- Palautus voidaan tehdä myös ilman matriisia M tilassa $\mathcal{O}(n)$.
  - Järjestetään BWT:n merkit vakaasti. Olkoon f näin saatu permutaatio eli $BWT[f(i)] \leq BWT[f(i+1)]$ ja yhtäsuuruuden tapauksessa $f(i) < f(i+1)$.

```
     0123456              i: 0123456
BWT: annb$aa           f(i): 4056312
                 BWT[f(i)]: $aaabnn
```

  - f on myös permutaatio M:n M':n rivien välillä.
  - Alkuperäinen teksti saadaan säännöllä

$$S[i] = BWT[f^{(i+2)}(0)] \ ,$$

    missä $f^{(k)}$ tarkoittaa f:n toistoa k kertaa: $f(f(f(\cdots f(k)\cdots)))$.
  - Myös loppuosataulukko saadaan helposti BWT:stä:

$$SA^{-1}[i] = f^{(i+1)}(0) \ ,$$

Esimerkki 4.5.7 *Kun $BWT =$ `annb$aa`, niin*

$$f(0) = 4,\ f(4) = 3,\ f(3) = 6,\ f(6) = 2,$$
$$f(2) = 5,\ f(5) = 1,\ f(1) = 0,\ f(0) = 4.$$

Siten

$$\begin{aligned} S &= BWT[3]BWT[6]BWT[2]BWT[5]BWT[1]BWT[0]BWT[4] \\ SA^{-1} &= (4,3,6,2,5,1,0) \end{aligned}$$

BW-muunnosta käytetään tiedontiivistyksessä.

- Muunnettu teksti on helpommin tiivistyvä kuin alkuperäinen, koska se sisältää paljon paikallista toistoa.
  - Esim. englanninkielisessä tekstissä `he`-alkuiset rotaatiot tulevat järjestämisen jälkeen peräkkäin, jolloin viimeisessä sarakkeessa on vastaavalla kohdalla paljon kirjaimia `t` ja `T`.
- BW-muunnokseen perustuvat myös useat tiivistetyt loppuosarakenteet.
- Tällä kurssilla tiivistettyjä loppuosarakenteita ei käsitellä tarkemmin, mutta keväällä 2006 on tulossa erikoiskurssi tiedontiivistämisestä.

4.6 Loppuosataulukon muodostaminen ajassa $\mathcal{O}(n \log n)$

U. Manber & G. Myers: Suffix Arrays: A New Method for On-line String Searches. *SIAM Journal on Computing* 22(5):935–948, 1993.

- Tarkastellaan nyt algoritmeja loppuosataulukon muodostamiseen.
- Tehtävänä on siis järjestää merkkijonon $S[0, n)$ loppuosat aakkosjärjestykseen.
- Normaali merkkijonojen järjestämisalgoritmi vie pahimmassa tapauksessa ajan $\Omega(n^2)$.
- Loppuosataulukko voidaan muodostaa $\mathcal{O}(n)$ ajassa vakioaakkostolla laskemalla ensin loppuosapuu ja siitä loppuosataulukko.
- Esitetään nyt algoritmeja, joilla loppuosataulukko voidaan muodostaa suoraan ilman loppuosapuuta ajassa $\mathcal{O}(n \log n)$.
- Algoritmit ovat aakkostoriippumattomia.

4.6.1 Karp-Miller-Rosenberg nimentätekniikka

- Merkitään $S_i^\ell = S[i \ldots \min\{i + \ell, n\})$.
- Muodostetaan taulukot $Name_\ell[0, n]$, jotka sisältävät kokonaislukuja väliltä $[0, n]$ ja toteuttavat kaikilla $i, j \in [0, n]$

$$Name_\ell[i] \leq Name_\ell[j] \iff S_i^\ell \leq S_i^\ell$$

- Taulukko $Name_\ell[0, n]$ antaa siis ℓ:n pituisten osajonojen ja ℓ:ää lyhyempien loppusosien nimennän, joka säilyttää jonojen keskinäisen järjestyksen.
- Erityisesti, kun $\ell \geq n$, saadaan kaikkien loppuosien nimentä. Siis $Name_\ell[i] = SA^{-1}[i]$, mistä saadaan helposti loppuosataulukko.
- Yllä oleva nimennän määritelmä ei ole yksikäsitteinen, mutta nimeksi kannattaa valita

$$Name_\ell[i] = |\{j \in [0, n] \mid S_j^\ell < S_i^\ell\}|$$

- Nimet on helppo laskea, kun jonojen S_i^ℓ, $0 \le i \le n$, aakkosjärjestys on annettu.

- Taulukko $Name_1[0, n]$ voidaan laskea järjestämällä yhden pituiset jonot, siis jonon S merkit, ajassa $\mathcal{O}(n \log n)$.

