løsninger på større og større problemer.

Hvis samme delproblem forekommer flere ganger, unngår vi dobbeltarbeid. Løsningene på delproblemene lagres nemlig, og vi slipper å løse dette delproblemet flere ganger.



3) Polynomisk antall delproblemer

En algoritmes effektivitet er direkte avhengig av antall delproblemer som løses: er det få (polynomisk), blir algoritmen effektiv, er det mange (eksponensielt) blir algoritmen ikke praktisk nyttig.



Divide and conquer vs. Dynamisk programmering

Divide and conquer (f.eks Quicksort)

- *Top down* (rekursivt kall)
- Best egnet når delproblemene er uavhengige av hverandre. Da re-beregnes ikke løsninger på delproblemer.
- Kun relevante delproblemer løses.

Dynamisk programmering

- *Bottom up*
- Egnet når delproblemer overlapper, ettersom løsninger lagres i en tabell og vi kan slå opp når vi støter på et delproblem vi allerede har løst.
- Ulempen er at vi beregner løsninger på alle delproblemer.

Metodene kan kombineres, såkalt *memoisering*, en divide and conquer algoritme kan lagre løsninger på delproblemer i en tabell og slå opp for å unngå dobbeltarbeid.

Fire enkle(?) steg

1. Beskriv strukturen i problemet, hvordan en løsning er satt sammen av delløsninger. Verifiser at optimaliseringsprinsippet faktisk holder.

2. Lag en formel for verdien av en optimal løsning (ut fra verdien av av optimale delløsninger).

3. Beregn verdien til en optimal løsning, *bottom up* (lagre verdien av optimale delløsninger i en tabell).

4. Konstruer en optimal løsning ut fra beregnede verdier. (Om vi faktisk ønsker løsningen, og ikke bare nøyer oss med verdien av en optimal løsning.)

Strenger som ligner (kap. 20.5)

En streng *P* er en *k*-approksimasjon av en streng *T* dersom *T* kan konverteres til *P* ved å utføre maksimalt *k* av følgende operasjoner:

Substitusjon	Et symbol i *T* byttes ut med et annet.
Tillegg	Et nytt symbol legges til i *T*.
Sletting	Et symbol slettes fra *T*.

Edit distance, *ED*(*P*,*T*), mellom to strenger *T* og *P* er det minste antall slike operasjoner som trengs for å konvertere *T* til *P.*

Eks.

logarithm → alogarithm → algarithm → algorithm (+a, -o, a/o)

T *P*

1.

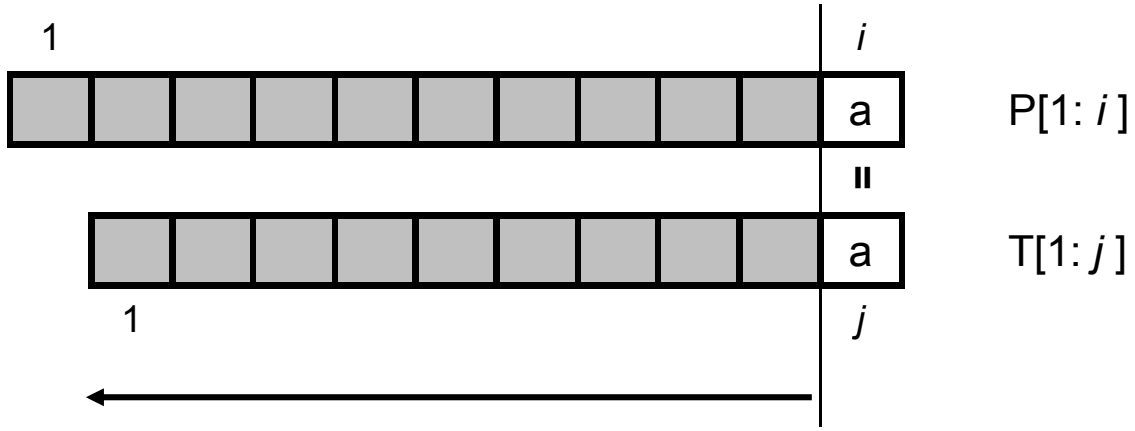
Strenger som ligner

Gitt to strenger *T* og *P*, ønsker vi å finne *edit distance* mellom disse.

La *D* [*i*, *j*] = *ED*(*P* [1: *i*], *T* [1: *j*]). (*Edit distance* mellom delstrenger)

Vi deler i følgende delproblemer:

1. Hvis *P* [*i*] = *T* [*j*], så er *D* [*i*, *j*] = *D* [*i* -1, *j* -1].



1.

Strenger som ligner

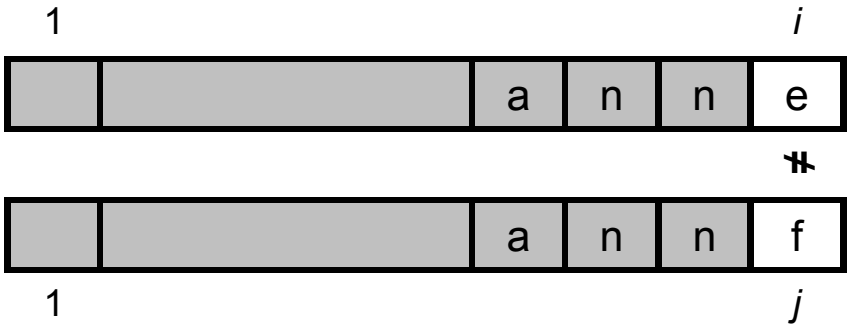
Gitt to strenger T og P, ønsker vi å finne *edit distance* mellom disse.

La $D[i, j] = ED(P[1: i], T[1: j])$. (*Edit distance* mellom delstrenger)

Vi deler i følgende delproblemer:

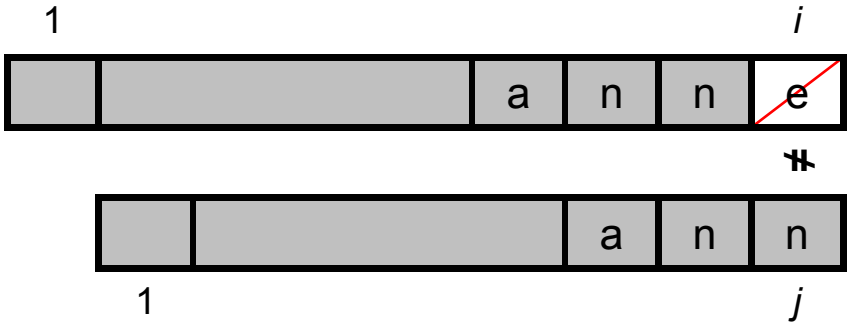
2. Hvis $P[i] \neq T[j]$, tenker vi oss en optimal sekvens av operasjoner som transformerer $T[1: j]$ til $P[1: i]$. Den siste operasjonen, for å få $T[j] = P[i]$, kan være:
   - Substitusjon i T – sette $T[j] = P[i]$,
   - Tillegg i T – legge til symbol i T i pos $T[j]$,
   - Sletting i T – sletting symbol i T på pos $T[j]$.

1. Substitusjon – sette $T[j] = P[i]$



Regner ut $D[i, j]$ ved å finne $D[i-1, j-1]$ (*edit distance* for det grå området), og legger til 1 for substitusjonen.

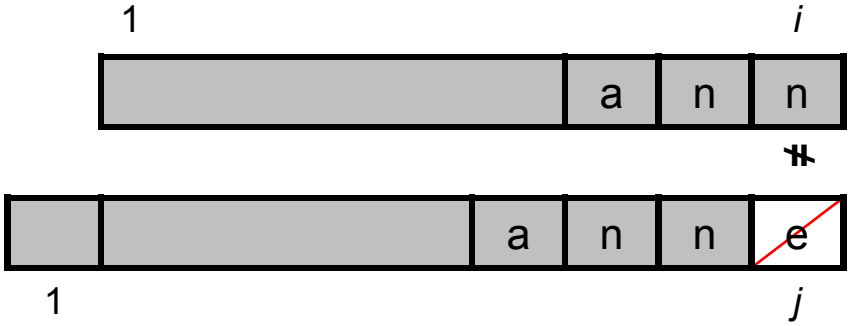
Tillegg i T – legge til symbol i T i pos $T[j]$ – sletting av $P[i]$



Å legge til en e i T er det samme som å slette en i P.

