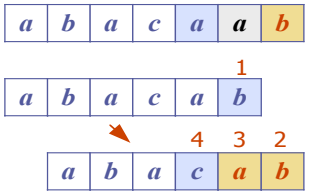
Strengsøgning



Strenge

En **streng** er en sekvens af tegn

Eksempler på strenge:
- Java-program
- HTML-dokument
- DNA-sekvens
- Digitaliseret billede

Et **alfabet** Σ er mængden af mulige tegn for en mængde af strenge

Eksempler på alfabeter:
- ASCII
- Unicode
- {A, C, G, T}
- {0, 1}

Lad P være en streng af længde m

En **delstreng** $P[i \,..\, j]$ af P er den delsekvens af P, der består af tegn med indeks imellem i og j

Et **prefix** af P er en delstreng af typen $P[0 \,..\, i]$
Et **suffix** af P er en delstreng af typen $P[i \,..\, m-1]$

Lad der være givet en streng T (tekst) og en streng P (mønster).
Strengsøgning (pattern matching) består i at finde en delstreng i T, der er lig med P

Anvendelser:
- Tekstredigeringsprogrammer
- Søgemaskiner
- Biologisk forskning

Rå kraft algoritme

Den rå kraft strengsøgningsalgoritme sammenligner mønsteret P med teksten T for ethvert muligt skift af P i forhold til T, indtil enten
- der er fundet overensstemmelse
- alle mulige placeringer af mønsteret er blevet afprøvet

Algoritmens køretid er $O(nm)$

Eksempel på værste tilfælde:
$T = aaa \ldots ah$
$P = aaah$
forekommer ofte for billeder og DNA-sekvenser, men sjældent for almindelig tekst

```
Algorithm BruteForceMatch(T, P)
    Input text T of size n and pattern
        P of size m
    Output starting index of a
        substring of T equal to P or −1
        if no such substring exists
    for i ← 0 to n − m
        { test shift i of the pattern }
        j ← 0
        while j < m ∧ T[i + j] = P[j]
            j ← j + 1
        if j = m
            return i {match at i}
    return -1 {no match anywhere}
```

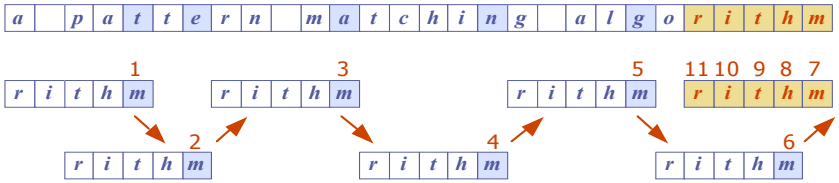

Boyer-Moore's heuristik

Boyer-Moore's strengsøgningsalgoritme er baseret på to heuristikker:

Spejl-heuristikken: Sammenlign P med en delsekvens af T baglæns

Hop-heuristikken: Når en tegnsammenligning mislykkes ved $T[i] = c$:
Hvis P indeholder tegnet c, så skift P således, at den sidste forekomst af c i P kommer ovenpå $T[i]$
Ellers, skift P således, at $P[0]$ kommer ovenpå $T[i + 1]$

Eksempel:



Sidste-forekomst-funktion

Boyer-Moore's algoritme forbehandler mønsteret P og alfabetet Σ for at konstruere den sidste-forekomst-funktion L, der afbilder fra Σ til heltal, hvor $L(c)$ er defineret som

det største indeks i, for hvilket $P[i] = c$, eller
-1, hvis der ikke eksisterer et sådan indeks

Eksempel:

$\Sigma = \{a, b, c, d\}$
$P = abacab$

c	a	b	c	d
$L(c)$	4	5	3	−1

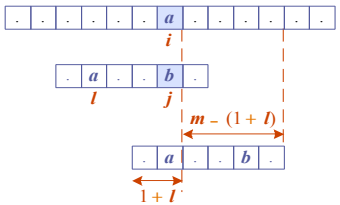
Funktionen L kan repræsenteres ved et array, der indiceres ved de numeriske koder for alfabetets tegn

L kan bestemmes i $O(m + |\Sigma|)$ tid, hvor m er længden af P, og $|\Sigma|$ er størrelsen af Σ

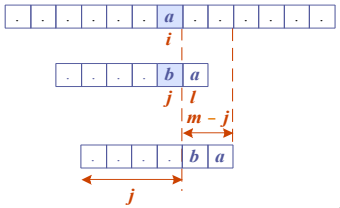
Boyer-Moore's algoritme

```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ)
    i ← m − 1
    j ← m − 1
    repeat
        if T[i] = P[j]
            if j = 0
                return i  { match at i }
            else
                i ← i − 1
                j ← j − 1
        else
            { character-jump }
            l ← L[T[i]]
            i ← i + m − min(j, 1 + l)
            j ← m − 1
    until i > n − 1
    return −1  { no match }
```

