Enumerative Data Compression with Non-uniquely Decodable Codes

***M. Oğuzhan Külekci*[1]**, Yasin Öztürk[1], Elif Altunok[1], Can Yılmaz Altınığne[2]

kulekci@itu.edu.tr

[1] Informatics Institute, Istanbul Technical University, Turkey
[2] School of Computer and Information Sciences, EPFL, Switzerland

Prague Stringology Conference 2020
PSC’20, 1 September 2020, Praque

Objective

Data compression is to represent data with less number of bits.

Almost all data compression literature is based on prefix codes.

How about the non-prefix-free codes?

Are they that much terrible to use ?

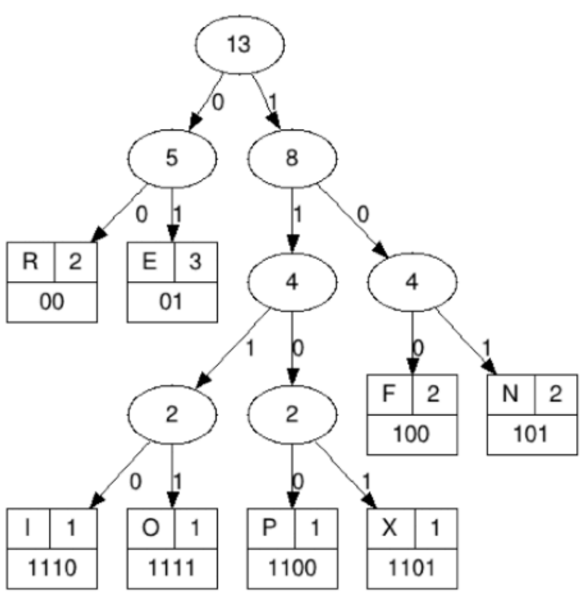
Can they serve for some purpose?

… ?????

PREFIX CODES

Data Compression : Represent data with less bits

Prefix-Free Codes: None of the codeword is a prefix of other, e.g., Huffman



$T =$ NONPREFIXFREE

$$\begin{array}{r|c|c|c|c|c|c|c|c|}
\Sigma: & \mathrm{E} & \mathrm{R} & \mathrm{F} & \mathrm{N} & \mathrm{I} & \mathrm{O} & \mathrm{P} & \mathrm{X} \\
F: & 3 & 2 & 2 & 2 & 1 & 1 & 1 & 1 \\
W: & 01 & 00 & 100 & 101 & 1110 & 1111 & 1100 & 1101
\end{array}$$

$$Huffman(T) = \\ = \underline{101}1111\underline{101}1100\underline{00}01\underline{100}1110\underline{1101}100\underline{00}01\underline{01}$$

	Codeword
E	01
R	00
F	100
N	101
I	1110
O	1111
P	1100
X	1101

Prefix-codes are uniquely decodable and require no extra effort to mark the codeword boundaries !

NON-PREFIX-FREE (NPF) CODES

Non-Prefix-Free (NPF) or Not-Uniquely Decodable Codes:
The assigned codewords can be a prefix of others.

T =	N	O	N	P	R	E	F	I	X	F	R	E	E
NPF(T) =	01	11	01	000	1	0	00	10	001	00	1	0	0

	Codeword
E	0
R	1
F	00
N	01
I	10
O	11
P	000
X	001

NPF codes are NOT uniquely decodable and REQUIRE extra data structures to mark the codeword boundaries !

Surely the average codeword length is better than prefix-codes. However, when augmented with extra space to mark codeword boundaries, they get worse !

How to efficiently mark codeword boundaries in NPF codes ?

Σ:	E	R	F	N	I	O	P	X
W :	0	1	00	01	10	11	000	001

T =	N	O	N	P	R	E	F	I	X	F	R	E	E
NPF(T) =	01	11	01	000	1	0	00	10	001	00	1	0	0
L =	2	2	2	3	1	1	2	2	3	2	1	1	1

1. Use a separate bitmap: **R/S dictionaries**

$$\underline{01}11\underline{01}000\underline{1}0\underline{00}10\underline{001}00\underline{1}0\underline{0}$$

$$\boxed{10101010011101010010111}$$

Keep this compressed with R/S dictionary schemes

Random access in O(1)-time via R/S dictionaries !

P. Ferragina and R. Venturini: *A simple storage scheme for strings achieving entropy bounds*, in Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2007, pp. 690–696.

K. Fredriksson and F. Nikitin: *Simple compression code supporting random access and fast string matching*, in Experimental Algorithms, Springer, 2007, pp. 203–216.

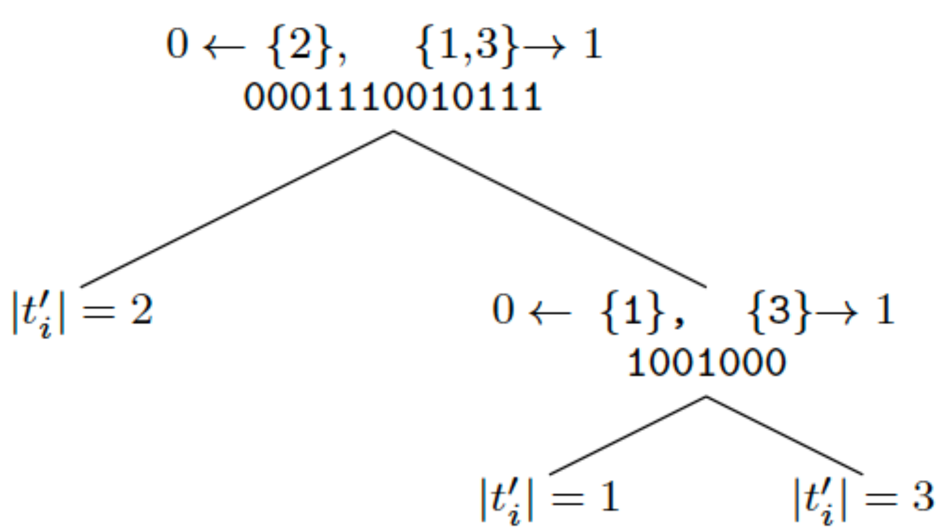
B. Adaş, E. Bayraktar, and M. O. Külekci: *Huffman codes versus augmented non-prefix-free codes*, in Experimental Algorithms, Springer, 2015, pp. 315–326.

