NTNU

Norges teknisk-naturvitenskapelige universitet

Tekstsøk, Datakompresjon

Helge Hafting

23. oktober 2022

Anvendelser for tekstsøk

- Fritekstsøk i dokumenter, nettsider og lignende
- Fritekstsøk i databaser
- Søkemotorer
- Søke etter repeterte strenger for datakompresjon
- DNA-matching

En enkel naiv algoritme

Tekst: rabarbra (lengde n)

Søkeord: bra (lengde m)

Skyv søkeordet langs teksten, se om det passer

- tegn som passer, vises med **fet skrift**
- første feil med *kursiv*
- dobbeltløkke for $n - m$ posisjoner, og m tegn i søkeordet.

Forsøk	r	a	b	a	r	b	r	a
0	*b*	r	a					

En enkel naiv algoritme

Tekst: rabarbra (lengde n)

Søkeord: bra (lengde m)

Skyv søkeordet langs teksten, se om det passer

- tegn som passer, vises med **fet skrift**
- første feil med *kursiv*
- dobbeltløkke for $n - m$ posisjoner, og m tegn i søkeordet.

Forsøk	r	a	b	a	r	b	r	a
1		*b*	r	**a**				

En enkel naiv algoritme

Tekst: rabarbra (lengde n)

Søkeord: bra (lengde m)

Skyv søkeordet langs teksten, se om det passer

- tegn som passer, vises med **fet skrift**
- første feil med *kursiv*
- dobbeltløkke for $n - m$ posisjoner, og m tegn i søkeordet.

Forsøk	r	a	b	a	r	b	r	a
2			**b**	*r*	a			

En enkel naiv algoritme

Tekst: rabarbra (lengde n)

Søkeord: bra (lengde m)

Skyv søkeordet langs teksten, se om det passer

- tegn som passer, vises med **fet skrift**
- første feil med *kursiv*
- dobbeltløkke for $n - m$ posisjoner, og m tegn i søkeordet.

Forsøk	r	a	b	a	r	b	r	a
3				*b*	**r**	a		

En enkel naiv algoritme

Tekst: rabarbra (lengde n)

Søkeord: bra (lengde m)

Skyv søkeordet langs teksten, se om det passer

- tegn som passer, vises med **fet skrift**
- første feil med *kursiv*
- dobbeltløkke for $n - m$ posisjoner, og m tegn i søkeordet.

Forsøk	r	a	b	a	r	b	r	a
4					*b*	r	a	

En enkel naiv algoritme

Tekst: rabarbra (lengde n)

Søkeord: bra (lengde m)

Skyv søkeordet langs teksten, se om det passer

- tegn som passer, vises med **fet skrift**
- første feil med *kursiv*
- dobbeltløkke for $n - m$ posisjoner, og m tegn i søkeordet.

Forsøk	r	a	b	a	r	b	r	a
5						**b**	**r**	**a**

En enkel naiv algoritme

Tekst: rabarbra (lengde n)

Søkeord: bra (lengde m)

Hele greia, $O(n \cdot m)$, $\Omega(n)$

Forsøk	r	a	b	a	r	b	r	a
0	*b*	r	a					
1		*b*	r	**a**				
2			**b**	*r*	a			
3				*b*	**r**	a		
4					*b*	r	a	
5						**b**	**r**	**a**

Boyer-Moore

- Se på *siste* tegn i søketeksten først
- Hvis det ikke passer, flytt søketeksten *så langt vi kan*

	r	a	b	a	r	b	r	a
0	b	r	*a*					
1			b	r	*a*			
2				b	r	*a*		
3						**b**	**r**	**a**

- Hvis det passer, se på nestsiste osv.