Regner ut $D[i, j]$ ved å finne $D[i-1, j]$ (*edit distance* for det grå området), og legger til 1 for slettingen i P.

Sletting i T – sletting symbol i T på pos $T[j]$



Regner ut $D[i, j]$ ved å finne $D[i, j-1]$ (*edit distance* for det grå området), og legger til 1 for slettingen i T.

2.

Strenger som ligner

Formelen (rekursiv definisjon) for $D[\,i, j\,]$ vil altså være som følger:

$$D[i,j] = \begin{cases} D[i-1, j-1] & \text{hvis } P[i] = T[j] \\ \min\{\ \underbrace{D[i-1,j-1]+1}_{\text{substitusjon}},\ \underbrace{D[i-1,j]+1}_{\substack{\text{tillegg i } T \\ \text{sletting i } P}},\ \underbrace{D[i,j-1]+1}_{\text{sletting i } T}\ \} & \text{ellers} \end{cases}$$

$$D[0,0] = 0, \quad D[i,0] = D[0,i] = i.$$

Løsningen, *edit distance* mellom strengene, finnes i $D[m,n]$ (P er av lengde m og T av lengde n).

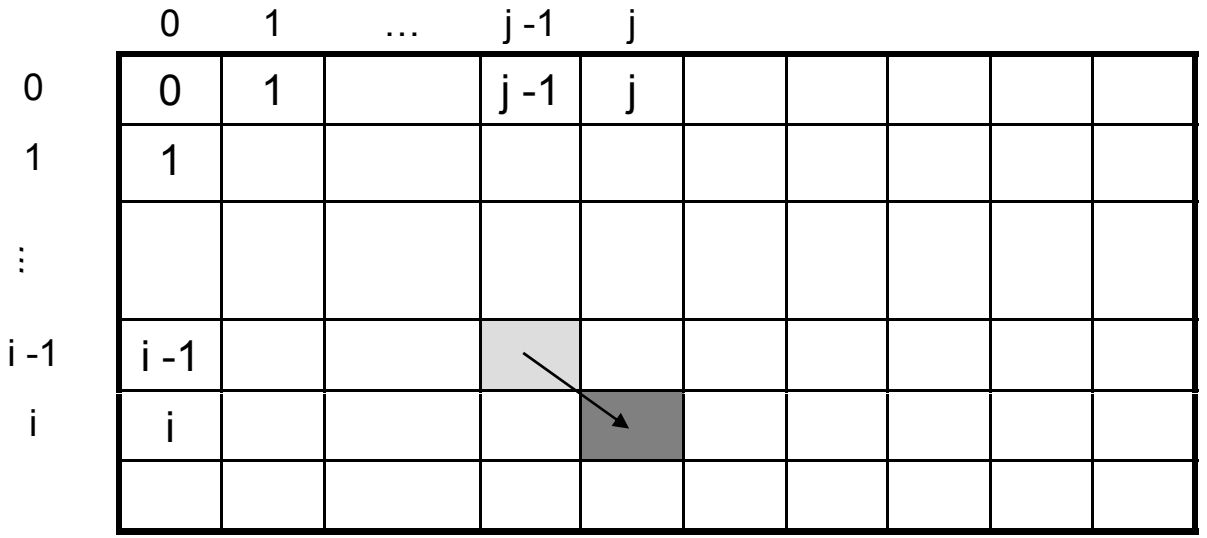
2.

Strenger som ligner

Formelen (rekursiv definisjon) for $D[\,i, j\,]$ vil altså være som følger:

$$D[i,j] = \begin{cases} D[i-1, j-1] & \text{hvis } P[i] = T[j] \\ \min\{\ \underbrace{D[i-1, j-1]+1}_{\text{substitusjon}},\ \underbrace{D[i-1, j]+1}_{\substack{\text{tillegg i } T \\ \text{sletting i } P}},\ \underbrace{D[i, j-1]+1}_{\text{sletting i } T}\ \} & \text{ellers} \end{cases}$$

$$D[0,0] = 0, \quad D[i,0] = D[0,i] = i.$$



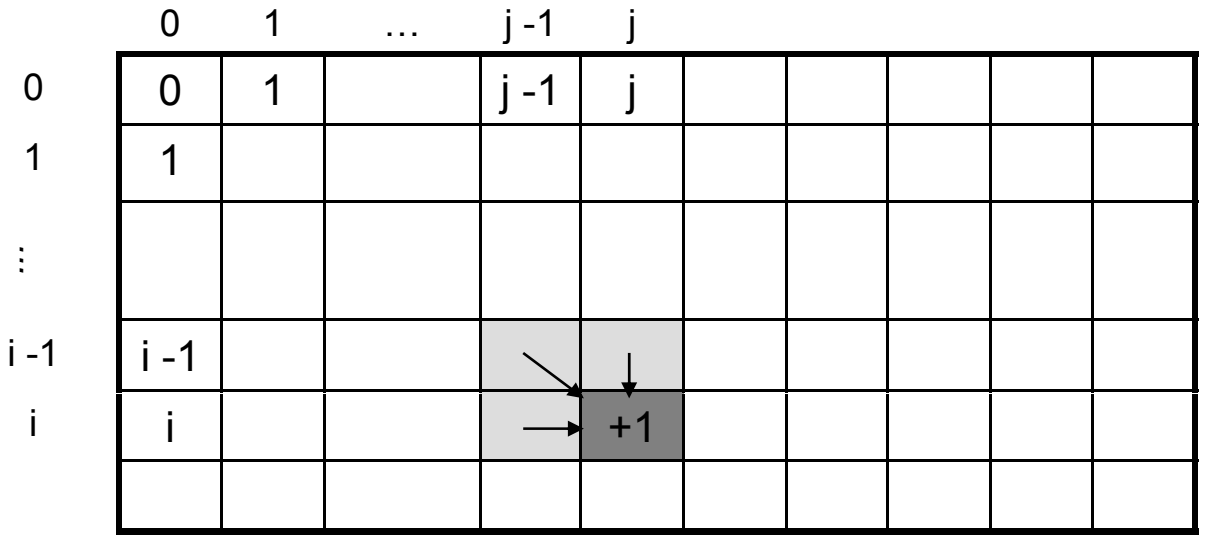
2.

Strenger som ligner

Formelen (rekursiv definisjon) for $D[\,i, j\,]$ vil altså være som følger:

$$D[i,j] = \begin{cases} D[i-1, j-1] & \text{hvis } P[i] = T[j] \\ \min\{\ \underbrace{D[i-1, j-1]+1}_{\text{substitusjon}},\ \underbrace{D[i-1, j]+1}_{\substack{\text{tillegg i } T \\ \text{sletting i } P}},\ \underbrace{D[i, j-1]+1}_{\text{sletting i } T}\ \} & \text{ellers} \end{cases}$$