How to efficiently mark codeword boundaries in NPF codes ?

Σ:	E	R	F	N	I	O	P	X
W :	0	1	00	01	10	11	000	001

T =	N	O	N	P	R	E	F	I	X	F	R	E	E
NPF(T) =	01	11	01	000	1	0	00	10	001	00	1	0	0
L =	2	2	2	3	1	1	2	2	3	2	1	1	1

2. Use **wavelet tree** to represent the codeword lengths



Create a wavelet-tree over the sequence of codeword lengths L .

Random access in O(log log σ) - time !

M. O. Külekci: *Uniquely decodable and directly accessible non-prefix-free codes via wavelet trees*, in Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on, IEEE, 2013, pp. 1969–1973.

B. Adaş, E. Bayraktar, and M. O. Külekci: *Huffman codes versus augmented non-prefix-free codes*, in Experimental Algorithms, Springer, 2015, pp. 315–326.

*Any **compressed integer representation** of the codeword lengths list is a candidate.*

Enumerative coding of integer vectors

Assume we have a d dimensional integer vector L.

$$L = \langle \ell_1, \ell_2, \ell_3, \ldots, \ell_d \rangle$$

We also know that the inner sum is v and each dimension is between 1 and k.

$$v = \ell_1 + \ell_2 + \ell_3 + \cdots + \ell_d \, , \quad 1 \le \ell_i \le k$$

Assuming all distinct L vectors of given v and k values are ordered, the rank of a vector in this ordered list specifies the vector.

$$d = 3, v = 6, k = 3$$

0	1	2	3
1	1	3	2
2	2	1	3
3	2	2	2
4	2	3	1
5	3	1	2
6	3	2	1

When rank is given as 4, the vector is $\langle 2,3,1 \rangle$.

If the vector is given as $\langle 3,1,2 \rangle$, then its rank is 5.

Number of d-dimensional distinct vectors such that...

Algorithm 1: $\psi(k, d, v)$

Input:
k: Maximum value of a dimension.
d: The number of dimensions.
v: The inner sum of the vectors.
Output:
Number of distinct d dimensional vectors with an inner sum of v .

```
if (v > k·d)||(v < d) then
    return 0;
if (d = 1)||(v = d) then
    return 1;
if (v = d + 1) then return d;
if (1 < v + k − k·d) then
 |   α = v + k − k·d
else
 |   α = 1
if (k < v − d + 1) then
 |   β = k
else
 |   β = v − d + 1
sum = 0;
for (i = α; i ≤ β; i+ = 1) do
 |   sum+ = ψ(k, d − 1, v − i);
end
return sum;
```

$\psi(k, d, v)$:

The total number of distinct d dimensional vectors whose inner sum is v , where each dimension is in range $[1,k]$.

0, *no such vector since* $d \le v \le k \cdot d$

1, *only one way to construct it, either* $\langle 1,1,\ldots,1 \rangle$ *or* $\langle v \rangle$

d, *There are d ways to construct it*

$$\left.\begin{array}{c} \langle 2,1,1,\ldots,1\rangle \\ \langle 1,2,1,\ldots,1\rangle \\ \ldots\ldots \\ \langle 1,1,1,\ldots,2\rangle \end{array}\right\} \text{d items}$$

otherwise, $\displaystyle\sum_{i=\alpha}^{i=\beta} \psi(k, d-1, v-i)$, *where*

$$\alpha = \begin{cases} 1, \; if \;\; v - k(d-1) \le 1 \\ v - k(d-1), \; otherwise \end{cases}$$

$$\beta = \begin{cases} k, \; if \;\; v - (d-1) > k \\ v - (d-1), \; otherwise \end{cases}$$

Iterate over all possible values for one dimension and recursively count on the remaining (d-1) dimensions with the updated sum v !

