Dynamisk programmering

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

- Har ingen ting med *programmering* å gjøre.
- "*Dynamisk*" er et ord som kan aldri brukes negativt.

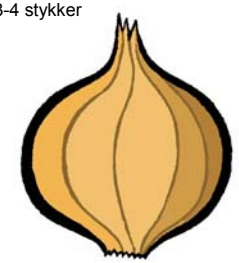
Hvilke problemer?

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 så lite delproblem at løsningen lett kan finnes.

Løsningene på delproblemene må så kunne kombineres til løsninger på større problemer. (Ofte må flere delproblemer kombineres, 2-3-4 stykker ikke uvanlig, flere er mulig.).

Dynamisk programmering brukes typisk til å løse optimaliserings-problemer. (Problemer hvor det kan være mange løsninger, og hvor vi ønsker å finne den beste.)



Optimalitetsprinsippet

Gitt et optimaliseringsproblem, og en funksjon *combine* (funksjonen 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, så er problemet egnet for løsning med dynamisk programmering. Da genereres bare optimale løsninger på delproblemer.

Overlappende delproblemer

For at dynamisk programmering skal være mer effektivt enn f.eks en rekursiv divide and conquer-algoritme, må delproblemene være overlappende.

Dvs. samme delproblem må forekomme som del i flere større problemer.

I dynamisk programmering utnytter vi det faktum at vi allerede har løst delproblemer og lagret svaret, slik at disse ikke løses flere ganger.

Hvis både optimalitetsprinsippet holder, og vi har overlappende delproblemer, så er sannsynligvis dynamisk programmering måten å løse problemet på.

Fire enkle(?) steg

1. Beskriv strukturen i en optimal løsning / strukturen i problemet.
2. Definer rekursivt verdien av en optimal løsning.
3. Beregn verdien til en optimal løsning, bottom up (lagr 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.)

Dynamisk programmering vs. Divide and conquer

Dynamisk programmering	Divide and conquer
Bottom up	Top down (rekursivt kall)
Memoisering	
Best egnet når delproblemene er avhengige av hverandre. Løsninger på alle delproblemer lagres i en tabell og behøver ikke re-beregnes.	Best egnet når delproblemene er uavhengige av hverandre. Da re-beregnes ikke løsninger på delproblemer.
Irrelevante delproblemer løses	Kun relevante delproblemer løses.
	Quicksort

Korteste sti (alle til alle) – Floyd

Vi skal finne korteste sti mellom alle par av noder i en graf G=(V,E). Kantene har lengde, men vi antar at det ikke finnes negative sykler.

1. En sti $p = \langle i, v_1, \ldots, v_k, j \rangle$ fra node i til node j består av endenodene i og j, og internnodene $v_1, \ldots, v_k$.

   For alle par av noder i, j, ser vi på korteste stier fra i til j som består av internnoder fra mengden $\{v_1, \ldots, v_k\}$, og lar p være en av disse.

   – Hvis v_k er en internnode i p, består p av delstiene p_1 (fra i til v_k) og p_2 (fra v_k til j). Delstiene p_1 og p_2 er stier med internnoder fra mengden $\{v_1, \ldots, v_{k-1}\}$; og p_1 og p_2 er korteste stier fra i til v_k og fra v_k til j, ellers ville ikke p vært en korteste sti fra i til j.

   – Hvis k ikke er en internnode i p, er p en korteste sti fra i til j med internnoder fra mengden $\{v_1, \ldots, v_{k-1}\}$.

   (Optimalitetsprinsippet holder.)

Korteste sti (alle til alle) – Floyd

2. La $d_{ij}^{(k)}$ være lengden av korteste sti fra i til j med internnoder fra mengden $\{v_1, \ldots, v_k\}$.

   Når $k = 0$ kan ikke stien bestå av internnoder i det hele tatt og $d_{ij}^{(0)}$ vil altså tilsvare lengden av kanten ij. (Vi bruker w_{ij} for lengden av kanten ij.)

   En rekursiv definisjon av $d_{ij}^{(k)}$ vil være som følger:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{hvis } k = 0 \\ \min\{ \underbrace{d_{ij}^{(k-1)}}_{v_k \text{ ikke med}}, \underbrace{d_{iv_k}^{(k-1)} + d_{v_k j}^{(k-1)}}_{v_k \text{ med i stien}} \} & \text{hvis } k \geq 1 \end{cases}$$

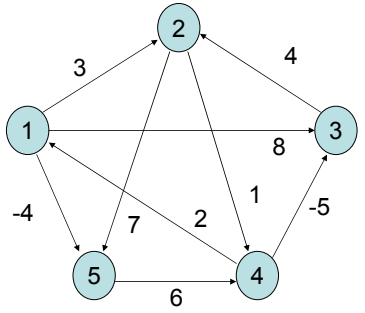
   Matrisen $D^{(n)} = (d_{ij}^{(n)})$, hvor n er antall noder i grafen, vil inneholde lengden av de korteste stiene mellom alle par av noder i, j.
   (Alle noder er nå lovlige mellomnoder.)

Korteste sti (alle til alle) – Floyd

3.

