INF2220 - Algoritmer og datastrukturer

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 11:
Dynamisk programmering

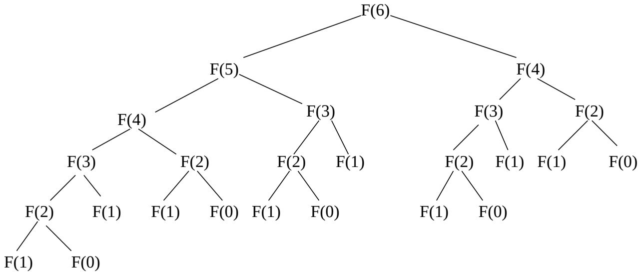
Dagens plan

- Dynamisk programmering
  - Floyds algoritme for korteste vei alle-til-alle
- Paradigmer for algoritmedesign

Dynamisk Programmering

$$F_n = F_{n-1} + F_{n-2}$$

```
int fib_r(int n)
{
    if (n<=1)
        return 1;
    else
        return fib_r(n-1) + fib_r(n-2);
}
```



Fibonacci Numbers by Dynamic Programming

```
int fib_dp(int n)
{
    int i;
    int [] f = new int [n+1]
    f[0] = 0;
    f[1] = 1;

    for (int i = 2; i <= n; i++) {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];

}
```

Dynamisk programmering

- Brukes først og fremst når vi ønsker optimale løsninger
- Må kunne dele det globale problemet i *delproblemer*
  - Disse løses typisk ikke-rekursivt ved å lagre del-løsningene i en tabell
- En optimal løsning på det globale problemet må være en sammensetning av optimale løsninger på (noen av) delproblemene

- Vi skal se på ett eksempel:
  Floyds algoritme for å finne korteste vei alle-til-alle i en rettet graf

Dynamisk programmering

- Brukes først og fremst når vi ønsker optimale løsninger
- Må kunne dele det globale problemet i *delproblemer*
  - Disse løses typisk ikke-rekursivt ved å lagre del-løsningene i en tabell
- En optimal løsning på det globale problemet må være en sammensetning av optimale løsninger på (noen av) delproblemene
- Vi skal se på ett eksempel:
  Floyds algoritme for å finne korteste vei alle-til-alle i en rettet graf

Korteste vei alle-til-alle (Floyd)

Vi ønsker å beregne den korteste veien mellom ethvert par av noder i en *rettet, vektet* graf

Grunnleggende idé

Hvis det går en vei fra node i til node k med lengde ik, og en vei fra node k til node j med lengde kj, så går det en vei fra node i til node j med lengde $ik + kj$

- Floyds algoritme:
  Denne betraktningen gjentas systematisk for alle tripler i, k og j:
  - Initielt: Avstanden fra node i til node j settes lik vekten på kanten fra i til j, uendelig hvis det ikke går noen kant fra i til j
  - Trinn 0: Se etter forbedringer ved å velge node 0 som mellomnode
  - Etter trinn k: Avstanden mellom to noder er den korteste veien som bare bruker nodene $0, 1, \ldots, k$ som mellomnoder

Korteste vei alle-til-alle (Floyd)

Vi ønsker å beregne den korteste veien mellom ethvert par av noder i en *rettet, vektet* graf

Grunnleggende idé

Hvis det går en vei fra node $\boldsymbol{i}$ til node $\boldsymbol{k}$ med lengde $\boldsymbol{ik}$, og en vei fra node $\boldsymbol{k}$ til node $\boldsymbol{j}$ med lengde $\boldsymbol{kj}$, så går det en vei fra node $\boldsymbol{i}$ til node $\boldsymbol{j}$ med lengde $\boldsymbol{ik} + \boldsymbol{kj}$

- Floyds algoritme:
  Denne betraktningen gjentas systematisk for alle tripler $\boldsymbol{i}$, $\boldsymbol{k}$ og $\boldsymbol{j}$:
  - Initielt: Avstanden fra node $\boldsymbol{i}$ til node $\boldsymbol{j}$ settes lik vekten på kanten fra $\boldsymbol{i}$ til $\boldsymbol{j}$, uendelig hvis det ikke går noen kant fra $\boldsymbol{i}$ til $\boldsymbol{j}$
  - Trinn 0: Se etter forbedringer ved å velge node 0 som mellomnode
  - Etter trinn $\boldsymbol{k}$: Avstanden mellom to noder er den korteste veien som bare bruker nodene 0, 1, . . . , $\boldsymbol{k}$ som mellomnoder