Number of distinct vectors complying with k,d,v parameters

Algorithm 1: $\psi(k, d, v)$

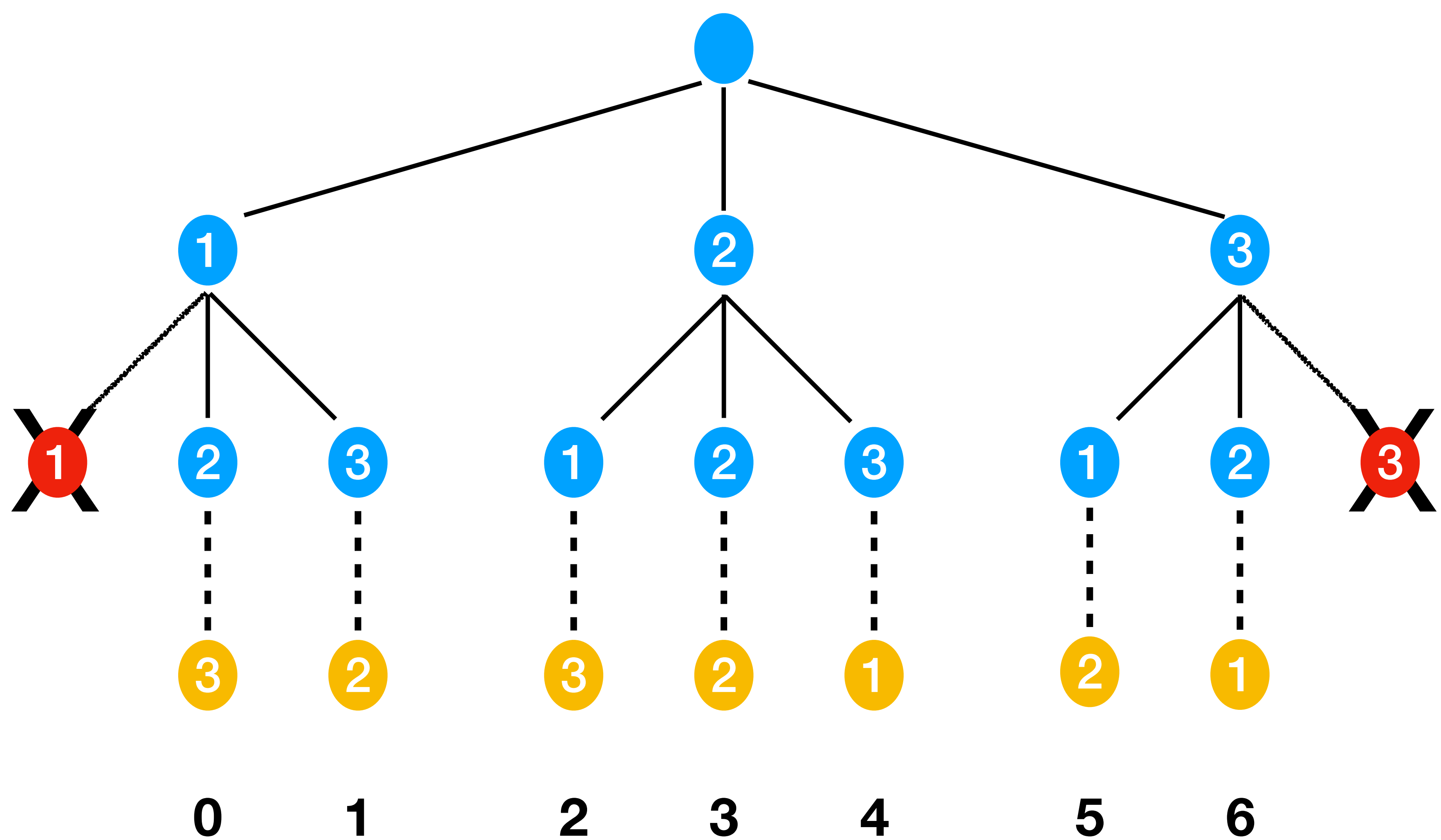
Input:
k: Maximum value of a dimension.
d: The number of dimensions.
v: The inner sum of the vectors.
Output:
Number of distinct d dimensional vectors with an inner sum of v.

if $(v > k \cdot d) || (v < d)$ **then**
  return *0*;
if $(d = 1) || (v = d)$ **then**
  return *1*;
if $(v = d + 1)$ **then return** d;
if $(1 < v + k - k \cdot d)$ **then**
  $\alpha = v + k - k \cdot d$
else
  $\alpha = 1$
if $(k < v - d + 1)$ **then**
  $\beta = k$
else
  $\beta = v - d + 1$
$sum = 0$;
for $(i = \alpha; i \leq \beta; i += 1)$ **do**
  $sum += \psi(k, d - 1, v - i)$;
end
return sum;

$\psi(k, d, v)$:

The total number of distinct d dimensional vectors whose inner sum is v, where each dimension is in range $[1,k]$.

This is akin to constructing the d-ary tree of height (k-1), where each inner node only creates children that accompany with the restrictions. For example, if d=3, k=3, v=6 then…



Vector-To-Index

We need methods to map a vector to its rank and vice versa.

Vector-To-Index: What is the rank (index) of $\langle 3{,}2{,}1 \rangle$ given that k = 3 ?

Notice d=3 and v=2+3+1= 6 are immediate from the vector.

Algorithm 2: VectorToIndex($\langle v_1, v_2, \ldots, v_d \rangle, d, k$)

Input: k: Maximum value of a dimension. d: The number of dimensions. $v_1 \cdots v_d$: Input vector.

Output: Rank of the input vector among lexicographically sorted vectors with the same inner sum of $\sum_i v_i$.

$v = v_1 + v_2 + \ldots + v_d$;

if $(d = 1) || (v = d)$ **then** $\boxed{\textbf{return } 0;}$

$index = 0$;

for $(i = 1; i < v_1; i+ = 1)$ **do**

  $\boxed{index+ = \psi(k, d-1, v-i);}$

end

$\boxed{index+ = \text{VectorToIndex}(\langle v_2, v_3, \ldots, v_d \rangle, d-1, k);}$

return $index$;

How many $\langle 1, *, * \rangle$, and $\langle 2, *, * \rangle$ vectors?

How many 2-dim vectors that sum up to 5 or 4?

Iterate the first dimension till the actual value

What is the index of $\langle 2{,}1 \rangle$?

Recurse with the reduced dimension….

