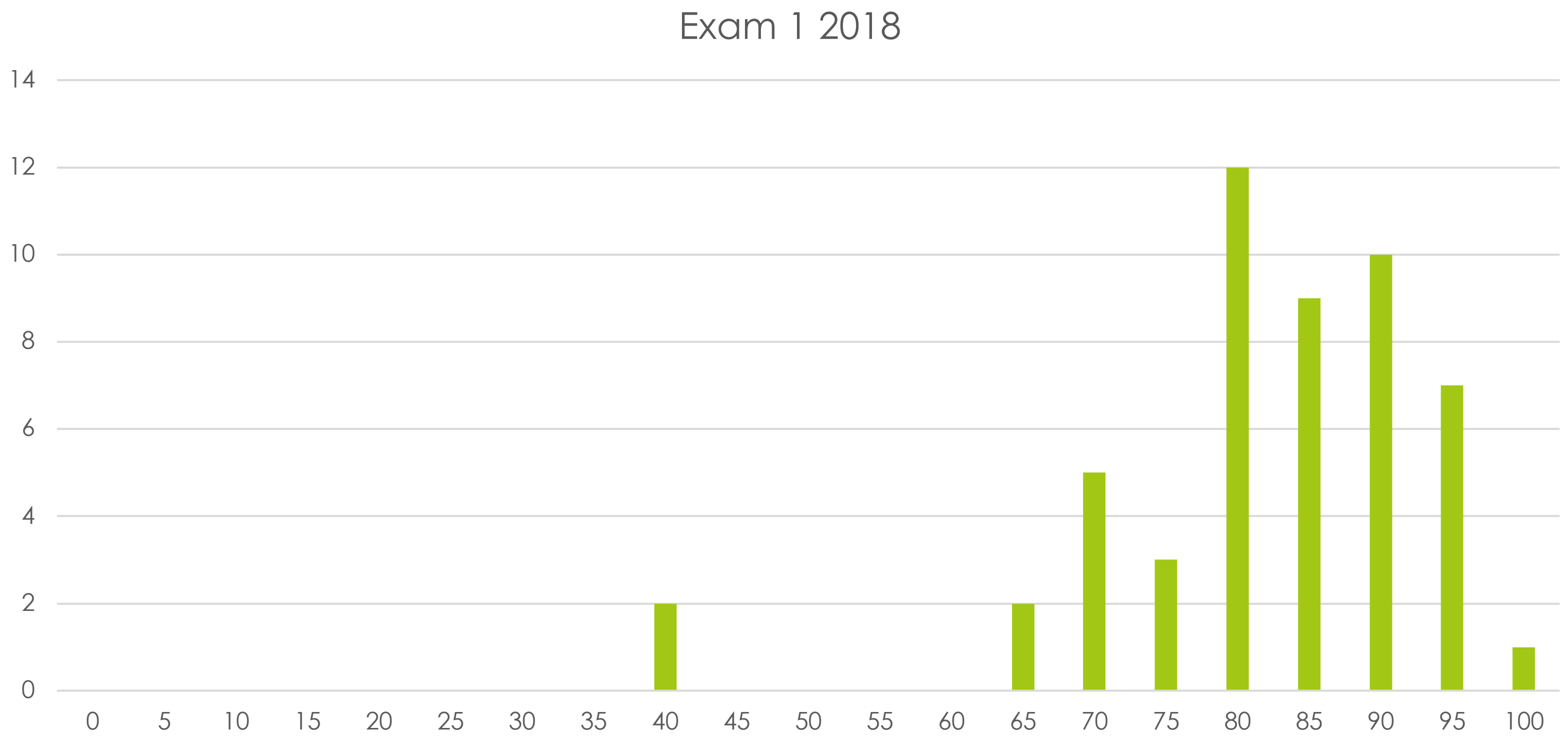
Data Representation and Compression

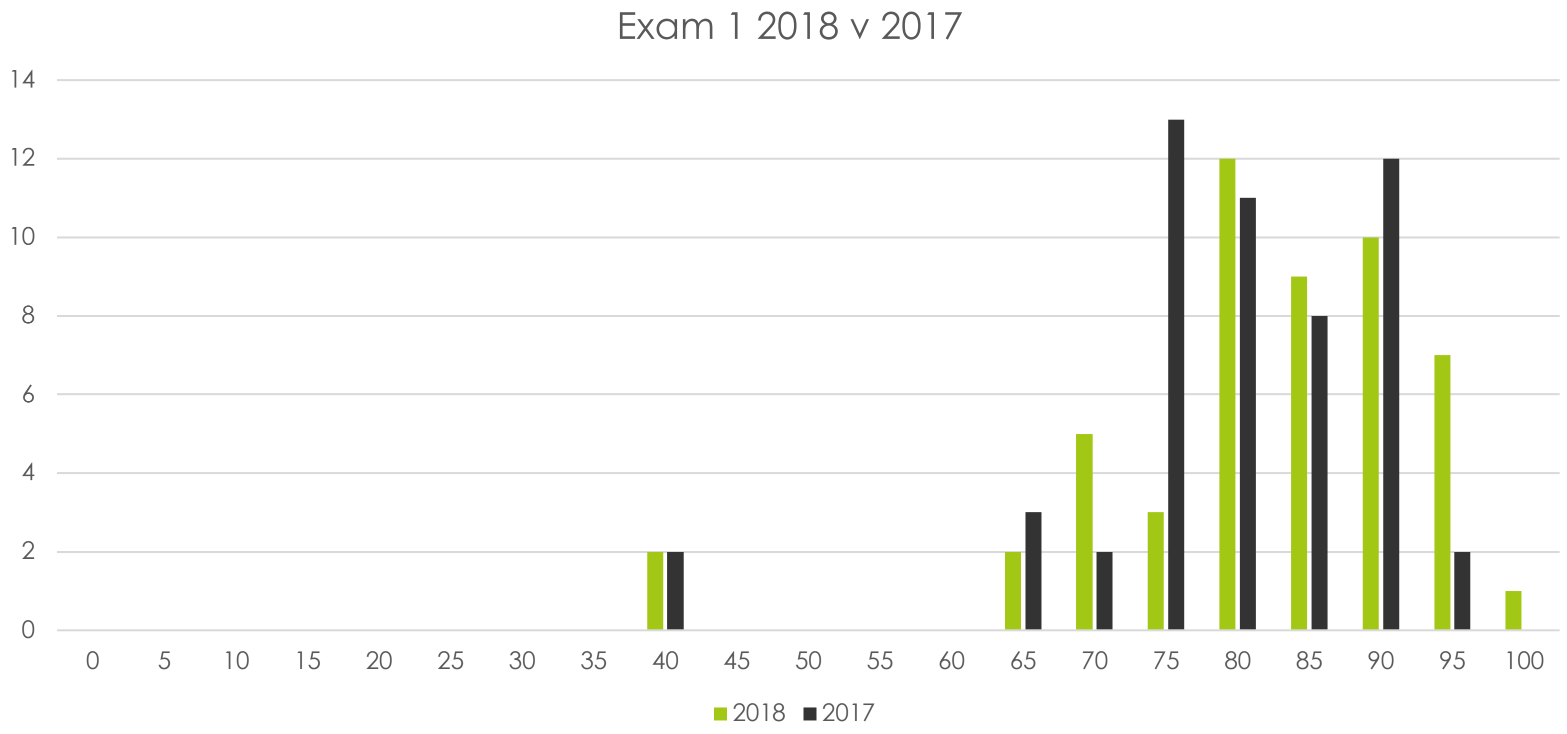
Announcements

- The first lab exam is tonight, during the lab session.
  - You may use your own computer
- PA last night?

Exam 1


Exam 1 2018
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100

Exam 1



Exam 1 2018 v 2017
14
12
10
8
6
4
2
0
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
2018
2017

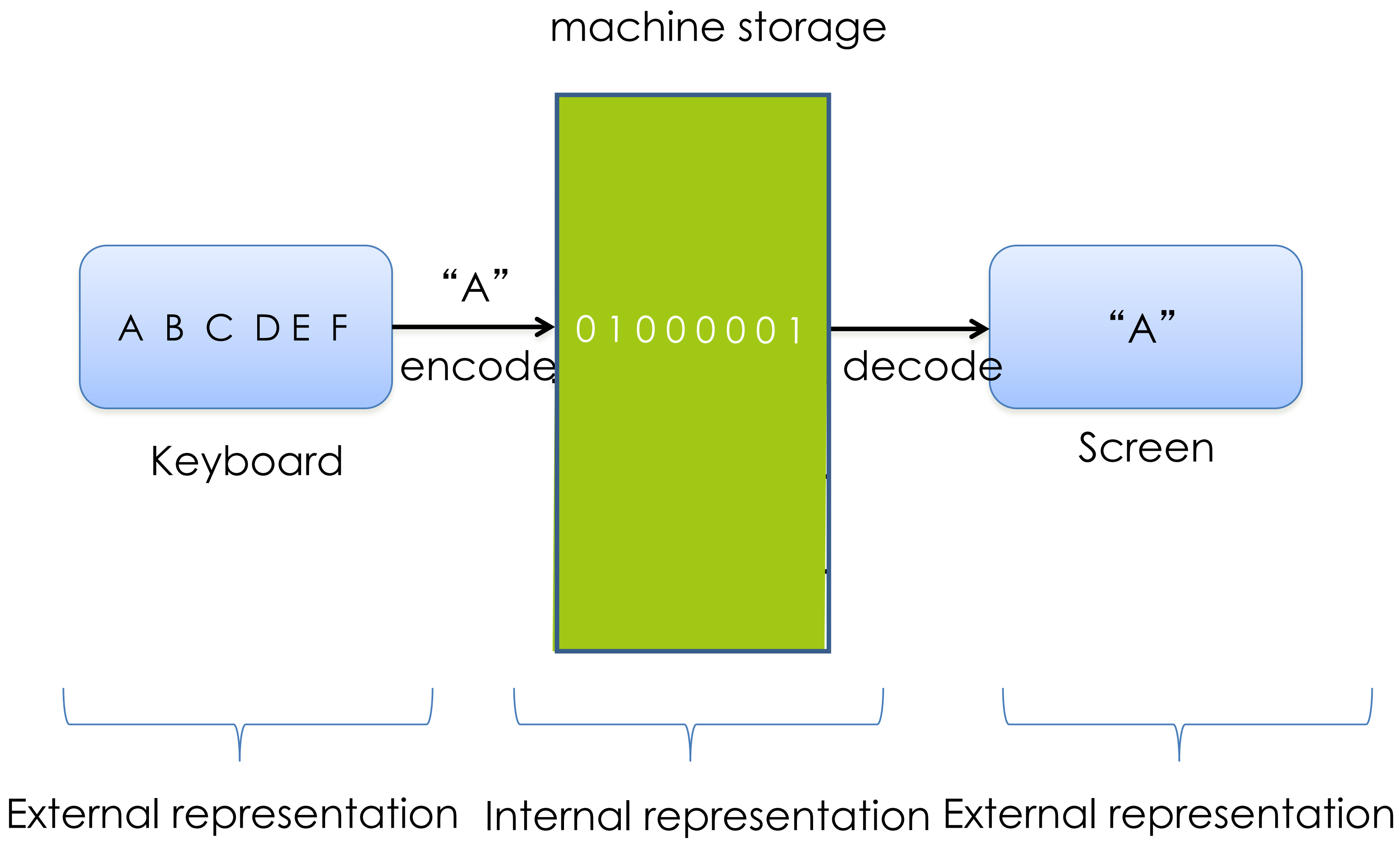
Review:

Data Representation

You should be able to

- Count in unsigned binary
  0, 1, 10, 11, 100, …
- Add in binary and know what overflow is
- Determine the sign and magnitude of an integer represented in two’s complement binary
- Determine the two’s complement binary representation of a positive or negative integer

Representing Data



machine storage

A B C D E F

Keyboard

“A”

encode

01000001

decode

“A”

Screen

External representation

Internal representation

External representation

Types interpret bits

- a 32-bit "word" might be
  1100 1100 1011 0111 0000 0000 0000 0000

- what this means depends on the machinery to interpret it, could be (**explore with 0xED**)

Type	Interpretation
“Raw” bits	1100 1100 1011 0111 0000 0000 0000 0000
Floating point number	6.59339×10^{-41}
String (Unicode UTF-16)	첷
RGB pixel color	🟨
Little-endian integer	47052

place-value syntax of numerals

representing non-negative integers (0, 1, 2, 3, …)

Place-value numerals (base 10)

- The *numeral* we write: 15627
- What it means:
  $7 \times 10^0 + 2 \times 10^1 + 6 \times 10^2 + 5 \times 10^3 + 1 \times 10^4$

- **Problem**: electronic circuitry for base-10 arithmetic is slow.
- **Solution**: use place-value numerals, but in base 2–*binary notation*

Place-value numerals in general

- Choose a number *b* for the **base** or **radix**
- Choose list of **digits**, there must be *b* of them
  - **base 10 example: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
  - **base 2 example: 0, 1**
  - **base 16 example: 0, 1, …, 9, A, B, C, D, E, F**
- To represent a quantity *n* in base *b*
  - integer divide *n* by *b* with remainder *r* (a **digit**)
  - repeat until the quotient is zero
  - the remainders are the digits in reverse order

Binary place-value example

- Base two, digits 0 and 1

- To represent “six”:
  - 6 // 2 = 3 remainder 0
  - 3 // 2 = 1 remainder 1
  - 1 // 2 = 0 remainder 1

Read the remainders from bottom to top to get bits from left to right

Binary numeral: 110

- What it means:
  $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 =$ “six”

Information Capacity and Range

- Remember: *k* bits can represent 2^k different things
- So *k*-bit binary numerals represent $0 \ldots 2^k - 1$
  - For $k = 3$,

000	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0	1	2	3	4	5	6	7

Ranges for typical computer “word” sizes

bits	minimum	maximum
8	0	$2^8 - 1$ (255)
16	0	$2^{16} - 1$ (65,535)
32	0	$2^{32} - 1$ (4,294,967,295)
64	0	$2^{64} - 1$ (18,446,744,073,709,551,615)

binary arithmetic

some familiar operations

Counting in binary

Binary numerals	Decimal equivalents
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11

Binary Arithmetic

- All the familiar methods work, but with only 1 and 0 for digits
- 1 + 1 = 10, 10 - 1 = 1, 10 + 1 = 11, ...
- Example:

```
1 1
 1010
+1010
-----
10100
```

Notice: we need more bits for the answer than we did for the operands.

Overflow: the first difficulty

- Machine word only has *k* bits for some **fixed** *k!*
- If *k* is 4, then we have **overflow** in the following:

```
1   1
  1010
+1010
-----
10100
```

- The machine retains only 0100 (the "least significant" bits), so (*n*+*n*) – *n* **not** always equal to *n* + (*n* – *n*)

Modular Arithmetic

- Dropping the overflow bit is **modular arithmetic**
- We can carry out any arithmetic operation modulo 2^k for the precision k. The example again for precision 4:

binary	**decimal**	
1 0 1 0	= 10	
+ 1 0 1 0	= 10	
(1) 0 1 0 0	= 20	= 4 mod 16

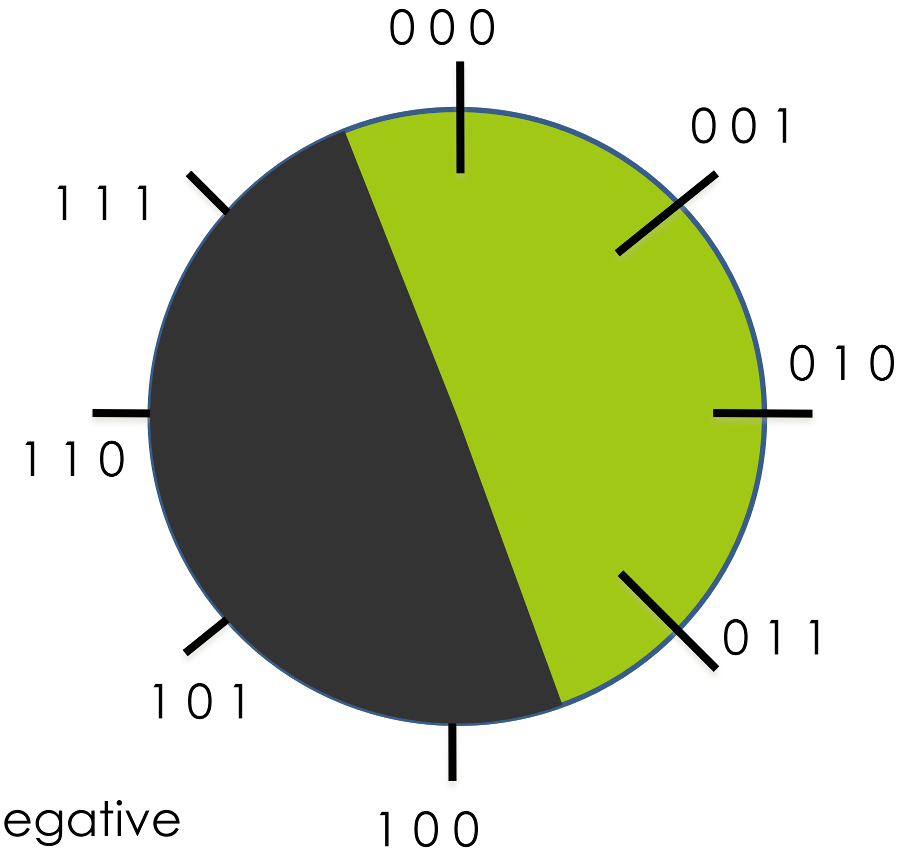
overflow can be ignored or signaled as an error

Representing negative integers

Two's complement is an approach for representing negative integers

- Define negative by addition: -x is value added to x to get 0
- Process:
  1. Write out the number in binary
  2. Invert the bits
  3. Add 1
- From and To two's complement use an identical process
- How does this work? Overflow…

All two's complement integers using 3 bits, arithmetic mod 8



positive integers and zero

negative integers

Bit pattern	**Decimal value**
0 1 1	**+ 3**
0 1 0	**+ 2**
0 0 1	**+ 1**
0 0 0	**0**
1 1 1	**- 1**
1 1 0	**- 2**
1 0 1	**- 3**
1 0 0	**- 4**

Adding + n to – n gives 0
For example: 011 + 101 = 000

Great! but how do we "read" two's complement integers?

