INF2220 - Algoritmer og datastrukturer

Institutt for informatikk, Universitetet i Oslo

INF2220, forelesning 11:
Huffman-koding & Dynamisk programmering

Dagens plan

- Grådige algoritmer
  - Huffman-koding
- Dynamisk programmering
  - Floyds algoritme for korteste vei alle-til-alle
- Paradigmer for algoritmedesign

Koding av tegn

Det finnes mange standarder for koding av tegn:

- Hollerith 12-bits hullkortkode
  Brukt på mekaniske datamaskiner før de elektroniske var oppfunnet
- BCD (Binary Coded Decimal)
  4-bits kode brukt av IBM til koding av desimale sifre
  (BCD ble også brukt om 6-bits tegn på noen tidlige datamaskiner)
- EBCDIC (Extended Binary Coded Decimal Interchange Code)
  IBMs 8-bits utvidelse av BCD til fullt tegnsett
- ASCII (American (National) Standard Code for Information Interchange)
  7-bits (senere 8-bits) tegn
- Unicode 8-bits tegn (Egentlig variabelt 8–32 bit)
  ISO-standard for alle tegn i alle språk

Huffman-koding

Motivasjon

- Store tekstfiler tar stor plass, og tar lang tid å overføre over nettet
- Tidligere var ASCII og EBCDIC de rådende standardene
  - Begge brukte 8-bits tegnkoder (ASCII brukte 7-bits kode + paritetsbit)
  - Dette ble ansett som sløsing med plass
- Unicode bruker (stort sett) 8-bits koder, men 32-bits koder blir også brukt
  - Unicode har $17 \cdot 2^{16} = 1\ 114\ 112$ mulige tegn
  - Selv i store tekstfiler blir bare noen få av disse brukt
  - Unicode gir minst dobbelt så store filer som ASCII
- *Konklusjon*

  Det er behov for datakompresjon!

Huffman-koding

Motivasjon

- Store tekstfiler tar stor plass, og tar lang tid å overføre over nettet
- Tidligere var ASCII og EBCDIC de rådende standardene
  - Begge brukte 8-bits tegnkoder (ASCII brukte 7-bits kode + paritetsbit)
  - Dette ble ansett som sløsing med plass
- Unicode bruker (stort sett) 8-bits koder,
  men 32-bits koder blir også brukt
  - Unicode har $17 \cdot \mathbf{2^{16}} = 1\ 114\ 112$ mulige tegn
  - Selv i store tekstfiler blir bare noen få av disse brukt
  - Unicode gir minst dobbelt så store filer som ASCII
- *Konklusjon*
  Det er behov for datakompresjon!

Huffman-koding

Motivasjon

- Store tekstfiler tar stor plass, og tar lang tid å overføre over nettet
- Tidligere var ASCII og EBCDIC de rådende standardene
  - Begge brukte 8-bits tegnkoder (ASCII brukte 7-bits kode + paritetsbit)
  - Dette ble ansett som sløsing med plass
- Unicode bruker (stort sett) 8-bits koder,
  men 32-bits koder blir også brukt
  - Unicode har $17 \cdot \mathbf{2}^{\mathbf{16}} = 1\ 114\ 112$ mulige tegn
  - Selv i store tekstfiler blir bare noen få av disse brukt
  - Unicode gir minst dobbelt så store filer som ASCII
- *Konklusjon*
  Det er behov for datakompresjon!

Idé og regler

- Hovedidé:
  Husk "Morse"
  Tegn som forekommer ofte ⇒ korte koder, sjeldne tegn ⇒ lange koder

- Regel 1:
  Hvert tegn som forekommer i filen, skal ha sin egen entydige kode
- Regel 2:
  Ingen kode er prefiks i en annen kode

- Eksempel på regel 2:
  Dersom 011001 er (binær)kode for et tegn, kan hverken 0, 01, 011, 0110 eller 01100 være kode for noe tegn

Idé og regler

- Hovedidé:
  Husk "Morse"
  Tegn som forekommer ofte ⇒ korte koder, sjeldne tegn ⇒ lange koder
- Regel 1:
  Hvert tegn som forekommer i filen, skal ha sin egen entydige kode
- Regel 2:
  Ingen kode er prefiks i en annen kode
- Eksempel på regel 2:
  Dersom 011001 er (binær)kode for et tegn, kan hverken 0, 01, 011, 0110 eller 01100 være kode for noe tegn