Korteste vei alle-til-alle (Floyd)

Vi ønsker å beregne den korteste veien mellom ethvert par av noder i en *rettet, vektet* graf

Grunnleggende idé

Hvis det går en vei fra node $\boldsymbol{i}$ til node $\boldsymbol{k}$ med lengde $\boldsymbol{ik}$, og en vei fra node $\boldsymbol{k}$ til node $\boldsymbol{j}$ med lengde $\boldsymbol{kj}$, så går det en vei fra node $\boldsymbol{i}$ til node $\boldsymbol{j}$ med lengde $\boldsymbol{ik} + \boldsymbol{kj}$

- Floyds algoritme:
  Denne betraktningen gjentas systematisk for alle tripler $\boldsymbol{i}$, $\boldsymbol{k}$ og $\boldsymbol{j}$:
  - Initielt: Avstanden fra node $\boldsymbol{i}$ til node $\boldsymbol{j}$ settes lik vekten på kanten fra $\boldsymbol{i}$ til $\boldsymbol{j}$, uendelig hvis det ikke går noen kant fra $\boldsymbol{i}$ til $\boldsymbol{j}$
  - Trinn 0: Se etter forbedringer ved å velge node 0 som mellomnode
  - Etter trinn $\boldsymbol{k}$: Avstanden mellom to noder er den korteste veien som bare bruker nodene $0, 1, \ldots, \boldsymbol{k}$ som mellomnoder

Optimal substruktur

$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

P = korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

Optimal substruktur

$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

P = korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

Optimal substruktur

$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

P $=$ korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

Optimal substruktur

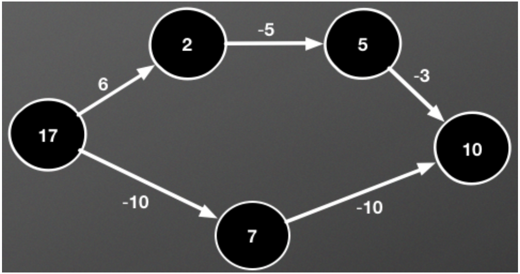
$V = \{1, 2, ..., n\}$
$V^{(k)} = \{1, 2, ..., k\}$ representerer de første k nodene

Fastsette en startnode (source) $i \in V$, en destinasjonsnode $j \in V$ og en $k \in \{1, 2, ..., n\}$.

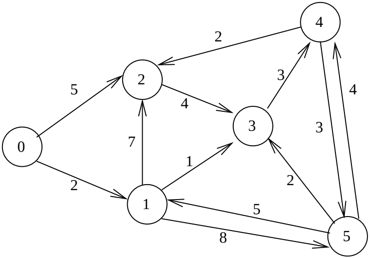
P = korteste $i - j$ vei der alle indre nodene er i $V^{(k)}$

k begrenser hvilke noder vi kan velge for en gitt sub-problem.

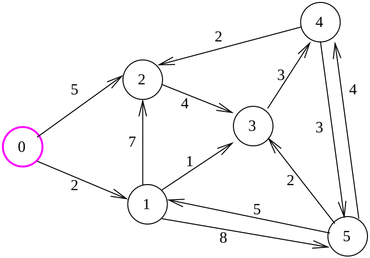
[i = 17, j=10, k=5]



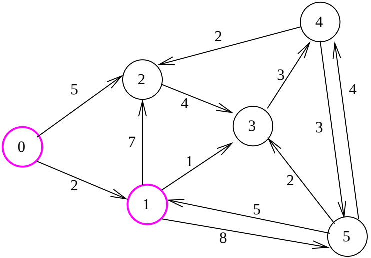
Hva er P?