Regelen om upassende tegn

- Hvis tegnet ikke fins i søketeksten, kan vi flytte *m* steg frem:

	m	e	t	e	o	r	i	t	t	s	t	e	i	n
0	s	t	e	i	*n*									
1						s	t	e	i	*n*				
2										**s**	**t**	**e**	**i**	**n**

- Hvis tegnet fins til venstre i søkeordet, kan vi flytte ordet så det passer med teksten
- Vi har vi en tabell for hvor mye vi kan flytte
- I praksis en tabell for hele alfabetet, hvor de fleste tegn gir et flytt på *m*. (Regel om «upassende tegn»)
- Tabellen lager vi ved å pre-prosessere søketeksten
- Tegn som fins i søketeksten, gir kortere flytt
  - En «s» i siste posisjon gir flytt på $m - 1$, fordi ordet starter på «s»
- $\Omega(n/m)$ for søket. Mye bedre!

Upassende tegn, fortsatt

- Hvis tegnet ikke fins i søketeksten, kan vi flytte m steg frem,
  - hvis mismatch var på *siste* tegn i søketeksten
  - med mismatch på *nestsiste* tegn kan vi flytte $m - 1$ steg
  - ved mismatch på nestnestsiste, flytter vi $m - 2$ steg osv.

	m	e	t	e	o	r	i	t	t	s	t	e	i	n
0	m	e	*n*	e										
1				m	e	n	*e*							

- Vi trenger altså en todimensjonal tabell:
  - En indeks er det upassende tegnet
  - Den andre indeksen er posisjonen i søketeksten
  - Verdien i cellen er hvor langt vi kan flytte fremover

Upassende tegn, lage tabellen

```
For hver posisjon p i søketeksten
   For hvert tegn x i alfabetet
      let mot start i søketeksten fra p
      hvis vi finner x etter i steg,
      sett Tab[p][x] = i
      hvis vi ikke finner x, Tab[p][x]=p+1
```

Regel om passende endelse

	r	e	n	n	e	n	e	
0	e	n	*e*					
1		e	n	*e*				
2			*e*	**n**	**e**			
					e	**n**	**e**	

- 0,1: Når siste posisjon treffer «n», kan vi bare flytte ett steg
- 2: Feil i første posisjon
  - Regel om «upassende tegn» lar oss bare flytte ett hakk
- Regel om «passende endelse» lar oss flytte to hakk her
- «ne» passet, og «ene» overlapper med seg selv
- Vi slår opp både «upassende tegn» og passende endelse», og bruker regelen som gir det lengste hoppet.

Passende endelse, tabell

- Tabellen for «passende endelse»
  - index er hvor mange tegn som passet
  - verdien i cellen er hvor langt vi kan flytte
- Lages ved å prøve ut om søketeksten overlapper med seg selv
  - ofte gjør den ikke det, og vi får lange hopp!

Galil sin regel

- Hvis vi søker etter «aaa» i «aaaaaa...», har vi dessverre $O(n \cdot m)$
  - søkeordet passer overalt, de samme a-ene sjekkes flere ganger
- Galil fant en måte å unngå unødvendige sammenligninger:
  - Når vi flytter søkeordet kortere enn den delen av søkeordet vi allerede har sjekket, trenger vi ikke sjekke det overlappende området omigjen.
  - Korte flytt skjer fordi søkeordet delvis matcher seg selv. Hvis det ikke hadde passet, hadde vi flyttet lenger.

Teksten	.	.	.	O	l	a	l	a	.	.	.
Mismatch O/a			l	*a*	**l**	**a**	**l**	**a**			
Nytt forsøk					**l**	**a**	**l**	**a**	l	a		

- Programmet trenger ikke sjekke den oransje regionen omigjen
- Dermed: $O(n)$ og $\Omega(n/m)$ for tekstsøk

Lenker

- Boyer og Moore sin artikkel:
  http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf
- Wikipedia:
  https://en.wikipedia.org/wiki/Boyer_moore_string_search_algorithm
- Animasjon (Fyll ut, og velg Boyer-Moore) Trenger java
  http://www.cs.pitt.edu/~kirk/cs1501/animations/String.html
- Demonstrasjon på Moore sin nettside:
  http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html

Run-length coding

- Enkleste form for datakompresjon
- En serie repetisjoner erstattes med et antall:
  - ABIIIIIIIIIIIIIBBBCDEFFFGH $\rightarrow$ AB12I3BCDE3FGH