$$D[0,0] = 0, \quad D[i,0] = D[0,i] = i.$$



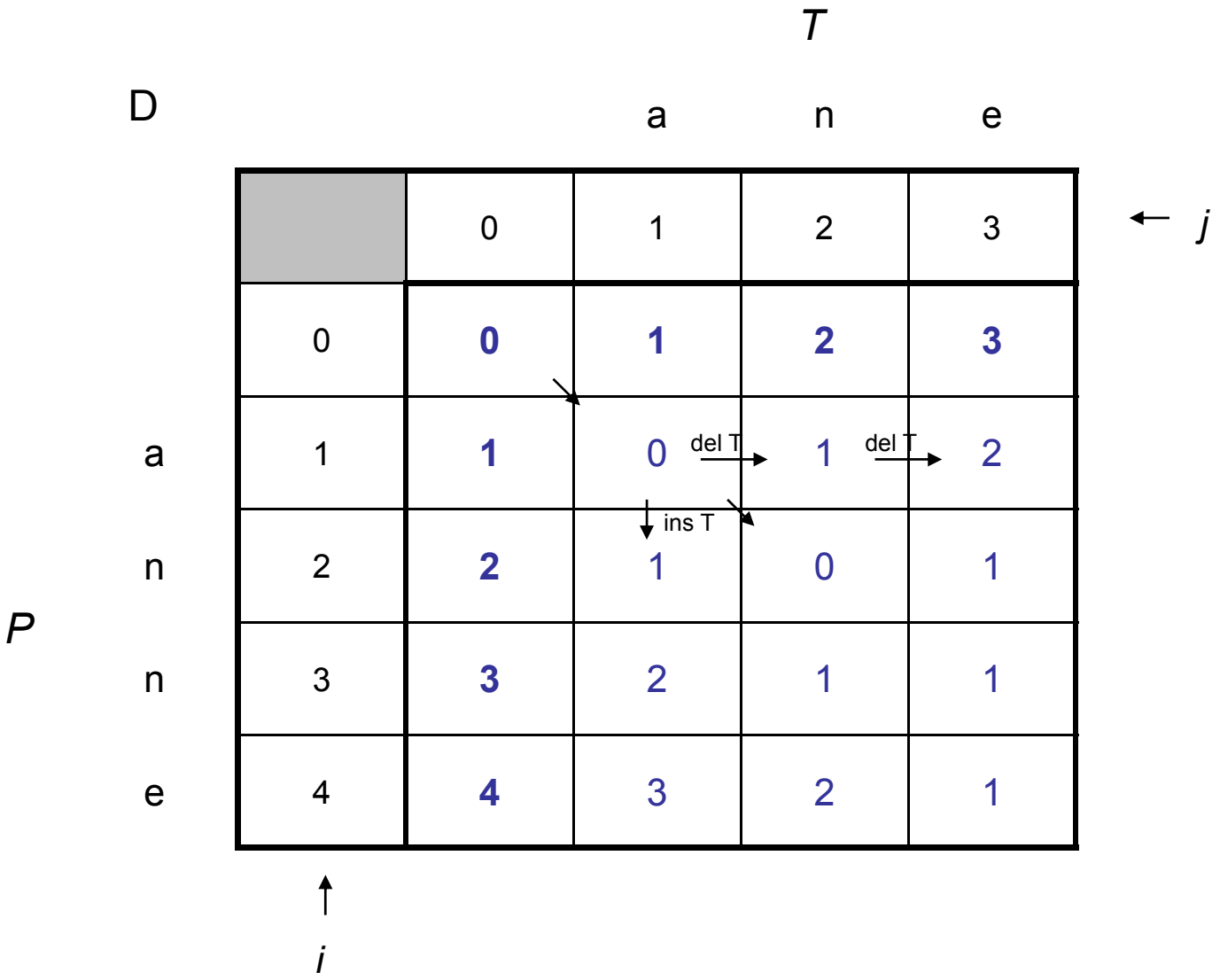
Strenger som ligner

```
function EditDistance ( P [1:n ], T [1:m ] )
    i ← 0
    j ← 0
    for i ← 0 to n do D[ i, 0 ] ← i
    for j ← 1 to m do D[ 0, j ] ← j
    for i ← 1 to n do
        for j ← 1 to m do
            If P [ i ] = T [ j ] then
                D[ i, j ] ← D[ i -1, j - 1 ]
            else
                D[ i, j ] ← min {  D[i -1, j - 1] +1,   D[i -1, j ] +1,   D[i, j - 1] +1  }
            endif
        endfor
    endfor
    return( D[ n, m ] )
end EditDistance
```

3./4.

Strenger som ligner

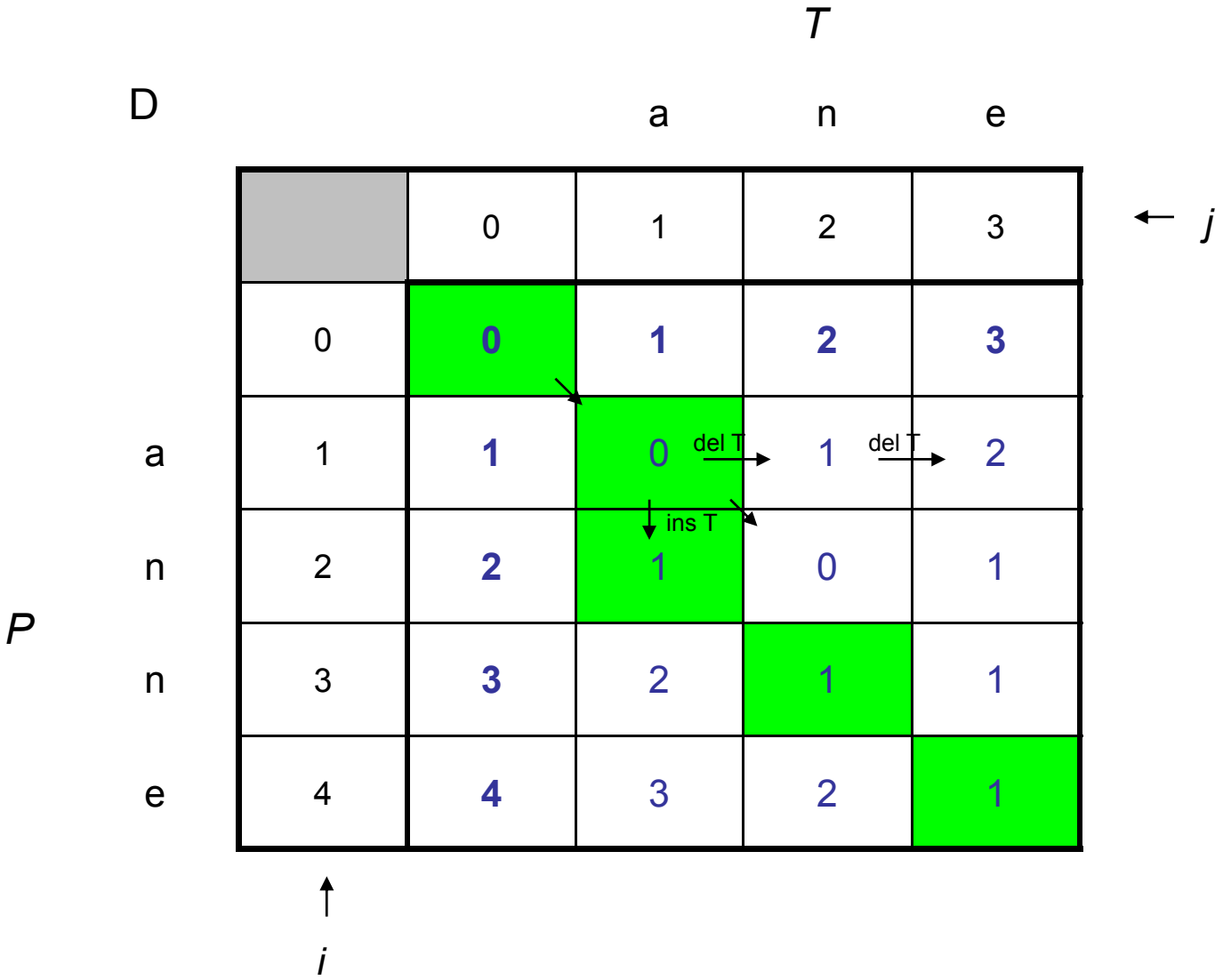
eks.



3./4.

Strenger som ligner

eks.



