LC029 정보검색

Chapter 5 : Index compression

인덱스 압축

Types of Compression Techniques

- Lossless compression 무손실 압축
  - All information is preserved. 모든 정보가 보존됨
- Lossy compression 손실 압축
  - Discard some information. 일부 정보가 버려짐
  - Make sense when the lost information is not important. 버려진 정보는 별로 중요하지 않음
  - Better compression ratios can be achieved. 대신 압축률이 좋음
- Several of the preprocessing steps can be viewed as lossy compression 몇 가지 전처리 단계는 손실 압축으로 보일 수 있다
  - case folding, stop words, stemming, ...

Reuters RCV1

	terms		nonpositional postings		positional postings	
	#	Δ%	# (K)	Δ%	# (K)	Δ%
unfiltered	484,494		109,971		197,879	
no number	473,723	-2	100,608	-8	179,158	-9
case folding	**391,523**	**-17**	96,969	-3	179,158	0
30 stop words	391,493	0	**83,390**	**-14**	**121,858**	**-31**
150 stop words	391,373	0	**67,002**	**-30**	**94,517**	**-47**
stemming	**322,383**	**-17**	63,812	-4	94,517	0

- Lossy compression makes sense when the lost information is not important.

COLLECTION STATISTICS

Estimate the Number of Terms 단어 수 추정하기

- It is often said that a language has a vocabulary of a certain size. 종종 언어는 특정 크기의 사전을 갖는다고 봄
  - Oxford English Dictionary has more than 600,000 words.
- 대규모 컬렉션의 어휘는 훨씬 더 크다.
- The vocabulary of large collections is much larger.
  - Include the name of people, locations, products, or scientific entities like genes 인명, 지명, 상품명, 학술용어 포함
  - 이들은 인버티드 인덱스에 들어가야 한다
  - These names need to be included in the inverted index.
- We estimate the number of distinct terms *M* in a collection documents 알려진 단어의 수를 M이라 추정한다
  - They need to be compressed. Why?

그것들은 압축되어야 한다. 이유는?

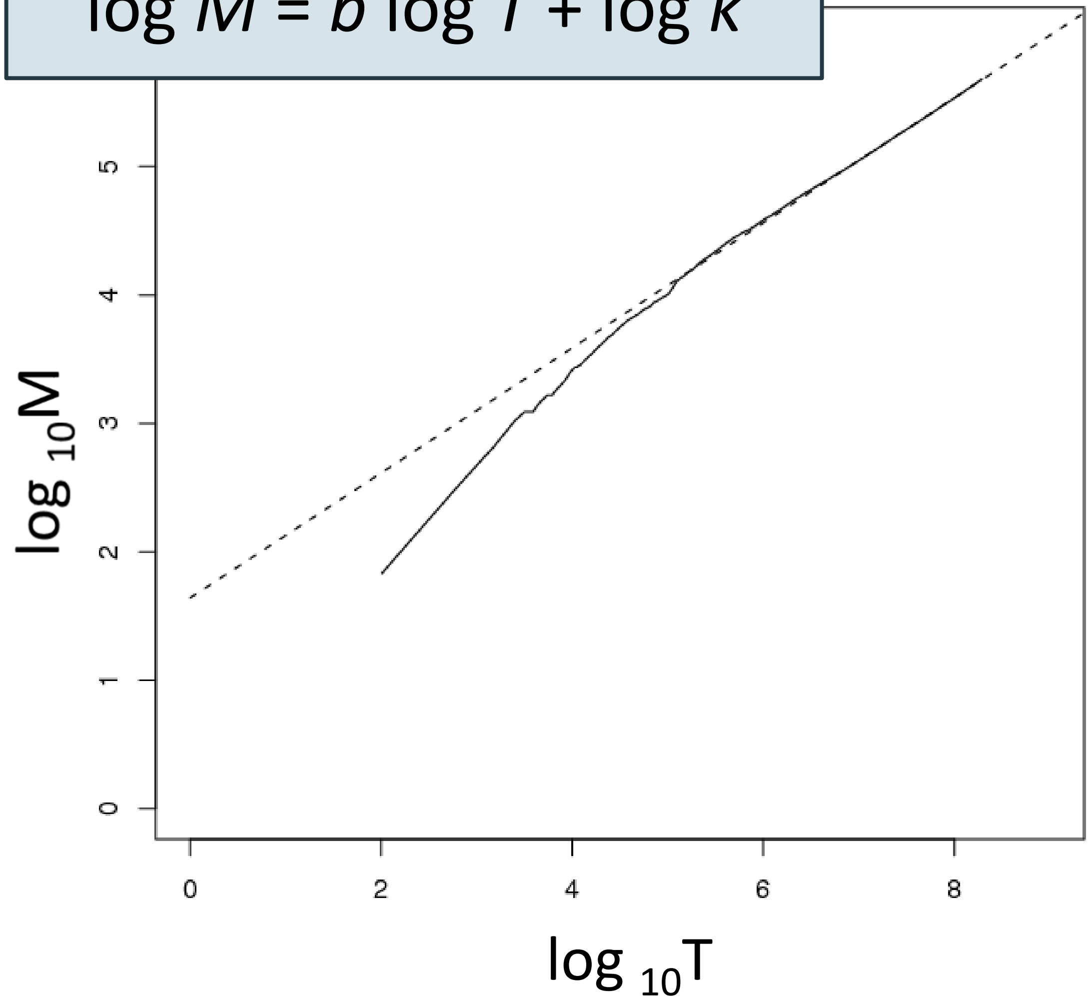
Heaps' Law 힙의 법칙

- Estimates vocabulary size as a function of collection size
  모음 크기의 함수로 어휘 크기를 추정한다.
  - $M = kT^b$
  - M is the vocabulary size.
  - T is the collection size, given as the number of tokens in the collection.
  - Typical values of k and b: $30 \leq k \leq 100$, $b \approx 0.5$
  - k depends on the nature of collection and how it is processed.
    case-folding and stemming : decrease k
    inclusion of numbers and spelling errors : increase k
- In log-log space, M and T are linear.
  - $\log M = b \log T + \log k$