- **Sign:** look at leftmost bit
  - **1 means negative, 0 means positive**

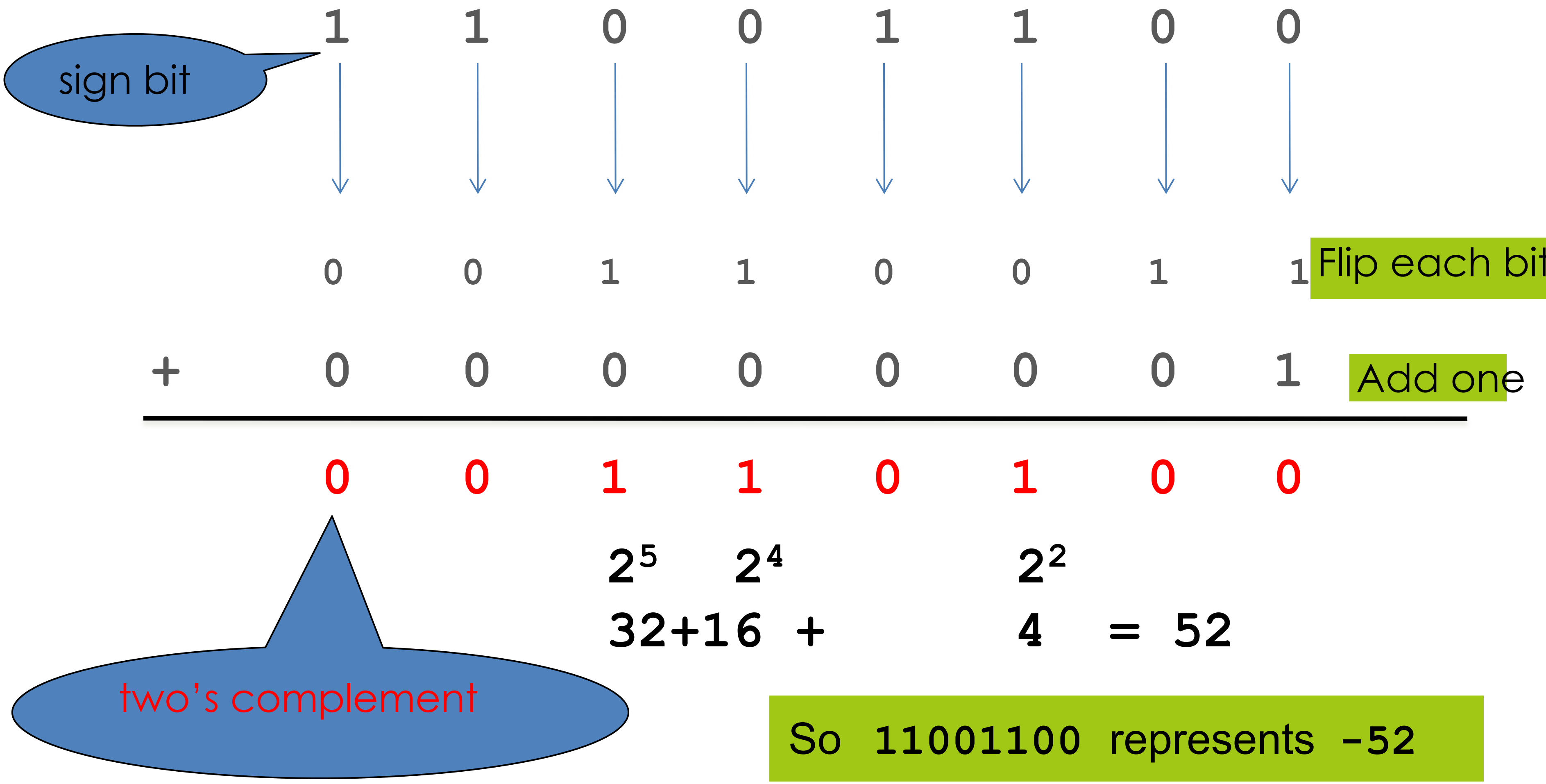
    e.g. with four bits 1010 represents a negative number
- **Magnitude:** if negative, compute the two's complement
  - flip each bit (one's complement)

    e.g. flip 1010 to get 0101
  - then add 1

    e.g. 0101 + 0001 = 0110, or
    $0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 + 0 \times 2^3 = 6$
  - **voilà!** 1010 represents negative six

Another Example

What value is this ?-bit signed integer?



sign bit

1 1 0 0 1 1 0 0

0 0 1 1 0 0 1 1 Flip each bit

\+ 0 0 0 0 0 0 0 1 Add one

0 0 1 1 0 1 0 0

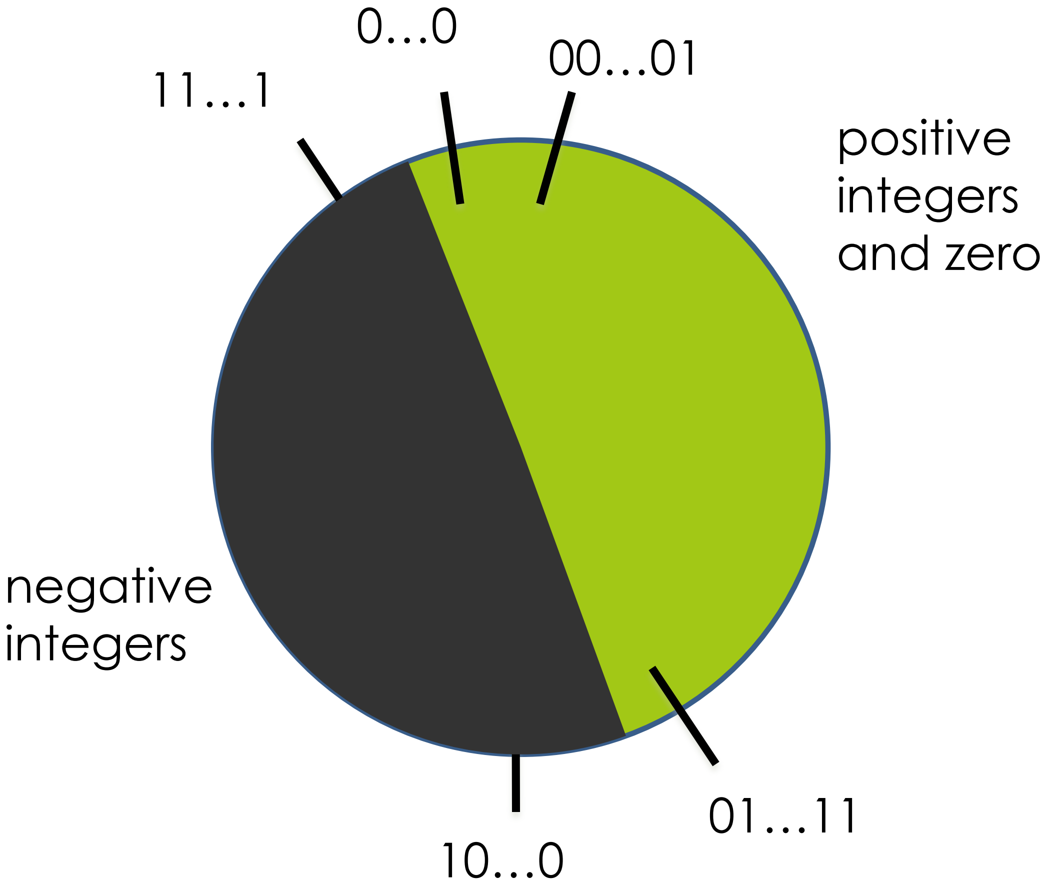
two's complement

2^5 2^4 2^2

32+16 + 4 = 52

So `11001100` represents -52

Range of Two's Complement Representations (for *k* bits)



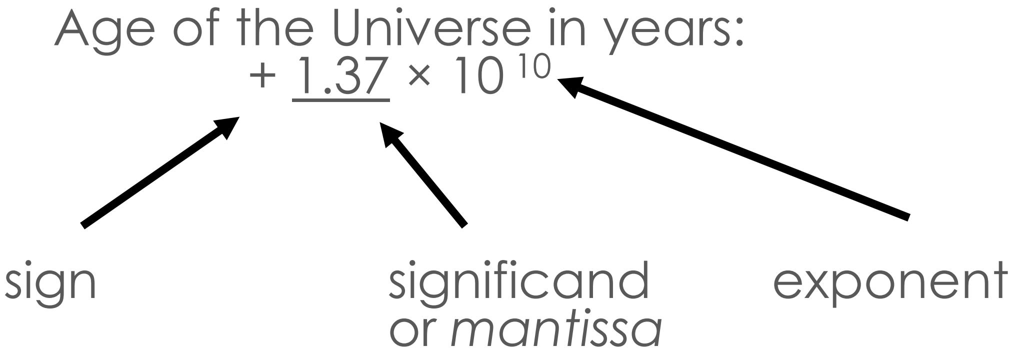
Bit pattern	**Decimal value**
00…00	0
00…01	+1
…	
01…11	$+2^{k-1}-1$
10…00	-2^{k-1}
…	
11…11	-1

From whole numbers to rational numbers

Real Numbers in the Machine?

- Real numbers measure **continuous** quantities; can we represent them exactly in the machine?
- Not possible with a fixed number of bits
- Can only approximate by rational numbers using **floating point representations**
- e.g. $\pi \approx 3.14159$

Floating point is based on scientific notation



Age of the Universe in years:

$+\ \underline{1.37} \times 10^{10}$

sign

significand
or *mantissa*

exponent

Idea: use same method, but with a binary number for each part (and remember, a fixed number of bits)

Binary and fractions

- Decimal 5.75 can be represented in binary as follows, because .75 = ½ + ¼ = $2^{-1} + 2^{-2}$

  $5.75 = 5 + 0.75$

  $= 101 + 0.11$ (i.e. $2^{-1} + 2^{-2}$)

  $= 101.11 = 1.0111 \times 10^{10}$

decimal

binary

In binary floating point the mantissa is a binary fraction, exponent is a binary integer, and the base of the exponent is always 2

101.11 has *mantissa* 1.0111 and *exponent* 10

Some Floating Point Anomalies

- Rounding error
  - remember, floating point with a fixed number of digits is an *approximation, no matter what base is used!*
  - in addition, there is no finite base two representation for 1/10
- Resolution
- Accumulation of errors: repeated operations may get further and further from the "true" value

Rounding in binary

```
>>> x = 1/10
>>> x
0.1
>>> y = 2/10
>>> y
0.2
>>> x + y
0.30000000000000004
>>> from decimal import Decimal
>>> Decimal(x)
Decimal('0.1000000000000000055511151231257827021181583404541015625')
>>> Decimal(y)
Decimal('0.200000000000000011102230246251565404236316680908203125')
>>> Decimal(x+y)
Decimal('0.3000000000000000444089209850062616169452667236328125')
>>>
```

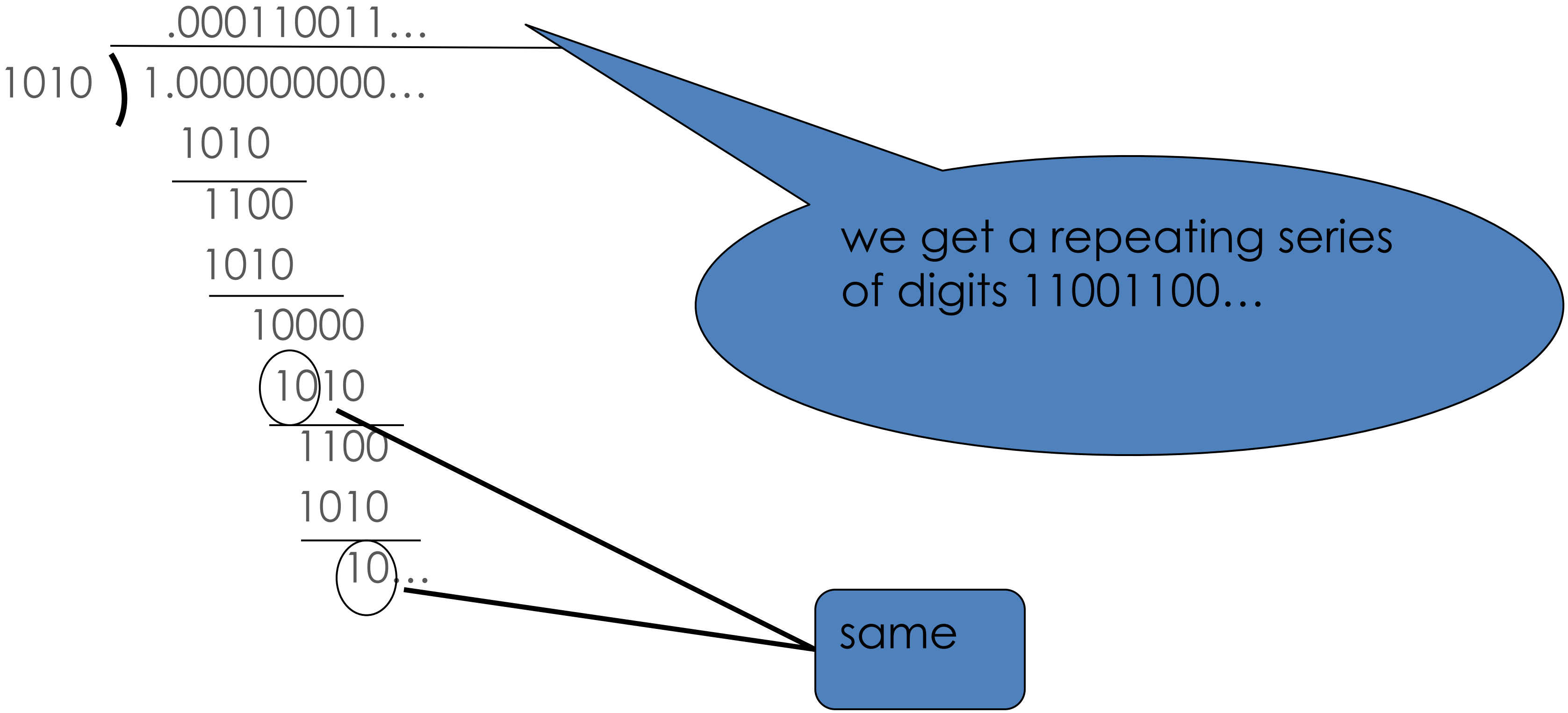
python prints a rounded value

Ack!
Whyyyy?

the actual value looks like this (in decimal)!

Why is 1/10 not exactly .1?

Let’s compute 1/10 using binary long division:



.000110011…

1010) 1.000000000…

1010

1100

1010

10000

1010

1100

1010

10…

we get a repeating series of digits 11001100…

same

Rounding in any base

- Floating point works with a finite fixed number of digits

- No matter what the base, some numbers can only be approximated
  - π, e, other irrationals
  - but also rationals needing more digits than we have in a machine word

data compression

squeezing out redundancy

Data Compression: Why?