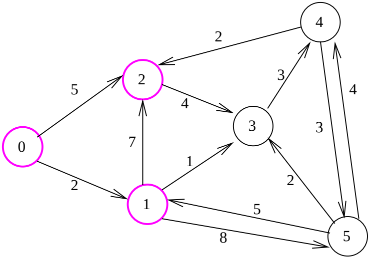
	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0



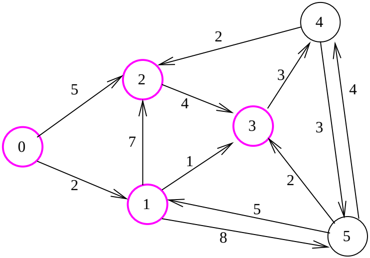
	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0



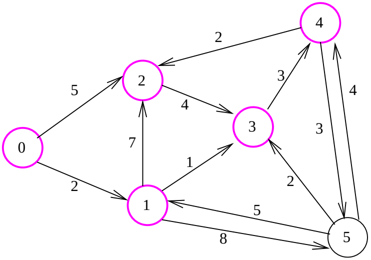
	0	1	2	3	4	5
0	0	2	5	3_1	∞	10_1
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	12_1	2	4	0



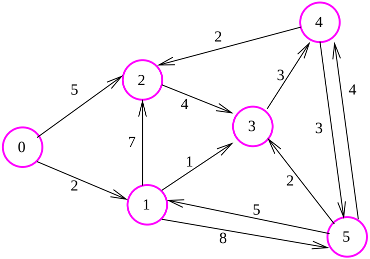
	0	1	2	3	4	5
0	0	2	5	3_1	∞	10_1
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	6_2	0	3
5	∞	5	12_1	2	4	0



	0	1	2	3	4	5
0	0	2	5	3_1	6_3	10_1
1	∞	0	7	1	4_3	8
2	∞	∞	0	4	7_3	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	6_2	0	3
5	∞	5	12_1	2	4	0



	0	1	2	3	4	5
0	0	2	5	3_1	6_3	9_4
1	∞	0	6_4	1	4_3	7_4
2	∞	∞	0	4	7_3	10_4
3	∞	∞	5_4	0	3	6_4
4	∞	∞	2	6_2	0	3
5	∞	5	6_4	2	4	0

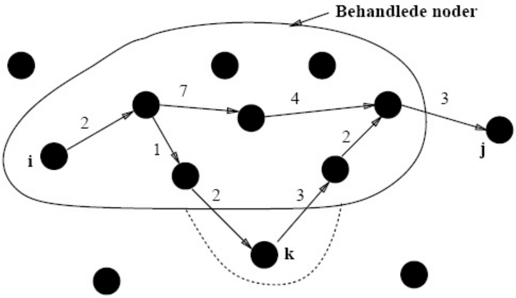


	0	1	2	3	4	5
0	0	2	5	3_1	6_3	9_4
1	∞	0	6_4	1	4_3	7_4
2	∞	15_5	0	4	7_3	10_4
3	∞	11_5	5_4	0	3	6_4
4	∞	8_5	2	5_5	0	3
5	∞	5	6_4	2	4	0

Hvorfor virker Floyd?

Floyd-invarianten:

avstand[*i*][*j*] vil være lik lengden av den korteste veien fra node ***i*** til node ***j*** som har alle sine indre noder behandlet



FØR:	ETTER:
A(i,j)=16	A(i,j)=13
A(i,k)=5	....
A(k,j)=8	....

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige. Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige. Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   $P_1 =$ korteste $i - k$ vei der alle interne noder er i $V^{(k-1)}$.
   $P_2 =$ korteste $k - j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k - 1$ er ferdige. Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i, j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i-k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k-j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k-1$ er ferdige. Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i,j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i - k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k - j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k - 1$ er ferdige. Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i, j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

Hva vet vi om P?

1. Hvis node k ikke er en indre node i P:
   P er korteste vei der alle interne noder er i $V^{(k-1)}$.
2. Hvis node k er en indre node i P:
   P_1= korteste $i - k$ vei der alle interne noder er i $V^{(k-1)}$.
   P_2= korteste $k - j$ vei der alle interne noder er i $V^{(k-1)}$.

Når vi gjør beregninger for k, vet vi at alle beregninger for $k - 1$ er ferdige. Trenger array av 3 dimensjoner (startnode, dest. node, prefix):

A[i,j,0] = 0 hvis i=j
A[i,j,0] = C_{ij} hvis (i, j) er en kant i grafen
A[i,j,0] = ∞ ellers