Heaps' Law : Reuters RCV1

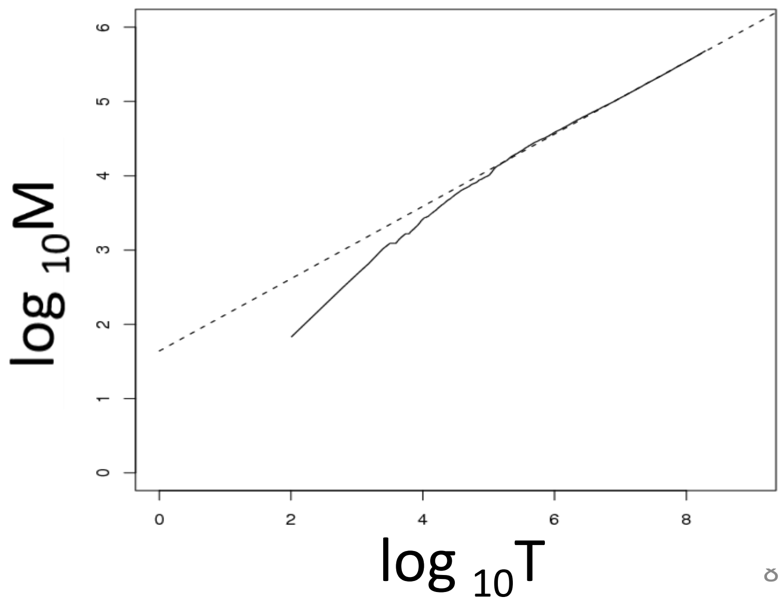
- $k = 44$
- $b = 0.49$
- $M = 44 \times T^{0.49}$
- If $T = 1{,}000{,}000$
  $M = 44 \times 10^{6 \times 0.49}$
  $= 44 \times 10^{2.94}$
  $\approx 38{,}323$
- Actual number is 38,365 terms.

$$\log M = b \log T + \log k$$



Heaps' Law

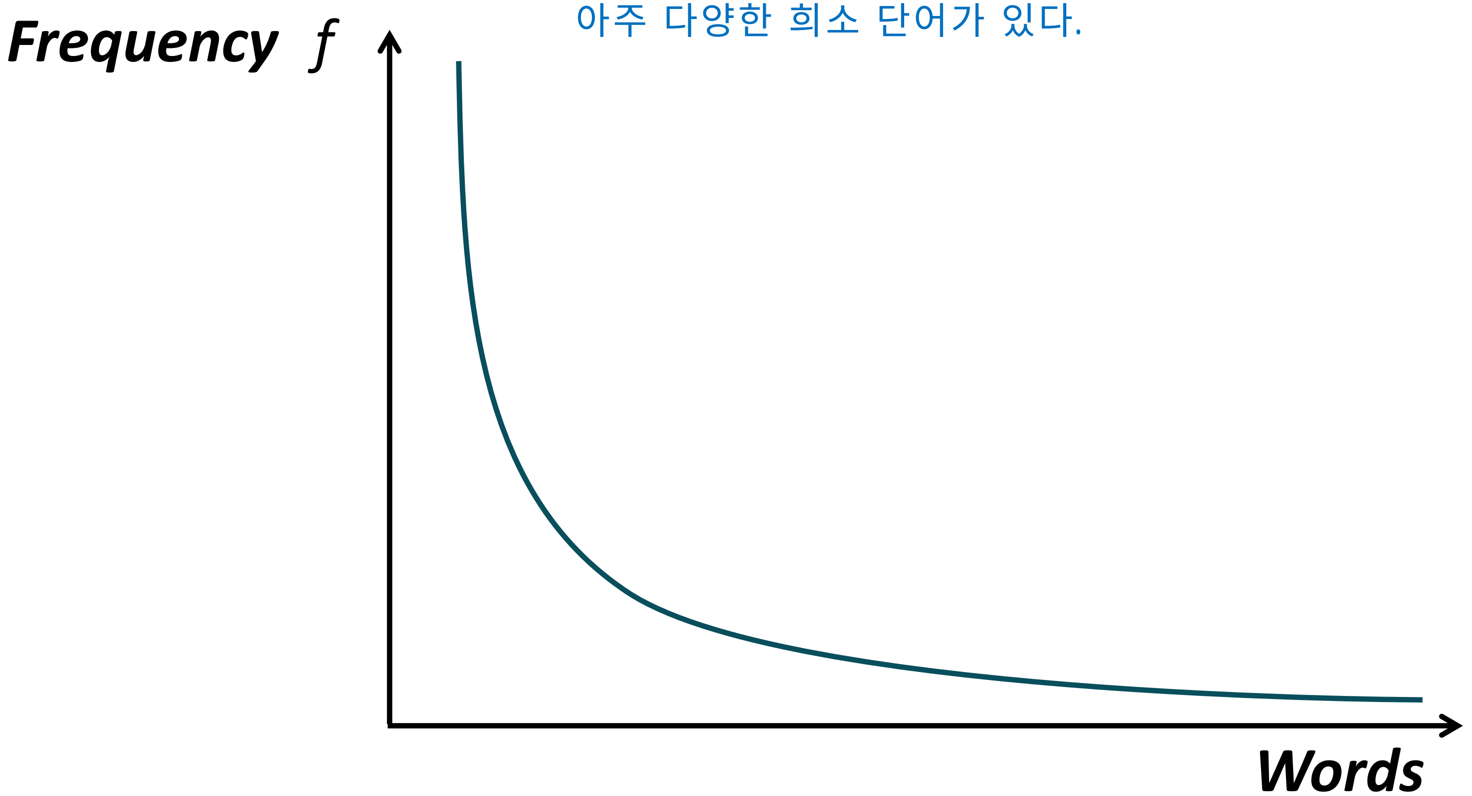
- Suggests that 가정
  - 사전은 계속 커짐(최대 단어 수 제한 없음)
  - The dictionary size continues to increase with more documents in the collection (no maximum vocab size)
  - The size of dictionary is quite large for large collections
  - 사전은 충분히 큼
- Therefore,
  - **Dictionary compression** is very important for an effective information retrieval system.
  - 효율적인 정보 검색 시스템을 위해
    사전 압축은 반드시 필요



Zipf's Law

- In natural language, there are a few very frequent terms and very many very rare terms.

보통 자연어에는 아주 흔한 소수의 단어가 있고,
아주 다양한 희소 단어가 있다.



Zipf's Law

- Estimates term frequency as a function of rank in a collection.