- Faster transmission
  - e.g. digital video impossible without compression
- Cheaper storage
  - e.g. OS X Mavericks compresses data in memory until it needs to be used

Compression and decompression

- Reduce storage and for faster transfer of data over networks

Input → compress → *Compressed data* → decompress → *Output*

- Would like two easily computable functions:

  `compress(m)`

  `decompress(m)`

with **`len(compress(m)) < len(m)`**

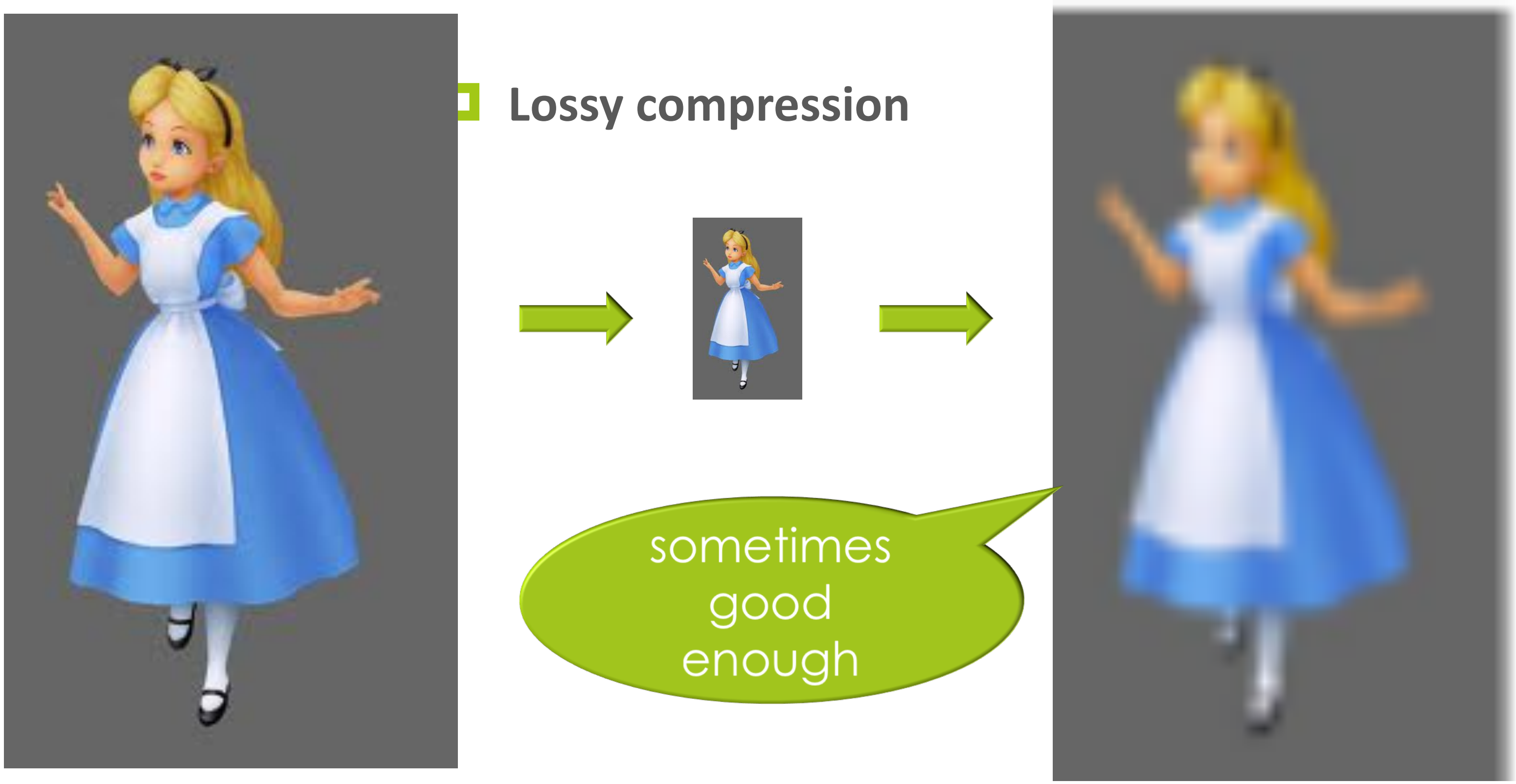
Data Compression: choices

- **Lossless compression**

good but
can be hard
to get

Data Compression: choices

- **Lossy compression**



Some Considerations

- What types of files would you use a lossless algorithm on?

- What types of files would you use a lossy algorithm on?

Data compression

- Types of compression
  - Lossless – encodes the original information exactly.

today

  - Lossy – approximates the original information.

tomorrow

Measuring information

What is information?

- information(*n*): knowledge communicated or received, or the act or fact of informing
  - Implicitly: a message, a sender, and a receiver
- How can we quantify how much information a message contains?

Which has more information?

023514

(A)

795

(B)

084664452746

(C)

Information

- More Digits = More Information
- Right?

Memorizing

Volunteer to memorize 10 digits

- 2737761413

Volunteer to memorize 100 digits

- 4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444

Memorizing

10-digit volunteer: What was the 8th digit?

100-digit volunteer: What was the 78th digit?

Which is easier to memorize?

Which contains more information?

Memorizing

Another volunteer to memorize 100 digits

- 4859937766824805299839179081504751450913524367800673622844553973169223820421306174607612086978543115

Is that harder to memorize than:

- 4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444

Why?

Which contains more information?

A key observation: redundancy

- Not all messages are equal
  - Some messages convey more information than others
  - Some messages are more likely to occur than others
- Our goal: encode messages so that each bit conveys as much information as possible

Idea 1: Algorithmic information theory

The amount of information

in a sequence of digits

is equal to

the length of the shortest program

that prints those digits.

Write a statement to print

```
44444444444444444444444444444444444444444444
44444444444444444444444444444444444444444444
444444444444
```

```
for i in range(100):
    print("4", end="")
```

Write a statement to print

48599377668248052998391790815047
51450913524367800673622844553973
16922382042130617460761208697854
3115

```
print("4859937766824805299839179081
  50475145091352436780067362284455
  39731692238204213061746076120869
  78543115”)
```

Therefore

4859937766824805299839179081504751450913524367800673622844553973169223820421306174607612086978543115

contains more information than

4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444

Pi and information

- How much information is stored in the digits of pi?
- In case they slipped your mind…

Pi 10000

314159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230
664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378
678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436
789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956
735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467
481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354
201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830
264252230825334468503526193118817101000313783875288658753320838142061717766914730359825349042875546873115956286388235
378759375195778185778053217122680661300192787661119590921642019893809525720106548586327886593615338182796823030195203
530185296899577362259941389124972177528347913151557485724245415069595082953311686172785588907509838175463746493931925
506040092770167113900984882401285836160356370766010471018194295559619894676783744944825537977472684710404753464620804
668425906949129331367702898915210475216205696602405803815019351125338243003558764024749647326391419927260426992279678
235478163600934172164121992458631503028618297455570674983850549458858692699569092721079750930295532116534498720275596
023648066549911988183479775356636980742654252786255181841757467289097777279380008164706001614524919217321721477235014
144197356854816136115735255213347574184946843852332390739414333454776241686251898356948556209921922218427255025425688
767179049460165346680498862723279178608578438382796797668145410095388378636095068006422512520511739298489608412848862
694560424196528502221066118630674427862203919494504712371378696095636437191728746776465757396241389086583264599581339
047802759009946576407895126946839835259570982582262052248940772671947826848260147699090264013639443745530506820349625
245174939965143142980919065925093722169646151570985838741059788595977297549893016175392846813826868386894277415599185
592524595395943104997252468084598727364469584865383673622262609912460805124388439045124413654976278079771569143599770
012961608944169486855584840635342207222582848864815845602850601684273945226746767889525213852254995466672782398645659
611635488623057745649803559363456817432411251507606947945109659609402522887971089314566913686722874894056010150330861
792868092087476091782493858900971490967598526136554978189312978482168299894872265880485756401427047755513237964145152
374623436454285844479526586782105114135473573952311342716610213596953623144295248493718711014576540359027993440374200
731057853906219838744780847848968332144571386875194350643021845319104848100537061468067491927819119793995206141966342
875444064374512371819217999839101591956181467514269123974894090718649423196156794520809514655022523160388193014209376
213785595663893778708303906979207734672218256259966150142150306803844773454920260541466592520149744285073251866600213
243408819071048633173464965145390579626856100550810665879699816357473638405257145910289706414011097120628043903975951
567715770042033786993600723055876317635942187312514712053292819182618612586732157919841484882916447060957527069572209
175671167229109816909152801735067127485832228718352093539657251210835791513698820914442100675103346711031412671113699
086585163983150197016515116851714376576183515565088490998985998238734552833163550764791853589322618548963213293308985
706420467525907091548141654985946163718027098199430992448895757128289059232332609729971208443357326548938239119325974
636673058360414281388303203824903758985243744170291327656180937734440307074692112019130203303801976211011004492932151
608424448596376698389522868478312355265821314495768572624334418930396864262434107732269780280731891544110104468232527
162010526522721116603966655730925471105578537634668206531098965269186205647693125705863566201855810072936065987648611
791045334885034611365768675324944166803962657978771855608455296541266540853061434443185867697514566140680070023787765
913440171274947042056223053899456131407112700040785473326993908145466464588079727082668306343285878569830523580893306
575740679545716377525420211495576158140025012622859413021647155097925923099079654737612551765675135751782966645477917
450112996148903046399471329621073404375189573596145890193897131117904297828564750320319869151402870808599048010941214
722131794764777262241425485454033215718530614228813758504306332175182979866223717215916077166925474873898665494945011
465406284336639379003976926567214638530673609657120918076383271664162748888007869256029022847210403172118608204190004
229661711963779213375751149595015660496318629472654736425230817703675159067350235072835405670403867435136222247715891
50495309844489333096340878076932599397805419341447377441842631298608099888

pi_tiny.c

- This C program is just 143 characters long!

```
long a[35014],b,c=35014,d,e,f=1e4,g,h;
main(){for(;b=c-=14;h=printf("%04ld",e+d/f))
for(e=d%=f;g=--b*2;d/=g)
d=d*b+f*(h?a[b]:f/5), a[b]=d%--g;}
```

- And it “decompresses” into the first 10,000 digits of Pi.

Program-size complexity

- There is an interesting idea here:
  - Find the shortest program that computes a certain output.
  - A very important idea in theoretical computer science. Can be used to define *incompressible data* (no shorter program will produce these data).

Idea 2: Shannon information theory

- We measure information content in bits
  - This is related to the fact that we can represent 2^k different symbols with k bits.
  - Turn the idea around and if we want to represent M different symbols, we need $\log_2 M$ bits
- **But** this is only true if the M symbols all have the same probability

The founder of information theory

Claude Shannon juggling sometime in the 1970s

Information content and bits

- Think of a file or network message as a symbol source
  - each symbol has a certain probability of occurring

- “information content” in Shannon’s sense is the same as the number of bits needed to represent the symbols in a symbol source

Probability and information content

- **Low probability** events have **high** information content; when you learn of them you get a lot of new information
    - *Barack Obama called me today!!!*
- **High probability** events have **low** information content.
    - *The sun rose in the east this morning. meh*
- Low probability events need more bits than high:

$\log_2(1/p)$ bits of information

Entropy the definition

$$H = \sum_{i=1}^{M} p_i \log_2 \frac{1}{p_i}$$

- Suppose a source of *M* different symbols with probabilities p_1, p_2, …, p_M,
- *H* is the ***entropy of the source*** (average number of bits/symbol)
  - For each probability p_i we multiply p_i times log $1/p_i$, and we add up the results

flips of an unfair coin

- **Example:** two symbols, **H** with probability 0.75 and **T** with probability 0.25;

  H = 0.75 * log (1/0.75) + 0.25 * log (1/0.25) ≈ 0.75 * .415 + 0.25 * 2 = .81125

- Roughly speaking this says each flip of our *unfair* coin carries less than one bit of information.

Encode / decode

squeezing out redundancy

2 common compression strategies:

- Exploit character-by-character non-uniformity
  - e.g., in English Pr[‘a’] = 0.0817 but Pr[‘b’] = 0.0149
- Exploit patterns between multiple characters
  - e.g. ‘q’ is almost always followed by ‘u’

Character-by-character coding

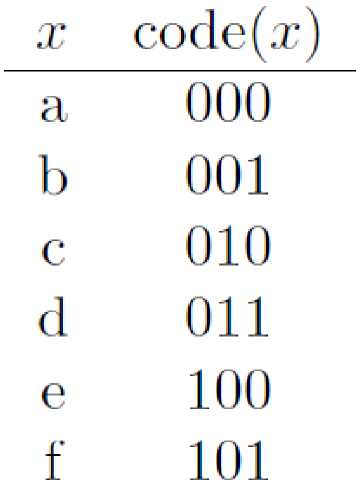
- Suppose each message $\mathbf{m}$ is a sequence of characters in some alphabet A = $\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_k\}$
  - e.g., A = the English alphabet,
    or A = all 7-bit ASCII characters

Character-by-character coding

- `encode(m)` outputs:
  1. An optional header containing any extra information needed for `decode`
  2. A sequence of bits encoding each character of `m`
  - i.e., `codetable(m)`

$$\texttt{code}(\texttt{m}_0)\,\texttt{code}(\texttt{m}_1)\ldots\texttt{code}(\texttt{m}_n)$$

- An example code table:



x	code(x)
a	000
b	001
c	010
d	011
e	100
f	101

Fixed length codes

x	code(x)
a	000
b	001
c	010
d	011
e	100
f	101

`encode("deadbeef")`

$$\underbrace{011}_{\texttt{d}}\underbrace{100}_{\texttt{e}}\underbrace{000}_{\texttt{a}}\underbrace{011}_{\texttt{d}}\underbrace{001}_{\texttt{b}}\underbrace{100}_{\texttt{e}}\underbrace{100}_{\texttt{e}}\underbrace{101}_{\texttt{f}}$$

What is **`decode(`**

"001000011010100011100") ?

$$\underbrace{001}_{\texttt{b}}\underbrace{000}_{\texttt{a}}\underbrace{011}_{\texttt{d}}\underbrace{010}_{\texttt{c}}\underbrace{100}_{\texttt{e}}\underbrace{011}_{\texttt{d}}\underbrace{100}_{\texttt{e}}$$

- Example: ASCII, Unicode
- Easy, but no compression

Codes

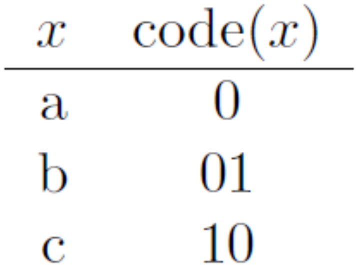
- A *codeword* is simply a binary string and a *code* is a *set* of codewords and their meanings.
- Must each codeword in a code necessarily have the same length? I.e. is every code a *fixed length code?*

(E.g., Morse code - not binary)

A non-code example

- Code words don’t all need to be the same length
- But not all codes have a unique decoding:

```
encode(“ba”) = 010
encode(“ac”) = 010
decode(“010”) = ?
```



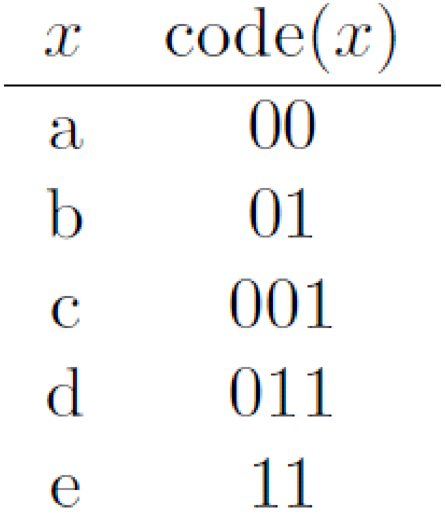
x	code(x)
a	0
b	01
c	10

Better, but more annoying…

- This code is fine in principle (everything is uniquely decodable).

- But decode is too hard. Try to decode

  0000101101010011



x	code(x)
a	00
b	01
c	001
d	011
e	11

Better, but more annoying…

x	code(x)
a	00
b	01
c	001
d	011
e	11

- What is **`decode(`** “00001011010011”**`)`** ?

  $$\text{“}\underbrace{00}_{\mathtt{a}}\underbrace{001}_{\mathtt{c}}\underbrace{011}_{\mathtt{d}}\underbrace{01}_{\mathtt{b}}\underbrace{00}_{\mathtt{a}}\underbrace{11}_{\mathtt{e}}\text{”}$$

  – How do you decode?

  – By trial and error, looking past the current the current, back and forth, hoping everything will work out in the end.

  – This look-ahead approach is too cumbersome.

What makes a code good?

