>>

Text Compression

∧ >>

❖ Text Compression

Problem: Efficiently encode a given string *T* by a smaller string *E*

Applications:

- Save memory and/or bandwidth

Huffman's algorithm

- computes frequency *f(c)* for each character *c*
- encodes high-frequency characters with short code word
- no code word is a prefix of another code word (e.g. can't have **`00`** + **`001`**)
- uses encoding tree to determine the code words
- many encodings are possible; aims to find optimal encoding

<< ^ >>

❖ ... Text Compression

Code ...

- mapping of each character to a binary code word (e.g. ascii)

Prefix code ...

- binary code such that no code word is prefix of another code word

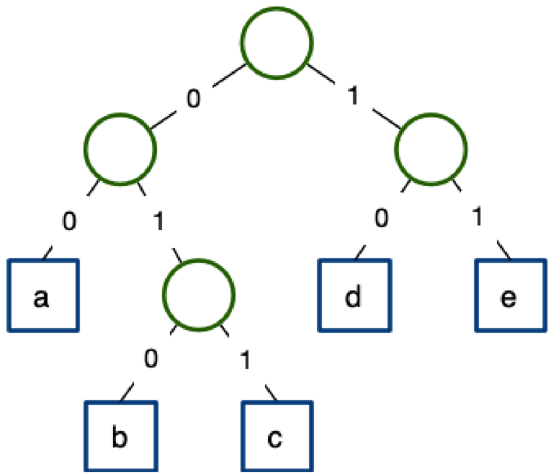
Encoding tree ...

- represents a prefix code
- each leaf stores a character
- code given by the path from the root to the leaf (0 for left, 1 for right)

❖ ... Text Compression

Example:

Encoding tree



Encoding table

char	a	b	c	d	e
code	00	010	011	10	11

Sample encoding

T = `deadcab`

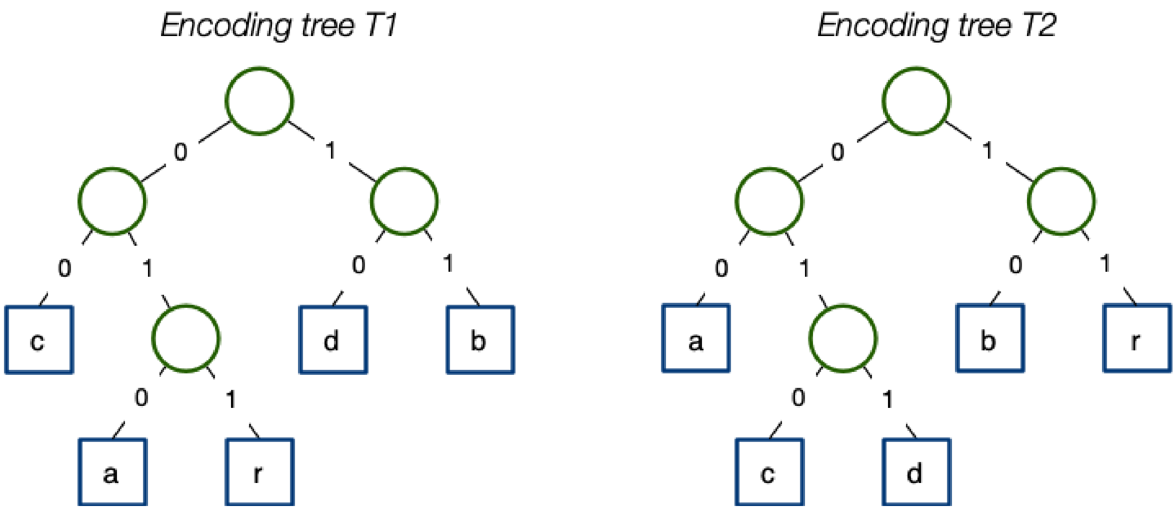
E = `1011001001100010`

T as ascii chars would require 56 bits; *E* needs only 16 bits

<< ^ >>

❖ ... Text Compression

Example: T = **`abracadabra`**



Encoding with T_1 is **`010.11.011.010.00.010.10.010.11.011.010`** i.e. 29 bits

Encoding with T_2 is **`00.10.11.00.010.00.011.00.10.11.00`** i.e. 24 bits

The dots are there only to distinguish characters; they are not part of the encoding.

❖ ... Text Compression

Text compression problem

Given a text T...

- find a *prefix code* that yields the *shortest encoding* of T

Some obvious strategies ...

- shorter codewords for frequent characters
- longer code words for rare characters

But how to ensure *optimal* encoding?

