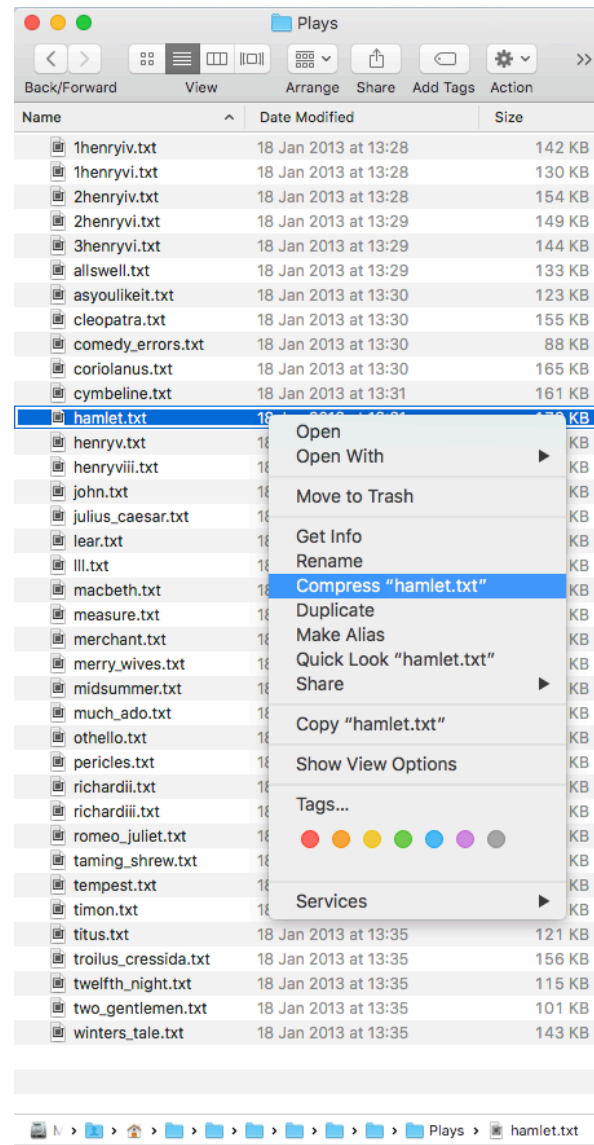


CSE202

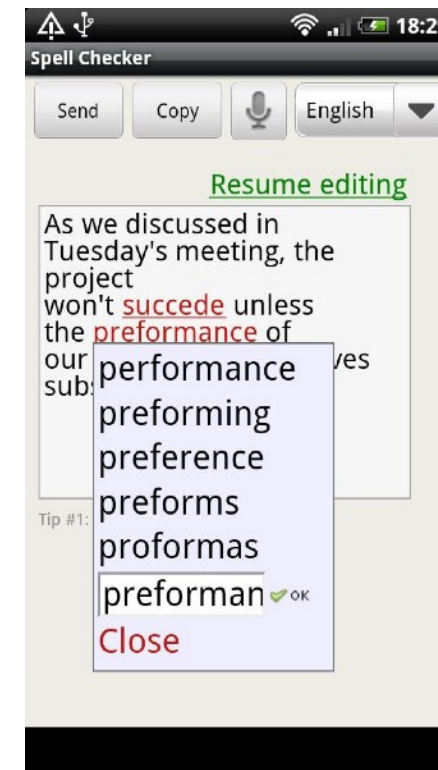
Design and Analysis of Algorithms

Week 12 — String Algorithms 2
Tries & Compression

Dictionaries & Compression



shrink text by
a factor 2.5,
without any loss



check existence
auto-complete

Both rely on specific trees:
the tries

Lossless Compression Must (Discover and) Use Regularity

Alphabet Σ , with $|\Sigma| = R$.

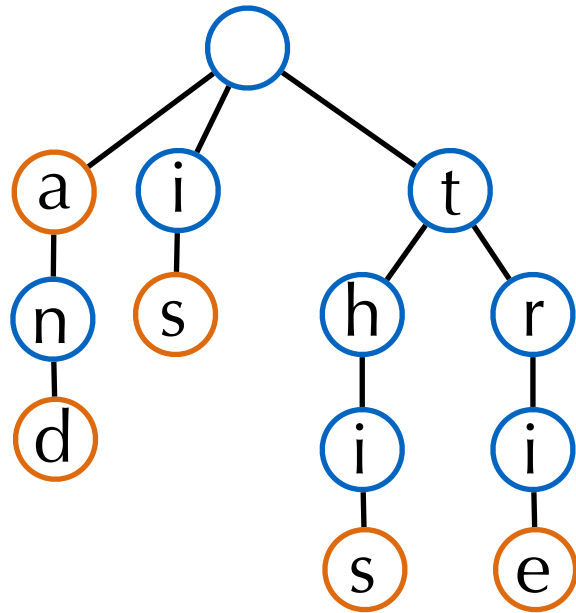
Not all R^n texts of length n can be reduced and still be distinct

An algorithm that compresses strings of length $2n$ by a factor 2 without loss can compress at most $1/R^n$ of them.