- Uniquely decodable
- Easy to decode (no lookahead)
- Encoded messages are short

Prefix (a.k.a. *prefix-free*) codes

- A code is a *prefix code* if **`code(x)`** is **not** a prefix of **`code(y)`** for any **`x≠y`**
  - e.g.,

x	code(x)
a	000
b	001
c	010
d	011
e	100
f	101

(in fact, any fixed-length code is a prefix code)

Bad and annoying, revisited

- Is this a Prefix code?
- No: **`code('a')`** is a prefix of **`code('b')`**.

x	code(x)
a	0
b	01
c	10

Bad and annoying, revisited

- Is this a Prefix code?
- No: `code('a')` is a prefix of `code('b')`.

x	code(x)
a	0
b	01
c	10

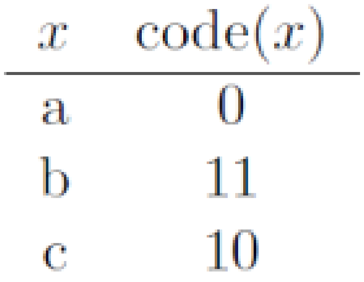
- Is this a Prefix code?

No: `code('a')` is a prefix of `code('c')`.
Also, `code('b')` is a prefix of `code('d')`.

x	code(x)
a	00
b	01
c	001
d	011
e	11

Another Example:

- Is this a Prefix code?
- Yes!



x	code(x)
a	0
b	11
c	10

Prefix codes are uniquely decodable

Let $\mathbf{b_0 b_1 \ldots b_n}$ be the bits of a coded message.

Read off the bits from left to right until $\mathbf{b_0 b_1 \ldots b_k}$ = `code(x)` for some **`x`**.

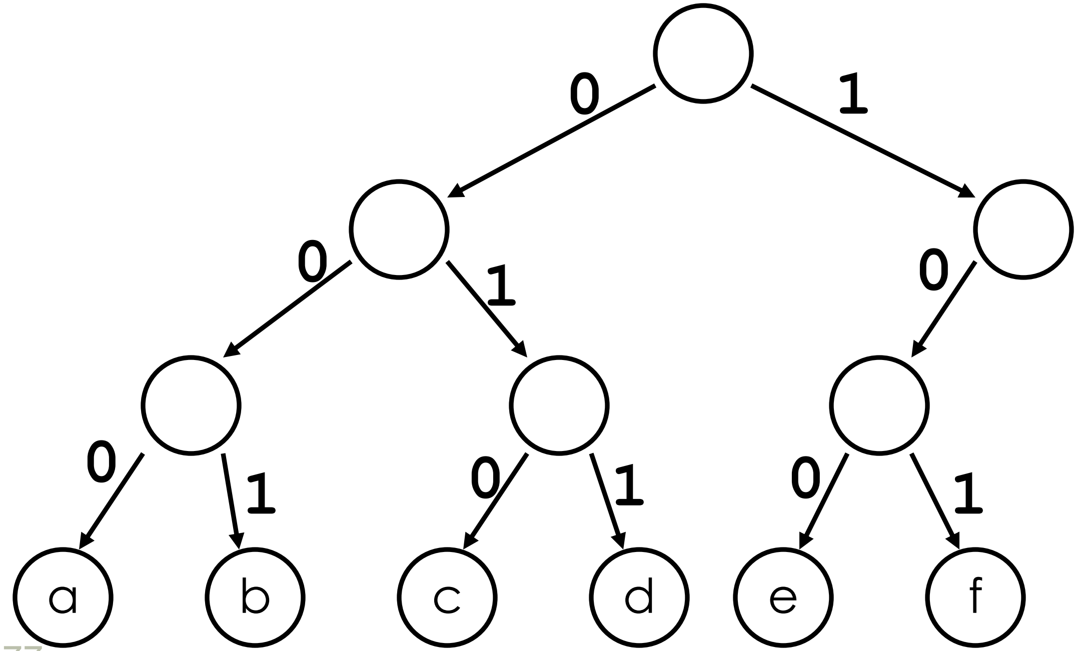
Note that **`k`** and **`x`** are both uniquely determined; otherwise we'd have found a prefix.

Repeat from b_{k+1} until done.

- Note: Prefix codes require no lookahead.

Decoding a prefix code message

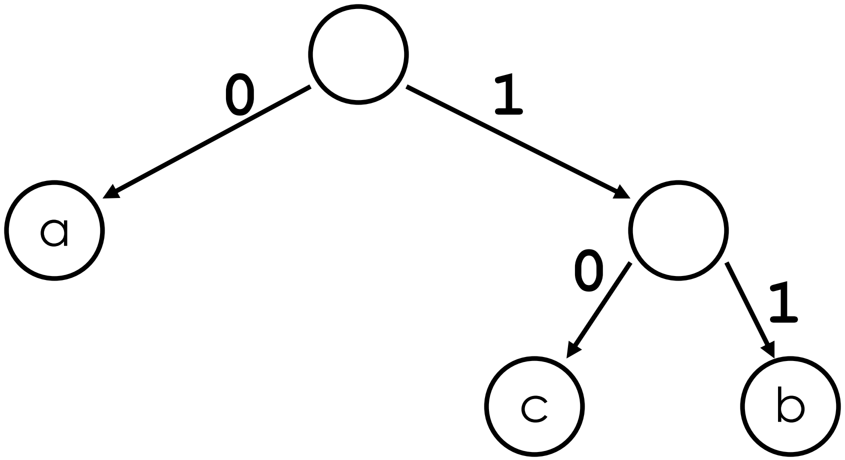
- Use a binary "prefix" tree
  - Start at root, walk left for each "0", walk right for each "1" until you reach a leaf
  - Return to root after you decode a character



x	code(x)
a	000
b	001
c	010
d	011
e	100
f	101

Decoding a prefix code message

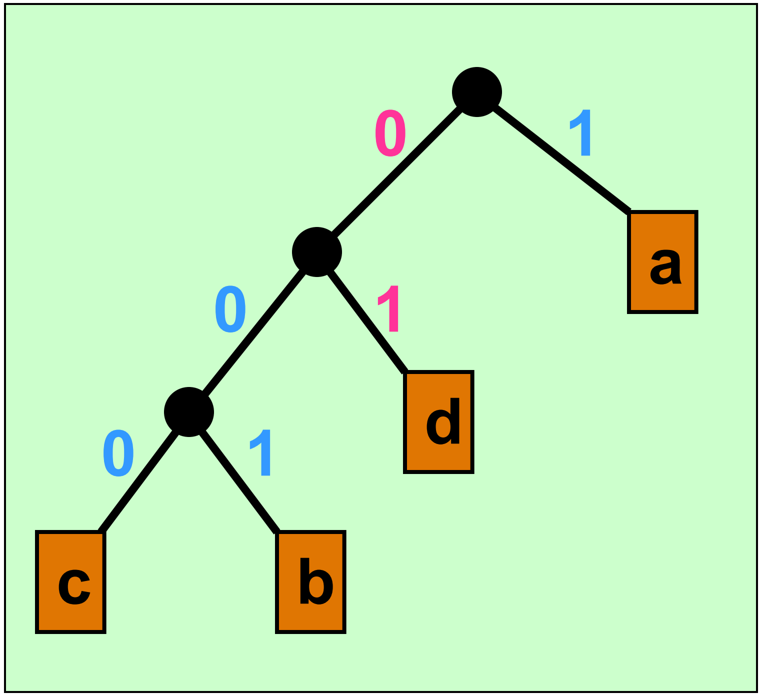
- Use a binary "prefix" tree
  - Start at root, walk left for each "0", walk right for each "1" until you reach a leaf
  - Return to root after you decode a character



x	code(x)
a	0
b	11
c	10

An optimal prefix tree is Full

- *Full*: every node
  - Is a leaf, or
  - Has *exactly* 2 children.

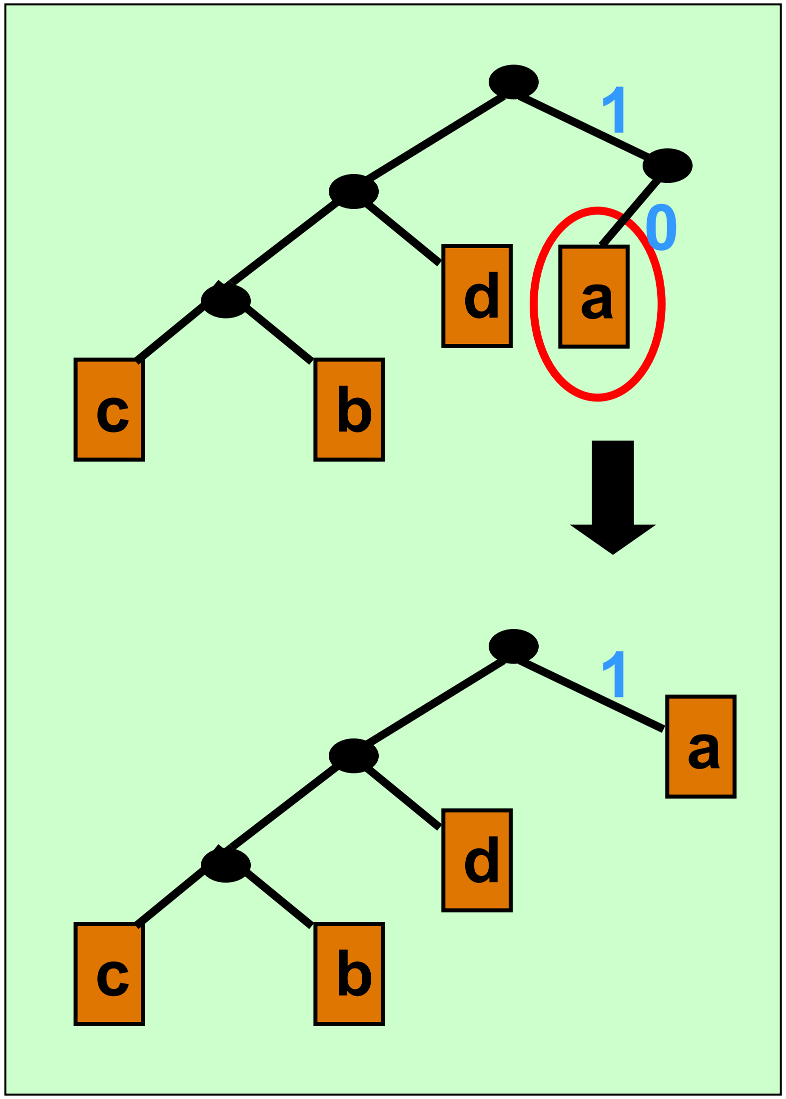


a=1, b=001, c=000, d=01

Why a full binary tree?

- A node with no sibling can be moved up 1 level, improving the code.

- An *optimal* prefix code for a string can *always* be represented by a full binary tree.

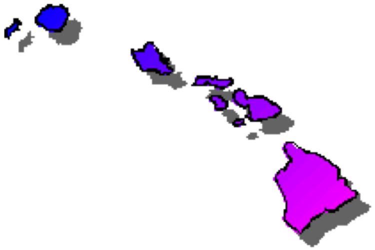


Today: Huffman Codes

- Compression:
  - Input: fixed-width character codes (e.g. 7-bit ASCII codes)
  - Output: Huffman codes (variable number of bits per character)
- Decompression:
  - Huffman codes to fixed-length codes
- Idea: squeeze out redundancy indicated by character probabilities

The Hawaiian Alphabet

- The Hawaiian alphabet consists of 13 characters.
  - ʼ is the okina which sometimes occurs between vowels (e.g. **KAMAʼAINA**)



ʼ
A
E
H
I
K
L
M
N
O
P
U
W

Specialized fixed-width encodings

- Suppose our text file is entirely in Hawaiian
- How many bits do we need for our 13 characters?
  - Are 3 bits enough? 000, 001, …, 111?
  - Are 4 bits enough? 0000, 0001, …, 1111?
- In general, for k *equally probable* characters we need $\lceil \log_2 k \rceil$ bits
- So for Hawaiian we need $\lceil \log_2 13 \rceil = 4$ bits

alternatively, figure it out using entropy

```
>>> h1 = [1/13]*13
>>> entropy(h1)
3.7004397181410926
```

not exactly 4 because some codes won't be used

Cost of Fixed-Width Encoding

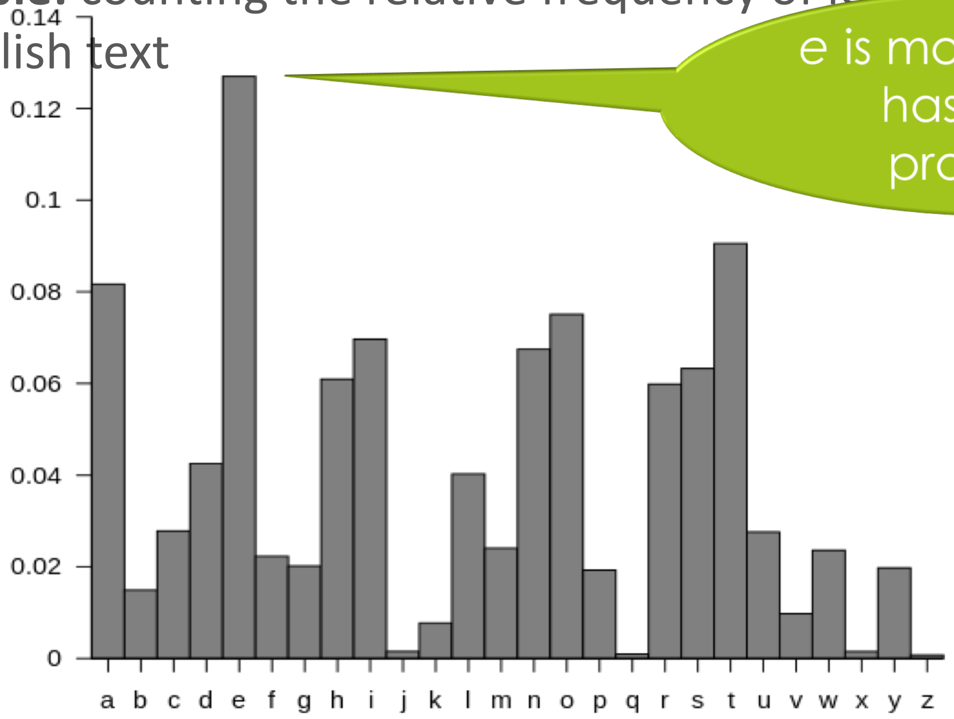
- With a fixed-width encoding scheme of *n* bits and a file with *m* characters, need *mn* bits to store the entire file.
  - Example: to store 1000 characters of Hawaiian we would need 4000 bits

- Can we do better? **Idea:** some characters are used much more often than others.
  - If we assign fewer bits to more frequent characters, and more bits to less frequent characters, then the overall length of the message might be shorter.

Use a method known as Huffman encoding named after David Huffman

Frequency counts as probabilities

- **Example:** counting the relative frequency of letters in a large corpus of English text



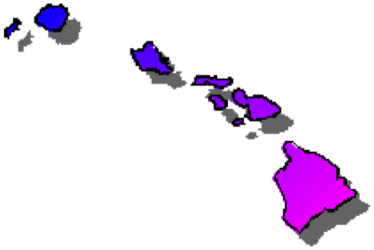
e is most frequent, has highest probability

image: Wikipedia

Hawaiian Alphabet Frequencies

- The table to the right shows each character along with its relative frequency in Hawaiian words.
- Smaller numbers mean less common characters
- Frequencies add up to 1.00 and can be viewed as *probabilities*

'	0.068
A	0.262
E	0.072
H	0.045
I	0.084
K	0.106
L	0.044
M	0.032
N	0.083
O	0.106
P	0.030
U	0.059
W	0.009



Entropy of the Hawaiian alphabet

- Using the probabilities we get

```
>>> a = [0.068, 0.262, 0.072, 0.045, 0.084, 0.106, 0.044,
0.032, 0.083, 0.106, 0.03, 0.059, 0.009]
>>> entropy(a)
3.3402829489193353
```

- Using Huffman's method we can get close to an *average* of 3.34 bits per character!
  - **example:** *ALOHA* can be encoded in 15 bits, only 3 bits per character

Huffman Coding: the process

1. Assign character codes
    1. Obtain character frequencies
    2. Use frequencies to build a *Huffman tree*
    3. Use tree to assign variable-length codes to characters (store them in a table)
2. Use table to encode (compress) ASCII source file to variable-length codes
3. Use tree to decode (decompress) to ASCII