$\psi(k, d, v)$: *The total number of distinct d dimensional vectors whose inner sum is v , where each dimension is in range $[1,k]$.*

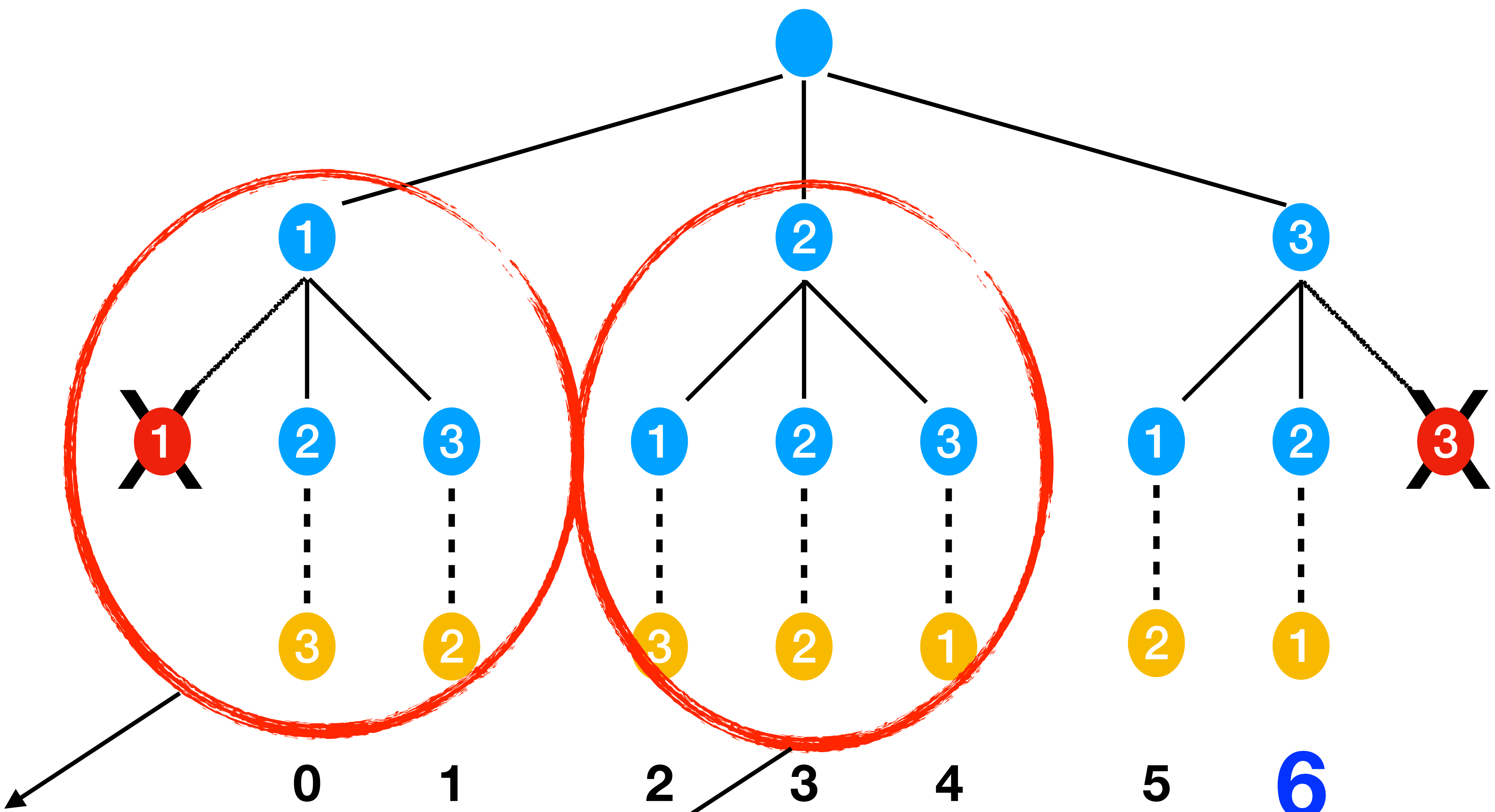
?

Vector-To-Index

We need methods to map a vector to its rank and vice versa.

Vector-To-Index: What is the rank (index) of $\langle 3,2,1 \rangle$ given that k = 3 ?

Notice d=3 and v=2+3+1= 6 are immediate from the vector.



How many **2 dim vectors sum up to 5 with k=3?**

How many **2 dim vectors sum up to 4 with k=3?**

What is the index of $\langle 2,1 \rangle$ with k=3?

Index-To-Vector

We need methods to map a vector to its rank and vice versa.

Index-To-Vector: What is the **d=3** dimensional vector with rank 6 and inner sum **v=6**, where each dimension is between 1 and **k=3**?

Algorithm 3: $IndexToVector(k, d, v, index)$

Input: k: Maximum value of a dimension d: The number of dimensions. v: The inner sum of the vectors. $index$: The rank of the vector among all possible vectors.