A[i,j,k] = min(A[i,j,k-1], A[i,k,k-1]+ A[k,j,k-1])

```java
public static void kortesteVeiAlleTilAlle(
  int[ ][ ] nabo, int[ ][ ] avstand, int[ ][ ] vei) {

  int n = avstand.length; // Forutsetning: arrayene er
                          // kvadratiske med samme dimensjon
  // Initialisering:
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      avstand[i][j] = nabo[i][j];
      vei[i][j] = -1;                // Ingen vei foreløpig
    }
  }

  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
          // Kortere vei fra i til j funnet via k
          avstand[i][j] = avstand[i][k] + avstand[k][j];
          vei[i][j] = k;
}}}}}
```

Tidsforbruket er åpenbart $\mathcal{O}(n^3)$

```java
public static void kortesteVeiAlleTilAlle(
  int[][] nabo, int[][] avstand, int[][] vei) {

  int n = avstand.length; // Forutsetning: arrayene er
                          // kvadratiske med samme dimensjon
  // Initialisering:
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      avstand[i][j] = nabo[i][j];
      vei[i][j] = -1;                 // Ingen vei foreløpig
    }
  }

  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
          // Kortere vei fra i til j funnet via k
          avstand[i][j] = avstand[i][k] + avstand[k][j];
          vei[i][j] = k;
}}}}}
```

Tidsforbruket er åpenbart $\mathcal{O}(n^3)$

Hvordan tolke resultatet av Floyd

- Ved start inneholder **nabo[*i*][*j*]** lengden av kanten fra ***i*** til ***j***, ∞ hvis det ikke er noen kant
- Floyd lar **nabo** være uendret og legger resultatet i **avstand** og **vei**
- **avstand[*i*][*j*]** er lengden av korteste vei fra ***i*** til ***j***, ∞ hvis det ikke er noen vei
- Når vi oppdager at den korteste veien fra ***i*** til ***j*** passerer gjennom en mellomnode ***k***, setter vi **vei[*i*][*j*]** $= \boldsymbol{k}$
- **vei** sier hva som er den korteste veien
  - $\boldsymbol{k_1} =$ **vei[*i*][*j*]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til ***j***
  - $\boldsymbol{k_2} =$ **vei[*i*][*k_1*]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til $\boldsymbol{k_1}$ osv.
  - Når **vei[*i*][*k_m*]** $= -1$, er $(\boldsymbol{i}, \boldsymbol{k_m})$ den første kanten i korteste vei fra ***i*** til ***j***

Hvordan tolke resultatet av Floyd

- Ved start inneholder **nabo[*i*][*j*]** lengden av kanten fra ***i*** til ***j***, ∞ hvis det ikke er noen kant
- Floyd lar **nabo** være uendret og legger resultatet i **avstand** og **vei**
- **avstand[*i*][*j*]** er lengden av korteste vei fra ***i*** til ***j***, ∞ hvis det ikke er noen vei
- Når vi oppdager at den korteste veien fra ***i*** til ***j*** passerer gjennom en mellomnode ***k***, setter vi **vei[*i*][*j*] = *k***
- **vei** sier hva som er den korteste veien
  - $k_1 =$ **vei[*i*][*j*]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til ***j***
  - $k_2 =$ **vei[*i*][k_1]** er den største verdi av ***k*** slik at ***k*** ligger på den korteste veien fra ***i*** til k_1 osv.
  - Når **vei[*i*][k_m]** $= -1$, er (i, k_m) den første kanten i korteste vei fra ***i*** til ***j***

Hvordan tolke resultatet av Floyd

- Ved start inneholder **nabo[*i*][*j*]** lengden av kanten fra $\boldsymbol{i}$ til $\boldsymbol{j}$, ∞ hvis det ikke er noen kant
- Floyd lar **nabo** være uendret og legger resultatet i **avstand** og **vei**
- **avstand[*i*][*j*]** er lengden av korteste vei fra $\boldsymbol{i}$ til $\boldsymbol{j}$, ∞ hvis det ikke er noen vei
- Når vi oppdager at den korteste veien fra $\boldsymbol{i}$ til $\boldsymbol{j}$ passerer gjennom en mellomnode $\boldsymbol{k}$, setter vi $\textbf{vei}[\boldsymbol{i}][\boldsymbol{j}] = \boldsymbol{k}$
- **vei** sier hva som er den korteste veien
  - $\boldsymbol{k_1} = \textbf{vei}[\boldsymbol{i}][\boldsymbol{j}]$ er den største verdi av $\boldsymbol{k}$ slik at $\boldsymbol{k}$ ligger på den korteste veien fra $\boldsymbol{i}$ til $\boldsymbol{j}$
  - $\boldsymbol{k_2} = \textbf{vei}[\boldsymbol{i}][\boldsymbol{k_1}]$ er den største verdi av $\boldsymbol{k}$ slik at $\boldsymbol{k}$ ligger på den korteste veien fra $\boldsymbol{i}$ til $\boldsymbol{k_1}$ osv.
  - Når $\textbf{vei}[\boldsymbol{i}][\boldsymbol{k_m}] = \boldsymbol{-1}$, er $(\boldsymbol{i}, \boldsymbol{k_m})$ den første kanten i korteste vei fra $\boldsymbol{i}$ til $\boldsymbol{j}$