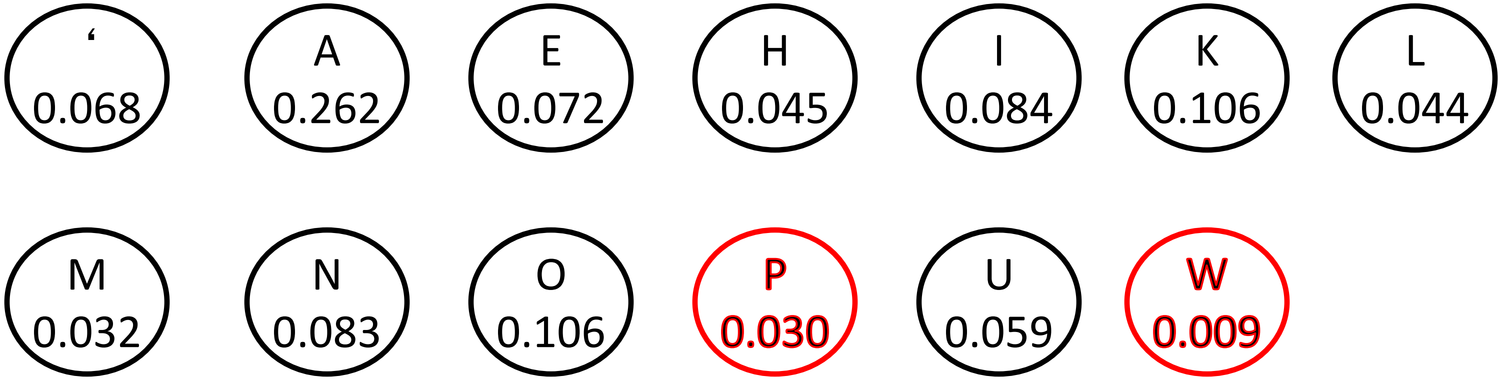
Key Idea

- Intuitively, place frequent characters near root (i.e., give them short codes)

- Build the prefix tree bottom up:
  - Consider leaves at maximum depth first

Building The Huffman Tree

- We use a tree structure to develop the unique binary code for each letter.
- Start with each letter/frequency as its own single-node tree
- Find the **two lowest-frequency** trees

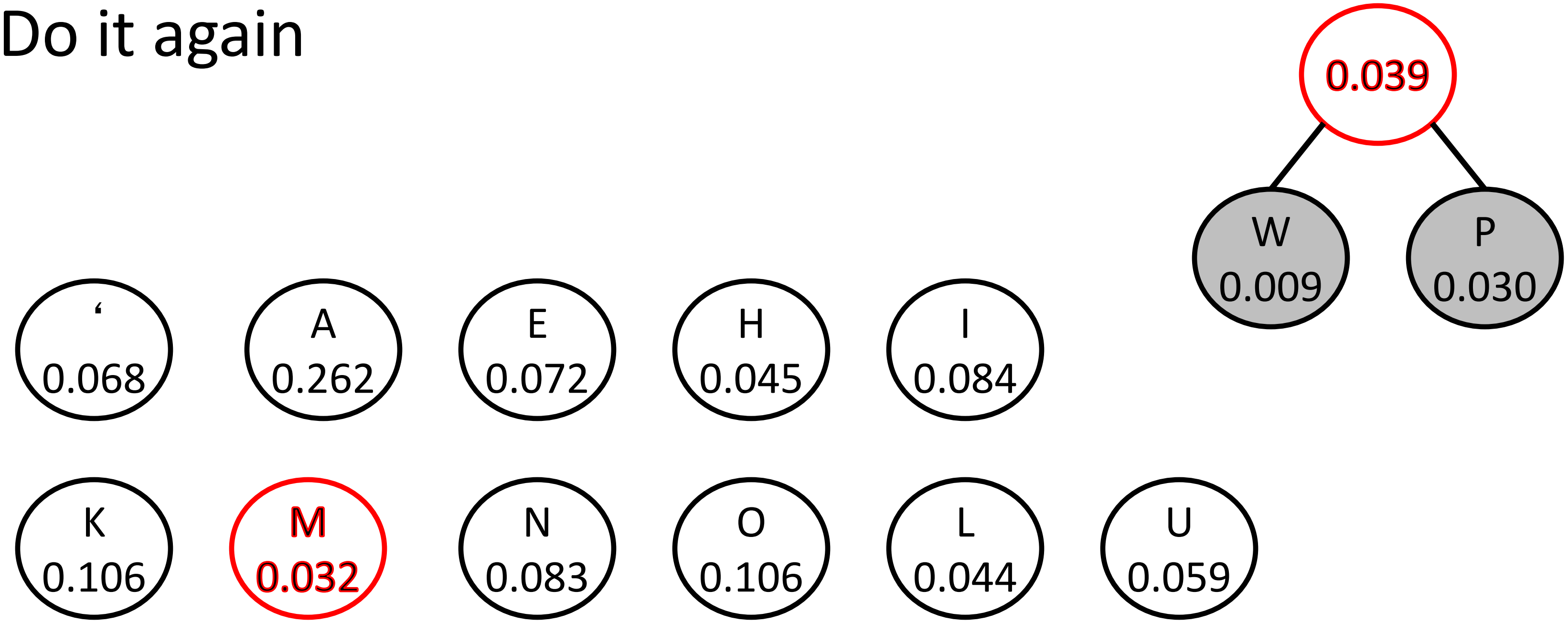


‘ 0.068
A 0.262
E 0.072
H 0.045
I 0.084
K 0.106
L 0.044

M 0.032
N 0.083
O 0.106
P 0.030
U 0.059
W 0.009

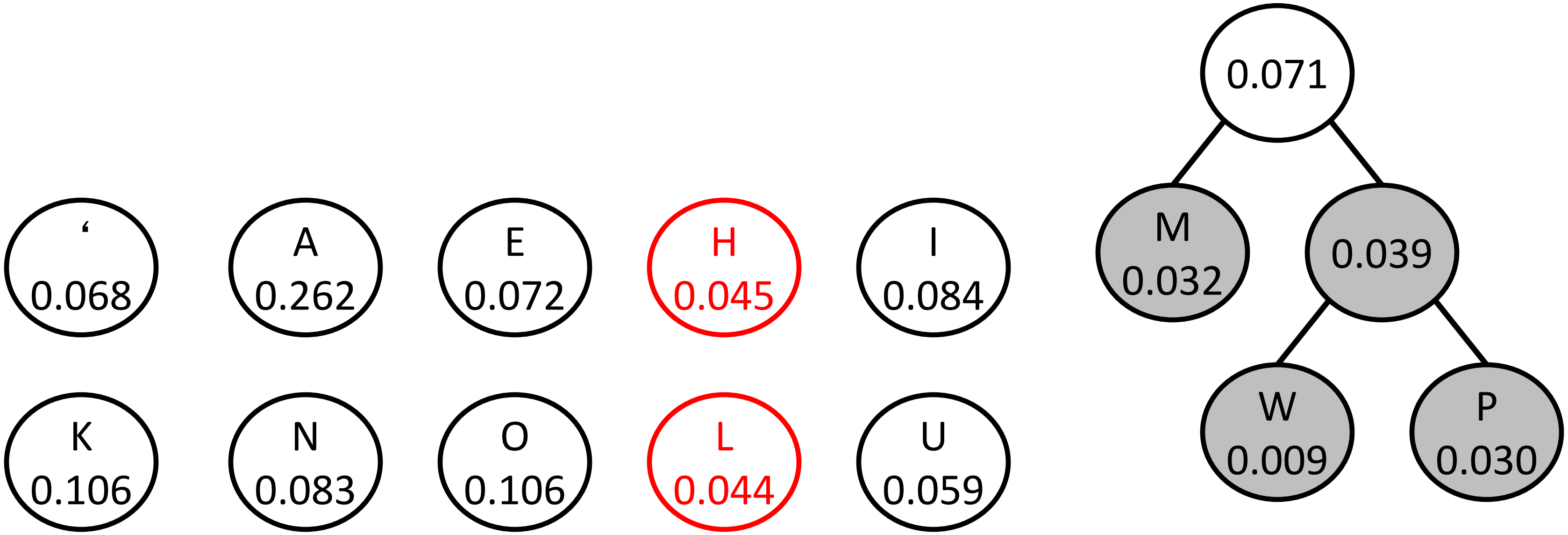
Building The Huffman Tree

- Combine **two lowest-frequency** trees into a tree with a new root with the sum of their frequencies.
- Do it again