Tilfælde 1: $1 + l < j$

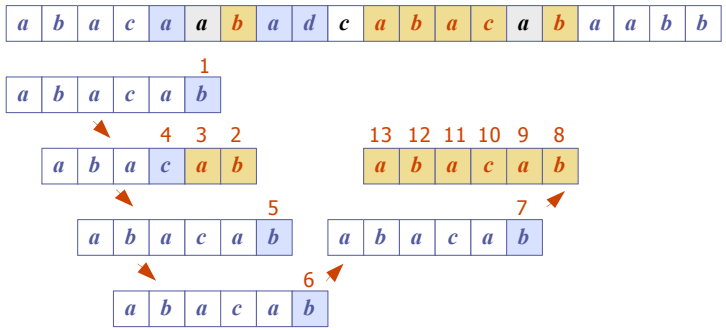


Tilfælde 2: $j \leq 1 + l$



Eksempel

c	a	b	c	d
$L(c)$	4	5	3	−1



Analyse

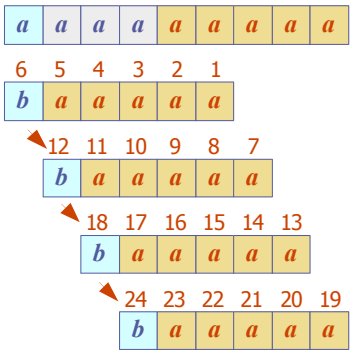
Boyer-Moore's algoritme kører i $O(nm + |\Sigma|)$ tid

Eksempel på værste tilfælde:
$T = aaa \ldots a$
$P = baaa$

Det værste tilfælde kan optræde for billeder og DNA-sekvenser, men forekommer sjældent for almindelig tekst

Boyer-Moore's algoritme er betydeligt hurtigere end rå kraft algoritmen for almindelig tekst

Den viste algoritme er en forsimpling af Boyer-Moore's oprindelige algoritme, som kører i $O(n + m + |\Sigma|)$ tid

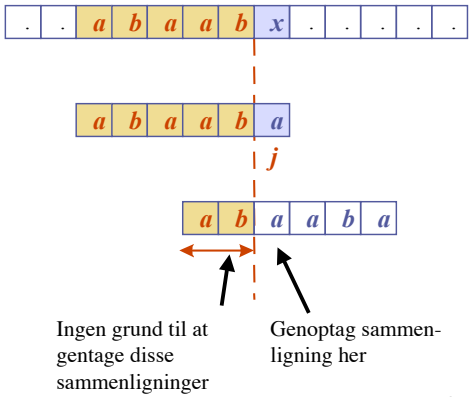


KMP-algoritmen (motivation)

Knuth-Morris-Pratt's algoritme sammenligner mønsteret i teksten **fra venstre imod højre**, men skifter mønsteret mere intelligent end rå kraft algoritmen (kaster ikke information væk)

Når en tegnsammenligning mislykkes, hvad er da det længste vi kan skifte mønsteret for at undgå overflødige sammenligninger?

Svar: det længste prefix af $P[0..j-1]$, som er suffix af $P[1..j-1]$



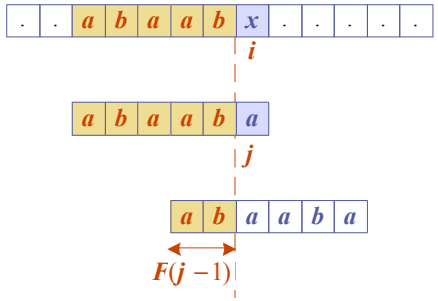
KMP's fejlfunktion

Knuth-Morris-Pratt's algoritme forbehandler mønsteret for at finde prefixer af mønsteret, der findes i mønsteret selv

Fejlfunktionen $F(j)$ defineres som længden af det længste prefix af $P[0..j]$, som er suffix af $P[1..j]$

Knuth-Morris-Pratt's algoritme ændrer på rå kraft algoritmen, så vi ved en mislykket sammenligning, $P[j] \neq T[i]$, sætter $j \leftarrow F(j-1)$

j	0	1	2	3	4	5
$P[j]$	a	b	a	a	b	a
$F(j)$	0	0	1	1	2	3



KMP-algoritmen

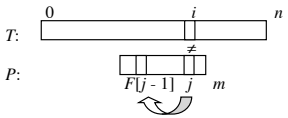
Fejlfunktionen kan repræsenteres ved et array

I hver iteration af while-løkken har vi, at

- i øges med 1, eller
- skiftets størrelse $i - j$ øges med mindst 1 (da $F(j-1) < j$)
- $i - j \leq n$