Run-length coding

- Enkleste form for datakompresjon
- En serie repetisjoner erstattes med et antall:
  - ABIIIIIIIIIIIIBBBCDEFFFGH → AB12I3BCDE3FGH
- I praksis litt mer komplisert
  - det kan jo være sifre i det vi komprimerer
  - ser vanligvis på «bytes», ikke «tekst»
  - må kunne skille mellom data og metadata

Run-length coding

- Enkleste form for datakompresjon
- En serie repetisjoner erstattes med et antall:
  - ABIIIIIIIIIIIIBBBCDEFFFGH → AB12I3BCDE3FGH
- I praksis litt mer komplisert
  - det kan jo være sifre i det vi komprimerer
  - ser vanligvis på «bytes», ikke «tekst»
  - må kunne skille mellom data og metadata
- Eks., bruker negativ byte for ukomprimerte sekvenser:
  - ABIIIIIIIIIIIIBBBCDEFFFGH → [-2]AB[12]I[3]B[-3]CDE[3]F[-2]GH
  - 25 byte ble redusert til 16

Run-length coding

- Enkleste form for datakompresjon
- En serie repetisjoner erstattes med et antall:
  - ABIIIIIIIIIIIIBBBCDEFFFGH → AB12I3BCDE3FGH
- I praksis litt mer komplisert
  - det kan jo være sifre i det vi komprimerer
  - ser vanligvis på «bytes», ikke «tekst»
  - må kunne skille mellom data og metadata
- Eks., bruker negativ byte for ukomprimerte sekvenser:
  - ABIIIIIIIIIIIIBBBCDEFFFGH → [-2]AB[12]I[3]B[-3]CDE[3]F[-2]GH
  - 25 byte ble redusert til 16
- Kan ikke komprimere ABABABABABAB...

Lempel-Ziv kompresjon

- Leser gjennom fila
- Input kopieres til output
- Hvis en lang nok sekvens kommer omigjen:
  - dropp den, skriv heller en referanse til output
  - format: repeter X tegn, som vi har sett Y tegn tidligere
- Hjelper hvis sekvensen er lenger enn en slik referanse
- Søker bakover i et sirkulært buffer
- Output kan komprimeres videre med Huffman-koding

Bakover-referanser

- Må være *kompakt*
  - ellers kan vi ikke referere til korte strenger
  - f.eks. 2–3 byte
- Å «se» langt bakover i datastrømmen, gir større sjanse for å finne repetisjoner.
  - men også lenger kjøretid
  - påvirker formatet på referansene våre
    - 1 byte kan peke 255 tegn bakover
    - 2 byte kan peke 65 536 tegn bakover
    - 3 byte kan peke 16 777 215 tegn bakover
- I blant kan vi ikke komprimere
  - Må derfor også ha en måte å si:
  - Her kommer X bytes ukomprimerte data
  - Slik informasjon tar også plass!

Hva kan komprimeres?

- Vurdering:
  - Skal dette være en del av en større ukomprimert blokk?
  - Evt. bakover-ref + header for kortere ukomprimert blokk
- Det vi komprimerer må altså være lenger enn samlet lengde for:
  - en bakover-referanse
  - header for en ukomprimert blokk
- Vi komprimerer ikke svært korte strenger, det hjelper ikke!

Eksempel

- Eksempeltekst:
  Problemer, problemer. Alltid problemer!
  Dette er dagens problem. Problemet er
  å komprimere problematisk tekst.
- Eksempeltekst med avstander:
  Problemer,[10] problemer[20]. Alltid p[30]roblemer!
  [40]Dette er d[50]agens prob[60]lem. Probl[70]emet er
  å [80]komprimere[90] problemat[100]isk tekst.[110]
- 110 tegn, inkludert linjeskift og blanke.