I. Tries (aka Digital Search Trees)

Tries

Children are sorted by alphabetical order



*Trie built from the strings
"and", "this", "is",
"a", "trie"*

Stores pairs (string, value)

Exercise:
auto-complete

```
class Node:
```

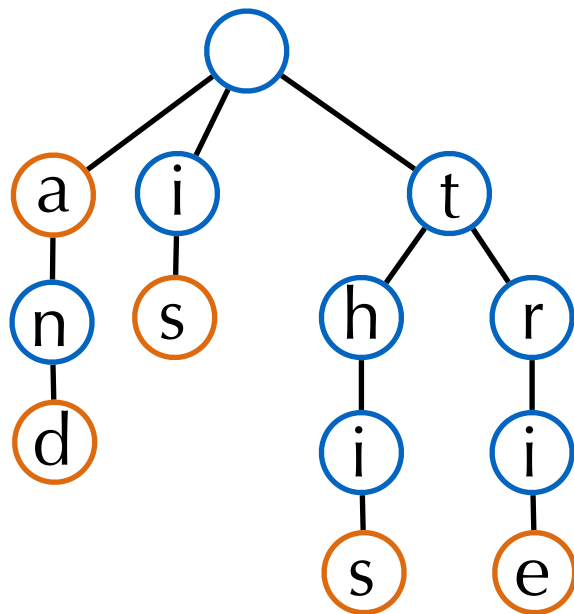
```
    def __init__(self, sizeAlpha, val=None):  
        self.val = val # None except at end of string  
        self.child = [None]*sizeAlpha
```

```
class Trie:
```

```
    def __init__(self, sizeAlpha=256):  
        self.root = None  
        self.sizeAlpha = sizeAlpha  
  
    def insert(self, str, val, index=0):  
        self.root=self._insert(self.root, str, val, index)  
  
    def find(self, str, index=0):  
        return self._find(self.root, str, index)  
  
    def longestprefix(self, str, index=0):  
        return self._longestprefix(self.root, str, index)
```

Find/Insert/LongestPrefix

Worst-case time *linear*
in length of st (**optimal**).



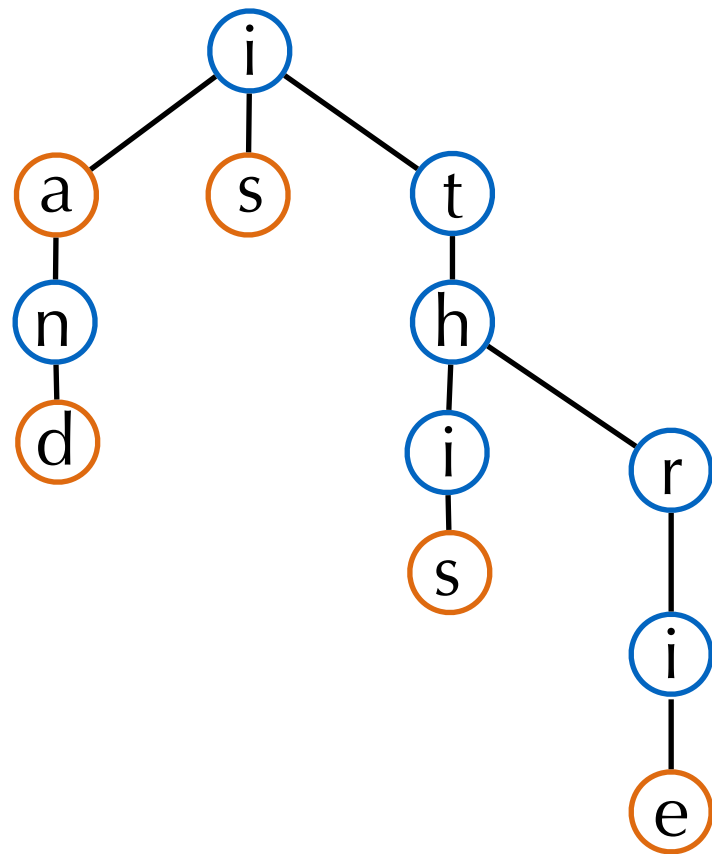
```
def _find(self, node, st, index):
    if node is None: return None
    if index == len(st): return node.val
    return self._find(node.child[ord(st[index])], st, index+1)
```

```
def _insert(self, node, st, val, index):
    if node is None: node = Node(sizeAlpha=self.sizeAlpha)
    if index == len(st): node.val = val
    else:
        v = ord(st[index])
        node.child[v] = self._insert(node.child[v], st, val, index+1)
    return node
```

```
def _longestprefix(self, node, st, index):
    if node is None: return -1
    if index == len(st) : return 0
    return 1+self._longestprefix(node.child[ord(st[index])], st, index+1)
```