Optimal matrisemultiplikasjon

Vi er gitt en sekvens $M_0, M_1, \ldots, M_{n-1}$ av matriser og ønsker å beregne produktet

$$M_0 \cdot M_1 \cdot \ldots \cdot M_{n-1}.$$

Vi gjør dette ved å sette parenteser, og multiplisere par av matriser, f.eks slik:

$$M_0 \cdot M_1 \cdot M_2 \cdot M_3 = (M_0 \cdot (M_1 \cdot M_2)) \cdot M_3$$

Parentesene kan settes på mange måter:

$(M_0 \cdot (M_1 \cdot (M_2 \cdot M_3)))$
$(M_0 \cdot ((M_1 \cdot M_2) \cdot M_3))$
$((M_0 \cdot M_1) \cdot (M_2 \cdot M_3))$
$((M_0 \cdot (M_1 \cdot M_2)) \cdot M_3)$
$(((M_0 \cdot M_1) \cdot M_2) \cdot M_3)$

Kostnaden (antall skalare multiplikasjoner) kan variere veldig mellom de ulike måtene å sette parenteser på.

Optimal matrisemultiplikasjon

Gitt to matriser

A = $p \times q$ matrise,

B = $q \times r$ matrise.

Kostnaden (antall skalare multiplikasjoner) ved å beregne $A \cdot B$ er $p \cdot q \cdot r$

($A \cdot B$) er en $p \times r$ matrise.

Eks.

Beregn $A \cdot B \cdot C$, hvor

A er en 10 × 100 matrise, B er en 100 × 5 matrise, og C er en 5 × 50 matrise.

Å beregne D = ($A \cdot B$) koster 5,000 og gir en 10 × 5 matrise.
Å beregne $D \cdot C$ koster 2,500.
Total kostnad for ($A \cdot B$) $\cdot$ C blir 7,500.

Å beregne E = ($B \cdot C$) koster 25,000 og gir en 100 × 50 matrise.
Å beregne $A \cdot E$ koster 50,000.
Total kostnad for A · (B · C) blir 75,000.

1.

Optimal matrisemultiplikasjon

Gitt en sekvens av matriser M_0, M_1, …, M_{n-1} , ønsker vi å beregne produktet på billigst mulig måte – vi må finne optimal parentes-struktur.

En parenterisering av sekvensen er en oppdeling i to delsekvenser, som hver for seg må parenteriseres:

$$(M_0 \cdot M_1 \cdot \ldots \cdot M_k) \cdot (M_{k+1} \cdot M_{k+2} \cdot \ldots \cdot M_{n-1})$$

Vi må prøve alle mulige k for å finne hvor det er best å dele sekvensen.

Hvert delingspunkt gir opphav til to delproblemer – parenterisering av venstre og høyre delsekvens.

Består sekvensen bare av *en* matrise, er denne en parenterisering av seg selv.

Har vi en optimal parenterisering av en sekvens av matriser, så må parenteriseringen av hver delsekvens også være optimal. Ellers kan vi jo bare bytte ut den som ikke var optimal med en bedre en.

(Optimalitetsprinsippet holder.)

2.

Optimal matrisemultiplikasjon

La d_0, d_1, …, d_n være dimensjonene til sekvensen av matriser M_0, M_1, …, M_{n-1}, slik at matrise M_i har dimensjon $d_i \times d_{i+1}$.

La $m_{i,j}$ være kostnaden av en optimal parenterisering av M_i, M_{i+1}, …, M_j

Formelen (rekursiv definisjon) for $m_{i,j}$ vil være som følger:

$$m_{i,j} = \min_{i \le k < j} \{ m_{i,k} + m_{k,j} + d_i d_{k+1} d_{j+1} \}, \text{ for alle } 0 \le i < j \le n-1$$

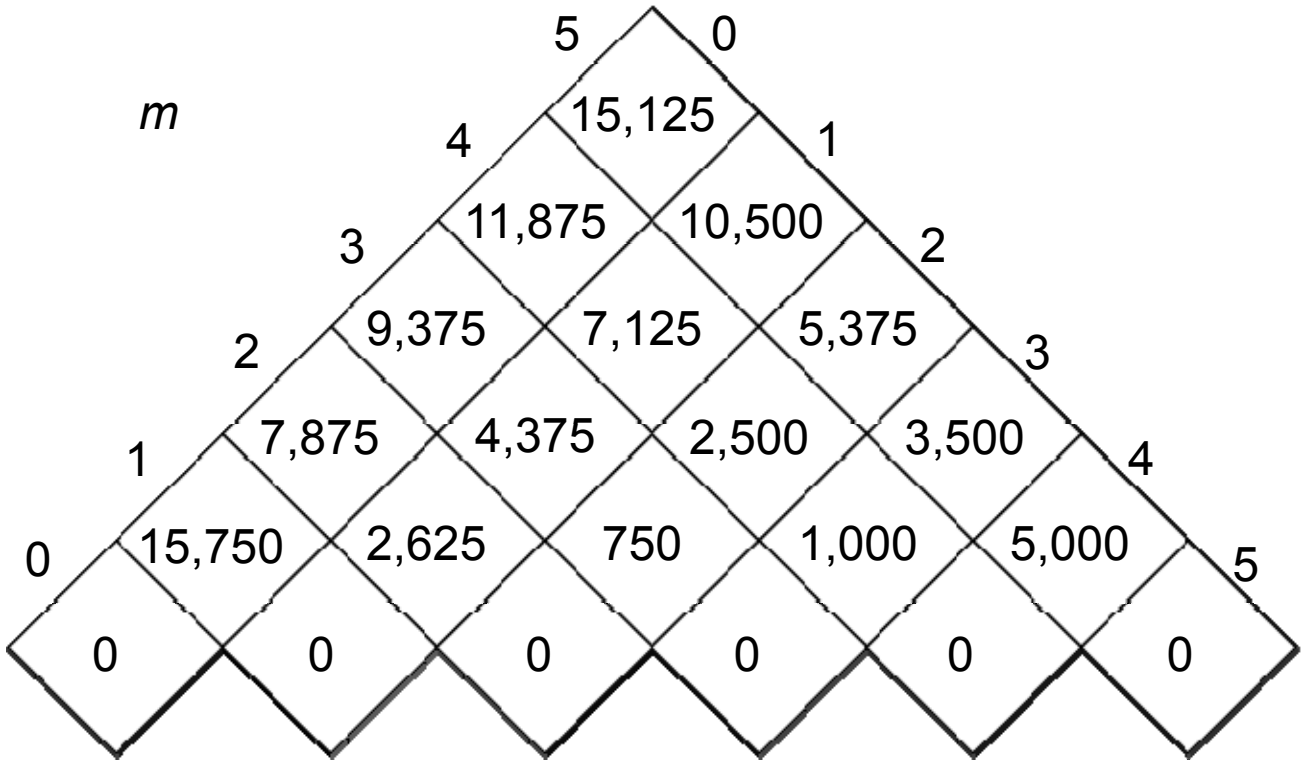
$$m_{i,i} = 0, \text{ for alle } 0 \le i \le n-1$$

Kostnaden av å utføre multiplikasjonen med den optimale parenteriseringen finner vi i $m_{0.n-1}$

3.