Idé og regler

- Hovedidé:
  Husk "Morse"
  Tegn som forekommer ofte ⇒ korte koder, sjeldne tegn ⇒ lange koder

- Regel 1:
  Hvert tegn som forekommer i filen, skal ha sin egen entydige kode
- Regel 2:
  Ingen kode er prefiks i en annen kode

- Eksempel på regel 2:
  Dersom 011001 er (binær)kode for et tegn, kan hverken 0, 01, 011, 0110 eller 01100 være kode for noe tegn

Algoritme

- Lag en frekvenstabell for alle tegn som forekommer i datafilen
- Betrakt hvert tegn som en node, og legg dem inn i en prioritetskø P med frekvensen som vekt
- Mens P har mer enn ett element
  - Ta ut de to minste nodene fra P
  - Gi dem en felles foreldrenode med vekt lik summen av de to nodenes vekter
  - Legg foreldrenoden inn i P
- Huffmankoden til et tegn (bladnode) får vi ved å gå fra roten og gi en '0' når vi går til venstre og '1' når vi går til høyre
- Resultatfilen består av to deler:
  - En tabell over Huffmankoder med tilhørende tegn
  - Den Huffmankodede datafilen

Algoritme

- Lag en frekvenstabell for alle tegn som forekommer i datafilen
- Betrakt hvert tegn som en node, og legg dem inn i en prioritetskø P med frekvensen som vekt
- Mens P har mer enn ett element
  - Ta ut de to minste nodene fra P
  - Gi dem en felles foreldrenode med vekt lik summen av de to nodenes vekter
  - Legg foreldrenoden inn i P
- Huffmankoden til et tegn (bladnode) får vi ved å gå fra roten og gi en '0' når vi går til venstre og '1' når vi går til høyre
- Resultatfilen består av to deler:
  - En tabell over Huffmankoder med tilhørende tegn
  - Den Huffmankodede datafilen

Algoritme

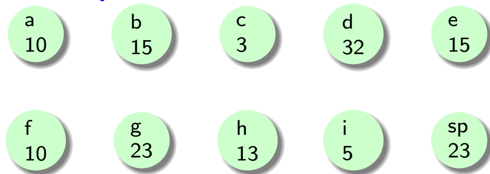
- Lag en frekvenstabell for alle tegn som forekommer i datafilen
- Betrakt hvert tegn som en node, og legg dem inn i en prioritetskø P med frekvensen som vekt
- Mens P har mer enn ett element
  - Ta ut de to minste nodene fra P
  - Gi dem en felles foreldrenode med vekt lik summen av de to nodenes vekter
  - Legg foreldrenoden inn i P
- Huffmankoden til et tegn (bladnode) får vi ved å gå fra roten og gi en '0' når vi går til venstre og '1' når vi går til høyre
- Resultatfilen består av to deler:
  - En tabell over Huffmankoder med tilhørende tegn
  - Den Huffmankodede datafilen

Algoritme

- Lag en frekvenstabell for alle tegn som forekommer i datafilen
- Betrakt hvert tegn som en node, og legg dem inn i en prioritetskø P med frekvensen som vekt
- Mens P har mer enn ett element
  - Ta ut de to minste nodene fra P
  - Gi dem en felles foreldrenode med vekt lik summen av de to nodenes vekter
  - Legg foreldrenoden inn i P
- Huffmankoden til et tegn (bladnode) får vi ved å gå fra roten og gi en '0' når vi går til venstre og '1' når vi går til høyre
- Resultatfilen består av to deler:
  - En tabell over Huffmankoder med tilhørende tegn
  - Den Huffmankodede datafilen

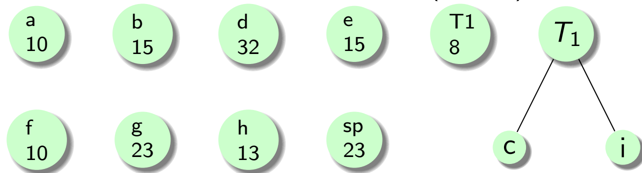
Eksempel

Initiell prioritetskø:



Bygging av treet

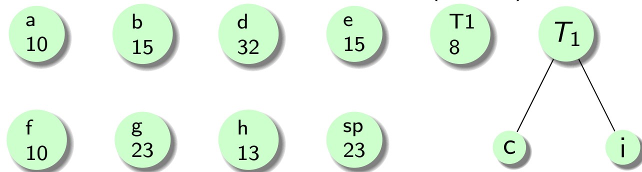
De 2 minste (sjeldnest forekommende) nodene er **c** og **i**.
Disse taes ut og erstattes med T_1 (vekt 8):