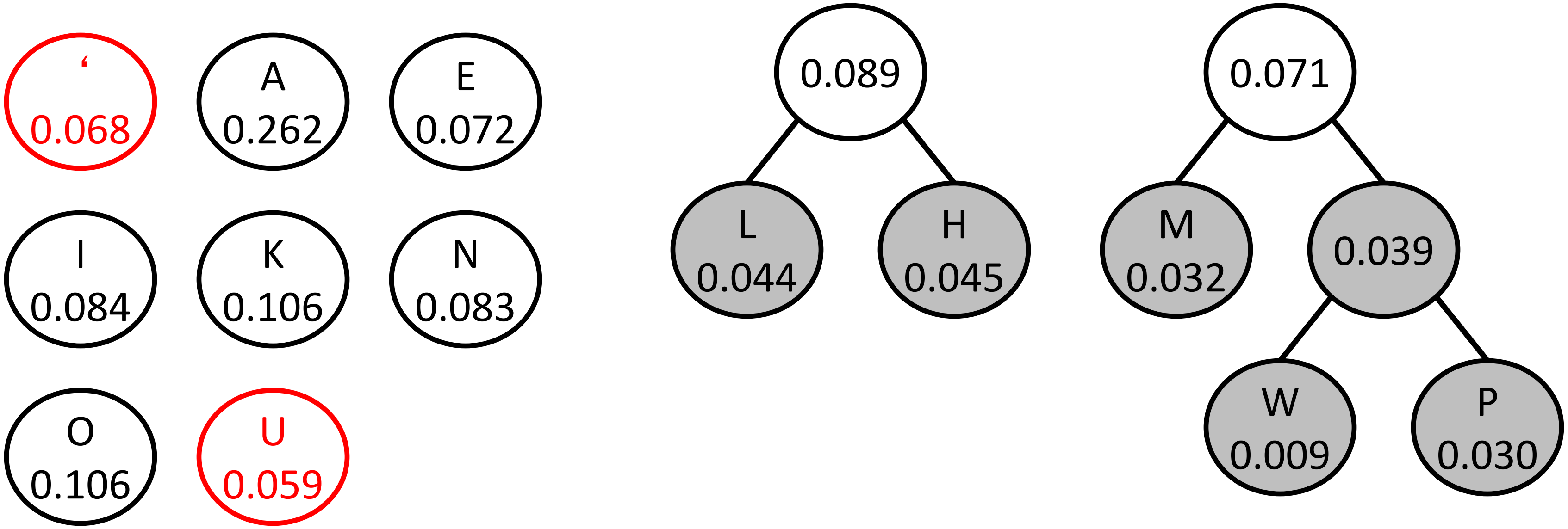


Building The Huffman Tree

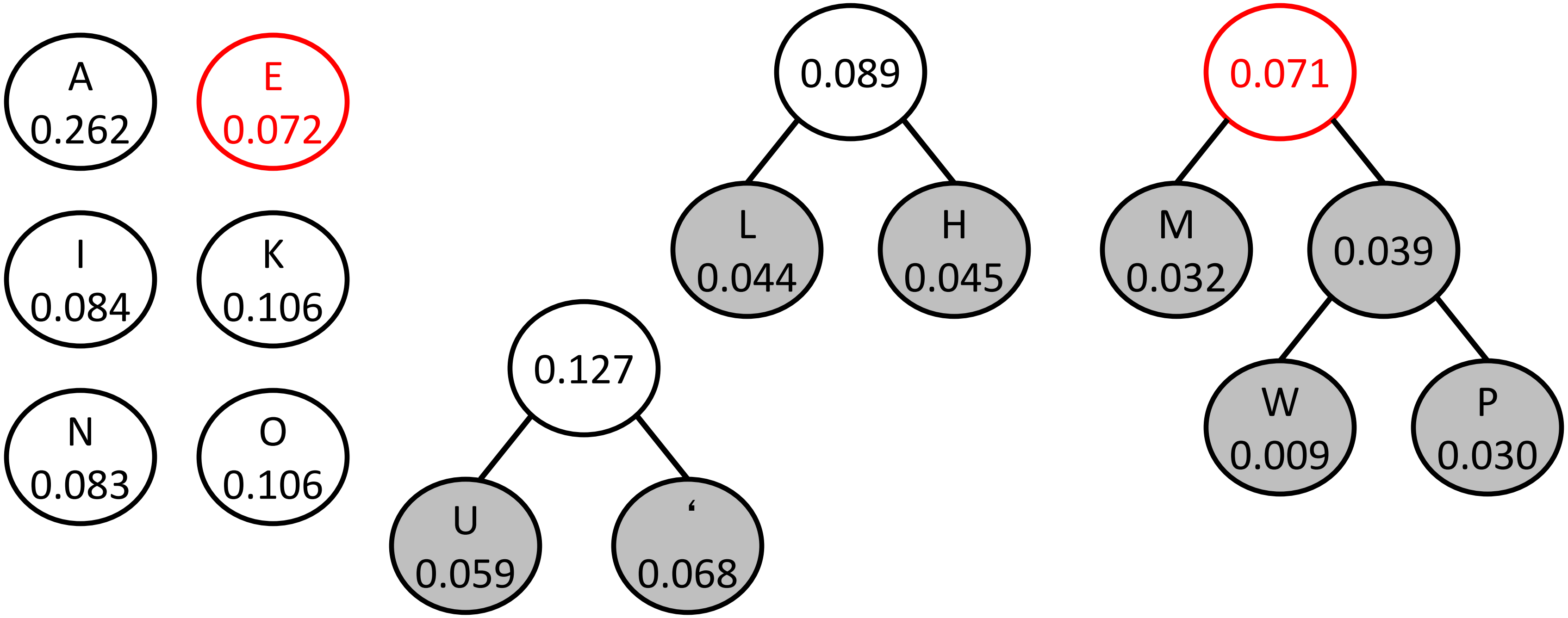
- …and again, as many times as possible



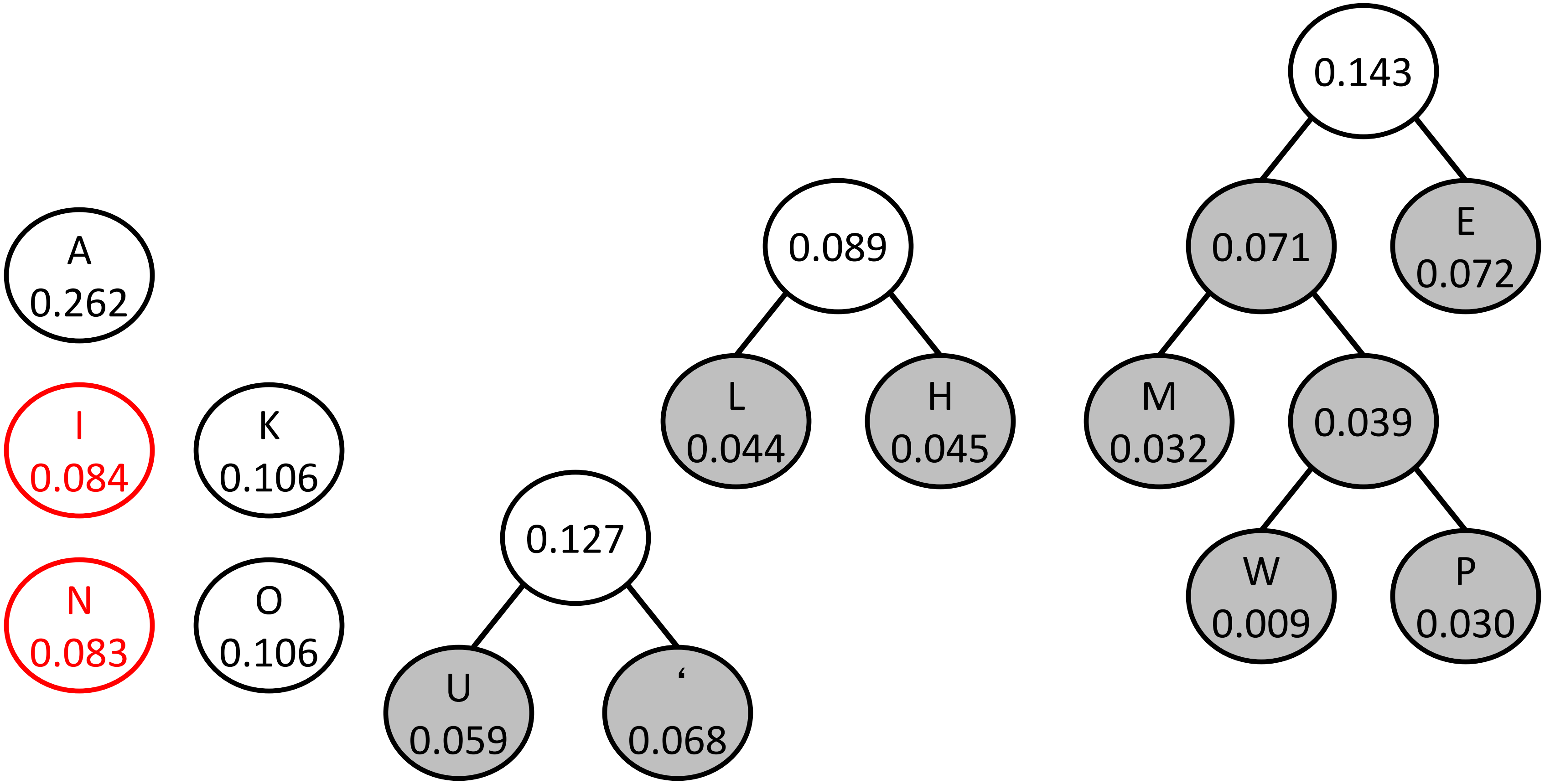
Building The Huffman Tree



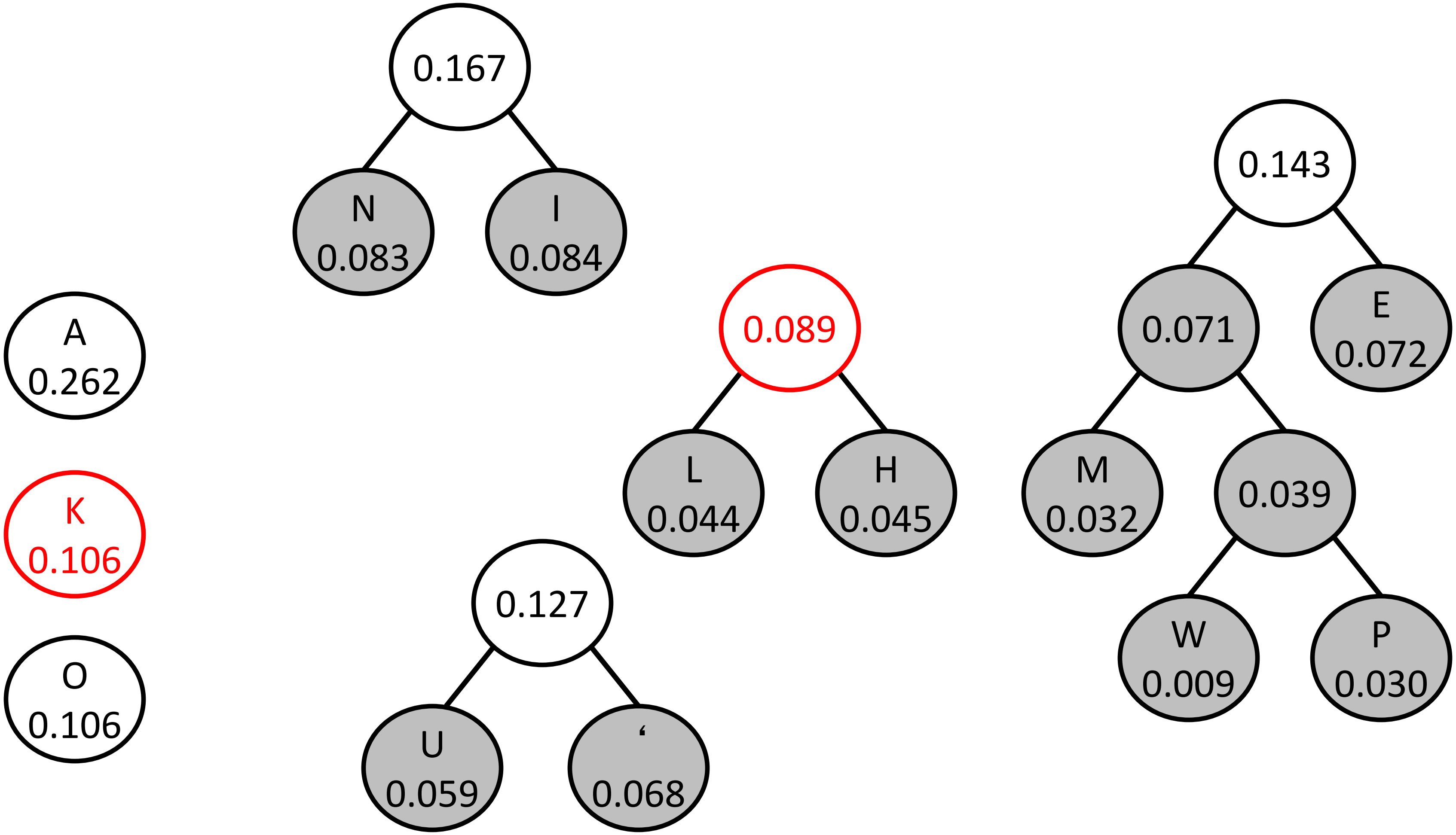
Building The Huffman Tree



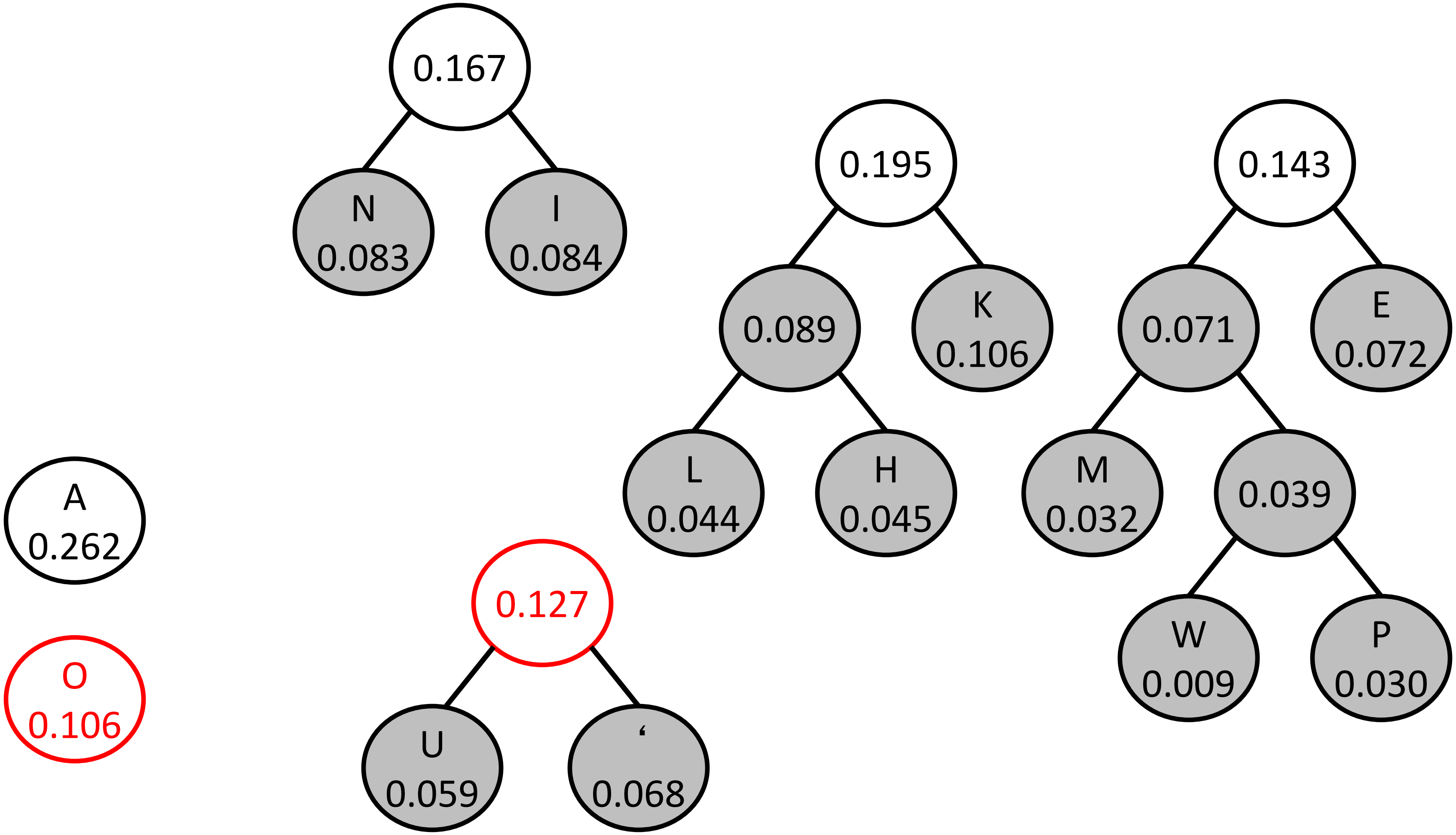
Building The Huffman Tree



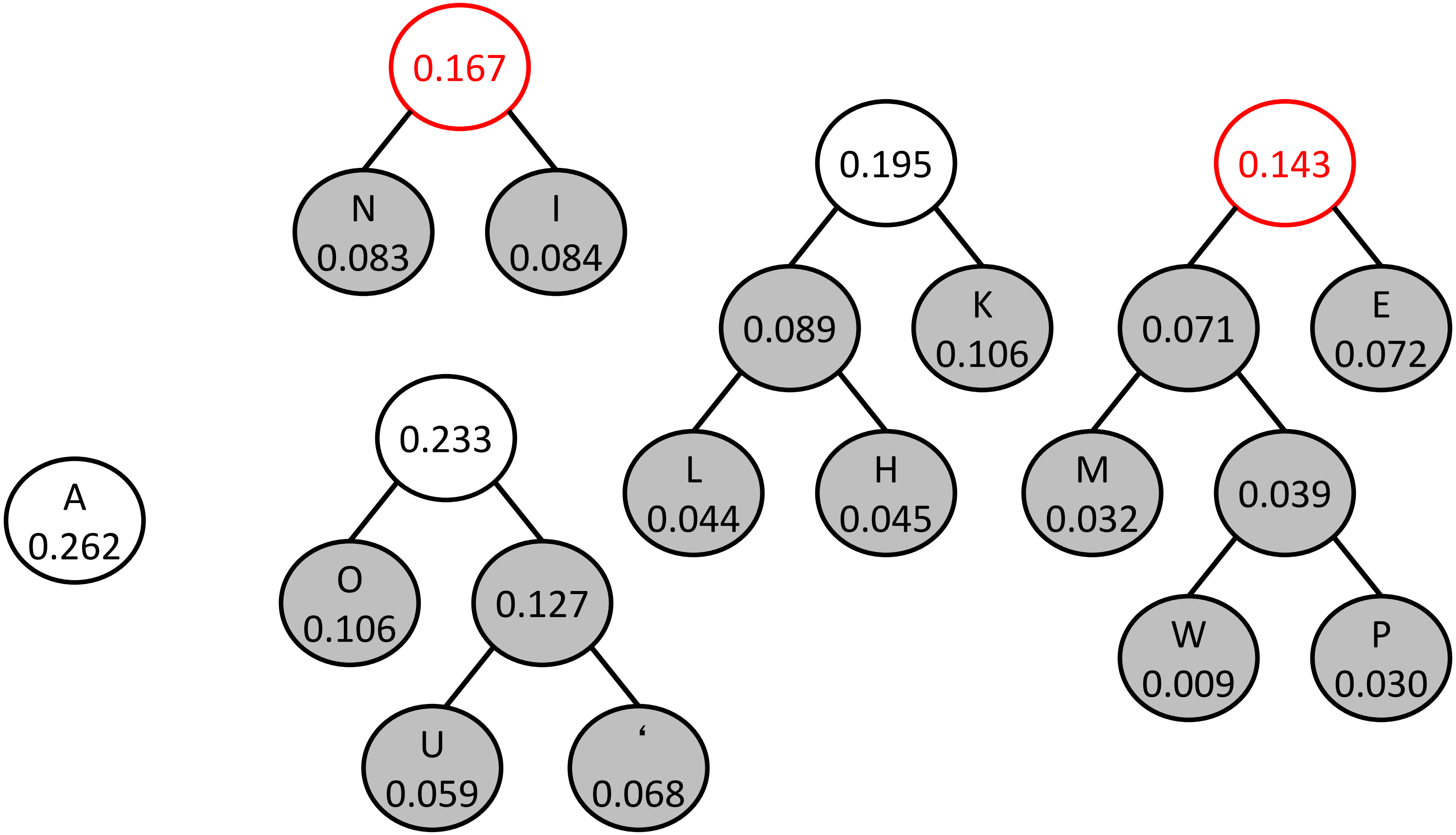
Building The Huffman Tree