Der er således højst $2n$ iterationer af while-løkken

KMP-algoritmen kører i optimal tid, $O(m+n)$




```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```


Beregning af fejlfunktionen

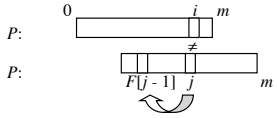
Fejlfunktionen kan repræsenteres ved et array og kan konstrueres i $O(m)$ tid

Konstruktionen svarer til KMP-algoritmen selv

I hver iteration af while-løkken har vi, at

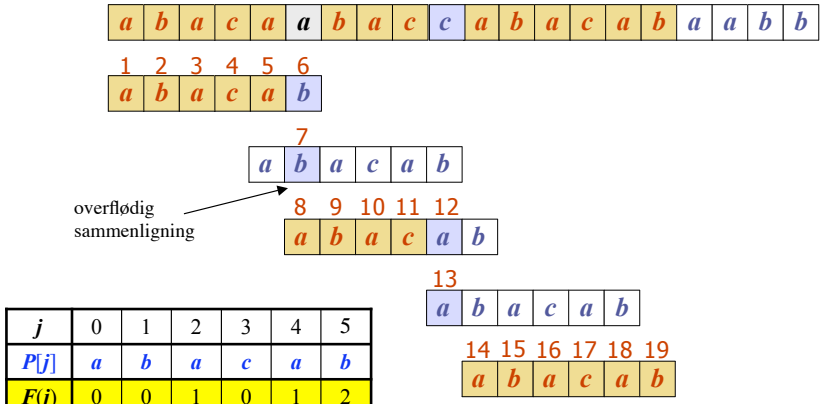
- i øges med 1, eller
- skiftets størrelse $i - j$ øges med mindst 1 (da $F(j-1) < j$)
- $i - j \leq m$

Der er således højst $2m$ iterationer af while - løkken




```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 characters}
            F[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 {no match}
            i ← i + 1
```

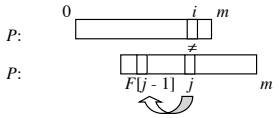
Eksempel



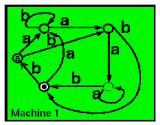
j	0	1	2	3	4	5
P[*j*]	*a*	*b*	*a*	*c*	*a*	*b*
***F*(*j*)**	0	0	1	0	1	2

Forbedret beregning af fejlfunktionen

```
Algorithm failureFunction(P)
    F[0] ← 0
    i ← 1
    j ← 0
    while i < m
        if P[i] = P[j]
            {we have matched j + 1 characters}
            if P[i] ≠ P[j + 1]
                F[i] ← j + 1
            else
                F[i] ← F[j + 1]
            i ← i + 1
            j ← j + 1
        else if j > 0 then
            {use failure function to shift P}
            j ← F[j − 1]
        else
            F[i] ← 0 {no match}
            i ← i + 1
```



Endelige tilstandsmaskiner

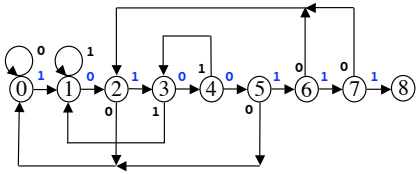


En **endelig tilstandsmaskine** er karakteriseret ved

1. en endelig mængde af tilstande
2. en starttilstand
3. en eller flere sluttilstande
4. en endelig mængde af inputsymboler
5. en funktion, `move`, der afbilder mængden af par bestående af et inputsymbol og en tilstand på mængden af tilstande

   `move(input, state)` ⟶ `state`

En endelig tilstandsmaskine til genkendelse af mønsteret `10100111`



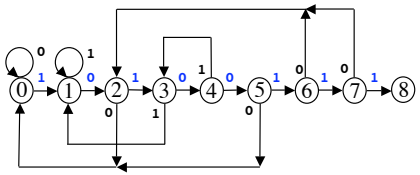
Starttilstand: 0 Sluttilstand: 8

Søgeeksempel:

Tekst: `1001110100101000101001110000111`

Tilstand: `012011123456234501234567 8`

Repræsentation af en endelig tilstandsmaskine



En endelig tilstandsmaskine kan repræsenteres ved hjælp af en tabel

move-tabel:

input\state	0	1	2	3	4	5	6	7
0	0	2	0	4	5	0	2	2
1	1	1	3	1	3	6	7	8

Strengsøgning ved hjælp af en endelig tilstandsmaskine

(1) Byg en endelig tilstandsmaskine ud fra mønsteret

(2) Kør maskinen på teksten

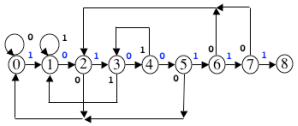
Algorithm *KMP_FSA_Match*(***T, P***)
```
move ← moveArray(P)
i ← 0
state ← 0
while i < n and state < m do
    state ← move[T[i], state]
if state = m then
    return i - m + 1
return −1 { no match }
```

Tidsforbrug: $O(n)$

Konstruktion af endelig tilstandsmaskine

Efterfølgertilstande til tilstand ***i***:

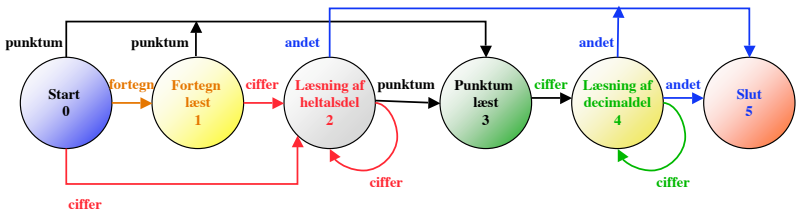
- match, så ***i*** + 1
- mismatch, så den tilstand, der svarer til
  "længst mulige match" (tag højde for mismatch)



Eksempel: 10100111
Tilstand 6
  1010011: gå til tilstand 7
  1010010: gå til tilstand for 010010 (tilstand 2),
    husk tilstand for 010011 (X = 1)
Tilstand 7
  10100111: gå til tilstand 8
  10100110: gå til tilstand for 0100110 (tilstand 2),
    husk tilstand 0100111 (X = 1)
    (disse er efterfølgere af forrige X)

Endelig tilstandsmaskine til indlæsning af decimaltal

(f.eks. -3.1415)



Rabin-Karp's algoritme

Ide: Brug hashing

- Beregn hashværdien af hver mulig delstreng af længde ***m*** i ***T***
- Sammenlign med hashværdien for ***P***
- Hvis to hashværdier er ens, så sammenlign delstrengen i ***T*** med ***P*** tegn for tegn

Der ikke brug for nogen hashtabel, kun dens størrelse

Eks: "tabelstørrelse" = 97, ***m*** = 5.
Find **15926**. Hashværdi = **18** (mod 97)