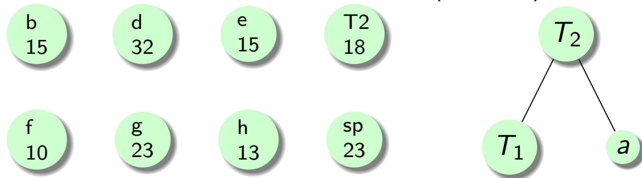
Dernest erstattes T_1 og a med T_2 (vekt 18):

Bygging av treet

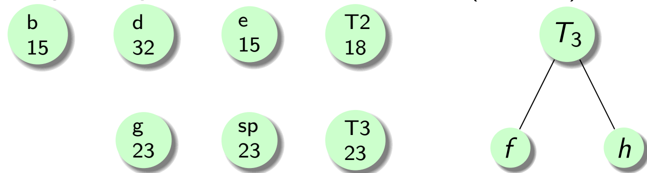
De 2 minste (sjeldnest forekommende) nodene er **c** og **i**.
Disse taes ut og erstattes med T_1 (vekt 8):



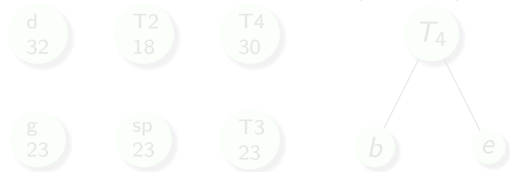
Dernest erstattes T_1 og a med T_2 (vekt 18):



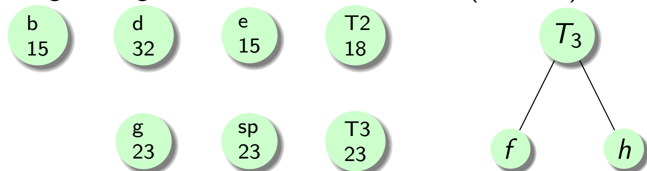
Så går f og h ut. De erstattes av T_3 (vekt 23):



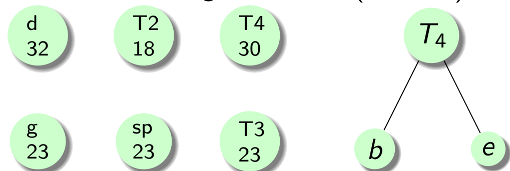
Så erstattes b og e med T_4 (vekt 30):



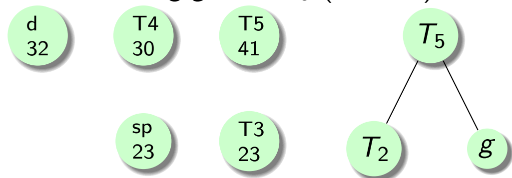
Så går f og h ut. De erstattes av T_3 (vekt 23):



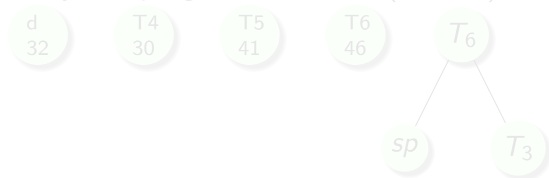
Så erstattes b og e med T_4 (vekt 30):



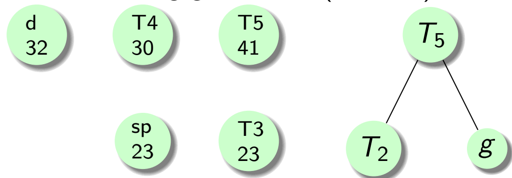
Dernest T_2 og g med T_5 (vekt 41):



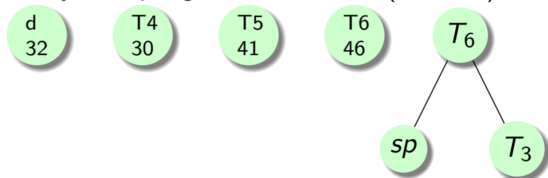
Så byttes sp og T_3 ut med T_6 (vekt 46):