Output: The $\langle v_1, v_2, \ldots, v_d \rangle$ vector with $v_1 + v_2 + \cdots + v_d = v$, and rank $index$ among all possible vectors with inner sum v.

for $(i = 1; i < d; i+ = 1)$ **do**
  $v_i = 1;$
  while $(z = \boxed{\psi(k, d-i, v-v_i)} < index)$ **do**
    $index- = z;$
    $v_i = v_i + 1;$
  end
  $\boxed{v = v - v_i;}$
end
$v_d = v;$

Iteratively find the value of each dimension from v_1 to v_d

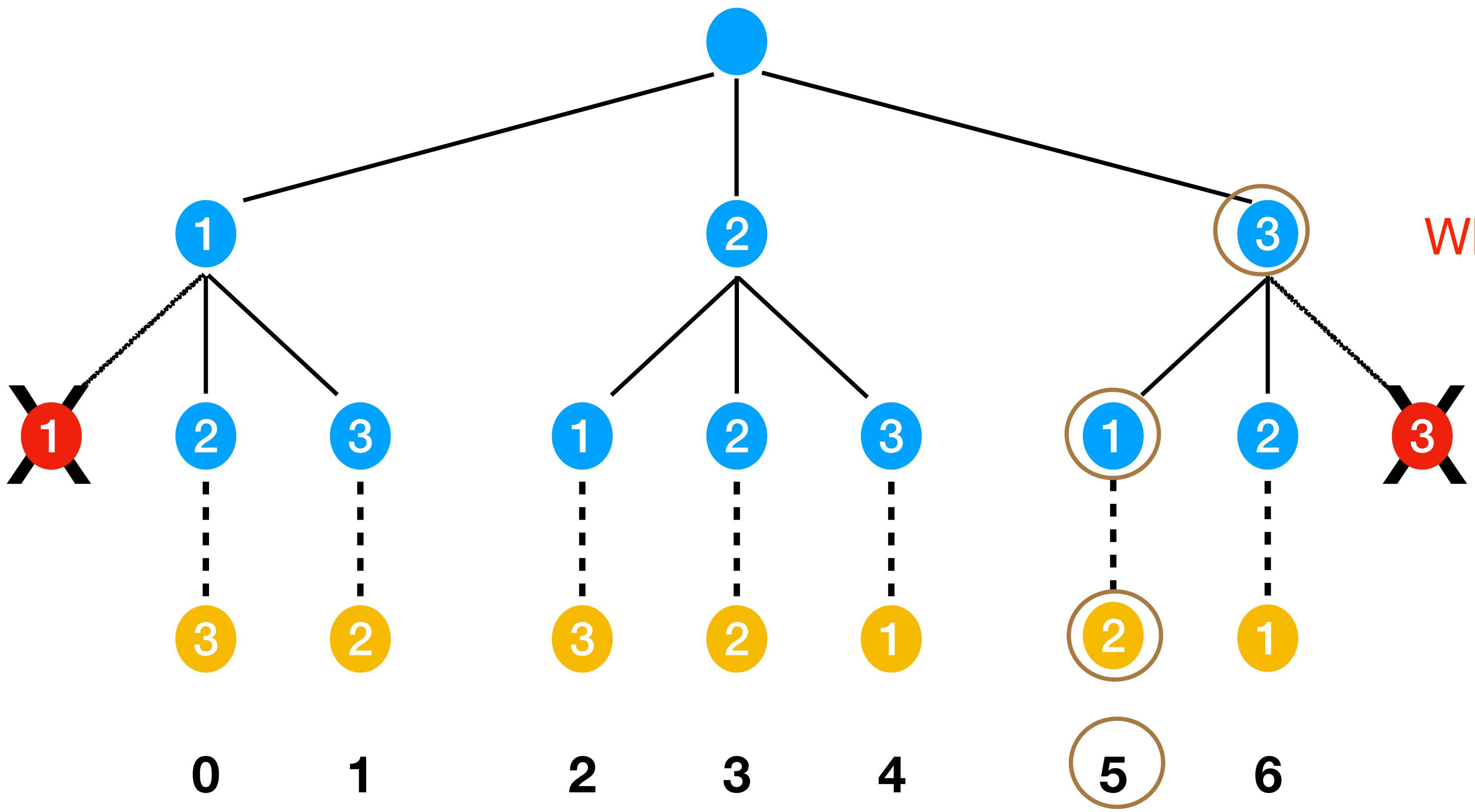
Count the number of vectors that rank before the queried index.

Adjust the inner sum once v_i is decided.

Index-To-Vector

We need methods to map a vector to its rank and vice versa.

Vector-To-Index: What is the 3-dim vector that ranks 5th, whose inner sum is 6 and each dimension is in [1,3] ?



What should v_1 be ? **3**

What should v_2 be ? **1**

v_3 is then 6-3-1 = **2**

Enumerative Compression — Overview

Σ:	E	R	F	N	I	O	P	X
W :	0	1	00	01	10	11	000	001

T =	N	O	N	P	R	E	F	I	X	F	R	E	E
NPF(T) =	01	11	01	000	1	0	00	10	001	00	1	0	0
L =	2	2	2	3	1	1	2	2	3	2	1	1	1

L	2	2	2	3	1	1	2	2	3	2	1	1	1	3	3
P	6			5			7			4			7		
Q	$VectorToIndex(\langle 2,2,2\rangle) =$ 3			$VectorToIndex(\langle 3,1,1\rangle) =$ 5			$VectorToIndex(\langle 2,2,3\rangle) =$ 1			$VectorToIndex(\langle 2,1,1\rangle) =$ 2			$VectorToIndex(\langle 1,3,3\rangle) =$ 0		

NPF The codeword stream

P stream encoding the sum of the codeword lengths in blocks of d

Q stream encoding the index of the corresponding d-dim codeword lengths vector.