- Taulukko $Name_{2\ell}[0, n]$ voidaan muodostaa taulukosta $Name_\ell[0, n]$ ajassa $\mathcal{O}(n)$ kahdentamalla:

  - Olkoon $Pair_{2\ell}[i] = (Name_\ell[i], Name_\ell[i+\ell])$, missä $Name_\ell[j] = 0$, jos $j \ge n$.
  - Arvo $Pair_{2\ell}[i]$ voidaan nähdä kahden mittaisena merkkijonona aakkostossa $[0, n]$ ja se toteuttaa yllämainitun järjestysehdon:

$$Pair_{2\ell}[i] \le Pair_{2\ell}[j] \Longleftrightarrow S_i^{2\ell} \le S_j^{2\ell}$$

  - Siis $Name_{2\ell}[0, n]$ voidaan laskea helposti, kun parit $Pair_{2\ell}[0, n]$ on ensin järjestetty.
  - Koska aakkosto on $[0, n]$ ja jonot vakiopituisia, voidaan järjestäminen tehdä ajassa $O(n)$ käyttämällä esimerkiksi lopusta alkuun kantalukujärjestämistä.

- Loppuosataulukko (tai sen käänteistaulukko) saadaan toistamalla kahdentamisaskelta $\lceil \log n \rceil$ kertaa eli laskemalla $Name_1$, $Name_2$, $Name_4$, $Name_8$, ...

Esimerkki 4.6.1 *Käänteisen loppuosataulukon laskenta jonolle* `banana`.

$Name_1$:		$Name_2$:		$Name_4$:	
4	b	4	ba	4	bana
1	a	2	an	3	anan
5	n	5	na	6	nana
1	a	2	an	2	ana
5	n	5	na	5	na
1	a	1	a	1	a
0		0		0	

$Name_8 = SA^{-1}$:

4	banana
3	anana
6	nana
2	ana
5	na
1	a
0	

- Nimitaulukoista ei tarvitse säilyttää kuin aina viimeksi laskettu. Tilavaatimus on tällöin $\mathcal{O}(n)$.

Lause 4.6.2 *Merkkijonon $S[0, n]$ loppuosataulukko voidaan laskea ajassa $\mathcal{O}(n \log n)$ ja tilassa $\mathcal{O}(n)$.*

- Karp-Miller-Rosenberg nimentätekniikkaa voidaan käyttää myös muihin ongelmiin.
- Loppuosataulukon muodostamiseen sitä sovelsivat ensin Manber ja Myers [?]. Heidän algoritminsa tarvitsee vain kaksi kokonaislukutaulukkoa ja kaksi bittivektoria.
- Kahdentaminen voidaan lopettaa heti, kun kaikki nimet ovat erilaisia. Tällöin aikavaatimus on $\mathcal{O}(n \log k)$, missä k on pisimmän toistuvan osajonon pituus.

Larsson-Sadakane-algoritmi

- Käytännössä tehokkaimman toteutuksen kahdentamistekniikalle ovat esittäneet Larsson ja Sadakane.
- Olkoon $N_\ell(p) = \{i \mid Name_\ell[i] = p\}$. Tällaisella samannimisten ryhmällä on seuraavat ominaisuudet:
    1. Jos $|N_\ell(p)| = k$, niin jokaisella $i \in N_\ell(p)$, $Name_{2\ell}[i], SA^{-1}[i] \in [Name_\ell[i], Name_\ell[i] + k)$.
    2. Erityisesti, jos $N_\ell(p) = \{i\}$ eli sen koko on yksi, niin $Name_{2\ell}[i], SA^{-1}[i] = Name_\ell[i]$.

- Algoritmin ideana on suorittaa kahdentaminen erikseen kullekin ryhmälle $N_\ell(p)$.
  - Ryhmän sisällä parien $Pair_{2\ell}[i]$ ensimmäinen komponentti on sama. Ne tarvitsee siis vain järjestää toisen komponentin mukaan.
  - Ominaisuuden 1 ansiosta uudet nimet $Name_{2\ell}[i]$ voidaan laskea muista ryhmistä riippumatta.
  - Jos ryhmän koko on yksi, sille ei tarvitse tehdä mitään.
- Järjestäminen toisen komponentin mukaan tehdään käyttäen ternääripikajärjestämistä. (Laskemisjärjestämistä ei voi käyttää, koska aakkoston koko voi olla paljon suurempi kuin ryhmän koko.)
- k:n kokoiselle ryhmälle aikaa voi kulua $\mathcal{O}(k \log k)$. Tämä näyttäisi johtavan pahimman tapauksen aikavaatimukseen $\mathcal{O}(n \log^2 n)$.
- Voidaan kuitenkin osoittaa, että koko algoritmin aikavaatimus on $\mathcal{O}(n \log n)$.
  - Voidaan ajatella, että ollaan järjestämässä jonoja $Name_1[i]Name_1[i+1]Name_2[i+2]Name_4[i+4]Name_8[i+8]\cdots$ käyttäen merkkijonopikajärjestämistä. Se vain tehdään leveyseikä syvyyssuuntaisesti.
  - Leveyssuuntaisuus ei muuta analyysia. Aikavaatimus on $\mathcal{O}(n \log n + DP)$, missä yksilöivien alkuosien kokonaispituus DP on nyt $\mathcal{O}(n \log n)$.