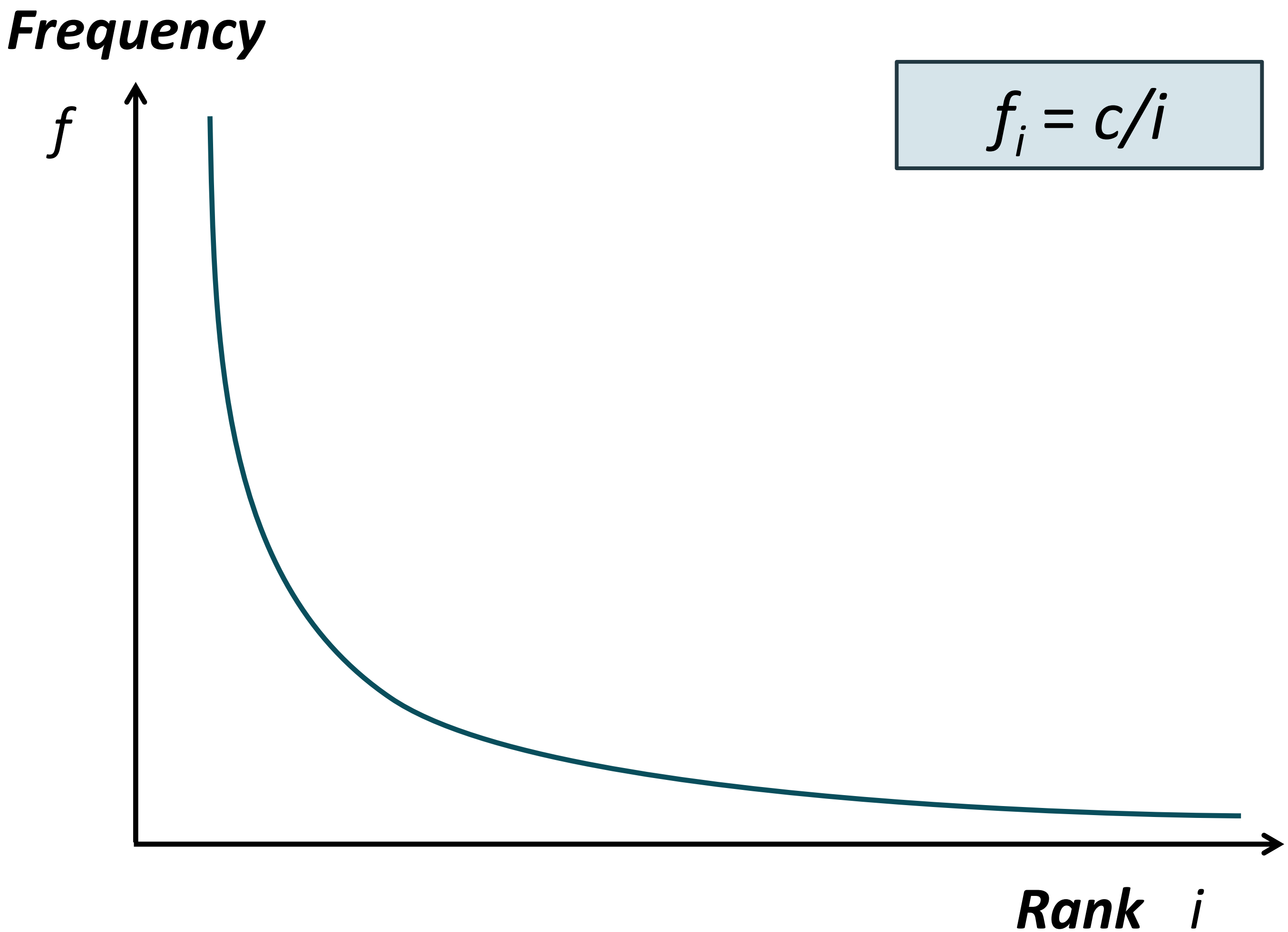
  $f_i = c/i$ (in log-log space, $\log f_i = \log c - \log i$)

  where f_i is the frequency of i-th most common term and
  c is a normalizing constant

- Example
  - Rank : the, of, and, … (assuming $c = 1$)