Den korteste veien fra i til j

- Hvis $\mathbf{vei[}\boldsymbol{i}\mathbf{][}\boldsymbol{j}\mathbf{]} = \mathbf{-1}$, passerer ikke den korteste veien gjennom noen mellomnoder og den korteste veien er $(\boldsymbol{i}, \boldsymbol{j})$
- Ellers, vi rekursivt beregne den korteste veien fra $\boldsymbol{i}$ til $\mathbf{vei[}\boldsymbol{i}\mathbf{][}\boldsymbol{j}\mathbf{]}$ og den korteste veien fra $\mathbf{vei[}\boldsymbol{i}\mathbf{][}\boldsymbol{j}\mathbf{]}$ til $\boldsymbol{j}$

```
Kortestevei(i,j){
  if (vei[i][j] == -1) //en kant
      output (i,j);
  else {
      kortestevei(i, vei[i][j]);
      kortestevei(vei[i][j], j);
  }
}
```

Hva gjør Floyd dynamisk?

Hovedløkken i Floyd ser slik ut:

```
for (int k = 0; k < n; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
         // Kortere vei fra i til j funnet via k
         avstand[i][j] = avstand[i][k] + avstand[k][j];
         vei[i][j] = k;
}}}}
```

- Algoritmeinvarianten forutsetter at ***i***– og ***j***–løkken fullføres før ***k***–verdien økes
- If-testen sikrer at for en gitt ***k*** kan hverken **avstand[*i*][*k*]** eller **avstand[*k*][*j*]** bli endret i ***i***– og ***j***–løkken
- Dermed er ***i***– og ***j***–løkkene uavhengige av hverandre og kan paralleliseres (de er uavhengige delproblemer)

Hva gjør Floyd dynamisk?

Hovedløkken i Floyd ser slik ut:

```
for (int k = 0; k < n; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
          // Kortere vei fra i til j funnet via k
          avstand[i][j] = avstand[i][k] + avstand[k][j];
          vei[i][j] = k;
}}}}
```

- Algoritmeinvarianten forutsetter at ***i***– og ***j***–løkken fullføres før ***k***–verdien økes
- If-testen sikrer at for en gitt ***k*** kan hverken **avstand[*i*][*k*]** eller **avstand[*k*][*j*]** bli endret i ***i***– og ***j***–løkken
- Dermed er ***i***– og ***j***–løkkene uavhengige av hverandre og kan parallelliseres (de er uavhengige delproblemer)

Hva gjør Floyd dynamisk?

Hovedløkken i Floyd ser slik ut:

```
for (int k = 0; k < n; k++) {
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
        // Kortere vei fra i til j funnet via k
        avstand[i][j] = avstand[i][k] + avstand[k][j];
        vei[i][j] = k;
}}}}
```

- Algoritmeinvarianten forutsetter at ***i***– og ***j***–løkken fullføres før ***k***–verdien økes
- If-testen sikrer at for en gitt ***k*** kan hverken **avstand[*i*][*k*]** eller **avstand[*k*][*j*]** bli endret i ***i***– og ***j***–løkken
- Dermed er ***i***– og ***j***–løkkene uavhengige av hverandre og kan parallelliseres (de er uavhengige delproblemer)

Paradigmer for algoritmedesign

Paradigmer for algoritmedesign

Her følger en oppsummering av tre viktige paradigmer for algoritmedesign som vi gjør bruk av i dette kurset:

- Splitt og hersk
- Grådige algoritmer
- Dynamisk programmering

Splitt og hersk

- Dette er en generell metode som bruker rekursjon til å designe effektive algoritmer
- Den går ut på å dele et gitt problem opp i mindre delproblemer, og så rekursivt løse hvert delproblem
- Rekursjonen stoppes når problemet er så lite at løsningen er triviell
- Til slutt settes løsningen av delproblemene sammen til en løsning av det opprinnelige problemet