```
3141592653589793238462643 3
31415 = 84 (mod 97)
 14159 = 94 (mod 97)
  41592 = 76 (mod 97)
   15926 = 18 (mod 97)
```


Algoritmedesign

Problem: Hashfunktionen afhænger af ***m*** tegn
Løsning: Beregn hashværdi for ***i***+1 ud fra hashværdi for ***i***
(forbedrer køretiden fra ***O*(*nm*)** til ***O*(*n*+*m*)**)

```
31415 = 84 (mod 97)
 14159  = (31415 - 3*10000)*10 + 9
        = (84 - 3*9)*10 + 9 (mod 97)
        = 579 = 94 (mod 97)
  41592  = (94 - 1*9)*10 + 2 = 76 (mod 97)
   15926 = (76 - 4*9)*10 + 6 = 18 (mod 97)
```

Problem: En fuldstændig sammenligning er nødvendig ved kollision
Afhjælpning: Benyt en meget stor (virtuel) tabelstørrelse

Beregning af hashværdi

Antag at en hashværdi er beregnet ud fra

$$h = T[i]d^{m-1} + T[i+1]\,d^{m-2} + \ldots + T[i+m-1],$$

hvor d er antallet af mulige tegn

Den næste værdi af h

$$T[i+1]d^{m-1} + T[i+2]\,d^{m-2} + \ldots + T[i+M-1]d + T[i+M]$$

kan da bestemmes ud fra h:

$$(h - T[i]d^{m-1})d + T[i+M]$$

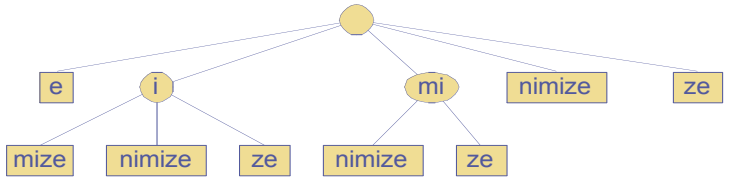
Empirisk undersøgelse af algoritmernes effektivitet

Søgning i "The Oxford English Dictionary" (2nd Edition), cirka 570 millioner tegn

Algoritme	*"to be or not to be"*	*"data"*
Rå kraft	1.23	1.74
Knuth-Morris-Pratt	2.16	2.93
Boyer-Moore	1.33	1.16
Rabin-Karp	2.64	3.69
Boyer-Moore-Horspool	**1.00**	**1.00**

fra G. H. Gonnet & R. Baeza-Yates:
Handbook of Algorithms and Data Structures