Optimal matrisemultiplikasjon

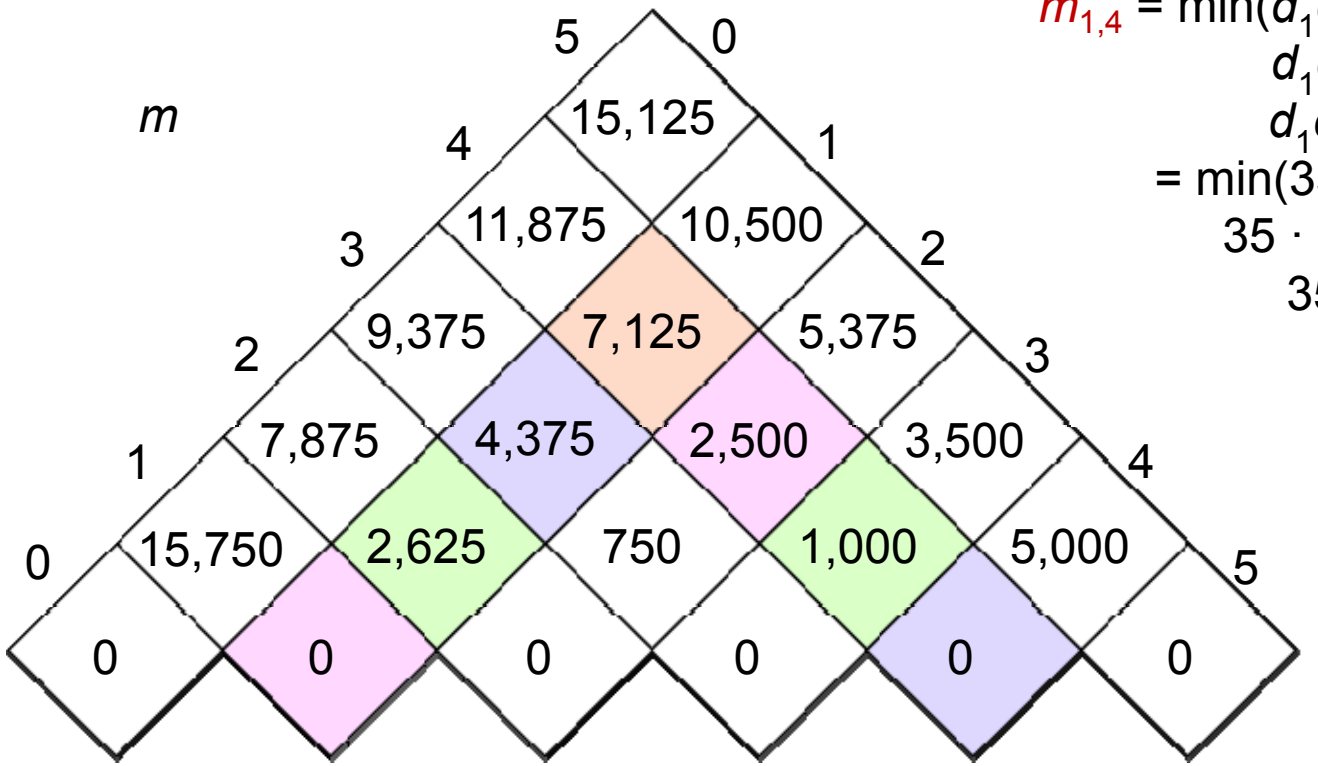
d	30	35	15	5	10	20	25



3.

Optimal matrisemultiplikasjon

d	30	35	15	5	10	20	25

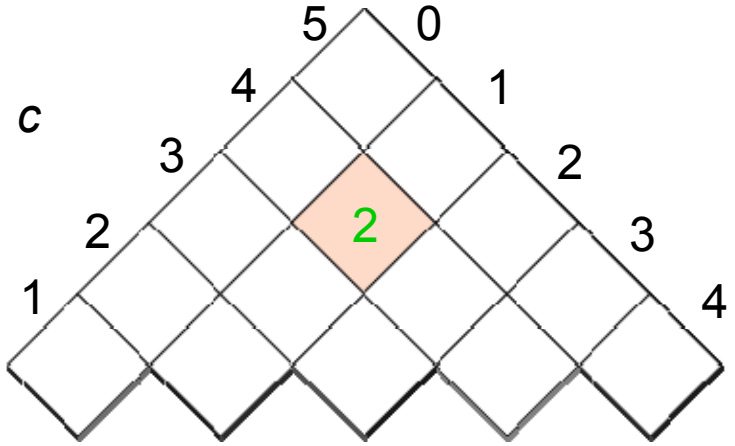


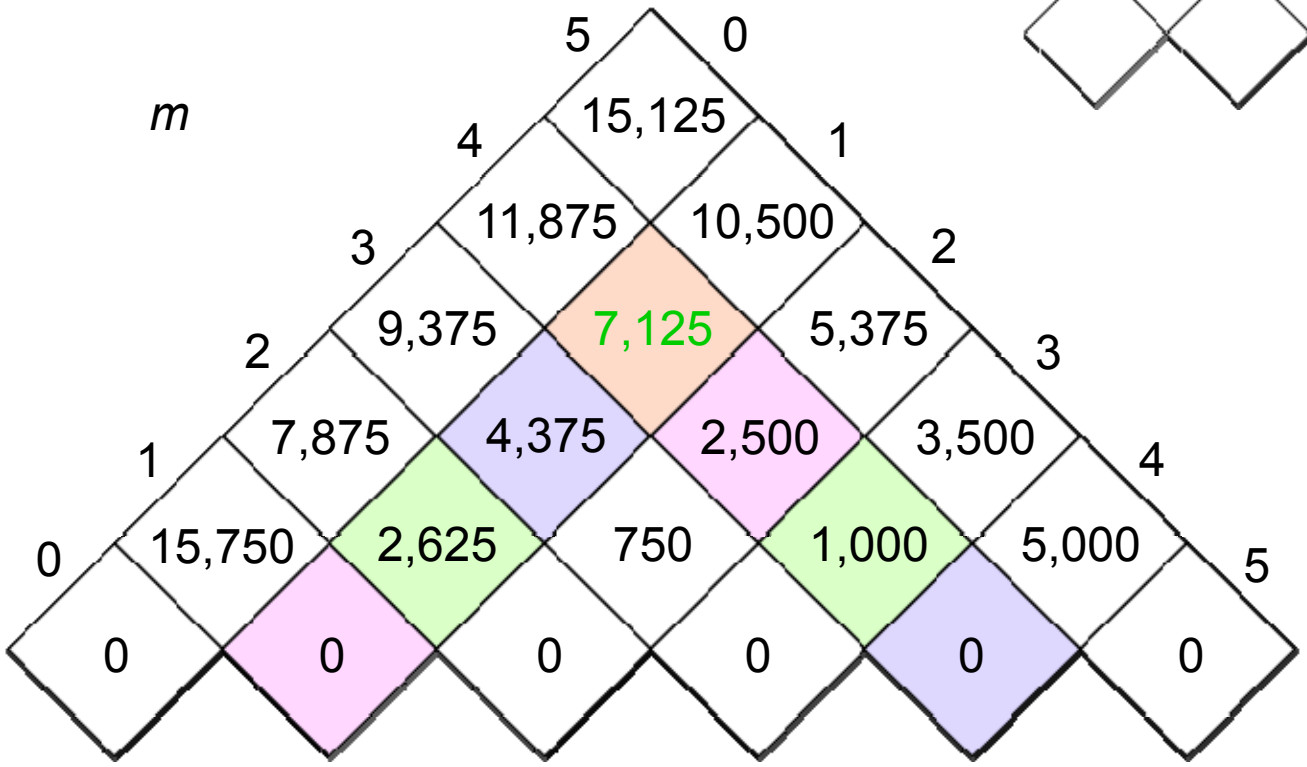
$$m_{1,4} = \min(d_1 d_2 d_5 + m(1,1) + m(2,4),$$
$$d_1 d_3 d_5 + m(1,2) + m(3,4),$$
$$d_1 d_4 d_5 + m(1,3) + m(4,4))$$
$$= \min(35 \cdot 15 \cdot 20 + 0 + 2{,}500,$$
$$35 \cdot 5 \cdot 20 + 2{,}625 + 1{,}000,$$
$$35 \cdot 10 \cdot 20 + 4{,}375 + 0)$$

3./4.