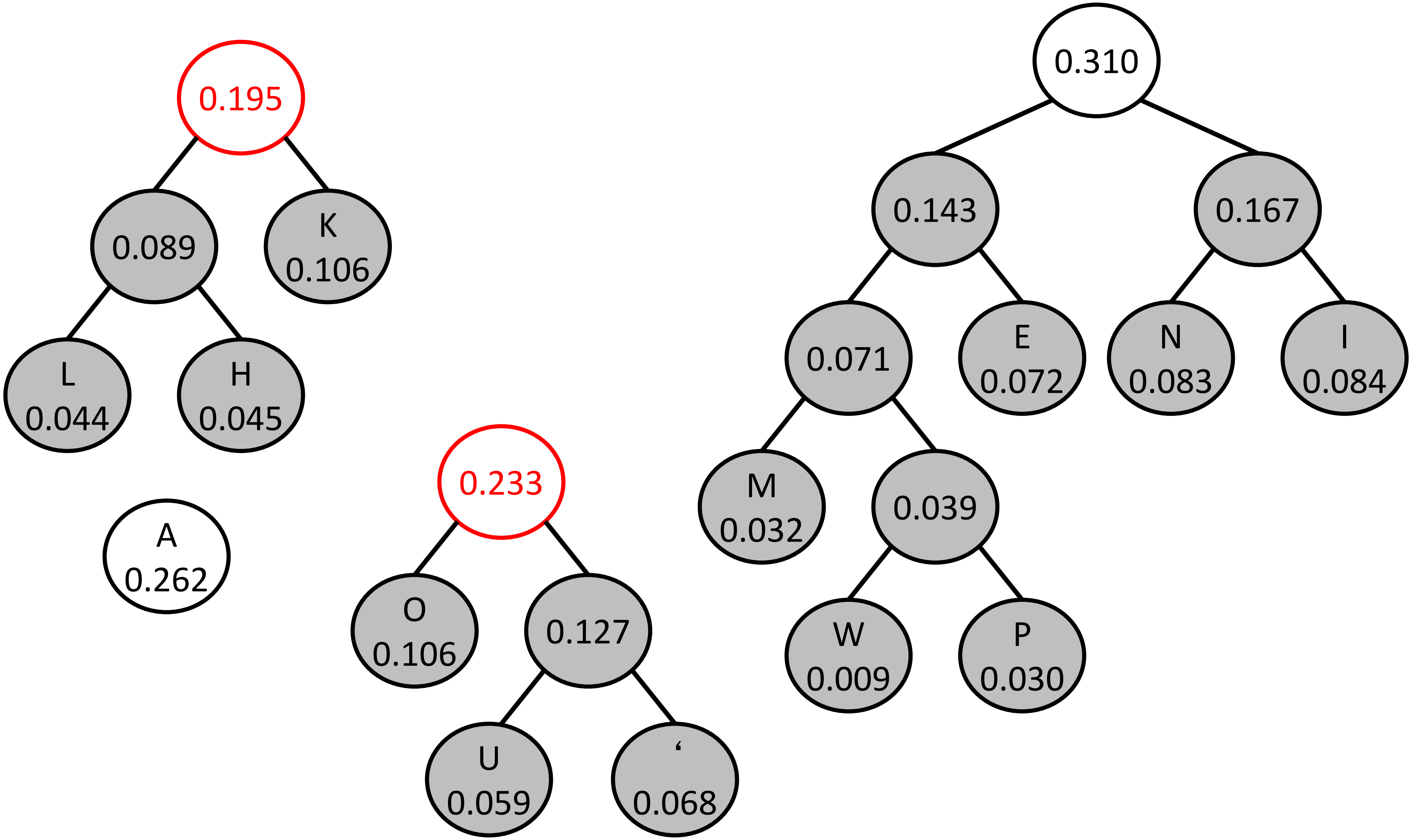
Building The Huffman Tree



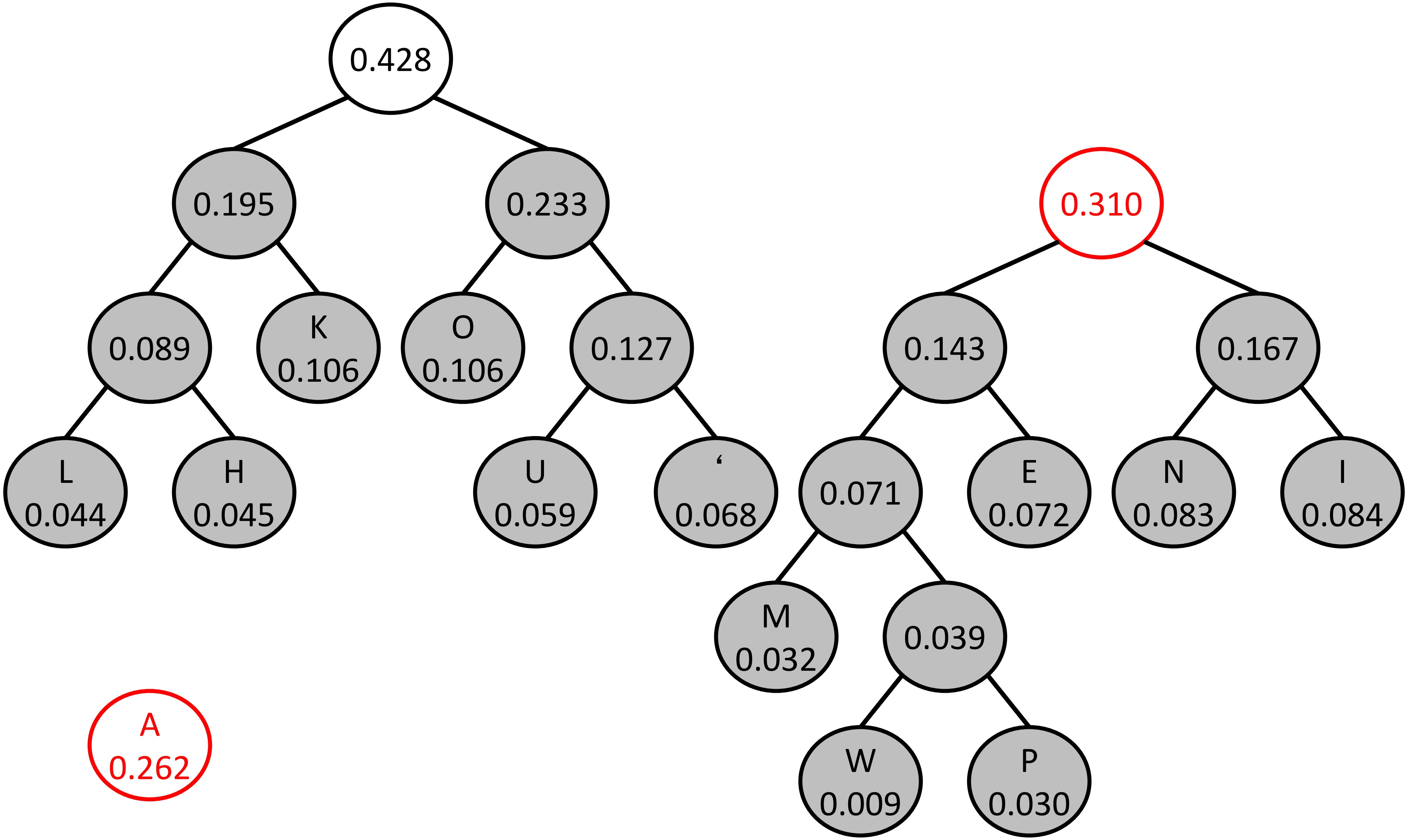
Building The Huffman Tree



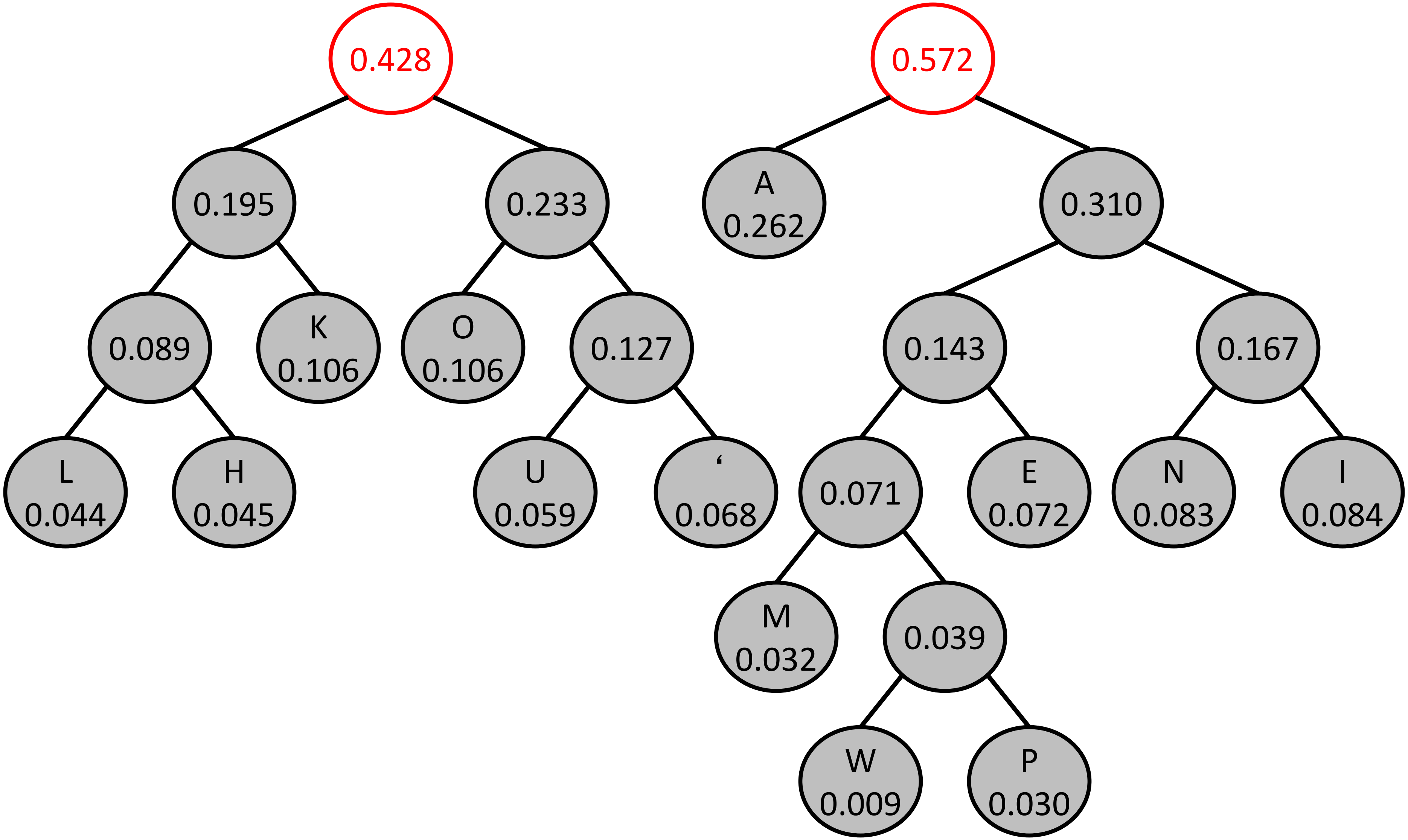
Building The Huffman Tree

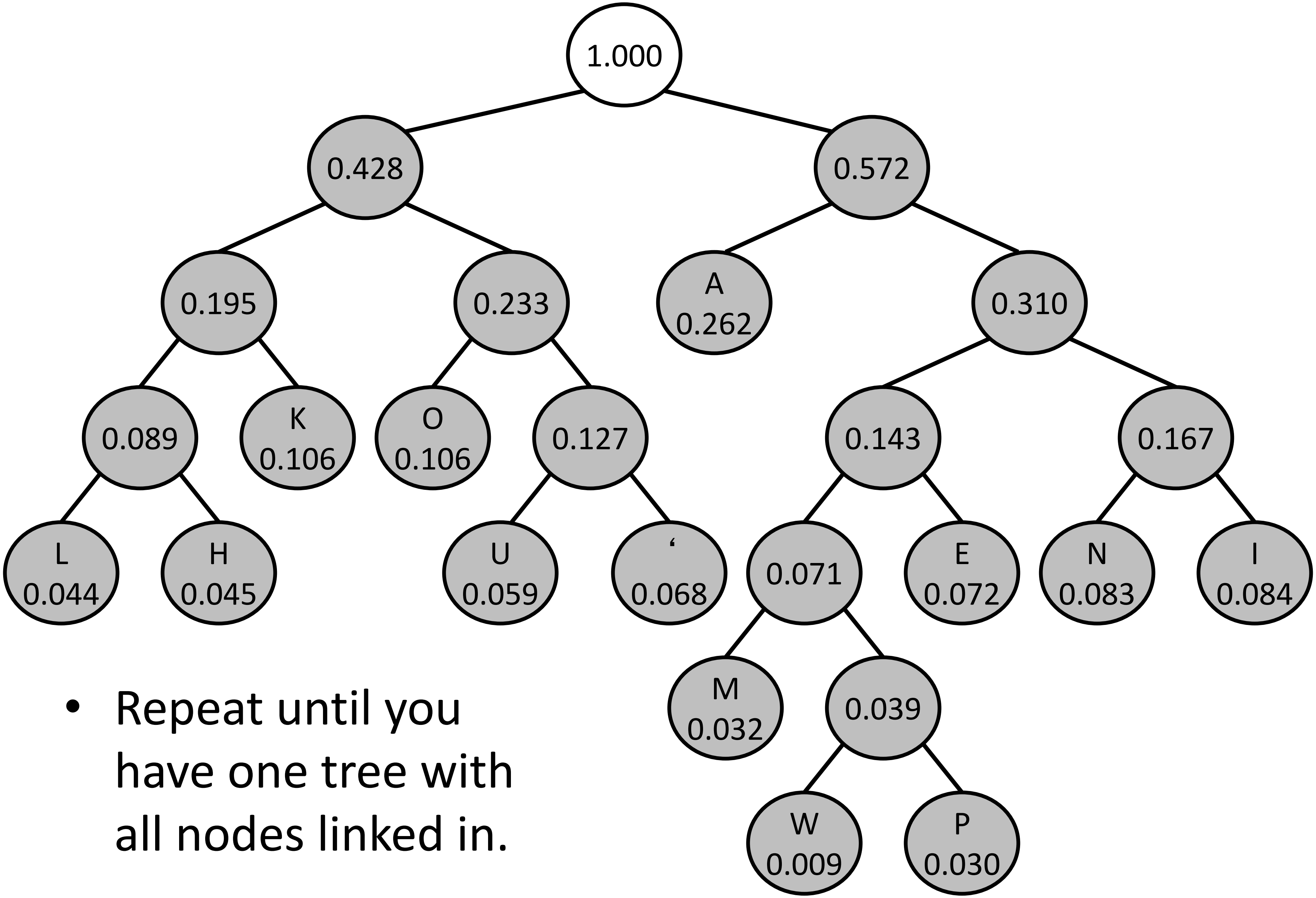


Building The Huffman Tree



Building The Huffman Tree

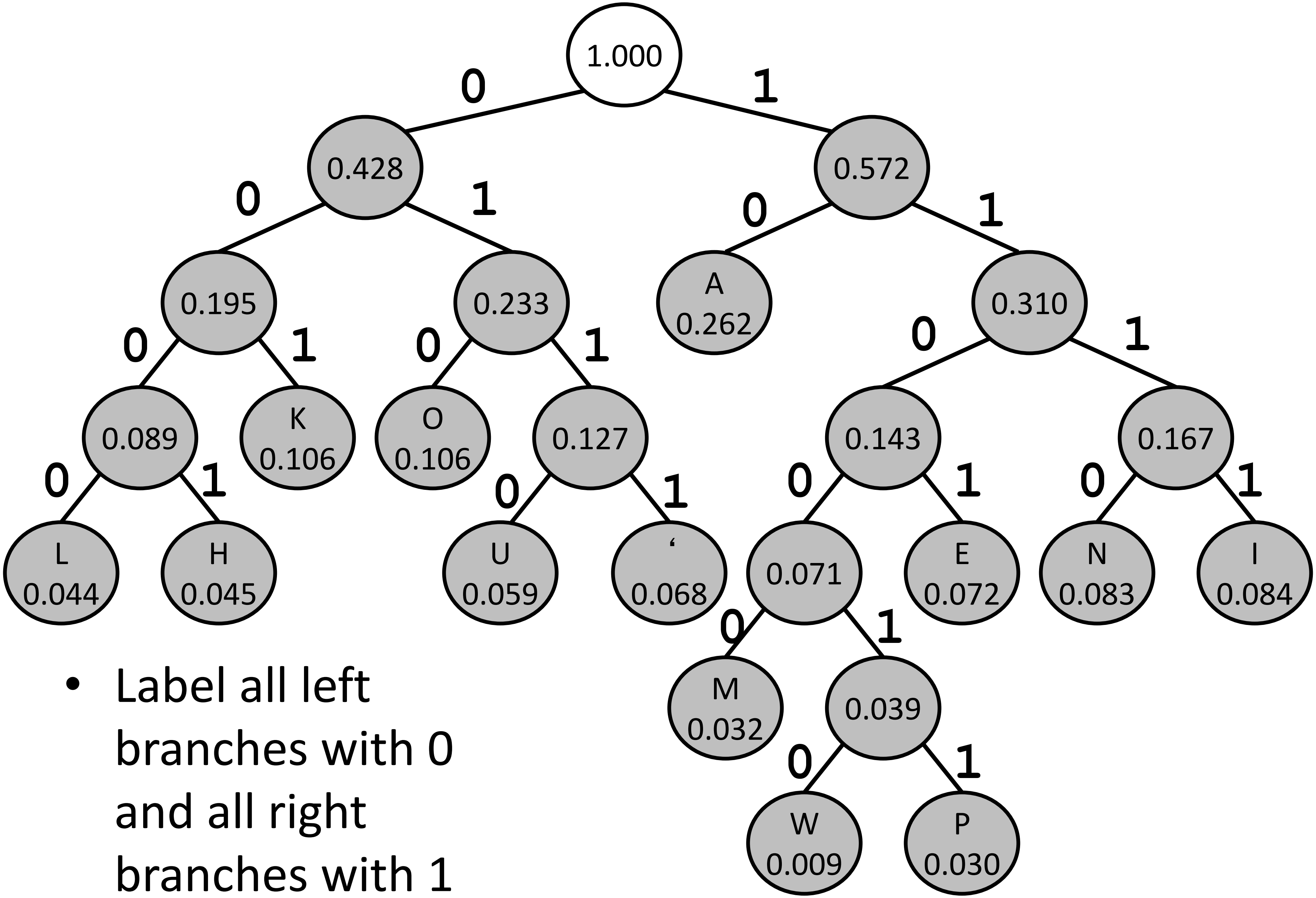




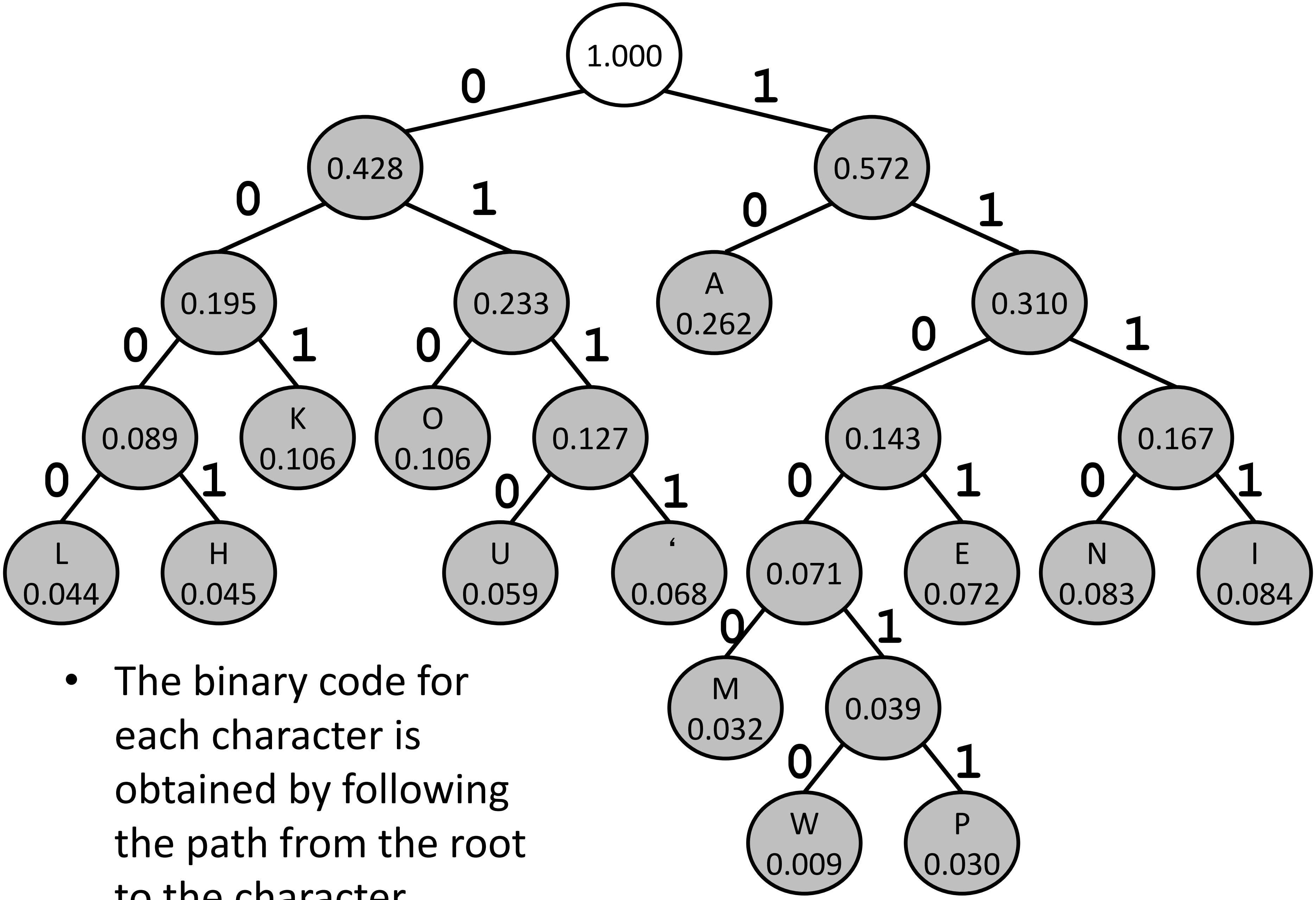
- Repeat until you have one tree with all nodes linked in.