    $f_1 = 1000$, $f_2 = 500$, $f_3 = 333$, $f_4 = 250$, $f_5 = 200$, …

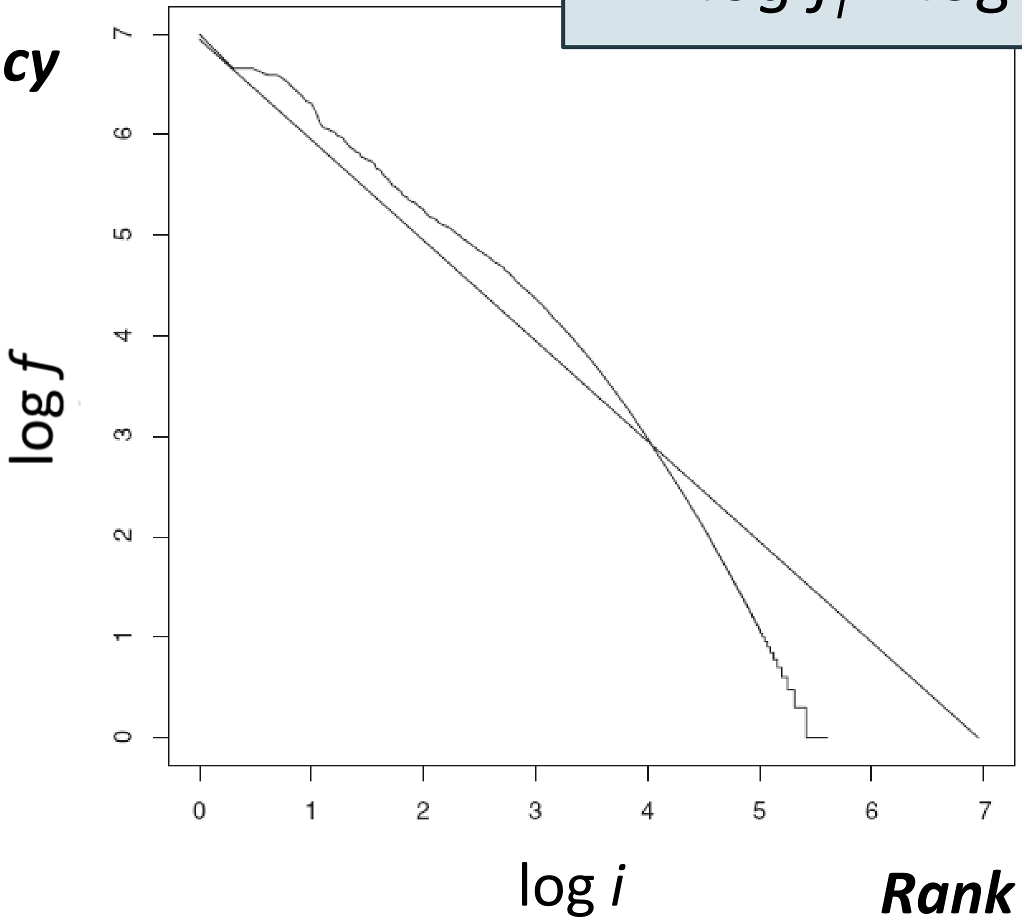
Zipf's Law



Zipf's Law

$$\log f_i = \log c - \log i$$

Frequency



Rank

한글 음절의 출현 빈도

- 약 3,100만 음절 조사
  - 전체 출현 음절은 약 2,200개임
  - 50개의 음절이 전체의 50%를 점유

순위	음절	누적%	순위	음절	누적%	순위	음절	누적%	순위	음절	누적%	순위	음절	누적%
1	이	3.2	11	로	21.4	21	리	31.8	31	제	39.5	41	아	45.8
2	다	5.7	12	기	22.8	22	자	32.6	32	국	40.2	42	연	46.3
3	의	8.1	13	지	24.1	23	수	33.4	33	과	40.9	43	라	46.9
4	는	10.3	14	사	25.1	24	시	34.3	34	그	41.5	44	성	47.5
5	에	12.3	15	서	26.2	25	으	35.0	35	해	42.2	45	들	48.0
6	을	14.0	16	은	27.2	26	있	35.8	36	전	42.8	46	상	48.5
7	하	15.7	17	도	28.2	27	어	36.6	37	부	43.4	47	원	49.0
8	한	17.2	18	를	29.2	28	구	37.3	38	것	44.0	48	여	49.6
9	고	18.7	19	대	30.1	29	인	38.1	39	일	44.6	49	보	50.1
10	가	20.1	20	정	31.0	30	나	38.8	40	적	45.2	50	장	50.5

Index Compression

- First, consider space for **dictionary**
  - Make it small enough to keep in main memory
- Then, space for **postings**
  - Reducing disk space decreases time to read from disk
  - Large search engines keep a significant part of postings in main memory
- We assume that each postings is a docID.
  - We do not consider frequency and positional information.

Index Compression

- **Dictionary** Compression
  - Dictionary-as-a-string Method
  - Blocked Storage
  - Blocked Storage + Front Coding
- **Postings** Compression
  - Variable Byte Encoding
- Huffman Code

DICTIONARY COMPRESSION

Why compress the dictionary?

- Dictionary is small compared with the postings file
- Then, why compress the dictionary?
  - For fast query processing, keep the dictionary in memory (or at least a large portion of it)
  - For fast start-up time
  - To share the memory with other applications (esp. enterprise search engine)

Data Structure for Dictionary

- Array of fixed-width entries
  - RCV1 : 400,000 terms x 28 bytes/term = **11.2MB**

term	document frequency	pointer to posting list
a	656,265	→
aachen	65	→
…	…	→
zulu	221	→

20 bytes (term) 4 bytes (document frequency, pointer to posting list)

Array of fixed-width entries

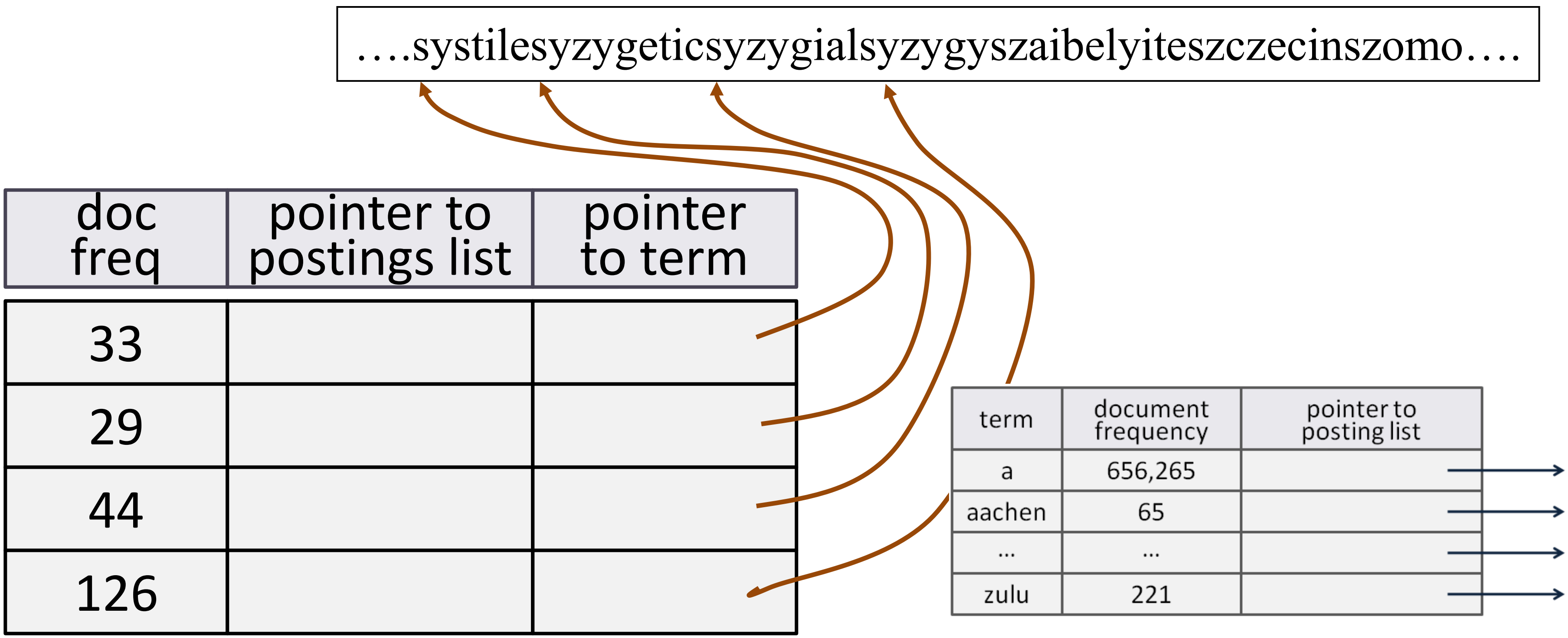
- We allot 20 bytes for each term.
  - According to RCV1 statistics:
    Written English : 4.5 characters / word
    Dictionary word in English : 7.5 characters / word

M	number of terms	400,000
	average number of bytes / token (without spaces/punctuation)	**4.5**
	average number of bytes / term	**7.5**

  - Thus, most of the bytes in the **term** column are wasted!!
  - And we still can't handle ***supercalifragilisticexpialidocious***.