Eksempler:

- Binærsøking
- Quick-sort
- Merge-sort

Grådige algoritmer

- Dette er en metode som brukes på optimaliseringsproblemer
- Den går ut på å løse problemet ved å foreta en rekke valg
- I hvert trinn gjør vi det valget som i øyeblikket ser ut til å bringe oss nærmest mulig løsningen
- **Merk**: Metoden virker ikke alltid
- Vi sier at problemer metoden virker på, har "grådighetsegenskapen":
  - En rekke lokale optimaliseringer vil føre til et globalt optimum

Eksempler:

- Dijkstras algoritme
- Prims algoritme
- Kruskals algoritme
- Huffman-koding

Dynamisk programmering

- Dette er en annen metode som brukes på optimaliseringsproblemer
- Metoden er noe vanskeligere å forstå enn Splitt og hersk og Grådige algoritmer
- Metoden bør brukes på problemer som ser ut til å trenge eksponensiell eksekveringstid
- Dynamisk programmering gir alltid algoritmer som er polynomiske i tid, og disse algoritmene er vanligvis enkle å programmere
- For at metoden skal virke, må problemet ha en viss struktur som vi kan utnytte for å oppnå denne enkle løsningen

Litt terminologi knyttet til dynamisk programmering:

- Enkle delproblemer:
  Det må finnes en måte å dele problemet opp i delproblemer som er enkle å beskrive
- Delproblem-optimalisering:
  En optimal løsning på det globale problemet må kunne settes sammen av optimale løsninger på delproblemene
- Overlapp av delproblemer:
  Optimale løsninger av urelaterte problemer kan inneholde felles delproblemer

Eksempler:

- Floyds algoritme
- Beregning av lange matriseprodukter
- Lengste felles delsekvens

Gitt en 'line graph' G=(V,E) med vekt på nodene:

Problem: Å finne *IS* med maximal totalvekt.

Gitt en 'line graph' G=(V,E) med vekt på nodene:

Problem: Å finne *IS* med maximal totalvekt.

Gitt en 'line graph' G=(V,E) med vekt på nodene:

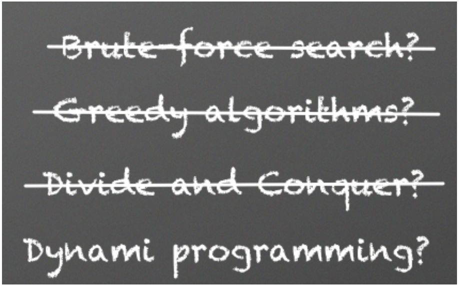
Problem: Å finne *IS* med maximal totalvekt.

Gitt en 'line graph' G=(V,E) med vekt på nodene:

Problem: Å finne *IS* med maximal totalvekt.

Gitt en 'line graph' G=(V,E) med vekt på nodene:

Problem: Å finne *IS* med maximal totalvekt.



DP

Første steg:

Hvordan ser den optimale løsningen ut? Tenk på strukturen av den optimale løsningen (i form av optimale sub-løsninger).

La S være løsningen. Dvs max-wt IS av G og V_n er den siste noden i G

- Case 1: $V_n \notin S$
  - S er max-wt IS av G', der G' er G uten V_n
- Case 2: $V_n \in S$
  - V_{n-1} kan ikke være i S
  - $S - \{V_n\}$ er max-wt IS av G", der G" er G uten V_{n-1} og V_n

Ide: Prøv begge mulighetene og returnerer den beste

Forslag algo. 1:

- rekursivt beregner
  - S_1= max-wt IS av G'
  - S_2= max-wt IS av G"
- returnerer S_1 eller $S_2 \cup \{V_n\}$

Hmmmm.....fornøyd?

La S være løsningen. Dvs max-wt IS av G og V_n er den siste noden i G

- Case 1: $V_n \notin S$
  - S er max-wt IS av G', der G' er G uten V_n
- Case 2: $V_n \in S$
  - V_{n-1} kan ikke være i S
  - $S - \{V_n\}$ er max-wt IS av G", der G" er G uten V_{n-1} og V_n

Ide: Prøv begge mulighetene og returnerer den beste

Forslag algo. 1:

- rekursivt beregner
  - S_1= max-wt IS av G'
  - S_2= max-wt IS av G"
- returnerer S_1 eller $S_2 \cup \{V_n\}$

Hmmmm.....fornøyd?