Optimal matrisemultiplikasjon

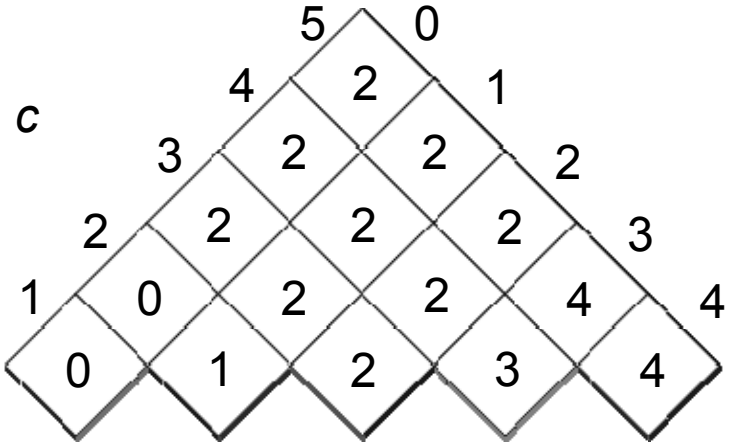
$M_1 \cdot M_2 \cdot M_3 \cdot M_4$
$= M_1 \cdot (M_2 \cdot M_3 \cdot M_4)$
$= (M_1 \cdot M_2) \cdot (M_3 \cdot M_4)$
$= (M_1 \cdot M_2 \cdot M_3) \cdot M_4$

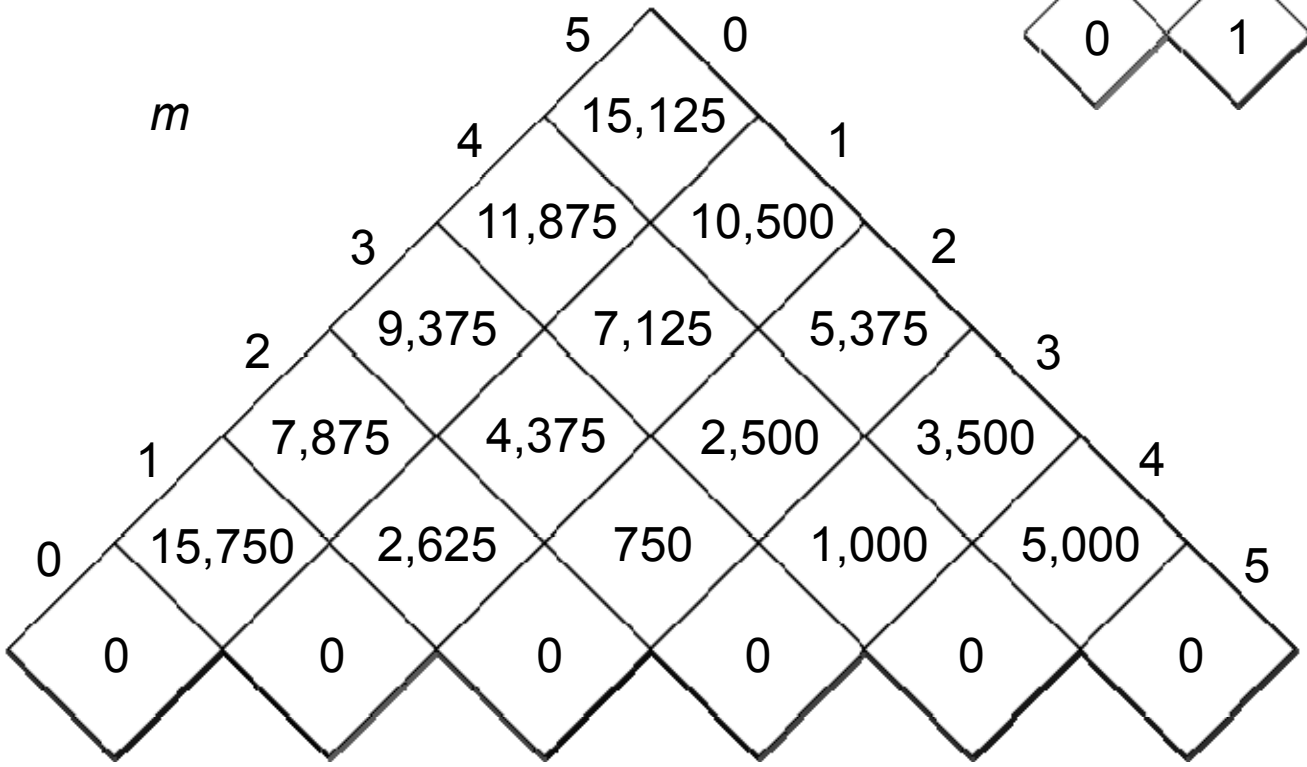




3./4.

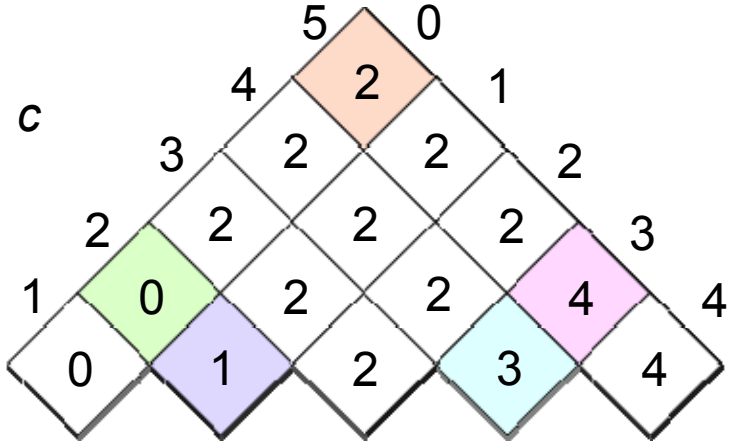
Optimal matrisemultiplikasjon





4.