4.7 Loppuosataulukon muodostaminen ajassa $\mathcal{O}(n)$

- Esitetään nyt algoritmi, joka laskee loppuosataulukon ajassa $\mathcal{O}(n)$, kun aakkosto on $[1, n]$.
- Jos aakkosto on joku muu kuin $[1, n]$, voidaan kukin merkki $S[i]$ korvata nimellä $Name_1[i]$, jolloin päästään aakkostoon $[1, n]$ muuttamatta loppuosien keskinäistä järjestystä.
- Siten millä tahansa aakkostolla jonon S loppuosat voidaan järjestää samassa ajassa kuin S:n merkit.

Osarekursio

- Tässä esitettävä algoritmi on yksi kolmesta ensimmäisestä lineaariaikaisesta loppuosataulukon muodostusalgoritmista. Ne kaikki julkaistiin kesäkuussa 2003.
  - Kim, Sim, Park & Park., CPM ’03
  - Ko & Aluru, CPM ’03
  - Kärkkäinen & Sanders, ICALP ’03
- Kaikilla on sama rakenne:
  0. Jaa loppuosat kahteen ryhmään S_A ja S_B.
  1. Järjestä S_A. Tämä tapahtuu palauttamalla ongelma loppuosataulukon muodotukseen jonolle, jonka pituus on $|A|$.
  2. Järjestä S_B käyttäen S_A:n järjestystä.
  3. Lomita S_A ja S_B.
- Rekursio poislukien kaikki pystytään tekemään lineaarisessa ajassa.
- Kaikissa algoritmeissa $|A| \leq 2n/3$. Siten aikavaatimus on $T(n) = \mathcal{O}(n) + T(2n/3) = \mathcal{O}(n)$.
- Olennaista on, että S_B:tä ei järjestetä rekursiivisesti.
- Sama osarekursiivinen rakenne esiintyi jo Farachin loppuosapuun muodostusalgoritmissa. Kimin ja kumppaneiden algoritmi on hyvin samanlainen.
  - $A = \{i \mid i \text{ on pariton}\}$.
  - Lomitus on hyvin monimutkaista.
- Ko-Aluru-algoritmissa A on pienempi joukoista $\{i \mid S_i < S_{i+1}\}$ ja $\{i \mid S_i > S_{i+1}\}$.

Kärkkäinen-Sanders-algoritmi

- Syötteenä on jono $S[0, n)$ aakkostossa $[1, n]$. Lisäksi oletetaan, että $S[n, n+3) = 000$.

Esimerkki 4.7.1

$$S[0,n) = \begin{array}{cccccccccccc} \scriptstyle 0 & \scriptstyle 1 & \scriptstyle 2 & \scriptstyle 3 & \scriptstyle 4 & \scriptstyle 5 & \scriptstyle 6 & \scriptstyle 7 & \scriptstyle 8 & \scriptstyle 9 & \scriptstyle 10 & \scriptstyle 11 \\ \mathtt{y} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{a} & \mathtt{b} & \mathtt{b} & \mathtt{a} & \mathtt{d} & \mathtt{o} \end{array}$$

Askel 0: Muodosta S_A.

- Määritellään kaikilla $k = 0, 1, 2$, $C_k = \{i \in [0, n] \mid i \bmod 3 = k\}$. Olkoon $A = C_1 \cup C_2$ ja $B = C_0$.

  Esimerkki 4.7.2 $B = C_0 = \{0, 3, 6, 9, 12\}$, $C_1 = \{1, 4, 7, 10\}$, $C_2 = \{2, 5, 8, 11\}$, *eli* $A = \{1, 4, 7, 10, 2, 5, 8, 11\}$.

Askel 1: Järjestä S_A.

- Muodosta jonot

  $$R_k = \Big[S[k \dots k+3)\Big]\Big[S[k+3 \dots k+6)\Big] \dots \Big[S[\max C_k \dots \max C_k + 3)\Big]$$

  $k = 1, 2$, joiden merkit ovat kolmen merkin pituisia S:n osajonoja.

- Olkoon R jonojen R_1 ja R_2 katenaatio. Sen loppuosat vastaavat S_A:n loppuosia.