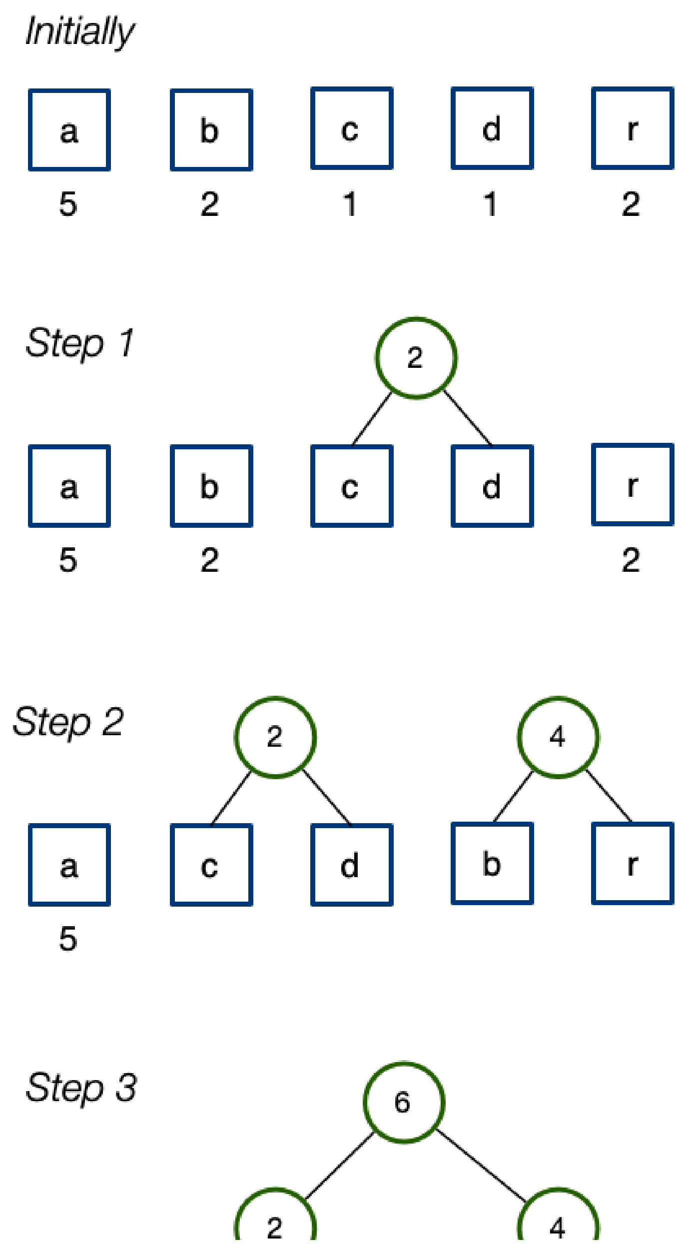
<< ^ >>

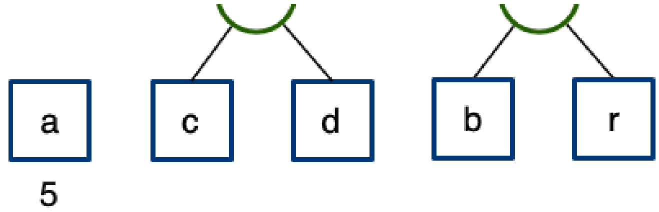
❖ ... Text Compression

Huffman's algorithm

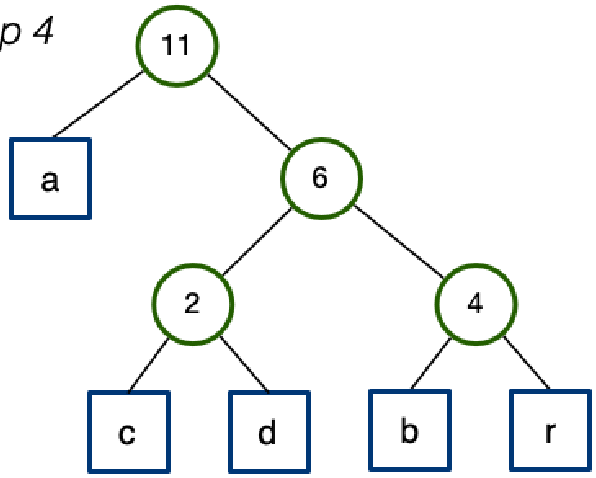
- computes frequency *f(c)* for each character
- successively combines pairs of lowest-frequency characters
- builds encoding tree "bottom-up"

Example: *T* = **`abracadabra`**





Step 4



Encoding table

char	a	b	c	d	r
code	0	110	100	101	111

❖ Huffman Code

Huffman's algorithm using a priority queue:

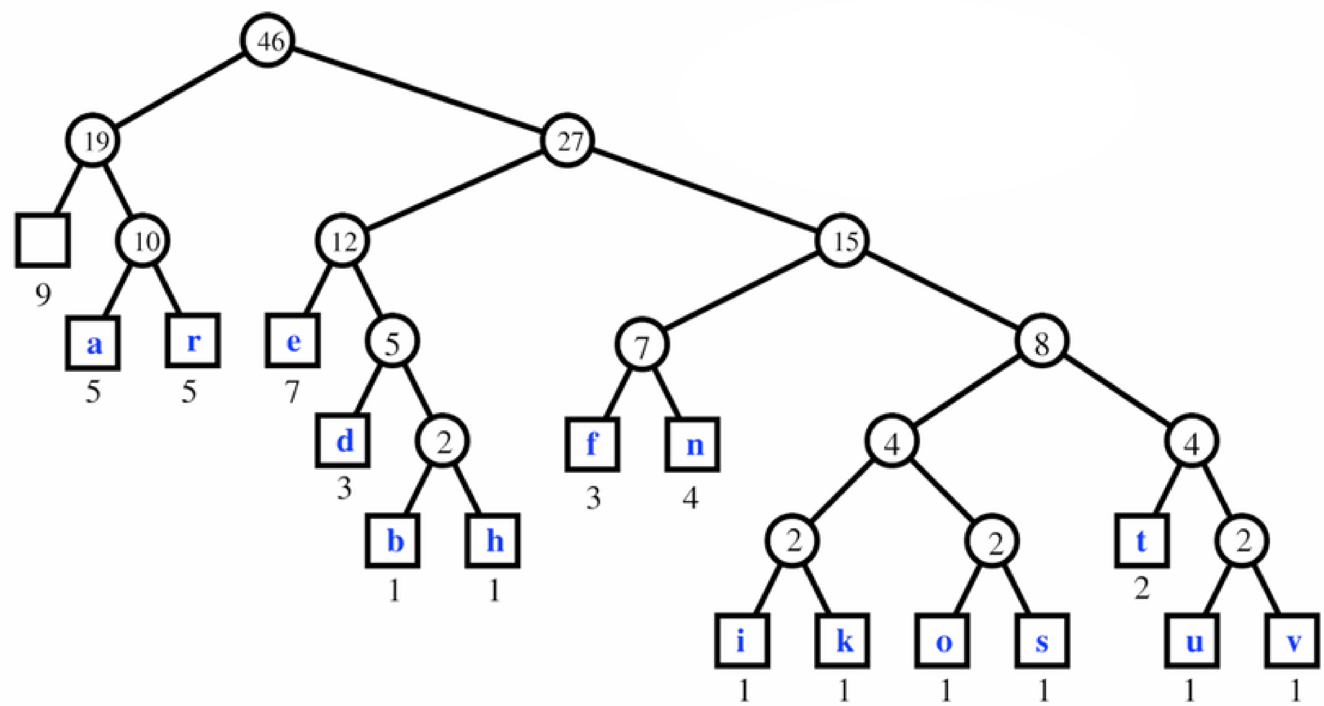
```
HuffmanCode(T):
|  Input  string T of size n
|  Output optimal encoding tree for T
|
|  compute frequency array
|  Q = new priority queue (ordered low key to high key)
|  for all characters c do
|     T = new single-node tree storing c
|     join(Q,T) with frequency(c) as key
|  end for
|  while |Q|≥2 do
|     f1 = Q.minKey(),  T1 = leave(Q)
|     f2 = Q.minKey(),  T2 = leave(Q)
|     T = new tree node with subtrees T1 and T2
|     join(Q,T) with f1+f2 as key
|  end while
|  return leave(Q)
```

<< ^ >>

❖ ... Huffman Code

Larger example: T = a fast runner need never be afraid of the dark

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



<< ^ >>

❖ Decompression

Decompression involves repeated traversal of paths in tree ...

```
decompress(B, T):
|  Input  bit-string B, encoding tree T
|  Output original string
|
|  start from root of Tree
|  for each b in Bits do
|     if b = 1 then
|        go right in Tree
|     else
|        go left in Tree
|     end if
|     if reached leaf then
|         print char in leaf
|         return to root of Tree
|     end if
|  end for
```

<< ∧

❖ Analysis of Huffman Encoding

Analysis of Huffman's algorithm:

- assume *length(T)* = m, vocab(T) = v
- build frequency table: scan entire input (m)
- build the tree: use frequency table (v) via priority queue ($\log_2 v$)

Gives complexity: *O(m + v log v)* time

Many variations exist to improve compression (e.g. gzip, bzip2, xz)

Produced: 9 Aug 2020