in Pascal and C, Addison-Wesley 1991

Tries



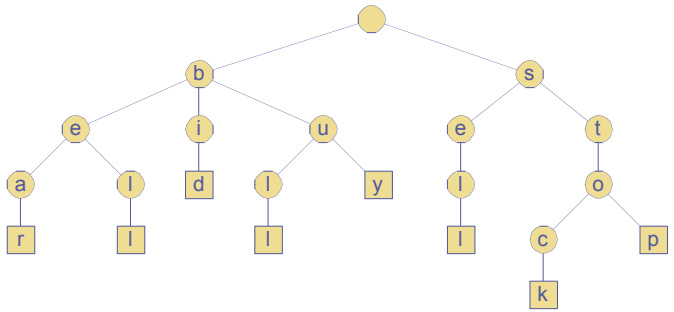
Forbehandling af strenge

- Forbehandling af mønsteret sætter hastigheden op for strengsøgning
  Efter forbehandling af mønsteret udfører KMP-algoritmen strengsøgning i tid, der er proportional med tekstens længde
- Hvis en tekst er lang, uforanderlig og søges i hyppigt (f.eks. Shakespeares værker), kan vi ønske at forbehandle teksten snarere end mønsteret
- Et **trie** (udtales *traj*) er en kompakt datastruktur til repræsentation af strenge, såsom ordene i en tekst
  Et trie muliggør strengsøgning i tid, der er proportional med mønsterets længde

Standard trie (1)

Et **standard trie** for en mængde af strenge ***S*** er et ordnet træ, hvor
- Hver knude, udtagen roden, indeholder et tegn
- Børnene af en knude er ordnet alfabetisk
- Vejene fra roden til de eksterne knuder giver strenge i ***S***

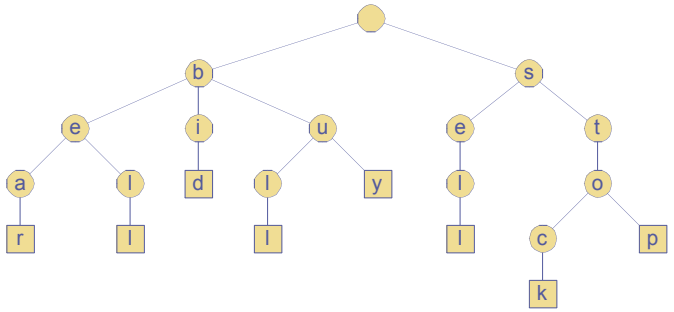
Eksempel: ***S*** = { bear, bell, bid, bull, buy, sell, stock, stop }



Standard trie (2)

Et standard trie bruger $O(n)$ plads og tillader søgning, indsættelse og fjernelse i $O(dm)$ tid, hvor:

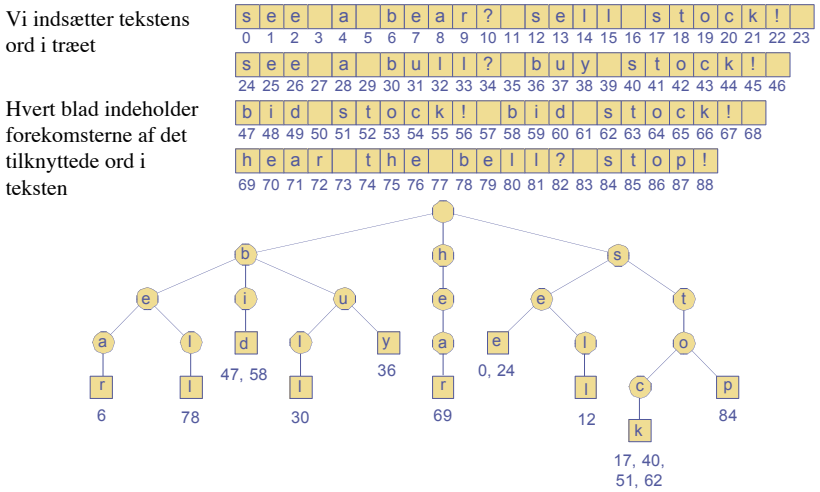
- ***n*** samlede længde af strengene i ***S***
- ***m*** længden af strengparameteren for operationen
- ***d*** størrelsen af alfabetet



Søgning af ord med et trie

Vi indsætter tekstens ord i træet