Eksempel

- Eksempeltekst med avstander:
  Problemer,[10] problemer[20]. Alltid p[30]roblemer!
  [40]Dette er d[50]agens prob[60]lem. Probl[70]emet er
  å [80]komprimere[90] problemat[100]isk tekst.[110]
- Komprimert:
  [12]Problemer, p[-11,8][8]. Alltid[-18,10][17]!
  Dette er dagens[-27,7][2]. [-65,8][17]t er
  å komprimere[-35,8][12]atisk tekst.
- Før komprimering, 110 tegn.
- Med 1 byte per tallkode, 84 tegn.
  Vi sparte 110-84=26 tegn, eller 23%
- se også Lz-demo

Kjøretid

- For hver tegnposisjon i input, må vi søke etter lengste match i bufferet.
- Fil med n tegn, sirkulært buffer med størrelse m.
- Teste alle posisjoner, i verste fall $O\left(nm^2\right)$
- I praksis går det bedre, særlig hvis data varierer en del
- Kan bruke Boyer-Moore tekstsøk for bedre kjøretid.

Lenker

- Lempel og Ziv sin artikkel:
  http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf
- Wikipedia:
  https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv

Kombinere LZ og Huffmann

- LZ leser input, og skriver
  - bakover-referanser
  - sekvenser med ukomprimerte tegn
- ukomprimerte tegn telles opp, og komprimeres videre med Huffmannkoding

LZW – Lempel Ziv Welsh

- Ligner LZ. Teoretisk samme kompresjon. Lettere å speede opp.
- Leser ett og ett tegn
- Bygger en ordliste (dictionary) underveis
  - til å begynne med, alle 1-byte «ord»
- Finn et (lengst mulig) ord, skriv ordnummeret (med færrest mulig bits!)
  - lagre nytt «ord» = dette ordet + neste tegn
- Kompresjon hvis ordene blir lengre enn numrene
- LZW+Huffman $\rightarrow$Deflate (brukt i zip)

LZW – Lempel Ziv Welsh

- Ligner LZ. Teoretisk samme kompresjon. Lettere å speede opp.
- Leser ett og ett tegn
- Bygger en ordliste (dictionary) underveis
  - til å begynne med, alle 1-byte «ord»
- Finn et (lengst mulig) ord, skriv ordnummeret (med færrest mulig bits!)
  - lagre nytt «ord» = dette ordet + neste tegn
- Kompresjon hvis ordene blir lengre enn numrene
- LZW+Huffman →Deflate (brukt i zip)
- Se eksempel «lzw»

Kombinere LZW og Huffmann

- LZW
  - leser input,
  - bygger en dictionary,
  - skriver «ordnumre»
- Noen «ord» forekommer oftere enn andre
- Programmet finner antall (frekvenser) for ulike ordnumre,
  - skriver Huffmankoder i stedet for ordnumre
  - ord som forekommer ofte, får kortere koder

BZip2 blokk-komprimering

- Komprimerer mer enn LZ-algoritmene

1. run-length coding
2. Burrows-Wheeler transformasjon (hoveddel)
3. Move-To-Front transformasjon (MFT)
4. run-length coding igjen
5. Huffmannkoding

Burrows Wheeler transformasjonen (BWT)

- Hoveddelen av BZ2 (blokksorteringen)
- Dette steget komprimerer ikke selv, men transformerer en blokk (typisk 900kB)
- Transformerer repeterte sekvenser (som ord) til repeterte tegn
- Repeterte *tegn* er lettere å komprimere videre!
- Transformasjonen er reversibel (for dekomprimering)

Eksempel, Burrows-Wheeler Transformasjon

- BWT på ordet «refererΩ». Tegnet «Ω» markerer slutten

Rotasjoner	Sortert
refererΩ	efererΩr
Ωrefererer	erererΩref
rΩreferere	erΩrefer
erΩreferer	erΩreferer
rerΩrefere	fererΩre
erΩrefer	refererΩ
rerΩrefe	rerΩrefe
erΩref	rerΩrefere
fererΩre	rΩreferere
efererΩr	Ωrefererer