La S være løsningen. Dvs max-wt IS av G og V_n er den siste noden i G

- Case 1: $V_n \notin S$
  - S er max-wt IS av G', der G' er G uten V_n
- Case 2: $V_n \in S$
  - V_{n-1} kan ikke være i S
  - $S - \{V_n\}$ er max-wt IS av G", der G" er G uten V_{n-1} og V_n

Ide: Prøv begge mulighetene og returnerer den beste

Forslag algo. 1:

- rekursivt beregner
  - S_1= max-wt IS av G'
  - S_2= max-wt IS av G"
- returnerer S_1 eller $S_2 \cup \{V_n\}$

Hmmmm.....fornøyd?

La S være løsningen. Dvs max-wt IS av G og V_n er den siste noden i G

- Case 1: $V_n \notin S$
  - S er max-wt IS av G', der G' er G uten V_n
- Case 2: $V_n \in S$
  - V_{n-1} kan ikke være i S
  - $S - \{V_n\}$ er max-wt IS av G", der G" er G uten V_{n-1} og V_n

Ide: Prøv begge mulighetene og returnerer den beste

Forslag algo. 1:

- rekursivt beregner
  - S_1= max-wt IS av G'
  - S_2= max-wt IS av G"
- returnerer S_1 eller $S_2 \cup \{V_n\}$

Hmmmm.....fornøyd?

La S være løsningen. Dvs max-wt IS av G og V_n er den siste noden i G

- Case 1: $V_n \notin S$
  - S er max-wt IS av G', der G' er G uten V_n
- Case 2: $V_n \in S$
  - V_{n-1} kan ikke være i S
  - $S - \{V_n\}$ er max-wt IS av G", der G" er G uten V_{n-1} og V_n

Ide: Prøv begge mulighetene og returnerer den beste

Forslag algo. 1:

- rekursivt beregner
  - S_1= max-wt IS av G'
  - S_2= max-wt IS av G"
- returnerer S_1 eller $S_2 \cup \{V_n\}$

Hmmmm.....fornøyd?

La S være løsningen. Dvs max-wt IS av G og V_n er den siste noden i G

- Case 1: $V_n \notin S$
  - S er max-wt IS av G', der G' er G uten V_n
- Case 2: $V_n \in S$
  - V_{n-1} kan ikke være i S
  - $S - \{V_n\}$ er max-wt IS av G", der G" er G uten V_{n-1} og V_n

Ide: Prøv begge mulighetene og returnerer den beste

Forslag algo. 1:

- rekursivt beregner
  - S_1= max-wt IS av G'
  - S_2= max-wt IS av G"
- returnerer S_1 eller $S_2 \cup \{V_n\}$

Hmmmm.....fornøyd?

La S være løsningen. Dvs max-wt IS av G og V_n er den siste noden i G

- Case 1: $V_n \notin S$
  - S er max-wt IS av G', der G' er G uten V_n
- Case 2: $V_n \in S$
  - V_{n-1} kan ikke være i S
  - $S - \{V_n\}$ er max-wt IS av G", der G" er G uten V_{n-1} og V_n

Ide: Prøv begge mulighetene og returnerer den beste

Forslag algo. 1:

- rekursivt beregner
  - S_1= max-wt IS av G'
  - S_2= max-wt IS av G"
- returnerer S_1 eller $S_2 \cup \{V_n\}$

Hmmmm.....fornøyd?

Enda bedre

Bruk en array A, der A[i] = verdien av max-wt IS av G_i

A[0] = 0
A[1] = w_1

For i = 2, 3, ..., n:

A[i] = max(A[i-1], A[i-2] + w_i)

Enda bedre

Bruk en array A, der A[i] = verdien av max-wt IS av G_i

A[0] = 0
A[1] = w_1

For i = 2, 3, ..., n:

A[i] = max(A[i-1], A[i-2] + w_i)

Enda bedre

Bruk en array A, der A[i] = verdien av max-wt IS av G_i

A[0] = 0
A[1] = w_1

For i = 2, 3, ..., n:

A[i] = max(A[i-1], A[i-2] + w_i)

Neste forelesning: 9. november

SORTERING