  Esimerkki 4.7.3 $R = [\mathtt{abb}][\mathtt{ada}][\mathtt{bba}][\mathtt{do0}][\mathtt{bba}][\mathtt{dab}][\mathtt{bad}][\mathtt{o00}]$.

- Korvataan R:n merkit KMR-tekniikan mukaisilla nimillä. Tämä vaatii R:n merkkien järjestämistä, mikä voidaan tehdä lopusta alkuun kantalukujärjestämisellä lineaarisessa ajassa. Olkoon R' muunnettu jono.

- Muodostetaan rekursiivisella kutsulla R:n loppuosataulukko, mistä saadaan S_A:n järjestys. Rekursiota ei tarvita, jos kaikki R':n merkit ovat erilaiset.

  Esimerkki 4.7.4 $R' = (1, 2, 4, 6, 4, 5, 3, 7)$ *and* $SA_{R'} = (8, 0, 1, 6, 4, 2, 5, 3, 7)$.

- Olkoon $rank(i)$, $i \in A$, loppuosan S_i järjestysluku joukossa S_A. Lisäksi määritellään $rank(n+1) = rank(n+2) = 0$. Kun $i \in C_0$, $rank(i) = \perp$.

  Esimerkki 4.7.5

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$rank(S_i)$	$\perp$	1	4	$\perp$	2	6	$\perp$	5	3	$\perp$	7	8	$\perp$	0	0

Askel 2: Järjestä S_B.

- Muodostetaan jokaiselle $S_i \in S_B$ pari $(t_i, rank(S_{i+1}))$. Huomaa, että $rank(S_{i+1})$ on aina määritelty.

- Selvästi kaikilla $i, j \in B$,

$$S_i \leq S_j \iff (S[i], rank(S_{i+1})) \leq (S[j], rank(S_{j+1})) \ .$$

- Parit $(S[i], rank(S_{i+1}))$ järjestetään lopusta alkuun kantalukujärjestämisellä.

Esimerkki 4.7.6 *$S_{12} < S_6 < S_9 < S_3 < S_0$ koska $(\mathtt{0}, 0) < (\mathtt{a}, 5) < (\mathtt{a}, 7) < (\mathtt{b}, 2) < (\mathtt{y}, 1)$.*

Askel 3: Lomita.

- Lomitetaan S_A ja S_B normaalilla vertailuihin perustuvalla lomitusalgoritmilla.

- Kahden loppuosan $S_i \in S_A$ ja $S_j \in S_B$ vertailussa erotetaan kaksi tapausta:

$$\begin{array}{ll} i \in C_1: & S_i \leq S_j \iff (S[i], rank(i+1)) \leq (S[j], rank(j+1)) \\ i \in C_2: & S_i \leq S_j \iff (S[i], S[i+1], rank(i+2)) \leq (S[j], S[j+1], rank(j+2)) \end{array}$$

Esimerkki 4.7.7 *$S_1 < S_6$ koska $(\mathtt{a}, 4) < (\mathtt{a}, 5)$ ja $S_3 < S_8$ koska $(\mathtt{b}, \mathtt{a}, 6) < (\mathtt{b}, \mathtt{a}, 7)$.*

Toteutus

- Alla on algoritmin täydellinen C++-kielinen toteutus.