- BWT er siste kolonne med tegn fra sortert liste, «rfrreΩeeer»
- Nå har vi mange like tegn ved siden av hverandre,
  - lettere å komprimere med run-length coding
  - Se også bw brukt på diverse filer

Reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Vet at «rfrreΩeeer» er siste kolonne i sortert liste
- Lista bestod av ulike *rotasjoner* av *samme* ord
  - alle kolonner inneholder de samme tegnene
- Lista var sortert
  - første kolonne må altså ha de samme tegnene, sortert
  - altså «eeeefrrrrΩ»
- Vi har nå to kolonner, i ei liste over rotasjoner
  - kan rotere sidelengs, så siste kolonne blir første, og første blir andre
  - dette er fortsatt en del av løsningen
  - sorterer vi dette, har vi de *to første* kolonnene
  - så kan vi legge på siste kolonne igjen
  - vi har nå tre kolonner. Repeter til vi har alle!
- Riktig rad er den som har «Ω» på siste plass

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
r
f
r
r
e
Ω
e
e
e
r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
r
f
r
r
e
Ω
e
e
e
r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Sortere

```
e
e
e
e
f
r
r
r
r
Ω
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

e	r
e	f
e	r
e	r
f	e
r	Ω
r	e
r	e
r	e
Ω	r

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
re
fe
re
re
ef
Ωr
er
er
er
rΩ
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Sortere

```
ef
er
er
er
fe
re
re
re
rΩ
Ωr
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
ef      r
er      f
er      r
er      r
fe      e
re      Ω
re      e
re      e
rΩ      e
Ωr      r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
ref
fer
rer
rer
efe
Ωre
ere
ere
erΩ
rΩr
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Sortere

```
efe
ere
ere
erΩ
fer
ref
rer
rer
rΩr
Ωre
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
efe     r
ere     f
ere     r
erΩ     r
fer     e
ref     Ω
rer     e
rer     e
rΩr     e
Ωre     r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
refe
fere
rere
rerΩ
efer
Ωref
erer
erer
erΩr
rΩre
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Sortere

```
efer
erer
erer
erΩr
fere
refe
rere
rerΩ
rΩre
Ωref
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
efer     r
erer     f
erer     r
erΩr     r
fere     e
refe     Ω
rere     e
rerΩ     e
rΩre     e
Ωref     r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refereΩ»?
- Rotere mot høyre

```
refer
ferer
rerer
rerΩr
efere
Ωrefe
erere
ererΩ
erΩre
rΩref
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Sortere

```
efere
erere
ererΩ
erΩre
ferer
refer
rerer
rerΩr
rΩref
Ωrefe
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
efere    r
erere    f
ererΩ    r
erΩre    r
ferer    e
refer    Ω
rerer    e
rerΩr    e
rΩref    e
Ωrefe    r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
refere
ferere
rererΩ
rerΩre
eferer
Ωrefer
ererer
ererΩr
erΩref
rΩrefe
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refereΩ»?
- Sortere

```
eferer
ererer
ererΩr
erΩref
ferere
refere
rererΩ
rerΩre
rΩrefe
Ωrefer
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
eferer   r
ererer   f
ererΩr   r
erΩref   r
ferere   e
refere   Ω
rererΩ   e
rerΩre   e
rΩrefe   e
Ωrefer   r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
referer
fererer
rererΩr
rerΩref
eferere
Ωrefere
erererΩ
ererΩre
erΩrefe
rΩrefer
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Sortere

```
eferere
ererer Ω
erer Ω re
er Ω refe
fererer
referer
rerer Ω r
rer Ω ref
r Ω refer
Ω refere
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
eferere  r
erererΩ  f
ererΩre  r
erΩrefe  r
fererer  e
referer  Ω
rererΩr  e
rerΩref  e
rΩrefer  e
Ωrefere  r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
referere
ferererΩ
rererΩre
rerΩrefe
efererer
Ωreferer
erererΩr
ererΩref
erΩrefer
rΩrefere
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Sortere

```
efererer
erererΩr
ererΩref
erΩrefer
ferererΩ
referere
rererΩre
rerΩrefe
rΩrefere
Ωreferer
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Legg til siste