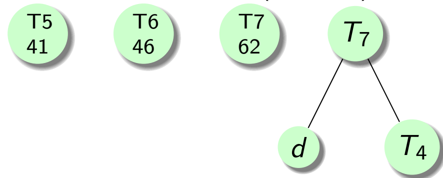
Dernest T_2 og g med T_5 (vekt 41):



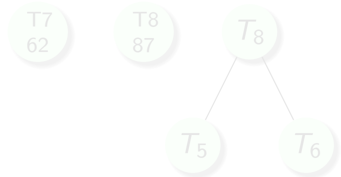
Så byttes sp og T_3 ut med T_6 (vekt 46):



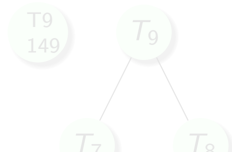
Så d og T_4 med T_7 (vekt 62):



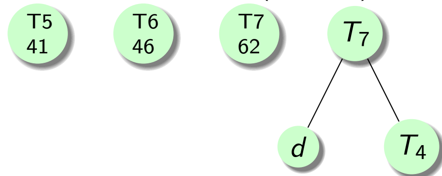
Så T_5 og T_6 med T_8 (vekt 87):



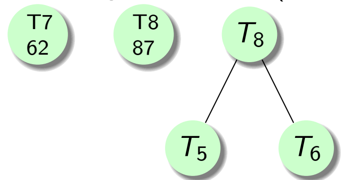
Endelig taes T_7 og T_8 ut av køen. Disse blir barn av rotnoden T_9:



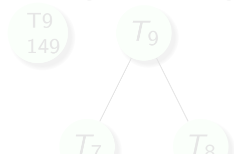
Så d og T_4 med T_7 (vekt 62):



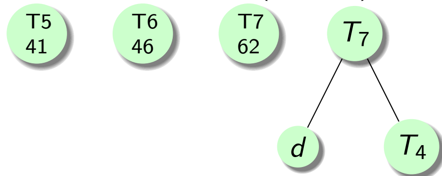
Så T_5 og T_6 med T_8 (vekt 87):



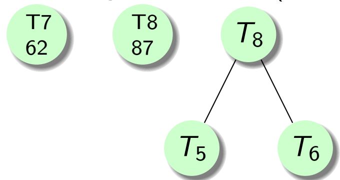
Endelig taes T_7 og T_8 ut av køen. Disse blir barn av rotnoden T_9:



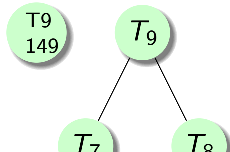
Så d og T_4 med T_7 (vekt 62):



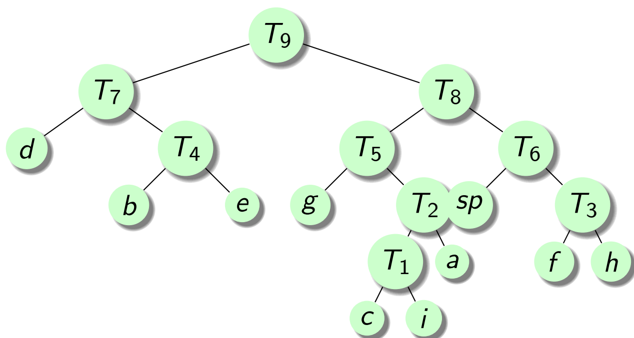
Så T_5 og T_6 med T_8 (vekt 87):



Endelig taes T_7 og T_8 ut av køen. Disse blir barn av rotnoden T_9:



Det ferdige kodetreet

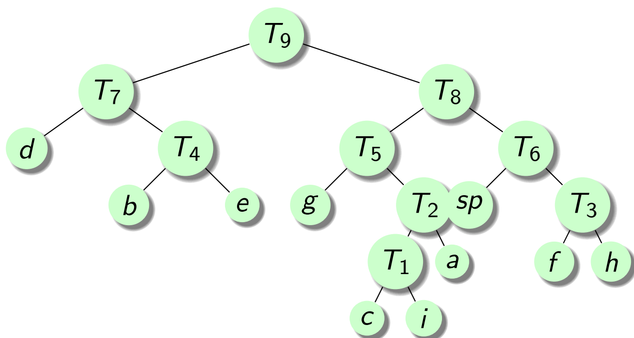


ser slik ut:

Det gir denne kodetabellen:
(venstre 0, høyre 1)

a	–	1011	f	–	1110
b	–	010	g	–	100
c	–	10100	h	–	1111
d	–	00	i	–	10101
e	–	011	sp	–	110