- Toteutus ei ole tehokkain mahdollinen; pyrkimys on lyhyyteen (50 riviä poislukien tyhjät rivit, kommentit, ja rivit, joilla on vain aaltosulku).

```
inline bool leq(int a1, int a2,   int b1, int b2) // lexicographic order
{ return(a1 < b1 || a1 == b1 && a2 <= b2); }                // for pairs
inline bool leq(int a1, int a2, int a3,   int b1, int b2, int b3)
{ return(a1 < b1 || a1 == b1 && leq(a2,a3, b2,b3)); }     // and triples
```

```
// stably sort a[0..n-1] to b[0..n-1] with keys in 0..K from r
static void radixPass(int* a, int* b, int* r, int n, int K)
{ // count occurrences
  int* c = new int[K + 1];                          // counter array
  for (int i = 0;  i <= K;  i++) c[i] = 0;         // reset counters
  for (int i = 0;  i < n;  i++) c[r[a[i]]]++;   // count occurrences
  for (int i = 0, sum = 0;  i <= K;  i++)   // exclusive prefix sums
  {  int t = c[i];  c[i] = sum;  sum += t; }
  for (int i = 0;  i < n;  i++) b[c[r[a[i]]]++] = a[i];      // sort
  delete [] c;
}

// find the suffix array SA of T[0..n-1] in {1..K}^n
// require T[n]=T[n+1]=T[n+2]=0, n>=2
void suffixArray(int* T, int* SA, int n, int K) {
  int n0=(n+2)/3, n1=(n+1)/3, n2=n/3, n02=n0+n2;
  int* R  = new int[n02 + 3];  R[n02]= R[n02+1]= R[n02+2]=0;
  int* SA12 = new int[n02 + 3]; SA12[n02]=SA12[n02+1]=SA12[n02+2]=0;
  int* R0   = new int[n0];
  int* SA0  = new int[n0];

  //******* Step 0: Construct sample ********
  // generate positions of mod 1 and mod 2 suffixes
  // the "+(n0-n1)" adds a dummy mod 1 suffix if n%3 == 1
  for (int i=0, j=0;  i < n+(n0-n1);  i++) if (i%3 != 0) R[j++] = i;

  //******* Step 1: Sort sample suffixes ********
  // lsb radix sort the mod 1 and mod 2 triples
  radixPass(R , SA12, T+2, n02, K);
  radixPass(SA12, R , T+1, n02, K);
  radixPass(R , SA12, T  , n02, K);

  // find lexicographic names of triples and
  // write them to correct places in R
  int name = 0, c0 = -1, c1 = -1, c2 = -1;
  for (int i = 0;  i < n02;  i++) {
    if (T[SA12[i]] != c0 || T[SA12[i]+1] != c1 || T[SA12[i]+2] != c2)
    {  name++;  c0 = T[SA12[i]];  c1 = T[SA12[i]+1];  c2 = T[SA12[i]+2]; }
    if (SA12[i] % 3 == 1) { R[SA12[i]/3]      = name; } // write to R1
    else                  { R[SA12[i]/3 + n0] = name; } // write to R2
  }
```

```
  // recurse if names are not yet unique
  if (name < n02) {
    suffixArray(R, SA12, n02, name);
    // store unique names in R using the suffix array
    for (int i = 0;  i < n02;  i++) R[SA12[i]] = i + 1;
  } else // generate the suffix array of R directly
    for (int i = 0;  i < n02;  i++) SA12[R[i] - 1] = i;

  //******* Step 2: Sort nonsample suffixes ********
  // stably sort the mod 0 suffixes from SA12 by their first character
  for (int i=0, j=0;  i < n02;  i++) if (SA12[i] < n0) R0[j++] = 3*SA12[i];
  radixPass(R0, SA0, T, n0, K);

  //******* Step 3: Merge ********
  // merge sorted SA0 suffixes and sorted SA12 suffixes
  for (int p=0,  t=n0-n1,  k=0;  k < n;  k++) {
#define GetI() (SA12[t] < n0 ? SA12[t] * 3 + 1 : (SA12[t] - n0) * 3 + 2)
    int i = GetI(); // pos of current offset 12 suffix
    int j = SA0[p]; // pos of current offset 0  suffix
    if (SA12[t] < n0 ? // different compares for mod 1 and mod 2 suffixes
        leq(T[i],       R[SA12[t] + n0], T[j],       R[j/3]) :
        leq(T[i],T[i+1],R[SA12[t]-n0+1], T[j],T[j+1],R[j/3+n0]))
    {       // suffix from SA12 is smaller
      SA[k] = i;  t++;
      if (t == n02) // done --- only SA0 suffixes left
        for (k++;  p < n0;  p++, k++) SA[k] = SA0[p];
    } else { // suffix from SA0  is smaller
      SA[k] = j;  p++;
      if (p == n0)  // done --- only SA12 suffixes left
        for (k++;  t < n02;  t++, k++) SA[k] = GetI();
    }
  }
  delete [] R; delete [] SA12; delete [] SA0; delete [] R0;
}
```

4.8 Eropeiteotos

- Loppuosapuun ja -taulukon nopea muodostaminen perustuu oleellisesti siihen, että järjestetään kaikki loppuosat. Pienemmän osajoukon järjestämien on hankalampaa.
  - Koko loppuosataulukko vie runsaasti tilaa, pienellä aakkostolla paljon enemmän kuin itse teksti. Esim. DNA aakkostolla, jonka koko on 4, yksi merkki mahtuu kahteen bittiin, mutta yksi alkukohdan osoitin vie tyypillisesti 32 bittiä.
  - Järjestäminen tavallisella merkkijonoalgoritmilla voi olla hidasta. m:n loppuosan järjestäminen n:n pituiselle tekstille voi viedä ajan $\mathcal{O}(mn)$.
- Esitetään nyt tekniikka, jolla loppuosien järjestämistä ja muita vertailuihin perustuvia algoritmeja ja tietorakenteita voidaan tehostaa loppuosataulukkoa pienemmässä tilassa.
- Sama tekniikka on myös Kärkkäinen-Sanders algoritmin taustalla.