$$R \times \text{num. strings} \leq \text{size(trie)} \leq R \times \text{num. strings} \times \text{average key length}$$

Ternary Search Tries



Save memory when
the alphabet is large
and the keys are long.

Each node has 3 children:
left for smaller in the alphabetic order
right for larger
middle for the next letters in the string

Exercise:
find, insert

Hybrid between BST & Tries.

II. Lempel-Ziv-Welsh Compression

Create a Code with Longer and Longer Prefixes

Look for the longest prefix already in the code, and extend it

Code: a map $A^* \rightarrow B^*$,
 A & B two alphabets.

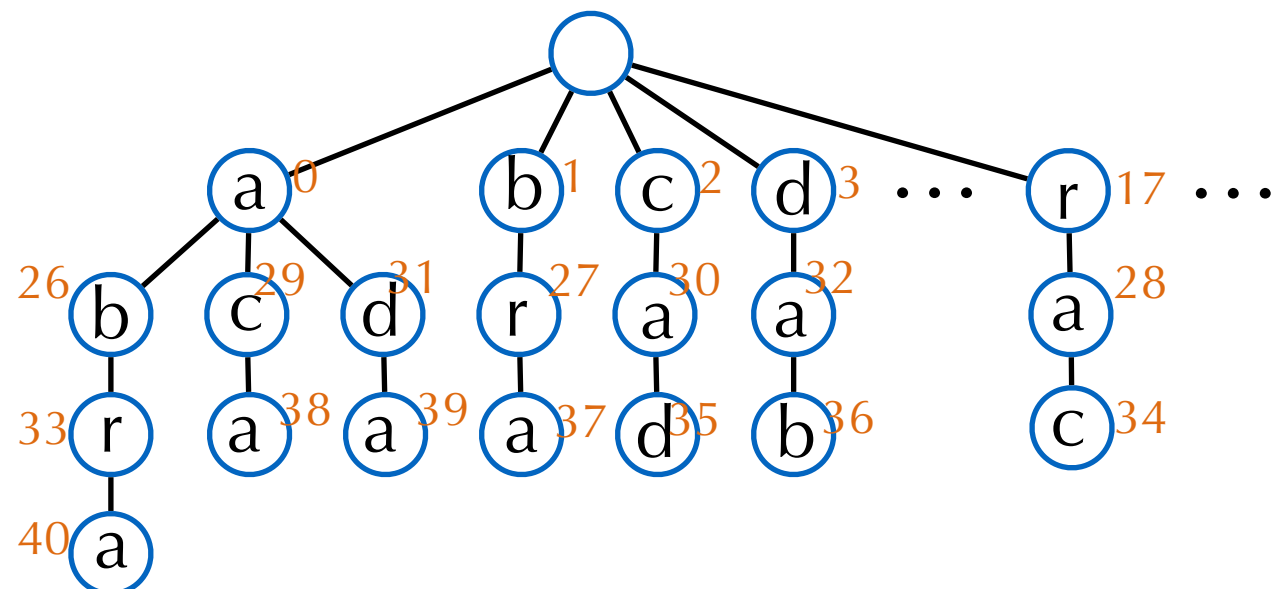
simplified example with $\Sigma = \{a, \dots, z\}$

abracadabracadabracadabra

0,1,17,0,2,0,3,26,28,30,32,27,29,31,33,0

input 25 characters in Σ

output 16 characters in \mathbb{N}



Trie growing during the encoding

$A = \{a, \dots, z\}$
 $B = \{0, \dots, 9\}$

a—z 0—25

ab 26

br 27

ra 28

ac 29

ca 30

ad 31

da 32

abr 33

rac 34

cad 35

... ...