```
efererer r
erererΩr f
ererΩref r
erΩrefer r
ferererΩ e
referere Ω
rererΩre e
rerΩrefe e
rΩrefere e
Ωreferer r
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refererеΩ»?
- Rotere mot høyre

```
refererer
fererer Ωr
rerer Ωref
rer Ωrefer
efererer Ω
Ωreferere
erererΩre
erer Ωrefe
er Ωrefere
r Ωreferer
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refereΩ»?
- Sortere

```
eferererΩ
erererΩre
ererΩrefe
erΩrefere
ferererΩr
refererer
rererΩref
rerΩrefer
rΩreferer
Ωreferere
```

Animasjon, reversere Burrows-Wheeler transformasjonen

- Hvordan gå fra «rfrreΩeeer» til «refereΩ»?
- Legg til siste

```
efererererΩr
erererΩref
ererΩrefer
erΩreferer
fererererΩre
refererererΩ ⟸ Der
rererΩrefe
rerΩrefere
rΩreferere
Ωrefererer
```

Move-to-front transformasjonen

- Komprimerer ikke data, men forbereder
- Initialiserer en tabell med alle byte-verdier. t[0]=0, t[1]=1, t[2]=2, ...
- Leser ett og ett tegn fra input
  - finn tegnet i tabellen, skriv index til output
  - flytt tegnet vi fant til første plass i tabellen (move to front)
- input: caaaaacbbbbbababababab

```
     inn:caaaaacbbbbbababababab
      ut:21000012000021111111
tabell
     0: aca....cb....abababab
     1: bac....ac....babababa
     2: cbb....ba....c.......
     3: ddd....dd....d.......
```

- Alle repeterte tegn blir til nuller

MTF gir mer effektiv Huffmankoding

- Eksempel

```
inn:caaaaacbbbbbbbaaaabb
 ut:21000012000000200010
```

-

	Frekv. før		Frekv. etter
a	9	0	14
b	9	1	3
c	2	2	3

- Før: like mange «a» som «b»
- Etter: overvekt av «0», som kan få kortere kode.

BZ2

- Burrows-Wheeler sorterer så vi får mange repetisjoner
  - 900 kB blokkstørrelse
- Move-to-front gjør ulike repetisjoner om til nuller
- Deretter fungerer run-length coding veldig bra!
- Huffmannkoding av det som blir igjen

Adaptiv kompresjon

- Huffmankoding av ei fil, bruker samme koding for hele fila
- Ei fil kan bestå av ulike deler (f.eks. norsk+engelsk)
  - Ulike deler har ulik bokstavfordeling
  - De komprimeres best med ulike Huffman-trær
- Adaptiv komprimering ser når frekvensene endrer seg, og lager nytt huffmanntre

Aritmetisk koding

- Kan komprimere bedre enn Huffmann
- Huffmann bruker faste koder, aldri mindre enn 1 bit
- Hvis sjansen for et symbol er 50%, vil begge kode det med 1 bit
- Hvis sjansen for et symbol er 90%:
  - Huffman: fremdeles 1 bit
  - Aritmetisk koding: 0,1 bit

Kompresjon og AI

- Noen forskere mener datakompresjon og AI er samme problem
  - AI: det *korteste* programmet som oppfører seg intelligent
- Å oppdage repeterte mønstre (kan nyttes for kompresjon) krever intelligens
- Mer intelligens gir bedre kompresjon
- Desimalene i π er et vanskelig datasett å komprimere. (mye variasjon) Men:
  - vi kjenner rekkeutviklinger som genererer π.
  - Et program med endelig lengde, kan generere hele rekka. ∞ kompresjon!
- Ei zipfil er vanskelig å komprimere, selv om det fins bedre kompresjon enn zip
  - Hvis vi pakker ut zipfila, kan vi komprimere bedre med bz2
  - Å oppdage at noe er zip-komprimert, og dermed kan behandles slik, krever intelligens...