Perusidea

- Valitaan sopiva osajoukko tai *otos* loppuosia S_A, $A \in [0, n]$, ja järjestetään ne. Olkoon $rank(i)$, $i \in A$, loppuosan S_i järjestysluku joukossa S_A.
- Kun halutaan verrata kahta loppuosas S_i ja S_j, valitaan ℓ siten, että $i + \ell$ ja $j + \ell$ ovat otoksessa A, tehdään vertailu:

  $$S_i \leq S_j \iff S[i \dots i + \ell)rank(i + \ell) \leq S[j \dots j + \ell)rank(j + \ell)$$

  Vertailun aikavaatimus on $\mathcal{O}(\ell)$.
- Ollakseen *sopiva* otoksen S_A täytyy toteuttaa kaksi ehtoa:
  1. Se voidaan järjestää nopeasti.
  2. Se mahdollistaa minkä tahansa loppuosien nopean vertailun, eli kaikilla i, j on olemassa pieni ℓ, jolla $i + \ell, j + \ell \in A$.

Eropeite

- Järjestäminen voidaan tehdä nopeasti, jos otos on jaksollinen.
  - Olkoon v jaksonpituus ja $D \in [0, v)$ otosjakso. Olkoon otos $A = \{i \in [0, n] \mid i \bmod v = D\}$.
  - Järjestäminen tehdään kuten Kärkkäinen-Sanders-algoritmin askeleessa 1. Muodostetaan jonot R_k, $k \in D$, joiden merkit ovat v:n pituisia osajonoja, katenoidaan ja järjestetään katenaation (kaikki) loppuosat esim. Kärkkäinen-Sanders-algoritmilla.
  - Eniten aikaa kuluu katenaation merkkien järjestämiseen (jotta ne voidaan korvata KMR-nimillä): $\mathcal{O}(n|D|)$.
- Nopea vertailu saavutetaan, kun otosjaksoksi D valitaan *eropeite* (difference cover).
  - D on eropeite jaksonpituudelle v, jos
$$\{(i - j) \bmod v \mid i, j \in D\} = [0, v) \ .$$

Esimerkki 4.8.1 $\{2, 4, 5\}$ *on eropeite jaksonpituudelle* 7.

	$i - j$
0	$2 - 2$
1	$5 - 4$
2	$4 - 2$
4	$2 - 5$
5	$2 - 4$
6	$4 - 5$

Lemma 4.8.2 *Jos D on eropeite jaksonpituudelle v, niin kaikilla kokonaisluvuilla i, j on olemessa $\ell \in [0, v)$, jolla $(i + \ell) \bmod v, (j + \ell) \bmod v \in D$.*

Todistus.

- Määritelmän mukaan on olemassa $i', j' \in D$, joilla $i' - j' \equiv i - j \pmod{v}$. Olkoon $\ell = (i' - i) \bmod v$.
- Tällöin
$$\begin{aligned} i + \ell &\equiv i' \in D \pmod{v} \\ j + \ell &\equiv i' - (i - j) \equiv j' \in D \pmod{v} \ . \end{aligned}$$

□

- Eropeiteotos mahdollistaa siis kahden loppuosa vertailun aina ajassa $\mathcal{O}(v)$.
- Jos kahdella loppuosalla tiedetään jo olevan v:n pituinen yhteinen alkuosa, niiden vertailu voidaan tehdä vakioajasa.
- Kaikille v on olemassa eropeite, jonka koko on $\mathcal{O}(\sqrt{v})$.
  - Colbourn and Ling [?] kuvaavat yksinkertaisen algoritmin, joka löytää ajassa $\mathcal{O}(\sqrt{v})$ eropeitteen, jonka koko on enintään $\sqrt{1.5v}+6$.
- Eropeiteotoksen koko on siis $\mathcal{O}(n|D|/v) = \mathcal{O}(n/\sqrt{v})$.

Sovelluksia

- Loppuosajoukon S_B, $|B| = m$, järjestäminen.
  - Muodostetaan eropeiteotos ajassa $\mathcal{O}(n\sqrt{v})$.
  - Järjestetään S_B ensin v:n ensimmäisen merkin perusteella ajassa $\mathcal{O}(vm)$.
  - Järjestetään loppuosat, joilla on sama v:n pituinen yhteinen alkuosa ajassa $\mathcal{O}(m \log m)$.
  - Kokonaisaikavaatimus on $\mathcal{O}(n\sqrt{v} + vm + m \log m)$ ja tilavaatimus $\mathcal{O}(m + n/\sqrt{v})$.
  - Jos esim. $m = n/\log n$ ja valitaan $v = \log n$, saadaan aikavaatimus $\mathcal{O}(m \log m) = \mathcal{O}(n)$ ja tilavaatimus $\mathcal{O}(n/\sqrt{\log n})$.
- Muita algoritmeja ja tietorakenteita
  - Yhdistämällä merkkijonotekniikoita, esim. lcp-vertailutekniikkaa, ja eropeiteotostekniikkaa saadaan tehokkaita algoritmeja ja tietorakenteita.
  - Esim. binäärihaku ajassa $\mathcal{O}(\log m + v)$.
- Loppuosataulukon muodostaminen.