Optimal matrisemultiplikasjon

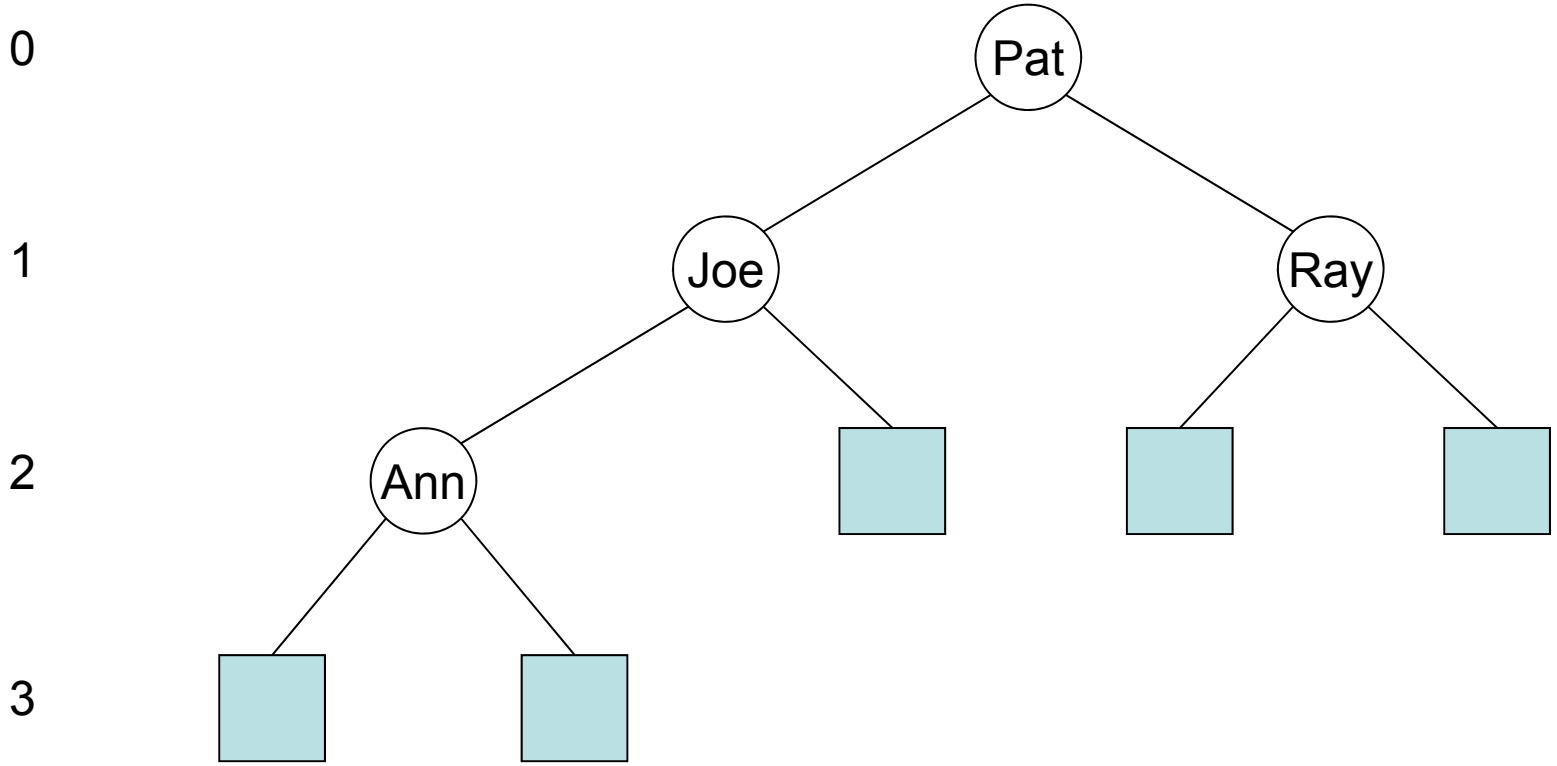


$$M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_4 \cdot M_5$$
$$= (M_0 \cdot M_1 \cdot M_2) \cdot (M_3 \cdot M_4 \cdot M_5)$$
$$= ((M_0) \cdot (M_1 \cdot M_2)) \cdot ((M_3 \cdot M_4) \cdot (M_5))$$

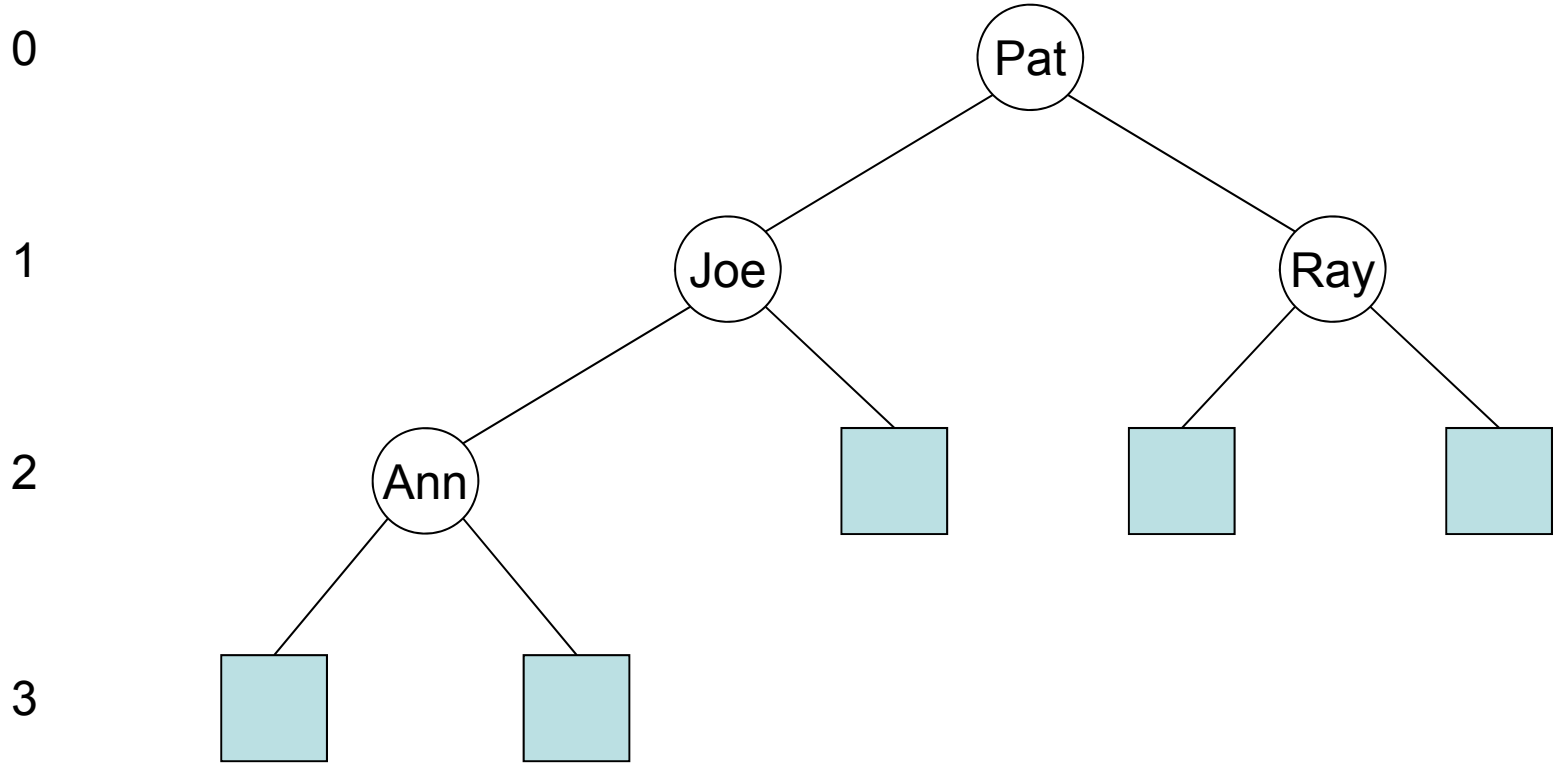
Optimal matrisemultiplikasjon

```
function OptimalParens( d[0 : n – 1] )
    for i ← 0 to n-1 do
        m[i, i] ← 0
    for diag ← 1 to n – 1 do  // hjelpevariabel for å fylle ut tabellen nivå for nivå
        for i ← 0 to n – 1 – diag do
            j ← i + diag
            m[i, j] ← ∞
            for k ← i to j – 1 do
                q ← m[i, k] + m[k + 1, j] + d[i] · d[k + 1] · d[j + 1]
                if q < m[i, j] then
                    m[i, j] ← q
                    c[i,j] ← k
                endif
    return m[0, n – 1]
end OptimalParens
```