Compression : Dictionary-as-a-String

- Store dictionary as one long string of characters:
  - A pointer to the next term shows the end of current word



Compression : Dictionary-as-a-String

- Total String Length = 400,000 terms X 8 Byte / term
  = **3.2MB**
- The size of pointer to term
  - Address space : 3.2MB
  - Size of pointers = $\log_2 3.2$ M = 22 bits = 3 bytes

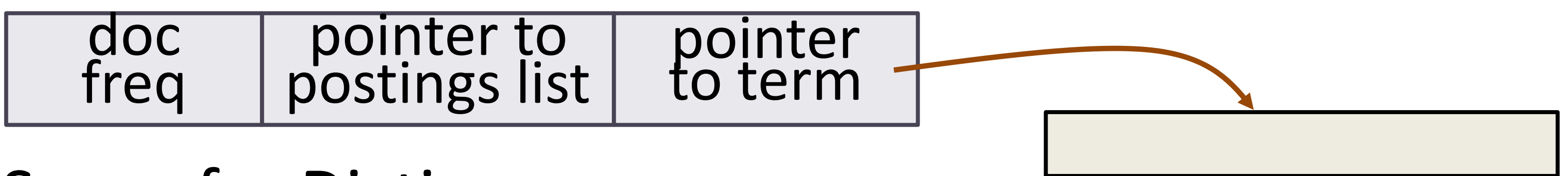
….systilesyzygeticsyzygialsyzygyszaibelyiteszczecinszomo….

doc freq	pointer to postings list	pointer to term
33		
29		
44		
126		

Space for Dictionary as a String

- 4 bytes for document frequency
- 4 bytes for pointer to postings list
- 3 bytes for pointer to term
- 8 bytes for term

Now average 11 bytes/term, not 20.



- Space for Dictionary

  400,000 terms X 19 B/term = **7.6MB**

  400,000 terms X 11 B/term + **3.2MB** = 4.4MB + **3.2MB** = **7.6MB**

- Note that **11.2MB** is required for fixed width

  Save 32% compared to fixed-width storage

Blocked Storage

- Further compress the dictionary by
  - Grouping terms in the string into blocks of size *k*
  - Keeping a term pointer only for the first term of each block
- Need to store **the length of the term** in the string as an additional byte at the beginning of each term

….7*systile*9*syzygetic*8*syzygial*6*syzygy*11*szaibelyite*8*szczecin*9*szomo*….

doc freq	pointer to postings list	pointer to term
33		
29		
43		
126		
7		

Assume k = 4

Space for Blocked Storage

- Eliminate $k - 1$ term pointers
- Need an additional k bytes for the length of each term
- So, we save

  $(k - 1) \times 3 - k = 2k - 3$ bytes per k-term block.

  When $k = 4$, we save 5 bytes per 4 term block.
- For RCV1, we save 400,000 / 4 X 5 = **0.5 MB** when $k = 4$

  So, the dictionary size is reduced to **7.1 MB**.

  Save 37% compared to fixed-width storage.
- We can save more with larger k.

  Then, why not go with larger k?

Space for Blocked Storage

- With larger *k*,
  - The dictionary is compressed more.
  - But dictionary search becomes prohibitively slow!!
- The lower limit of compressed dictionary size (RCV1)

  400,000 X (4 + 4 + 1 + 8) = **6.8 MB**

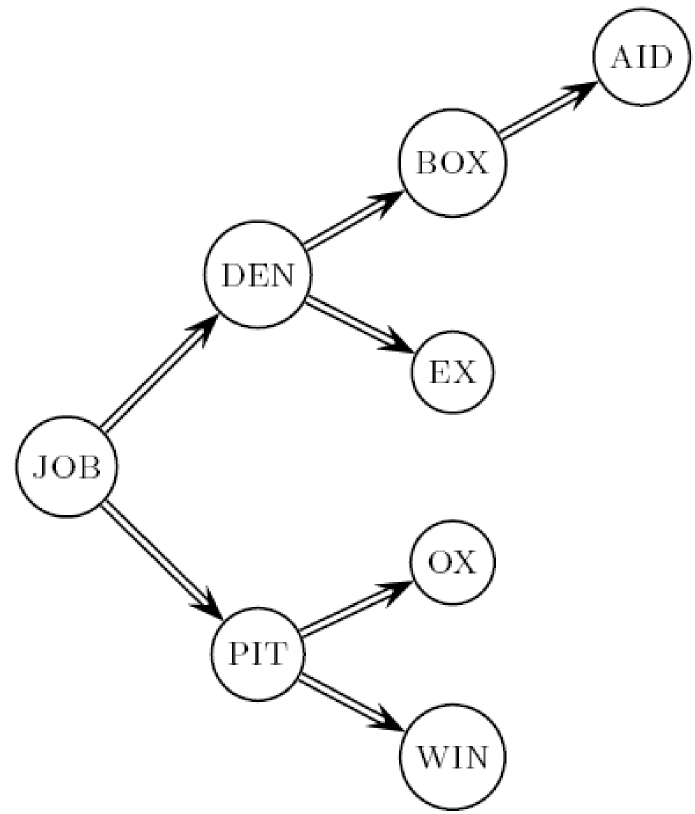
….7*systile*9*syzygetic*8*syzygial*6*syzygy*11*szaibelyite*8*szczecin*9*szomo*….

doc freq	pointer to postings list	pointer to term
33		
29		
43		
126		
7		

Dictionary Search without Blocked Storage

- Assuming each dictionary term equally likely in query, average number of comparisons is:

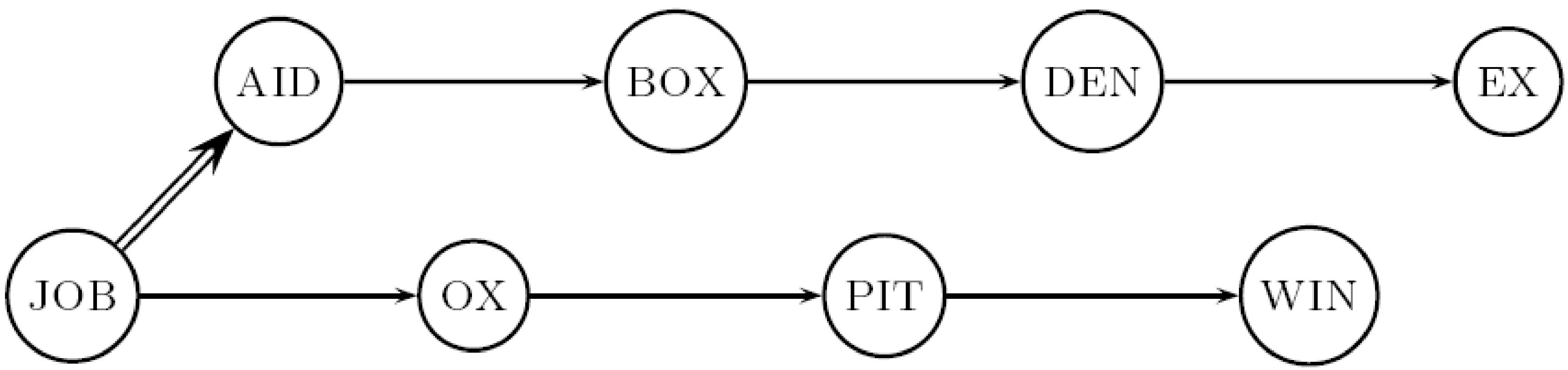
  $(1 + 2\cdot2 + 3\cdot4 + 4) / 8 = 2.6$