- Kärkkäinen-Sanders-algoritmi on varsin samanlainen kuin aikasempi Farachin algoritmi. Olennaisin ero huomattavasti yksinkertaisempi lomitus. Se perustuu siihen, että $\{1, 2\}$ on eropeite jaksonpituudelle 3.
- Eropeiteotoksen avulla loppuosataulukko voidaan myös muodostaa käyttäen alilineaarinen määrä lisätilaa itse tekstin ja loppuosataulukon lisäksi. Kaikki aiemmin kuvatut algoritmit vaativat $\mathcal{O}(n)$ lisätilan.

Luku 5

Tiedon tiivistäminen

T. Bell, J. Cleary, ja I. Witten: *Text Compression*. Prentice-Hall, 1990.

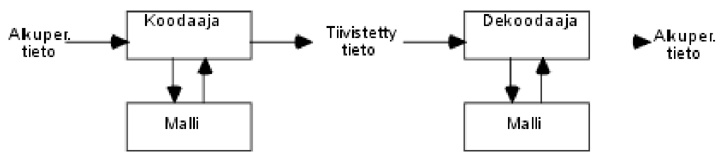
T. Bell, J. Cleary, ja I. Witten: Modeling for Text Compression, *Computing Surveys*, 21(4):557–591, 1989.

5.1 Johdanto

- Tiedon tiivistäminen on redundanssin eliminoimista.
- Tarpeet
  - tiedonsiirto
  - talletustila
    - faksi: 1.5×10^6 bit/sivu
    - värikuva (24-bit, 600 dpi): 7.5×10^8 bit/sivu
    - tv-kuva (PAL): 2.7×10^8 bit/s
    - CD-ääni: 1.5×10^6 bit/s
  - Sivuvaikutus: tiedon salaus
- Tiivistys/purku
  - Tiivistyksessä tieto koodataan vähemmän tilaa vaativaan muotoon (bittijonoksi).
  - Purussa tiedon alkuperäinen esitysmuoto palautetaan.

- Hukkaava/hukkaamaton tiivistys
  - *Hukkaamaton* (lossless) tiivistysmenetelmä tuottaa purussa täsmälleen alkuperäisen tiedon.
  - *Hukkaava* (lossy) tiivistysmenetelmä tuottaa purussa likimääräisesti alkuperäisen tiedon.
    * Sopii sinne, missä ei tarvita täydellistä tarkkuutta, esim. kuvien ja äänen käsittelyssä.
    * Pyritään eroon merkityksettömästä informaatiosta, esim. kohina tai ihmissilmälle/-korvalle erottumattomat yksityiskohdat.
    * Purku tapahtuu deterministisesti (hukkaus tapahtuu tiivistyksessä).
    * Hukkaavat menetelmät ovat hukkaamattomia huomattavasti tehokkaampia.
- Malli/koodaus
  - Alkuperäinen tieto (sanoma, merkkijono, tiedosto) analysoidaan *(tiivistys)mallia* käyttäen. Mallin perusteella valitaan koodattavat yksiköt eli tieto jaetaan peräkkäisiin osajonoihin.
    * Valinta tapahtuu esiintymistiheyden, pituuden tms. perusteella.
    * Jos tiivistettävälle tiedolle ei ole annettu mallia etukäteen, tarvitaan *mallintaja*, joka analysoi syöttöjonon ja muodostaa mallin.
  - Malliin liittyy *koodaustaulu* (dictionary, code table), joka sisältää koodattavien yksiköiden ja koodisanojen vastaavuudet ja jota *koodaaja* käyttää.
    * Koodisanat ovat tavallisesti bittijonoja.
    * Jokainen jono koodisanoja pitää olla yksikäsitteisesti tulkittavissa.
    * Usein vaaditaan, että koodi on *välittömästi tulkittavissa* (instantaneous) eli että koodisana tunnistetaan heti kun sen viimeinen bitti on kohdattu. Tällöin koodi on ns. *alkuosakoodi* (prefix code): koodisana ei ole minkään toisen koodisanan alkuosa.
    * Kooditaulun on oltava *täydellinen* (complete) siinä mielessä, että sillä on pystyttävä koodaamaan kaikki mahdolliset merkkijonot.