Practical Experiment

```
_MAXASCII = 127      # 2^7 - 1
_SIZE = 2            # num bytes for the code words
_MAXCOUNT = 65535   # 2^(8*_SIZE) - 1
```

ASCII characters use
7 bits out of 1 byte

Here, each character
is encoded in 2 bytes

```
def compress(inname, outname):
    prefix = Trie(sizeAlpha=_MAXASCII)
    for c in range(_MAXASCII): prefix.insert(chr(c), c)
    count = _MAXASCII # initial size of the code
    with open(inname, "r", encoding="ascii") as f_in, \
        open(outname, "wb") as f_out:
        text = f_in.read() # make it a long string
        index = 0
        while index < len(text):
            k = prefix.longestprefix(text, index)
            f_out.write(prefix.find(text[index:index+k]).to_bytes(_SIZE, byteorder='big'))
            if index+k < len(text) and count < _MAXCOUNT:
                count += 1
                prefix.insert(text[index:index+k+1], count)
            index += k
        f_out.write(_MAXASCII.to_bytes(_SIZE, byteorder='big')) # marks end of file
```

```
>>> compress("hamlet.txt", "hamlet.lzw")
```

86258

179372

hamlet.lzw

hamlet.txt

Compression ratio:

2.08

Decoding: Reconstruct the Code on the Fly

0,1,17,0,2,0,3,26,28,30,32,27,29,31,33,0

No trie in the decoding phase

a
b
r
a
c
a
d
ab
ra
...

a—z	0—25
ab	26
br	27
ra	28
ac	29
ca	30
ad	31
da	32
abr	33
rac	34
cad	35
...	...

Exceptional case

a|b|ab|aba
0,1,26,28


ab 26
ba 27
aba 28

0,1,26,28
a
b
ab

ab 26
ba 27
? has to be aba

Resulting Code

```
def getnext(str,index):  
    res = str[index]  
    for i in range(1,_SIZE): res = res*256+str[index+i]  
    return res
```



```
def uncompress(inname,outname):  
    dic = [chr(c) for c in range(_MAXASCII)]+[None]*(_MAXCOUNT-_MAXASCII)  
    count = _MAXASCII # initial size of the code  
    with open(inname, "rb") as f_in, \  
        open(outname, "w", encoding="ascii") as f_out:  
        text = f_in.read() # make it a long string  
        val = dic[getnext(text,0)]  
        for index in range(_SIZE,len(text),_SIZE):  
            f_out.write(val)  
            new = dic[getnext(text,index)]  
            if new is None: new=val+val[0] # exceptional case  
            elif new==_MAXASCII: break # end of file  
            if count<_MAXCOUNT:  
                count += 1  
                dic[count]=val+new[0]  
            val = new
```

```
>>> uncompress("hamlet.lzw","hamlet2.txt")
```

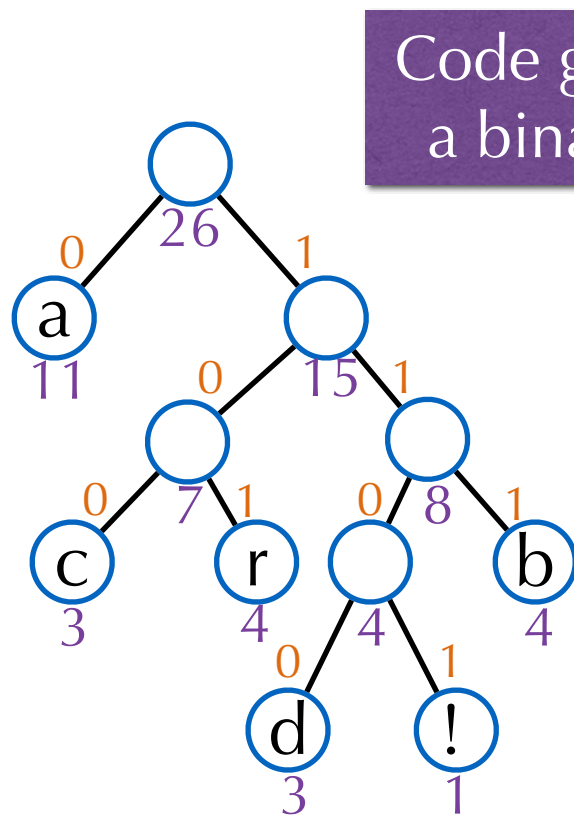
```
% diff -s hamlet.txt hamlet2.txt  
Files hamlet.txt and hamlet2.txt are identical
```

III. Huffman Encoding

Recall (CSE102)

Use fewer bits for frequent letters

abracadabracadabracadabra!