Dictionary Search with Blocked Storage

- Binary search down to *k*-term block, and then linear search through *k* terms in a block.
- When $k = 4$, average number of comparisons is:

  $(1 + 2{\cdot}2 + 3{\cdot}2 + 4{\cdot}2 + 5) / 8 = 3$



Front Coding

- Sorted words commonly have long common prefix.
- So, store differences only (for last *k-1* in a block of *k*).

8***automata***8***automate***9***automatic***10***automation***

→8***automat**** ***a***1◊***e***2◊***ic***3◊***ion***

Encodes ***automat***

Extra length beyond ***automat***

RCV1 Dictionary Compression

Technique	Size in MB
Fixed-Width	11.2
Term Pointers into String	7.6
With Blocked Storage k = 4	7.1
With Blocked Storage + Front Coding	5.9

POSTINGS COMPRESSION

Reuters RCV1 Statistics

symbol	statistic	value
N	documents	800,000
L_{ave}	average number of tokens / doc	200
M	number of terms	400,000
	average number of bytes / token (including spaces/punctuation)	6
	average number of bytes / token (without spaces/punctuation)	4.5
	average number of bytes / term	7.5
	number of non-positional postings	100,000,000

Postings Lists

- The postings file is much larger than the dictionary.
- A posting for our example is simply a docID, excluding frequency and position information.
- For Reuters RCV1,
  - The collection size:
    800,000 docs X 200 tokens / doc X 6 B / token = 960MB
  - How many bits to address docID?
    $\log_2 800{,}000 \approx 20$ bits / docID = 2.5 B / docID
  - Uncompressed posting file size:
    100,000,000 postings X 2.5 B / posting = 250MB
  - *cf*. 100,000,000 postings X 4 B / posting = 400MB

Postings Lists

- The bottom line for space requirement of postings file is 250MB.
- Our goal is to store each posting compactly, using less than 20 bits per docID.
- Key Idea
  - Instead of absolute docID, we **use the difference** between two adjacent docID in the postings file.
  - Our observation is that the postings for frequent terms are close together.
  - So, the difference or **gap** can be represented with less space.

Postings Lists

- We store the list of documents containing a term in increasing order of docID.

computer → 33, 47, 154, 159, 202 …

- Consequence: it suffices to store ***gaps***.

computer → 33, **14**, **107**, **5**, **43** …

- Hope: most gaps can be represented with far fewer than 20 bits.

Postings Lists

- A term like ***the*** occurs in virtually every doc.
  - So, 20 bits/posting is too expensive.
  - In fact, we can represent the gap with 1 bit.
- A term like ***arachnocentric*** occurs in one doc out of a million.
  - So, we should represent the gap with $\log_2 1{,}000{,}000 = 20$ bits.
- Conclusion: we need a ***variable byte encoding*** method.

Variable Byte Encoding

- Our Goal
  - Encode every gap with as few bits as needed for that gap.
- **Variable Byte Encoding** achieves this goal,
  by using short byte codes for small numbers.

Variable Byte Encoding

- Begin with one byte to store gap *G* and dedicate 1 bit in it to be used as a **continuation bit** *c.*
- If $G \leq 127$, encode it in the 7 available bits and set $c = 1$.
- Otherwise, encode *G*'s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm.
- Set the continuation bit of the last byte to 1 ($c = 1$). Continuation bit of other bytes is set to 0 ($c = 0$).

Variable Byte Encoding : Example

docIDs	824	829	215406
gaps		5	214577
VB code	00000110 10111000	10000101	00001101 00001100 10110001

Postings stored as the byte concatenation

000001101011100010000101000011010000110010110001

Key property: VB-encoded postings are Uniquely decodable.

For a small gap (5), VB uses one byte.

Other Variable Length Codes

- Instead of bytes, we can also use a different unit of alignment.
  - 32 bits (words)
  - 16 bits
  - 4 bits (nibbles)
- Variable Byte Code wastes space if you have many small gaps.
  - Nibbles do better in such cases.

RCV1 Compression

Data structure	**Size in MB**
RCV1 collection	**960**
dictionary, fixed-width	11.2
dictionary, term pointers into string	7.6
with blocked storage, k = 4	7.1
with blocked storage & front coding	5.9
postings, uncompressed (32-bit words)	**400.0**
postings, uncompressed (20 bits)	**250.0**
postings, variable byte encoded	**116.0**

Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Only 13% of the total size of the text in the collection
  - 960 MB for RCV1 collection vs 116 MB for VB encoding postings
- However, we've ignored positional information
  - Therefore, space savings are less for indexes used in practice
  - But techniques substantially the same

MORE ON COMPRESSION

Huffman Code

- Developed in the 1950s by David Huffman
- Use the frequency distribution of characters or words
- Variable length code
  - More frequent symbols are assigned shorter codes.
- It has the prefix property.
  - No code is the prefix of any other code.
  - Any bit stream is uniquely decodable with a given Huffman code.
  - Transmission errors are almost always automatically filtered out.

Huffman Code : Encoding

- Gather character (or word) frequency from a corpus.
- Build a Huffman tree, a binary tree, according to the frequency distribution.
- From the tree, we can get a Huffman Code Table.
- Using the table, encode each character (or word) into Huffman code.