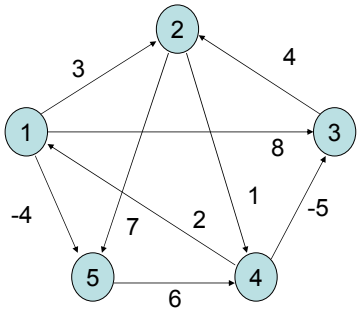
```
function FloydWarshall ( W )
    n ← rows( W )
    D(0) = W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                dij(k) ← min( dij(k-1),  dik(k-1) + dik(k-1) )
            endfor
        endfor
    endfor
    return(D(n))
end NaivStringMatcher
```

Korteste sti (alle til alle) – Floyd



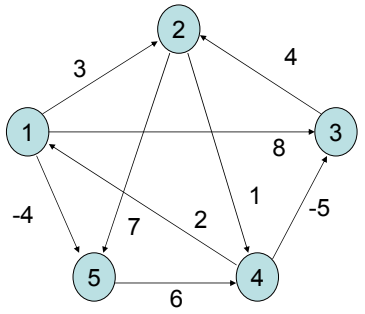
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Korteste sti (alle til alle) – Floyd



$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

Korteste sti (alle til alle) – Floyd



$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Strenger som ligner

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 *P* og *T* er det minste antall slike operasjoner som trengs for å konvertere *P* til *T*.

Eks.

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

Strenger som ligner

Vi skal finne edit distance mellom to strenger *T* og *P*.

1. La $D[n, m] = ED(P[1{:}n], T[1{:}m])$. (Edit distance mellom delstrenger)

   – Hvis $P[i] = T[j]$, så er $D[i, j] = D[i\text{-}1, j\text{-}1]$.

   – Hvis $P[i] \neq T[j]$, så må vi se på en optimal sekvens av operasjoner som transformerer $T[1{:}j]$ til $P[1{:}i]$. Vi antar UTAG at vi først har endret på $P[1{:}i\text{-}1]$, $T[1{:}j\text{-}1]$ og at vi gjør siste operasjon for å få $T[j] = P[i]$, dette kan være:
   - Substitusjon i T – sette $T[j] = P[i]$.
   - Tillegg i T – legge til symbol i *T* på pos $T[j]$ – sletting av P $[i]$.
   - Sletting i T – sletting symbol i *T* på pos $T[j]$.

   Hvis ikke operasjonene gjort for å få $T[1{:}j\text{-}1] = P[1{:}i\text{-}1]$ var optimale, så snakker vi ikke nå om en optimal sekvens av operasjoner som gir $T[1{:}j] = P[1{:}i]$.

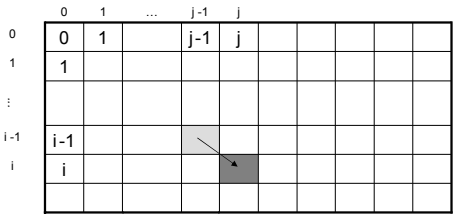
   (Optimalitetsprinsippet holder)

Strenger som ligner

2. En rekursiv definisjon av $D[i, j]$ vil altså være som følger:

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

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

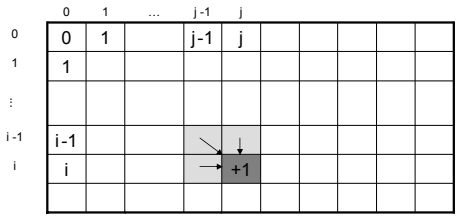


Strenger som ligner

2. En rekursiv definisjon av $D[i, j]$ vil altså være som følger:

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

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

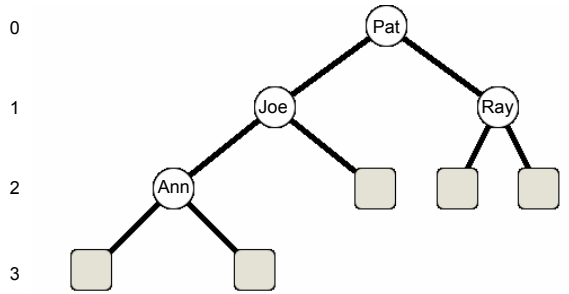


Strenger som ligner

3.

```
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 ]
            endif
                D[ i, j ] ← min {  D[i -1, j - 1] +1,   D[i -1, j] +1,   D[i, j - 1] +1  }
        endfor
    endfor
    return( D[ n, m ] )
end EditDistance
```

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) $K_0, \ldots, K_{n-1}$.
  - n+1 bladnoder tilsvarer intervaller $I_0, \ldots, I_n$ mellom nøklene.
  - sannsynlighetsvektorer **p** og **q** for nøklene og intervallene mellom dem, hhv.
  - 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 $A(T,n,\mathbf{p},\mathbf{q})$.

  $\min_T A(T,n,\mathbf{p},\mathbf{q})$.

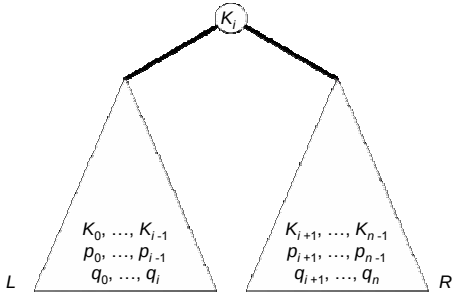
- p_i-ene og q_i-ene er i utgangspunktet sansynligheter (tall i intervallet [0,1], som summerer til 1), men vi kan slakke på kravet og anta at de er positive reele tall, det er uansett bare tall som sammenliknes.

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

Optimale søketrær

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

   L er del binært tre for nøklene K_0 , … K_{i-1} , R et tre for nøklene K_{i+1} , … K_n .



Optimale søketrær

Gjennomsnittlig antall operasjoner for et søketre T er gitt ved følgende formel:

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

For enkelthets skyld skriver vi: $A(T) = A(L) + A(R) + \sigma(\mathbf{p},\mathbf{q})$

(Optimalitetsprinsippet holder)

Optimale søketrær

1. En rekursiv formel kan lages som følger:

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

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

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

Optimale søketrær

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

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

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