Det ferdige kodetreet



ser slik ut:

Det gir denne kodetabellen:

(venstre 0, høyre 1)

a	–	1011	f	–	1110
b	–	010	g	–	100
c	–	10100	h	–	1111
d	–	00	i	–	10101
e	–	011	sp	–	110

Eksempel:

caddie hadde bad i hagebed

har følgende Huffmankode:

10100101 10000101
01011110 11111011
00000111 10010101
10011010 10111011
11101110 00110100
1100

Vi brukte altså 84 bits på 26 tegn med dette 10-tegns alfabetet

Dekodingstabell

00	–	d	10101	–	i
010	–	b	1011	–	a
011	–	e	110	–	sp
100	–	g	1110	–	f
10100	–	c	1111	–	h

Eksempel:

caddie hadde bad i hagebed

har følgende Huffmankode:

10100101 10000101
01011110 11111011
00000111 10010101
10011010 10111011
11101110 00110100
1100

Vi brukte altså 84 bits på 26 tegn med dette 10-tegns alfabetet

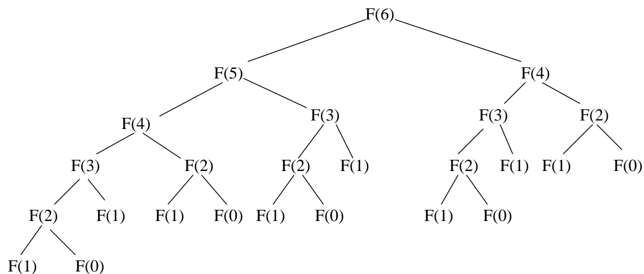
Dekodingstabell

00	–	d	10101	–	i
010	–	b	1011	–	a
011	–	e	110	–	sp
100	–	g	1110	–	f
10100	–	c	1111	–	h

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, . . . , **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 **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, . . . , **k** som mellomnoder

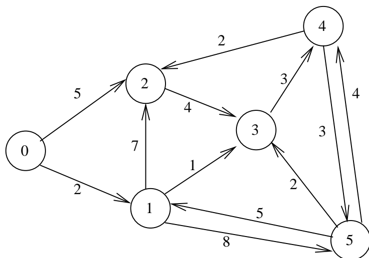
Korteste vei alle-til-alle (Floyd)

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

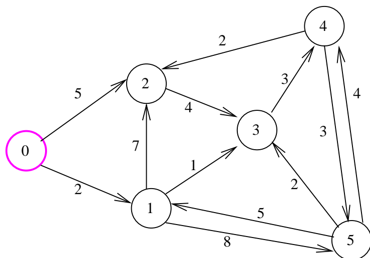
Grunnleggende idé

Hvis det går en vei fra node **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**

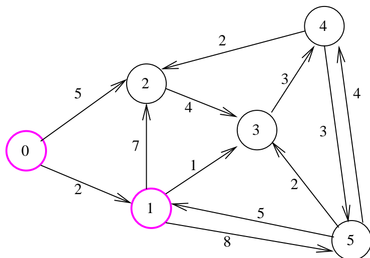
- Floyds algoritme:
  Denne betraktningen gjentas systematisk for alle tripler **i**, **k** og **j**:
  - Initielt: Avstanden fra node **i** til node **j** settes lik vekten på kanten fra **i** til **j**, uendelig hvis det ikke går noen kant fra **i** til **j**
  - Trinn 0: Se etter forbedringer ved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



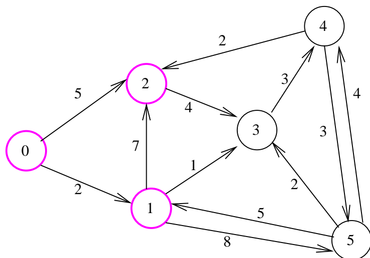
	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0



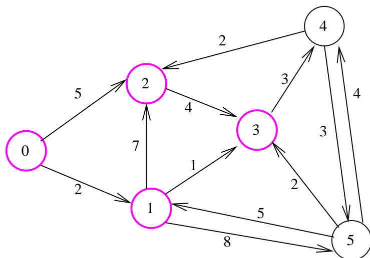
	0	1	2	3	4	5
0	0	2	5	∞	∞	∞
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	∞	2	4	0