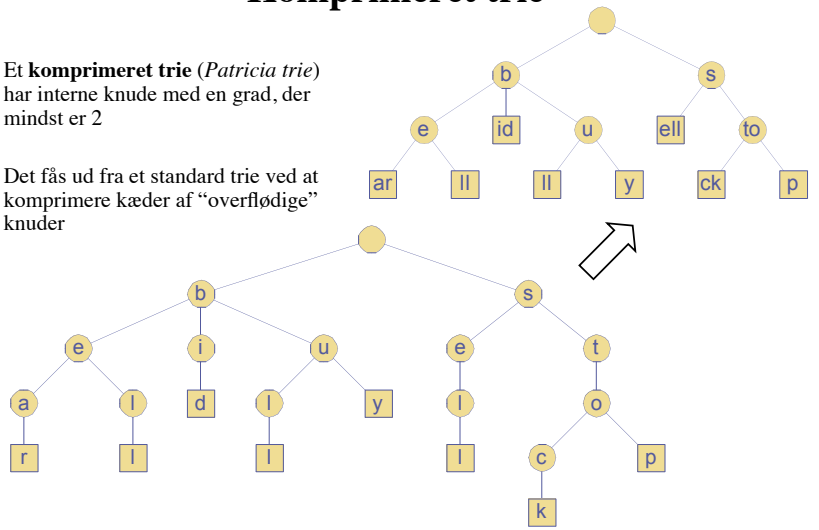
Hvert blad indeholder forekomsterne af det tilknyttede ord i teksten



Komprimeret trie

Et **komprimeret trie** (*Patricia trie*) har interne knude med en grad, der mindst er 2

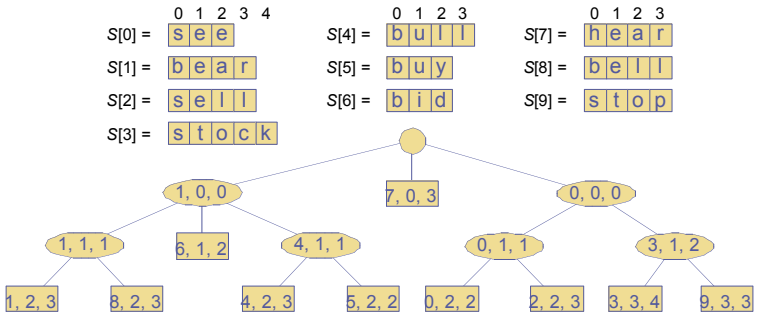
Det fås ud fra et standard trie ved at komprimere kæder af "overflødige" knuder



Kompakt repræsentation

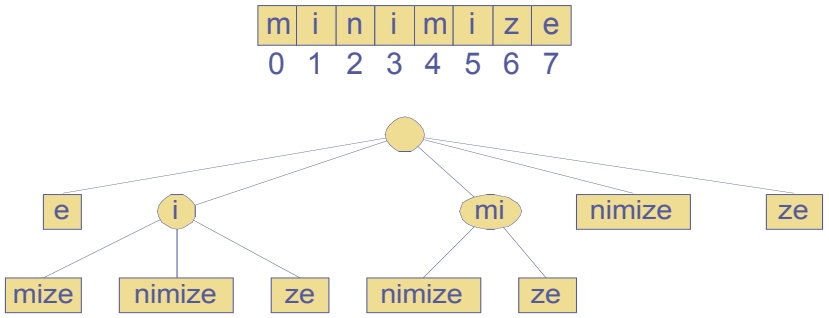
Kompakt repræsentation af et komprimeret trie for et array af strenge:

- Knuderne indeholder indeksintervaller i stedet for delstrenge
- Bruger $O(s)$ plads, hvor s er antallet af strenge i arrayet



Suffix-trie (1)

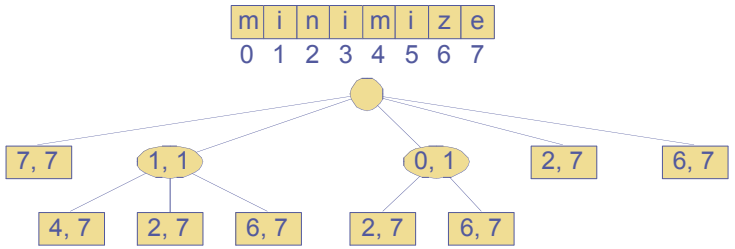
Suffix-trie'et for en streng X er det komprimerede trie for alle suffixer i X



Suffix-trie (2)

En kompakt repræsentation af suffix-trie'et for en streng X af længde n fra et alfabet af størrelse d

- bruger $O(n)$ plads
- tillader strengsøgning i X i $O(dm)$ tid, hvor m er mønsterets længde (afhænger ikke af n)



Komprimering

Filkomprimering

Komprimering reducerer størrelsen af en fil

- for at spare **plads** ved lagring
- for at spare **tid** ved transmission

Komprimering benyttes til

tekst: nogle bogstaver er hyppigere end andre

grafik: store ensartede områder

lyd: gentagne mønstre

Run-length encoding

Komprimering ved tælling af gentagelser

Komprimering af **tekst**:

Strengen