Using the Tree to Assign Codes

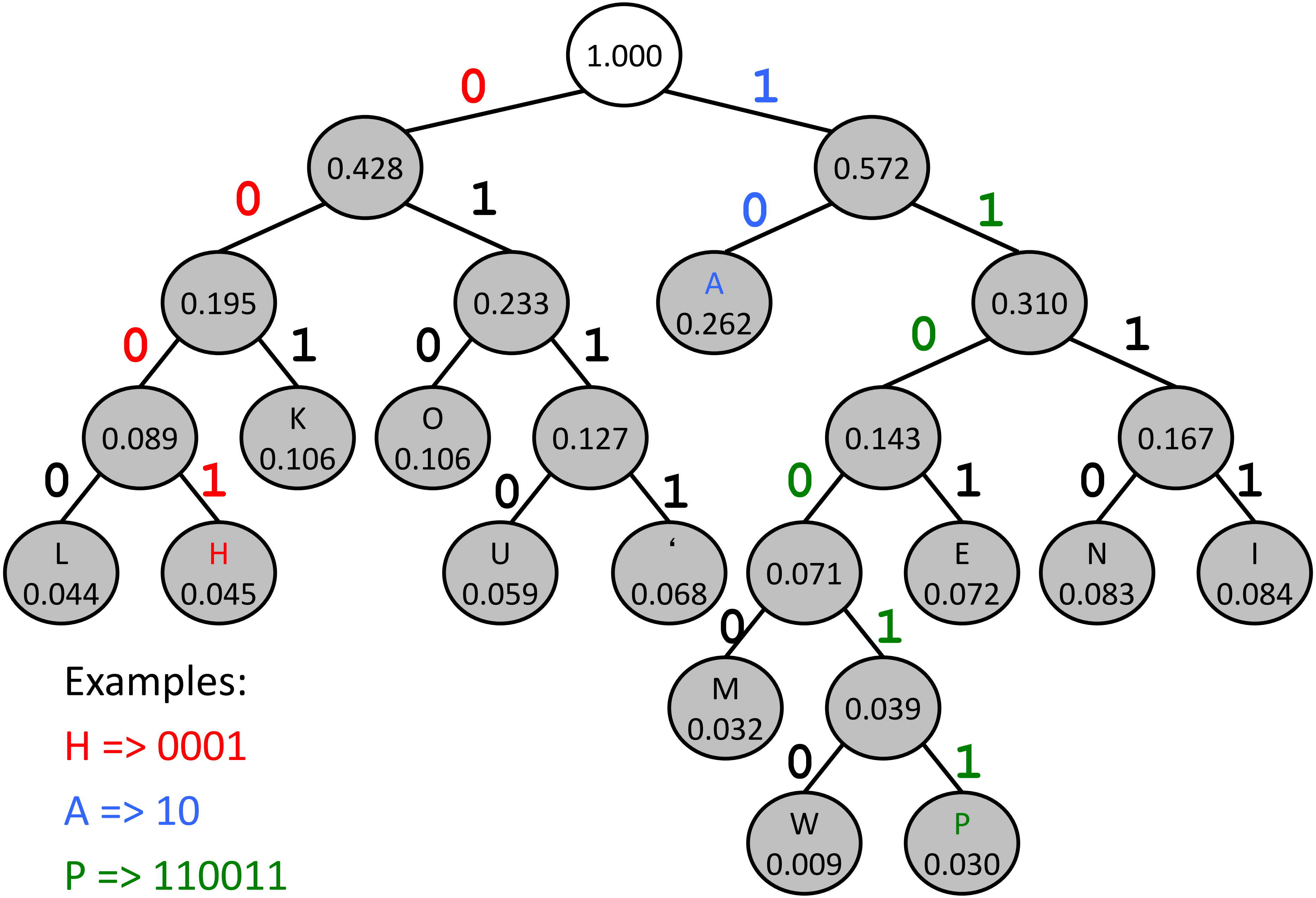
- The path from the root to each character determines the code



- Label all left branches with 0 and all right branches with 1



- The binary code for each character is obtained by following the path from the root to the character.



Examples:

H => 0001

A => 10

P => 110011

Fixed Width vs. Huffman Coding

'	0000
A	0001
E	0010
H	0011
I	0100
K	0101
L	0110
M	0111
N	1000
O	1001
P	1010
U	1011
W	1100

'	0111
A	10
E	1101
H	0001
I	1111
K	001
L	0000
M	11000
N	1110
O	010
P	110011
U	0110
W	110010

ALOHA

Fixed Width:

0001 0110 1001 0011 0001

20 bits

Huffman Code:

10 0000 010 0001 10

15 bits

How about…

- humuhumunukunukuapua'a (the reef triggerfish)
- 4454445444344434264242 = 84
- vs 22*4 = 88

How close did we get to minimum bits?

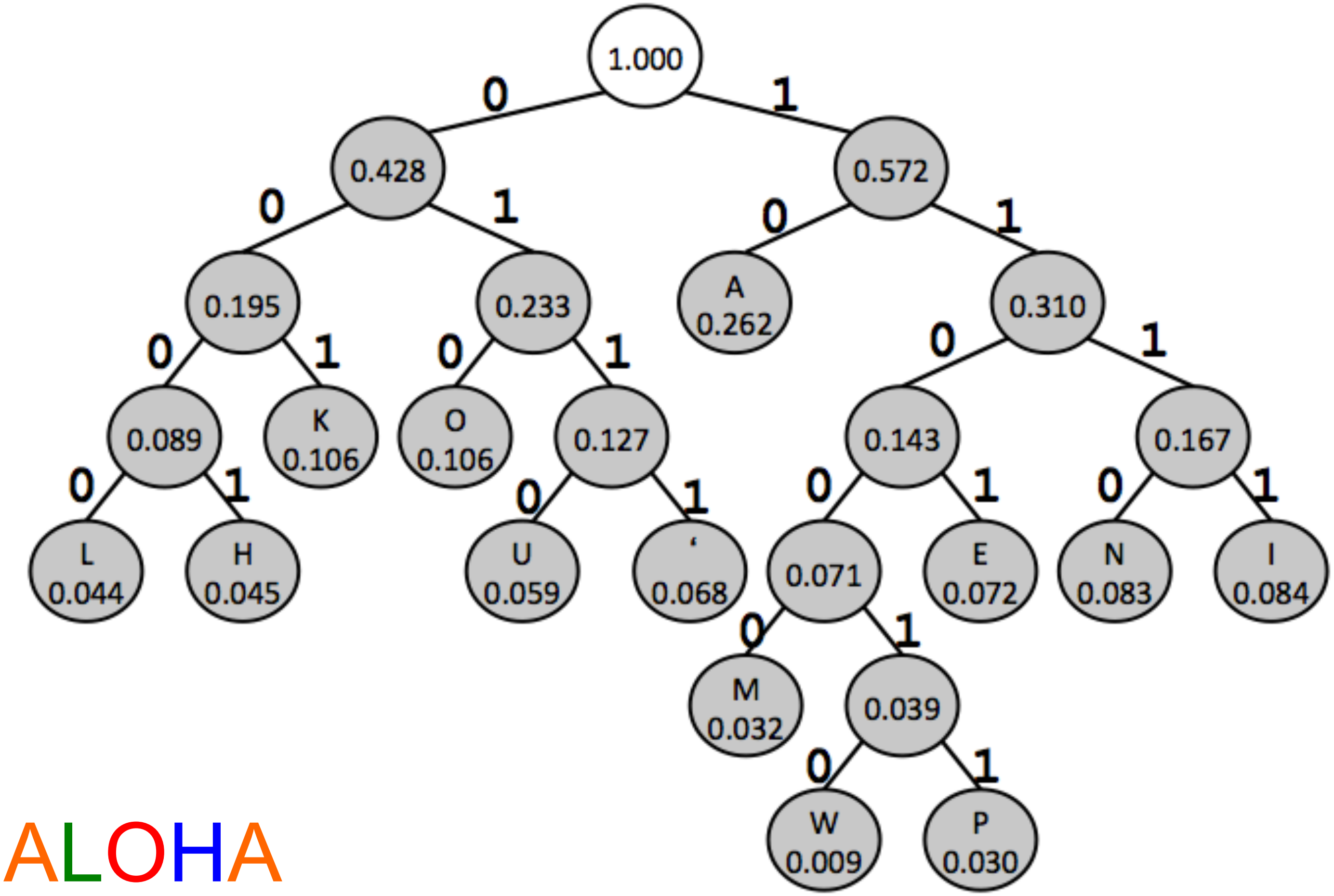
- We calculated the entropy as about 3.34 bits per character
- The average Huffman code length, weighted by the probabilities:

```
>>> ps = [.068, .262, .072, .045, .084, .106, .044, .032,
.083, .106, .030, .059, .009]
>>> code_lengths = [4, 2, 4, 4, 4, 3, 4, 5, 4, 3, 6, 4, 6]
>>> weighted_avg(ps, code_lengths)
3.374
```

pretty close!

Decoding

100000010000110



ALOHA

- To find the character use the bits to determine path from root

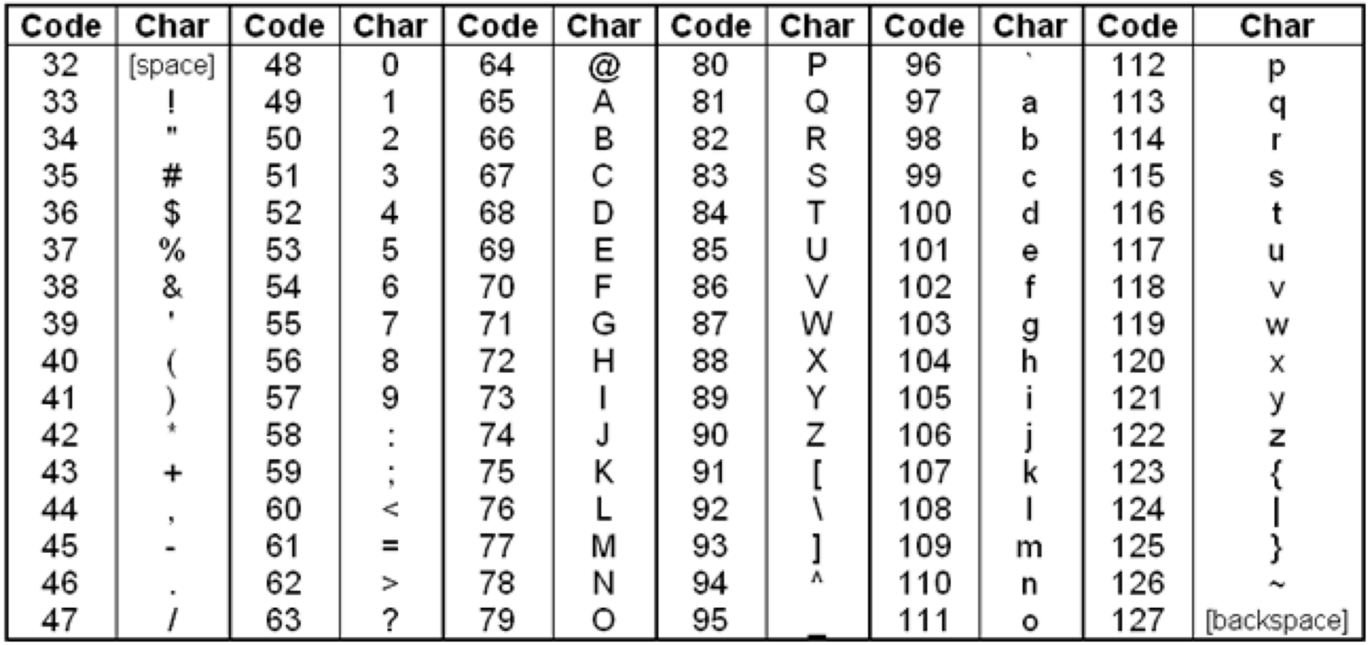
parity bits

error correction

Noisy Communication Channels

- Suppose we’re sending ASCII characters over the network
- Network communications may erroneously alter bits of a message
- Simple error detection method: **the parity bit**

Reminder: ASCII table



Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
32	[space]	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	[backspace]

- 2^7 (128) characters
- 7 bits needed for binary representation
- (Not shown: control characters like tab and newline, values 0…31)

Parity

- Idea: for each character (sequence of 7 bits), count the number of bits that are 1

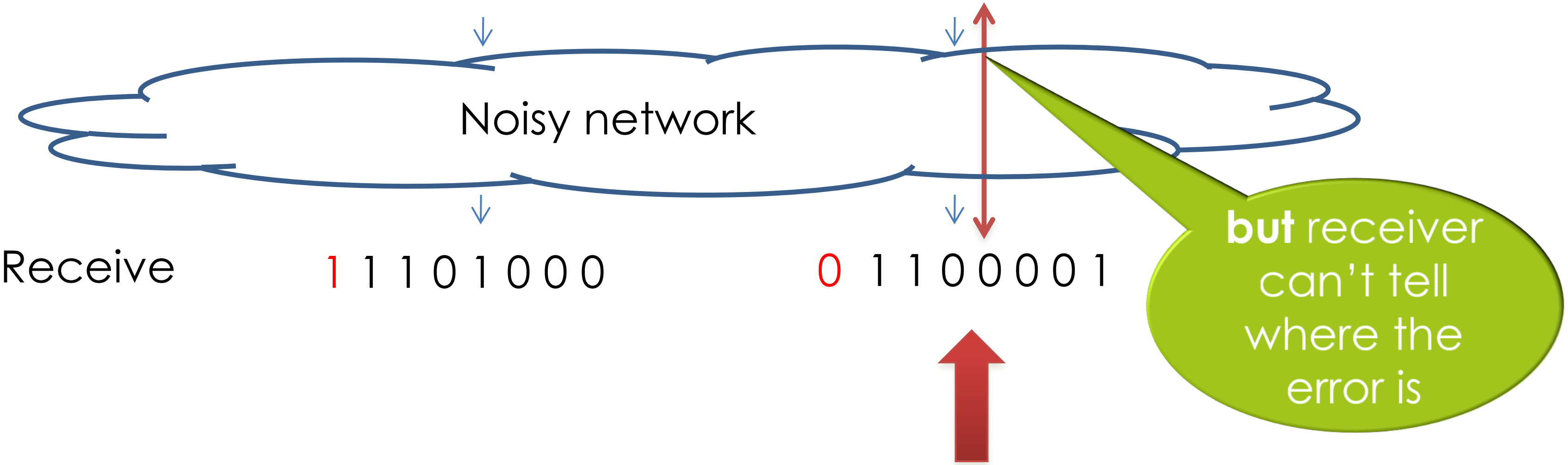
- Sender and receiver agree to use *even parity* (or *odd parity*); sender sends **extra** leftmost bit
  - Even parity: Set the leftmost bit so that the number of 1′s in the byte is even.

Parity Example

- “M” is transmitted using **even parity**.
- “M” in ASCII is 77_{10}, or 100 1101 in binary
  - four of these bits are 1
- Transmit 0 100 1101 to make the number of 1-bits **even**.
- Receiver counts the number of 1-bits in character received
  - if odd, something went wrong, request retransmission
  - if even, proceed normally
  - Two bits could have been flipped, giving the illusion of correctness. **But** the probability of 2 or more bits in error is low.

Parity Example

	H	**I**	
7 bit code	1 1 0 1 0 0 0	1 1 0 1 0 0 1	
Transmit	1 1 1 0 1 0 0 0	0 1 1 0 1 0 0 1	**Even parity**
Receive	1 1 1 0 1 0 0 0	0 1 1 0 0 0 0 1	



but receiver can't tell where the error is

Odd number of ones. There must be an error in transmission

Parity and redundancy

- An ASCII character with a correct parity bit contains *redundant information*
- …because the parity bit is *predictable* from the other bits
- This idea leads into the basics of information theory