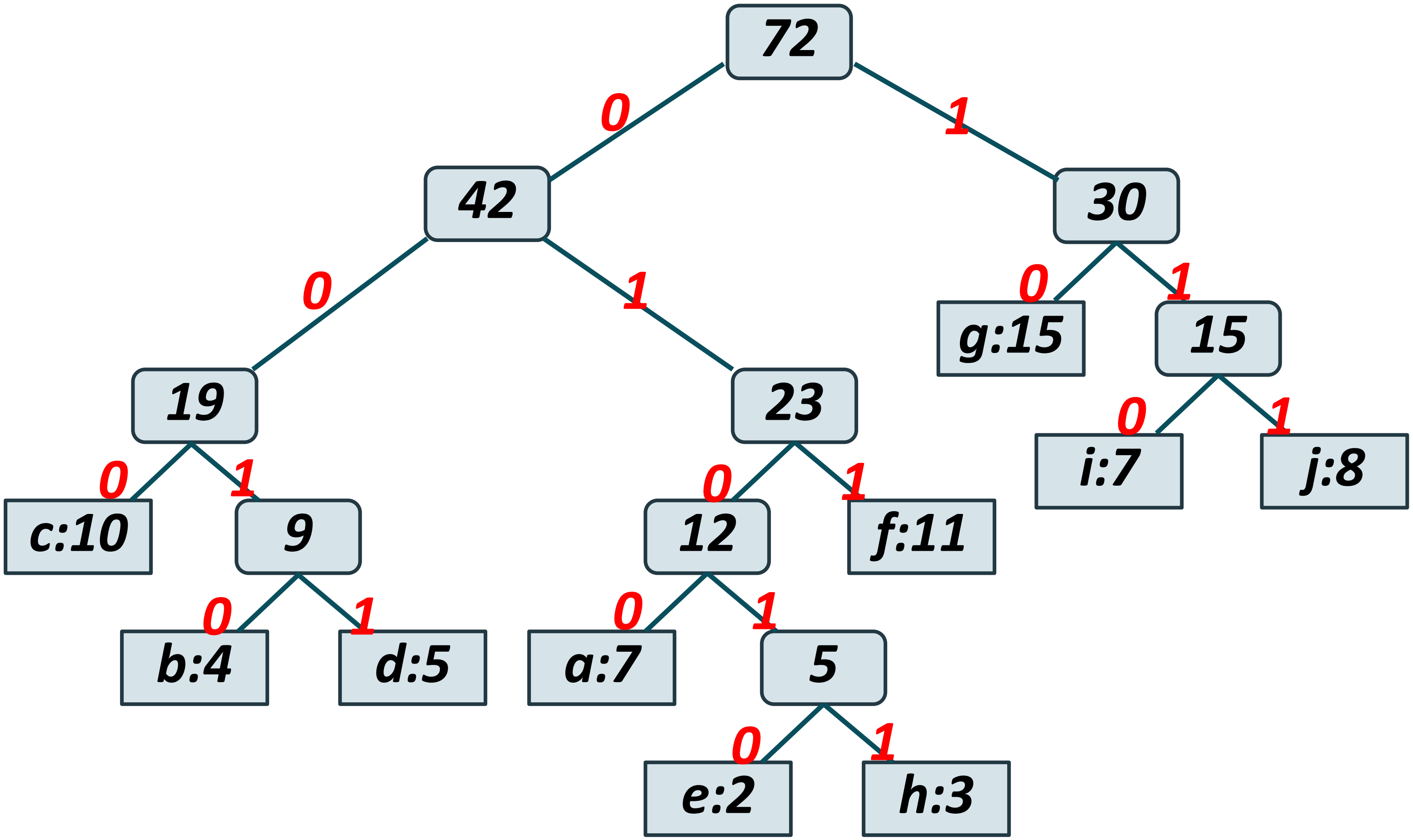
Huffman Code : Encoding

- Gather character (or word) frequency from a corpus.

Symbol	Frequency
a	7
b	4
c	10
d	5
e	2
f	11
g	15
h	3
i	7
j	8

Huffman Code : Encoding

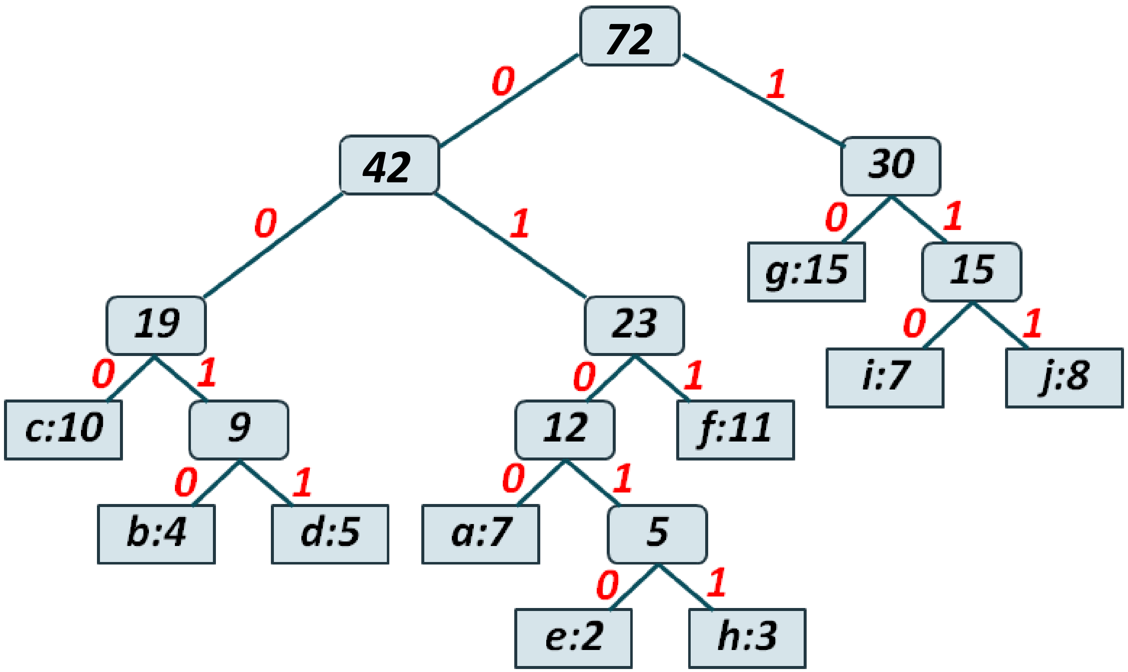
- Build a Huffman tree, binary tree, according to the frequency distribution.



Sym	Freq
a	7
b	4
c	10
d	5
e	2
f	11
g	15
h	3
i	7
j	8

Huffman Code : Encoding

- From the tree, we can get a Huffman Code Table.