Code given by a binary trie

a	0
c	100
r	101
d	1100
!	1101
b	111

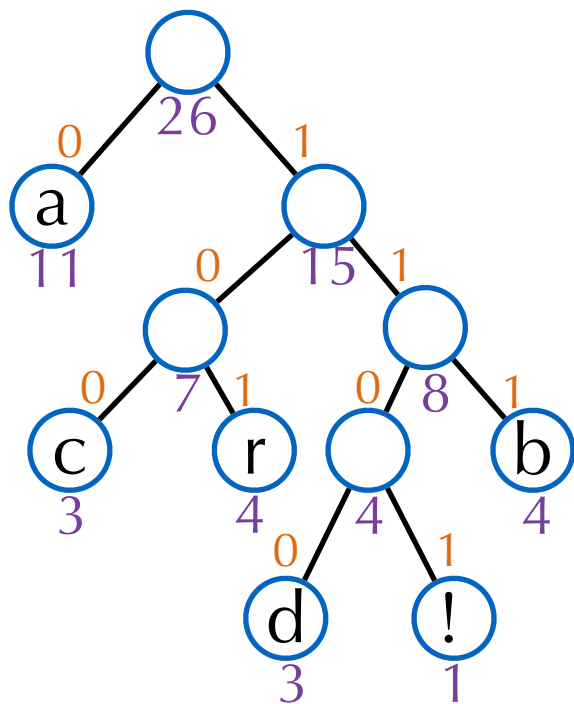
letter	num. occurrences
a	11
b	4
c	3
d	3
r	4
!	1

Encoding/Decoding straightforward given the trie

011110101000110001111010100011000111101010001100011110101101

26 letters \longrightarrow only 60 bits (plus trie)

Optimal Tries



Def. Weighted external path length:

$$W(T) := \sum_{\text{leaf } \ell} \text{weight}(\ell) \times \text{depth}(\ell).$$

number of occurrences

$W(T)$ = length of the encoded string

Observations: there is an optimal trie such that

two (sibling) leaves ℓ_1, ℓ_2 of minimal weights n_1, n_2 are at its lowest level;

Otherwise,
exchange

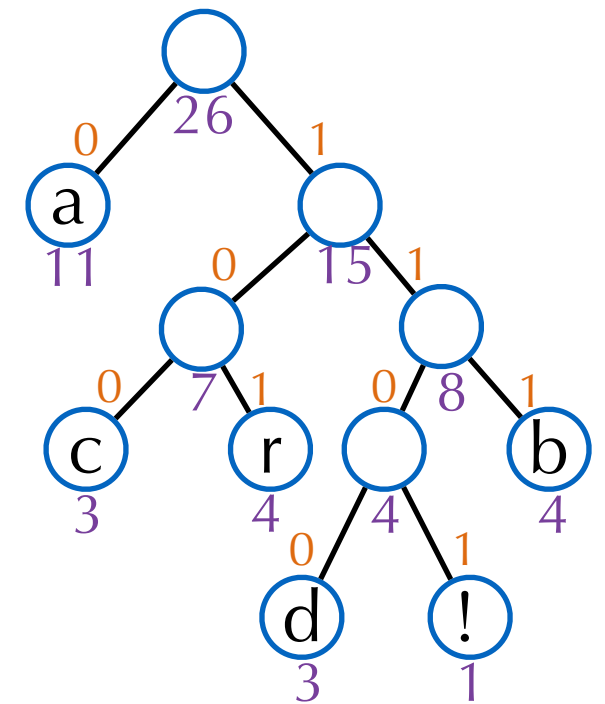
the trie T^* obtained by replacing ℓ_1, ℓ_2 by a leaf ℓ of weight $n_1 + n_2$ is optimal.

$$W(T) = W(T^*) + n_1 + n_2.$$

Huffman's Algorithm

```
class NodeHuffman:
```

```
    def __init__(self, val, child=[None, None]):  
        self.val = val  
        self.child = child
```