```
AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD
```

omkodes til

```
4A3BAA5B8CDABCB3A4B3CD
```

Med escape-tegn (her '\'):

```
\4A\3BAA\5B\8CDABCB\3A\4B\3CD
```

Run-length encoding er normalt ikke særlig effektiv for tekstfiler

Run-length encoding

Komprimering af (sort-hvid raster) **grafik**:

```
00000000000001111111111111000000000     13 14  9
00000000000111111111111111110000000     11 18  7
00000000111111111111111111111110000      8 24  4
00000001111111111111111111111111000      7 26  3
00000111111111111111111111111111110      5 30  1
00001111111000000000000000001111111      4  7 18 7
00001111100000000000000000000011111      4  5 22 5
00001110000000000000000000000000111      4  3 26 3
00001110000000000000000000000000111      4  3 26 3
00001110000000000000000000000000111      4  3 26 3
00001110000000000000000000000000111      4  3 26 3
00001111100000000000000000000001110      5  4 23 3 1
00000001110000000000000000000111000      7  3 20 3 3
01111111111111111111111111111111111      1 35
01111111111111111111111111111111111      1 35
01111111111111111111111111111111111      1 35
01111111111111111111111111111111111      1 35
01111111111111111111111111111111111      1 35
01100000000000000000000000000000011      1  2 31 2
```

Besparelse:

19*36 - 63*6 bit = 116 bit

svarende til 23%

Fixed-length encoding

Strengen
ABRACADABRA (11 tegn)

fylder

11 * 8 bit = 88 bit	i byte-kode
11 * 5 bit = 55 bit	i 5-bit-kode
11 * 3 bit = 33 bit	i 3-bit-kode (kun 5 forskellige bogstaver)

D forekommer kun 1 gang, mens **A** forekommer 5 gange.

Vi kan forsøge at benytte korte koder for bogstaver, der forekommer hyppigt.

Variable-length encoding

Hvis **A** = **0**, **B** = **1**, **R** = **01**, **C** = **10** og **D** = **11**, kan

ABRACADABRA

kodes som

0 1 01 0 10 0 11 0 1 01 0 (kun 15 bit)

Men denne kode kan kun afkodes (dekomprimeres), hvis der anvendes skilletegn (f.eks. et blanktegn)

Problemet skyldes, at nogle koder er præfiks for andre.
For eksempel er **A** præfiks for **R**

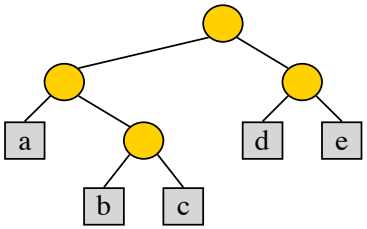
Trie til kodning (1)

En **kode** er en afbildning af ethvert tegn i alfabetet til et binært kodeord

En **prefixkode** er en binær kode, hvor intet kodeord er et prefix af noget andet kodeord

Et **trie til kodning** repræsenterer en prefixkode
- Hvert blad indeholder et tegn
- Kodeordet for et tegn fås ved at følge vejen fra roden til det blad, der indeholder tegnet (0 for et venstre barn, og 1 for et højre barn)

00	010	011	10	11
a	b	c	d	e



Trie til kodning (2)

Givet en tekststreng X ønsker vi at finde en prefixkode for tegnene i X, der giver et kort kodeord for X

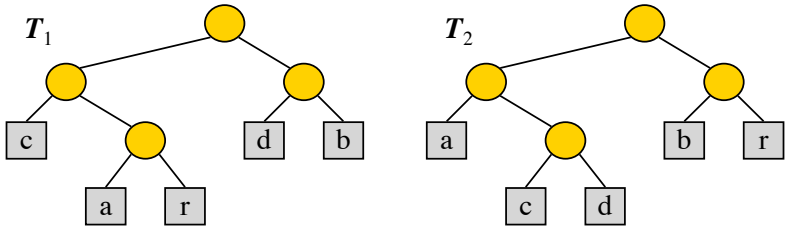
- Hyppige tegn bør have en korte kodeord
- Sjældne tegn bør have et langt kodeord

Eksempel:

X = abracadabra

T_1 koder X til 29 bits

T_2 koder X til 24 bits



Huffman's algoritme

Givet en streng X konstruerer Huffman's algoritme en prefixkode, der minimerer længden af kodeordet for X

En hob-baseret prioritetskø benyttes som hjælpedatastruktur

Algoritmen er **grådig**, men giver en optimal prefixkode

Køretid:
$O(n+d \log d)$, hvor n er længden af X, og d er antallet af forskellige tegn i X

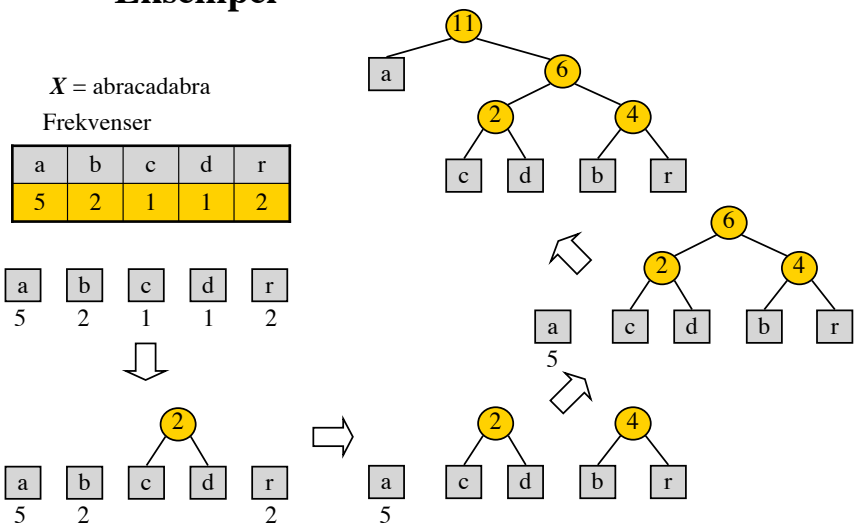
```
Algorithm HuffmanEncoding(X)
  Input string X of size n
  Output optimal encoding trie for X
  C ← distinctCharacters(X)
  computeFrequencies(C, X)
  Q ← new empty heap
  for all c ∈ C
    T ← new single-node tree storing c
    Q.insert(getFrequency(c), T)
  while Q.size() > 1
    f1 ← Q.minKey()
    T1 ← Q.removeMin()
    f2 ← Q.minKey()
    T2 ← Q.removeMin()
    T ← join(T1, T2)
    Q.insert(f1 + f2, T)
  return Q.removeMin()
```


Eksempel

X = abracadabra

Frekvenser

a	b	c	d	r
5	2	1	1	2



Problemer for Huffmans algoritme

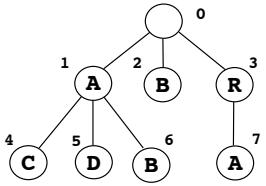
- Kodetræet skal medsendes (typisk 255 bytes)
- To gennemløb af filen (frekvensbestemmelse + kodning)
- Typisk 25% pladsreduktion, men ikke optimal

LZW-komprimering

(Lempel, Ziv og Welch, 1977)

Opbyg successivt en ordbog i form af et trie

Eksempel: **ABRACADABRA**



Kodning: **ABR1C1D1B3A**

Søgemaskiner

Søgemaskiner

En **søgemaskine** er et edb-baseret værktøj til at finde information på nettet

Arkitektur:

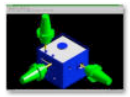
Web crawler (spider) indsamler information fra netsiderne ved at forfølge hyperhægter. Stopper aldrig

Indexer benytter den indsamlede information til opbygning af datastrukturer, der gør søgning hurtig

Søgemaskine (search engine, retriever) tillader brugere at hente information fra de opbyggede datastrukturer

- Grænseflade til forespørgsler
- Opslag i database med henvisninger til mere end 2 milliarder dokumenter (fylder mere end 4000 GB på disk)
- Rangering af dokumenter

Indeksering



Søgningen effektiviseres ved hjælp af en ordbog (et **indeks**)

Ordbogen realiseres som en **inverteret fil**, hvor hver post indeholder et ord og en samling henvisninger til de dokumenter, hvori ordet forekommer

Ordene kaldes **indekstermer**

Samlingen af henvisninger kaldes **forekomstlisten**

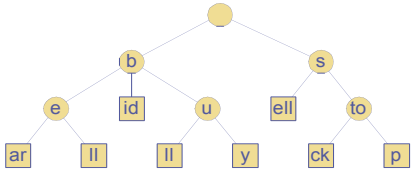
bell	10, 22, 23
buy	12
stop	1, 22
bid	24, 27
sell	27
bull	1
stock	33, 102
bear	22, 33
...	...

Implementering af en inverteret fil

Datastrukturer:

- Et array, der indeholder forekomstlisterne (i vilkårlig orden)
  Listerne holdes sorteret (for hurtigt at kunne bestemme fællesmængde)
  Gemmes på disk

- Et komprimeret trie for mængden af indekstermer, hvor hver eksterne knude lagrer indekset på forekomstlisten for den tilhørende term
  Holdes i arbejdshukommelsen



Web crawler

- Henter netsider med henblik på indeksering
- Bredde-først-søgning
  (1) Begynd med en side, *P*
  (2) Find hyperhægterne (URLs) på *P* og tilføj dem til en kø
  Overgiv *P* til indexeren
  (3) Hent *P* fra køen og gå til (2)

Spørgsmål:
- Hvorledes dybt skal der søges inden for et netsted?
- Hvor hyppigt skal sider besøges?

Rangering

Fund skal præsenteres i en rækkefølge

Hvilken?
Relevans, friskhed, popularitet, pålidelighed

Metoder til rangering af nøgleord:
- Tilstedeværelse i dokumenttitel
- Afstand fra start af dokument
- Hyppighed i dokument
- Hægtepopularitet (hvor mange sider peger på siden?)