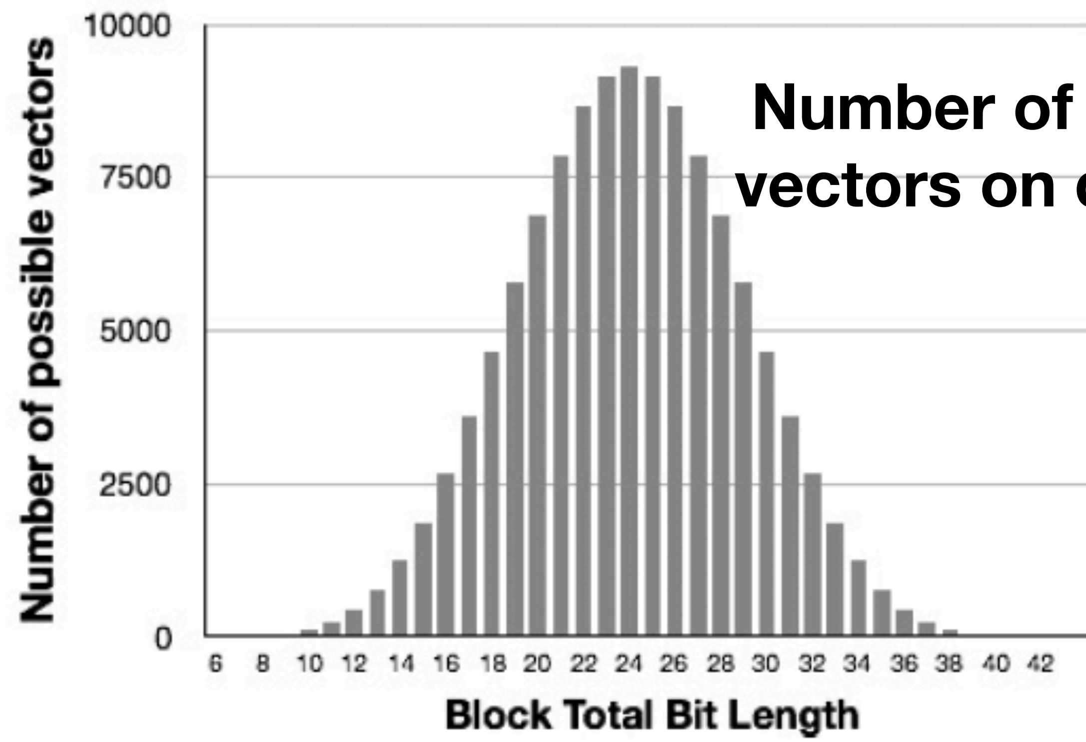
Encoding the P-stream

Algorithm 4: Encode(T, d)

Input: $T = t_1 t_2 \cdots t_n$ is the input data, where $t_i \in \Sigma = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_\sigma\}$. d is the chosen block length.

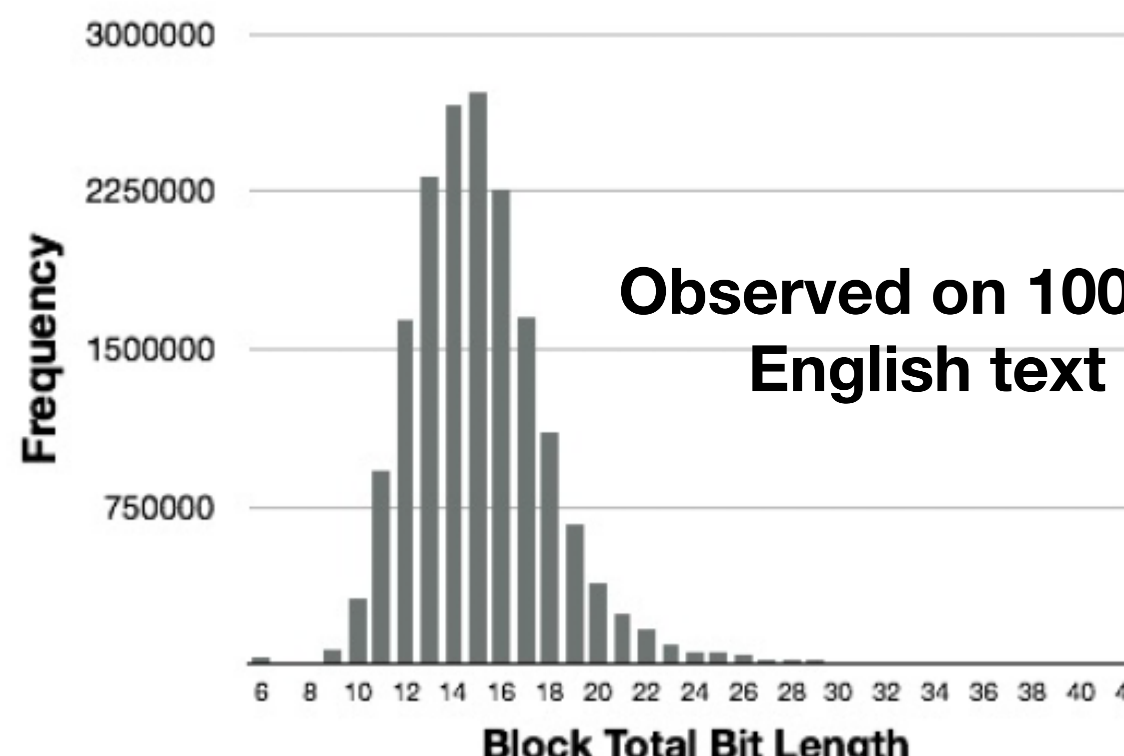
Output: The codeword bit-stream and the compressed $\langle p_i, q_i \rangle$ list.

```
r = ⌈n/d⌉ ;
B = ∅ ;
Generate the NPF codeword set
  W = {w_1, w_2, ..., w_σ};
k = ⌊log(σ + 1)⌋;
for (i = 0; i < r; i+ = 1) do
    p_i = 0;
    for (j = 0; j < d; j+ = 1) do
        ε_h = T[i · d + j + 1];
        B ← B w_h;
        vec[j + 1] = |w_h|;
        p_i+ = vec[j + 1];
    end
    Encode p_i into Pstream with an
      adaptive coder;
    if (p_i ≠ d)&&(p_i ≠ k · d) then
        q_i = VectorToIndex(vec[], d, k) ;
        Encode q_i into Qstream with an
          adaptive coder by using the
          sum value as the context;
end
```



Number of distinct vectors on d=6, k=7

The integers in P are between d and k.d.