Start from a forest with
one tree per letter

Combine two trees
of minimal weight

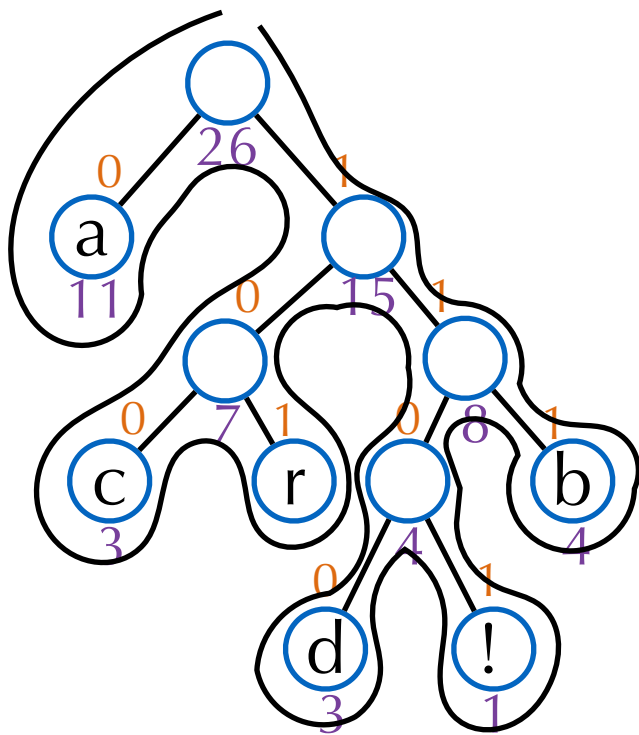
```
def maketree(text, wordsize):  
    numocc = getnumberoccurrences(text, wordsize)  
    minpq = PQ() # Priority queue of trees by weights  
    for c in range(2**(8*wordsize)):  
        if numocc[c]>0:  
            minpq.insert(-numocc[c], NodeHuffman(c))  
    while minpq.size>1:  
        n1, l1 = minpq.deletemax()  
        n2, l2 = minpq.deletemax()  
        minpq.insert(n1+n2, NodeHuffman(None, child=[l1, l2]))  
    return minpq.deletemax()[1]
```

Thm. Huffman's algorithm constructs a prefix-free code with minimal $W(T)$.

Proof by
induction on the
number of nodes

Communicating the Trie

Use a preorder traversal,
with 0 for nodes and 1 for leaves



01a001c1r001d1!1b

```
def writetrie(trie, out):  
    if trie.val is None:  
        out.extend('0')  
        writetrie(trie.child[0], out)  
        writetrie(trie.child[1], out)  
    else:  
        out.extend('1')  
        out.extend(format(trie.val, '08b'))  
  
def readtrie(barray, index)  
    if barray[index]:  
        return NodeHuffman(int(barray[index+1:index+9]), index+9)  
    else:  
        left, index = readtrie(barray, index+1)  
        right, index = readtrie(barray, index)  
        return NodeHuffman(None, child=[left, right]), index
```

Exercise:
modify for wordsize > 1

Summary

Lempel-
Ziv-Welsh

captures repetitions, regularity,
easy to decode,
works in one pass

Huffman

exploit differences in frequencies,
requires two passes,
the code must be transmitted
(possibly compressed) as well

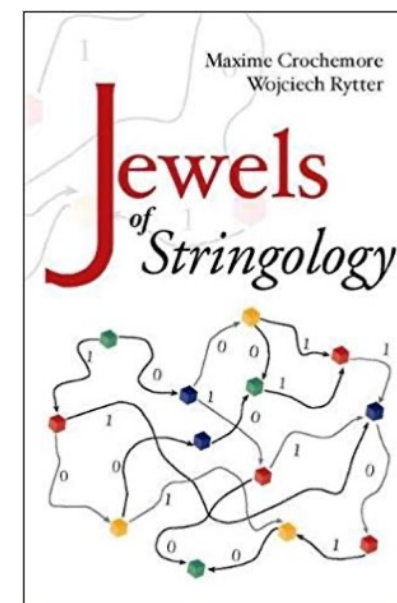
Compression routines (e.g., gzip) use (variants of) both!

text \longrightarrow LZ \longrightarrow Huffman

References for this lecture

The slides are designed to be self-contained.

They were prepared using the following books that I recommend if you want to learn more:



Next

Assignment: optimality of the Huffman encoding

Next tutorial: precomputing indexes with suffix arrays

Next week: P vs NP

Feedback

Moodle for the slides, tutorials and exercises.

Questions or comments: Bruno.Salvy@inria.fr