Optimale søketrær

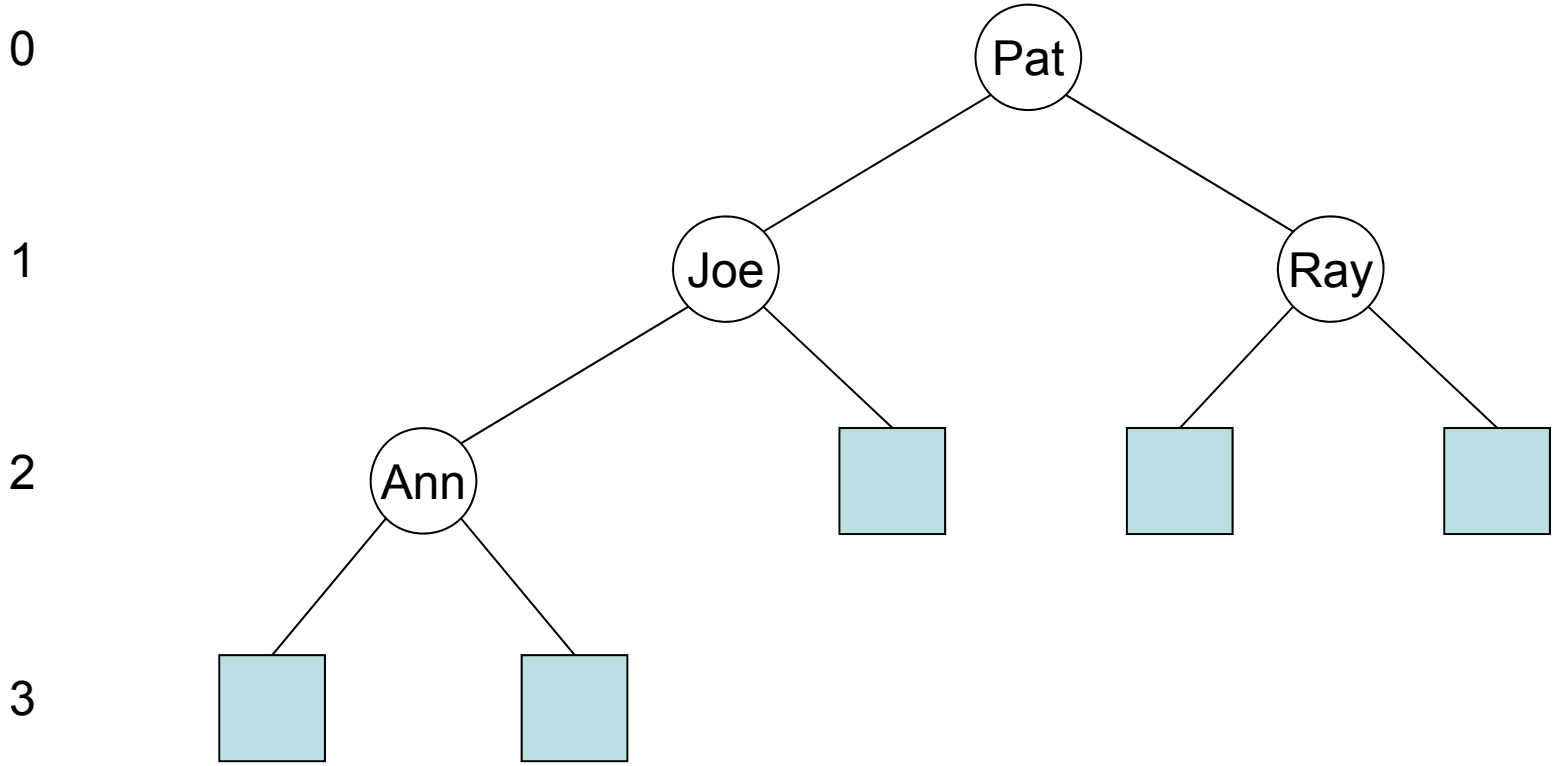


Optimale søketrær



	Ann		Joe		Pat		Ray	
	p_0		p_1		p_2		p_3	
q_0		q_1		q_2		q_3		q_4
3	3	3	2	2	1	2	2	2

Optimale søketrær



	Ann		Joe		Pat		Ray	
	p_0		p_1		p_2		p_3	
q_0		q_1		q_2		q_3		q_4
3	3	3	2	2	1	2	2	2

Gjennomsnittlig søketid: $3p_0 + 2p_1 + p_2 + 2p_3 + 3q_0 + 3q_1 + 2q_2 + 2q_3 + 2q_4$

Optimale søketrær

For generelle søketrær får vi følgende formel for gjennomsnittlig antall sammenlikninger som gjøres:

- T et tre med n nøkler (søkeord lagret i interne noder) K_0, …, K_{n-1}.
- $n+1$ bladnoder tilsvarer intervaller I_0, …, I_n mellom nøklene.
- sannsynlighetsvektorer **p** og **q** for nøklene og intervallene mellom dem.
- d_i er nivået til nøkkel K_i, e_i er nivået til bladnode som korresponderer med I_i.

$$A(T,n,\mathbf{p},\mathbf{q}) = \sum_{i=0}^{n-1} p_i(d_i+1) + \sum_{i=0}^{n} q_i e_i$$

Optimale søketrær

- Hvis p_i-ene er like og q_i-ene er like, vil det komplette binære søketreet være det optimale.

- Men noen ord kan være oftere søkt etter enn andre (p_i-ene ulike) , derfor kan det lønne seg med skjeve trær og deltrær, for å få ord som det ofte blir søkt etter så høyt som mulig opp i treet.

- Vi ønsker å finne det optimale binære søketreet T over alle mulige, gitt søkesannsynlighetene (p_i-ene og q_i-ene). Dvs treet som minimerer gjennomsnittlig antall sammenlikninger $A(T,n,\mathbf{p},\mathbf{q})$:

$$\min_T A(T,n,\mathbf{P},\mathbf{Q})$$

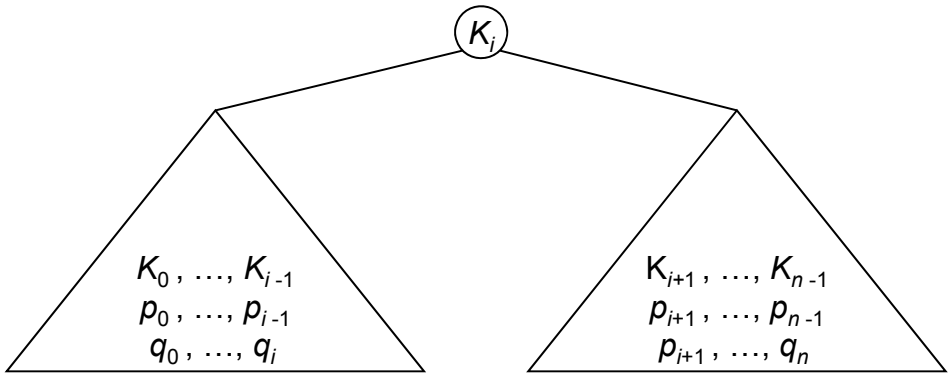
- p_i-ene og q_i-ene er i utgangspunktet sannsynligheter (tall i intervallet [0,1], som summerer til 1), men vi kan slakke på kravet og anta at de er positive reelle tall, det er uansett bare tall som sammenliknes.

La $\sigma(\mathbf{p},\mathbf{q}) = \sum_{i=0}^{n-1} p_i + \sum_{i=0}^{n} q_i$

1.

Optimale søketrær

Et binært tre T for nøklene K_0 , … K_{n-1} består av en rot med nøkkel K_i , og to deltrær L og R.



Vi må prøve alle mulige røtter K_i for å finne hvor det er best å dele treet.

Hver mulige rot gir opphav til to delproblemer – optimalisering av de korresponderende venstre og høyre deltrærne.

Gjennomsnittlig antall operasjoner for treet T er gitt ved:

$$A(T,n,\mathbf{p},\mathbf{q}) = A(L,i,p_0,\ldots,p_{i-1},q_0,\ldots,q_i) + A(R,n-i+1,p_{i+1},\ldots,p_{n-1},q_{i+1},\ldots,q_n) + \sigma(\mathbf{p},\mathbf{q})$$

For enkelthets skyld skriver vi A(T) = A(L) + A(R) + σ(**p**,**q**)

2.

Optimale søketrær

La T_{ij} være et søketre for nøklene $K_i, \ldots, K_j$, $0 \le i, j \le n - 1$.
(T_{ij} er det tomme treet om $j < i$.)

La $\sigma(i,j) = \sum_{k=i}^{j} p_k + \sum_{k=i}^{j+1} q_k$

Formelen (rekursiv definisjon) for $A(T_{i,j})$ vil være som følger:

$$A(T_{ij}) = \min_{i \le k \le j}\{A(T_{i,k-1}) + A(T_{k+1,j})\} + \sigma(i,j)$$

$$A(T_{ii}) = 0 + 0 + \sigma(i,i)$$

Kostnaden av det optimale treet finner vi i $A(T_{0,n-1})$.

Optimale søketrær

En algoritme kan nå relativt enkelt lages ved å fylle ut en tabell med verdiene for $A(T_{ij})$ som definert av formelen (samme måte som for matrisemultiplikasjon):

$$A(T_{ij}) = \min_{i \le k \le j}\{A(T_{i,k-1}) + A(T_{k+1,j})\} + \sigma(i,j)$$

$$A(T_{ii}) = 0 + 0 + \sigma(i,j)$$

I tillegg til verdien $A(T_{ij})$ må også en representasjon av de aktuelle trærne vedlikeholdes, slik at vi finner selve treet.

Fyller ut ulike typer tabeller

Bottom up