  - Esim. Tekstiä voidaan tiivistää antamalla kullekin merkkiparille koodi.
    * Malli: tekstin jako kahden merkin osajonoiksi
    * Koodaus: osajonojen korvaus niitä vastaavilla koodeilla.
  - Tyypillisesti jokaiseen mallin tilaan liittyy (koodattavien yksiköiden) *seuraajajakauma*.
  - Mallien tyyppejä:
    * Kontekstimalli: m edeltävää merkkiä määräävät seuraajajakauman.
    * Automaattimalli: DFA:n tila määrää seuraajajakauman.
    * Kielioppimalli: kielioppi rajoittaa seuraajajakaumaa.
  - Purussa *dekoodaaja* muuntaa bittijonon symboleiksi, jotka muunnetaan alkuperäisiksi merkeiksi samaa mallia käyttäen.



- On line/off line
  - *On line* -tiivistys: yksi läpikäynti
  - *Off line* -tiivistys: monta läpikäyntiä
- Kiinteä/mukautuva tiivistys
  - *Kiinteä* (staattinen) menetelmä: koodaustaulu ei muutu tiivistyksen aikana.
  - *Mukautuva* (oppiva, adaptive) menetelmä: koodaustaulu muuttuu tiivistysprosessin aikana.
    * *Esiintymättömyysongelma* (zero-frequency problem): sanomassa esiintymättömille symboleille ei voida antaa todennäköisyyttä nolla.
  - *Puoliadaptiivinen* (semiadaptive) menetelmä toimii tiivistyksessä kaksivaiheisesti (off line): ensimmäisessä vaiheessa muodostetaan koodaustaulu, jota käytetään varsinaisessa tiivistyksessä toisessa vaiheessa. Koodaustaulu on osa tiivistettyä tietoa.

- Kiinteän- ja vaihtuvanmittainen koodi
  - Koodataan esim. merkit 0–9.
  - *Kiinteänmittainen* koodi
    - Valitaan koodit $0000, 0001, \ldots, 1001$, jolloin merkkijonon 280198 koodin pituus on 24 bittiä.
  - Vaihtuvanmittainen koodi (pitää olla alkuosakoodi)
    - Valitaan 3-bittiset koodit $000, \ldots, 101$ (0-5) ja 4-bittiset koodit $1100, 1101, 1110, 1111$ (6-9). Merkkijonon 280198 koodin pituus on 21 bittiä.
- Arviointikriteerejä
  - Tiivistystulos
    - Merkinnät:
      - s: alkuperäinen esitysmuoto bitteinä
      - t: tiivistetyn muodon pituus bitteinä
    - Tiivistyssuhde: s/t (kirjallisuudessa esiintyy useita variaatioita)
    - Tiivistyshyöty: $1 - t/s$
    - Tärkein mitta (tekstin tiivistyksessä): bittiä/merkki (bits/c)
    - Vertailu: 3 bit/c

tiivistyssuhde (8-bit)	2,7
tiivistyssuhde (7-bit)	2,3
tiivistyshyöty	63 %

  - Tiivistysaika ja -tila
  - Purkuaika ja -tila
  - Virheensieto

5.2 Informaatioteoriaa

Informaatio

- Tapahtuman E sisältämälle informaatiolle $I(E)$ tulisi olla voimassa seuraavat ehdot:

a) informaation määrä on aina ei-negatiivinen,

b) kahden toisistaan riippumattoman tapahtuman antama informaatio on erillisten tapahtumien informaatioiden summa,

c) mitä epätodennäköisempi tapahtuma, sitä suurempi informaatio siitä saadaan.

- Määritellään $I(E) = -\log p_E$, missä p_E on tapahtuman E todennäköisyys.
- Informaation yksikkönä käytetään bittiä, jos logaritmin kantalukuna on 2.

5.2.1 Entropia

- Aakkoston $\Sigma = \{a_1, \ldots, a_\sigma\}$ *keskimääräinen informaatio, entropia*:

$$H(\Sigma) = -\sum_{i=1}^{\sigma} p(a_i) \log(p(a_i)), \tag{5.1}$$

  jos symbolien välillä ei ole riippuvuuksia.

- Entropia antaa alarajan koodatun tiedon keskimääräiselle koodipituudelle.
- *Entropiaa parempaan tiivistykseen ei voi päästä* (keskimäärin)!
  - Aakkoston $\Sigma = \{a_1, \ldots, a_\sigma\}$ merkkijonoa S, $|S| = n$, vastaavan tulosjonon pituus on keskimäärin vähintään $nH(A)$, koska tiivistetyn tiedon täytyy sisältää alkuperäisen jonon kaikki informaatio.
- Entropia riippuu aina käytetystä mallista, ts. koodattavien symbolien valinnasta.
  - Mahdollisimman heikko malli on sellainen, missä symbolien todennäköisyydet ovat samat. Merkitään tätä entropiaa H_{-1}:llä: $H_{-1}(\Sigma) = -\sum_{i=1}^{\sigma} p(a_i) \log(p(a_i)) = -\sigma(1/\sigma) \log(1/\sigma) = \log \sigma$.