Observed on 100MB English text

We encode P with adaptive arithmetic coding.

Encoding the Q-stream

Algorithm 4: Encode(T, d)

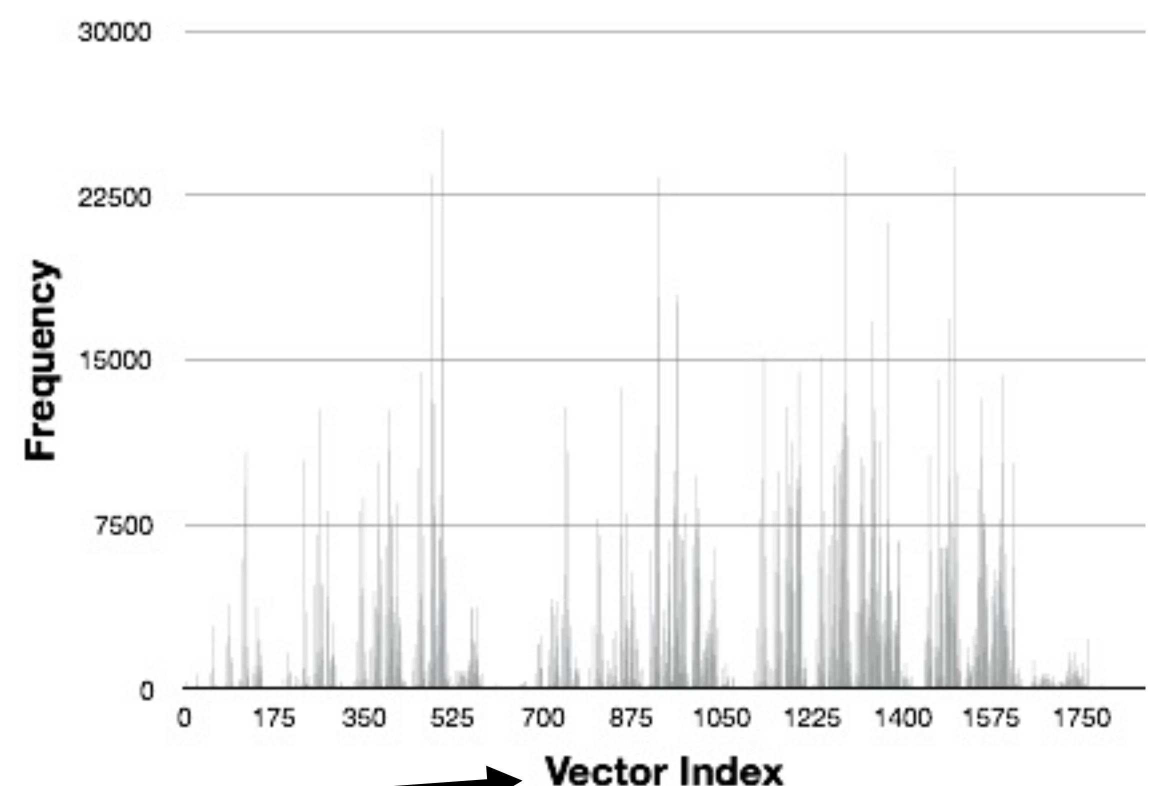
Input: $T = t_1 t_2 \cdots t_n$ is the input data, where $t_i \in \Sigma = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_\sigma\}$. d is the chosen block length.

Output: The codeword bit-stream and the compressed $\langle p_i, q_i \rangle$ list.

$r = \lceil \frac{n}{d} \rceil$;
$B = \emptyset$;
Generate the NPF codeword set $W = \{w_1, w_2, \ldots, w_\sigma\}$;
$k = \lfloor \log(\sigma + 1) \rfloor$;
for $(i = 0; i < r; i+ = 1)$ **do**
    $p_i = 0$;
    for $(j = 0; j < d; j+ = 1)$ **do**
        $\epsilon_h = T[i \cdot d + j + 1]$;
        $B \leftarrow B w_h$;
        $vec[j + 1] = |w_h|$;
        $p_i + = vec[j + 1]$;
    end
    Encode p_i into $Pstream$ with an adaptive coder;
    if $(\boxed{p_i \neq d)\&\&(p_i \neq k \cdot d})$ **then**
        $q_i = \boxed{VectorToIndex}(vec[], d, k)$;
        Encode q_i into $Qstream$ with an adaptive coder by using the sum value as the context;
end

$q_i = 0$ for sure, so no encoding

The value of p_i is the context of the vector index q_i. Therefore, we use again an adaptive coder with p_i context to encode the Q-stream, the indices of the d-dimensional vectors.



Assuming d=6, k=7, for the block size of 15 bits, there are 1875 distinct vectors. The frequencies of occurrences on English text is shown above.

Decoding…

Algorithm 5:

$\mathrm{Decode}(B, Pstream, Qstream, d, n, W)$

Input: B is the NPF codeword bit stream.
$Pstream$ is the compressed p_i values.
$Qstream$ is the compressed q_i values.
$W = \{w_1, w_2, \ldots, w_\sigma\}$ is the NPF codeword set.