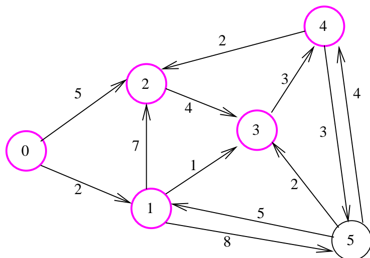
	0	1	2	3	4	5
0	0	2	5	3_1	∞	10_1
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	∞	0	3
5	∞	5	12_1	2	4	0



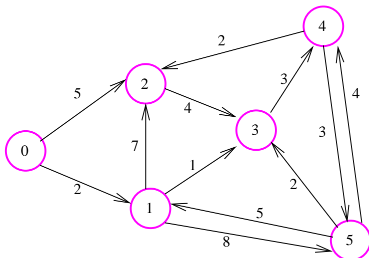
	0	1	2	3	4	5
0	0	2	5	3_1	∞	10_1
1	∞	0	7	1	∞	8
2	∞	∞	0	4	∞	∞
3	∞	∞	∞	0	3	∞
4	∞	∞	2	6_2	0	3
5	∞	5	12_1	2	4	0



	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

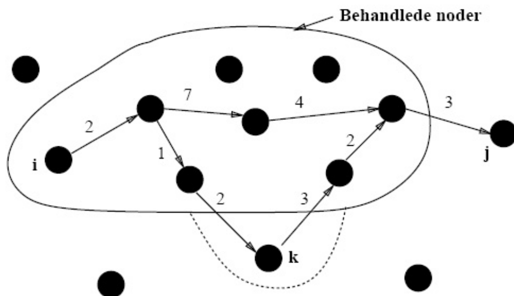


	0	1	2	3	4	5
0	0	2	5	3_1	6_3	9_4
1	∞	0	6_4	1	4_3	7_4
2	∞	15_5	0	4	7_3	10_4
3	∞	11_5	5_4	0	3	6_4
4	∞	8_5	2	5_5	0	3
5	∞	5	6_4	2	4	0

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	....

```
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 $\mathcal{O}(\mathbf{n^3})$

```
public static void korte steVeiAlleTilAlle(
  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 $\mathcal{O}(\mathbf{n^3})$

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] = k**
- **vei** sier hva som er den korteste veien
  - $\mathbf{k_1} = \mathbf{vei[i][j]}$ er den største verdi av **k** slik at **k** ligger på den korteste veien fra **i** til **j**
  - $\mathbf{k_2} = \mathbf{vei[i][k_1]}$ er den største verdi av **k** slik at **k** ligger på den korteste veien fra **i** til $\mathbf{k_1}$ osv.
  - Når $\mathbf{vei[i][k_m]} = -1$, er $(\mathbf{i}, \mathbf{k_m})$ 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
  - $\mathbf{k_1} = \mathbf{vei[i][j]}$ er den største verdi av **k** slik at **k** ligger på den korteste veien fra **i** til **j**
  - $\mathbf{k_2} = \mathbf{vei[i][k_1]}$ er den største verdi av **k** slik at **k** ligger på den korteste veien fra **i** til $\mathbf{k_1}$ osv.
  - Når $\mathbf{vei[i][k_m]} = -1$, er $(\mathbf{i}, \mathbf{k_m})$ 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 $\mathbf{vei[i][j] = k}$
- **vei** sier hva som er den korteste veien
  - $\mathbf{k_1 = vei[i][j]}$ er den største verdi av **k** slik at **k** ligger på den korteste veien fra **i** til **j**
  - $\mathbf{k_2 = vei[i][k_1]}$ er den største verdi av **k** slik at **k** ligger på den korteste veien fra **i** til $\mathbf{k_1}$ osv.
  - Når $\mathbf{vei[i][k_m] = -1}$, er $\mathbf{(i, k_m)}$ den første kanten i korteste vei fra **i** til **j**

Den korteste veien fra *i* til *j*

- Hvis $\mathbf{vei[i][j] = -1}$, passerer ikke den korteste veien gjennom noen mellomnoder og den korteste veien er $\mathbf{(i, j)}$
- Ellers, vi rekursivt beregne den korteste veien fra **i** til **vei[i][j]** og den korteste veien fra **vei[i][j]** til **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 parallelliseres (de er uavhengige delproblemer)
- Dette er definisjonen på dynamisk programmering

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)
- Dette er definisjonen på dynamisk programmering

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)
- Dette er definisjonen på dynamisk programmering

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

Neste forelesning: 5. november

SORTERING