Symbol	Huffman Code	Freq
a	0100	7
b	0010	4
c	000	10
d	0011	5
e	01010	2
f	011	11
g	10	15
h	01011	3
i	110	7
j	111	8

Huffman Code : Encoding

- Variable length code
- It has the prefix property.
  - No code is the prefix of any other code.
  - Any bit stream is uniquely decodable with a given Huffman code.

Symbol	Huffman Code
a	0100
b	0010
c	000
d	0011
e	01010
f	011
g	10
h	01011
i	110
j	111

Huffman Code : Encoding

- Using the table, encode each character (or word) into Huffman code.

feedabeefchade

↓

0110101001010001101000010010100101001100001011010000110101

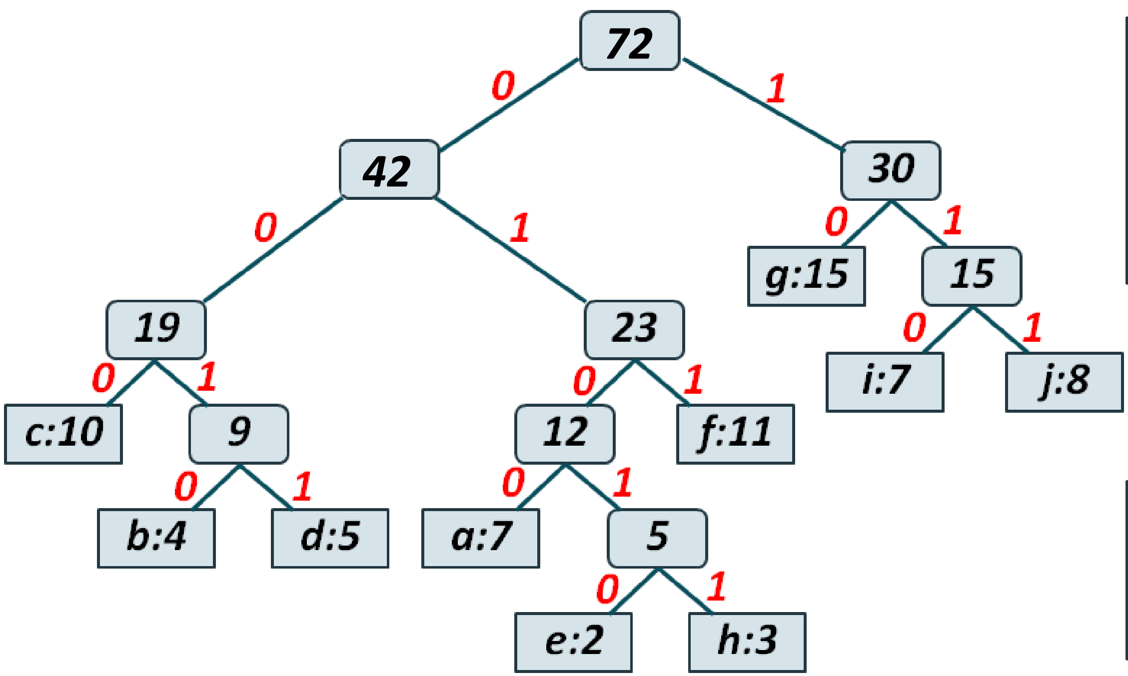
Symbol	Huffman Code
a	0100
b	0010
c	000
d	0011
e	01010
f	011
g	10
h	01011
i	110
j	111

Huffman Code : Decoding

- Use the same Huffman tree as used in encoding.
- From the root of the Huffman tree, traverse down until reach the leaves of the tree.
- Each character (or word) is decoded at the leaves.

Huffman Code : Decoding

- From the root of the Huffman tree, traverse down until reach the leaves of the tree.
- Each character (or word) is decoded at the leaves.

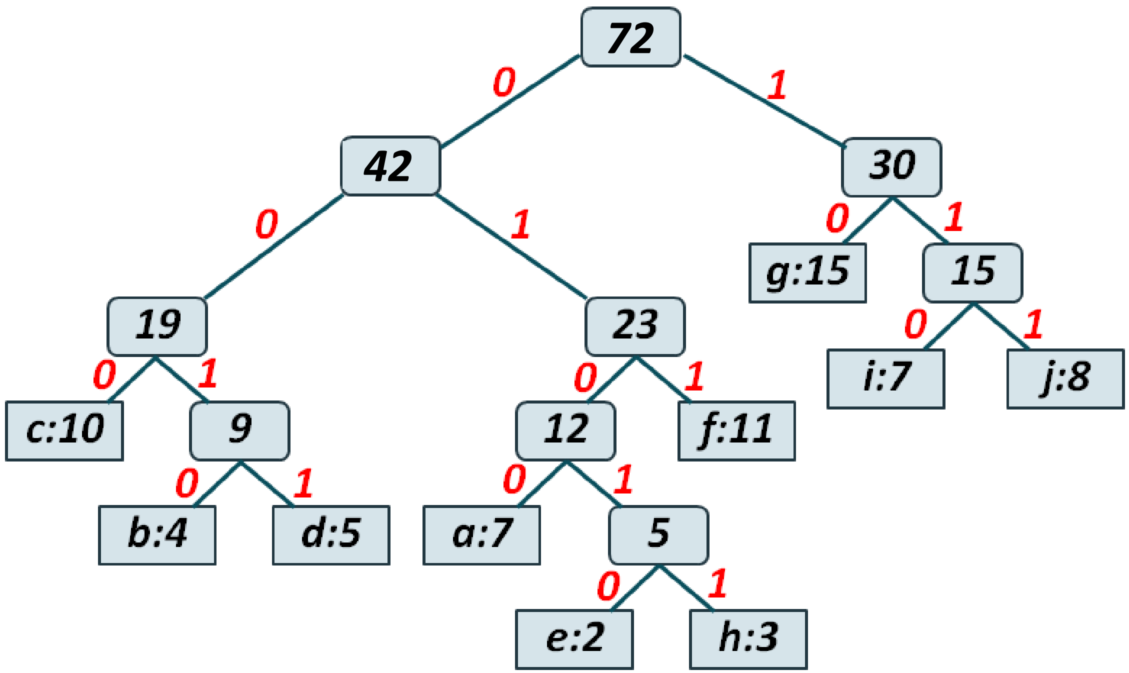


01101010010100011010
00010010100101001100
0010110100001101010

feed

Huffman Code : Decoding

- Transmission errors are almost always automatically filtered out.



01101010010100011010
00010010100101001100
0010110100001101010

feedabeefchade

01101100101000110100
00100101001010011000
0101101000011010

ffbgdabeefchade