Output: The original data sequence $T = t_1 t_2 \cdots t_n$

$r = \lceil \frac{n}{d} \rceil$;
$k = \lfloor \log(\sigma + 1) \rfloor$;
for $(i = 0; i < r; i+ = 1)$ **do**
  Decode p_i from the $Pstream$;
  if $p_i = d$ **then**
    $\langle v_1, v_2, \ldots, v_d \rangle = \langle 1, 1, \ldots, 1 \rangle$;
  else if $p_i = k \cdot d$ **then**
    $\langle v_1, v_2, \ldots, v_d \rangle = \langle k, k, \ldots, k \rangle$;
  else
    Decode q_i from the $Qstream$ by using p_i as the context ;
    $\langle v_1, v_2, \ldots v_d \rangle \leftarrow IndexToVector(k, d, p_i, q_i)$;
  end
  for $(j = 1; j \leq d; j+ = 1)$ **do**
    $w_h \leftarrow$ Read next v_j bits from B;
    $t_{i \cdot d + j} = \epsilon_h$;
  end
end

Step 1. Decode p_i from the P-stream.

…. now we know how many bits to read from the codeword stream

Step 2. Decode q_i from the Q-stream by using the decoded p_i as the context

…. now we know the index of the d-dim vector, and then by using the IndexToVector(), we generate the vector

Step 3. Decode the symbols from the NPF codeword stream

…. since the codeword lengths are in the decoded vector, easy to construct the actual symbols

Experimental Results…

File	Size	Symbols	Entropy	Huffman Stat.	Huffman Adapt.	Arithmetic Stat.	Arithmetic Adapt.	NPF RS	NPF WT	Non-uniquely decodable d=2	d=4	d=6
sprot34.dat	109MB	66 (k=6)	4.762	4.797	4.785	4.764	4.749	5.434	5.178	4.869	4.790	**4.698**
chr22.dna	34MB	5 (k=2)	2.137	2.263	2.195	2.137	**1.960**	2.957	2,616	2.468	2.466	2.462
etext99	105MB	146 (k=7)	4.596	4.645	4.595	4.604	4.558	5.140	4,553	4.632	4.570	**4.553**
howto	39MB	197 (k=7)	4.834	4.891	4.779	4.845	**4.731**	5.300	4.215	4.856	4.759	4.736
howto.bwt	39MB	198 (k=7)	4.834	4.891	3.650	4.845	**3.471**	5.300	4.215	4.143	3.950	3.949
jdk13c	69MB	113 (k=6)	5.531	5.563	5.486	5.535	5.450	6.404	5.658	5.577	5.460	**5.275**
rctail96	114MB	93 (k=6)	5.154	5.187	5.172	5.156	5.139	5.766	5.408	5.164	5.020	**4.818**
rfc	116MB	120 (k=6)	4.623	4.656	4.573	4.626	4.529	5.094	4.853	4.685	4.555	**4.463**
w3c2	104MB	256 (k=8)	5.954	5.984	5.700	5.960	5.659	6.648	5.820	5.826	5.686	**5.617**

Table 1. Compression ratio comparison between the proposed scheme, NPF rank/select and wavelet tree [1], arithmetic, and Huffman coding in terms of bits/symbol.

Compression ratio is better than the RS / WT schemes, very close and even better than the prefix codes (Huffman, arithmetic)

- *k is defined by the alphabet size* $k = \lfloor \log(\sigma + 1) \rfloor$
- *d is the block size in symbols akin to dimension of the enumerated vectors.*
- *Making d larger improves the performance, but computationally gets harder….*

Experimental Results…

File	Codeword Stream	Pstream			Qstream		
		d=2	d=4	d=6	d=2	d=4	d=6
sprot34.dat	2.686	1.476	0.909	0.659	0.707	1.196	1.353
chr22.dna	1.494	0.718	0.504	0.399	0.256	0.468	0.568
etext99	2.516	1.316	0.789	0.580	0.800	1.265	1.457
howto	2.618	1.451	0.885	0.655	0.787	1.256	1.464
howto.bwt	2.618	1.183	0.781	0.604	0.342	0.552	0.726
jdk13c	3.263	1.449	0.871	0.642	0.866	1.327	1.370
rctail96	2.878	1.462	0.893	0.659	0.824	1.250	1.281
rfc	2.516	1.472	0.911	0.677	0.697	1.128	1.271
w3c2	3.436	1.548	0.949	0.706	0.841	1.301	1.475

How many bits are used for the NPF, P and Q streams ?

With the current model, there is a trade-off between P and Q streams.

When d increases :

- **compression of P values gets improved,**
- **compression of Q stream gets worse**

(… maybe another modeling for Q would be better ???)

Conclusions…

- **An initial attempt to investigate not-much-addressed non-prefix-free codes**

- **Compression ratio seems compatible with the prefix codes.**

- **However computational load needs a lot improvement, current implementation is order of magnitudes slower than the prefix alternatives.** *Algorithm engineering, time-memory trade off, recursions to be replaced by iterations ?*

- **Tight theoretical bounds comparing the compression performances of prefix and non-prefix codes needs to be studied.**

- **The inherent ambiguity of the NPF codes surely suffering in data compression, but they can serve for privacy/security purposes in text processing, e.g, privacy-preserving text similarity** *(SISAP'19)*, **reducing the load of encryption (***SEA'18***), and maybe others ?**

Give a chance :) to non-prefix codes in data compression and possibly in other text processing algorithms